# ESCUELA SUPERIOR DE INGENIEROS
## UNIVERSIDAD DE SEVILLA
## INGENIERÍA SUPERIOR DE TELECOMUNICACIÓN
## PROYECTO FIN DE CARRERA

# Modelado y Simulación de moduladores Sigma – Delta SC mediante leguajes de descripción de HARDWARE


Autor del Proyecto
**Andrés Felipe Talero Alvarado**

Director del Proyecto
**Francisco V. Fernández Fernández**
**Fernando Medeiro Hidalgo**

(Departamento de Electrónica y electromagnetismo, Universidad de Sevilla
Instituto de Microelectrónica de Sevilla, IMSE-CNM, CSIC)



Sevilla, Septiembre 2002

*A mis padres y hermana*
*A mi familia y amigos*
*A todo aquel que haya sufrido la carrera conmigo*

# ÍNDICE

# PREFACIO

El procesado de las señales en el dominio digital ha tenido mucho auge, incluso en el caso de que las señales sean analógicas, debido a que se aprovechan todas las ventajas de los circuitos digitales, tales como robustez y programabilidad.

Uno de los circuitos más importantes en dichos sistemas es el Convertidor Analógico – Digital (CA/D), que transforma una señal analógica a una señal digital. Dentro de los muchos tipos de CA/D que existen, nos centraremos en los CA/D $\Sigma\Delta$, que combinan sobremuestreo y conformación del ruido para reducir la potencia del ruido de cuantización, que es inherente en toda conversión analógica – digital, en la banda de la señal.

El interés por los convertidores $\Sigma\Delta$ se ha incrementado en los últimos años, debido a dos razones:

- Primero, a diferencia de otros convertidores que necesitan bloques básicos muy precisos para obtener una resolución alta, los convertidores $\Sigma\Delta$ muestran baja sensibilidad a las imperfecciones de sus bloques básicos.

- Por otro lado el número de aplicaciones con interés industrial también ha crecido.

Dado que los moduladores $\Sigma\Delta$ se encuentran en la interfaz analógica – digital, carecen de metodologías semiautomáticas para su diseño, como ocurre en los circuitos puramente digitales. Por lo tanto, se necesitan herramientas que ayuden a simular el diseño.

Este proyecto fin de carrera, pretende satisfacer la necesidad de simuladores de comportamiento para convertidores $\Sigma\Delta$ en la técnica de condensadores en conmutación (SC).

La descripción de los bloques básicos que tienen los moduladores $\Sigma\Delta$-SC se ha hecho mediante los lenguajes de descripción de hardware (HDL) VHDL y VERILOG. Para la simulación se ha utilizado la herramienta ADVANCE – MS de MENTOR. Las ventajas de utilizar los HDL son:

- Se obtienen una precisión alta al incluir las no idealidades de cada bloque básico.

- Un bajo consumo de CPU.

- Una alta flexibilidad al ser viables simulaciones de arquitecturas genéricas así como la de bloques aislados

- La posibilidad de combinar estos bloques con otros descritos en lenguajes como VHDL – AMS que utilice simulación en tiempo continuo.

Se ha desarrollado también una herramienta llamada VSIDES, que introduciendo una sintaxis muy sencilla para describir un modulador $\Sigma\Delta$, genera el código VHDL O VERILOG correspondiente.

Por último, utilizando la herramienta HDL – DESIGNER, se permite al usuario la posibilidad de describir la topología mediante captura de esquemáticos de una manera sencilla.

El contenido de este proyecto está organizado como sigue:

- En el capítulo 1 se explican los conceptos básicos de la conversión analógica – digital en general y se presentan los principios sobre los que trabajan los CA/Ds $\Sigma\Delta$, su diagrama de bloques básicos, así como los distintos tipos de arquitecturas $\Sigma\Delta$.

- En el capítulo 2 se analizan los distintos errores que se presentan en los moduladores $\Sigma\Delta$-SC

- En el capítulo 3 se describe la problemática asociada a la simulación de comportamiento en los lenguajes VHDL y VERILOG. También se exponen los modelos de comportamiento asociados a cada bloque básico, su diagrama de flujo genérico así como su forma de implementación en VHDL. Por último se ilustra la incorporación de algunas rutinas en VHDL para el procesamiento de los resultados.

- En el capítulo 4 se hace una descripción general de la herramienta ilustrando los tres tipos de forma de descripción de topologías $\Sigma\Delta$ mediante un ejemplo. Por último, se incluyen demostraciones y ejemplos de dos tipos de arquitecturas $\Sigma\Delta$-SC.

- En el apéndice A se describe a fondo la herramienta VSIDES. Esta herramienta permite generar un fichero VHDL con la descripción de la topología $\Sigma\Delta$ utilizada a partir de una sintaxis sencilla. También explica como añadir nuevos bloques básicos a VSIDES en el caso de que se necesiten.

- En el apéndice B se describe a fondo la forma de simular un modulador $\Sigma\Delta$-SC mediante captura de esquemáticos. También explica como añadir nuevos bloques básicos para utilizarlos en la captura de esquemáticos.

- En el apéndice C se muestra el código C de VSIDES

- En el apéndice D se le da al lector la posibilidad de aprender a utilizar funciones escritas en lenguaje C y utilizarlas con los simuladores ADVANCE – MS y MODELSIM.

- Por ultimo en el apéndice E se ilustra el código del bloque más importante del modulador, que es el integrador y su implementación en los lenguajes VHDL y VERILOG.

# CAPÍTULO 1 Introducción

## 1.1. Convertidores Sigma-Delta

Los convertidores de sobremuestreo son aquellos en que la frecuencia de muestreo es mucho más grande que la frecuencia de Nyquist de la señal que está siendo convertida [1]. Su éxito es debido al hecho de que pueden resolver algunos problemas encontrados en otras arquitecturas para implementaciones digitales CMOS, principalmente la alta selectividad de los filtros analógicos y la gran sensibilidad a las imperfecciones de la circuitería y los entornos de ruido.

De hecho los convertidores de sobremuestreo relajan los requerimientos de la circuitería analógica, pero obligan a que la circuitería digital sea más compleja y rápida [2]. Esto se debe a las siguientes circunstancias:

- Por un lado, el uso de una frecuencia de muestreo muy alta relaja las especificaciones para el filtro antialiasing por lo que puede ser implementado con un filtro pasivo de primer orden.

- Los filtros críticos se hacen en el plano digital donde son más robustos frente a imperfecciones de la circuitería.

- Combinando sobremuestreo y modulación Sigma-Delta, los convertidores A/D se generan con una gran resolución, robustez y relativa insensibilidad a efectos no ideales [3].

Estas ventajas explican que los convertidores de sobremuestreo, y especialmente los que incorporan modulación $\Sigma\Delta$, sean convenientes para implementar interfaces A/D de altas prestaciones en tecnologías submicrométricas de bajo voltaje [4], donde aunque no es fácil obtener un funcionamiento analógico preciso es posible implementar circuitos digitales muy rápidos.

Sin embargo, las grandes diferencias encontradas, incluso a nivel conceptual entre los convertidores tradicionales y aquellos basados en modulación $\Sigma\Delta$, en gran medida hacen difícil reusar las técnicas de diseño descubiertas. Por lo tanto es necesario el descubrimiento de nuevas tecnologías que permiten obtener convertidores $\Sigma\Delta$ optimizados. Esta tarea se basa en un conocimiento profundo de la operación del modulador y como es afectado por las no idealidades de la circuitería.

Los convertidores $\Sigma\Delta$, descubiertos por Inose en 1962 [3], operan con una cantidad de datos temporales redundantes usando técnicas de sobremuestreo con cuantizadores de baja resolución, algunas veces de un único bit y aplicando técnicas de procesamiento de señal, promediando en el caso más simple, para combinar estos datos temporales, por lo que se incrementa la resolución efectiva.

Este paralelismo temporal es la base de la robustez de los convertidores $\Sigma\Delta$, que empezando desde sus más tempranas implementaciones en la banda de audio [2], han ampliado su rango de aplicación desde la instrumentación [2][5] hasta las comunicaciones [6][7][8][9]. Mientras que los convertidores tradicionales necesitan alta precisión en sus componentes, los convertidores $\Sigma\Delta$ muestran muy baja sensibilidad a las imperfecciones de la circuitería. Por lo tanto estas arquitecturas son adecuadas para la conversión A/D de alta resolución en tecnologías VLSI.

Estas ventajas han promovido el interés de los investigadores en los convertidores de sobremuestreo. Como consecuencia de esta actividad, hay muchas arquitecturas disponibles para implementar convertidores $\Sigma\Delta$, desde single-loop de bajo orden [2][3][10], hasta los más sofisticados con múltiples lazos de realimentación [6][8] [11][12][13][14][15][16]. Muchas de estas arquitecturas han sido exitosamente implementadas como:

- 20 bits de resolución efectiva en instrumentación [5][17][18 ]
- 16 bits en adquisición de datos y audio [19][20][21][22]
- 12 bits en comunicaciones [6][23]

Sin embargo dos factores han restringido el uso de estos convertidores en ASICs industriales:

- Por un lado la ya mencionada proliferación de arquitecturas de modulador, con diversos criterios respecto a la elección de sus parámetros de diseño, es muy difícil para diseñadores inexpertos.

- Por otro lado la poca presencia de herramientas específicas para la síntesis y verificación producen un gran consumo de recursos en la optimización del diseño, especialmente cuando el error dominante no es el error de cuantización sino los errores relacionados con las imperfecciones de la circuitería [24][25].

Hasta ahora las herramientas que se tenían para el diseño y simulación de moduladores Sigma-Delta, eran de dos tipos:

- El primer tipo son herramientas escritas en código C. Estas herramientas normalmente son muy rápidas. El inconveniente es que normalmente son específicas para un tipo concreto de circuito. Además no son herramientas estándar que utilicen la mayoría de los diseñadores, por lo que tienen que invertir tiempo en el aprendizaje de su uso.

- El otro tipo son herramientas descritas con Simulink. Estas herramientas son lentas en comparación con las herramientas descritas en código C, pero proveen al usuario una interfaz gráfica sencilla basada en la interconexión de elementos sencillos, muy fácil de manejar por el usuario.

La solución propuesta en este proyecto es utilizar bs lenguajes de descripción de Hardware (HDL) y más específicamente VHDL Y VERILOG, para simular el comportamiento de los moduladores Sigma-Delta. Las ventajas de esta solución con respecto a las dos anteriores son:

- VHDL Y VERILOG son dos estándares en la descripción de circuitos digitales. Esto hace posible que a los diseñadores de Sigma-Delta familiarizados con programación en estos lenguajes se les pueda dar el modelo de comportamiento de todos los bloques que componen los moduladores.

- La posibilidad de simular un circuito de señal mixta compuesto por los bloques correspondientes a moduladores Sigma-Delta y aquellos correspondientes a la parte digital, sin necesidad de cambiar de simulador.

- La posibilidad de combinar estos bloques con otros escritos con VHDL-AMS y VERILOG-A que necesiten simularse en tiempo continuo.

# 1.2. Partes de un convertidor Sigma-Delta

La Figura 1.1 muestra un diagrama de bloques de un convertidor de sobremuestro $\Sigma\Delta$ que incluye los siguientes elementos:

- *Filtro Antialiasing*: Elimina de la señal de entrada las componentes espectrales por encima de la mitad de la frecuencia de muestreo. El sobremuestreo relaja los requerimientos de este filtro por lo que un simple filtro pasivo de primer orden es suficiente.

- *Modulador*: En este bloque la señal es muestreada y cuantizada. Adicionalmente también es posible filtrar el error inherente en la cuantización, conformando su densidad espectral de potencia de forma que la mayoría de su potencia permanezca fuera de la banda de la señal, donde el error es eliminado por el filtro digital. La salida del modulador es codificada en un número reducido de bits, normalmente uno, a la frecuencia de muestreo.

- *Decimador*: Es este un bloque puramente digital. Después de filtrar todos los componentes fuera de la banda de la señal, que incluye una gran parte de la potencia del error de cuantización, los datos son decimados para reducir la frecuencia de muestreo a la frecuencia de Nyquist. El resultado es la señal codificada en un gran número de bits a la frecuencia de Nyquist.

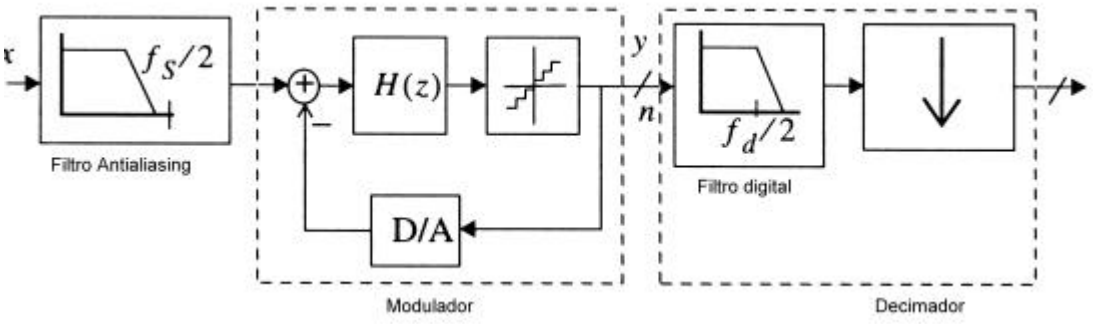



**Figura 1.1. Diagrama de bloques del convertidor Sigma-Delta**

La Figura 1.2 ilustra el procesamiento de la señal, por los bloques del convertidor. El muestreo y la cuantización han sido separados en el modulador.

Entre los bloques del convertidor, el modulador es el más difícil de diseñar debido a lo siguiente:

- Por un lado el sobremuestreo simplifica el filtro antialiasing a un simple filtro paso baja RC.

- Por otro lado el decimador [26] es un bloque puramente digital, cuyo diseño está altamente automatizado y puede ser hecho con herramientas de CAD ya disponibles.

- El modulador situado entre el plano analógico y el digital, encierra muchos mecanismos de error que degradan el funcionamiento del convertidor como son los siguientes
    - Errores de cuantización inherentes en la operación.
    - Gran cantidad de no idealidades que afectan a los bloques del modulador

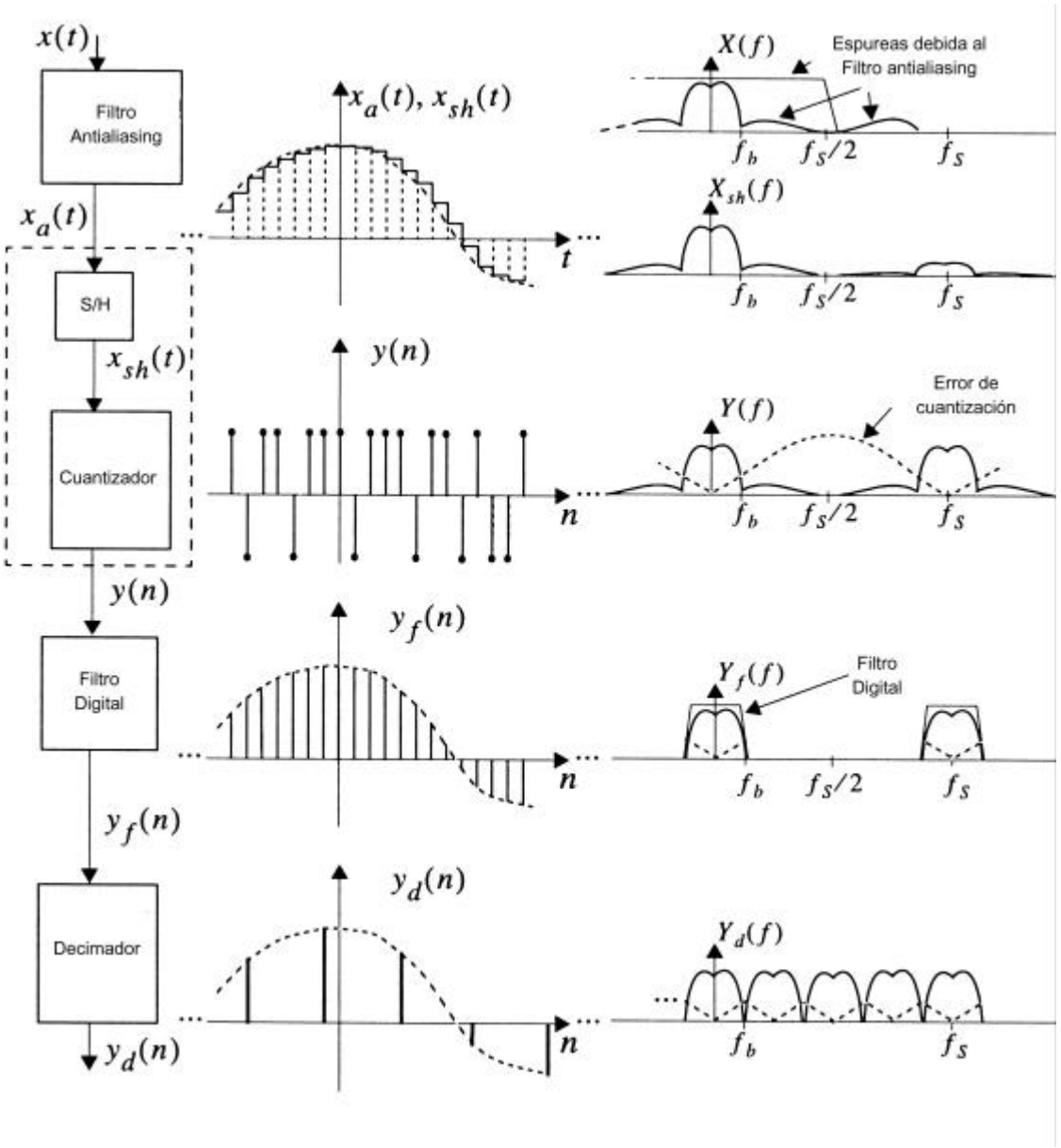



**Figura 1.2. Procesamiento de la señal en un convertidor Sigma-Delta**

# 1.3. Conceptos básicos

Antes de continuar es conveniente definir algunos conceptos para caracterizar a los convertidores Sigma - Delta.

## 1.3.1. Ruido de cuantización

La siguiente figura muestra la curva característica de un cuantizador ideal, cuyo error, como se puede apreciar es una función no lineal de la entrada.

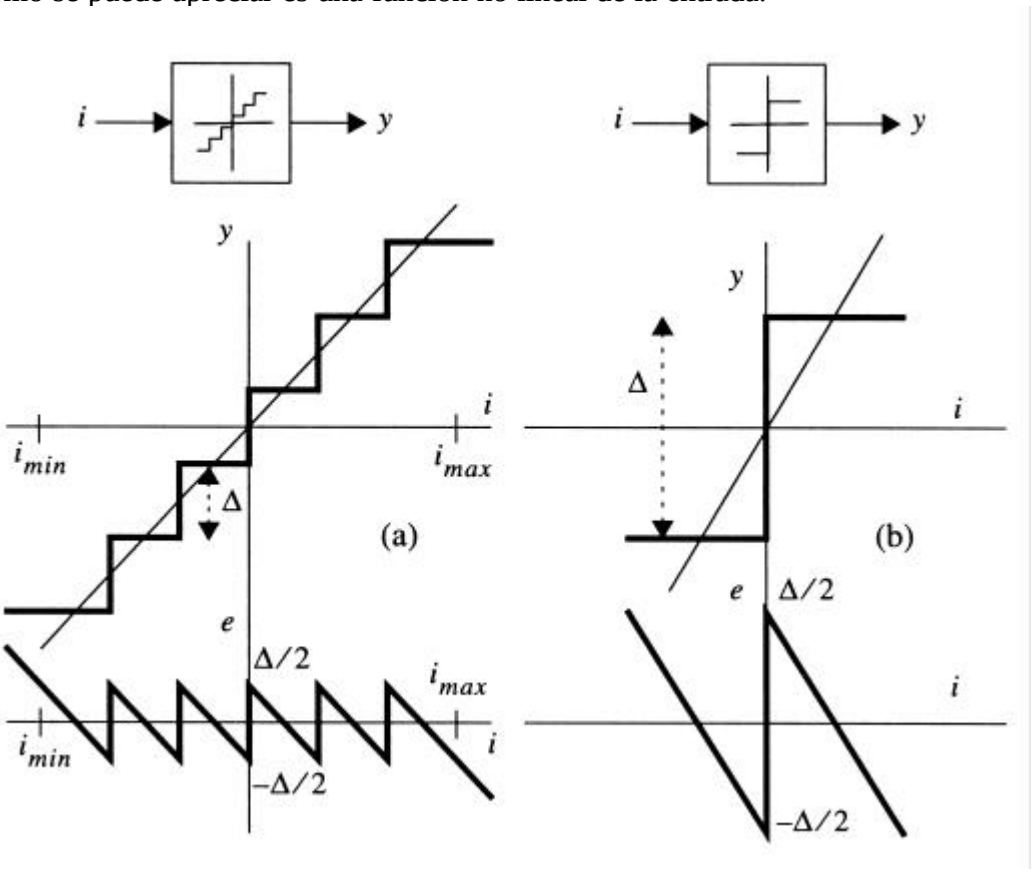



**Figura 1.3. Curva de transferencia y error de un cuantizador multibit y un comparador**

Esto puede ser representado como sigue:

$$y = g_q i + e \tag{1.1}$$

donde:

- $g_q$ denota la ganancia del cuantizador.
- e denota el error de cuantización.

Si la entrada del cuantizador esta confinada al intervalo [$i_{min}$, $i_{max}$] el error de cuantización está confinado en el intervalo [$-\Delta/2$, $-\Delta/2$] siendo $\Delta$ la separación entre niveles consecutivos del cuantizador.

Para entradas fuera de ese intervalo el valor absoluto del error de cuantización crece monótonamente. Esta situación es conocida como sobrecarga del cuantizador.

### 1.3.2. Potencia de ruido de cuantización

Si la entrada del cuantizador varía aleatoriamente de muestra en muestra en el intervalo [$i_{min}$, $i_{max}$] y el número de niveles del cuantizador es grande, se puede mostrar entonces [27]que el error de cuantización se distribuye uniformemente en el rango [-$\Delta/2$, -$\Delta/2$], que tiene una densidad espectral de potencia constante, como un ruido blanco. La potencia de ruido de cuantización, $s^2(e)$ se distribuye uniformemente en el rango [-$f_S/2$, $f_S/2$], su densidad espectral de potencia es igual a,

$$S_E(f) = \frac{s^2(e)}{f_S} = \frac{\Delta^2}{12 f_S} \tag{1.2}$$

### 1.3.3. Razón de sobremuestreo

Si la frecuencia de muestreo es igual a la frecuencia de Nyquist, la potencia del error de cuantización en la banda de la señal es $\Delta^2/12$, que es toda la potencia de ruido. Pero por otro lado, si la frecuencia de muestreo es más grande que la de Nyquist, una porción más baja de la potencia del ruido de cuantización permanece en la banda de la señal. Esto puede ser calculado como sigue:

$$P_Q = \int_{-f_d/2}^{f_d/2} S_E(f)df = \frac{\Delta^2 f_d}{12 f_S} = \frac{\Delta^2}{12M} \tag{1.3}$$

siendo $f_d$ la frecuencia de Nyquist

La potencia en banda del ruido de cuantización es inversamente proporcional a la razón entre la frecuencia de muestreo y la frecuencia de Nyquist. Esta fracción se denota usualmente por M, y se llama razón de sobremuestreo. Acorde a la ecuación anterior, un incremento en M implica una reducción de 3 dB/ octava en la potencia de ruido de cuantización.

### 1.3.4. Espectro de salida

Es la distribución en potencia de la señal de salida de un modulador en todas las posibles frecuencias. En la Figura 1.4 podemos ver un ejemplo de espectro de salida de un modulador $\Sigma\Delta$.

### 1.3.5. SNR

Esta es la fracción entre la potencia de salida a la frecuencia de la señal senoidal de la entrada y la potencia ($P_Q$) del ruido en banda (IBN). Se suele dar en dB. La potencia de una señal senoidal es $A^2/2$, siendo A la amplitud de la señal. Nos quedaría en el caso ideal:

$$SNR(dB) = 10\log_{10}\left(\frac{A^2/2}{P_Q}\right) \tag{1.4}$$

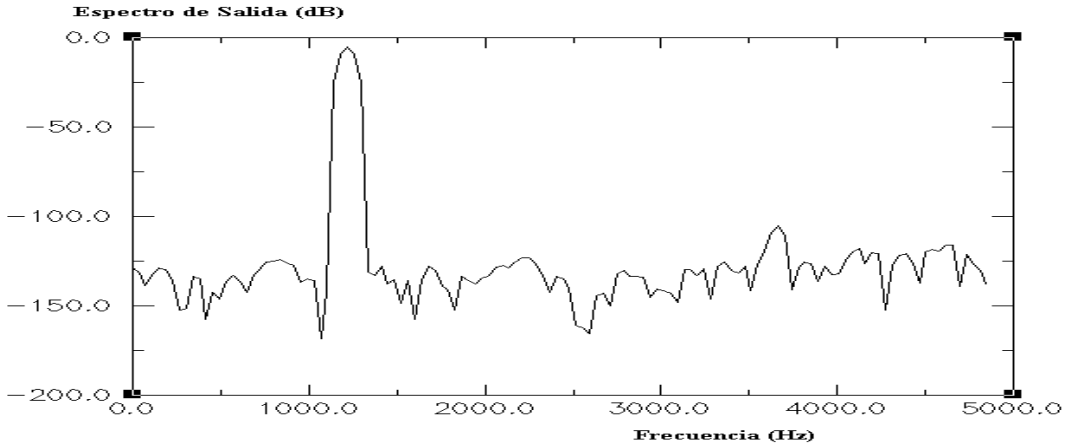



**Figura 1.4. Espectro de salida de un modulador sigma-delta**

## 1.3.6. Rango dinámico(DR)

Se define el rango dinámico como la razón entre la potencia de salida a la frecuencia de una señal senoidal de entrada y la potencia de salida cuando la entrada es una sinusoide de la misma frecuencia, pero de una amplitud pequeña, por lo que no puede ser distinguido del ruido o lo que es lo mismo con SNR = 0dB.

Idealmente, el rango de escala completo de la entrada del modulador está aproximadamente dado por la del cuantizador. En una cuantización de un único bit, este rango es igual a $\pm\Delta/2$. Por otro lado, la potencia de salida atendiendo a (1.4) para una entrada con una SNR de 0dB es

$$DR(dB) = 10\log_{10}\left[\frac{(\Delta/2)^2}{2P_Q}\right] \tag{1.5}$$

## 1.3.7. Resolución efectiva(B)

El rango dinámico de un convertidor ideal de B bits es dado por [10]:

$$DR = 3 \cdot 2^{(2B-1)} \tag{1.6}$$

Manipulando la expresión y poniendo el número de bits o resolución efectiva del modulador en función de su DR(dB),

$$B(bit) = \frac{DR(dB) - 1.76}{6.02} \tag{1.7}$$

# 1.4. Arquitecturas de moduladores Sigma -Delta

La Figura 1.5. muestra el esquema de un modulador $\Sigma\Delta$. Su salida, *y*, es sustraída de su señal de entrada, *x*, que ha sido muestreada a una frecuencia mucho más alta que la frecuencia de Nyquist.

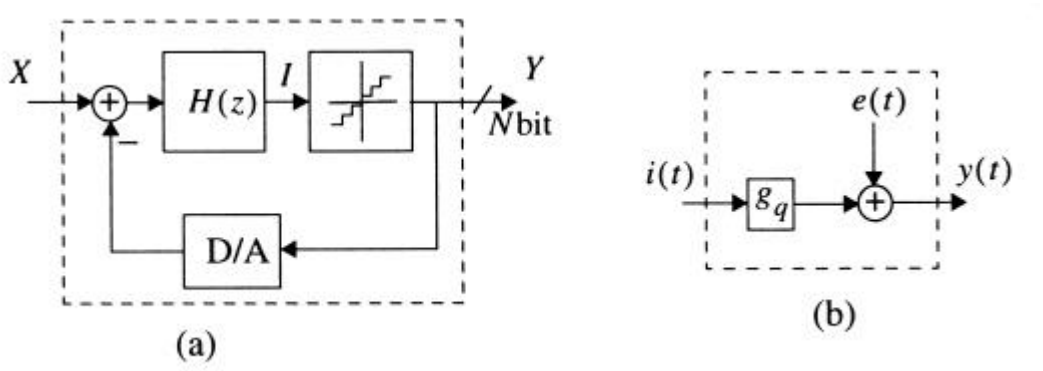



**Figura 1.5. Estructura básica de un modulador Sigma-Delta**

El resultado, después de pasar a través de un filtro de tiempo discreto, *H(z)*, sirve como una entrada al mismo cuantizador, que normalmente tiene un número reducido de niveles.

Si la ganancia del filtro es alta en el intervalo de frecuencia de interés y baja fuera de él, el error de cuantización, es atenuado debido al lazo de realimentación. Para calcular el error de cuantización en los moduladores $\Sigma\Delta$, se deben tener en cuanta las siguientes consideraciones:

- En los moduladores $\Sigma\Delta$, el error de cuantización es completamente determinado por la señal de entrada.
- El número de niveles del cuantizador normalmente es pequeño. De hecho, muchas arquitecturas populares incorporan un comparador simple para el funcionamiento de la cuantización interna.
- Para entradas estáticas, la entrada del cuantizador varía de muestra a muestra, en múltiplos o submúltiplos de la separación consecutiva entre niveles $\Delta$.

Existen un gran número de arquitecturas de moduladores $\Sigma\Delta$. En todos los casos se ha perseguido una reducción de la potencia del ruido de cuantización mediante dos estrategias [28]:

- Incrementar el orden del filtro *H(z)*, resultando una más eficiente cancelación del ruido de cuantización. En estas arquitecturas la resolución extra cuando la razón de sobremuestreo se incrementa es aproximadamente 1.5 bit / octava.

- Aumentar la resolución del cuantizador interno, con lo que $\Delta$ y la densidad espectral de potencia se reduce. De hecho, idealmente en promedio, cada bit extra en la cuantización interna resulta en un bit extra en la resolución efectiva del modulador.

## 1.4.2. Modulador de primer y segundo orden

En la siguiente figura podemos ver un modulador de primer orden. La única diferencia entre este modulador y el de Figura 1.5 es la inclusión de dos factores de ganancia $g_1$ y $g_1$' para la entrada y señal realimentada respectivamente, comúnmente llamados pesos o ganancias del integrador. Normalmente $g_1 = g_1^{'}$.

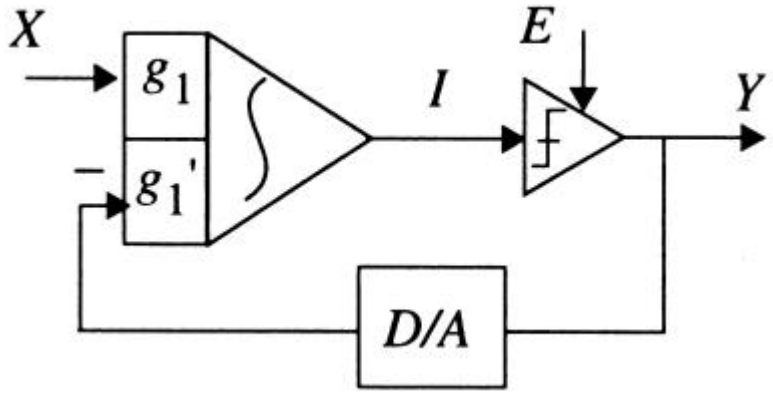



**Figura 1.6. Modulador Sigma-Delta de primer orden**

Incluyendo dos integradores en el lazo, tenemos el siguiente modulador [29].

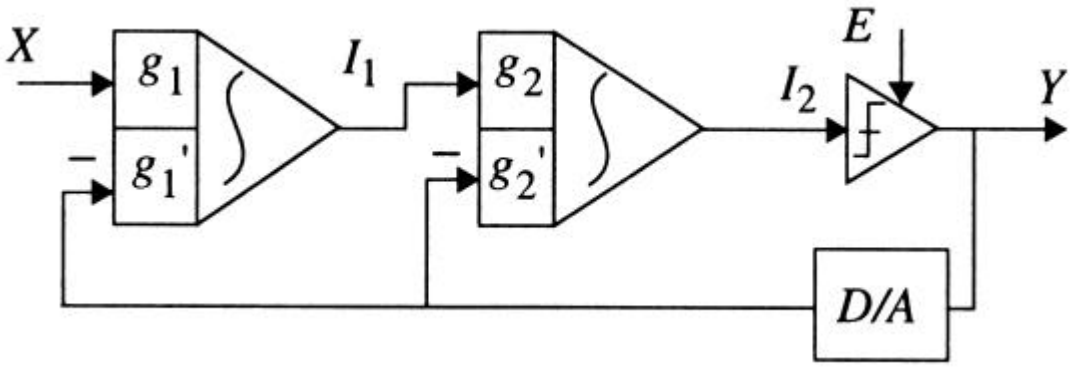



**Figura 1.7. Modulador Sigma-Delta de segundo orden**

Para garantizar la estabilidad del lazo es necesario que $g_2^{'} = 2g_1g_2$ [29].

La Figura 1.8 muestra las ventajas de un modulador de segundo orden con respecto a uno de primer orden, comparando ambas funciones de transferencia para el ruido de cuantización en el dominio de la frecuencia. En el modulador de segundo orden la densidad espectral disminuye en la región de las bajas frecuencias, y se incrementa en la región de las altas frecuencias. Se produce un decremento del ruido de cuantización.

Un incremento de la razón de sobremuestreo causa una disminución de 15 dB /octava en el ruido en banda o un incremento de 2.5 bits/octava en la resolución efectiva, en lugar de 1.5 bits/octava como el caso de un modulador de primer orden.

Para M =128 el rango dinámico de un modulador de segundo orden es 34 dB más alto que uno de primer orden. Esta diferencia se incrementa a 40 dB para M =256, que corresponde a 5.7 y 6.7 bits de resolución extra, respectivamente.

Además de la ventaja que supone el rango dinámico alcanzable, el modulador de segundo orden muestra un patrón de ruido más pequeño en presencia de señales estáticas [25]. Esta reducción se obtiene gracias al uso de la doble integración, que parcialmente decorrelaciona la señal de entrada y el error de cuantización.

De hecho, la presencia de tonos inútiles en la banda de la señal decrece cuando el orden del modulador se incrementa

.

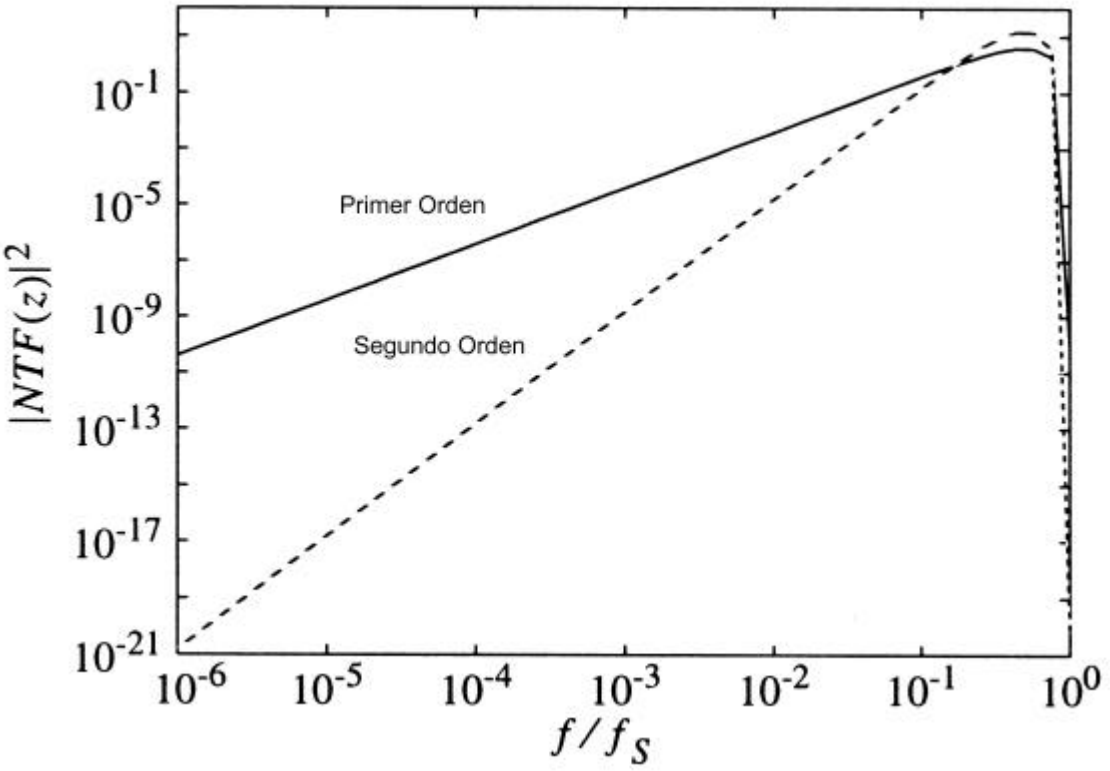



**Figura 1.8. Funciones de transferencia de ruido como función de la frecuencia**

## 1.4.3. Moduladores de un único lazo (single-loop) de alto orden

En la Figura 1.9 podemos ver un modulador de alto orden genérico.

Los moduladores $\Sigma\Delta$ single-loop con orden mayor que dos tienen tendencia a la inestabilidad. Un modulador se considera estable si para entradas acotadas y cualquier condición inicial del integrador, las variables de estado que son las salidas de los integradores, permanecen también acotadas a lo largo del tiempo.

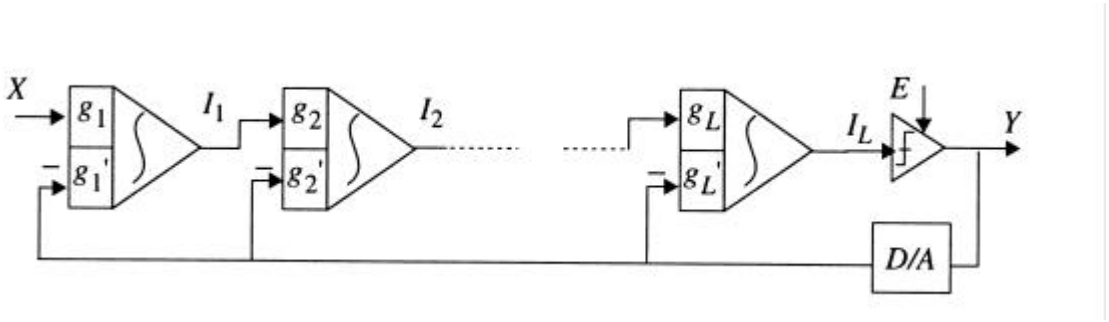



**Figura 1.9. Modulador Sigma-Delta de orden alto**

Un modulador de primer orden es intrínsicamente estable para cualquier entrada en el rango (- $\Delta/2$, $\Delta/2$).

De la misma manera la estabilidad de un modulador de segundo orden es garantizada en el rango (-0.9· $\Delta/2$, 0.9· $\Delta/2$) siempre que $g_2'/(g_1 g_2) > 1.25$.

Sin embargo, cuando el orden es más grande que dos, no es posible matemáticamente determinar una condición de estabilidad. Usando simulación de comportamiento como se muestra en [8], se comprueba que la selección apropiada de los componentes de escala de un modulador con un orden mayor que dos es condicionalmente estable. Por lo tanto, la operación es estable si la entrada está confinada a un intervalo dado que depende de la arquitectura del modulador y de las condiciones iniciales. Para entradas o condiciones fuera de sus rangos respectivos estos moduladores son inestables.

Adicionalmente, opuestamente a los moduladores de bajo orden, las oscilaciones debido a una entrada transitoria excesiva, como la producida por el encendido del circuito, no desaparece aunque la señal de entrada esté acotada después del transitorio.

En la práctica se pueden obtener algunos moduladores de alto orden usando algunas técnicas [30]:

- Selección apropiada de los coeficientes de escala o pesos de integración, para reducir la ganancia fuera de banda para el ruido de cuantización para un nivel de ruido de cuantización que mantenga la estabilidad del lazo.

- Tomando ventaja de la naturaleza acotada de las señales de las salidas del integrador debido a requerimientos de la circuitería o incluyendo límitadores después de los últimos integradores de la cadena.

- Reseteo global de los integradores cuando se detecte una operación inestable. La detección de la inestabilidad se puede hacer en el integrador, utilizando comparadores para determinar si una variable interna de estado ha sobrepasado un cierto límite o monitorizando la longitud de las series de pulsos consecutivos a la salida del modulador.

### 1.4.4. Moduladores Sigma-Delta en cascada

Una alternativa a los moduladores single-loop son las arquitecturas en cascada [31][32][33][34] cuyo bloque genérico puede verse en la Figura 1.10.

Su función se basa en la conexión en cascada de moduladores de bajo orden (0, 1, 2) cuya estabilidad está garantizada. El ruido de cuantización en una etapa es entonces remodulado por la siguiente y después cancelado en el dominio digital. Como resultado, idealmente, se obtiene la entrada del modulador más el ruido cuantización de la última etapa, este último atenuado por una función de orden igual al número de integradores en cascada.

Una implementación de un modulador de cuarto orden es el de la Figura 1.11 [35] cuya relación entre los coeficientes se puede ver en la Tabla 1.1.

Por lo tanto con esta técnica es posible obtener un modulador $\Sigma\Delta$ estable

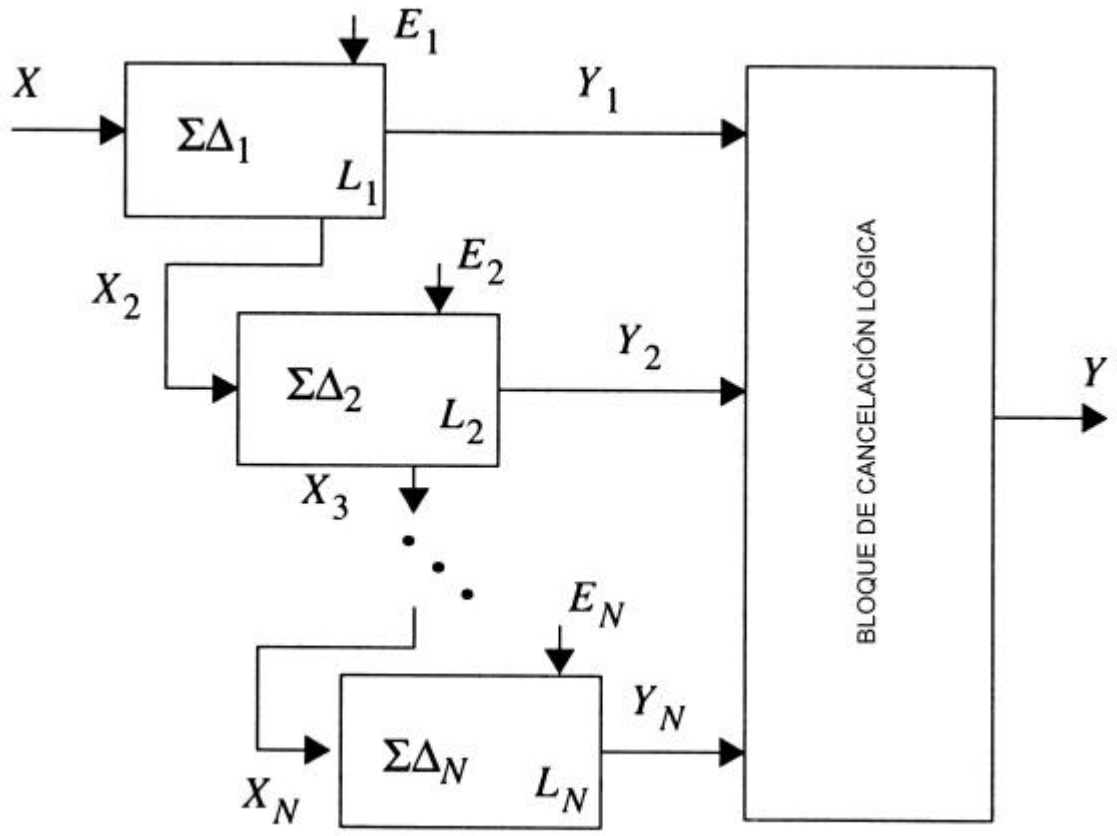



**Figura 1.10. Diagrama de bloques de un modulador en cascada Sigma-Delta**

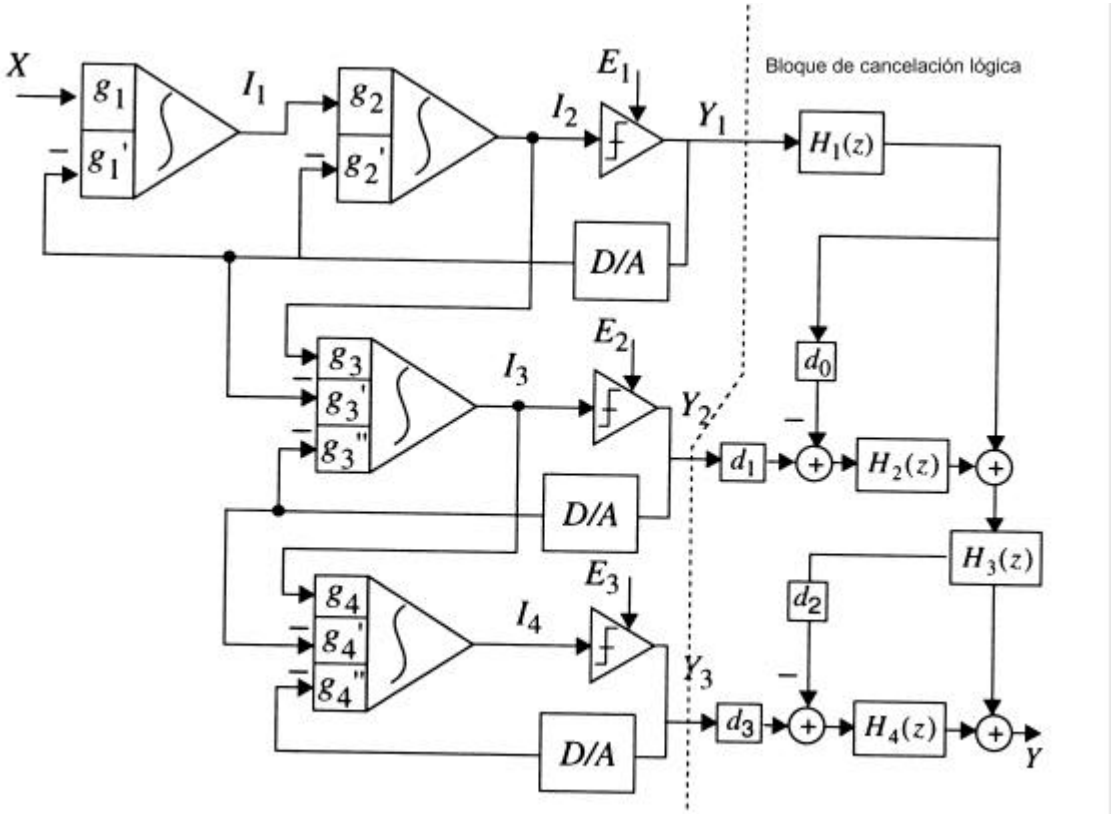



**Figura 1.11. Modulador en cascada de 4º orden 2-1-1**

| Analógico | Digital /Analógico | Digital |
|---|---|---|
| $g_1' = g1$ | $d_0 = 1 - g_3' / (g_1 g_2 g_3)$ | $H_1(z) = z^{-1}$ |
| $g_2' = 2 g_1' g2$ | $d_1 = g_3'' / (g_1 g_2 g_3)$ | $H_2(z) = (1 - z^{-1})^2$ |
| $g_4' = g_3'' g4$ | $d_2 = \left(1 - \frac{g_3'}{g_1 g_2 g_3}\right)\left(1 - \frac{g_4'}{g_3 g_4}\right) \cong 0$ | $H_3(z) = z^{-1}$ |
| | $d_3 = g_4'' / (g_1 g_2 g_3 g_4)$ | $H_4(z) = (1 - z^{-1})^3$ |

**Tabla 1.1. Relación entre coeficientes de la arquitectura 2-1-1**

Otra posibilidad para incrementar la resolución efectiva de los moduladores $\Sigma\Delta$ es incrementar el número de niveles de la cuantización interna [36][37]. Estos convertidores, tienen ventajas importantes:

- El anteriormente mencionado incremento de la resolución efectiva no depende de la razón de sobremuestreo, produciendo un rendimiento de 6 dB de reducción del ruido de cuantización por bit extra del cuantizador interno.

- El uso de la cuantización multibit mejora la estabilidad de los lazos de alto orden debido a que es más fácil anticipar la saturación del cuantizador

- Finalmente, con el gran número de niveles del cuantizador, el análisis aproximado más exacto se basará en la linealización de su curva de transferencia.

Sin embargo, sin técnicas apropiadas, la linearidad de un modulador ΣΔ multibit está limitada por la del convertidor D/A necesario en el camino de realimentación [38][39].

La Tabla 1.2 muestra una clasificación de las arquitecturas de los moduladores ΣΔ más usadas.

| | Ventajas | Desventajas |
|---|---|---|
| Single – loop, 1 – bit bajo orden | • Estabilidad<br>• Circuitería simple<br>• Rango de entrada útil máximo | • Se necesita alto valor de M<br>• Presencia de patrones de ruido |
| Single – loop, 1 – bit, alto orden | • Gran SNR para pequeña M<br>• Patrones de ruido pequeños | • Condicionalmente estable<br>• Rango de entrada útil más pequeño que el rango de entrada completo<br>• Necesidad de integradores de baja ganancia |
| Cascadade alto orden | • Gran SNR para pequeña M<br>• Estabilidad garantizada<br>• Rango de entrada útil máximo | • Sensibilidad a las imperfecciones de la circuitería<br>• Gran complejidad de la parte digital |
| Multi - bit | • Gran SNR para muy pequeña M<br>• Estabilidad mejorada<br>• Patrones de ruido pequeños | • Circuitería analógica y digital más compleja<br>• Sensibilidad a la no linealidad del DAC |

**Tabla 1.2. Sumario de arquitecturas Sigma - Delta**

# CAPÍTULO 2 Modelado de errores de moduladores Sigma-Delta

## 2.1. Introducción

Existen otros mecanismos de error aparte del ruido de cuantización asociados a las implementaciones eléctricas de los bloques de los convertidores $\Sigma\Delta$, aunque los convertidores $\Sigma\Delta$ sean menos sensibles que otras implementaciones. La importancia de estos errores es que se pueden llegar a convertir en las fuentes de error predominantes.

Es necesario analizar las no idealidades de los bloques por dos motivos:

- Por un lado la obtención de modelos de comportamiento que soporten una simulación en el dominio del tiempo rápida.

- Por otro lado la aproximación de las ecuaciones que expresan la potencia del error causado por la no idealidad como una función de si mismo y de otras variables de diseño.

Las no idealidades se pueden agrupar en dos categorías:

- Aquellas que producen cambios en la función de transferencia de la señal (*STF*) y la función de transferencia del ruido de cuantización (*NTF*). Este tipo de error depende fuertemente de la arquitectura del modulador

- Aquellas cuyo efecto puede ser modelado como una fuente de error en la entrada del integrador y que no altera la posición de los polos de su función de transferencia. Tradicionalmente, el análisis aproximado de estas no idealidades se limita a considerar la contribución del primer integrador de la cadena. Esto es posible porque los errores de este tipo generados en el primer integrador son añadidos directamente a la señal de entrada, por lo que no son atenuados en la banda base. Las contribuciones de los demás integradores a la potencia de error en banda son atenuadas por las diferentes potencias de la razón de sobremuestreo.

## 2.2. Ganancia finita en DC

En DC se supone que el integrador tiene una ganancia infinita, que es una característica imposible de obtener en la práctica.

Un integrador ideal como el visto en la Figura 2.12 [40] tiene como ecuaciones en diferencias finitas

$$v_{o,n} = v_{i,n-1} + v_{o,n-1} \tag{2.1}$$

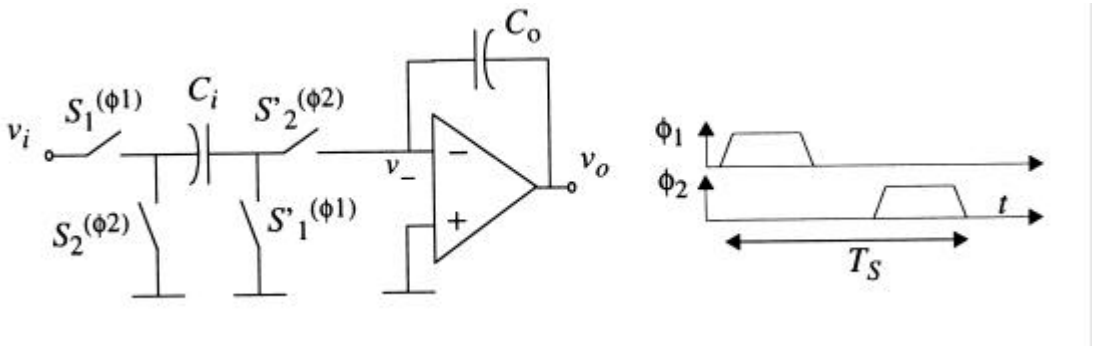



**Figura 2.12. Integrador SC ideal y sus fases de reloj**

Si consideramos el integrador SC de la Figura 2.13 dónde el amplificador ha sido modelado con una fuente de tensión controlada por tensión de ganancia Av>>1 las ecuaciones para el circuito son las siguientes,

$$v_{o,n} \cong \sum_{i=1}^{N^{\circ}ramas} \frac{A_V \cdot g_i}{A_V + 1 + g_i}\left(v_{i1,n-1} - v_{i2,n}\right) + \frac{A_V + 1}{A_V + 1 + g_i} v_{o,n-1} \tag{2.2}$$

En el dominio en z tenemos

$$\frac{v_o(z)}{v_i(z)} \cong \sum_{i=1}^{N^{\circ}ramas} \frac{g_i \cdot z^{-1}}{1-(1-m)z^{-1}}\left[v_{i1}(z) - v_{i2}(z)\right] \tag{2.3}$$

dónde $A_V >> 1$ y $m = g_i / A_V$, siendo $g_i = C_1 / C_2$ es el peso o ganancia del integrador.

Este resultado es conocido como pérdida de integración debido a que sólo una parte de la salida del integrador en el periodo anterior se añade a la nueva entrada. También produce un aumento en la potencia de ruido de cuantización. Este crecerá conforme aumente el orden del modulador. Para las arquitecturas en cascada se reduce cuanto más alto sea el orden del primer modulador. Sin embargo, este no puede ser de orden mayor que dos, debido a que estas arquitecturas son inestables.

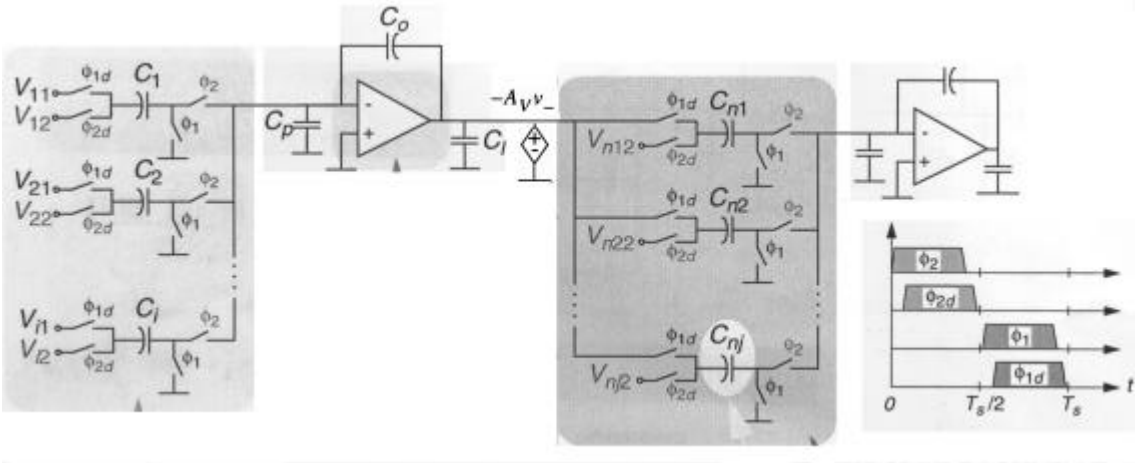



**Figura 2.13. Integrador SC con ganancia finita**

La Figura 2.14 muestra la ganancia del amplificador mínima para que la pérdida de resolución respecto al caso ideal sea sólo de un bit como una función de la razón de sobremuestreo.

Las curvas con orden especificado corresponden a moduladores single-loop. . El resto corresponden a moduladores en cascada en el que cada dígito expresa el orden de una etapa. Por ejemplo 2-1 se refiere a un modulador en cascada de dos etapas y tercer orden formado por una etapa de segundo orden y una de primero.

El precio que hay que pagar para solucionar los problemas de estabilidad asociados con los moduladores single-loop, es la aparente gran sensibilidad de las arquitecturas en cascada con respecto a las single-loop debido a su ganancia en DC.

Por otro lado se puede observar como el modulador 1-1-1-1 presenta mucha mas sensibilidad a la ganancia del integrador que el 2-1, siendo ambos de tercer orden.

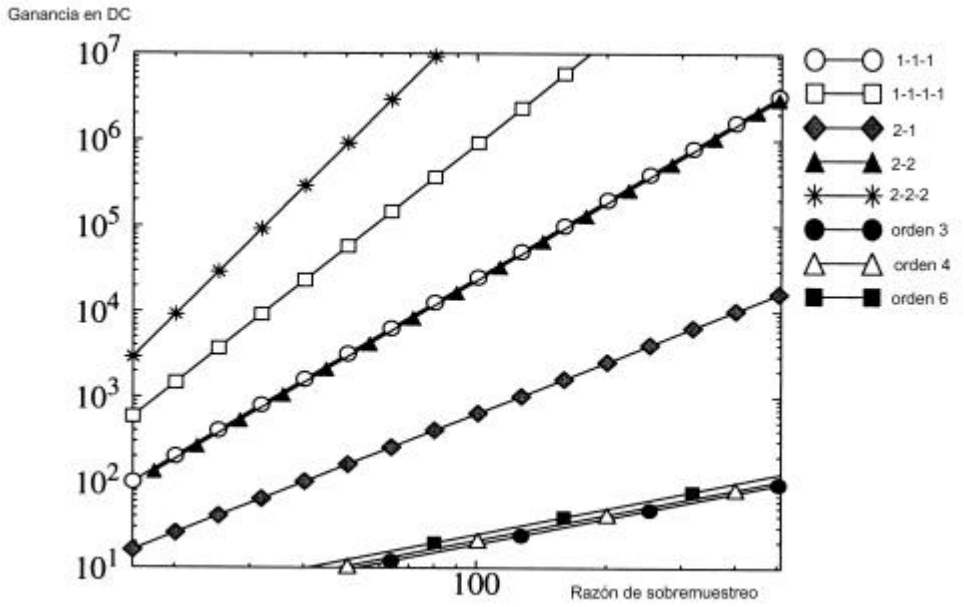



**Figura 2.14. Ganancia necesaria en DC para la pérdida de sólo un bit con respecto al ideal**

# 2.3. Desapareamiento de capacidades

En los circuitos SC, las ganancias tienen que ver con relaciones entre capacidades. Aunque estas relaciones se pueden obtener con mucha más precisión que los valores absolutos de las capacidades en sí mismas, éstas no están exentas de error durante el proceso de fabricación, por lo que las ganancias difieren de sus valores nominales.

Este tipo de error modifica la función de transferencia del integrador. Esta influencia depende de la arquitectura del modulador.

Por otro lado, la sensibilidad de los moduladores en cascada respecto a la ganancia finita del amplificador, se puede reducir a través de técnicas de circuito, mientras que el error debido al desapareamiento no se puede eliminar.

La Figura 2.15 muestra la desviación en la ganancia del integrador obtenida mediante simulación. Vemos que mientras que la desviación no sea mayor que el 1 % no se degrada la SNR. Esto es para un modulador de segundo orden, pero puede ser aplicado a otros moduladores single-loop.

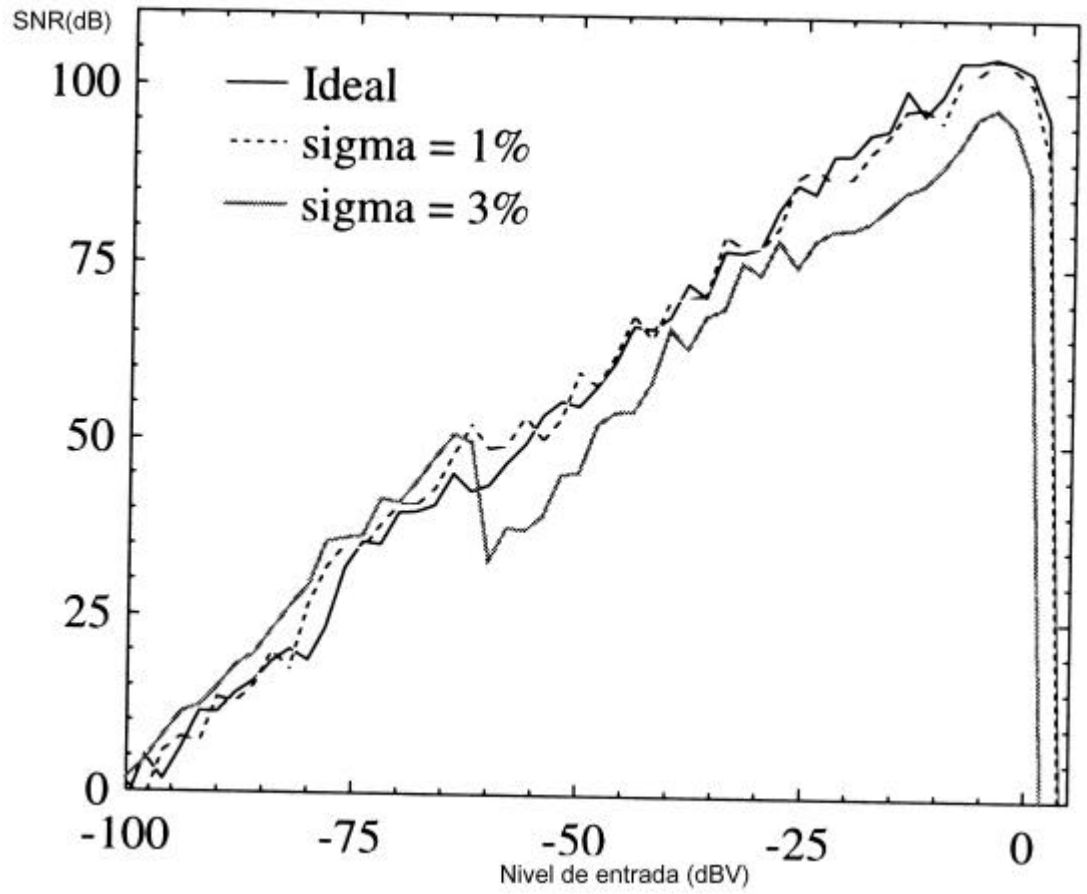



**Figura 2.15. Peor caso de SNR para algunas desviaciones en los pesos del integrador**

El problema crece cuando se consideran moduladores en cascada. La cancelación digital del ruido de cuantización, que es la base de operación de estos moduladores, requiere que ciertas relaciones entre las ganancias de los integradores y los coeficientes digitales sea exacta.

En las implementaciones SC, el desapareamiento entre la relación entre capacidades, modifica el valor de la ganancia de los integradores por lo que las relaciones entre capacidades varían, resultando incompleta la cancelación del ruido de cuantización de la primera etapa y por lo tanto aumento de la degradación del SNR.

# 2.4. Error de establecimiento o settling

## 2.4.1. Evolución de la tensión de salida

Consideraremos al integrador de la Figura 2.12. La evolución del voltaje de salida del integrador se puede ver en la Figura 2.16. Durante la fase de muestreo que corresponde al valor alto del reloj, la tensión no permanece constante, sino que hay una redistribución de carga como se puede ver en la siguiente figura. Analizaremos las ecuaciones para las distintas fases.

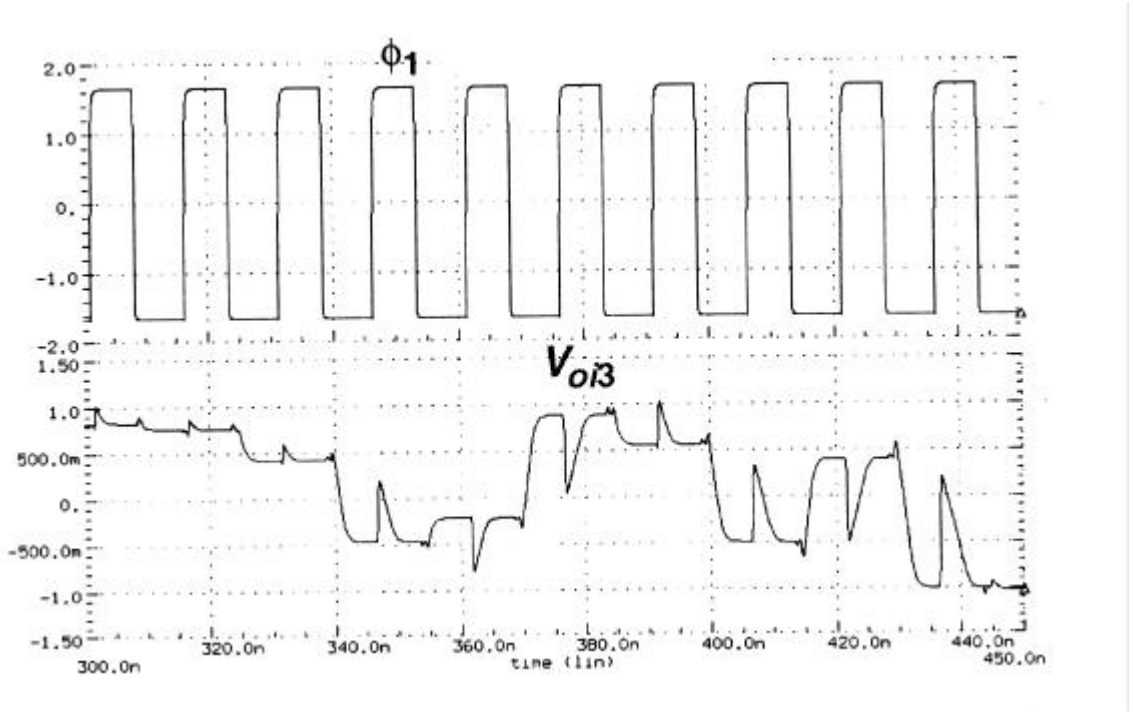



**Figura 2.16. Evolución de la tensión de salida de un integrador SC**

## 2.4.2. Fase de integración

Para el análisis se ha considerado un integrador de un solo polo como el que se puede ver en la Figura 2.17.

Debido al hecho de que la salida del integrador tiene alta impedancia, el principio de conservación de la carga hace que el voltaje de salida del integrador $v_o$, salte en la dirección opuesta a la del incremento final, presentando una discontinuidad en el instante en que los interruptores $f_2$ y $f_2^{'}$ se cierran [41]. Una evolución similar ocurre en la entrada del nodo del amplificador cuyo voltaje $v_a$, experimenta un salto hacia voltajes negativos si el incremento final de $v_o$ es positivo.

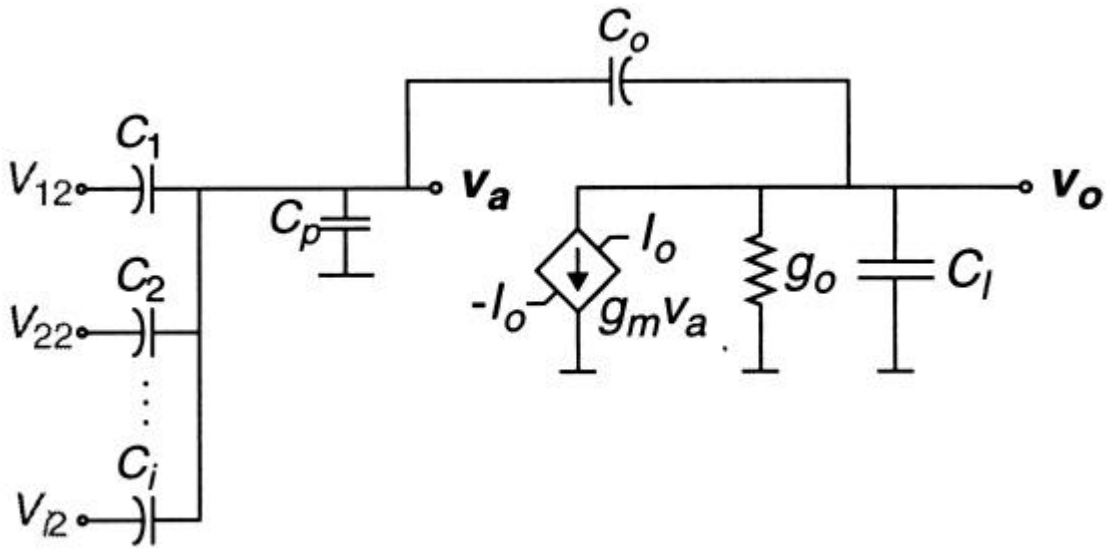



**Figura 2.17. Circuito equivalente durante la fase de integración**

Después de esta variación inicial, la transferencia de la carga del amplificador almacenada en $C_i$ pasa a $C_o$, durante el resto de la fase de integración. Por lo que $v_a(t)$, alcanza el cero y $v_o(t)$ su valor final $v_{of}$ impuesto por la función de transferencia del integrador.

El valor de $v_a(t)$ alcanzado inmediatamente después de la conmutación de los interruptores se puede obtener aplicando el principio de conservación de la carga antes y después de la conmutación y quedaría

$$v_{a\,i,i} = \frac{\left(1+\frac{C_l}{C_o}\right)}{C_{eq,i}} \sum_{j=0}^{n^o ramas} C_j (V_{j2} - V_{j1}) + \frac{C'}{C_{eq,i}} v_{a,n-1} \tag{2.4}$$

Siendo

$$C_{eq,i} = C_p + \sum_{j=0}^{N^o ramas} C_j + C_l \left( 1 + \frac{C_p + \sum_{j=0}^{N^o ramas} C_j}{C_o} \right) \tag{2.5}$$

Donde el valor de la tensión de salida en el instante inicial es

$$v_{o\,i,i} = v_{o,n-1} + \frac{C_o}{C_o + C_l} (v_{a\,i,i} - v_{a,n-1}) \tag{2.6}$$

Podemos distinguir tres regiones de operación [42]:

- *Zona lineal*: Pequeños incrementos de la entrada $|v_{ai}| < I_o/g_m$, donde $I_o$ es la máxima corriente de salida que puede ser suministrada por el amplificador y $g_m$ es la transconductancia. En este caso la salida del amplificador no se satura en corriente. La tensión en el nodo a, queda

$$v_a\left(\frac{T_s}{2}\right) = v_{a\,i,i} \cdot \exp\left(-\frac{g_m}{C_{eq,i}} \cdot \frac{T_s}{2}\right) \tag{2.7}$$

- *Zona saturada*: en la práctica al amplificador usualmente comienza la fase de integración fuera de su zona lineal, por lo que $|v_{ai}| > I_o/g_m$. Por lo que durante la fase de integración la fuente de corriente suple una corriente constante de valor Io. La tensión en el nodo a queda,

$$v_a\left(\frac{T_s}{2}\right) = v_{ai,i} - \frac{I_o}{C_{eq\,i}} \operatorname{sgn}(v_{a\,i,i}) \frac{T_s}{2} \tag{2.8}$$

- *Zona parcialmente saturada*: Durante la primera parte de la respuesta transitoria, la fuente de corriente controlada sminstra una fuente de corriente constante de valor Io. El amplificador entra en zona lineal cuando la igualdad $|v_a(t_o)| = I_o/g_m$ se alcanza y el tiempo en el que ocurre eso se puede calcular como,

$$t_o = \frac{C_{eq,i}}{I_o} |\, v_{ai,i} \,| - \frac{C_{eq\,i}}{g_m} \tag{2.9}$$

La tensión $v_a$ se calcula como sigue,

$$v_a\left(\frac{T_s}{2}\right) = \frac{I_o}{g_m} \operatorname{sgn}(v_{a\,i,i}) \exp\left[-\frac{g_m}{C_{eq,i}}\left(\frac{T_s}{2} - t_{o,i}\right)\right] \tag{2.10}$$

La salida del amplificador quedaría como sigue,

$$\begin{aligned} v_o\left(\frac{T_s}{2}\right) &= v_o(0) - \left(1 + \frac{C_p}{C_o}\right) v_a(0) - \\ &\sum_{j=0}^{N°ramas} \frac{C_j}{C_o}\left(V_{j2} - V_{j1}\right) + \left(1 + \frac{C_p + \sum_{j=0}^{N°ramas} C_j}{C_o}\right) v_a\left(\frac{T_s}{2}\right) \end{aligned} \tag{2.11}$$

## 2.4.3. Fase de muestreo

El settling del integrador llega a ser un factor limitante en los moduladores ΣΔ de altas frecuencias. Normalmente sólo se consideran los errores debido a la fase de integración, mientras que el error durante la fase de muestreo se omite.

La función de transferencia de carga durante el muestreo causa saltos en el voltaje del integrador.

El circuito equivalente durante la fase de muestreo se puede ver en la Figura 2.18

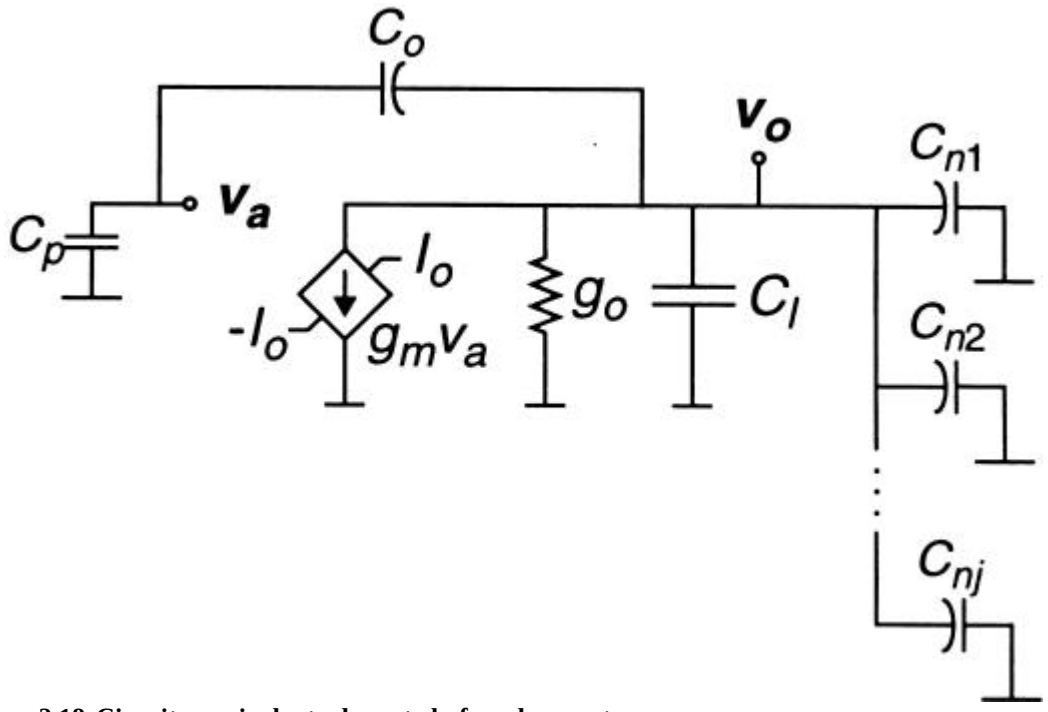



**Figura 2.18. Circuito equivalente durante la fase de muestreo**

El valor de $v_a$(t) alcanzado inmediatamente después de la conmutación de los interruptores se puede obtener aplicando el principio de conservación de la carga antes y después de la conmutación y quedaría

$$v_{a\,i,s} = v_a\left(\frac{T_s}{2}\right) - \sum_{j=1}^{N^o ramas} \frac{C_{nj}}{C_{e\,q,s}}\left[V_o\left(\frac{T_s}{2}\right) - \left(V_{nj\,2} - V_{an}\left(\frac{T_s}{2}\right)\right)\right] \tag{2.12}$$

siendo

$$C_{e\,q,s} = C_p + (C_l + \sum_{j=1}^{N^o ramas} C_{nj})\left(1 + \frac{C_p}{C_o}\right) \tag{2.13}$$

donde el valor de la tensión en el instante inicial es,

$$v_{o\,i,s} = v_o\left(\frac{T_s}{2}\right) + \left(1 + \frac{C_p}{C_o}\right)\left[v_{a\,i,s} - v_a\left(\frac{T_s}{2}\right)\right] \tag{2.14}$$

Al igual que en la fase de integración, podemos distinguir tres zonas de operación [42]:

- *Zona lineal*: Pequeños incrementos de la entrada $|v_{ai}| < I_o/g_m$, donde $I_o$ es la máxima corriente de salida que puede ser suministrada por el amplificador y $g_m$ es la transconductancia. En este caso la salida del amplificador no se satura en corriente. La tensión en el nodo a, queda

$$v_a(T_s) = v_{ai,s} \cdot \exp\left(-\frac{g_m \cdot T_s}{2C_{eq,s}}\right) \tag{2.15}$$

- *Zona saturada*: en la práctica el amplificador usualmente comienza la fase de muestreo estando fuera de su zona lineal, por lo que $|v_{ai}|>I_o/g_m$. Por lo que durante la fase de integración la fuente de corriente suple una corriente constante de valor $I_o$. La tensión en el nodo a queda,

$$v_a(T_s) = v_{ai,s} - \frac{I_o}{C_{eq,s}}\text{sgn}(v_{ai,s})\frac{T_s}{2} \tag{2.16}$$

- *Zona parcialmente saturada*: Durante la primera parte de la respuesta transitoria, la fuente de corriente controlada suministra una fuente de corriente constante de valor $I_o$. El amplificador entra en zona lineal cuando la igualdad $|v_a(t_o)|=I_o/g_m$ se alcanza y el instante en el que ocurre eso se puede calcular como,

$$t_{o,s} = \frac{T_s}{2} + \frac{C_{eq,s}}{I_o}|v_{ai,s}| - \frac{C_{eq,s}}{g_m} \tag{2.17}$$

La tensión $v_a$ se calcula como sigue,

$$v_a(T_s) = \frac{I_o}{g_m}\text{sgn}(v_{ai,s})\exp\left[-\frac{g_m}{C_{eq,s}}(T_s - t_{o,s})\right] \tag{2.18}$$

La salida del amplificador quedaría como sigue,

$$v_o(T_s) = v_o\left(\frac{T_s}{2}\right) + \left(1+\frac{C_p}{C_o}\right)\left[v_a(T_s) - v_a\left(\frac{T_s}{2}\right)\right] \tag{2.19}$$

# 2.5. Ruido térmico

Veremos las expresiones para el ruido térmico para un integrador como el de la Figura 2.19.

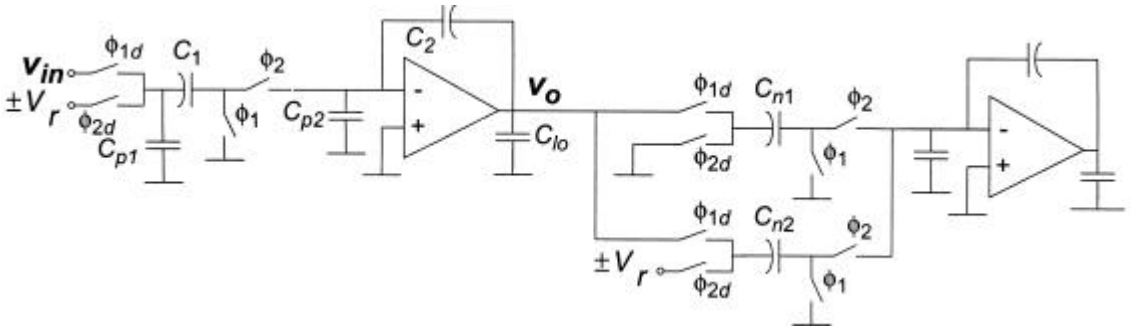



**Figura 2.19. Integrador SC de una rama**

En este tipo de arquitectura el ruido generado por la resistencia de llave de los interruptores, se muestrea junto con la señal a la entrada del condensador.

La necesidad de minimizar los errores en la transferencia de carga hace necesario una selección de constantes de tiempo que sean pequeñas comparadas con el periodo de reloj. Esto implica una frecuencia de corte para el ruido algunas veces más grande que la mitad de la frecuencia de muestreo. Este fenómeno, que puede incrementar la potencia de ruido térmico por un factor de aproximadamente 100 en un diseño típico, constituye el límite de resolución en los moduladores $\Sigma\Delta$-SC.

Consideraremos para el análisis lo siguiente

- En este modelo las fuentes de ruido han sido sustituidas por fuentes de tensión de valor igual a la raíz cuadrada del valor de ruido correspondiente.

- Las resistencias en ON conectadas a cada par de llaves han sido sustituidas por una resistencia equivalente cuyo valor ha sido supuesto idéntico para todas las ramas.

- Para simplificar el análisis se ha supuesto ganancia infinita para el amplificador.

- Consideraremos también que las fuentes son de ruido blanco y no correlacionadas, que permiten la aplicación del principio de superposición.

La Figura 2.20 muestra un modelo simplificado usado para sacar las ecuaciones de ruido. La figura (a) corresponde al circuito equivalente en la fase de muestreo, mientras que la figura (b) corresponde al circuito equivalente en la fase de integración:

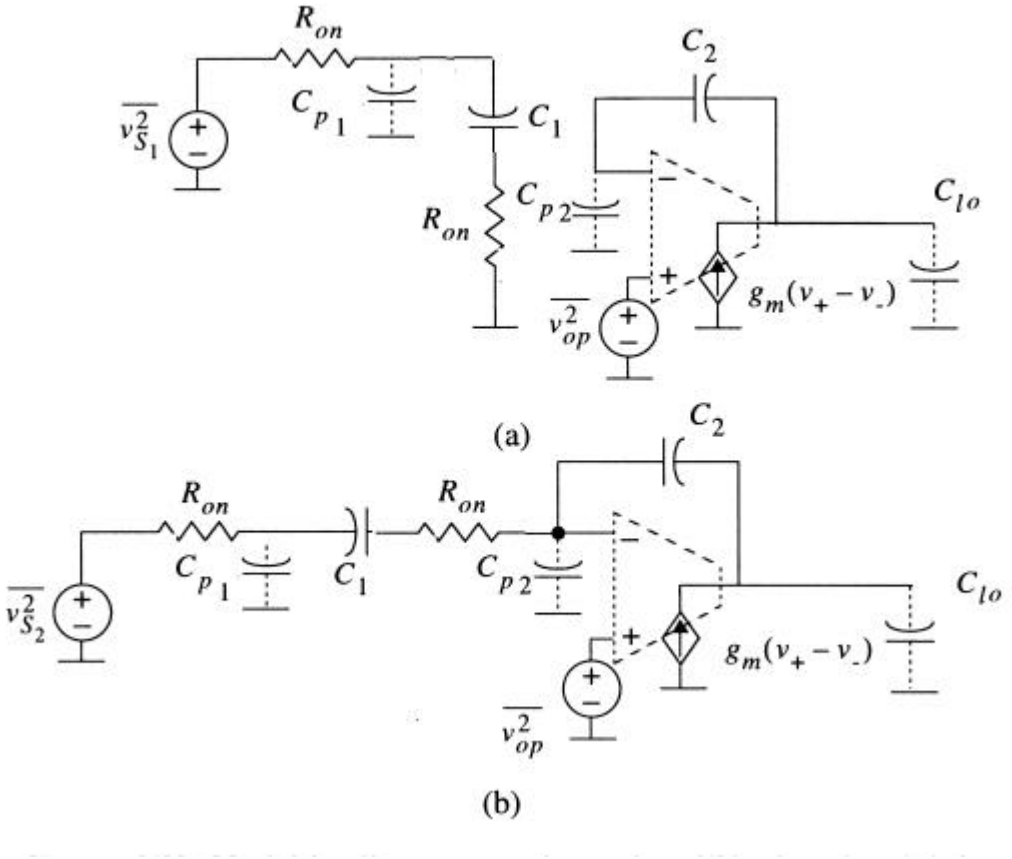



**Figura 2.20. Modelo de análisis de ruido de un integrador SC**

Durante la fase de muestreo, el ruido de los interruptores $f_1$ y $f_{1d}$ es muestreado por el condensador de entrada $C_{11}$. Al mismo tiempo, este ruido es filtrado por un filtro paso baja formado por la resistencia en ON de los interruptores y las capacidades, con una frecuencia de corte algunas veces más grande que la mitad de la frecuencia de muestreo. Tendríamos que la densidad espectral de potencia de ruido de ambos interruptores sería:

$$S_{f1d}(f) \cong 2kTR_{ON}\left[\frac{1}{2R_{ON}(2C_1 + C_{p1})f_s}\left(\frac{t_l}{T_s}\right)^2 + \left(\frac{t_s}{T_s}\right)\right]$$
$$S_{f1}(f) \cong 2kTR_{ON}\left[\frac{C_1 + C_{p1}}{2R_{ON}(2C_1 + C_{p1})f_s}\left(\frac{t_l}{T_s}\right)^2 + \left(\frac{t_s}{T_s}\right)\right] \qquad (2.20)$$

donde

$t_i$ es el tiempo efectivo de muestreo y

$t_s$ es el tiempo efectivo de integración

Que se pueden calcular como sigue

$$\begin{aligned} t_s &= T_s\,(1-dc) - ird \\ t_s &= T_s \cdot dc - srd \end{aligned} \tag{2.21}$$

donde

*dc* es el duty cicle
*ird* es la reducción del tiempo de integración
*srd* es la reducción del tiempo de muestreo

La densidad espectral de ruido debido al amplificador se puede calcular como sigue:

$$\begin{aligned} S_{OP}(f) &\cong Inpsd\left[\frac{BW_{OP}}{fs}\left(\frac{t_s}{T_s}\right)^2 + \left(\frac{t_l}{T_s}\right)\right] \\ BW_{OP} &\cong \frac{g_m \cdot C_2}{2(C_1C_2 + C_{lo}C_2 + C_1C_{lo} + C_{lo}C_{p2} + C_2C_{p2} + 2g_m \cdot R_{ON}C_1C_2} \end{aligned} \tag{2.22}$$

Donde *Inpsd* es la densidad espectral de ruido del amplificador referida a la entrada

Se ha supuesto que el producto gm·Ron es pequeño, lo que es lo mismo decir que la constante de tiempo RC es mucho más pequeña que la constante de tiempo del integrador.

En las implementaciones CMOS *Inpsd* tiene dos principales contribuciones [43]:

- Una componente de ruido flicker cuya densidad espectral es en primera aproximación inversamente proporcional a la frecuencia.
- una componente de ruido blanco de origen térmico

Ambas componentes están sujetas a la operación de muestreo. Sin embargo, la densidad de ruido flicker decrece muy rápido con el aumento de la frecuencia, por lo que llega a ser despreciable al ruido térmico.

El tratamiento de las contribuciones de ruido del interruptor $f_2$ se asemeja a la del amplificador. La respuesta en frecuencia es dada por:

$$\begin{aligned} S_2(f) &\cong 4kTR_{ON}\left[\frac{BW_2}{f_s}\left(\frac{t_S}{T_s}\right)^2 + \left(\frac{t_i}{T_s}\right)\right] \\ BW_2 &\cong \frac{C_{lo}C_{p2} + C_2C_{p2} + C_{lo}C_2 + 2C_1C_2R_{ON}g_m}{4R_{ON}C_1\left[C_1C_2 + C_{lo}C_{p2} + C_2C_{p2} + C_{lo}C_1 + C_{lo}C_2 + 2C_1C_2R_{ON}g_m\right]} \end{aligned} \tag{2.23}$$

La densidad espectral de potencia de ruido será la suma de las anteriores distribuciones.

El ruido térmico de los integradores se traslada a la salida degradando el funcionamiento. Sin embargo, para la dinámica del error, solo el ruido del primer integrador de la cadena se tomará en cuenta, porque se añade directamente a la señal y aparece sin filtrar en el espectro de salida.

Las contribuciones para la potencia de ruido térmico en banda del resto de los integradores son atenuados por diferentes potencias de la razón de sobremuestreo, dependiendo de la posición del integrador y en general pueden ser reducidos.

# 2.6. Distorsión debido a las capacidades no lineales

Las imperfecciones de los condensadores usados como elementos de memoria en los circuitos SC, influyen de manera crítica en su operación. Por otro lado la dependencia de los valores de las capacidades en la tensión almacenada, provoca un error en la transferencia de carga que es una fuente de distorsión en la salida del modulador.

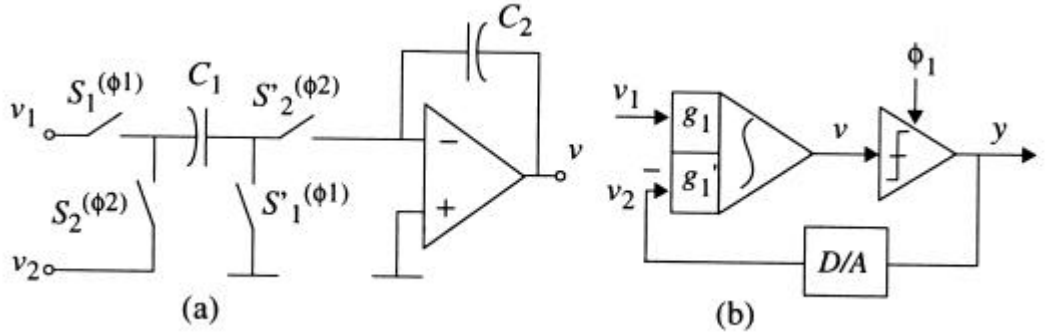



**Figura 2.21. Integrador SC**

Consideraremos el integrador SC de la Figura 2.21. Se supondrá que los condensadores tienen una dependencia no lineal con el voltaje almacenado de la siguiente forma:

$$C(v) = C^O(1 + a\,v + b\,v^2) \tag{2.24}$$

donde

$C^o$ representa la capacidad cuando el condensador está descargado
$a, b$ son los coeficientes no lineales

El valor de estos coeficientes se expresa en p.p.m/v y p.p.m/$V^2$ dependiendo de la técnica usada para la implementación del condensador.

En la práctica la dependencia no lineal con la tensión del condensador de integración es despreciable en el resultado final, por lo que no se considerará en el análisis. Considerando un amplificador operacional ideal y un elemento de carga diferencial almacenado en el condensador $C_1$ durante la fase $f_1$ tenemos,

$$dq1 = dq2 \Rightarrow C_1^o(1 + a\,v + b\,v^2)dv = C_2 dv \tag{2.25}$$

Integrando esta expresión entre el valor inicial y el valor final de la tensión de cada condensador después de un ciclo de reloj, tenemos la siguiente relación

$$V_n = V_{n-1} + g_1\left[V_{1,n-1}\left(1+\frac{a}{2}V_{1,n-1}+\frac{b}{3}V_{1,n-1}^2\right)-V_{2,n-1}\left(1+\frac{a}{2}V_{2,n-1}+\frac{b}{3}V_{2,n-1}^2\right)\right] \quad (2.26)$$

# 2.7. Ganancia no lineal

La ganancia no lineal en lazo abierto de los amplificadores introduce componentes de error como distorsión harmónica en el espectro de salida del modulador.

La no linealidad de la ganancia se manifiesta por su dependencia con la salida del amplificador. En la práctica todos los amplificadores presentan una ganancia no lineal debido a la transición entre la región lineal y la de saturación que es gradual [43], como la mostrada en la Figura 2.22. Se observa que la ganancia presenta un máximo en el centro de la escala y decrece con respecto al voltaje de salida hasta alcanzar el final de la región lineal, momento en el que abruptamente se sobrepasa el límite entre esta y la región de saturación.

Se supondrá que la dependencia del voltaje de salida se puede aproximar como un polinomio como sigue:

$$A_v = A_0\left(1+\sum_{i=1}^{\infty} g_i v^i\right) \quad (2.27)$$

Si consideramos el caso particular de la Figura 2.23, $g_1 = g_3 = 0$ debido a que la ganancia es simétrica. La simetría en la curva de ganancia es par y la función de transferencia, por lo tanto sólo consideraremos los términos pares y se truncarán a partir del término quinto por considerar que empiezan a ser despreciables. La ecuación anterior nos quedaría:

$$A_v = A_0\left(1+g_2 v^2+g_4 v^4\right) \quad (2.28)$$

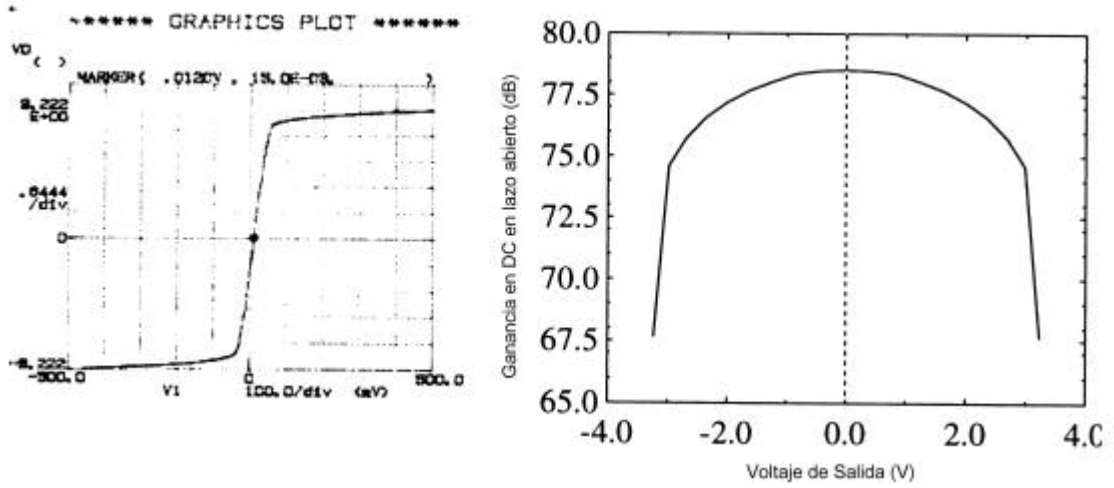



**Figura 2.22. (a) Curva de Transferencia DC de un amplificador CMOS**

**(b) Ganancia DC en lazo abierto como función del voltaje de salida**

Para entradas senoidales, se producen armónicos cuya potencia puede dominar la potencia de error en banda.

Para amplificadores reales, puede que la curva de la ganancia del integrador no sea simétrica. Sin embargo cada disimetría puede ser pequeña en un amplificador correctamente diseñado, por lo que los armónicos de orden par serán prácticamente nulos con respecto a los de orden impar.

La Figura 2.23 muestra el resultado de la simulación de comportamiento de una arquitectura en cascada de cuarto orden. Cuando se compara el espectro al caso ideal, se observa un incremento del ruido en la banda de la señal, claramente dominado por la aparición del tercer y quinto armónico.

# 2.8. Comparador

El impacto de las no idealidades del comparador en los moduladores ΣΔ es mucho más pequeña que la del integrador. Esto es debido a la posición que el comparador ocupa en el lazo del modulador.

En el caso del offset del comparador, éste es atenuado por la ganancia en DC de los integradores que lo preceden en el lazo. Debido a esto, especialmente cuando el número de los integradores en el lazo aumenta, los moduladores se hacen prácticamente insensibles a este error.

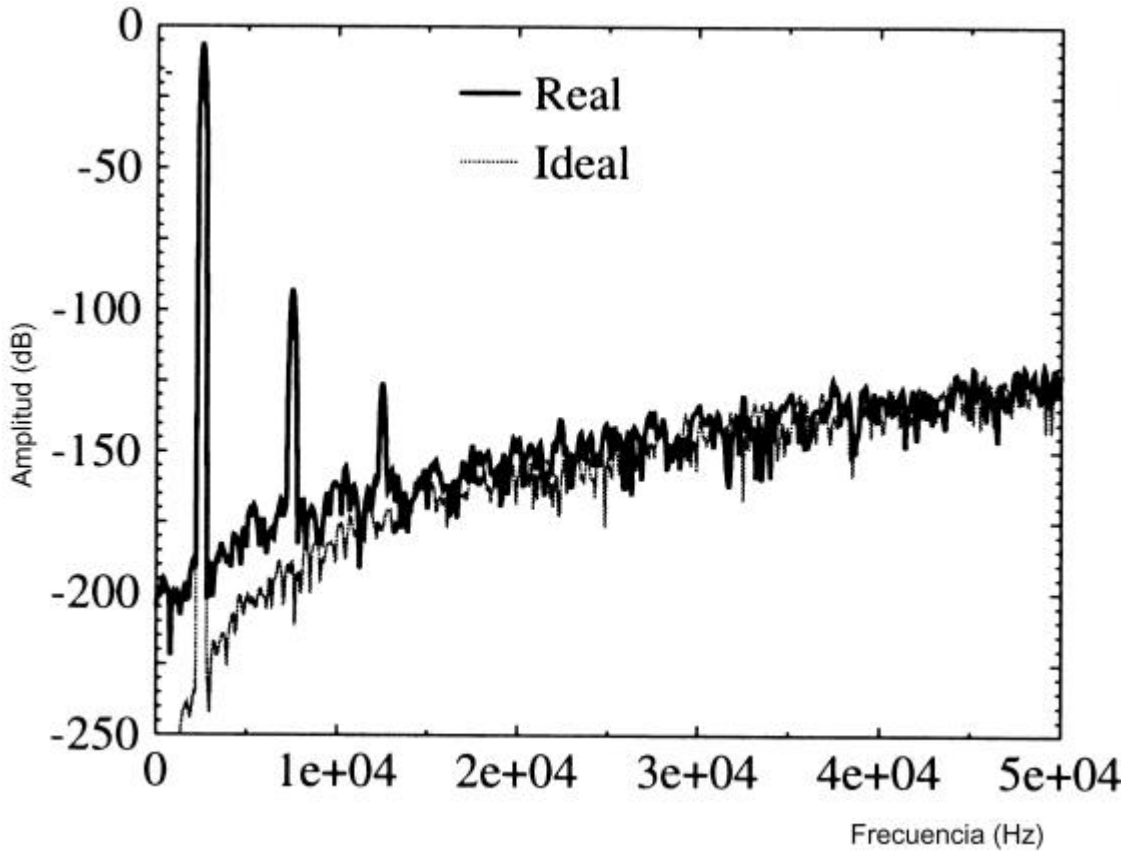


**Figura 2.23. Distorsión harmónica debida a la dinámica del integrador**

El mismo razonamiento puede ser aplicado a la histéresis del comparador, cuyo error recibe el mismo tratamiento que el ruido de cuantización.

En este caso no hay problemas relacionados con la no linealidad de la conversión.

La histéresis del comparador, lleva a una pérdida de resolución debida al hecho que para las señales siguientes al umbral de comparación, existe una resistencia a cambiar el estado cuando el nivel de la entrada puede haber sobrepasado el umbral.

Este fenómeno que es apreciado en ambas etapas cambia generando un ciclo de histéresis, debido a que el comparador tiene una memoria de su estado previo, siendo necesario hacer que conmute al estado correcto. En adición a este tipo de histéresis, que se llama determinista, hay otro de naturaleza aleatoria.

La Figura 2.24 muestra un ejemplo del offset y de la histéresis del comparador.

# 2.9. Cuantizador

Los moduladores con más de dos niveles de cuantización interna, también llamados moduladores multi-bit, son prácticamente insensibles a las no idealidades del cuantizador. Sin embargo, la conversión D/A de las señales en el lazo de realimentación, pueden ser afectadas por el error de no linealidad.

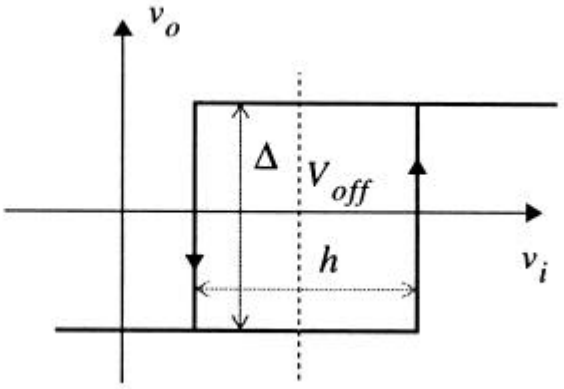



**Figura 2.24. Curva de transferencia de un comparador con histéresis**

En algunas arquitecturas, el error de conversión D/A es directamente añadido a la entrada del modulador y aparece a la salida, generalmente como distorsión. Debido a que los errores de no linealidad no se atenúan, la linealidad de los convertidores $\Sigma\Delta$ está subordinada a la del convertidor D/A interno, que afecta considerablemente al diseño.

Sin embargo, la aparición de técnicas novedosas [44][45][46], que han permitido paliar esta sensibilidad extrema, ha favorecido el uso de arquitecturas multi – bit y hacer su tratamiento interesante en términos de simulación de comportamiento.

En la Figura 2.25 se puede ver un sistema A/D/A de 3 bits.

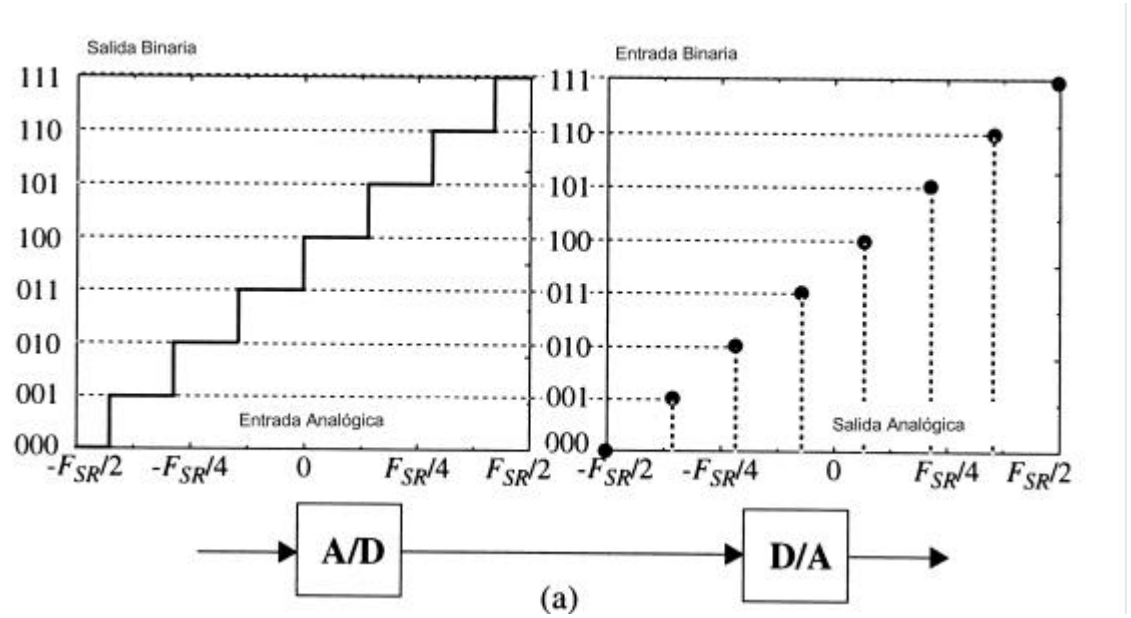



**Figura 2.25. Sistema A/D/A de 3 bits**

Un paso previo antes del modelado de un convertidor multi – bit A/D y D/A es el análisis de los mecanismos de error de su función de transferencia como la Figura 2.26.

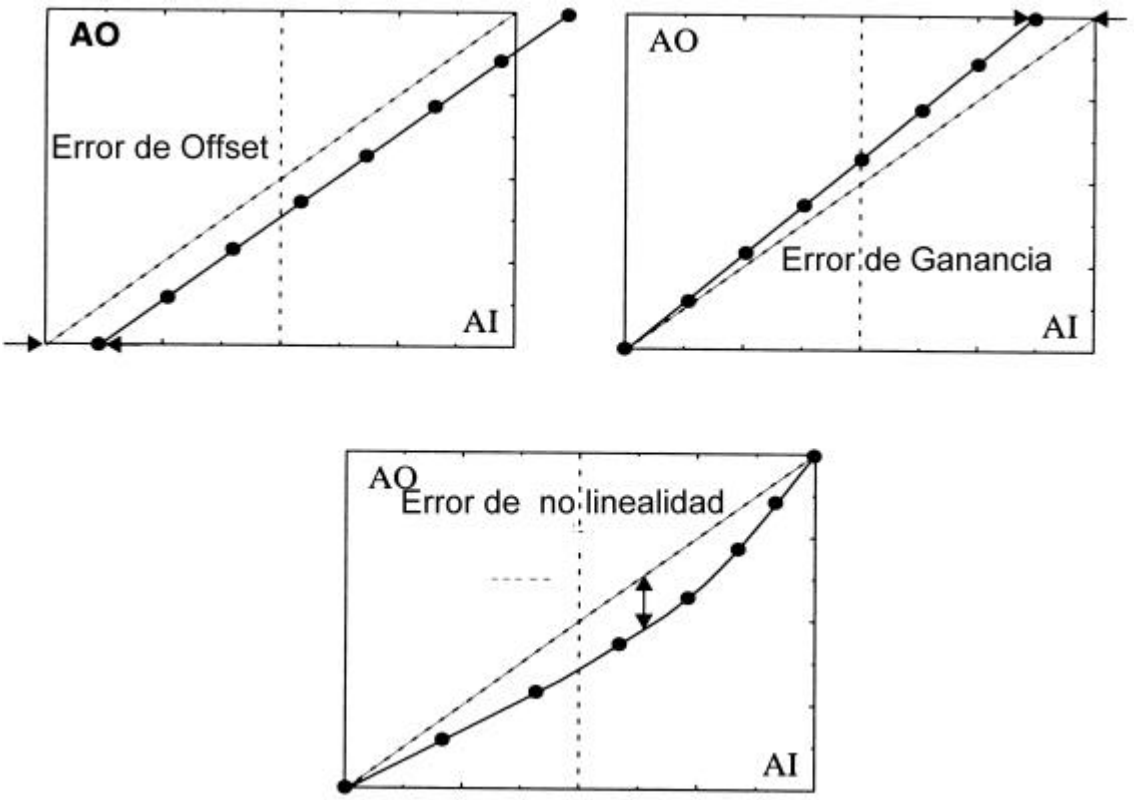



**Figura 2.26. Errores de la curva de transferencia en un cuantizador de 3 bits**

En ella se puede ver como el error de offset, supone tan sólo un desplazamiento horizontal de la curva, mientras que el error de ganancia altera la pendiente ideal de la curva. La no linealidad afecta de manera desigual a los valores de salida que se alcanzan, respecto al caso ideal. Entre los errores mostrados, el error correspondiente a no linealidad es el que presenta más problemas en un caso práctico.

Se representará la no linealidad de un convertidor por su no linealidad integral (INL), que se define como:

- Para un convertidor A/D, la máxima diferencia entre los valores analógicos que el actual umbral entre dos niveles adyacentes y su valor ideal una vez que el error de offset ha sido corregido como en la Figura 2.27.a.

- Para un convertidor D/A, la máxima diferencia entre la salida actual analógica y su valor ideal una vez que el error de offset y de ganancia hayan sido corregidos como en la Figura 2.27.b.

Se supondrá que el error de no linealidad será de tercer orden. Para ello se han usado los modelos de la Figura 2.28.

En un convertidor A/D, se añade primero un *off* a la entrada ideal $X_a$, y es multiplicado por el factor $g$ . El resultado sirve como entrada a un bloque no ideal con una función de transferencia $\Phi(.)$. Finalmente, la entrada $y_a$ es cuantizada por un convertidor A/D ideal.

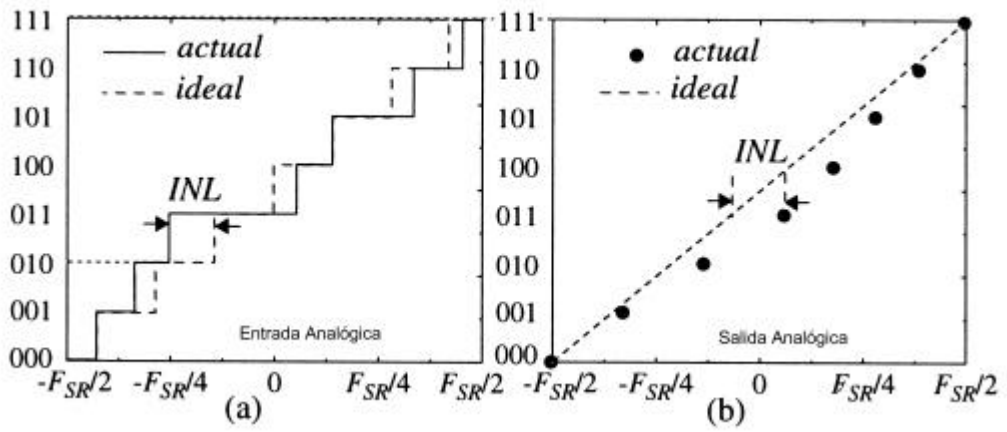



**Figura 2.27. Concepto de INL**

Para el convertidor D/A el modelo es el dual del anterior. Primero la entrada digital $x$, es trasladada al plano analógico mediante un convertidor ideal D/A. El resultado $x_a$ se pasa a través de un bloque no lineal y finalmente se añaden la ganancia y el error de offset.

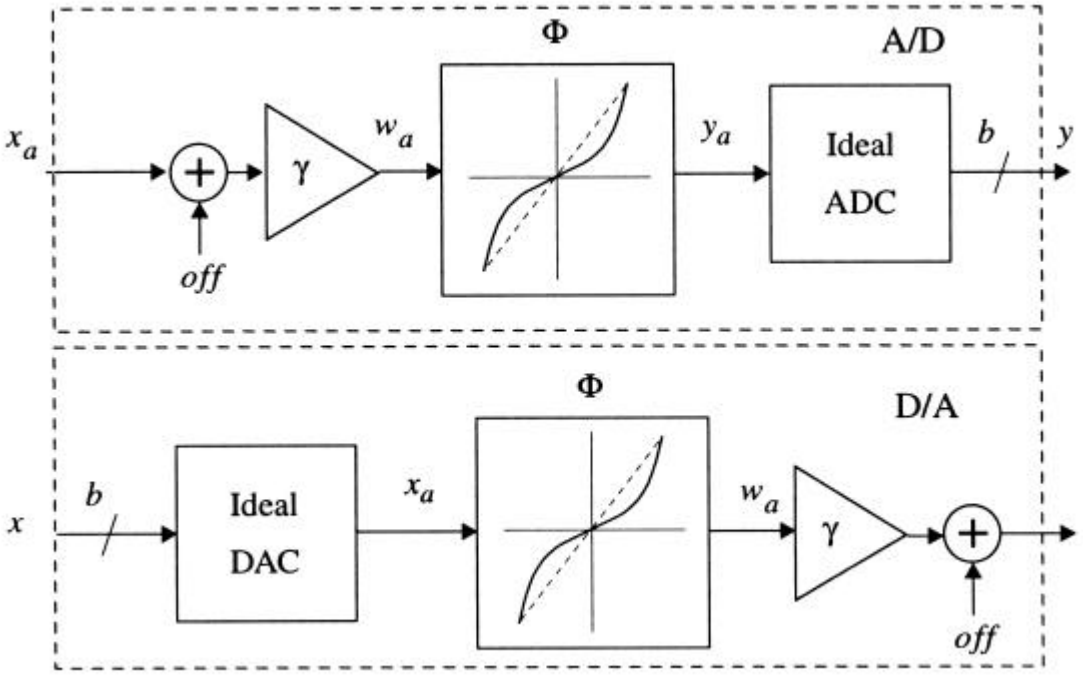



**Figura 2.28. Modelo de simulación de comportamiento de un convertidor A/D y D/A**

Los parámetros $f, g, off$ se pueden calcular como siguen para un convertidor A/D [28]:

$$y_a = \Phi(w_a) = (1-e_0)w_a + \frac{e_0}{A^2} w_a^3$$

$$e_0 = \frac{\sqrt{27}}{2^b - 2} INL \qquad w_a = g\,(x_a + off)$$

$$A = \left(2^{b-1} - 1\right) q \qquad g = 1/(1 + qe_g) \qquad (2.29)$$

$$l_1 = -F_{SR}/(2G) + q/2 \qquad off = (l_1 e_g - e_{off})q$$

Para un convertidor D/A, sólo cambiaría $l_1$, $y_a$, $w_a$ y tendríamos:

$$l_1 = -F_{SR}/2G$$

$$w_a = \Phi(x_a) = (1-e_0)x_a + \frac{e_0}{A^2} x_a^3 \qquad (2.30)$$

$$y_a = g\,(w_a + off)$$

donde

- $b$ es el número de bits

- *q* es la distancia entre valores correspondientes a escalones adyacentes, también llamados (LSB), que se obtiene dividiendo la escala completa $F_{SR}$, por el número de niveles.
- El parámetro G denota la ganancia nominal del convertidor que puede ser diferente de la unidad.

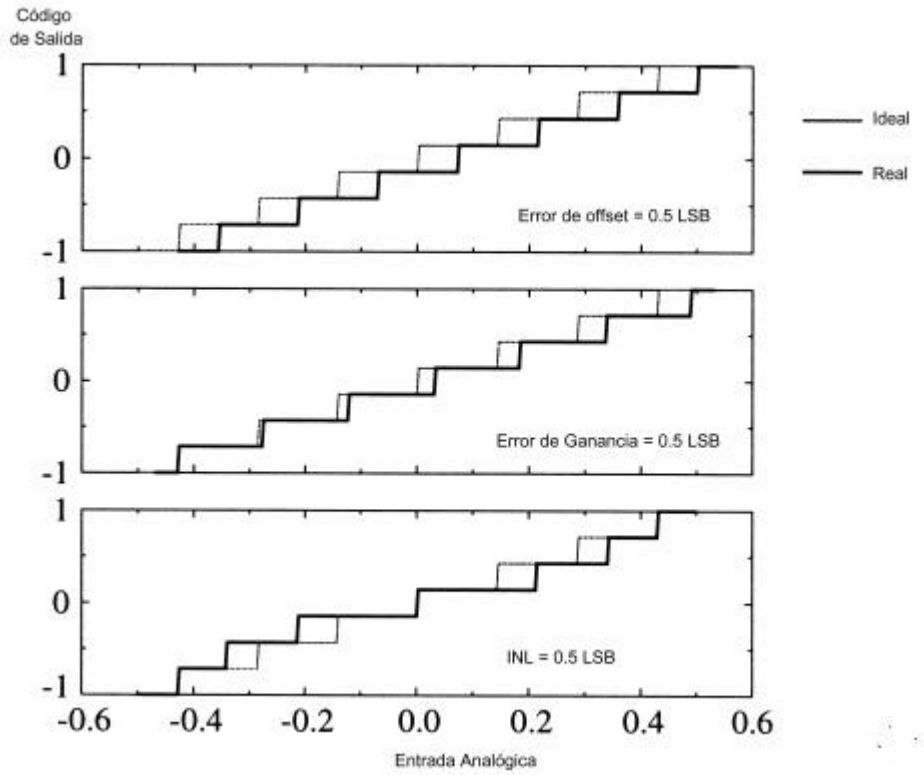



**Figura 2.29. Curvas de transferencia de un convertidor A/D de 3 bits**

En estas expresiones las unidades del error de ganancia, el error de offset, y el error de no linealidad se expresan en LSBs. Podemos ver un ejemplo en la Figura 2.29 que muestran las curvas de transferencia obtenidas incluyendo 0.5 LSB de offset, ganancia y error de no linealidad, en el modelo del convertidor A/D de 3 bits.

# CAPÍTULO 3 Modelado de comportamiento

## 3.1. Introducción

La simulación es un paso fundamental del proceso de diseño de los convertidores Sigma – Delta. Esto es aplicable a cualquier sistema electrónico con circuitos integrados, cuyos componentes no pueden ser modificados una vez manufacturados. Es crucial validar la síntesis a través de algún procedimiento que permita precisar la emulación del comportamiento de los circuitos reales.

Para celdas básicas este proceso es generalmente llevado a través de simulación eléctrica [47] cuya precisión, proveniente de modelos precisos para dispositivos físicos es alta.

En la simulación de circuitos digitales es fácil incrementar el grado de abstracción de celdas básicas simuladas eléctricamente de sistemas digitales simulados a nivel lógico sin pérdida de fiabilidad. Esto es debido a dos hechos:

- Por un lado, la estructura de circuitos digitales, que permiten una clara jerarquización de los sistemas.

- Por otro lado, observando una clase de reglas de diseño es suficiente para garantizar que la operación de los componentes de la jerarquía no afecta al resto de la circuitería.

Esto no ocurre en los circuitos analógicos o de señal mixta porque [48][49]:

- Por una lado, no es posible considerar los elementos de un circuito como entidades con funcionalidad independiente

- Por otro lado, la jerarquía en si misma no está bien definida.

En tal caso, es cierto que, si nosotros queremos preservar la calidad de la simulación al mismo tiempo que el grado de abstracción se incrementa, tenemos que continuar recurriendo a la simulación eléctrica.

Desafortunadamente, aunque teóricamente es posible, la simulación eléctrica de sistemas complejos, puede llegar a ser imposible en términos de recursos computacionales o tiempo transcurrido.

Particularmente, para convertidores de sobremuestreo, el análisis de su funcionamiento implica la extracción de un gran número de muestras de la salida del modulador que en términos de simulación eléctrica significa un análisis transitorio muy largo [49][50].

Si se usara este tipo de análisis, serían necesarios dos o tres semanas de tiempo de CPU para estimar la relación señal-ruido de un layout extraído de un modulador típico $\Sigma\Delta$, suponiendo que los ordenadores disponibles pudieran permanecer con la memoria requerida y con los recursos de cálculo.

Hay, por lo tanto, una clara necesidad de otros procedimientos que, con una pequeña pérdida de precisión, puedan acelerar la simulación. Se presentan algunas alternativas:

***a) Simulación eléctrica con macromodelos de celdas básicas.***

Se basa en la utilización de algoritmos numéricos que resuelven las ecuaciones diferenciales que provienen del análisis del circuito. Esta solución, es muy extendida entre los diseñadores de señal-mixta. Es una solución lenta, aunque el tratamiento de las celdas básicas se simplifique. Los algoritmos para resolver las ecuaciones diferenciales, son excesivamente costosos en tiempo de CPU [51].

***b) Simulación multi-nivel***

La aproximación intermedia, implica usar simuladores multi-nivel como ELDO [52] SABER [53] SWITCAP-2 [48]. En ELDO y SABER las partes críticas del sistema o aquellas que requieren una gran precisión son simuladas numéricamente, y modelado de comportamiento para emular el resto del sistema.

Es una solución interesante debido a que la solución de ecuaciones complejas es realizada por el simulador. Sin embargo la solución numérica de ecuaciones sigue siendo costosa en tiempo de CPU, por lo que la simulación multi-nivel es útil sólo cuando el número de muestras en el dominio del tiempo y la longitud del análisis transitorio no es excesivo.

Por otro lado en SWITCAP - 2 provee la posibilidad de simulaciones multi-nivel con circuitos de capacidades conmutadas y señales de control digital arbitrarias. Sin embargo, los modelos para elementos críticos de la topología, no incluían algunos de las no idealidades más limitantes en los moduladores $\Sigma\Delta$.

***c) Simulación de comportamiento***

La última opción, intenta traer la simulación de circuitos de señal mixta más cerca de la simulación lógica. El sistema se divide en subcircuitos o bloques constructivos, y cada bloque se describe con ecuaciones que relacionan su salida con su entrada y sus variables de estado internas. La precisión será mayor cuanto más aproximadamente describan el comportamiento real de cada bloque estas ecuaciones, por lo que resulta importante disponer de un modelado de comportamiento preciso, lo que implica un conocimiento profundo de la circuitería analógica y de los mecanismos de error.

Esta simulación es aplicable a circuitos controlados por señales periódicas de reloj que forman un importante grupo de los circuitos analógicos. En este tipo de circuitos no importan las evoluciones temporales de las señales de cada bloque, sino que tan sólo hace falta determinar el valor final en cada periodo.

A pesar de la pérdida de precisión con respecto a la simulación eléctrica, el tiempo de CPU se reduce drásticamente.

Entre las herramientas que se han usado para describir el comportamiento de los moduladores ΣΔ se encuentran los lenguajes de descripción de Hardware (HDL). Un lenguaje de descripción de Hardware, es un lenguaje que tiene construcciones especiales y semánticas para modelar, representar y simular el comportamiento funcional de un circuito en el tiempo .

La solución propuesta en este proyecto es utilizar los lenguajes de descripción de Hardware (HDL) y más específicamente VHDL [54] y VERILOG [55], para describir el comportamiento de los moduladores Sigma-Delta. La ventaja de esta solución es:

- VHDL Y VERILOG son dos estándares en la descripción de circuitos digitales. Esto hace posible que a los diseñadores de Sigma-Delta familiarizados con programación en estos lenguajes se les pueda dar el modelo de comportamiento de todos los bloques que componen los moduladores.

- La posibilidad de simular un circuito de señal mixta compuesto por los bloques correspondientes a moduladores Sigma-Delta y aquellos correspondientes a la parte digital, sin necesidad de cambiar de simulador

- La posibilidad de poder combinar estos bloques con otros escritos para HDL analógicos como VHDL-AMS y VERILOG-A que permitirían poder simular con una misma herramienta los bloques digitales y analógicos y aquellos como los moduladores Sigma-Delta, que usan simulación en tiempo discreto.

## 3.2. Simulación de comportamiento y bloques básicos

Definiremos los siguientes términos:

- *Bloque básico*: elemento de un sistema cuya salida es una función de su entrada y de su estado interno. Las leyes que rigen estas variables son llamadas modelo de comportamiento del bloque básico.

- *Simulación de comportamiento*: Un procedimiento de cálculo que permite la evaluación de la respuesta de un sistema una vez que la respuesta de sus bloques construidos se conozca en forma de modelo de comportamiento.

La primera definición implica la partición de un sistema complejo en bloques básicos con funcionalidad independiente. Se deben cumplir las dos siguientes condiciones:

- En dichos sistemas no hay lazos globales de realimentación que puedan relacionar la salida instantánea de un bloque básico consigo mismo.

- Si un lazo existe, debería haber un retraso de la señal para poder permitir esta dependencia.

Esta última condición se cumple en los moduladores $\Sigma\Delta$-SC. Consideraremos el integrador de la Figura 3.30. La salida del integrador cambia durante la fase $f_1$, que es cuando la carga almacenada en el condensador $C_i$, durante $f_2$, es transferida al condensador $C_o$. Esto significa, considerando que ambas fases tengan la misma duración, que la salida del integrador cambia acorde al valor de la entrada y la salida en la mitad del ciclo de reloj anterior.

Si el integrador es seguido por otro idéntico como en el caso de un modulador de segundo orden, que la salida del primero no es muestreada por el último hasta $f_1$, lo que supone medio periodo de retraso más.

Por lo tanto, el dato transmitido de un integrador a otro se obtiene con los valores de la entrada y salida del integrador anterior en el ciclo de reloj anterior. Cuando el lazo está cerrado a través de un cuantizador y un convertidor D/A, no habrá dependencia instantánea entre la salida del integrador y su entrada, por lo que es posible considerar al integrador como un bloque básico. Este razonamiento se puede extender a otros bloques del modulador $\Sigma\Delta$-SC.

El conocimiento de la funcionalidad de bloques básicos a través de sus ecuaciones es esencial para la simulación de comportamiento de cualquier sistema. Las ecuaciones pueden ser simples, que representen propiamente la salida del sistema bajo condiciones ideales, por ejemplo

$$v_o\left[nT_s\right] = \frac{C_i}{C_o} v_i\left[\left(n-1\right)T_s\right] + v_o\left[\left(n-1\right)T_s\right] \tag{3.1}$$

donde $T_s$ es el periodo de muestreo.

Los bloques fundamentales de los moduladores $\Sigma\Delta$ son;

- Integradores
- Comparadores
- Cuantizadores
- Convertidores D/A
- Bloques digitales

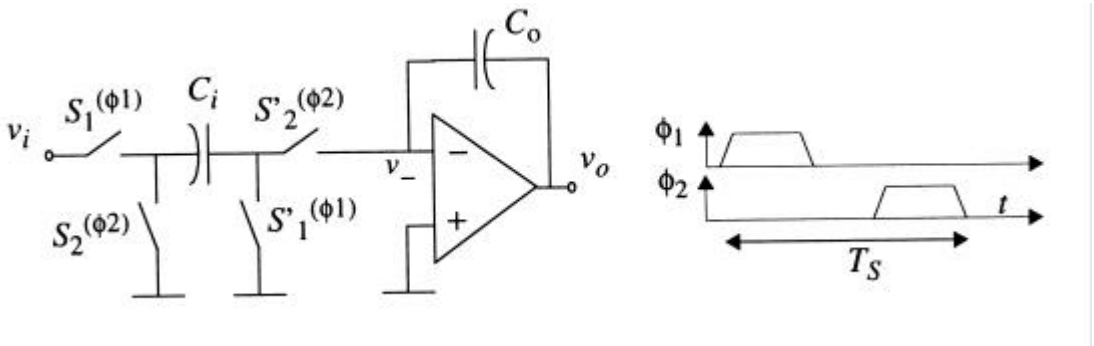



**Figura 3.30. Integrador SC y sus fases de reloj**

A través de la adecuada conexión de estos pocos bloques, se pueden obtener un gran número de arquitecturas cuyo comportamiento puede ser afectado por los errores. Se supondrá que la parte digital no degrada el funcionamiento del modulador.

Existen además, otros bloques básicos que son bloques auxiliares con un modelo de comportamiento muy simple especialmente útil para incrementar la abstracción de las simulaciones como:

- Sumadores
- Multiplicadores
- Retrasos
- Amplificadores

Por ejemplo el sistema de la Figura 3.31. donde D es un elemento de retraso que representa la funcionalidad del integrador de la Figura 3.30.

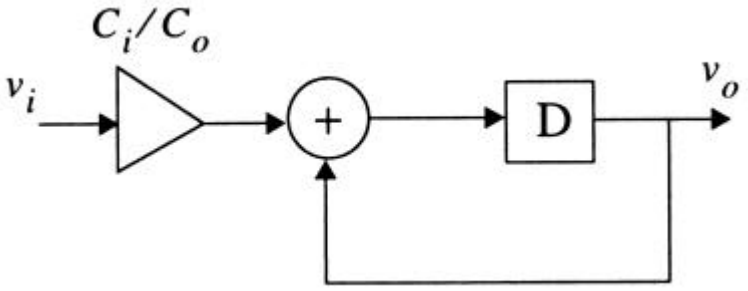



**Figura 3.31. Diagrama de bloques de un integrador ideal**

# 3.3. Características de VHDL y VERILOG

La versión de VHDL utilizada en este proyecto se refiere al estándar de la IEEE conocido como STD 1076-1993.

La versión de VERILOG utilizada en este proyecto se refiere al estándar IEEE 1364 - 2001

### 3.3.1. Estructura de un fichero en VERILOG

Un *modulo* en VERILOG [55] es una encapsulación de información representando la estructura, comportamiento y otras propiedades importantes de un bloque básico. La declaración de un módulo siempre se termina por la palabra *endmodule*.

Un módulo puede tener puertos que provean un interfaz con el entorno. Los nombres en un puerto corresponden a las señales eléctricas del hardware, siendo determinados por el entorno externo.

A un módulo también se le pueden pasar parámetros para que internamente se operen con ellos. Estos módulos necesitarán de variables locales para operar dentro de ellos.

El funcionamiento de un bloque básico se puede implementar abstractamente a través de declaraciones de procedimientos. Esto se llama en VERILOG modelo de comportamiento.

Los modelos de comportamiento tienen una parte de declaraciones iniciales y otra que suele tener un comportamiento síncrono porque el flujo de actividad está sincronizado normalmente con una señal de reloj y cíclico porque ese flujo de actividad se repite.

El esquema de esta descripción se puede ver en la Figura 3.32 que corresponde a un bloque que va generando muestras de una señal senoidal cada periodo:

Para simular un circuito que contenga varios bloques básicos, VERILOG necesita usar un módulo de test, que es aquél que no tiene ningún puerto de entrada o salida. VERILOG necesita señales internas (cables), que se usen para establecer la conectividad con otros bloques instanciados dentro del modulo. Un módulo que contiene información estructural tendrá una o más instancias de otros módulos. Como se puede ver en la Figura 3.33, correspondiente a un modulador single – loop de segundo orden como el de la Figura 1.7.

### 3.3.2. Estructura de un fichero en VHDL

Una entidad en VHDL es la representación de la estructura de un bloque básico. La declaración de una entidad siempre termina con la palabra END y el nombre de la entidad.

Una entidad puede tener puertos que provean un interfaz con el entorno. Los nombres en un puerto corresponden a las señales eléctricas del hardware, siendo determinados por el entorno externo.

A una entidad también se le pueden pasar parámetros para que internamente se operen con ellos, estos parámetros reciben el nombre de GENERICS.

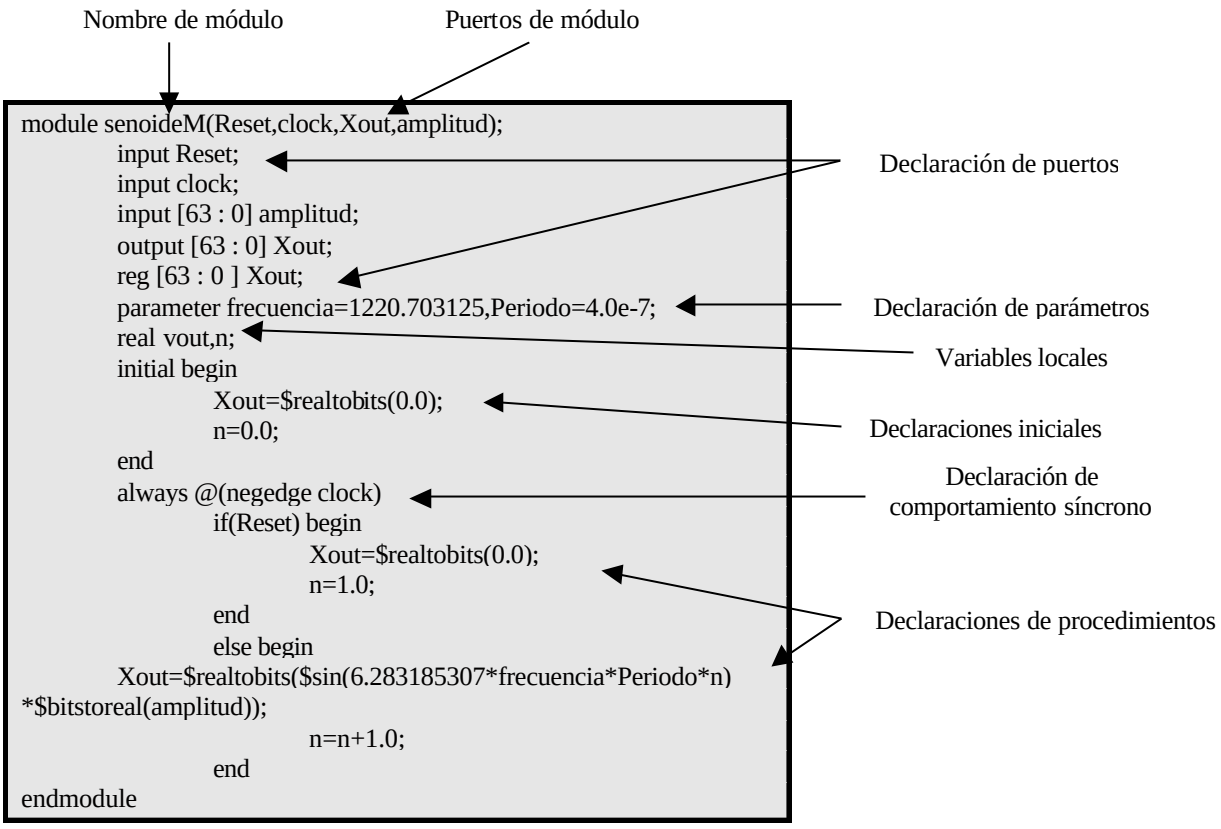



**Figura 3.32. Descripción un módulo en VERILOG**

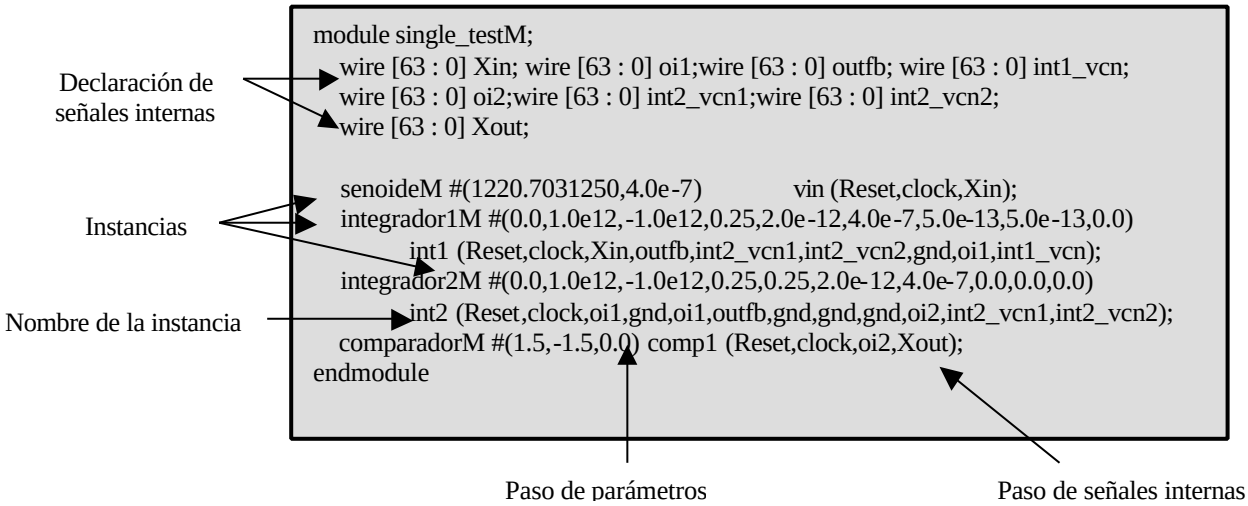



**Figura 3.33. Módulo de test en VERILOG**

El comportamiento de una entidad se describe a través de su arquitectura. Dentro de una arquitectura puede haber varios procesos que se ejecuten al mismo tiempo, cada uno de ellos tendrá un comportamiento síncrono con su señal de reloj correspondiente. Estos procesos necesitarán de variables locales para operar dentro de ellos. Todo esto se puede ver en la Figura 3.32 correspondiente a un bloque que genera muestras de una señal senoidal cada periodo.

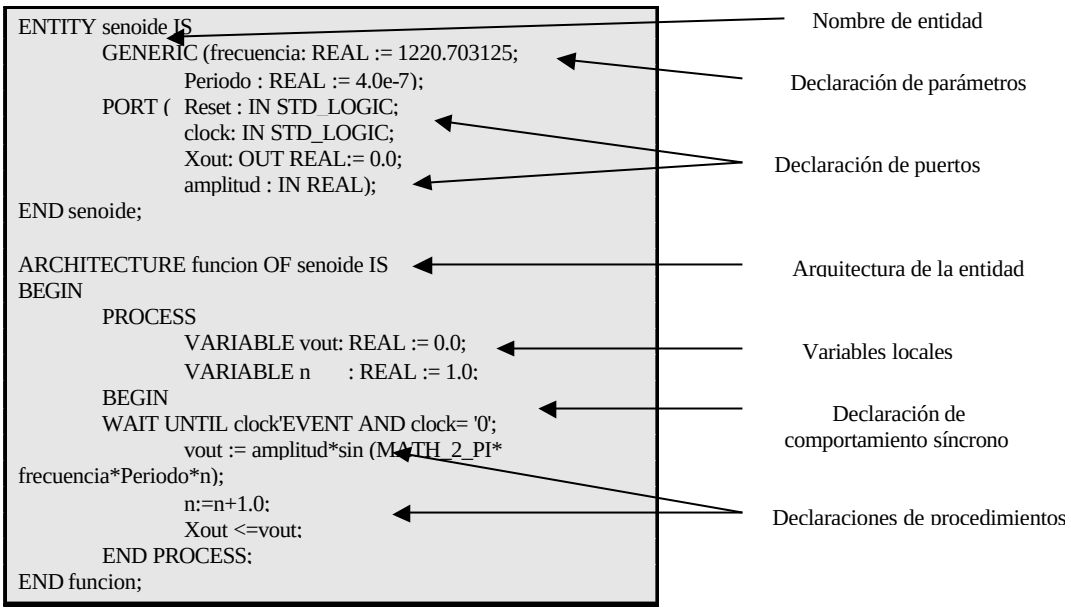



**Figura 3.34. Descripción de un bloque básico en VHDL**

Para simular un circuito que contenga varios bloques básicos, VHDL necesita usar una entidad de test, que es aquella que no tiene ningún puerto de entrada o salida y ningún parámetro de entrada. VHDL, en la descripción de la arquitectura necesita señales internas (cables), que se usen para establecer la conectividad con otros bloques instanciados dentro de la entidad. Una entidad que contiene información estructural tendrá una o más instancias de otras entidades, como se puede ver en la Figura 3.35 correspondiente a un modulador single – loop de segundo orden como el de la Figura 1.7.

## 3.3.3. Limitaciones en VERILOG

El lenguaje tiene una serie de limitaciones importantes que se han ido solventando de la siguiente manera.

### 3.3.3.1. Limitación en el número de puertos.

Un módulo en VERILOG, no puede tener un número indeterminado de puertos de entrada – salida. Esto significa que para bloques básicos con un número indeterminado de entradas, como los integradores o como los sumadores, se han tenido que programar varios módulos, uno por cada bloque con un número de entradas diferentes.

En el caso del integrador, se han programado bloques para integradores de una, dos o tres ramas. Para el sumador se han programado sólo un bloque de dos entradas. Y uno de una entrada que es un amplificador.

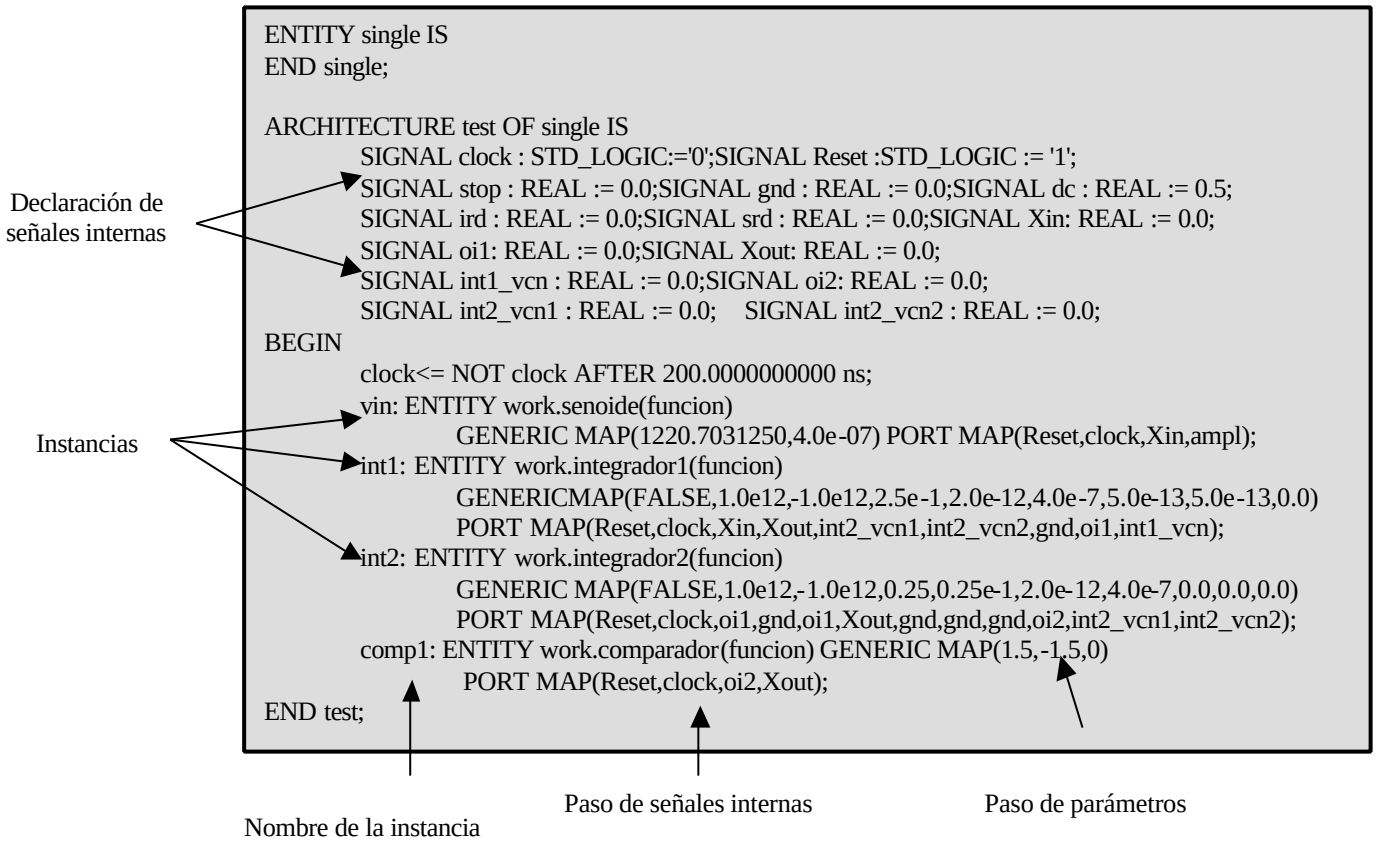



**Figura 3.35. Test en VHDL**

### 3.3.3.2. Imposibilidad de variar los parámetros

La simulación de moduladores $\Sigma\Delta$, requiere de análisis transitorios y análisis paramétricos, necesarios para ajustar los parámetros de diseño. Estos análisis necesitan variar esos parámetros.

En VERILOG existen puertos, que se corresponden a señales eléctricas del hardware y parámetros que se le pasan a esos bloques, para que operen internamente con ellos. Estos parámetros son constantes y no se pueden variar durante el transcurso de una simulación, esto es un inconveniente para los tipos de análisis anteriormente mencionados.

Para solventar este inconveniente todos los parámetros que se pueden necesitar variar durante el transcurso de la simulación se le pasan al módulo como puertos de entrada. Esto incrementa el número de entradas necesarias al bloque básico. Esto se ilustra en la Figura 3.36.

### 3.3.3.3. Necesidad de transformar las señales a señales binarias

VERILOG está pensado para simular el comportamiento de circuitos digitales. Por ello, los puertos, tienen que ser necesariamente señales digitales, de un único o varios bits. Esto imposibilitaría simular el comportamiento de moduladores Sigma-Delta cuyas entradas y salidas son analógicas, esto no ocurre en VERILOG – A [56], cuyas entradas y salidas también pueden ser señales reales.

Para solventar este problema, VERILOG dispone de una serie de funciones (system tasks) que transforman una señal digital de 64 bits en un número real y viceversa. Estas system tasks son $bitstoreal() y $realtobits(), que transforma una señal digital en un número real y viceversa.

Esto se puede ver en la Figura 3.37:

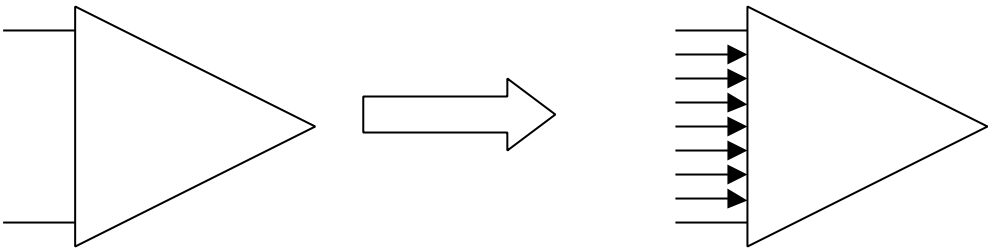


**Figura 3.36. Incremento de entradas de un bloque básico**

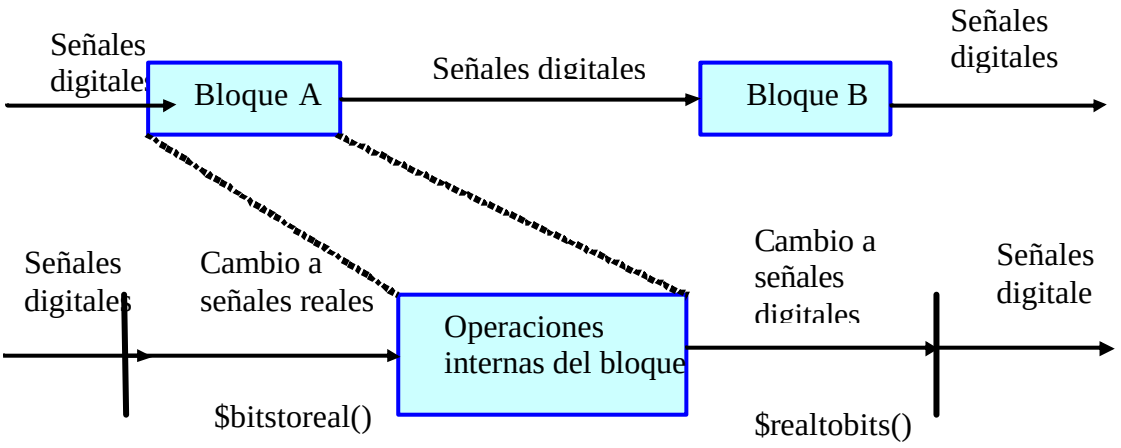



**Figura 3.37. Transformación de señales digitales a reales y viceversa**

### 3.3.3.4. PLI

VERILOG al estar pensado para simulación de circuitos digitales, no tiene implementado funciones matemáticas básicas como seno, coseno...

Para solventar esto se utiliza un interfaz de lenguaje de programación (PLI) que permite crear un lenguaje VERILOG con system tasks definidas por el usuario que son implementadas en el lenguaje de programación C.

Un simulador crea una clase de estructuras cuando compila un modulo de VERILOG. Estas estructuras de datos contienen información topológica y de otro tipo sobre el diseño. PLI incluye una biblioteca de funciones de C que pueden acceder directamente a las estructuras de datos del diseño, dando al usuario acceso a gran cantidad de información para poder soportar otras aplicaciones.

Se han implementado mediante este método funciones matemáticas básicas como seno, exponencial, logaritmo y raíz cuadrada. También se han implementado dos funciones para escritura de números en un fichero.

El procedimiento para crear estas system tasks mediante PLI, se explicará en el apéndice D.

### 3.3.4. Limitaciones en VHDL

Las limitaciones en el número de puertos y la imposibilidad de variar los parámetros de entrada también son características de VHDL.

Sin embargo, VHDL a diferencia de VERILOG, dispone además de una biblioteca de funciones matemáticas básicas y tiene contemplada la posibilidad de poder trabajar con puertos de entrada – salida no solamente digitales. Por tanto no necesita convertir las señales digitales a reales y viceversa.

VHDL dispone de una biblioteca para escritura de archivos, sin embargo, dicha biblioteca no viene implementada en el simulador ADVANCE MS. Para solventar esto, se ha llevado a cabo un procedimiento parecido a las funciones PLI en VERILOG que consiste en encapsular funciones C como si fueran funciones de VHDL. El procedimiento para encapsular funciones C en VHDL se explicará en el apéndice D.

### 3.3.5. Sincronización en VHDL y VERILOG

El modelado que se hace en VHDL y VERILOG de los distintos bloques básicos esta guiado por eventos. Esto significa que únicamente nos preocupa el valor que toma cada variable al final de cada fase de reloj, no sus valores intermedios. La Figura 3.38 muestra la división del reloj en dos fases:

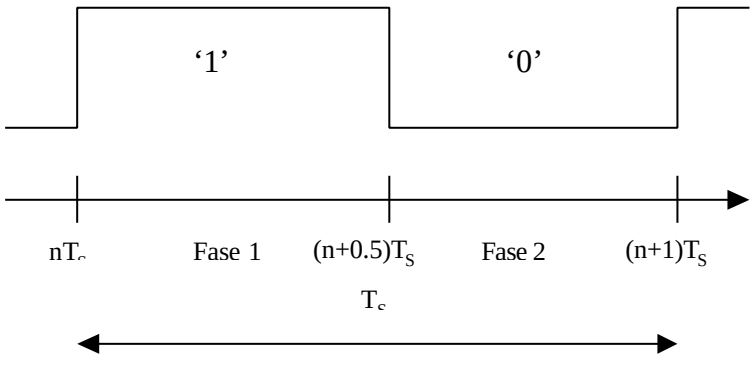



**Figura 3.38. Diagrama de fases de tiempo discreto**

Como se puede ver en la figura anterior la sincronización en VHDL se ha dispuesto que sea un ‘1’ o estado alto para la fase 1 y un ‘0’ o estado bajo para la fase 2, que corresponden respectivamente a la fase de muestreo e integración del modulador.

En VHDL y VERILOG hay dos tipos de bloque básicos:

- Aquellos que necesitan de una sincronización con los flancos de un reloj, como los integradores, los comparadores, cuantizadores y convertidores DA y los bloque de retraso.

- Aquellos que solo dependen de sus entradas para cambiar como lo sumadores y amplificadores, y los multiplicadores.

Los bloques en VERILOG Y VHDL se han construido de forma síncrona, por lo que necesitan una señal de control para cambiar su estado. Esto se hace en ambos lenguajes de la siguiente forma.

En VHDL se utiliza la sentencia *WAIT UNTIL señal_de_control'EVENT* para sincronización. Esta señal de control será la señal de reloj para el primer tipo de bloque básico y será la señal o señales de entrada para el segundo tipo de bloques básicos.

En VERILOG se utiliza el identificador *"@(señal de control)"*. De igual forma la señal de control, será la señal de reloj para el primer tipo de bloque básico y las señales de entrada para el segundo tipo.

Cuando la señal de control es la señal de reloj, se puede sincronizar para que la salida solamente cambie con los flancos de bajada o los flancos de subida del circuito. Sin embargo, esta opción se ha descartado para el primer tipo de bloques debido a que se necesitan tomar salidas distintas de estos bloques en función de que estén en la fase de muestreo o la fase de integración.

# 3.4. Integrador

El integrador se puede considerar como el bloque fundamental porque debido a su posición las no idealidades afectan en gran medida la operación de los moduladores $\Sigma\Delta$ [2].

Particularmente, usando técnicas SC, entre las causas que empeoran la operación de los moduladores se encuentran [57][58][59]:

- Error de transferencia de carga incompleta
- Error de no linealidad
- Ruido térmico

Todas estas son asociadas con las no idealidades de los componentes básicos del integrador SC, que son amplificadores, capacidades y llaves analógicas

Las no idealidades se estudiaron en el capítulo anterior, para obtener expresiones que recogieran el impacto en la operación del modulador. Aquí es donde se nota la importancia de establecer modelos computacionales que se puedan usar para la simulación de comportamiento.

A continuación se procederá a describir el modelado de los distintos mecanismos de error del integrador y se comentará como se han implementado tanto en VHDL como VERILOG. Como se ha comentado anteriormente hay tres modelos de integradores según correspondan a integradores de una, dos o tres ramas. Aquí se comentará el modelo para el integrador de una rama, obviándose el resto, por tener tan sólo varias entradas adicionales

## 3.4.1. Consideraciones generales para el integrador

El modelo del integrador de una rama tiene como entradas aparte de las dos entradas del integrador las necesarias (*Xin1* y *Xin2*) para incluir todos los parámetros que se puedan necesitar variar durante la simulación. Estos parámetros en VERILOG son necesarios convertirlos primero a VARIBLES reales, porque internamente no se puede trabajar como señales digitales. Estos parámetros son:

- la ganancia finita (*A0*),
- los parámetros de ganancia no lineal (*A1*,*A2*,*A3*,*A4*),
- los parámetros de condensador no lineal (*alpha*, *beta*),
- las capacidades parásitas del condensador de muestreo (*cp1*) y del condensador de integración (*cp2*),
- la capacidad de salida (*clo*),
- la desviación en los pesos de los integradores (*sigma*),
- la resistencia en ON de la llave (*Ron*),
- la transconductancia del amplificador (*gm*),
- la intensidad máxima de salida del integrador (*imax*),
- la densidad espectral equivalente de ruido a la entrada (*inpsd)*

Otras entradas necesarias son el Reset, necesario para resetear el circuito y el reloj, que controla que se tome la salida al final de la fase de integración o de muestreo.

El integrador además tiene entradas y salidas adicionales que son necesarias para el cálculo del error de establecimiento o settling, esto será explicado a continuación y son las siguientes

- Las tensiones almacenadas en los condensadores de muestreo del integrador siguiente de la cadena (X*vcn1*,*Xvcn2*,*X vcn3*)
- Como salida, la tensión almacenada en el condensador de muestreo (X*vc1*) necesaria para el integrador anterior de la cadena si lo hubiera.

Como GENERICS en VHDL o parámetros en VERILOG necesita

- (*INT_TYPE*) que sirve para que se tenga en cuenta en el cálculo el error de establecimiento
- El condensador de integración (*C2*)
- El peso de integración (*Peso1*)
- Las limitaciones positiva (*osp*) y negativa (*osn*) en la tensión de salida
- El periodo de muestreo (*Periodo*), necesario para el cálculo del ruido térmico como se vio en las ecuaciones del capítulo anterior

- El valor de los condensadores de muestreo del integrador siguiente de la cadena (*Cn1, Cn2, Cn3*). Estos últimos, también son necesarios para el cálculo del error de establecimiento

Esto se puede ver en la Figura 3.39, correspondiente a la entidad VHDL del integrador de una sola rama. Los parámetros en VERILOG y los puertos coinciden en nombre con los descritos para VHDL.

```
ENTITY integrador1 IS
        GENERIC (INT_TYPE : BOOLEAN := FALSE;
                 osp : REAL := 1.0e12;osn : REAL := -1.0e12;
                 Peso: REAL := 1.0;C2:REAL := 2.0e-12;
                 Periodo : REAL := 4.0e-7;
                 Cn1,Cn2,Cn3: REAL := 0.0);
        PORT (  Reset,clock: IN STD_LOGIC;
                 Xin1,Xin2, Xvcn1,Xvcn2,Xvcn3: IN REAL;
                 Xout,Xvc1 : OUT REAL:= 0.0;
                 A0,A01,A02,A03,A04: IN REAL;
                 Cp1,Cp2,Clo, sigma,alpha,beta : IN REAL;
                 Ron,gm,INPSD,Imax, dc,ird,srd : IN REAL);
END integrador1;
```

**Figura 3.39. Entidad VHDL correspondiente al integrador de una rama**

## 3.4.2. Modelo de ruido térmico

### 3.4.2.1. Modelo genérico

Debido a la simulación de comportamiento en el dominio del tiempo, solo aquellos fenómenos que presenta cambios con una frecuencia menor o igual a la del reloj pueden ser modelados. Por lo tanto el modelado de mecanismos naturales continuos es posible solo cuando su influencia es circunscrita a cada ciclo de reloj individual, o en otras palabras, esta influencia puede ser claramente particionada.

Esto no ocurre con el ruido térmico debido a lo siguiente [28]:

- Primero es un fenómeno de naturaleza continua que varía aleatoriamente. Este problema es parcialmente solucionable porque aunque la base de operación de los ordenadores está en algoritmos deterministas, es posible construir rutinas que generan números aleatorios con muy baja correlación.

- Segundo su frecuencia de corte está generalmente por encima de la frecuencia de muestreo. Esta es la principal dificultad. La reducción del paso interno de un algoritmo de cómputo para adaptar la velocidad de cambio de las señales, está en los simuladores numéricos. Por lo tanto, la principal ventaja de los simuladores de comportamiento estaría definitivamente perdida.

En el caso particular de los convertidores de sobremuestreo, existe una aproximación al problema más cerca de la simulación de comportamiento.

Como la frecuencia de reloj es algunas veces más grande que la frecuencia de las señales que están siendo procesadas, la modulación del ruido blanco, permite un ruido equivalente que en las frecuencias de interés, presenta una densidad espectral constante.

La modulación de este ruido, es sólo efectiva para frecuencias cercanas a la frecuencia de muestreo. Tomando este hecho, el ruido térmico equivalente de entrada del modulador, puede ser precalculado e insertado a la tasa de reloj usando un generador de números aleatorios. Por lo tanto, la densidad espectral de potencia que se obtiene es igual a aquella del ruido térmico modulado en la región de bajas frecuencias.

Teniendo en cuenta las expresiones obtenidas para la densidad del ruido térmico en el capítulo anterior, la densidad equivalente de ruido que tendremos será:

$$S_{eq} = S_{f_{1d}} + S_{f_1} + S_{OP} + S_2 \tag{3.2}$$

Por lo tanto en cada periodo, el voltaje extra almacenado en los condensadores de muestreo que representa adecuadamente el ruido térmico equivalente del modulador es dado por

$$d = \sqrt{12\, f_S S_{eq}} \tag{3.3}$$

Donde $d$ representa un número aleatorio en el rango $(-d/2,\ d/2)$ y ha sido aplicado que la potencia de una señal con densidad de probabilidad constante es, $d^2/12$.

En la Figura 3.40se muestra el diagrama de flujo correspondiente

### 3.4.2.2. Modelo en VHDL y VERILOG

Al ser la densidad espectral de potencia del ruido constante, esta sólo hace falta que se calcule una sola vez durante la simulación. Sin embargo el número aleatorio debe generarse una vez cada semiperiodo de muestreo, para que se le pueda sumar a las tensiones de entrada un número distinto cada vez. En la Figura 3.41 se puede ver el modelado del ruido en VHDL. En VERILOG, se obviará por ser semejante.

## 3.4.3. Modelo de error de establecimiento

Las ecuaciones obtenidas en el capítulo anterior, para el error de establecimiento son para el modelo de un solo polo del integrador. La existencia de un solo polo es sólo una aproximación pero que es válida para amplificadores con un margen de fase mayor de 60 grados, esto es cuando el segundo polo está lo suficientemente lejos del primero.

En la Figura 3.42 se muestra como sería un diagrama de flujo para la programación del error de establecimiento, de acuerdo a las ecuaciones del capítulo anterior. Se presenta el caso del settling de la fase de integración. El de la fase de muestreo sería similar, pero cambiando la *i* de integración por la *s* de muestreo.

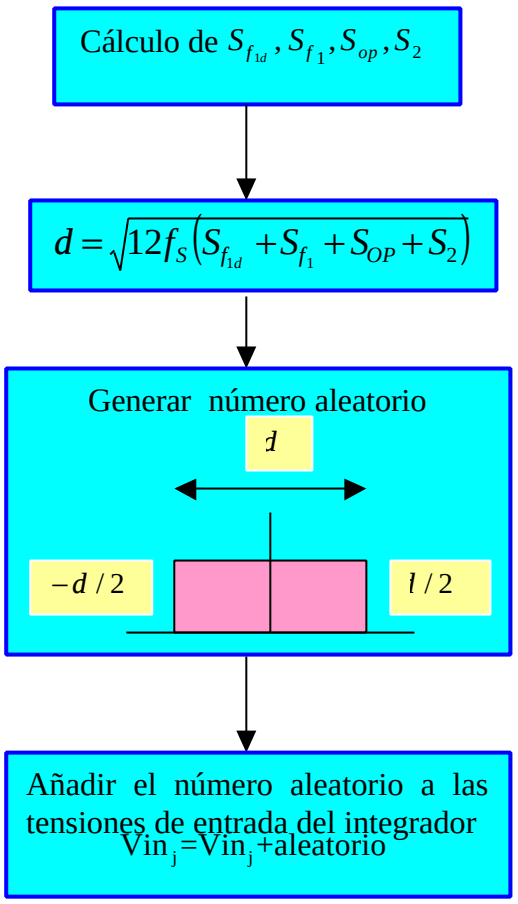



**Figura 3.40. Diagrama de flujo de ruido térmico**

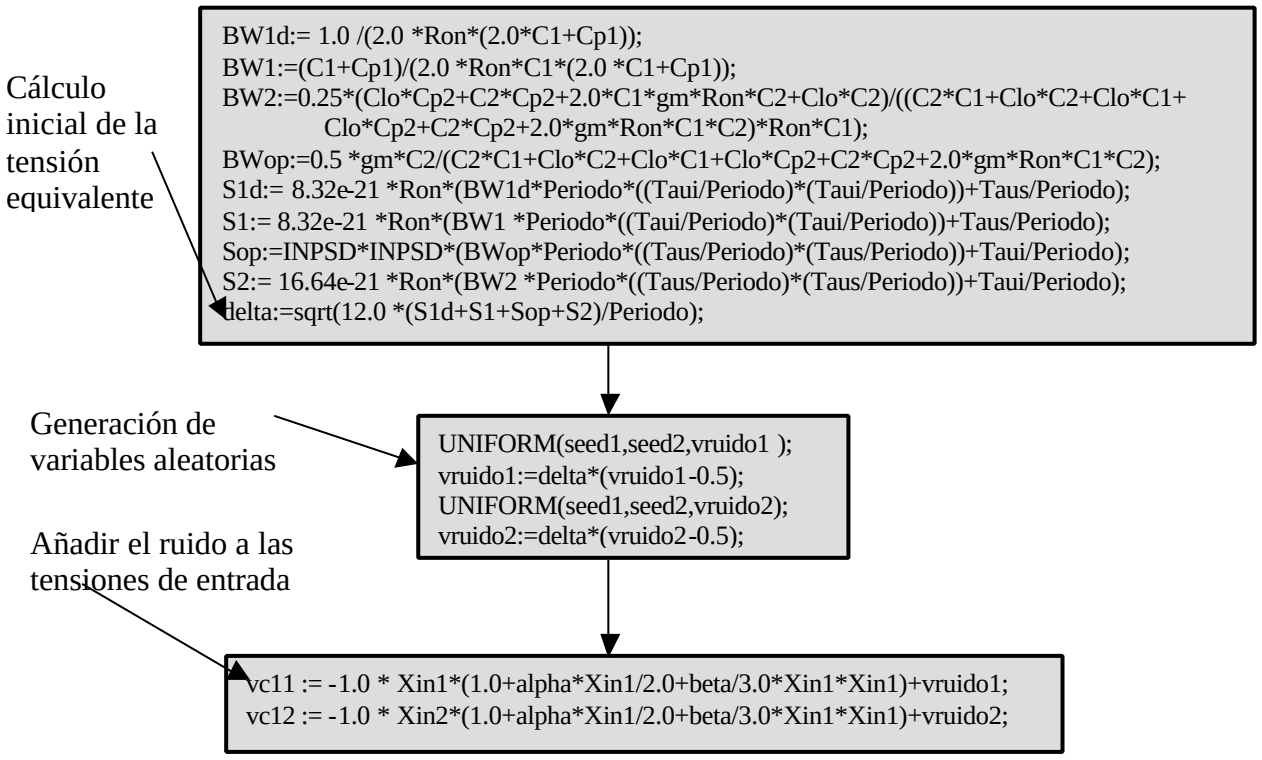



**Figura 3.41. Modelo de ruido térmico en VHDL**

La dificultad de modelar este error, radica que en la ecuación (2.12), hace referencia a las tensiones almacenadas en los condensadores de muestreo y al valor de dichas capacidades del integrador siguiente de la cadena como se puede apreciar en la Figura 2.13. Esto tiene como consecuencia que se necesiten dar como entradas al integrador las tensiones de los condensadores de muestreo del siguiente integrador. Estas tensiones tienen que ir en un lazo de realimentación como se puede ver en la Figura 3.43.

El modelo de este error en VHDL se puede ver en la Figura 3.44. La única diferencia con VERILOG es que en las tensiones de los condensadores de muestreo, habría que transformarlas antes a señales reales de la siguiente manera:

Xvcn1-> $bitstoreal(Xvcn1)

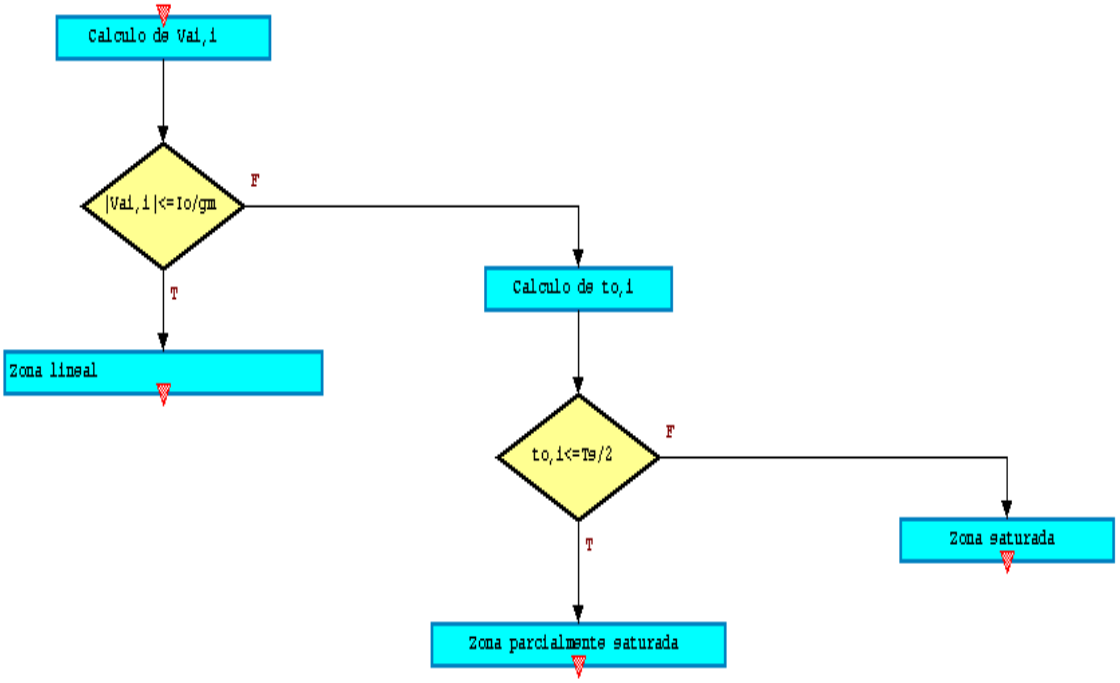



**Figura 3.42. Diagrama de flujo del calculo del error de establecimiento para la fase de integración**

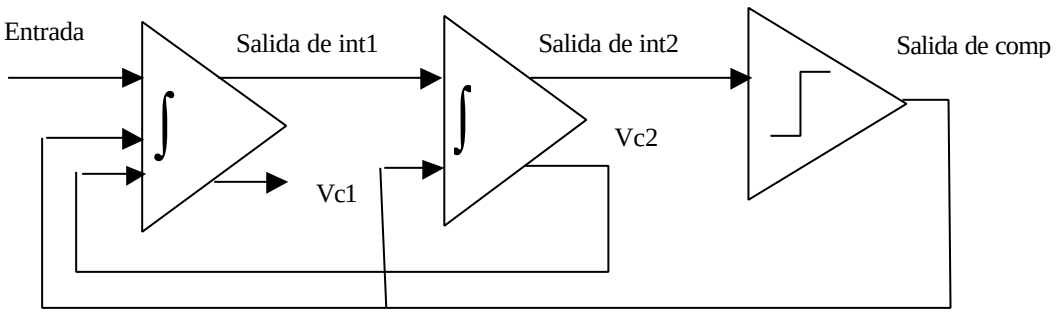



**Figura 3.43. Esquema de conexiones necesario para el error de establecimiento**

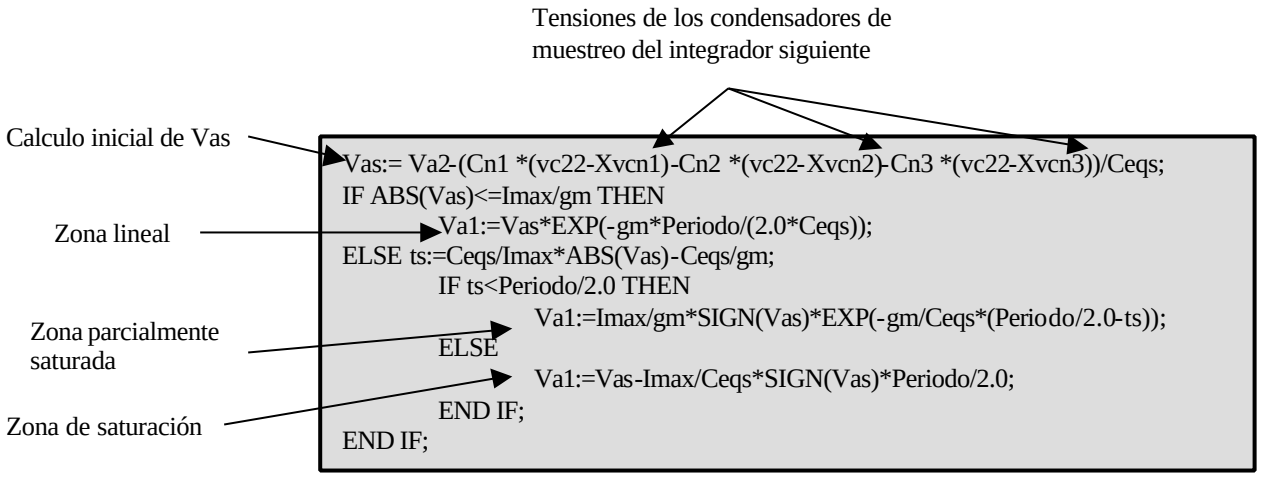



**Figura 3.44. Modelo de error de settling en VHDL**

## 3.4.4. Modelo de desviación en los pesos de los integradores

La desviación en los pesos de los integradores o mismatch, es un error que vendrá dado por las condiciones de fabricación del chip, por lo tanto no es un error que varíe durante el transcurso de la simulación como el caso del ruido térmico sino que este no cambiará durante el transcurso de la simulación.

Para modelarlo se toma una variable aleatoria de distribución gaussiana con media, el peso de integración de la rama y varianza sigma. El resultado será el nuevo peso de integración de la rama, como se puede ver en la Figura 3.45. Ese peso de integración será el nuevo que habrá que usar en las demás ecuaciones.

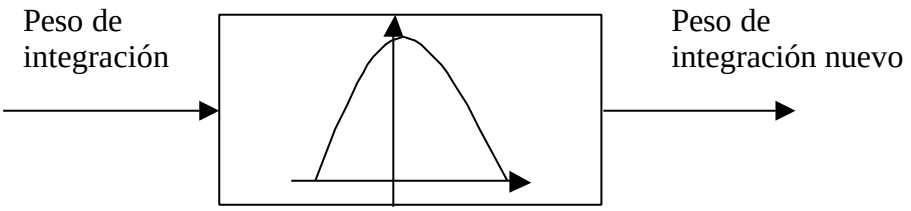



**Figura 3.45. Modelo de desviación en el peso de integración o mismatch**

## 3.4.5. Modelado de errores no lineales

### 3.4.5.1. Capacidad no lineal

Idealmente la ecuación en diferencias finitas que representa el comportamiento de in integrador SC es

$$V_{o,n}C_0 = (V_{1,n-1} - V_{2,n})C_1 + V_{o,n-1}C_0 \tag{3.4}$$

Se considerarán que las capacidades están modeladas de la siguiente forma:

$$C_i(v_i) = C_i^0(1 + a\,v_i + b\,v_i^2 + ...) \tag{3.5}$$

El voltaje de salida quedaría

$$V_{o,n} = V_{o,n-1} + g_1\left[V_{1,n-1}\left(1 + \frac{1}{2}V_{1,n-1} + \frac{1}{3}V_{1,n-1}^2\right) - V_{2,n}\left(1 + \frac{1}{2}V_{2,n} + \frac{1}{3}V_{2,n}^2\right)\right] \tag{3.6}$$

En esta ecuación no se ha introducido la dependencia no lineal de C2, porque no afecta prácticamente a los resultados como se verá en el capítulo 4.

#### 3.4.5.2. Ganancia finita y no lineal

Se debe usar un procedimiento numérico para simular la dependencia de la ganancia en lazo abierto del amplificador con su salida. Considerando un modelo de ganancia finita, la operación del amplificador puede ser aproximada por:

$$V_{o,n} = \frac{A_v \cdot g_i}{A_v + 1 + g_i}\left(V_{1,n-1} - V_{2,n}\right) + \frac{A_v + 1}{A_v + 1 + g_i}V_{o,n-1} \tag{3.7}$$

Donde la ganancia es

$$A_v = A_0\left(1 + g_1 V_{o,n} + g_2 V_{o,n}^2 + ...\right) \tag{3.8}$$

Iterando estas expresiones y las del error anterior llegamos rápidamente al valor final en unos pocos pasos como se puede ver en el diagrama de flujo de la Figura 3.46.

Dentro del bucle se tiene que incluir el efecto de establecimiento incompleto del voltaje del condensador de entrada. Una resistencia en ON no nula de los interruptores analógicos puede llevar a una constante RC que es excesiva en comparación a la duración de la fase de muestreo. Al final de esta fase, el voltaje almacenado en el condensador Ci es dado por

$$V_i^{'} = V_i\left[1 + \exp\left(-\frac{Ts}{2RC}\right)\right] \tag{3.9}$$

A continuación se muestra en la Figura 3.47 cómo se modelaría estos errores en VHDL. En VERILOG, se obviará por ser semejante, excepto en el cambio de las tensiones de entrada Xin1 y Xin2 por $bitstoreal(Xin1) y $bitstoreal(Xin2) necesario para transformar las señales a números reales.

### 3.4.6. Modelo completo del integrador

La Figura 3.48 muestra un diagrama de flujo de las operaciones en el modelo del integrador completo. El diagrama de flujo tiene dos ramas correspondientes a las dos fases de reloj. El código VHDL y VERILOG completo y comentado se puede ver en el apéndice E.

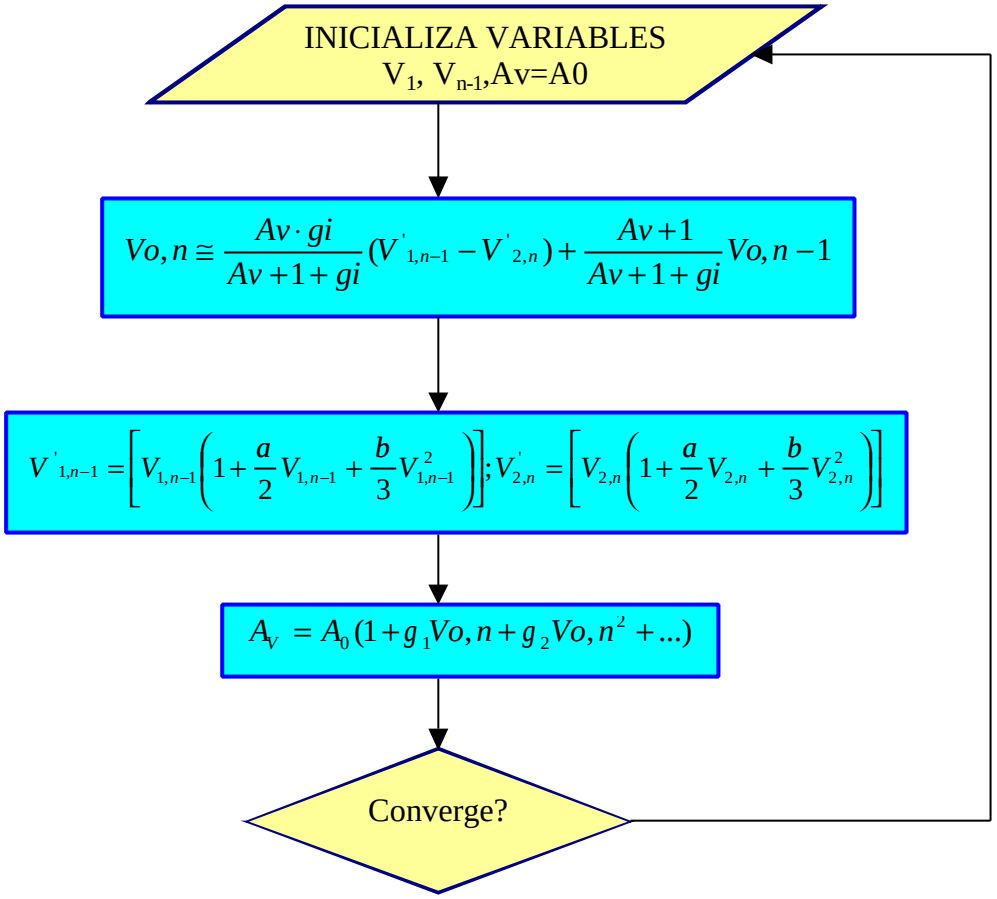



**Figura 3.46. Diagrama de flujo en presencia de no linealidades**

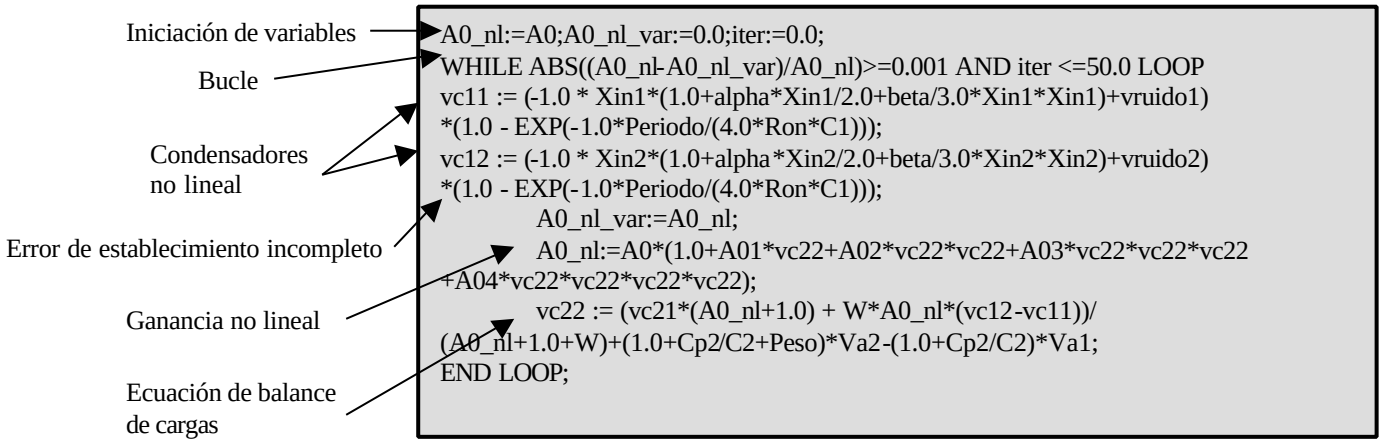



**Figura 3.47. Modelo de errores no lineales en VHDL**

Durante la fase de muestreo, el valor final de los voltajes de la entrada de los condensadores se almacena teniendo en cuenta el valor de los mismos y la resistencia en ON de los interruptores.

Durante la fase de integración, el ruido térmico equivalente de entrada se calcula y se añade al voltaje muestreado. En un proceso iterativo se comienza calculando el voltaje de salida del integrador incluyendo los efectos de ganancia finita y no lineal del amplificador operacional, los condensadores no lineales, la respuesta transitoria y la limitación del rango de salida. La convergencia de este proceso se produce después de dos o tres iteraciones

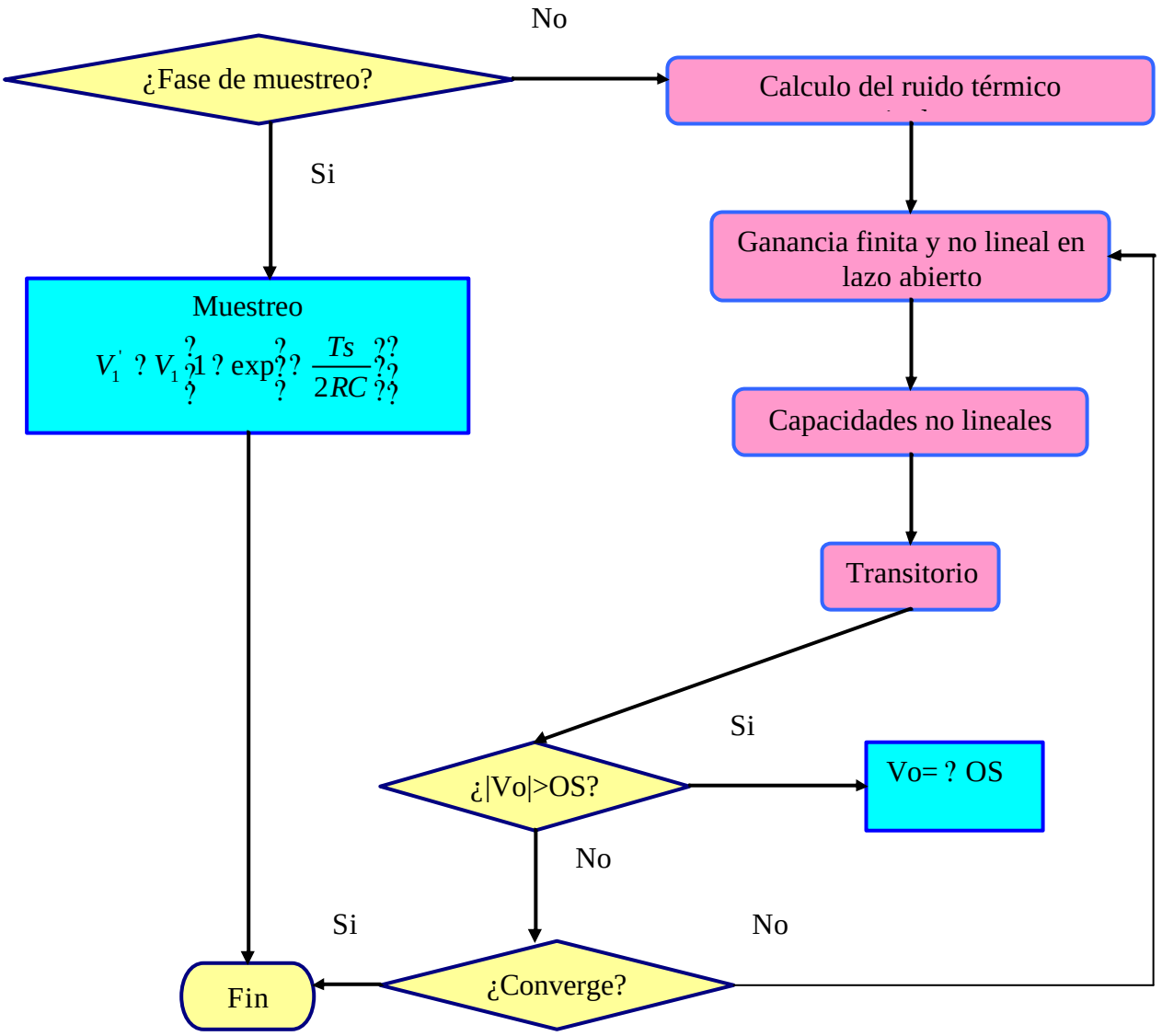



**Figura 3.48. Modelo completo del integrador**

# 3.5. Comparador

## 3.5.1. Modelo genérico

El modelo de la Figura 3.49 presenta tres zonas bien diferenciadas:

- En la primera de ellas, en respuesta a una entrada $V_i$, el valor de salida es una función del signo de $V_i$, proveniente de $|V_i-V_{off}|>V_{his}$, donde $V_{off}$ y $V_{his}$ representa el offset y la histéresis del comparador respectivamente.

- En la otra la salida se determina aleatoriamente en el modelo de la dinámica de histéresis.

- En la última simplemente no cambia por ser el modelo determinista.

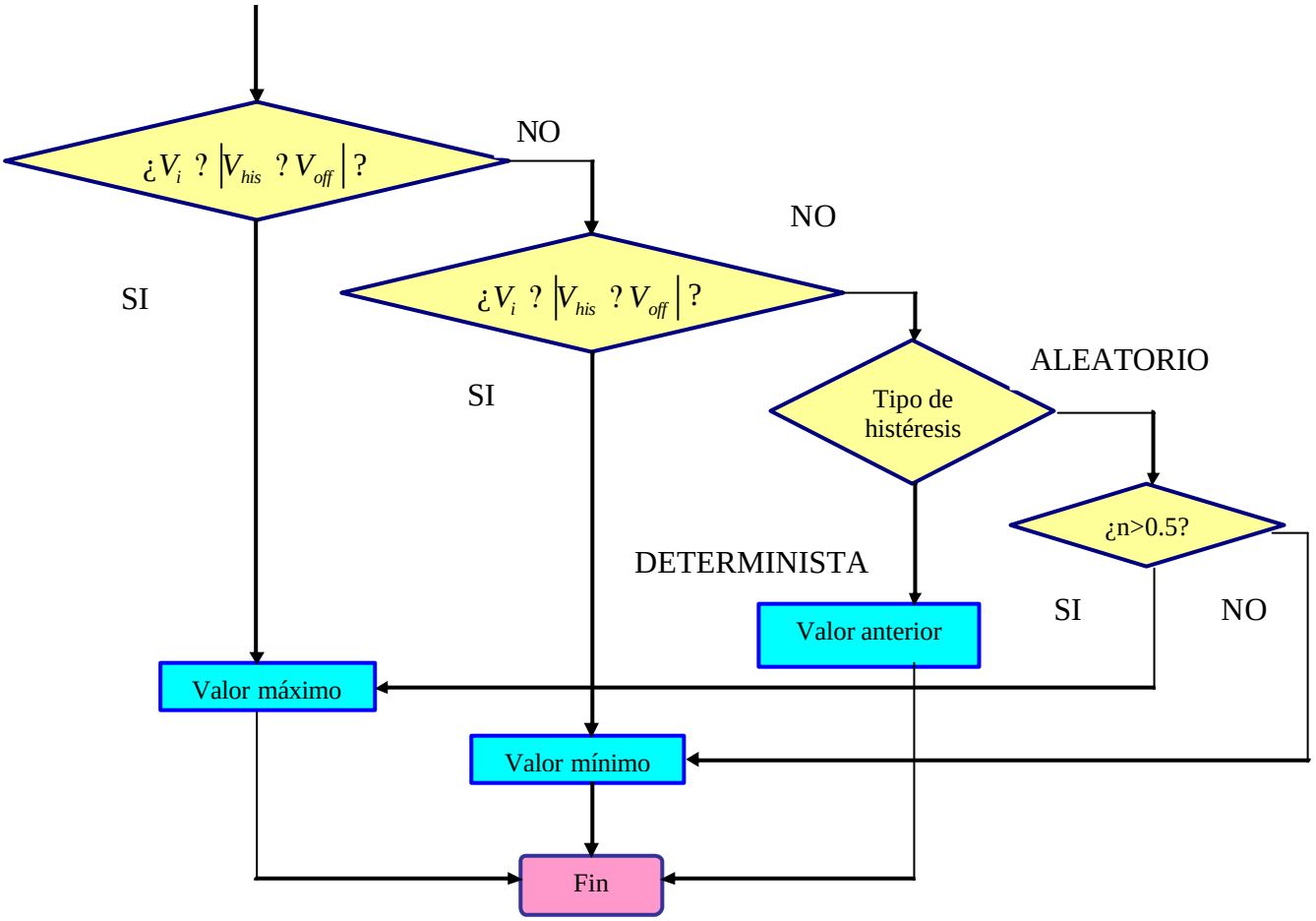



**Figura 3.49. Diagrama de flujo del modelo del comparador**

## 3.5.2. Modelo en VHDL y VERILOG

El modelo de comparador tiene como entradas aparte de la tensión de entrada, los parámetros que se pueden necesitar variar durante la simulación que son: la tensión de offset (*Voff*) y la tensión de histéresis (*Vhis*), ambas en voltios. Otras entradas necesarias son el Reset, que sirve para resetear el bloque a su estado inicial que es a la tensión máxima, y el reloj, que sirve para que el comparador solo cambie su salida con los flancos de bajada del reloj. Esto se puede ver en la Figura 3.50.

Como GENERICS en VHDL o parámetros en VERILOG necesita el valor máximo y el valor mínimo de salida y el tipo de histéresis que tiene el comparador, 0 determinista o 1 aleatoria.

```
ENTITY comparador IS
        GENERIC (Vmax: REAL := 1.5;Vmin: REAL := -1.5;
                Htype: INTEGER :=0);
        PORT (  Reset,Clock : IN STD_LOGIC
                Xin: IN REAL;
                Xout: OUT REAL:= Vmax;
                Voff,Vhis : IN REAL);
END comparador;
```

**Figura 3.50. Entidad del comparador en VHDL**

Como hemos visto en el modelo general, el comparador tiene tres zonas diferenciadas. El modelo de implementación en VHDL se puede ver en la Figura 3.51.

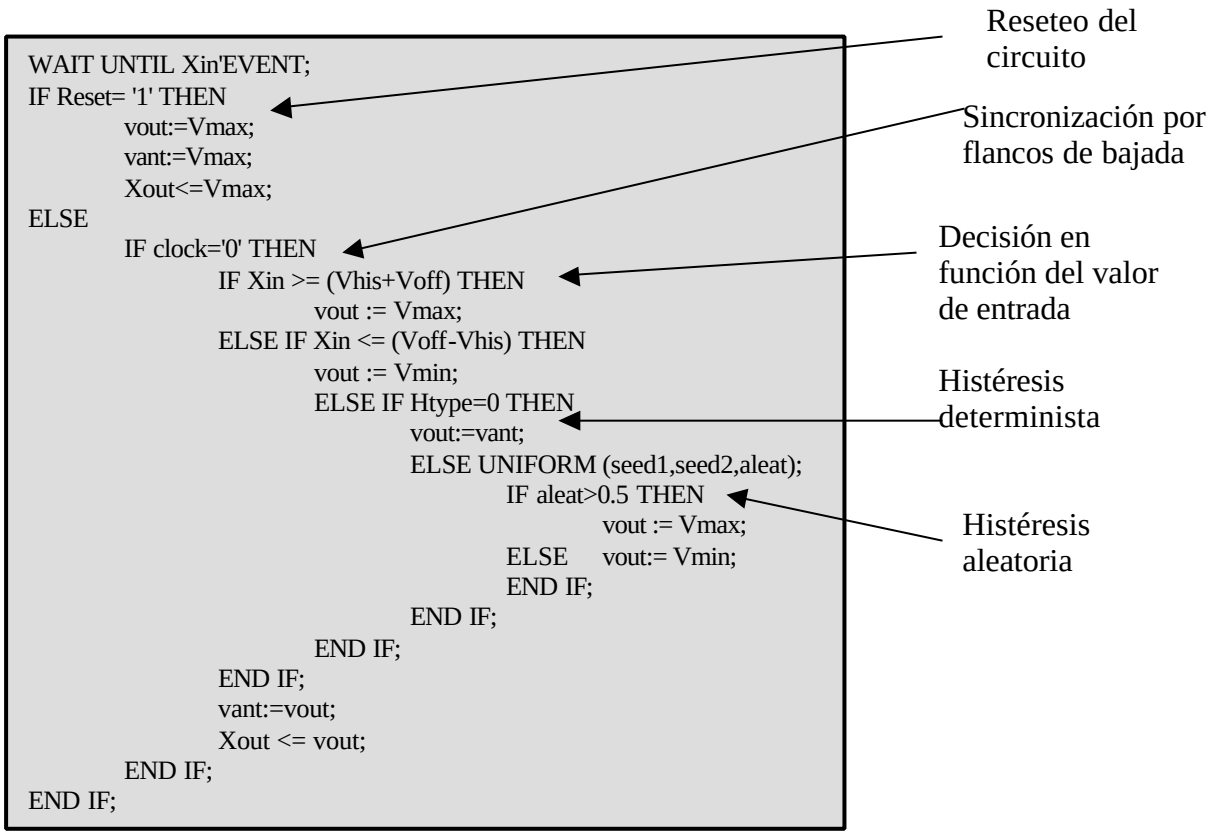



**Figura 3.51. Implementación del comparador en VHDL**

La única diferencia con el código de VERILOG es la necesidad de transformar la señal de salida y de entrada de señales digitales a reales y viceversa como puede apreciarse en la Figura 3.52

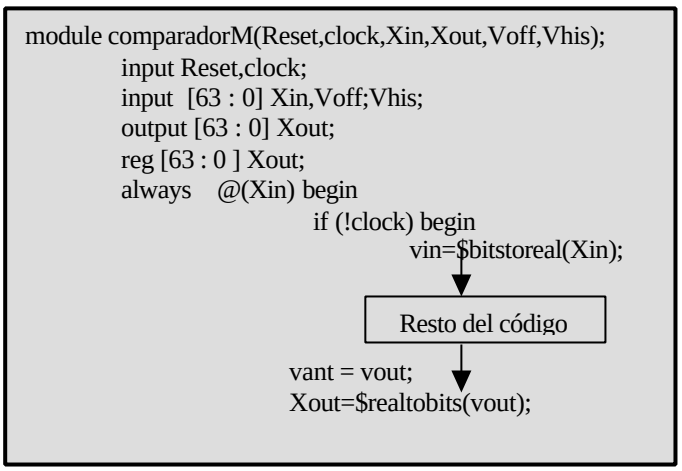



**Figura 3.52. Diferencia de VERILOG con VHDL en el modelo de comparador**

# 3.6. Cuantizador y convertidor D/A

## 3.6.1. Modelo genérico

El modelo genérico implementado en VHDL y VERILOG corresponde con el de la Figura 2.28. Las ecuaciones correspondientes han sido explicadas en el capítulo anterior.

## 3.6.2. Modelo en VHDL y VERILOG

El modelo de convertidor digital analógico y el del cuantizador tiene como entradas aparte de la tensión de entrada, los parámetros que se pueden necesitar variar durante la simulación que son el error de offset (*eoff*), el error de ganancia (*egain*) ambos expresados en % de la escala completa y el error de no linealidad (*enl*) expresado en número de LSBs. Otras entradas necesarias son el Reset que sirve para resetear el bloque a su estado inicial que es a la tensión máxima y el reloj que sirve para que el comparador solo cambie su salida con los flancos de bajada del reloj.

Como GENERICS en VHDL o parámetros en VERILOG necesita el número de bits de resolución (*Nbits*), el valor máximo (*Vmax*) y mínimo (*Vmi*n) de tensión digital de salida en el caso del cuantizador y de entrada en el caso del convertidor. Y el valor máximo ($V_{DD}$) y mínimo ($V_{SS}$) de tensión analógica de entrada en el caso de cuantizador y de salida en el caso del DAC. El parámetro de escala completa ($F_{SR}$) que aparece en el modelo genérico se ha sustituido por las tensiones máxima y mínima, habiendo una relación entre ellos que viene dado por

$$F_{SR} = V_{DD} + V_{SS} \tag{3.10}$$

Esto se puede ver en la Figura 3.53 que muestra la entidad para VHDL del convertidor DA. Para el caso del cuantizador, tendría exactamente las mismas entradas, pero el nombre de la entidad cambiaría a *cuantizadorAD*.

```
ENTITY convertidorDA IS
        GENERIC (VDD: REAL := 1.5;VSS : REAL := -1.5;
                Vmax: REAL := 1.5;Vmin: REAL := -1.5;
                Nbits: REAL := 1.0);
        PORT (  Reset : IN STD_LOGIC;
                Xin: IN REAL; Xout: OUT REAL:= VDD;
                Eoff,Egain,Enl: IN REAL);
END convertidorDA;
```

**Figura 3.53. Entidad del convertidor DA en VHDL**

Se procederá a ilustrar el Convertidor DA, solamente considerando que el cuantizador es semejante. La implementación en VHDL, que es la que se va a comentar, tiene una primera parte para reseteo del circuito. Sigue con la inicialización de los parámetros descritos en las ecuaciones (2.29) y en (2.30). Como se puede observar en la Figura 2.28 el circuito comienza por un bloque CDA ideal, seguido de un bloque no lineal, para a continuación sumarle un factor de ganancia y un offset. Todo esto se puede ver en la Figura 3.54. La inicialización de los parámetros, solo al igual que en el caso del ruido en el integrador sólo se necesita hacerse vez.

Al igual que se comentaba para el caso del comparador, en este caso, la única diferencia que existe entre el modelo en VERILOG y el de VHDL es que el primero necesita de la conversión de las señales de entrada digitales a señales reales y de las señales de salida de reales a digitales

# 3.7. Bloques auxiliares

Existen tres bloques auxiliares necesarios para los moduladores Sigma – Delta en cascada. Estos bloques son necesarios para la parte de cancelación lógica del modulador. Son los siguientes: sumador o amplificador, multiplicador y bloque de retraso.

En principio estos bloques se suponen ideales por estar implementados digitalmente, pero en el caso del sumador o amplificador dispone de algunas no linealidades que se le han añadido y que serán explicadas a continuación.

## 3.7.1. Sumador

### 3.7.1.1. Modelo genérico

El modelo genérico de sumador se puede ver en la Figura 3.55:

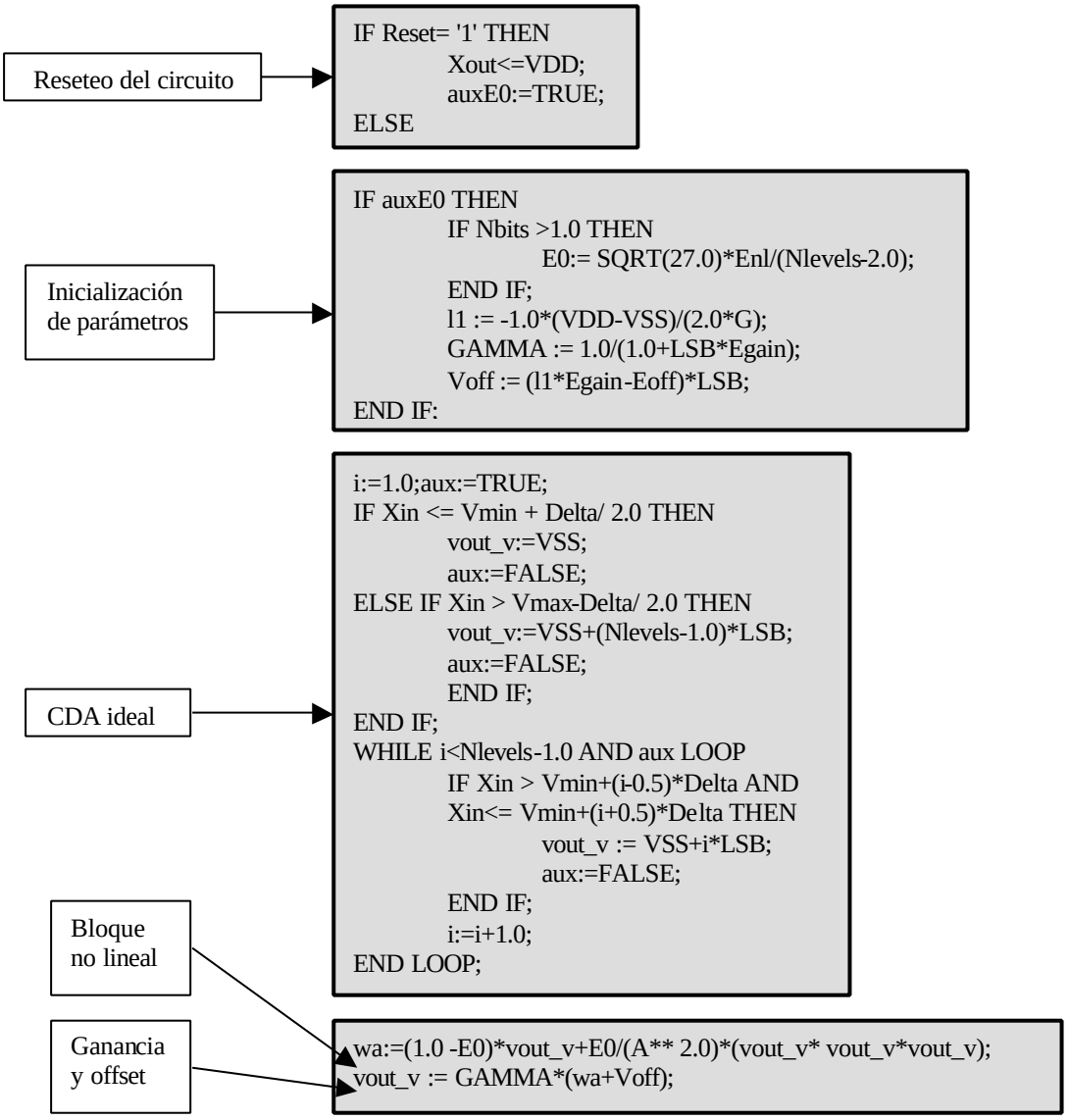



**Figura 3.54. Implementación de un CDA en VHDL**

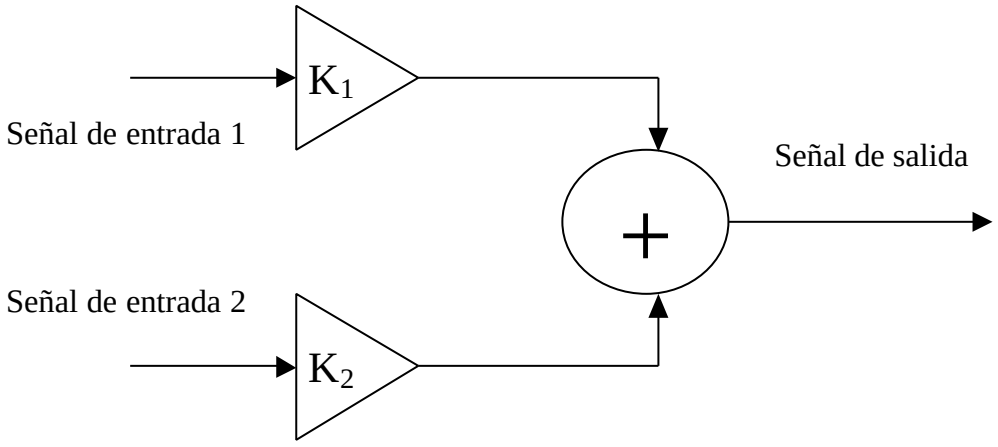



**Figura 3.55. Bloque genérico de un sumador**

El modelo genérico de un sumador no tiene contemplado ninguna no linealidad, sin embargo, tanto en VHDL como en VERILOG se han añadido los siguientes errores:

- Error de ganancia
- Error de ganancia no lineal
- Ruido térmico

Estos errores tienen asociados la siguiente ecuación.

$$Vout = \sum_{j=0}^{N^o ramas} K_j V_j \left(1 + gerr\right) + g_{nl1} V_j + g_{nl2} V_j^2 + ruido \tag{3.11}$$

Al igual que ocurría con el integrador el ruido se modela con una variable aleatoria distribuida uniformemente en el rango [-0.5,0.5], multiplicado por la tensión equivalente que tendría la densidad espectral de potencia del ruido térmico a la salida. Esto se puede ver en la siguiente ecuación

$$ruido = aleat(-0.5,0.5) \cdot \sqrt{12 f_S \cdot onoise^2 / 2} \tag{3.12}$$

donde:

- *fs* es la frecuencia de muestreo
- *onoise* es la densidad espectral de potencia del ruido a la salida

### 3.7.1.2. Modelo en VHDL y VERILOG

En VHDL y VERILOG, como se ha comentado no se puede tener un número de entradas infinitas, por lo que debido a este inconveniente se ha implementado tan solo un sumador de dos ramas y un sumador de una rama, o lo que es lo mismo, un amplificador. Para ilustrar el modelado se explica el caso del sumador de dos ramas, por ser más complejo.

El modelo del sumador tiene como entradas aparte de las tensiones de entrada el error de ganancia (*gerr*) los errores de no linealidad (*gnl1,gnl2*) y la densidad espectral de ruido a la salida (*onoise*). También tiene al igual que en los modelos anteriores la entrada de Reset.

Como GENERICS en VHDL o parámetros en VERILOG tiene:

- los pesos o ganancias de cada una de las ramas (*Peso1* y *Peso2*).
- el periodo de muestreo, necesario para el calculo del ruido térmico equivalente (*Periodo*).
- la limitación en las tensiones de salida positiva y negativa (*osp* y *osn*). Estos dos parámetros, en principio, se consideran ideales, es decir, que no tienen limitación en la tensión de salida.

Esto se puede apreciar en la siguiente figura, que muestra la entidad de un sumador.

```
ENTITY sumador IS
        GENERIC (osp : REAL := 1.0e12;osn : REAL := -1.0e12;
                Peso1: REAL := 1.0;Peso2 : REAL := 1.0;
                Periodo : REAL := 4.0e-7);
        PORT (  Reset: IN STD_LOGIC;
                Xin1,Xin2: IN REAL;
                Xout : OUT REAL:= 0.0;
                gerr,gnl1,gnl2,onoise : IN REAL);
END  sumador;
```

**Figura 3.56. Entidad del sumador en VHDL**

El modelado en VERILOG no se ilustrará por ser semejante a VHDL excepto en las entradas y salidas. El sumador tiene una parte para reseteo del circuito, Sigue con el cálculo de las no linealidades descritas anteriormente. Finalmente, limita el valor de salida en función de los parámetros osp y osn, que para el caso de los moduladores Sigma – Delta permanecerán ideales. La implementación del convertidor completo se puede ver en la Figura 3.57.

## 3.7.2. Multiplicador

Este bloque se ha implementado tanto en VHDL como en VERILOG de forma ideal, cuyo comportamiento genérico puede verse en la Figura 3.58.

Al ser un bloque muy sencillo y no tener ningún error implementado en ambos lenguajes, no tiene ninguna entrada añadida como en los casos anteriores. El modelado en VHDL se puede ver en la Figura 3.59.

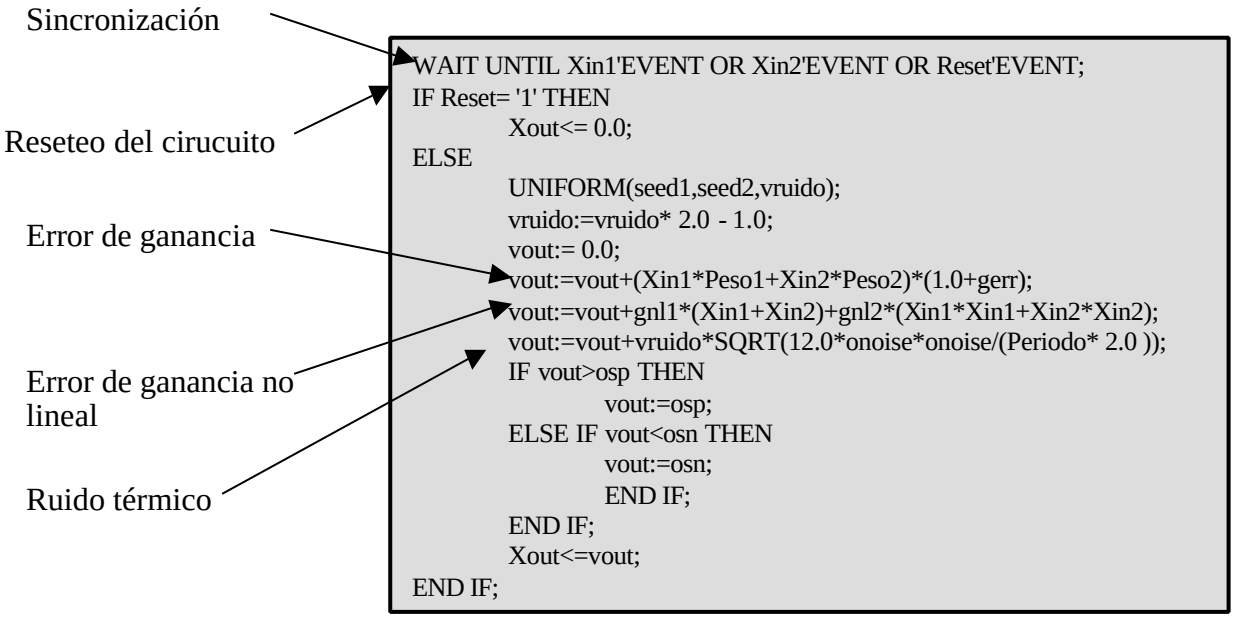



**Figura 3.57. Implementación de un sumador completo en VHDL**

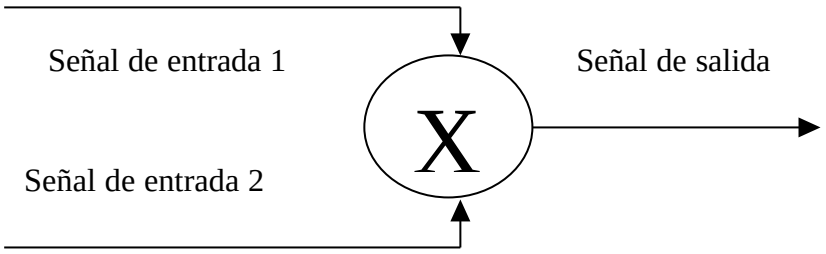



**Figura 3.58. Modelo genérico de un multiplicador**

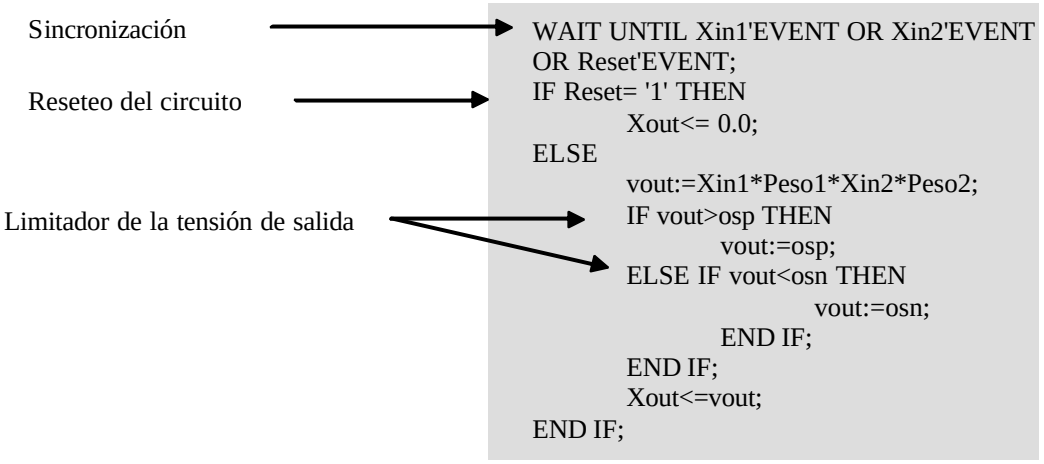



**Figura 3.59. Implementación de un multiplicador en VHDL**

## 3.7.3. Retraso

Este bloque se ha implementado tanto en VHDL como en VERILOG de forma ideal. Al igual que ocurre con el multiplicador no hay que añadir ninguna entrada adicional como parámetro debido a que no se ha implementado ningún error. El comportamiento genérico puede verse en la siguiente figura.

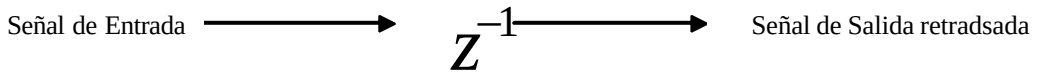



**Figura 3.60. Modelo genérico de un bloque de retraso**

La implementación del cloque de retraso en VHDL se puede ver en la Figura 3.61.

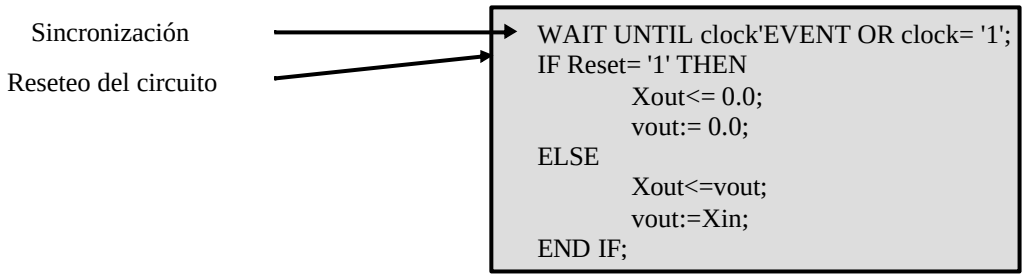



**Figura 3.61. Implmentación de un elemento de retraso den VHDL**

# 3.8. Otras funciones en VHDL

En VHDL se han implementado otra serie de funciones para el cálculo de la SNR y del espectro de potencia de la señal de salida. Estas funciones son las siguientes:

- Norm: Calcula la norma de un vector de números reales de tamaño n, según la fórmula

$$norm = \sqrt{\sum_{n=0}^{N} x^2[n]} \tag{3.13}$$

- Bit_inverso: El algoritmo de bit inverso es necesario para el cálculo de la FFT (Fast Fourier Transform) [60]. Este algoritmo calcula el número equivalente de un número binario de n bits al invertir cada uno de los bits de lugar. En la Figura 3.62 se presenta un ejemplo de este algoritmo para dos bits. En la Figura 3.63 se puede ver un diagrama de flujo de este algoritmo para su implementación en VHDL.

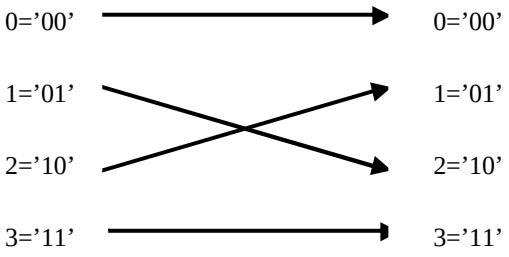



**Figura 3.62. Ejemplo del algoritmo de bit inverso para 2 bits**

- Hanning: Devuelve una ventana de hanning de tamaño *N*, cada muestra se puede calcular como sigue:

$$X(k) = \frac{1}{2}\left[1 - \cos\left(\frac{k+1}{N+1}\right)\right] \rightarrow k = 0,1,2,...,N-1 \tag{3.14}$$

- snr: Devuelve la SNR de la señal de salida

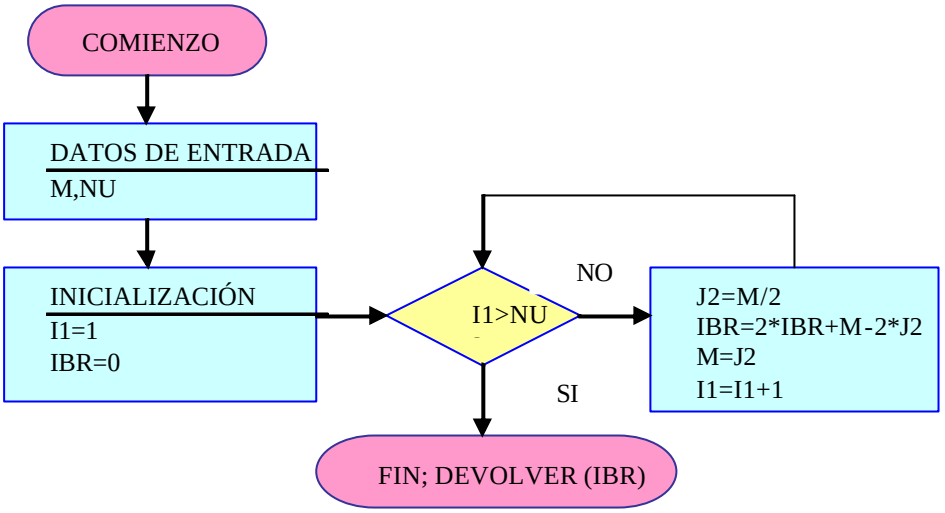



**Figura 3.63. Diagrama de flujo del algoritmo de bit inverso**

- Fft: Calcula la transformada de fourier discreta (DFT) que se calcula de la siguiente manera.

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-jnk} \rightarrow k = 0,1,2,...,N-1 \qquad (3.15)$$

Debido a que el número de multiplicaciones necesarias para realizar esta operación es del orden de $n^2$, siendo n la el vector de datos al que se quiere calcular la DFT, resulta un algoritmo muy costoso en tiempo de CPU. Para ello, se ha implementado el algoritmo de Cooley – Tukey [60]. Este algoritmo consigue reducir el número de multiplicaciones al orden de n, disminuyendo drásticamente el tiempo de CPU.

El diagrama de flujo de este algoritmo para su implementación en VHDL se puede ver en la Figura 3.64.

- frecuencia: Devuelve un vector de tamaño n/2, siendo n el número de muestras, en el que cada elemento coincide con una frecuenta equiespaciada según la frecuencia de muestreo (fs). Cada muestra se puede calcular como sigue:

$$X(k) = k \cdot fs / N \rightarrow k = 0,1,2,..., N/2-1 \qquad (3.16)$$

- espectro: Devuelve un vector de tamaño n/2, siendo n el número de muestras, en el que cada elemento es la potencia de la señal de salida a la frecuencia dada por el vector devuelto por la función frecuencia

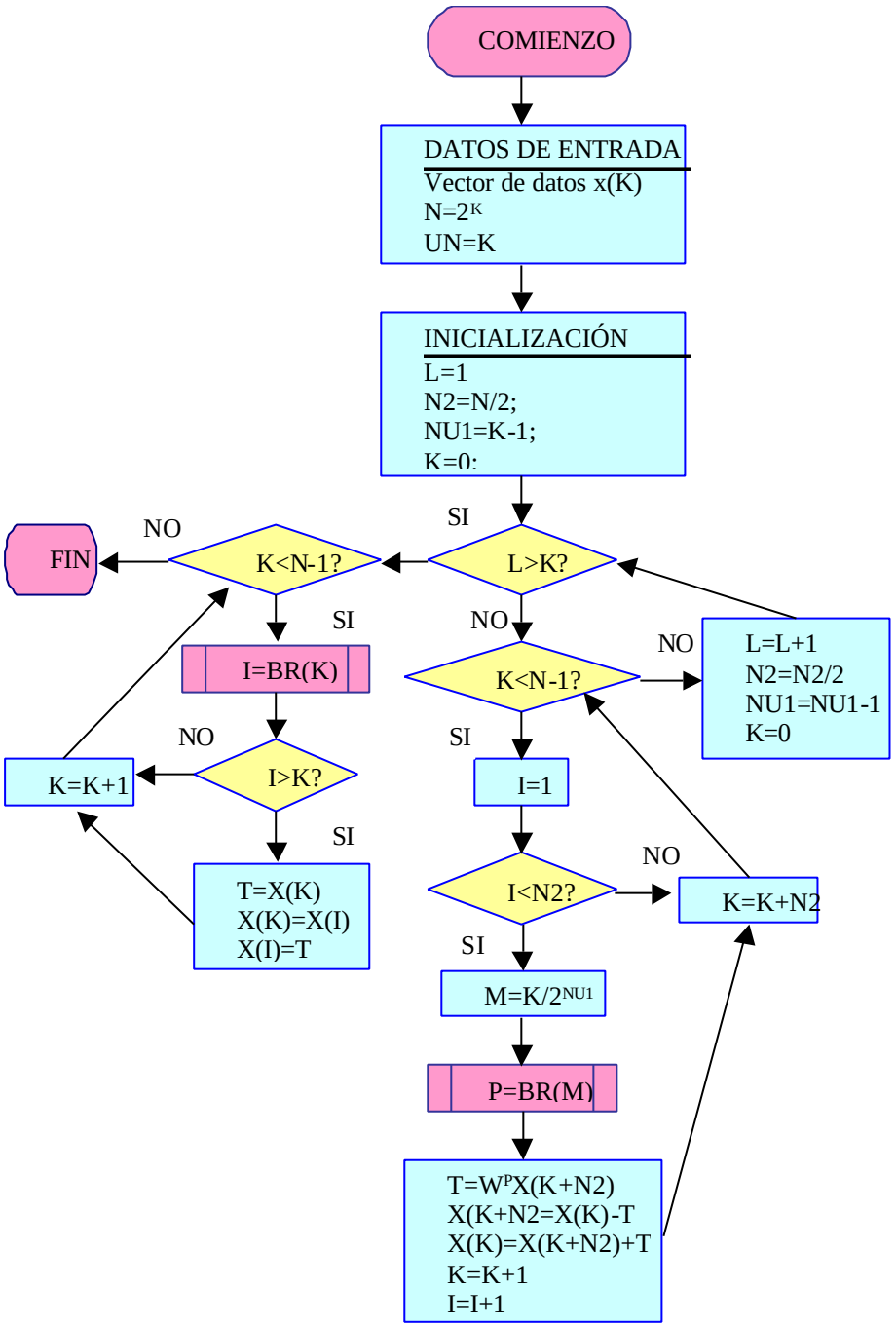



**Figura 3.64. Diagrama de flujo del algoritmo deFFT**

# CAPÍTULO 4 Descripción de la herramienta

## 4.1. Introducción

En este capítulo se procederá a explicar la herramienta desarrollada en este proyecto. Hasta ahora los simuladores existentes para simular estructuras sigma – delta no habían implementado éstos en un lenguaje de descripción de Hardware. Este proyecto pretende crear un simulador para este tipo de circuitos basado en HDL. Las principales características que se pueden encontrar son:

- Precisión elevada: Modelado de comportamiento de las principales no idealidades de los moduladores $\Sigma\Delta - SC$.

- Coste temporal bajo: Como se puede ver en el apartado 4.8 el tiempo de simulación es de 15 segundos para un modulador $\Sigma\Delta - SC$ en cascada 2-1-1 con 65536 muestras de la señal de salida.

- Alta flexibilidad: La herramienta está enfocada especialmente para $\Sigma\Delta - SC$ pero no está dedicada exclusivamente a este tipo de bloques, incluso bloques aislados, pueden ser simulados. Aparte, al estar escrito en un lenguaje de descripción de Hardware, permite la interconexión con otros bloques y otros circuitos, no exclusivos de $\Sigma\Delta - SC$. Esto permite la simulación de grandes circuitos de señal mixta.

- Interfaz gráfica: En el caso de que los usuarios no tengan mucha soltura con el HDL, disponen una interfaz gráfica, que facilita el trabajo.

- Elevado número de análisis: Para VHDL permite un amplio abanico de posibilidades en el procesado de la señal.

- Lenguaje de descripción de arquitecturas: Permite describir un elevado número de tipologías de circuitos, y no está específicamente ligado a los moduladores $\Sigma\Delta$. Lenguaje sencillo para los usuarios sin mucha soltura con HDL.

En la Figura 4.65 se puede ver un diagrama de funcionamiento de la herramienta.

En primer lugar se necesita describir el circuito. Esto se puede hacer de tres formas:

- *Descripción pura mediante instancias de VHDL o VERILOG*: Esta es la forma más dificultosa para el usuario, ya que este tiene que conocer a la perfección cuántos parámetros y cuántas entradas tiene cada bloque básico. En el caso particular de los integradores, tiene que tener especial cuidado en conectar los puertos pertenecientes a las tensiones de los condensadores de muestreo, ya que éstas tienen que conectarse como entradas en el integrador precedente en la cadena. Otra de las desventajas es que el usuario tiene que tener un conocimiento profundo del lenguaje.

- *Descripción mediante un NETLIST*: Esta forma es más sencilla para el usuario y permite la descripción de la topología mediante un netlist. Este netlist sirve como entrada a la herramienta VSIDES, explicada en profundidad en el Apéndice A y que genera un fichero VHDL o VERILOG válido para simular. Tiene como ventaja que el usuario no tiene que tener en cuenta que hay que realimentar las tensiones de los condensadores de muestreo y que no tiene que introducir manualmente todas las no idealidades como entradas. Sin embargo el usuario tiene que aprender una sintaxis nueva, aunque sencilla, lo que es un inconveniente.

- *Descripción mediante esquemáticos*: Esta es la forma más sencilla para el usuario y permite la interconexión mediante símbolos correspondientes a cada uno de los bloques básicos. De esta manera tampoco hay que tener en cuenta la necesidad de realimentar las tensiones de los condenadores de muestreo. Sin embargo estos bloques no son los necesarios para la simulación por lo que es necesario obtener un fichero VHDL o VERILOG válido. Esto se hace con la herramienta *esquematic* que será explicada en profundidad en el apéndice B.

La compilación y la simulación de la herramienta se puede hacer con ADVANCE – MS [61] para el caso de VHDL y de MODELSIM [62] para el caso de VERILOG. La versión que se ha utilizado para los anteriores programas ha sido la v1.3_1.1 en el caso de ADVANCE – MS y la 5.5E en el caso de MODELSIM.

En principio se podría haber utilizado únicamente MODELSIM para simular los moduladores ΣΔ descritos en ambos lenguajes de programación. Sin embargo, la utilización de ADVANCE – MS, se debe permitir para la compatibilidad de los bloques que componen los moduladores con otros cuyo funcionamiento sea necesario describirlo en tiempo continuo.

Finalmente, el procesamiento de la señal se hará de dos formas: cogiendo la secuencia de datos temporales y procesarlo con MATLAB para el caso de VERILOG, debido a que en este lenguaje no se han podido implementar las rutinas para procesamiento de señal. En el caso de ADVANCE – MS, también puede dar la secuencia temporal del nodo de salida, pero también puede procesar directamente los resultados mediante rutinas, que se han explicado en el capítulo anterior, que escriben directamente en un fichero la SNR o el espectro de potencia del nodo de salida. Todo esto se explicará en más detalle en los apartados posteriores.

A continuación se procederá a explicar cada forma de describir los circuitos y para ello se utilizará como ejemplo un modulador single –loop de segundo orden como el que se puede observar en la Figura 1.7.

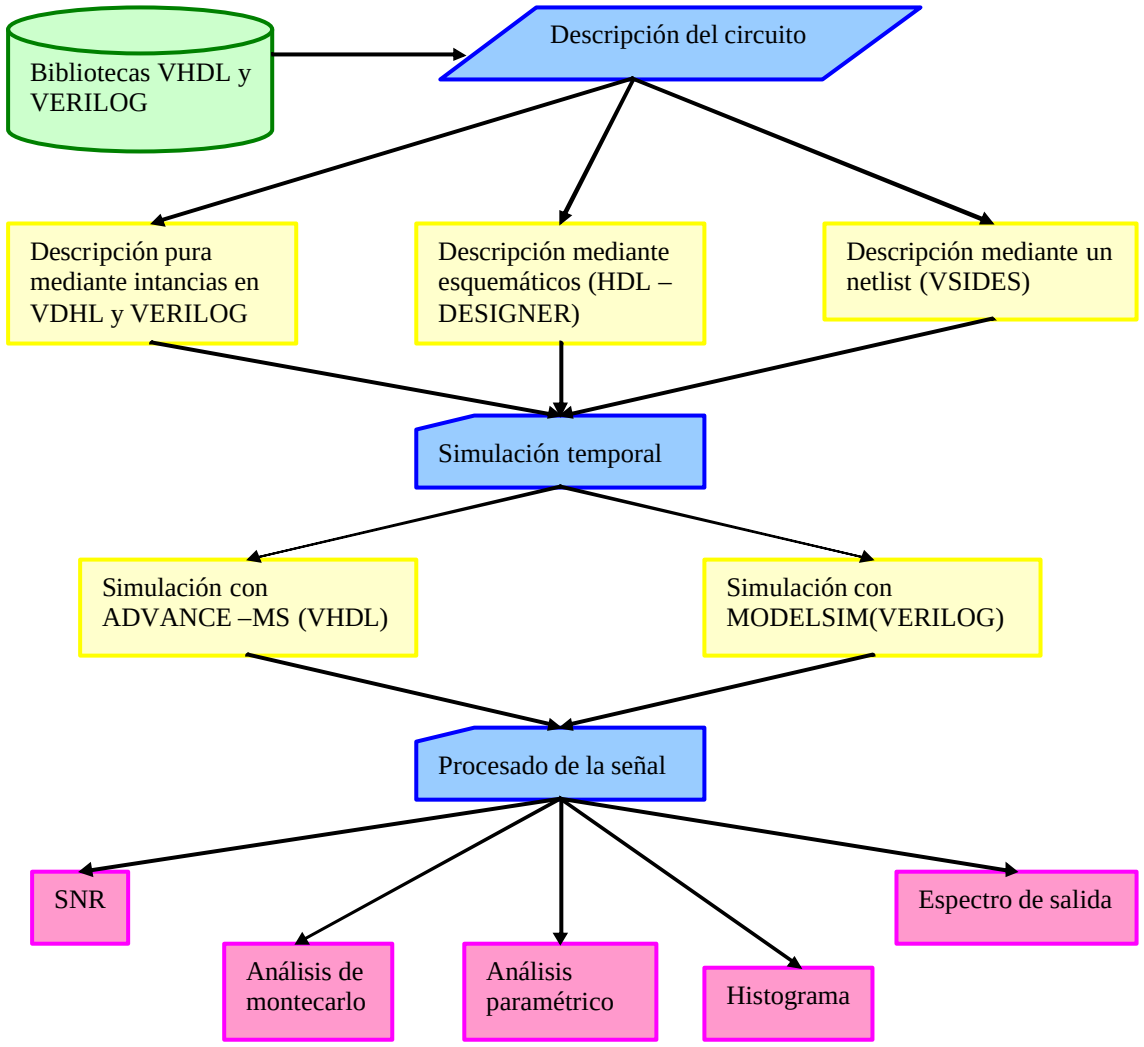



**Figura 4.65. Diagrama de flujo de la herramienta**

# 4.2. Descripción mediante instancias en los lenguajes

## 4.2.1. Descripción en VHDL

En la Figura 4.66, se puede observar el diagrama de flujo genérico de la descripción de un modulador Sigma - Delta en VHDL.

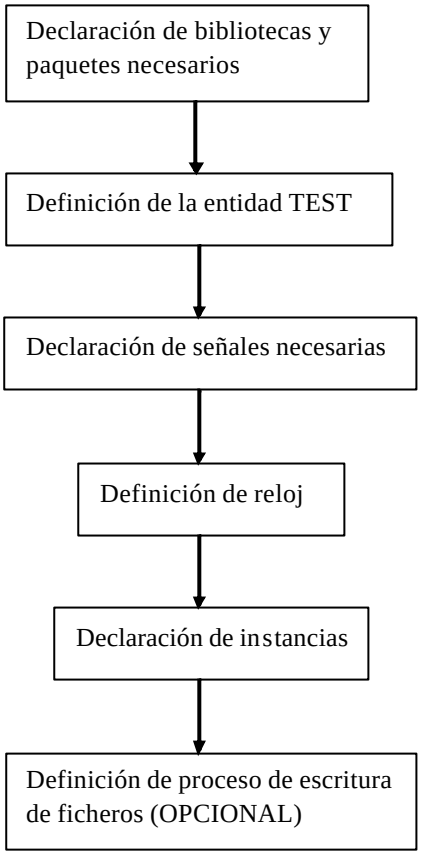



**Figura 4.66. Diagrama de descripción de un modulador Sigma - Delta en VHDL**

El ejemplo del modulador single – loop de segundo orden se puede ver en Figura 4.67.

Se puede observar que en primer lugar habrá que incluir las bibliotecas y paquetes necesarios para la simulación. La biblioteca es la IEEE. Los paquetes estándar que hay que incluir son el paquete matemático MATH_REAL, los paquetes lógicos, STD_LOGIC_ARITH y STD_LOGIC_1164.

En segundo lugar, habrá que incluir los paquetes no estándar siguientes: *encapsula_gauss, encapsula_uniforme, vector_funcions, espectro_function, snr_function, encapsula_copiar* y *encapsula_escr,* estos dos últimos para la escritura de ficheros.

Se definirá una ENTITY (o entidad) cuyo contenido esté vacío para que nos sirva para testar el circuito (Entidad test).

La arquitectura del la entidad comenzará definiendo todas las señales necesarias para el circuito que son:

- Señales correspondientes a los puertos de entrada - salida de los bloques básicos que tendrán que estar inicializadas a cero.

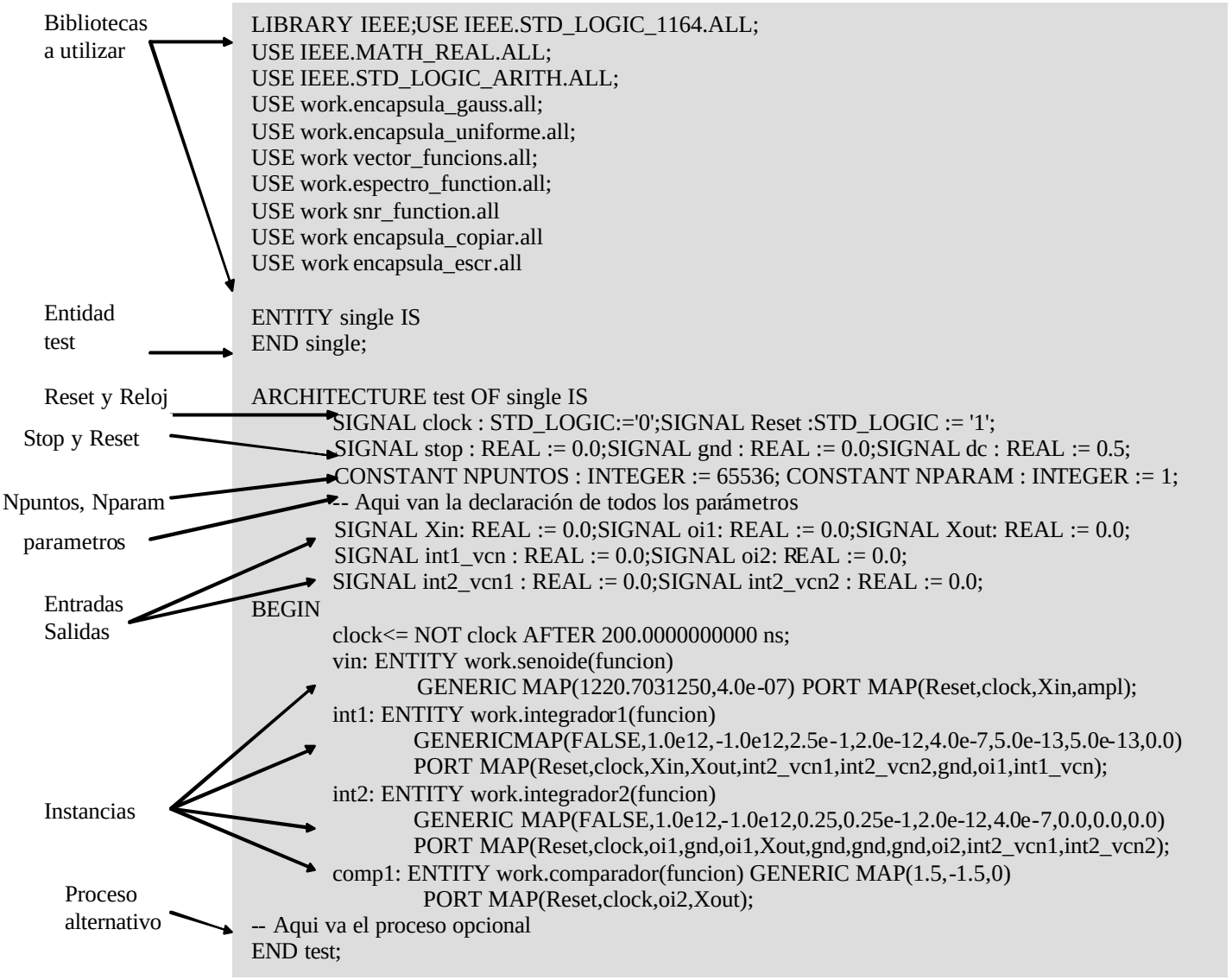



**Figura 4.67. Descripción pura de un modulador single –loop de segundo orden en VHDL**

- Señales correspondientes a los parámetros que se introducen como entradas en los bloques básicos, que tendrán que estar inicializados a su valor correspondiente.

- La señal de *Reset* y la de *Reloj*, que tendrán que estar inicializadas a '1' lógico y a '0' lógico respectivamente.

- La constante *NPUNTOS*, que tomará el valor del número de muestras necesarias para la simulación y la constante *NPARAM* que tomará el valor por defecto de 1. Estos parámetros sólo son necesarios en el proceso opcional que se explicará a continuación.

- La señal *stop*, necesaria únicamente en el proceso opcional, que sirve para llevar una cuenta del número de muestra y la señal *"gnd"* que tiene que estar inicializada a cero y sirve, para tomar un puerto de entrada como "tierra".

El reloj habrá que variarlo con un periodo igual al periodo de muestreo que queremos para simular el modulador.

A continuación se describirán las instancias correspondientes al modulador. Habrá que tener especial cuidado con la conexión de los puertos correspondientes a las tensiones de los condensadores de muestreo de los integradores.

En este ejemplo las tensiones de los condensadores de muestreo que tienen que introducirse como entradas al primer integrador reciben el nombre de *int2_vcn1* y *int2_vcn2*. Las entradas y salidas, asi como los parámetros necesarios para introducir en cada uno de los bloques básicos se han descrito en el capítulo anterior. El orden preciso se puede ver en el apéndice E.

De manera opcional se podrá definir un proceso para la escritura de datos a un fichero. Este proceso, pone al comienzo el reset a '0' necesario, para que los distintos bloques básicos comiencen a funcionar. Al llegar al número de muestras necesarias para la simulación guarda las muestras en un fichero llamado *result.dat*. Esto se puede ver en la Figura 4.68.

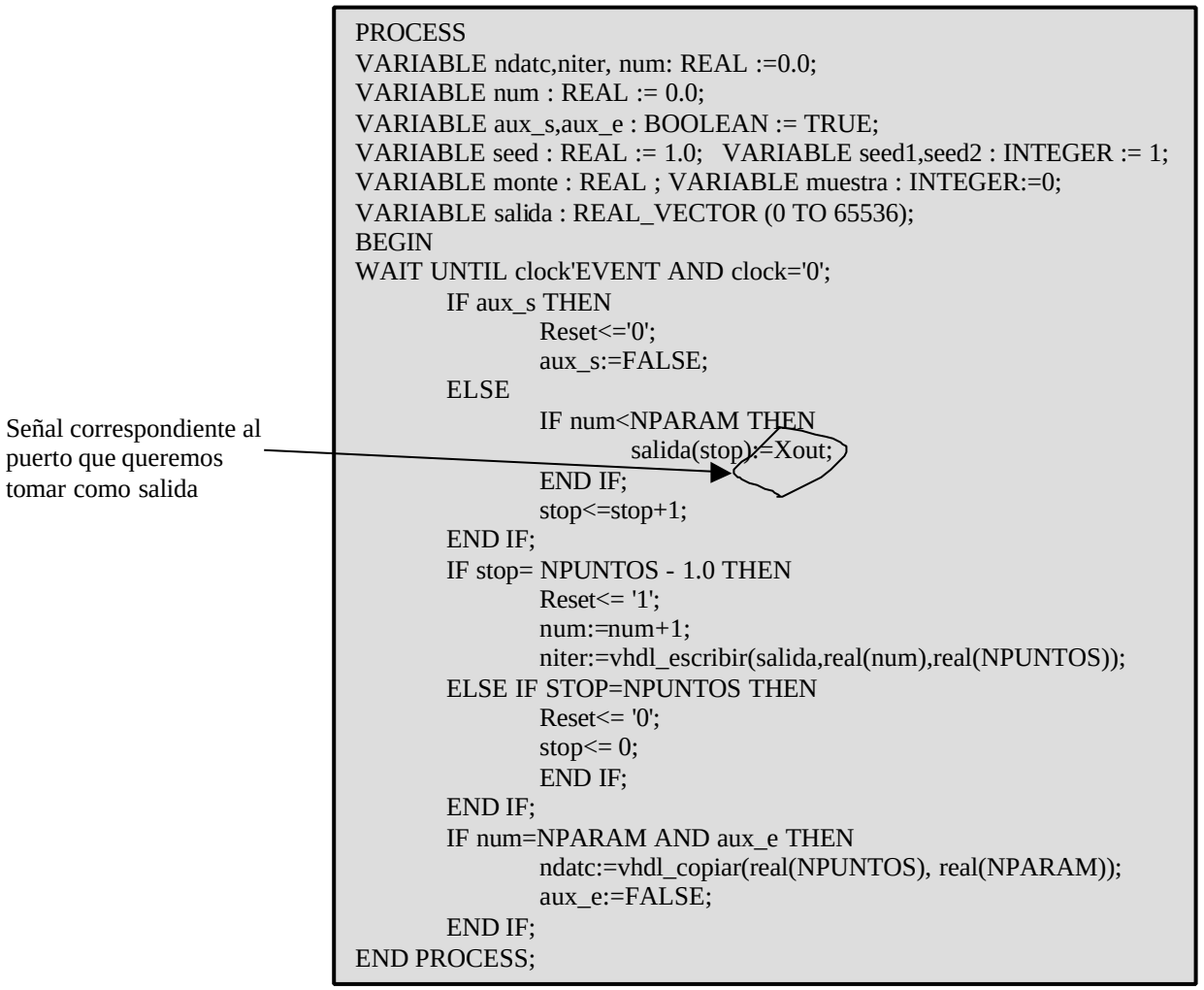



**Figura 4.68. Proceso de escritura a fichero en VHDL**

## 4.2.2. Descripción en VERILOG

En la Figura 4.69 se puede ver el diagrama de flujo genérico de la descripción de un modulador sigma-delta en VERILOG.

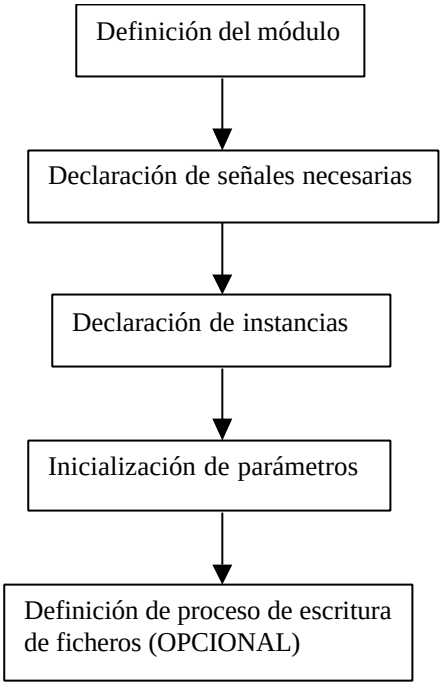



**Figura 4.69. Diagrama de descripción de un modulador Sigma-Delta en VERILOG**

A diferencia de lo que ocurre en VHDL, VERILOG no necesita que se declare ninguna biblioteca para su funcionamiento correcto. Pero antes de arrancar el simulador, el usuario tiene que haber configurado las funciones PLI, para que el simulador las cargue al iniciar la simulación como se explica en el apéndice D.

El ejemplo del modulador single – loop de segundo orden se puede ver en la Figura 4.70.

Lo primero que hay que hacer es definir el modulo de test, que es aquel que no tiene ningún puerto de entrada – salida.

Seguidamente tenemos que declarar todas las señales que nos hacen falta:

- Señales correspondientes a los puertos de entrada - salida de los bloques básicos. Estas señales tendrán se declararán con la palabra reservada *"wire[63,0]"*
- Señales correspondientes a los parámetros que se introducen como entradas en los bloques básicos. Se declaran con la palabra reservada *"reg [63,0]"*. Estos a diferencia de lo que ocurría en VHDL, no pueden inicializarse hasta la sección *initial()*: En esta sección tendrán que inicializarse al valor que queremos que tenga durante el transcurso de la simulación.

- La señal de Reset y la de Reloj, tampoco pueden inicializarse hasta la sección *initail()*:En esta sección tendrán que inicializarse a '1' lógico y a '0' lógico respectivamente.

- La constante NPUNTOS, que tomará el valor del número de muestras necesarias para la simulación y la constante NPARAM que tomará el valor por defecto de 1, se declararán con la palabra reservada *"real"* y tampoco podrán inicializarse hasta la sección *initial.*

A continuación se describirán las instancias correspondientes al modulador. Habrá que tener especial cuidado con la conexión de los puertos correspondientes a las tensiones de los condensadores de muestreo de los integradores.

En la sección *initial* se inicializarán todos los parámetros y constantes, teniendo en cuenta que los parámetros para inicializarlos hay que transformarlos de números reales a señales digitales

Ejemplo: ganancia = $realtobits(1000.0);

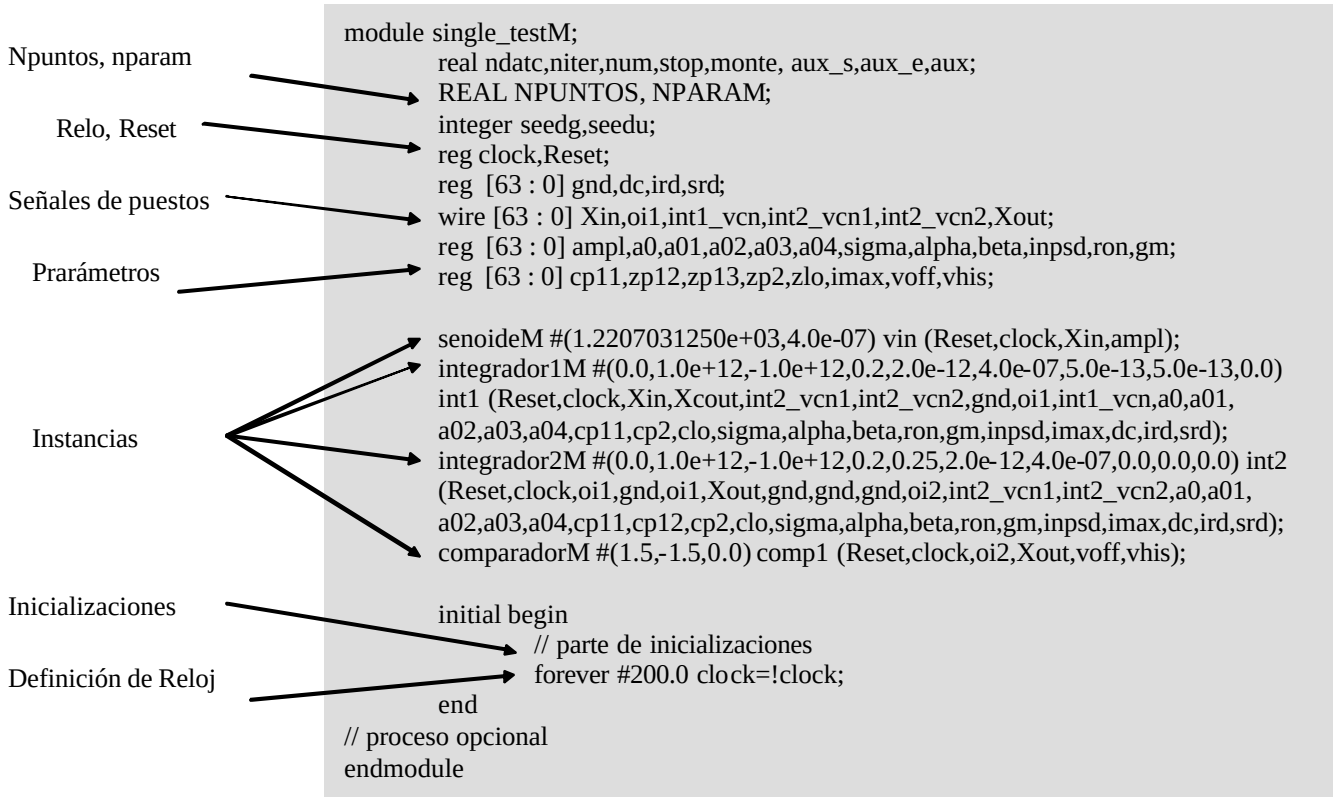



**Figura 4.70. Descripción de un modulador single .loop de segundo orden en VERILOG**

De manera opcional se podrá definir un bloque síncrono "always" sincronizado por los flancos de bajada de reloj, que sirve para la escritura de datos a un fichero. Este bloque "always" realiza exactamente lo mismo que el proceso correspondiente en VHDL. La única diferencia es que los proceso de escritura a fichero se realizan mediante funciones PLI.Esto se puede ver en la Figura 4.71.

```
always @(negedge clock) begin
        if(aux)
                aux=0;
        else if (aux_s) begin
                        Reset = 0; aux_s=0;
                        niter=$escribir($bitstoreal(Xout),num+1.0,stop);
                end else begin
                        if (num<NPARAM)
                                niter=$escribir($bitstoreal(Xout),num+1.0,stop+1.0);
                        stop=stop+1.0;
                end
                if (stop==NPUNTOS) begin
                        Reset = 1; num=num+1.0;
                end else if(stop==NPUNTOS+1.0) begin
                        Reset = 0; stop=0.0;
                        end
                if(num==NPARAM)
                if(aux_e) begin
                        ndatc=$copiar(NPUNTOS,NPARAM);
                        aux_e=0;
                end
        end
```

**Figura 4.71. Proceso síncrono de escritura en fichero en VERILOG**

# 4.3. Descripción mediante netlist

Esta manera de describir una estructura $\Sigma\Delta$, necesita del aprendizaje de una sintaxis propia descrita en profundidad en el apéndice A. La herramienta VSIDES transforma este fichero con el netlist de la estructura a un fichero VHDL o VERILOG válido para la simulación. El fichero con el netlist tiene que tener obligatoriamente extensión *.par*. Para ilustrar el proceso completo supondremos que vamos a escribir el netlist de un modulador $\Sigma\Delta$ de segundo orden llamado *single1.par*. Para obtener el fichero en VHDL o VERILOG hay que utilizar

*% vsides nombre_fichero(single1.par) –opción (vhd o v)*

La opción será vhd o v según queramos el archivo resultante en lenguaje VHDL o VERILOG.

En la Figura 4.72 se puede ver el netlist de un modulador $\Sigma\Delta$ de segundo orden. Vemos que el fichero de netlist se puede dividir en cinco partes

```
#Modulador sigma-delta single loop

#Sentencias de control
alimentacion 1.5 -1.5 1.5 -1.5;
nmuestras 65536 256;
output snr Xout;
reloj frec=2.5e6;

#Fuente de entrada
fuente vin Xin 0.5 1220.703125;

#Estructura del modulador
integrador1 int1 oi1 Xin xout 0.25 modelo int;
integrador2 int2 oi2 oi1 gnd 0.25 oi1 xout 0.25 modelo int;
comparador comp1 Xout oi2 modelo comp;

#Modelos de los distintos elementos
modelo integrador1 int cp11=1.5e-12 cp12=1.5e-12 cp13=1.5e-12 cp2=1.5e-12
                    clo=1.0e-12 ron=700 gm=GM imax=IMAX a0=1000 inpsd=1.6e-9
modelo comparador comp htype=1 vhis=30e-3 voff=30e-3;

#Definición de parámetros
param GM=4.2e-3;
#param GM=monte 10 gauss(4.2e-3 0.5e-3);
          # A sustituir si se quiere una análisis de montecarlo de la transconductancia
          # con media 4.2mA/V y desviación típica 0.5mA/V
param IMAX=1.5e-3;
```

**Figura 4.72. Esquema de un fichero de netlist de un modulador de segundo orden**

- *Sentencias de control*: Con estas sentencias se definen el nodo que queremos tomar como salida, el tipo de análisis que queremos hacer, el número de muestras de salida, la frecuencia de muestreo, la tasa de sobremuestreo, y las tensiones máxima y mínima de los cuantizadores o comparadores.

- *Fuente de entrada*: Se define la frecuencia y la amplitud de la señal de entrada

- *Estructura del modulador*: Se define la topología del modulador $\Sigma\Delta$ que puede estar compuesta por integradores, comparadores, cuantizadores o convertidores DA, sumadores, retrasos...

- *Modelos de los elementos*: En este bloque se definen los parámetros de las no idealidades de cada uno de los bloques que componen el modulador. Los parámetros no tienen necesariamente que ser especificados su valor, sino que se puede definirlos mediante un nombre al que posteriormente se le especificará su valor en la parte de definición de parámetros.

- *Definición de parámetros*: En este bloque se asignan valores a los parámetros que se han definido mediante un nombre en la parte de modelos. También sirve para variar alguno de esos parámetros, o bien, asociándoles una distribución de probabilidad y variándolos aleatoriamente, o bien, haciendo que varíen en un rango que se especifique.

# 4.4. Descripción mediante esquemáticos

La descripción del circuito mediante esquemáticos se hace en la herramienta HDL – DESIGNER de MENTOR [63]. El proceso a seguir para obtener un fichero VHDL o VERILOG valido se puede ver en la Figura 4.73.

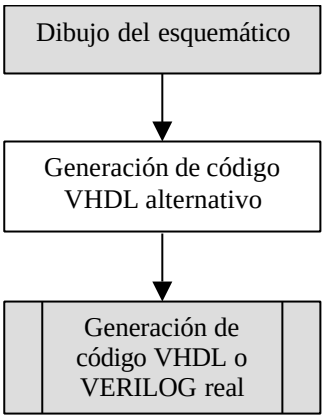



**Figura 4.73. Diagrama de flujo de la descripción mediante esquemáticos**

El HDL – DESIGNER hace uso de bloques no reales, que contienen solo las entradas y salidas de cada bloque básico, pero no tienen todas las entradas necesarias, ya sean parámetros, o en el caso de los integradores, los puertos correspondientes a las tensiones de los integradores de muestreo. Esto se debe, a lo difícil que sería conectar en un esquemático los bloques con todas las entradas y salidas anteriormente mencionadas. Estos bloques no reales no tienen ninguna arquitectura asociada o funcionamiento asociado. Debido a esto, es necesario que existe una manera que no sea difícil para el usuario, que transforme este código VHDL alternativo a uno real como se explica a continuación.

Como se puede ver en la Figura 4.73. el proceso se divide en tres partes que son:

- *Dibujo del esquemático*: En esta herramienta dispondremos de símbolos correspondientes a cada uno de los bloques básicos de las arquitecturas $\Sigma\Delta$. Para acceder a la biblioteca de esquemáticos tenemos que arrancar la herramienta y nos aparecerá el *design browser*, que se puede ver en la Figura 4.74, con los bloques que se tienen diseñados o importados.

  Para acceder a cualquiera de los esquemáticos que tengamos guardados, tan sólo hace falta hacer *doble clic* sobre el nombre del esquemático.

Si se quiere crear un esquemático nuevo, entonces desde el menú *File*, se tiene que escoger la opción de *create a new block diagram*. Para crear un modulador se tienen que conectan los distintos elementos de la estructura ΣΔ que queremos simular. Estos elementos son los integradores, comparadores, cuantizadores, convertidores DA, sumadores, retrasos y también son necesarias la señal de entrada al circuito y un bloque que sirve para indicar la frecuencia de muestreo. Cada uno de los símbolos tienen propiedades para cambiarles los distintos parámetros correspondientes a las no idealidades implementadas. Esto será explicado en más detalle en el apéndice B.

En la Figura 4.75 se muestra el esquemático de un modulador ΣΔ de segundo orden. Una vez creado, para salvarlo se escogerá del menú *File*, la opción de *save as*.

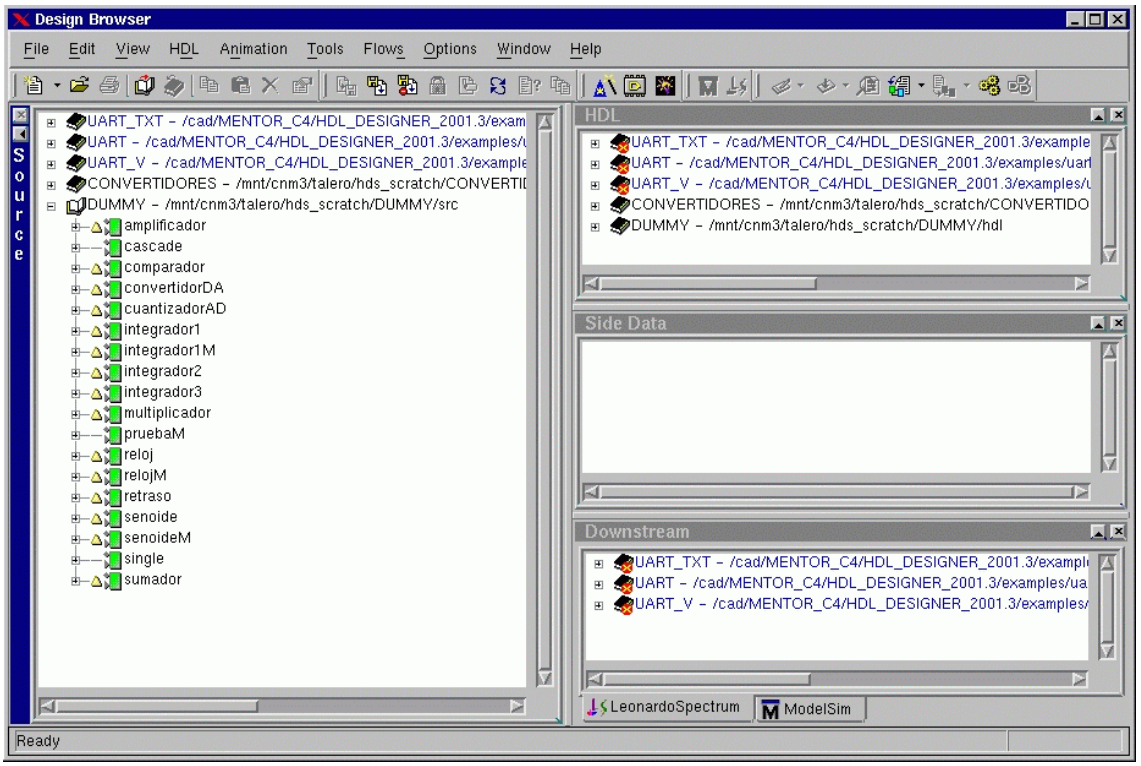



**Figura 4.74. Ventana desde dónde se accede a la biblioteca de esquemáticos.**

- *Generación de código VHDL alternativo*: La herramienta HDL – DESIGNER permite generar un fichero VHDL correspondiente a este esquemático. Este fichero VHDL contiene instancias a unos bloques ficticios que no sirven para simular nuestro circuito. Supongamos que se ha guardado con el nombre de *"single1.vhd"*. Para ello es necesario generar un fichero que contenga las instancias a los bloques reales necesarios para simular cualquier topología ΣΔ.

- *Generación de código VHDL real*: Para generar el fichero con el código VHDL O VERILOG real utilizaremos la herramienta *schematic*. Esta herramienta genera un fichero con el mismo nombre que el fichero VHDL alternativo pero con extensión *".par"*. Este fichero contendrá un netlist que posteriormente se le pasará a la herramienta VSIDES para generar el fichero VHDL O VERILOG final. El comando para realizarlo es el siguiente:

*% schematic nombre_archivo_sin_extensión(single1) –formato*

siendo formato *vhd* o *v* según sea VHDL o VERILOG el fichero final.

**¡Error!Marcador no definido.**
**Figura 4.75. Esquemático de un modulador sigma – delta de segundo orden**

Al ejecutar el programa se le preguntará al usuario cuál nodo entre todos los posibles, que coincide con los nombres que tienen los nodos del esquemático, quiere escoger como nodo de salida. Para ello se mostrará una lista con todos los nodos posibles. Seguidamente se preguntará el tipo de análisis que se quiere realizar, entre los tres posibles: temporal, espectro de salida y SNR. Finalmente se preguntara al usuario cuántas muestras quiere generar de la salida y la razón de sobremuestreo.

Si continuamos con el ejemplo de la Figura 4.75 la ejecución de schematic produciría lo siguiente, donde se ha escogido un número de muestras de 65536 y una tasa de sobremuestreo de 256 como se muestra en la Figura 4.76.

```
% schematic single_struct.vhd –vhd
Xout      oi2       Xin       oi1       gnd

Escoge un node de salida entre los anteriores: Xout
Tipo de análisis
0   Analisis temporal
1    Espectro de potencia
2    SNR
Número de muestras a generar: 65536
Tasa de sobremuestreo : 256
%
```

**Figura 4.76. Proceso de ejecución del programa schematic**

# 4.5. Compilado y simulación

Una vez que se ha descrito el circuito es necesario compilarlo y simularlo. Dependiendo del lenguaje de programación este será simulado en ADVANCE – MS para VHDL o MODELSIM para VERILOG. Es necesario crear un fichero con extensión *.do* en el que se le indique al simulador el tiempo necesario para simular en nanosegundos y que devuelva el control al sistema operativo(exit) después de haber realizado la simulación. El tiempo de simulación se calcula de la siguiente forma:

Tiempo de simulación = Periodo de muestreo*(numero de muestras +2)*Variaciones de parámetro

que en el caso de un modulador con frecuencia de muestreo de 2.5 MHz, 65536 muestras y no querer variar ningún parámetro sería 26215200 nanosegundos. Por lo tanto, en el fichero .do quedaría y supondremos que lo hemos llamado *sigma1.do*

```
run 26215200
exit
```

Supongamos que hemos creado un fichero con un modulador $\Sigma\Delta - SC$ llamado *sigma1.vhd*. Para compilarlo hay que ejecutar lo siguiente:

- ADVANCE – MS: vacom nombre_archivo(sigma1.vhd)
- MODELSIM: vlog –WORK biblioteca nombre_archivo (sigma1.vhd)

Para arrancar el simulador hay que decirle el nombre del archivo *.do* que hemos creado e indicarle el nombre de la entidad y la arquitectura para el caso de VHDL y el nombre del modulo y la biblioteca donde se ha compilado para el caso de VERILOG. En nuestro caso la biblioteca donde siempre se han compilado todos los modulos se llama CONVERTIDORES. El módulo de VERILOG se llama single1 y la arquitectura en VHDL es TEST perteneciente a la entidad single1. Para arrancar la herramienta habría que escribir los siguientes:

- ADVANCE – MS: vasim –c –do nombre_dichero_do(sigma1.do) ENTIDAD (single1) ARQUITECTURA (test)

- MODELSIM: vsim –c –do nombre_dichero_do(sigma1.do) BIBLIOTECA(convertidores).MODULO(single1)

# 4.6. Procesado de resultados

En el caso de MODELSIM, la simulación da un fichero llamado result.dat en tantas columnas como variaciones se hayan hecho de un parámetro en un anális de Montecarlo o en uno paramétrico y en cada columna los resultados temporales del nodo de salida. Este fichero tendrá que ser cargado por MATLAB para procesarlo y realizar algún tipo de los análisis descritos más abajo, ya que VERILOG no tiene las rutinas necesarias para el procesamiento de la señal.

La simulación temporal con ADVANCE MS da un fichero llamado result.dat. Este fichero tiene una columna con los resultados temporales de la salida en el caso de que se haya elegido un análisis temporal o en el dominio del tiempo.

Tendrá dos columnas una para los valores de la frecuencia y otra para los valores de potencia del la señal de salida en el caso de que se haya escogido ver el espectro de salida.

Tendrá dos columnas una para los valores del parámetro que se haya querido variar en un análisis de Montecarlo o análisis paramétrico y la otra los valores resultantes de la SNR en el nodo de salida.

**a) Análisis en el dominio del tiempo**

Permite una visualización de las formas de onda.

**b) Análisis dinámico**

La caracterización dinámica de los convertidores es especialmente importante para aplicaciones de frecuencias medias y altas. Los análisis dinámicos incluyen el espectro de salida, la SNDR o relación señal – (ruido + distorsión) como una función del nivel de entrada o frecuencia.

**c) Análisis de Montecarlo**

Este análisis tiene en cuenta las fluctuaciones de las ganancias de los integradores y de las especificaciones de bloques básicos. Esta capacidad es especialmente interesante cuando se utilizan arquitecturas en cascada, debido a su sensibilidad al desapareamiento. Por otro lado, las especificaciones terminales de los bloques básicos, en el diseño a nivel eléctrico, pueden cambiar debido a variaciones de los parámetros eléctricos, cambios en la temperatura, etc. Estos cambios pueden ser incluidos en el análisis de Montecarlo para evaluar su impacto en el funcionamiento del modulador. El análisis de Montecarlo sólo se puede usar para obtener la SNR del nodo de salida.

**d) Análisis paramétrico**

A través de estos análisis es posible realizar algunos de los análisis previos con no idealidades definidas como parámetros variables. Esto nos permite explorar el espacio de diseño y monitorizar individualmente las influencias de parámetros críticos del diseño. El análisis de Montecarlo solo se puede usar para obtener la SNR del nodo de salida.

# 4.7. Ejemplos

En este apartado se van a explicar dos circuitos para ilustrar los moduladores ΣΔ. La validación de la simulación con ADVANCE-MS se llevará a cabo con ASIDES, que es un simulador de comportamiento para moduladores ΣΔ-SC.

La comparación se realizará siempre de la misma manera. Se tomarán las secuencias temporales de cada simulador y se procesarán con MATLAB, comparando posteriormente los resultados.

Se van a analizar dos ejemplos:

- El primero de los ejemplos que se va a analizar es un modulador ΣΔ-SC de segundo orden y lazo simple, al que se le aplicarán cada una de las no idealidades implementadas en VHDL y VERILOG, comparando estos resultados con ASIDES.

- El segundo ejemplo muestra el análisis de un modulador ΣΔ- SC utilizando una arquitectura en cascada y multibit. En concreto se analizará una arquitectura 2-1-1 multibit.

## 4.7.1. Modulador Sigma – Delta lazo simple de segundo orden

En la Figura 4.73 se muestra un esquemático del circuito empleando los bloques del HDL – DESIGNER. Los parámetros ideales de cada bloque se indican a continuación.

- Reloj: frecuencia de muestreo: 2.5 MHz. M= 256
- *Senoide*: Amplitud =1.0; Frecuencia =1220.703125 Hz;
- *Integrador1*: C2 =2pF; Peso = 0.25;
- *Integrador2*: C2 =2pF; Peso1=0.25; Peso2=0.25;
- *Comparador*: $V_{max} = -V_{min} = 1.5$ v

El espectro obtenido con ADVANCE-MS a partir de la simulación y posterior procesado de la secuencia temporal es exactamente igual que el obtenido capturando la secuencia temporal que se obtiene con ASIDES, en el caso ideal, y que posteriormente se podrá comprobar con las no idealidades.

Para el caso ideal se ha obtenido que tanto ASIDES como ADVANCE – MS dan una SNR de 97.43 dB.

#### 4.7.1.1. Efecto de la ganancia DC finita y lineal de los integradores

Los parámetros que hay que añadir al caso ideal en los integradores son los siguientes.

- *Integrador2*: A0=10;
- *Integrador1*: A0=10;

La Figura 4.77 y Figura 4.78comparan los espectros en el caso ideal y real tras el análisis tanto de las secuencias obtenidas de ASIDES como las de ADVANCE – MS, comprobando la similitud entre ambos.

Para estos datos se ha comprobado que tanto ASIDES como ADVANCE – MS producen la misma SNR que en este caso es 75.38 dB, frente a los 97.43 dB que daba el caso ideal.

#### 4.7.1.2. Efecto de la ganancia DC finita y no lineal de los integradores

Los parámetros que difieren con el caso ideal son:

- *Integrador2*: A0=1000 a01=0.05 a02=0.05;
- *Integrador1*: A0=1000 a01=0.05 a02=0.05;

En la Figura 4.79 y la Figura 4.80 podemos apreciar como se deforma el espectro ideal aumentando el ruido dentro de la banda de la señal tanto en un simulador como en otro.

En este caso ASIDES y ADVANCE – MS difieren en la SNR, pero tan sólo con una diferencia de 0.5 dB. Concretamente ASIDES da 97.05 dB y ADVANCE 96.55 dB.

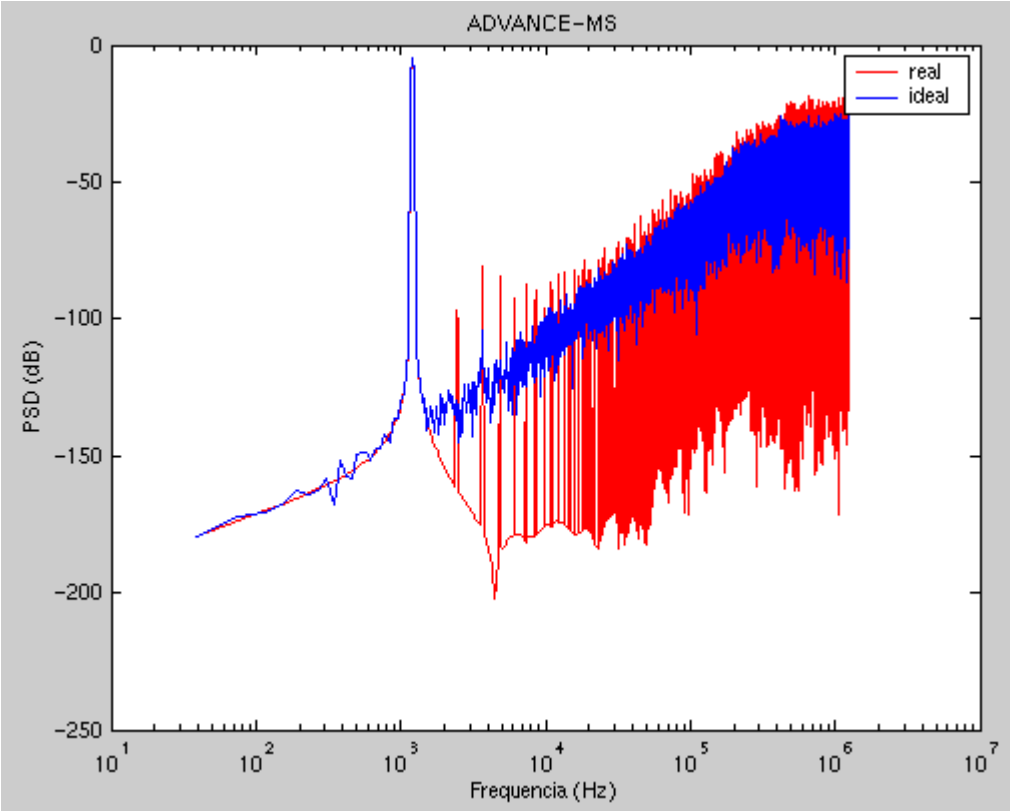



**Figura 4.77. Espectro real con ganancia finita e ideal simulado con ADVANCE – MS**

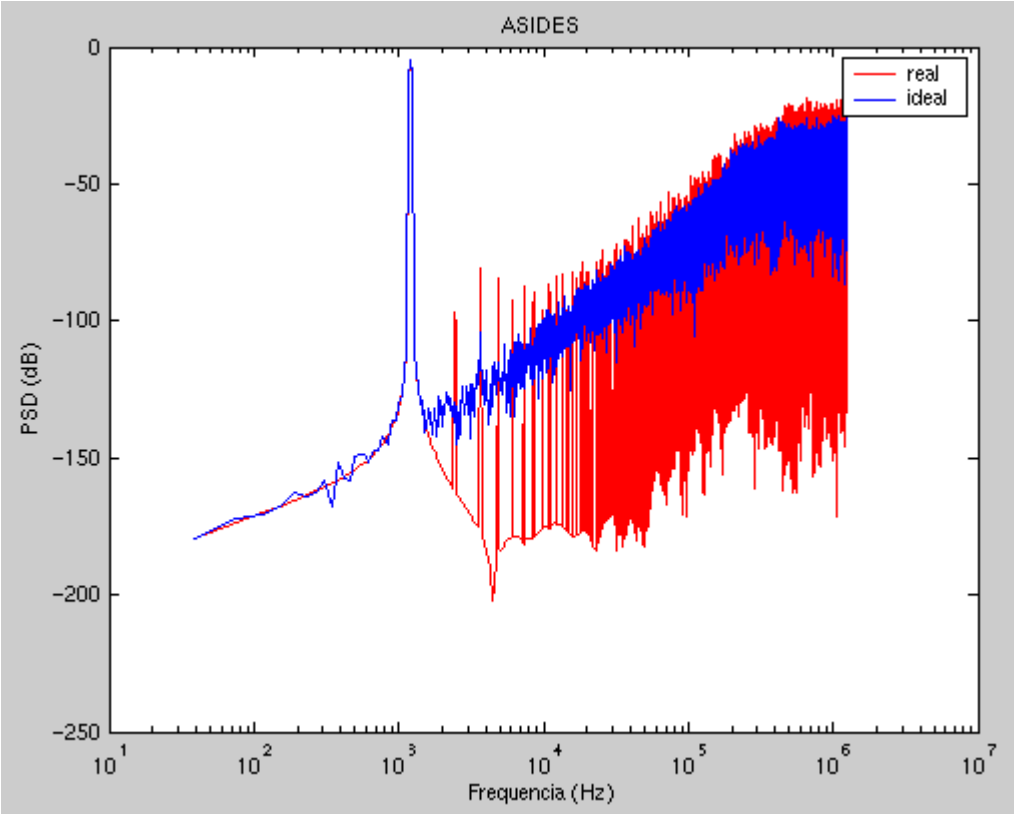



**Figura 4.78. Espectro real con ganancia finita e ideal simulado con ASIDES**

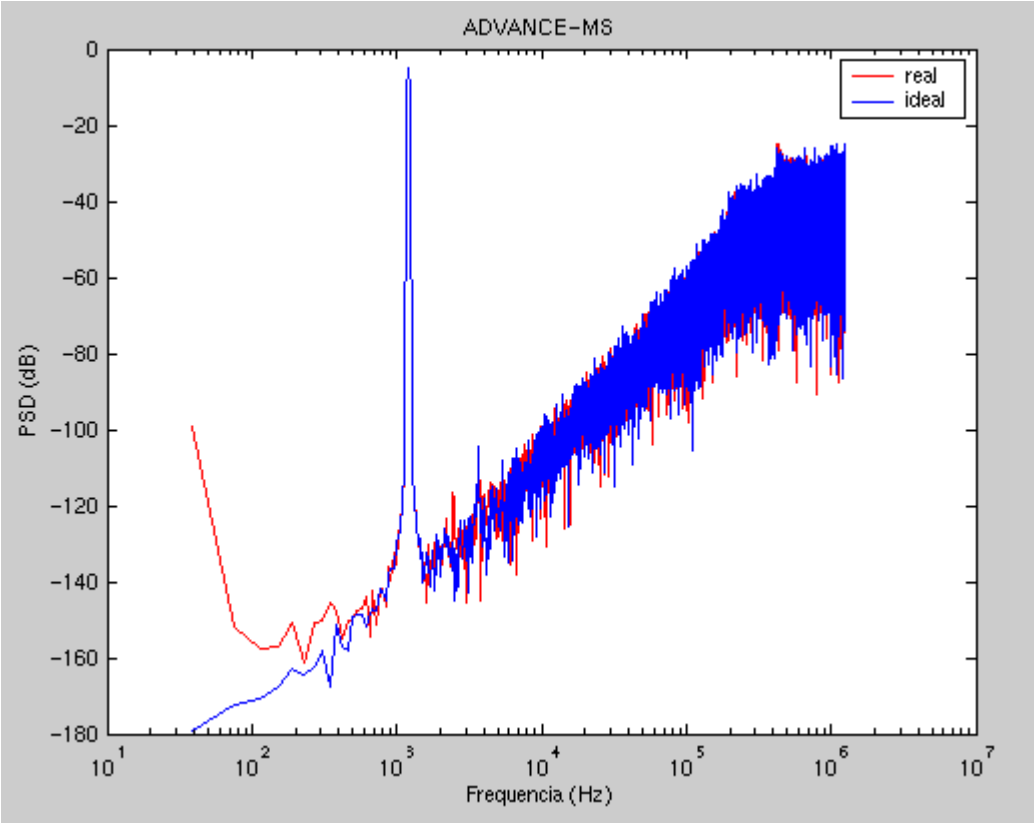



**Figura 4.79. Espectro real con ganancia finita y no lineal e ideal simulado con ADVANCE-MS**

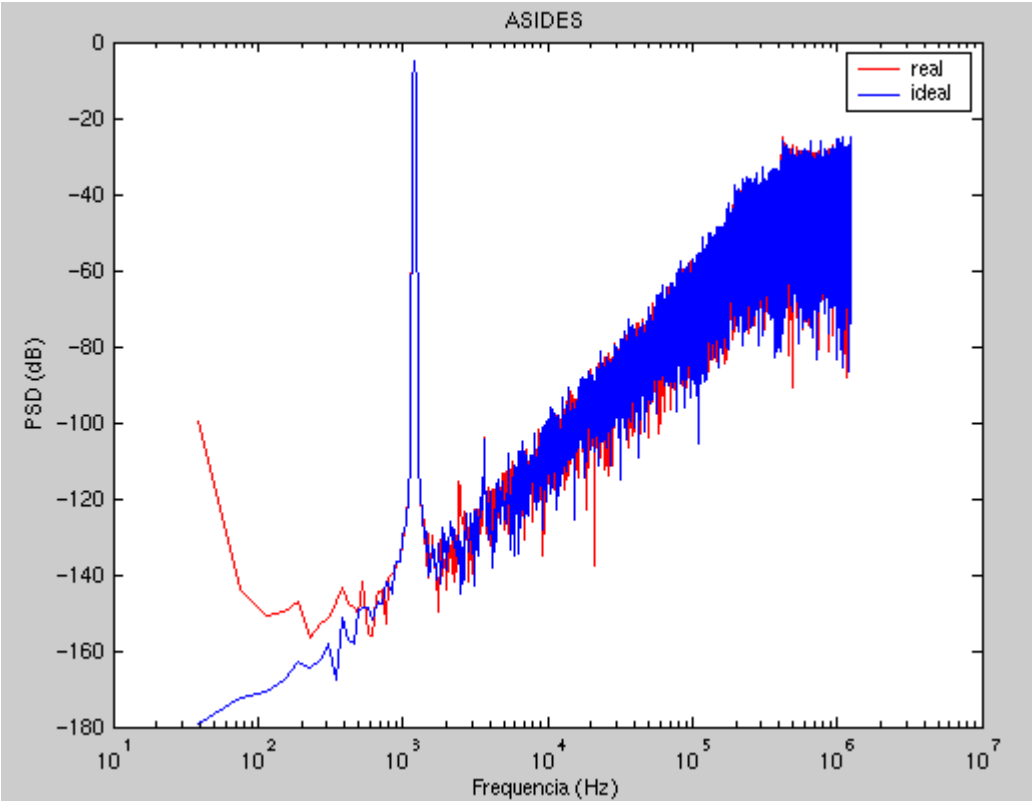



**Figura 4.80. Espectro real con ganancia finita y no lineal e ideal simulado con ASIDES**

#### 4.7.1.3. Efecto de la no linealidad de los condensadores de muestreo

Los parámetros que difieren con el caso ideal son:

- *Integrador2*: alpha=0.01; beta=0.01;
- *Integrador1*: alpha=0.01; beta=0.01;

En la Figura 4.81 y en la Figura 4.82 se puede observar como el espectro sufre una importante deformación con la presencia de un "offset" de ruido y la aparición de armónicos en diferentes frecuencias consecuencia de la no linealidad.

Como consecuencia de esta no linealidad, la SNR sufre un importante decremento, pero sigue siendo semejante tanto en el caso de ASIDES como en el de ADVANCE – MS con SNR= 51.88 dB

En la Figura 4.83 y en Figura 4.84 nos muestra la SNR para el caso de los condensadores no lineales tanto para el caso ideal como para el caso con los valores de condensadores no lineales mostrados anteriormente variando la amplitud de la señal de entrada. Lo importante en esta gráfica es el punto de inflexión a partir del cual empieza a decrecer la SNR. Vemos que para valores muy bajos de la tensión de entrada existe cierta diferencia entre ASIDES y ADVANCE – MS.

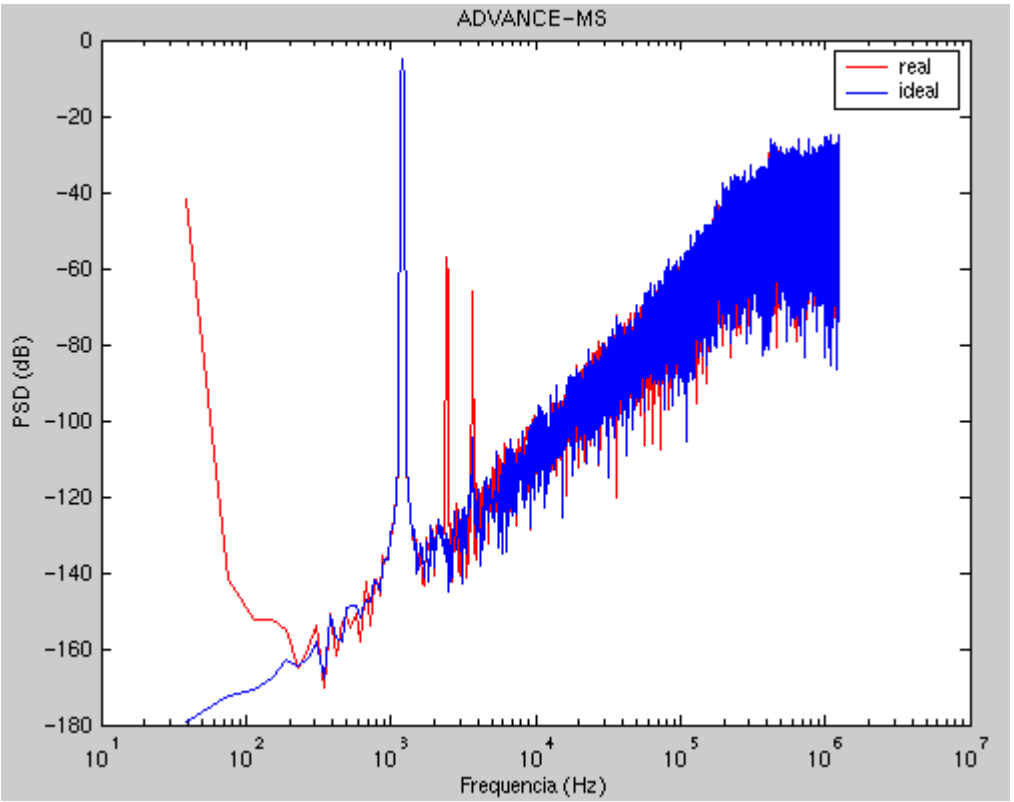



**Figura 4.81. Espectro real con condensadores no lineales e ideal simulado con ADVANCE – MS**

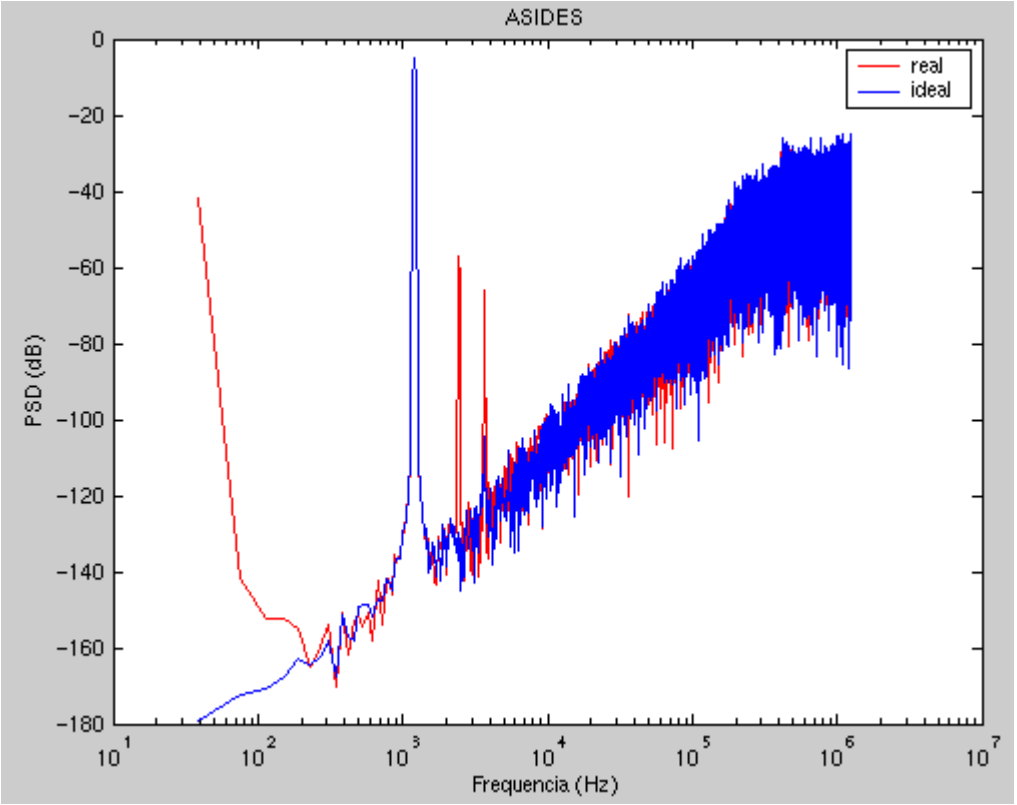



**Figura 4.82. Espectro real con condensadores no lineales simulado con ADVANCE – MS**

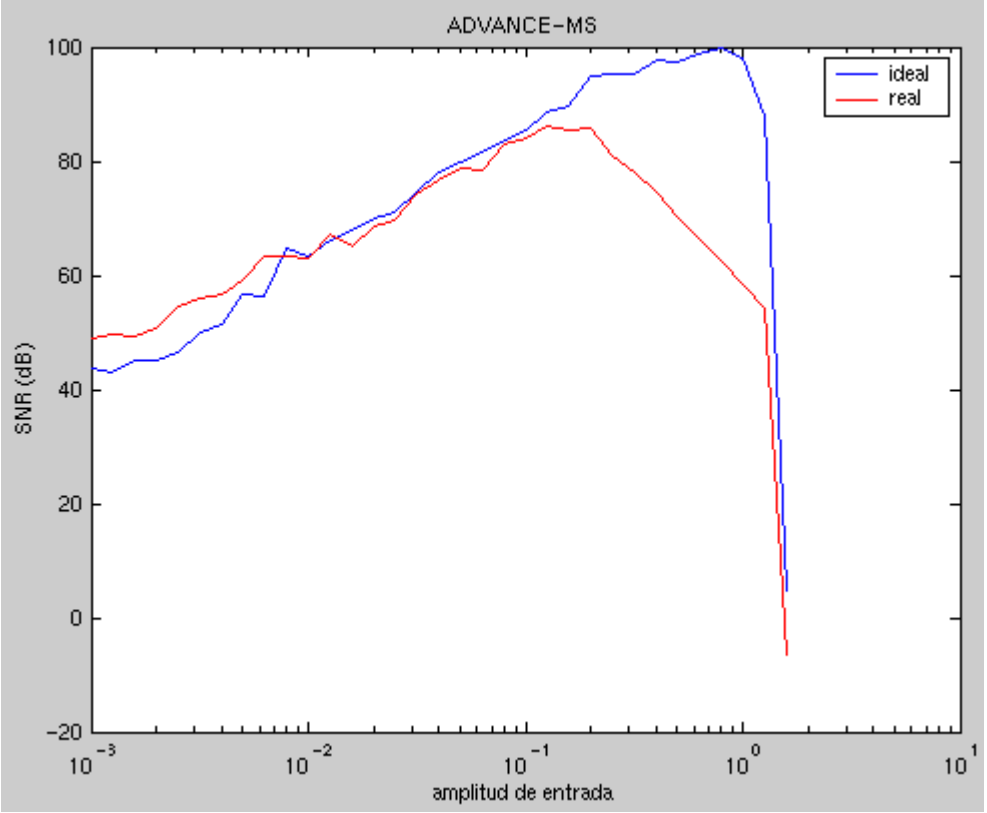



**Figura 4.83. SNR en función de la amplitud de entrada para el caso ADVANCE - MS.**

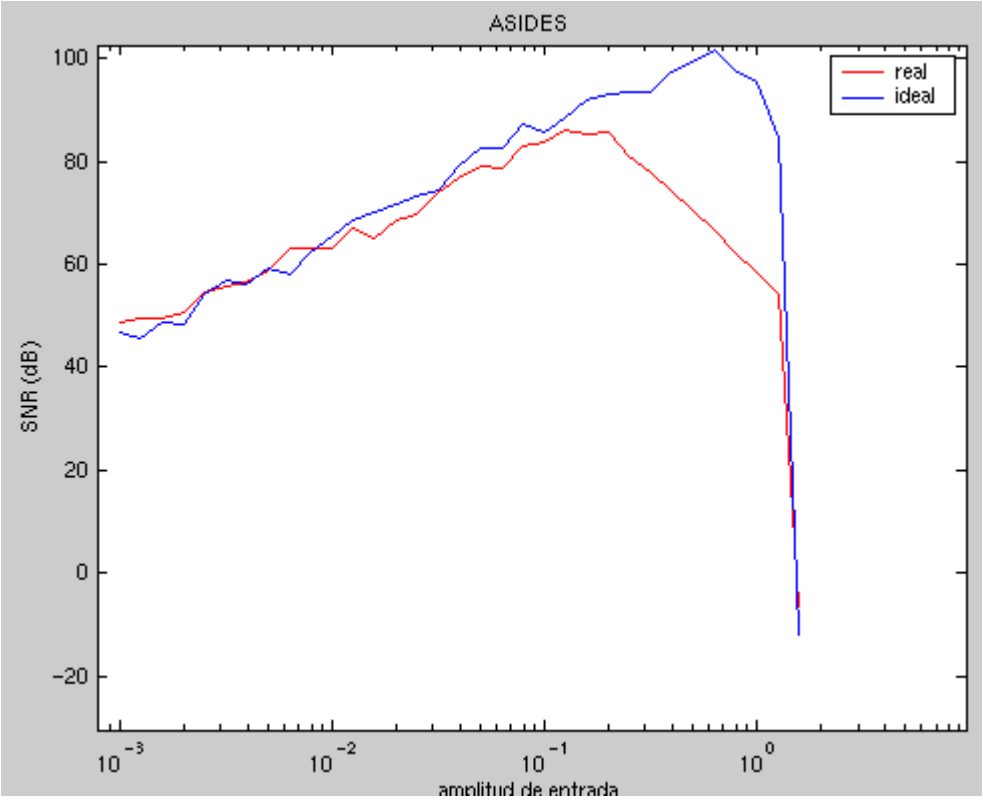



**Figura 4.84. SNR en función de la amplitud de entrada para el ASIDES**

### 4.7.1.4. Efecto del desapareamiento en la ganancia de los integradores

Los parámetros que difieren en el caso ideal son:

- *Integrador2*: sigma =0.0001;
- *Integrador1*: sigma =0.0001;

En este caso los resultados han variado alrededor de 1dB en la simulación de los distintos simuladores. En concreto ASIDES da una SNR de 97.33 y ADVANCE – MS de 98.33.

Para esta simulación ASIDES no permitía variar el parámetro de la varianza de los condensadores, por lo que se tuvo que estimar que valor había que poner en los pesos de los integradores en ASIDES. Según el capítulo anterior, este error se modela tomando como pesos, valores aleatorios de una distribución gaussiana, de media el peso de integración y de varianza sigma. El valor que hay que poner en los pesos de integración en ASIDES coincide con el primer valor aleatorio que diera ADVANCE – MS para ese peso de integración y esa varianza de condensador no lineal.

En la Figura 4.85 y en la Figura 4.86 se pueden comprobar los espectros de los dos simuladores, donde se observa de nuevo que los resultados en los dos simuladores apenas difieren.

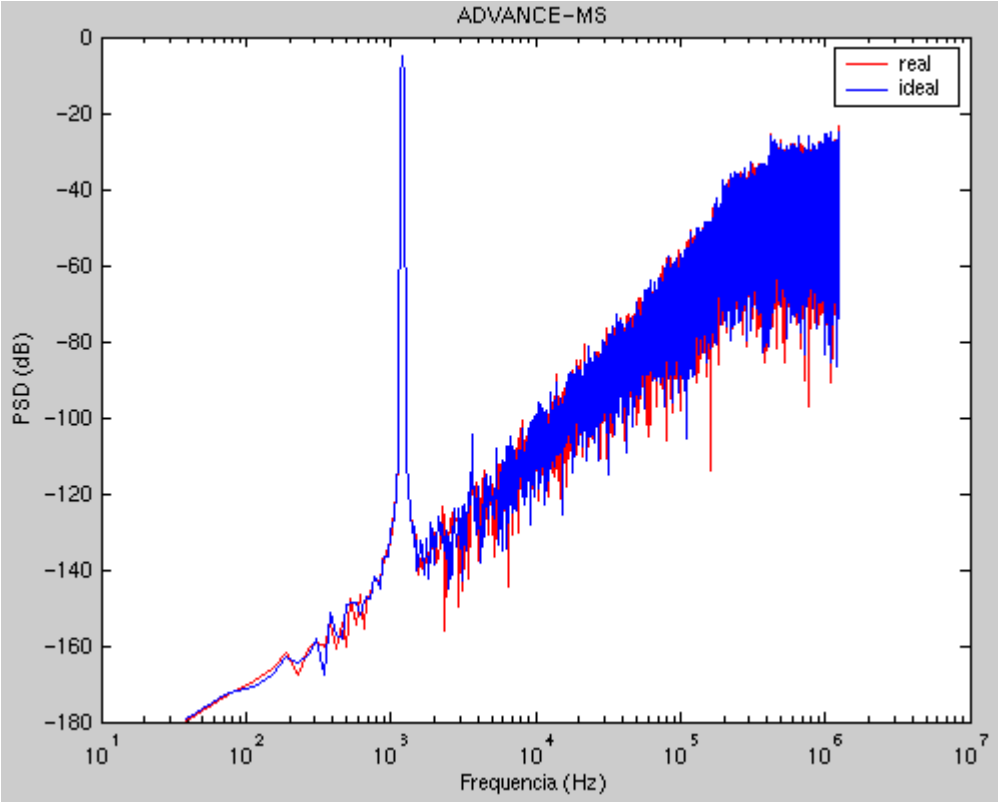



**Figura 4.85. Espectro real con desviación en el peso de integración e ideal simulado con ADVANCE - MS.**

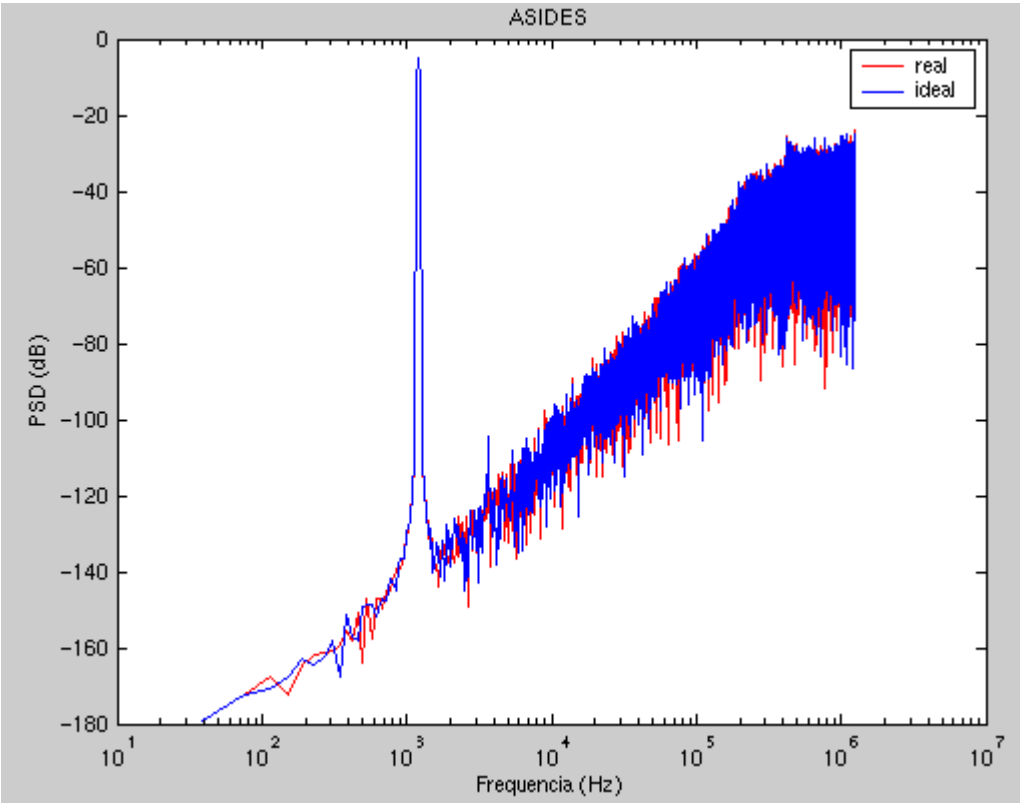



**Figura 4.86. Espectro real con desviación en el peso de integración e ideal simulado con ASIDES.**

#### 4.7.1.5. Efecto del ruido térmico de los integradores

Los parámetros que difieren en el caso ideal son:

- *Integrador2*: ron =2k; clo =1pF; cp11 =1pF; cp2 =1pF; gm =100e-3; inpsd = 7n $V/Hz^{1/2}$
- *Integrador1*: ron =2k; clo =1pF; cp11 =1pF; cp12 =1pF; cp2 =1pF; gm =100e-3; inpsd = 7n $V/Hz^{1/2}$

La Figura 4.87 y la Figura 4.88 muestran los resultados a los que llegamos con las secuencias aleatorias de ruido provenientes de los dos simuladores, mostrando que éstas no son exactamente iguales. Se puede apreciar que el efecto del ruido térmico no es otro que, debido a su densidad espectral de potencia plana en toda frecuencia, elevar el ruido en banda en el entorno de la señal.

En este caso, la SNR para ASIDES es 97.33 y la de ADVANCE – MS es 97.73, es decir, solo 0'4 dB de ganancia, que puede ser debido simplemente a las diferentes muestras aleatorias que producen cada simulador.

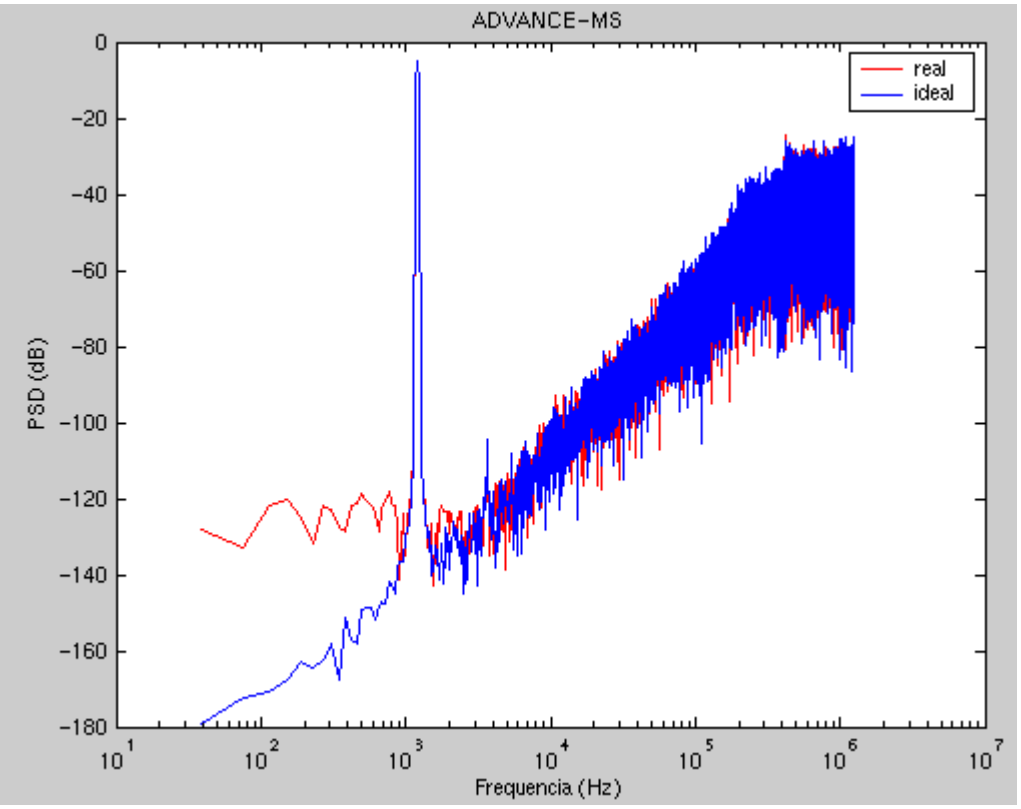



**Figura 4.87. Espectro real con ruido térmico e ideal simulado con ADVANCE – MS**

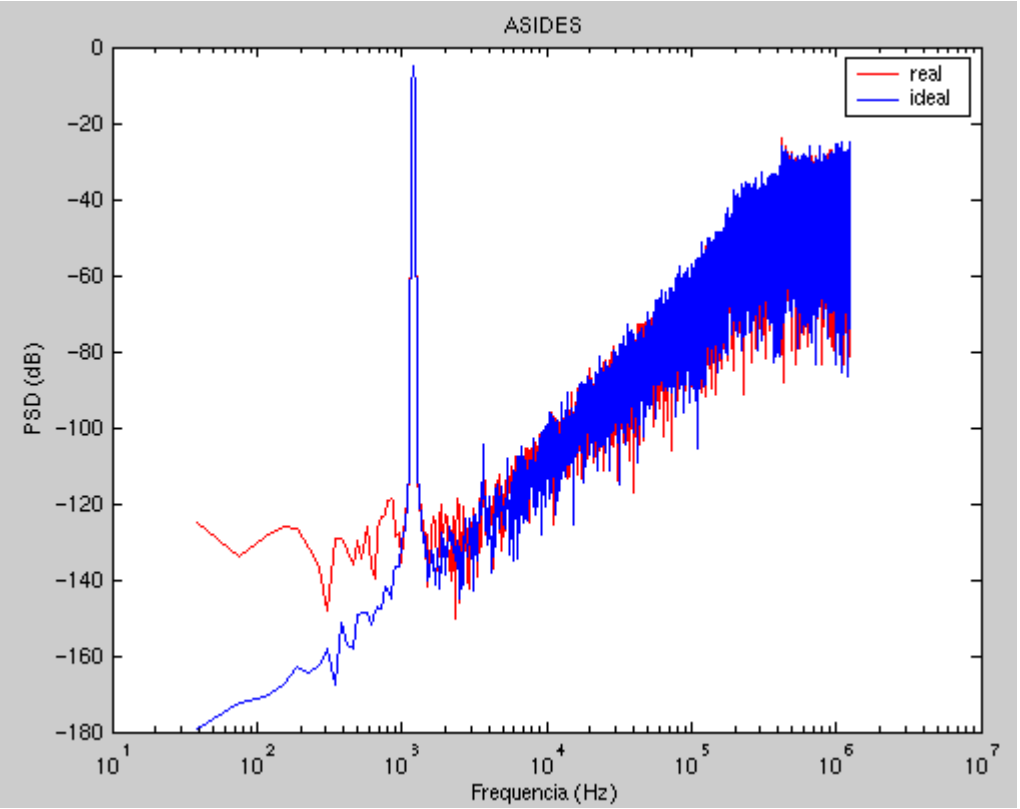



**Figura 4.88. Espectro real con ruido térmico e ideal simulado con ASIDES**

### 4.7.1.6. Efecto del error de establecimiento o settling

Los parámetros que difieren en el caso ideal son:

- *Integrador2*: gm =3e-6; clo =1p; imax =60e-6 A
- *Integrador1*: gm =3e-6; clo =1p; imax =60e-6 A

En la Figura 4.89 y en la Figura 4.90 se observan la similitud entre los resultados de ambos simuladores. En este caso la SNR coincide para ambos dando una SNR de 60.74 dB.

En la Figura 4.91 se puede comprobar el efecto que esta no idealidad tiene sobre la SNR y se aprecia como a medida que va creciendo la transconductancia del integrador, disminuye el efecto en la SNR, aumentando gradualmente

### 4.7.1.7. Ejemplo agrupando algunos errores

En este ejemplo, los parámetros que difieren del caso ideal son:

- *Integrador2*: gm =10e-6; clo = cp1=cp2=1p; imax =60e-6 A; ron=2k; inpsd=7n;
- *Integrador1*: gm =10e-6; clo ==cp11=cp12=cp2=1p; imax =60e-6 A; ron=2k; inpsd=7n;

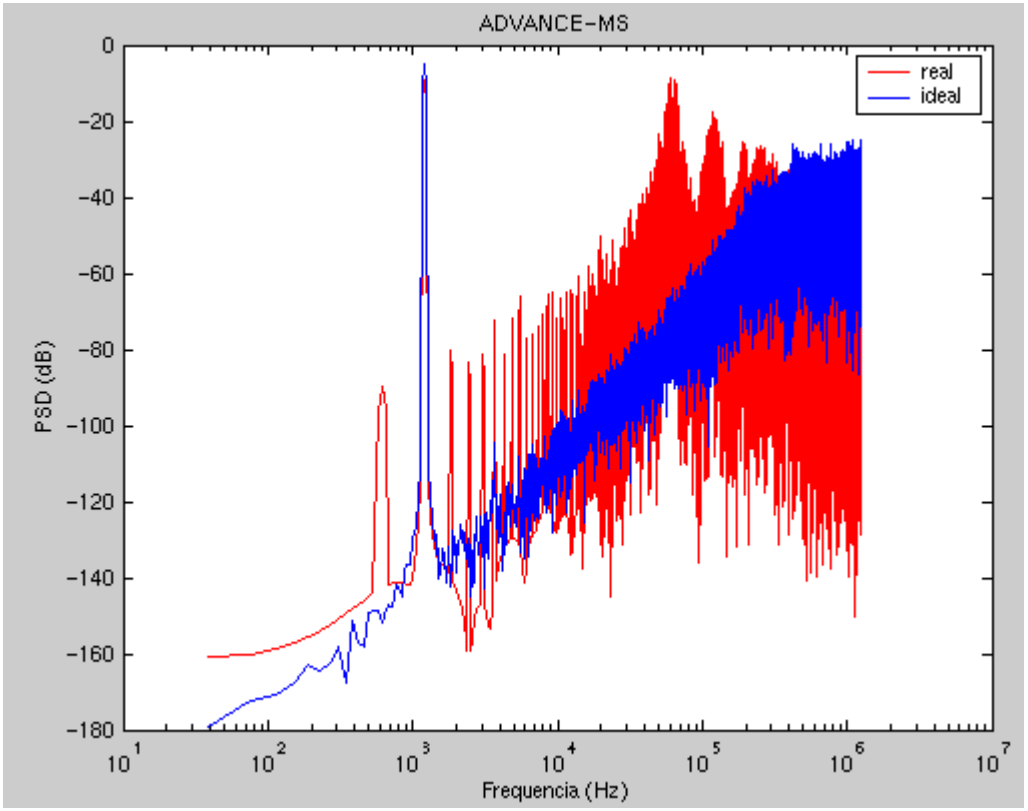



**Figura 4.89. Espectro real con error de establecimiento e ideal simulado con ADVANCE – MS**

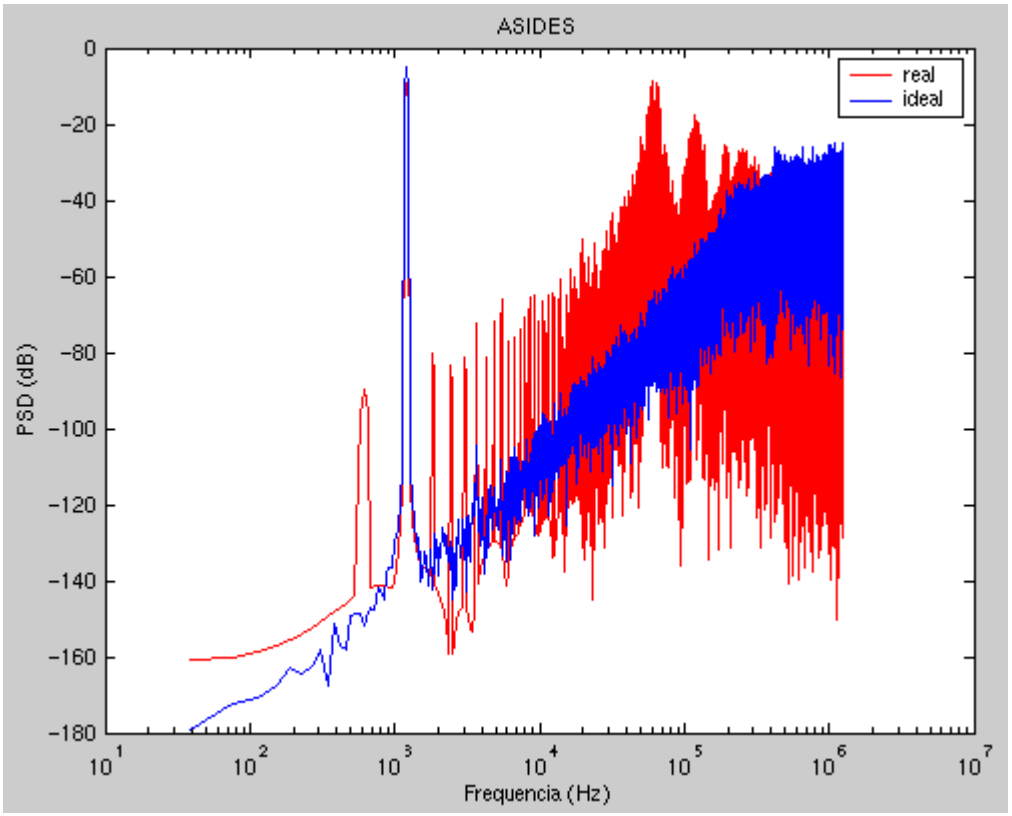



**Figura 4.90. Espectro real con error de establecimiento e ideal simulado con ASIDES**

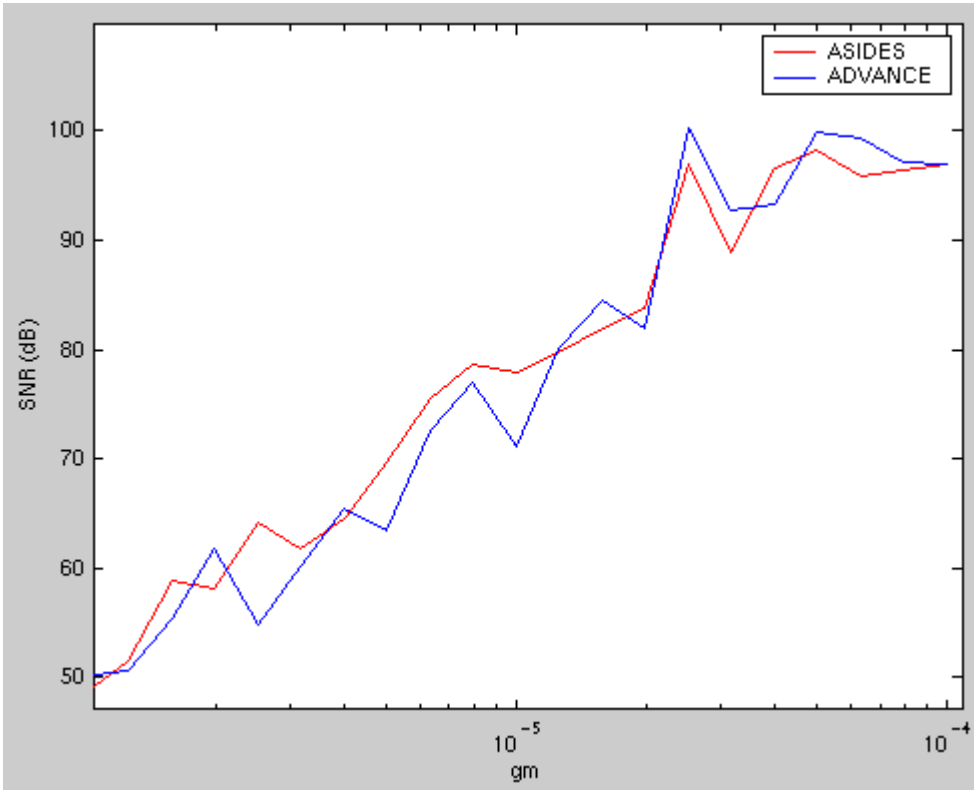



**Figura 4.91. Efecto del error de establecimiento en la SNR en función de la transconductancia**

Los errores que se encuentran en este ejemplo son los de ganancia finita, error de establecimiento y ruido térmico. Vemos que la diferencia entre los espectros obtenidos con ADVANCE – MS y con ASIDES es mínima. La diferencia encontrada en la SNR es tan sólo de 0.15 dB, obteniendose 74.1 en ASIDES y 74'25 en ADVANCE – MS.

## 4.7.2. Modulador Sigma – Delta SC cascada 2-1-1 y multibit (3 bits)

La Figura 4.94 muestra el diagrama de bloques de los moduladores de lazo simple que integran el modulador en cascada y la Figura 4.95 la lógica de cancelación que se aplica a la salida de dichos moduladores.

Los parámetros que se tienen en el caso ideal son los siguientes:

- Reloj: frecuencia de muestreo: 35.2 KHz. M = 16
- *Senoide*: Amplitud =1.0; Frecuencia = 275 kHz;
- *Integrador1*: C2 =2pF; Peso = 0.25;
- *Integrador2*: C2 =2pF; Peso1=0.25; Peso2=0.25;
- *Integrador3*: C2 =2pF; Peso1= 0.375; Peso2= 0.375; Peso 3= 0.25;
- *Integrador4*: C2 =2pF; Peso1=1; Peso2=1; Peso3=2.
- *Comparador*: $V_{max} = -V_{min} = 2$ v
- *Cuantizador*: $V_{max} = -V_{min} = 2$ v; $V_{DD} = -V_{SS} = 2$ v
- *Dac*: $V_{max} = -V_{min} = 2$ v $V_{DD} = -V_{SS} = 2$ v

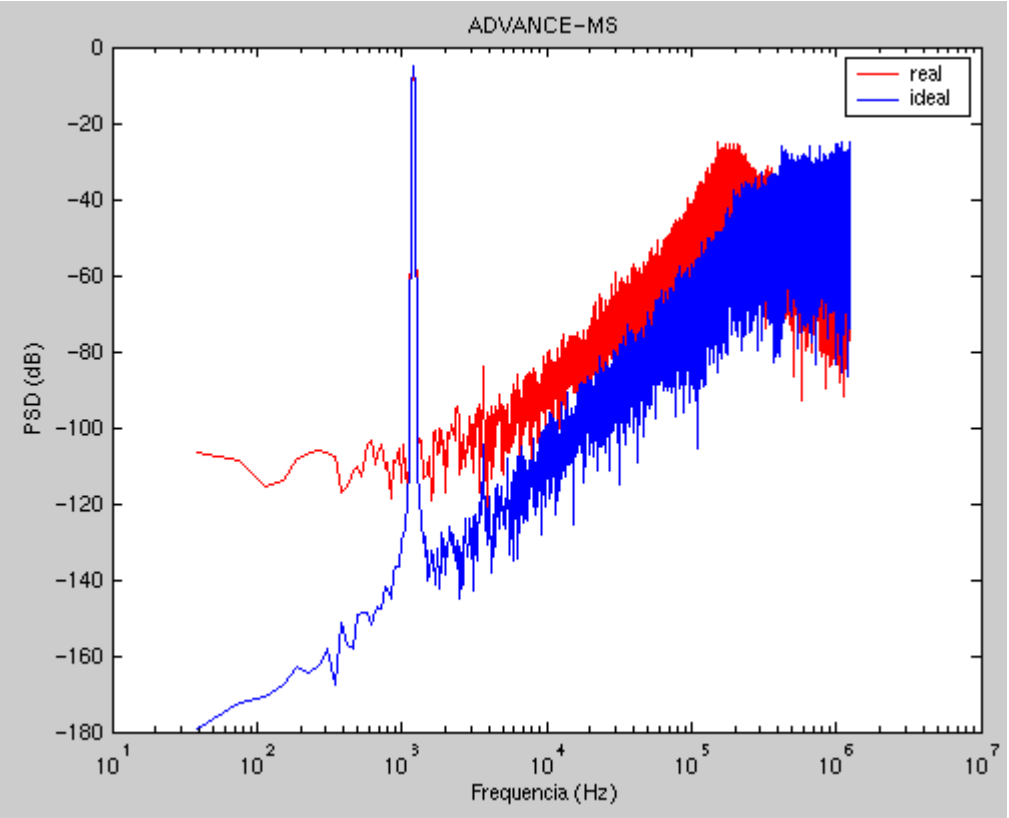



**Figura 4.92. Espectro real e ideal con varios errores simulado con ADVANCE – MS**

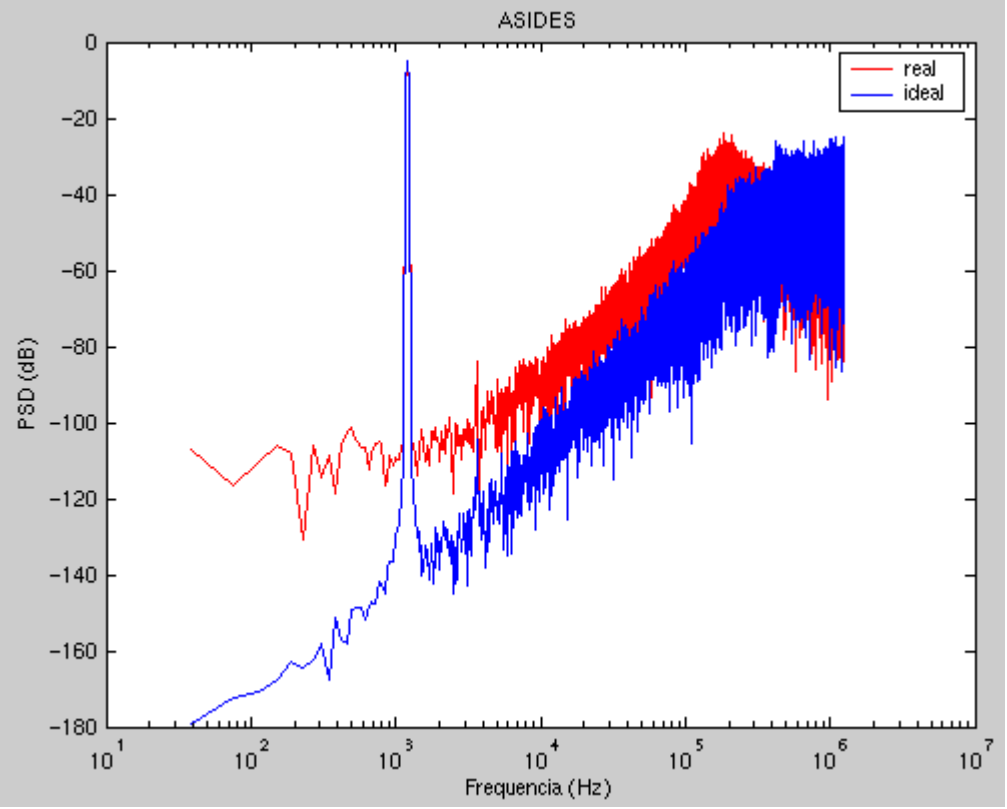



**Figura 4.93. Espectro real e ideal con varios errores simulado con ASIDES**

En este caso tanto ASIDES como ADVANCE – MS da un valor de SNR de 83.50.

Se estudiarán tres efectos que ejercen una alta influencia en el comportamiento de este tipo de moduladores, como es la desviación en el valor de la ganancia de los integradores o "mismatch", la no linealidad integral del CD/A (INL) y el efecto de la ganancia DC de los integradores finita. Todos estos efectos se compararán con el caso ideal anteriormente descrito y serán aplicados en la primera etapa del modulador, por ser esta la más influyente.

**¡Error!Marcador no definido.**
**Figura 4.94. Modulador Sigma – Delta SC cascada 2-1-1 multibit**

**¡Error!Marcador no definido.**
**Figura 4.95. Lógica de cancelación de un modulador Sigma – Delta SC cascada 2-1-1 multibit**

### 4.7.2.2. Efecto de la desviación en el peso de los integradores de la primera etapa

Los parámetros que difieren en el caso ideal son:

- *Integrador2*: sigma =0.0001
- *Integrador1*: sigma =0.0001

La Figura 4.96 y la Figura 4.97 muestran una comparativa de los espectros de la señal de salida del modulador en el caso real e ideal. Al igual que comentábamos para el caso del modulador de segundo orden, aquí se puede apreciar una diferencia en el espectro que puede ser debida a que ASIDES no permitía variar el parámetro de la varianza de los condensadores, por lo que se tuvo que estimar que valor había que poner en los pesos de los integradores en ASIDES, que coincidiera con el primer valor aleatorio que diera ADVANCE – MS para ese peso de integración y esa varianza de condensador no lineal.

LA SNR en ambos difiere tan solo 0.2 dB en ambos casos. En concreto sale utilizando las secuencias temporales de ASIDES 82.42 dB y las de ADVANCE 82.22 Db

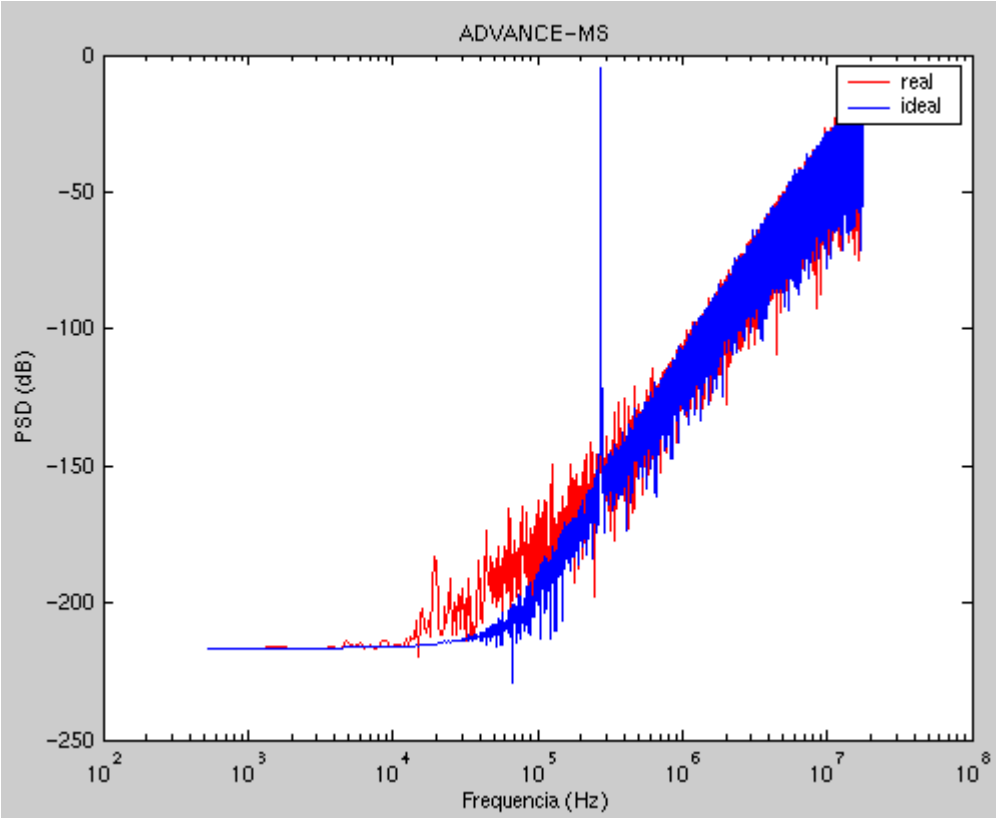



**Figura 4.96. Espectro real con desviación en los pesos de los integradores e ideal simulado con ADVANCE - MS**

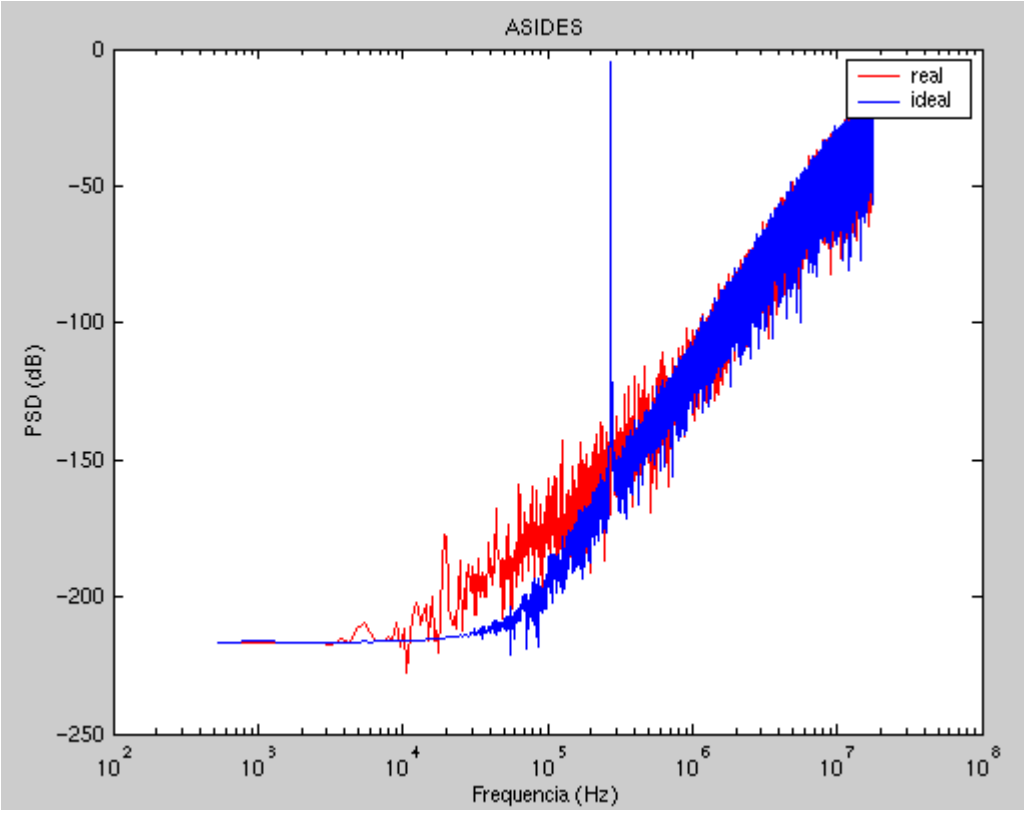



**Figura 4.97. Espectro real con desviación en los pesos de los integradores e ideal simulado con ASIDES**

#### 4.7.2.3. Efecto de la no linealidad en el CDA multibit

Los parámetros que difieren en el caso ideal son:

- CDA: inl =0.1 LSB

En la Figura 4.98 y en la Figura 4.99 se muestra el efecto de esta no idealidad en las dos herramientas. La SNR que se obtiene con ADVANCE – MS es de 77.70 dB y la de ASIDES 77.85 dB, teniendo una diferencia de sólo 0.15 dB.

#### 4.7.2.4. Efecto de la ganancia DC finita de los amplificadores

Este efecto se recoge en el modulador ideal efectuando un análisis paramétrico de la ganancia finita en DC (parámetro AV) del amplificador en las mimas condiciones que se describieron anteriormente.

El resultado se muestra en la Figura 4.100 donde se observa claramente que si no se supera un determinado umbral de la ganancia DC del primer integrador, estamos perdiendo resolución del convertidor, ya que podemos tener valores de SNR superiores.

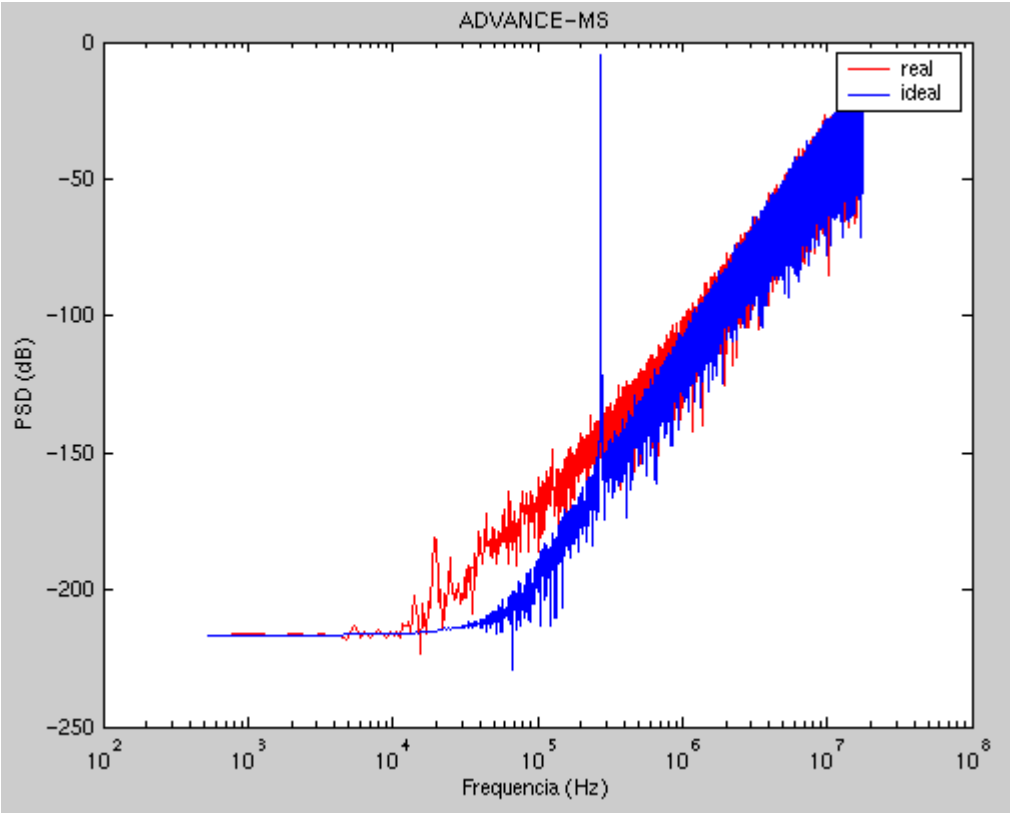



**Figura 4.98. Espectros real con INL e ideal simulado con ADVANCE - MS**

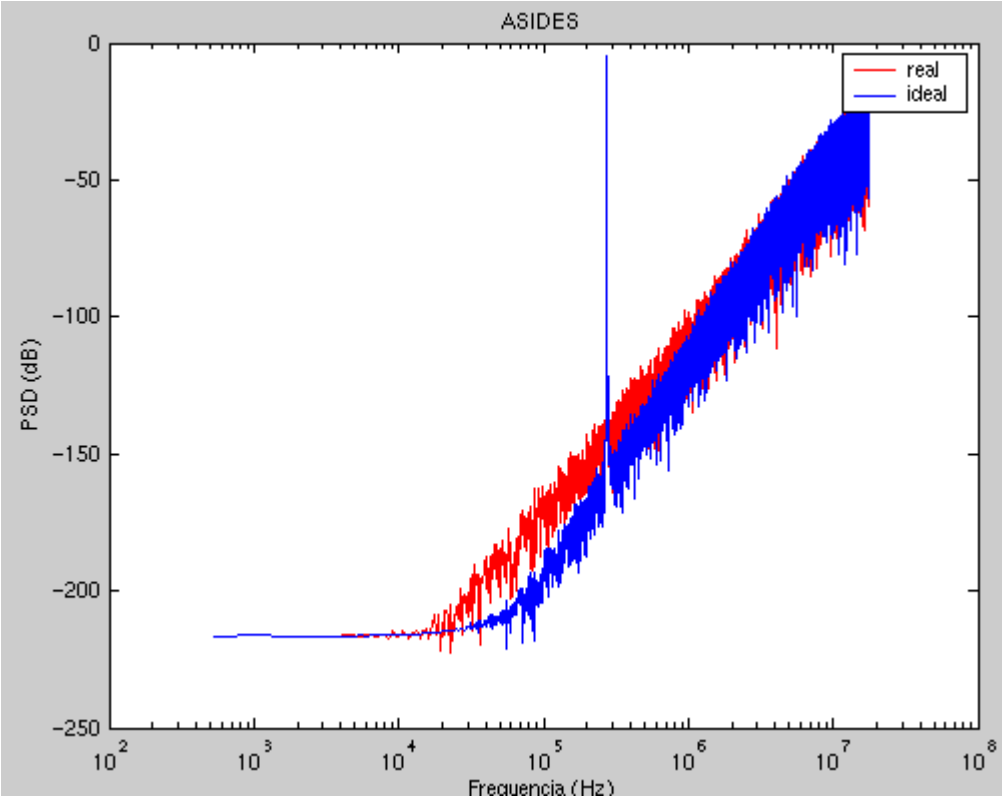



**Figura 4.99. Espectro real con INL e ideal simulado con ASIDES**

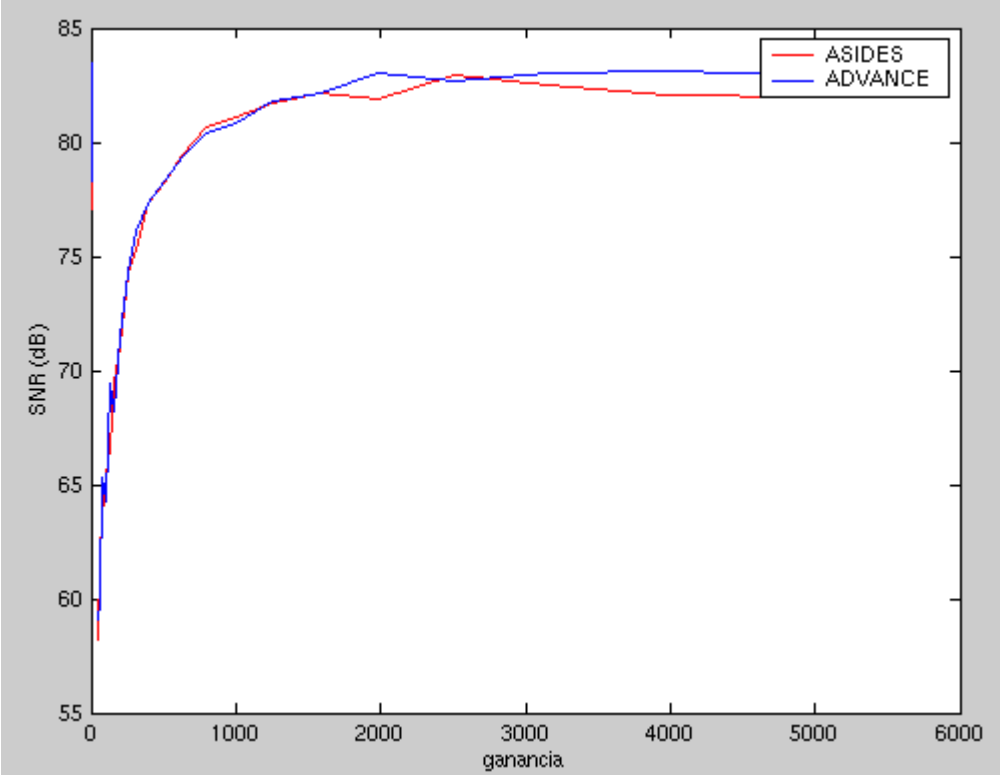



**Figura 4.100. SNR en función de la ganancia para ambos simuladores**

### 4.7.2.5. Ejemplo con varios errores

Los parámetros que difieren del caso ideal son:

- Integrador1: A0=100; gm =20m; imax= 2mA; clo=cp1=cp2=1p; ron=2k; inpsd=2n;
- Integrador2: A0=100; gm =20m; imax= 2mA; clo=cp11=cp12=cp2=1p; ron=2k; inpsd=2n;
- CDA: inl=0.1;

En este ejemplo se ha incluido el error de no linealidad del CDA, la ganancia finita, el ruido térmico y el error de establecimiento.

Los espectros simulados con ASIDES y con ADVANCE – MS (Figura 4.101 y 110) se pueden ver que se diferencia muy poco. La diferencia encontrada en la SNR, es de 0.65 dB, obteniéndose en ASIDES 65.43 dB y 64.78 en ADVANCE – MS.

# 4.8. Tiempos de simulación

En este apartado se muestra el tiempo que tarda en simular ADVANCE – MS un modulador $\Sigma\Delta - SC$ de lazo simple y de segundo orden y uno en cascada 2-1-1 en un ordenador compartido *SunFire800* con 4GB de RAM y 4 procesadores *UltraSparcIII* a 750 MHz con 8 MB de memoria Cache y sistema operativo *Solaris 8* comparándolos con la herramienta ASIDES [28] y SDTOOLBOX [64] hecha en MATLAB SIMULINK.

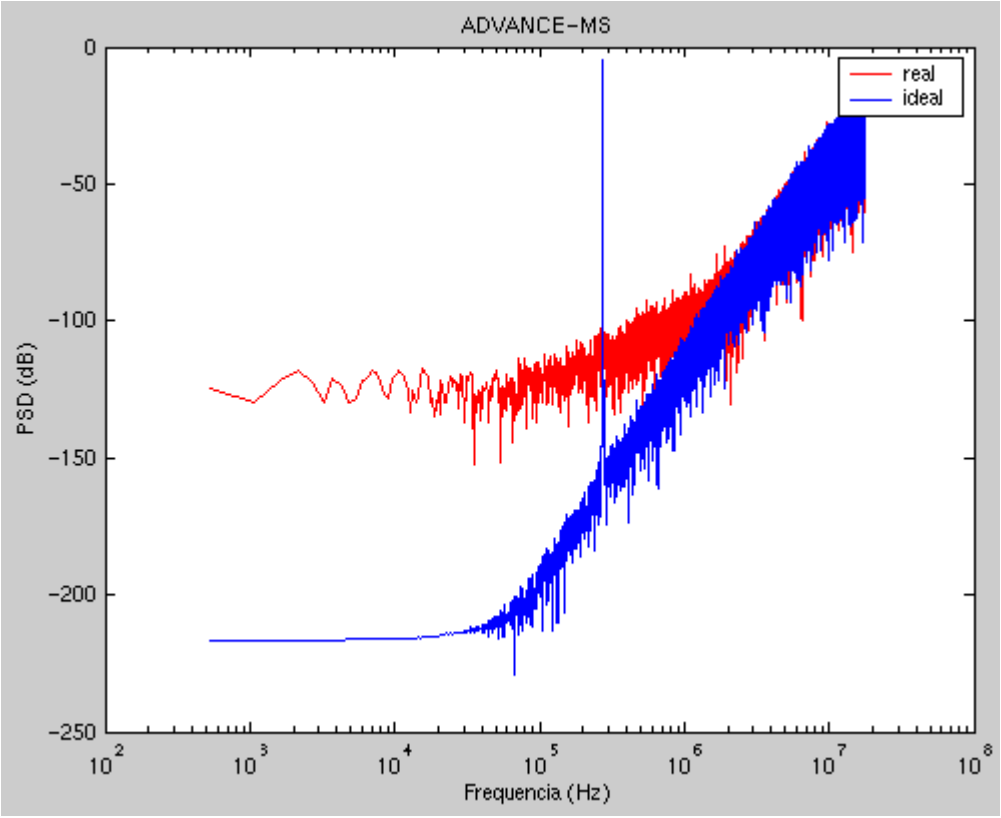



**Figura 4.101. Espetro real e ideal con algunos errores simulado con ADVANCE – MS.**

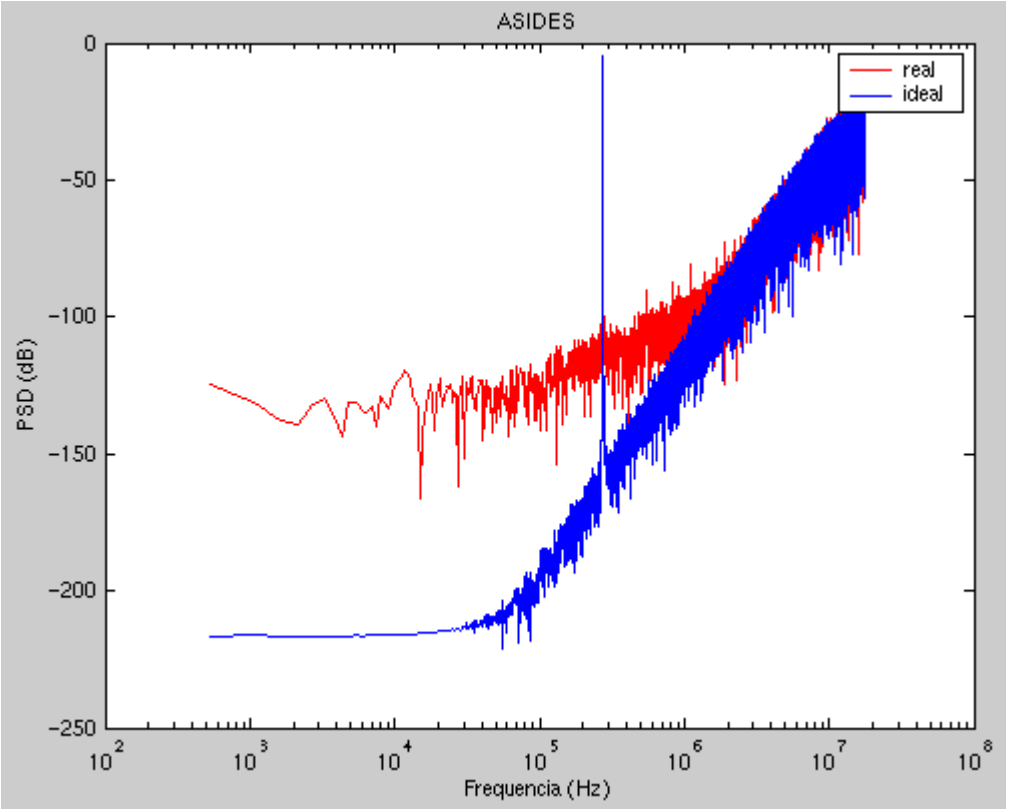



**Figura 4.102. Espectro real e ideal con varios errores simulado con ASIDES**

El tiempo que tarda en ejecutarse una simulación mediante ADVANCE – MS se puede dividir en los siguientes subtiempos:

- Tiempo de compilación y arranque: Es el tiempo que tarda en compilarse el fichero VHDL y en arrancar ADVANCE – MS en modo comando.
- Tiempo de simulación: Es el tiempo que tarda en simularse el número de muestras necesarias, que en este caso son 65536.
- Tiempo de procesado: Es el tiempo que tarda ADVANCE – MS en escribir las muestras en un fichero.
- Tiempo de cálculo de SNR: Es el tiempo que tarda en calcular la SNR del nodo que queramos tomar como salida
- Tiempo de cálculo de la densidad espectral de potencia: Tiempo que tarda en calcular la densidad espectral del nodo que hayamos tomado como salida.

En la Tabla 4.3 se muestra los tiempos anteriormente descritos para la arquitectura en cascada 2-1-1 y single-loop.

| Subtiempo | Cascada 2-1-1 | Single-loop segundo orden |
|---|---|---|
| Tiempo compilación y arranque | 7.5 | 7.5 |
| Tiempo simulación | 4 | 2.5 |
| Tiempo procesado | 11 | 3 |

| | | |
|---|---|---|
| Tiempo calculo SNR | 4 | 4 |
| Tiempo calculo densidad de potencia | 4 | 4 |

**Tabla 4.3. Tiempos de ADVANCE – MS**

En ASIDES no se puede separar el tiempo de simulación del de procesado. Para el cascada 2-1-1 tarda 3 segundos, mientras que epara el single-loop sale 1 segundo.

En SDTOOLBOX el tiempo de simulación es de 16 segundos para un cascada 2-1-1 y de 13 segundos para un single –loop de segundo orden. En esta herramienta no hace falta la compilación y el arranque de la herramienta consiste en arrancar SIMULINK dentro de MATLAB.

# APÉNDICE A VSIDES Manual de usuario

## A.1. Descripción de la herramienta

La Figura A.1 muestra un diagrama de bloques de VSIDES. Esta herramienta nos permite generar un fichero de salida para modelar el comportamiento de un modulador ΣΔ-SC en los lenguajes VERILOG HDL y VHDL. En estos lenguajes, se incluyen una biblioteca de bloques básicos afectador por errores debidos a sus implementaciones físicas. La Tabla A.1. recopila los bloques básicos fundamentales y las no idealidades cubiertas por el simulador. Éstas son incluidas en modelos asociados a cada bloque, con la posibilidad de definir tantos modelos como sean necesarios para la misma clase de bloques.

| Bloque Básico | | No idealidad | Consecuencias |
|---|---|---|---|
| Integrador | Amplificador Operacional | Ganancia finita y no lineal | Incremento de ruido de cuantización<br>Distorsión harmónica |
| | | Limitaciones dinámicas | Ruido de muestreo incompleto,<br>Distorsión harmónica |
| | | Rango de Salida | Sobrecarga,<br>Distorsión harmónica |
| | | Ruido Térmico | Ruido blanco |
| | Llaves | Resistencia ON<br>Ruido Térmico | Ruido de muestreo incompleto,<br>Ruido Blanco |
| | Capacidades | No linealidad<br>Desapareamiento | Incremento de ruido de cuantización<br>Distorsión harmónica |
| Comparador | | Histéresis,<br>FOCET | Incremento de ruido de cuantización |
| Cuantizador ,<br>Convertidor Digital/Analógico | | Error de offset,<br>Error de ganancia,<br>No linealidad | Incremento de ruido de cuantización<br>Distorsión harmónica |
| Sumador,<br>Amplificador | | No linealidad,<br>Error de ganancia,<br>Ruido Térmico | Distorsión harmónica<br>Incremento de ruido de cuantización,<br>Ruido Blanco |

**Tabla A.1. Principales bloques básicos y no idealidades**

El fichero de entrada a VSIDES comienza describiendo un netlist, donde se establece la topología y las no idealidades a tener en cuenta. Este generara un fichero para VHDL o VERILOG que se simulará en el dominio del tiempo usando descripciones funcionales de bloques básicos.

El fichero VHDL o VERILOG generado contendrá las instancias de estos bloques básicos descritos para ambos lenguajes. Estas instancias tienen como entradas aparte de las descritas para cada bloque, los distintos parámetros asociados a cada modelo de bloque. Esto es así debido a la necesidad de variar los parámetros durante el transcurso de la simulación.

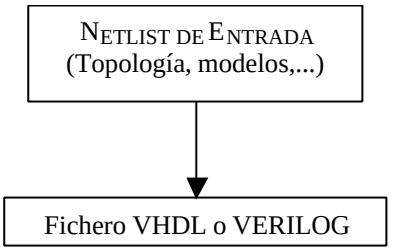



**Figura A.1. Diagrama de bloques de VSIDES**

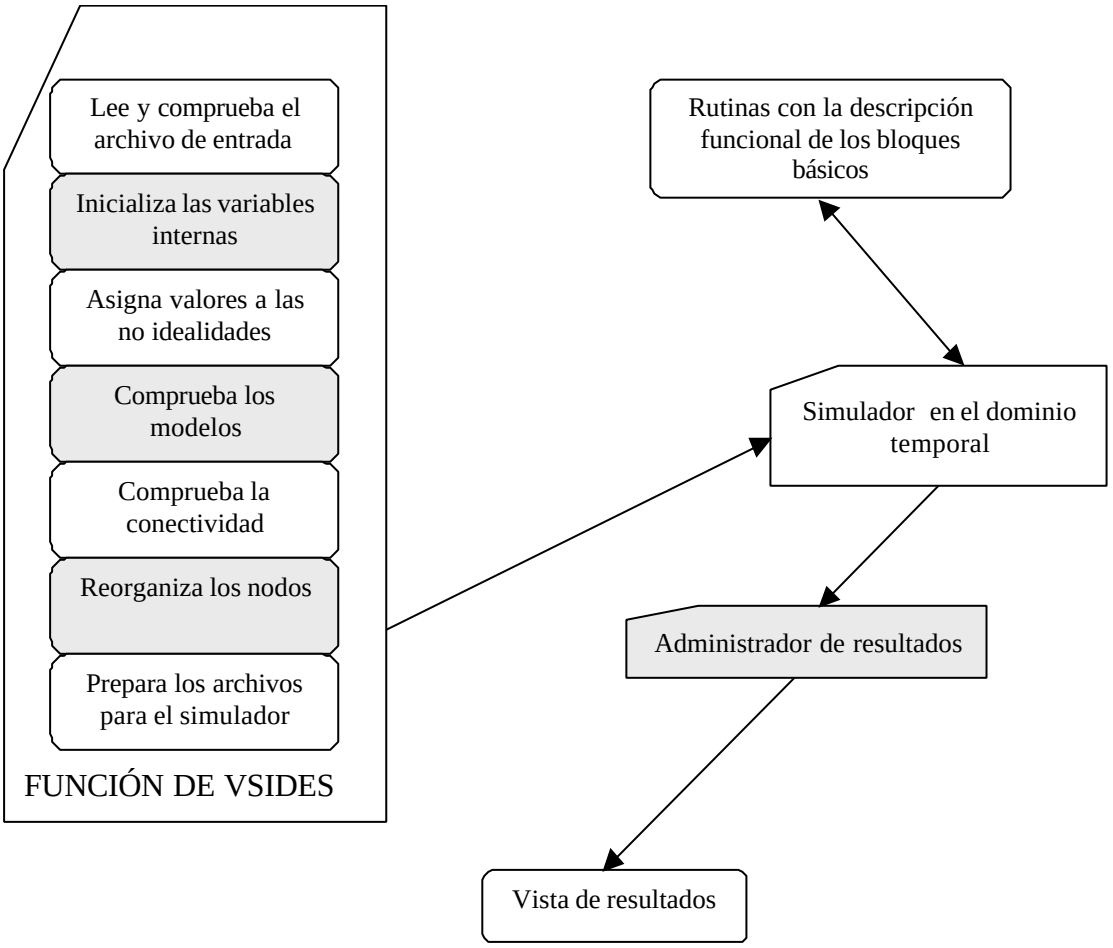



**Figura A.2. Diagrama de flujo de VSIDES**

La organización interna de la herramienta es simple y está ilustrada en el diagrama de flujo de la figura A.2. Todos los bloques básicos son representados a través de rutinas que describen su funcionalidad. Dichas rutinas son invocadas de acuerdo a la conectividad expresada por el usuario y actualizan los voltajes de los nodos de salida de los bloques acordes al voltaje de los nodos de entrada. Su estado interno y las no idealidades están incluidas en el modelo asociado.

# A.2. Manual de usuario

## A.2.1. Cuestiones sintácticas

Los particulares comandos del programa se obtienen a través de un fichero de entrada con la extensión ".par" cuyo contenido será cubierto en la siguiente sección. La línea de comando para ejecutar el programa es:

*% vsides nombre_de_archivo –formato*

siendo formato *vhd* o *v* según sea VHDL o VERILOG el fichero de salida.

Sigue a continuación, una lista de cuestiones sintácticas generales:

- Una expresión precedida de "#" será interpretada como un comentario
- Para reducir el número de palabras reservadas, se distingue sólo el uso de minúsculas
- Sólo se utilizarán cadenas para incluir los nodos de los circuitos. Por ejemplo xout, xin, out1. La palabra *"gnd"* se utilizará para referirse al nodo de tierra.

## A.2.2. Formato de entrada

El formato del fichero de entrada ha sido organizado como sigue

- Definición de topología
- Definición de reloj
- Sentencias de control
- Definiciones de modelos
- Definición de parámetros

### A.2.2.1. Definición de topología

Esta sección describe cómo hacer una instancia de cada bloque básico para definir la arquitectura del modulador

*A.2.2.1.1. Generador de señal*

Al menos debe haber un generador de señal incluido en el fichero para que sea un fichero válido de entrada. El formato para el generador de señal es:

*fuente nombre nodo_de_salida <ampl.=> valor_de_amplitud valor_de_frecuencia ;*

o bien,

fuente nombre nodo_de_salida <ampl.=>nombre_de_amplitud valor_de_frecuencia;

El segundo caso activa la posibilidad de variar la amplitud mediante un incremento definido posteriormente.

Ejemplos:

*fuente vin ampl = 0.5 1220.703125;*
*fuente entrada ampl = AMP 0.275e6;*
*fuente entrada1 1.0 0.5e6;*

*A.2.2.1.2. Integrador*

La sintaxis para el bloque del integrador difiere según sea de 1, 2 o 3 ramas:

*integrador1 nombre nodo_salida nodo_entrada+ nodo_entrada- ganancia <modelo nombre_modelo>;*

*integrador2 nombre nodo_salida nodo_entrada1+ nodo_entrada1- ganancia nodo_entrada2+ nodo_entrada2- ganancia <modelo nombre_modelo>;*

*integrador3 nombre nodo_salida nodo_entrada1+ nodo_entrada1- ganancia nodo_entrada2+ nodo_entrada2- ganancia nodo_entrada3+ nodo_entrada3- ganancia <modelo nombre_modelo>;*

El modelo del integrador será explicado posteriormente. El nombre del integrador debe comenzar con una letra seguida de cualquier combinación de letras y números. Los nodos de salida y entrada les ocurren lo mismo a excepción de palabras reservadas en los leguajes VHDL y VERILOG como *out* e *in*

Ejemplos:

*Integrador1 oi1 vin1 gnd 0.25;*
*Integrador3 oi3 oi1 gnd 0.375 oi1 out1 0.375 oi1 out2 0.25 modelo int_mod;*

*A.2.2.1.3. Comparador*

Su formato es,

*comparador nombre nodo_salida nodo_entrada <modelo nombre_modelo>;*

Sólo se permite un nodo de entrada al comparador.

Ejemplos:

*comparador comp1 cout cin;*
*comparador comp2 salida entrada modelo comp_mod;*

*A.2.2.1.4. Cuantizador*

Su formato es:

*cuantizador nombre nodo_salida nodo_entrada numero_bits <modelo nombre_modelo>;*

Este estandarizado caso de comparador se incluye para permitir la simulación de arquitecturas $\Sigma\Delta$ multi-bit. Como en los casos anteriores el modelo del cuantizador real se explicará en apartados posteriores.

Ejemplos:

*cuantizador cuant1 qout qin 3;*
*cuantizador cuant2 salida entrada 5 modelo cuant_mod;*

*A.2.2.1.5. Convertidor DA*

Su formato es,

*convertidor nombre nodo_salida nodo_entrada numero_bits <modelo nombre_modelo>;*

Sólo se permite un nodo de entrada al bloque. El modelo asociado se explicará en apartados posteriores.

Ejemplos:
*convertidor conv1 cv1 cv2 3;*
*convertidor conv2 salida entrada 5 modelo conv_mod;*

*A.2.2.1.6. Amplificador*

Su formato es:

*amplificador nombre nodo_salida nodo_entrada ganancia <modelo nombre_modelo>;*

El modelo asociado se explicará en apartados posteriores.

Ejemplos:

*amplificador amp1 xout xin 0.2;*
*amplificador amp2 salida entrada 0.1 modelo ampl_mod;*

*A.2.2.1.7. Sumador*

Su formato es:

*Sumador nombre nodo_salida nodo_entrada1 ganancia nodo_entrada2 ganancia <modelo nombre_modelo>;*

El modelo asociado se explicara en apartados posteriores

Ejemplos:

*sumador sum1 xout xin1 0.1 xin2 0.2;*
*sumador sum2 salida entrada1 0.5 entrada2 1 modelo sum_mod;*

*A.2.2.1.8. Multiplicador*

Su formato es:

*multiplicador nombre nodo_salida nodo_entrada1 ganancia nodo_entrada2 ganancia;*

Este bloque no tiene modelo asociado, sólo está implementado el modelo ideal.

Ejemplos:

*Multiplicador mul1 salida ent1 0.2 ent2 2.0;*

*A.2.2.1.9. Retraso*

Su formato es:

*retraso nombre nodo_de_salida nodo_de_entrada;*

Ejemplos:
*retraso ret1 xout xin;*

**A.2.2.2. Definición de reloj**

La sentencia para la definición del reloj es:

*reloj frec = número/ nombre <dc = número/ nombre ird = número/ nombre srd = número/ nombre>;*

*frec* es la frecuencia de muestreo. Se puede dar el valor o un nombre para la definición posterior del parámetro. *dc* es el duty cycle que por defecto es cero. *ird* y *srd* sirven para indicar una reducción de los tiempos de integración y muestreo, por lo que la reducción efectiva de la fase de muestreo es y la de la fase de integración son:

*Tmuestreo=1/2*Frecuencia_muestreo –srd*
*Tintegración=1/2*Frecuencia_muestreo-ird*

Obviamente, estas reducciones pueden ser muy importantes para altas frecuencias de operación. Por defecto los valores de ambos parámetros son cero.

Ejemplos:

*reloj frec = 2.5e6 dc = 0.2 ird = 0.01;*
*reloj frec =FS ird =IRD1;*

**A.2.2.3. Sentencias de control:**

Es necesaria al menos una sentencia para definir las tensiones de salida de los DACs y de los comparadores o cuantizadores. Ambas en un principio pueden ser diferentes, si bien, se tomarán las mismas para todos los elementos idénticos.

El formato es:

alimentación tensión_máxima_analógica tensión_mínima_analógica tensión_máxima_digital tensión_minima_digital;

Ejemplos:

*alimentación 2 –2 2 –2;*
*alimentación 1.5 –1.5 2 –2;*

También es necesaria una sentencia de salida para definir el nodo en el que se va a tomar la salida. Sólo se puede poner un único nodo de salida. Existen tres tipos de análisis que se pueden realizar.

- Análisis transitorio: Da la secuencia temporal de cada una de las muestras del nodo que se haya tomado como salida. No se permiten análisis paramétricos con este tipo de salida,

- SNR: Da la SNR respecto del nodo que se haya tomado como salida. Este tipo de análisis es el único que se permite con análisis paramétricos como se verá posteriormente.

- PSD: Da como resultado el espectro de potencia del nodo que se haya tomado como salida. No se permiten análisis paramétricos con este tipo de salida.

El formato es:

*output tipo_análisis nodo_de_salida;*

donde tipo_análisis puede ser *temp*, *snr* o *psd* según queramos la secuencia de salida temporal, o la snr, o el espectro de potencia

Ejemplo:

*output temp salida;*
*output snr salida2;*

Es necesario especificar también el número de puntos que se requieren para el análisis así como la razón de sobremuestreo. Estas dos magnitudes tienen que ser potencia de dos. Su formato es:

*npuntos numero_de_puntos tasa_sobremuestreo;*

Ejemplo:

*npuntos 65536 256;*

### A.2.2.4. Definiciones de modelos

Las no idealidades de los bloques son definidas por la sentencia

modelo tipo_elemento nombre_modelo par1= valor/ nombre par2= valor/ nombre ... pari= valor/ nombre/;

donde el nombre_de_modelo es el nombre de modelo especificado en la línea correspondiente del bloque. El tipo de bloque para ser modelado puede ser cualquiera de los siguientes: *Integrador, Comparador, Sumador, Cuantizador, DAC, Amplificador*.

Lo que sigue son las asignaciones de los parámetros del modelo con el formato: parámetro = valor, o bien, parámetro =nombre, donde nombre se define en una sentencia del tipo param. Los parámetros por defecto en los diversos bloques aparecen en la Tabla A.2.

El parámetro *clo* del modelo del integrador se refiere sólo a la carga capacitiva en la salida del integrador debido sólo a las siguientes contribuciones:

- Capacidad parásita del op-amp
- Capacidad parásita de integración
- Capacidad parásita del comparador/ cuantizador

Una posible carga resultante de la conexión de la salida del integrador a otro integrador es automáticamente computada por el parser, por lo que no debe ser incluida en *clo*.

El sumador o amplificador puede ser usado para simular un amplificador de tensiones por lo que permite la definición de la curva entrada/salida (error de ganancia y error de no linealidad) y ruido térmico.

La salida de un cuantizador multi-bit puede ser directamente conectada a la señal de realimentación. En este caso se supondrá que hay un DAC ideal para convertir la señal digital de la salida del cuantizador.

### A.2.2.5. Definición de parámetros

La sentencia param sirve para asignar un parámetro que aparece en una sentencia de modelo. Su sintaxis es:

*Param nombre_de_parámetro= expresión;*

| Bloque | Nombre | Descripción | Valor por defecto |
|---|---|---|---|
| Integrador | imax | Máxima corriente de salida | Infinita |
| | Gm | Transconductancia | 1 mA/V |
| | Osp | Output swing + | Infinito |
| | osn | Output swing - | -Infinito |
| | a0 | Ganancia en DC | Infinita |
| | a01 | No linealidad de la ganancia de primer orden | 0 $V^{-1}$ |
| | a02 | No linealidad de la ganancia de segundo orden | 0 $V^{-2}$ |
| | a03 | No linealidad de la ganancia de tercer orden | 0 $V^{-3}$ |
| | a04 | No linealidad de la ganancia de cuarto orden | 0 $V^{-4}$ |
| | c2 | Capacidad de realimentación | 2 pF |
| | alpha | No linealidad del condensador de primer orden | 0 $V^{-1}$ |
| | beta | No linealidad del condensador de segundo orden | 0 $V^{-2}$ |
| | cp2 | Capacidad parásita de entrada | 0 pF |
| | cp11, cp12, cp13 | Capacidades parásitas de los condensadores de muestreo | 0 pF |
| | clo | Capacidad de carga | 0 pF |
| | ron | Resistencia de llave | 0 Ω |
| | inpsd | R.M.S. valor del ruido térmico equivalente a la entrada del OPAMP | 0 V/ $Hz^{-1/2}$ |
| | sigma | Desviación estándar del peso del integrador | 0 |
| | int_type | Tipo de integrador, 0 ideal, 1 real | 0 |
| Comparador | voff | Tensión de offset | 0 V |
| | vhis | Tensión de histéresis | 0 V |
| | htype | Tipo de histéresis<br>0= determinista, 1= aleatoria | 0 |
| Cuantizador/ DAC | eoff | Offset | 0 FS |
| | egain | Error de ganancia | 0 FS |
| | enl | No linealidad integral | 0 LSB |
| Amplificador / Sumador | osp | Output swing + | Infinito |
| | osn | Output swing - | - Infinito |
| | gerr | Error relativo de ganancia | 0 |
| | gnl1 | No linealidad de ganancia de primer orden | 0 $V^{-1}$ |
| | gnl2 | No linealidad de ganancia de primer orden | 0 $V^{-2}$ |
| | onoise | R.M.S. valor del ruido térmico a la salida | 0 V/ $Hz^{-1/2}$ |

**Tabla A.2. Parámetros de modelo**

La expresión puede ser definida de tres formas distintas:

*a) Una constante*

param GB =1e-3;

*b) Una sentencia sweep con el formato*

*param nombre_de_parámetro = sweep( lin número_de_puntos_totales(-1) valor_inicial valor_final);*

*param nombre_de_parámetro = sweep( dec número_de_puntos_por_decada valor_inicial valor_final);*

Ejemplos:

*param gb =sweep(dec 10 10 10000);*

Esto produce una variación del parámetro gb desde 10 a 10000 de forma logarítmica con 10 puntos por década.

*param gb2=sweep(lin 9 10 100);*

Esto produce una variación del parámetro *gb2* desde 10 a 100 con 10 puntos, o sea un incremento de 10.

El parser generará el fichero VHDL o VERILOG correspondiente para generar una columna de resultados temporales correspondiente a cada uno de los valores del parámetro. Esto servirá para obtención del SNR.

*c) Una sentencia de parámetros aleatorios con el formato*

*param nombre_de_parámetro = gauss( media desviación_tipica);*

*param nombre_de_parámetro = uniforme (valor_inicial valor_final);*

Ejemplos:

*Param inl = gauss (5 0.1);*

Esto produce una variación del parámetro *inl* de forma gaussiana con una media de 5 y desviación estándar de 0.1.

*d) Sentencia monte:* Sirve para definir el número de simulaciones de Montecarlo que se van a realizar

*monte numero*;

Ejemplos:

*monte = 10*;

Esto hace que se tomen 10 valores de los parámetros que se hayan tomado como aleatorios.

## A.2.3. Formato de Salida

La información de salida del parser está organizada en algunos ficheros. El nombre de cada fichero es formado con el nombre del archivo de entrada sin la extensión ".par" y una de las siguientes extensiones.

- *".do"* Contiene un fichero que se ejecutará al arrancar el compilador correspondiente con el tiempo necesario para simular todos las muestras que se le especifiquen al programa.

- *".vhd"* Contiene el fichero VHDL con todas las instancias necesarias y el proceso adecuado para hacer análisis paramétricos.

- *".v"* Contiene el fichero VERILOG con todas las instancias necesarias y el proceso adecuado para hacer análisis paramétricos.

- *"sin extensión"* Contiene un ejecutable que compila el fichero correspondiente VHDL o VERILOG y arranca la herramienta de simulación correspondiente, ya sea ADVANCE-MS o MODELSIM y simula el número de muestras necesarias descritas en el fichero *".do"*.

## A.2.4. Mensajes de error

Durante la ejecución del parser pueden aparecer algunos mensajes de error. Estos son explicados a continuación:

- *Formato de sintaxis incorrecto*: Se debe establecer la sintaxis correcta en la sentencia de ejecución del parser.

- *No se pudo abrir el fichero*: El fichero de entrada al parser no se ha encontrado.

- *Opción incorrecta*: Las opciones son –v o –vhd según se quiera el fichero de salida en VERILOG o VHDL.

- *Formato de fichero de entrada inválido*: tiene que tener la extensión "*.par*"

- *Ningún bloque en el fichero*: No se ha encontrado ningún elemento en el fichero de entrada

- *Frecuencia de muestreo no puede ser parámetro en sentencia MONTE o SWEEP*: El valor del reloj debe ser un parámetro constante en los lenguajes VHDL y VERILOG y no se puede cambiar durante la simulación.

- *Falta sentencia de alimentación*: No se ha encontrado ninguna sentencia de alimentación.

- *Sentencia de alimentación repetida*: Se ha encontrado más de una sentencia de alimentación y no se sabe cuál es la correcta.

- *Sentencia de reloj repetida*: Se ha encontrado mas de una sentencia de reloj y no se sabe cuál es la correcta.

- *Falta sentencia de reloj*: No se ha encontrado ninguna sentencia con el reloj.

- *Frecuencia de muestreo debe ser asignada*: En toda sentencia de reloj debe aparecer el valor de la frecuencia.

- *Falta sentencia de número de puntos*: Al menos debe haber una sentencia con el número de muestras a sacar

- *Sentencia de número de puntos repetida*: Se ha encontrado más de una sentencia con el número de muestras a calcular y no se sabe cuál es la correcta.

- *Falta sentencia output*: No se ha encontrado ninguna sentencia con el nodo de salida.

- *Sentencia output repetida*: Se ha encontrado más de una sentencia de salida y no se sabe cuál es la correcta.

- *Sentencia Sweep o Monte repetida*: sólo puede haber una sentencia del tipo sweep o monte por fichero. Sólo se puede variar un parámetro.

- *Nodo flotante*: Existe un nodo de entrada a un bloque que no está conectado a ningún otro elemento.

- *No hay correspondencia con ningún parámetro en sentencia de modelo*: Se encontró definido un parámetro en una sentencia param que no se ha definido en una sentencia de modelo

- *No hay correspondencia con ningún parámetro en sentencia param*: Se ha definido un parámetro en una sentencia de modelo que no se le ha dado ningún valor en una sentencia param.

- *No se encontró correspondencia con modelo*: No se ha encontrado ningún modelo asociado al definido en la sentencia del bloque.

- *Tipo de parámetro no se corresponde con modelo de bloque*: Ese parámetro no tiene correspondencia en el modelo de dicho bloque

- *Nodo de salida duplicado*: Se han encontrado dos o más bloques con el mismo nodo de salida.

- *Etiqueta duplicada*: Se han encontrado dos modelos o dos instancias de bloque con la misma etiqueta.

- *Nodo de sentencia output, no se corresponde con ningún nodo del circuito*: No hay ningún nodo que se corresponda con el nodo de la sentencia output.

### A.2.5. Ejemplo completo

Presentaremos un ejemplo de un modulador single-bit con dos integradores

```
#Modulador sigma-delta single loop

#Sentencias de control
alimentacion 1.5 -1.5 1.5 -1.5;
nmuestras 65536 256;
output snr Xout;
reloj frec=2.5e6;

#Fuente de entrada
fuente vin Xin 0.5 1220.703125;

#Estructura del modulador
integrador1 int1 oi1 Xin xout 0.25 modelo int;
integrador2 int2 oi2 oi1 gnd 0.25 oi1 xout 0.25 modelo int;
comparador comp1 Xout oi2 modelo comp;

#Modelos de los distintos elementos
modelo integrador1 int cp11=1.5e-12 cp12=1.5e-12 cp13=1.5e-12 cp2=1.5e-12
                    clo=1.0e-12 ron=700 gm=GM imax=IMAX a0=1000 inpsd=1.6e-9
modelo comparador comp htype=1 vhis=30e-3 voff=30e-3;

#Definición de parámetros
param GM=4.2e-3;
#param GM=monte 10 gauss(4.2e-3 0.5e-3);
          # A sustituir si se quiere una análisis de montecarlo de la transconductancia
          # con media 4.2mA/V y desviación típica 0.5mA/V
param IMAX=1.5e-3;
```

# A.3. Ampliación de nuevos modelos

En este apartado se explicará como incluir nuevos modelos de forma que VSIDES los reconozca. En la Figura A.3 se encuentra los pasos a seguir para modificarlo.

Trabajaremos sobre un ejemplo concreto. Supongamos que se quiere añadir un bloque llamado *Rectificador* con dos nodos de entrada y uno de salida en cuyo modelo asociado se pueden distinguir los parámetros Vsat e Imax, cuyos valores ideales van a ser infinito. La instancia asociada a VSIDES sería,

*rectificador etiqueta nodo_salida nodo_entrada1 nodo_entrada2 Tensión_salida <modelo nombre_modelo>;*

y la instancia asociada al modelo sería,

*modelo rectificador nombre_modelo vsat = número/etiqueta imax = número/ etiqueta;*

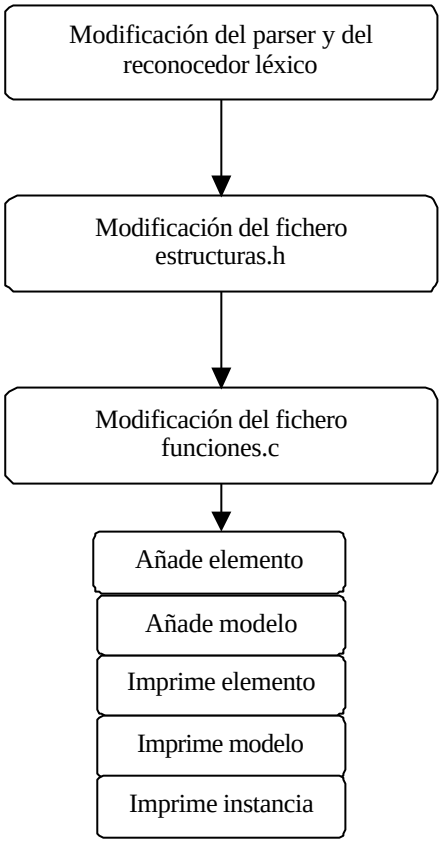



**Figura A.3. Diagrama de flujo para insertar un nuevo bloque**

## A.3.2. Modificación del parser VSIDES y del reconocedor léxico

Se necesitan seguir los siguientes pasos:

- En primer lugar, se necesitan incluir en el reconocedor léxico (**lexer.l**), las nuevas palabras claves. En nuestro ejemplo serán *rectificador, vsat e imax*. Las palabras claves asociadas a parámetros de segundo orden deben pasarle al parser su valor como cadena de caracteres.

```
rectificador {yylval.caracter='t';
   return RECTIFICADOR;}
vsat {yylval.nombre=strdup(yytext);
   return vsat;}
imax {yylval.nombre=strdup(yytext);
   return imax;}
```

- En segundo lugar, se necesitan incluir los token necesarios en el parser (**parser.y**) correspondientes a las nuevas palabras claves, que serán en este caso *RECTIFICADOR, VSAT e IMAX*. Los token asociados a parámetros de segundo orden deben tener como valor asociado <*nombre*>.

```
%token <nombre> VSAT
%token <nombre> IMAX
%token <caracter> RECTIFICADOR
```

- En la regla "*parámetro*" se deben incluir los token asociados a parámetros, en nuestro ejemplo, *VSAT e IMAX*.

```
parametro : VOFF | VHIS | HTYPE | A0 | A01 | A02 | A03 | A04 | CP2 | CP11
| CP12 | CP13 | SIGMA | ALPHA | BETA | RON | GM | INPSD | GERR | GNL1 | GNL2
| ONOISE | EOFF | EGAIN | ENL | CLO | C2 | IMAX | OSP | OSN | INT_TYPE  | IRD
| DC | SRD | JITTER | FREC | VSAT | IMAX;
```

- En la regla "*elemento*" se debe incluir el token asociado al bloque, que en nuestro ejemplo es, *RECTIFICADOR.*

```
elemento : AMPLIFICADOR | SUMADOR | COMPARADOR | CUANTIZADOR |
CONVERTIDOR| INTEGRADOR1 | INTEGRADOR2 | RECTIFICADOR;
```

- En la regla "*sentencia*" se debe incluir un token llamado de la misma manera que el bloque que queremos incluir, pero en minúsculas. En nuestro ejemplo sería, *rectificador*, cuyo token hay que definirlo como sigue:

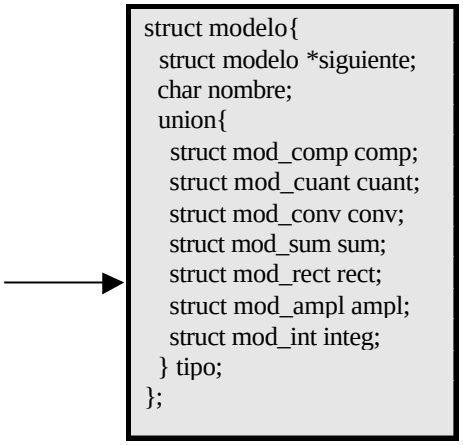
```
rectificador: RECTIFICADOR  NOMBRE nodo nodo nodo numero ';'{
anadir_elemento('t',$2,$3,$4,$5,NULL,NULL,NULL,NULL,NULL,$6,0,0);
}
|          RECTIFICADOR NOMBRE nodo nodo nodo numero MODELO NOMBRE ';' {
anadir_elemento('t',$2,$3,$4,$5,NULL,NULL,NULL,NULL,$8,$6,0,0);
}
;
```

La función anadir_elemento viene explicada en el apéndice C. En este caso se ha elegido como carácter asociado al elemento la letra '*t*'.

## A.3.3. Modificación del fichero estructuras.h

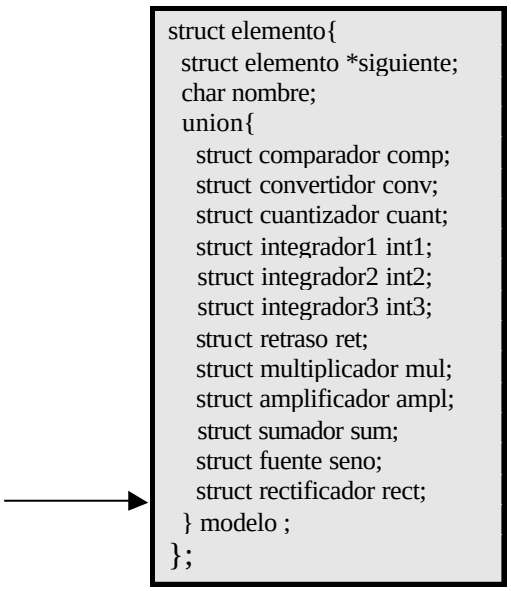
Hay que seguir los siguientes pasos:

- En primer lugar en el fichero *estructuras.h* se debe incluir un nuevo campo en la unión de la estructura *struct modelo*, que corresponda a la estructura asociada al nuevo modelo. En nuestro ejemplo será *struct mod_rect rect*:



```
struct modelo{
  struct modelo *siguiente;
  char nombre;
  union{
    struct mod_comp comp;
    struct mod_cuant cuant;
    struct mod_conv conv;
    struct mod_sum sum;
    struct mod_rect rect;
    struct mod_ampl ampl;
    struct mod_int integ;
  } tipo;
};
```

- De la misma manera se debe incluir un nuevo campo en la unión de la estructura elemento que corresponda a la estructura asociada al nuevo bloque. En nuestro ejemplo será *struct rectificador rect*:



```
struct elemento{
  struct elemento *siguiente;
  char nombre;
  union{
    struct comparador comp;
    struct convertidor conv;
    struct cuantizador cuant;
    struct integrador1 int1;
    struct integrador2 int2;
    struct integrador3 int3;
    struct retraso ret;
    struct multiplicador mul;
    struct amplificador ampl;
    struct sumador sum;
    struct fuente seno;
    struct rectificador rect;
  } modelo ;
};
```

- En tercer lugar en el fichero *estructuras.h* hay que definir dos nuevas estructuras correspondientes a la instancia y al modelo. La estructura asociada a la instancia del bloque debe tener como campos el nombre de la instancia y el nombre del modelo correspondiente, el nodo de salida y los nodos de entrada. También debe contener un campo para el valor de *Vsat* y un puntero a struct modelo.

```
struct rectificador{
        char *nombre;
        char *salida;
        char *entrada1;
        char *entrada2;
        double Vsat;
        struct modelo *mod;
        char *nombre_mod;
};
```

- La estructura asociada al modelo debe contener un campo como nombre de modelo y un *struct nombre* por cada parámetro de segundo orden del modelo.

```
struct mod_rect{
  char *nombre;
  struct nombre Vsat;
  struct nombre Imax;
};
```

## A.3.4. Modificación del fichero funciones.c

Habrá que realizar los siguientes pasos:

- En primer lugar, se procederá a la inserción de una variable global denominada rect y la inicializaremos a 0.

- En este fichero se tienen que modificar las siguientes funciones:

  - anadir_elemento();
  - anadir_modelo();
  - imprime_elemento();
  - imprime_modelo();
  - imprime_instancia();

### A.3.4.1. anadir_elemento

En esta función, habrá que añadir un nuevo caso en el *"switch"* correspondiente al nuevo elemento que queremos añadir. Se procederán a añadir los diversos parámetros que se le han pasado a esta función al puntero a la estructura *rectificador*. Seguidamente se añadirán a las funciones *anadir_nodo* y *anadir_lista* los nodos de salida y entrada respectivamente. Finalmente se llamara a la función *anadir_modelo*, si el parser no encontró ningún modelo asociado. El ejemplo correspondiente sería:

```
case 't' :{
  ultimo->nombre='t';
  ultimo->modelo.rect.nombre=nombre;
  ultimo->modelo.rect.salida=salida;
  ultimo->modelo.rect.entrada1=ent1;
  ultimo->modelo.rect.entrada2=ent2;
  ultimo->modelo.rect.vsal=peso1;
  anadir_nodo(&sal,salida);
  anadir_lista(&ent,ent1);
  anadir_lista(&ent,ent2);
  if(nmod!=NULL)
    ultimo->modelo.rect.nombre_mod=nmod;
  else{
    ultimo->modelo.rect.nombre_mod="defrect";
    if(!rect) anadir_modelo('t',"defrect");
    rect++;}
  break;   }
```

### A.3.4.2. anadir_modelo

En esta función habrá que añadir un nuevo caso en el *"switch"*, Se procederán a añadir el nombre y los valores por defecto de los parámetros asociados, que en nuestro caso son 0, a través de la función rellena_nombre. A continuación se añadirán los parámetros definidos en la instancia del modelo si los hay. En nuestro ejemplo quedaría:

```
  case 't' :{
    ult_mod->nombre='t';
    ult_mod->tipo.rect.nombre=nombre;
    rellena_nombre(&ult_mod->tipo.rect.vsat,0);
    rellena_nombre(&ult_mod->tipo.rect.imax,0);
    if(strcmp(nombre,"defrect"))
      for(upar=ppar;upar!=NULL;upar=upar->siguiente)
        if(!strcmp(upar->nombre,"vsat"))
          rellena_par(&ult_mod->tipo.rect.vsat);
        else if (!strcmp(upar->nombre,"imax"))
            rellena_par(&ult_mod->tipo.rect.imax);
           else{
             fprintf(stderr,"%d: Tipo de parametro (%s) no corresponde
            con modelo de rectificador\n",lineno,upar->nombre);
             exit(1);
            }
    borrar_lista(&ppar);
    break;
}
```

### A.3.4.3. imprime_elemento

En esta función habrá que añadirle un nuevo caso al *"switch"*. Se pretende imprimir cada uno de los nodos, tanto de salida como de entrada, pertenecientes al nuevo elemento. Para ello se la pasará a la función *imprime_nodo* cada uno de los nodos correspondientes al elemento. En nuestro ejemplo quedaría:

```
      case 't' :{
        imprimir_nodo(busca->modelo.rect.salida);
        imprimir_nodo(busca->modelo.rect.entrada1);
        imprimir_nodo(busca->modelo.rect.entrada2);
        break;
       }
```

### A.3.4.4. imprime_modelo

En esta función habrá que añadirle un nuevo caso al switch. Se utilizará la función *imprimir_para*m que viene explicada en el apéndice C. A esta función hay que pasarle los campos de la estructura *rectificador, nombre* y los correspondientes a cada uno de los parámetros de la siguiente forma:

```
    case 't':{
      imprimir_param("SIGNAL",bmodelo->tipo.rect.nombre,"_vsat",bmodelo-
>tipo.rect.vsat.valor,bmodelo->tipo.rect.vsat.nombre);
      imprimir_param("SIGNAL",bmodelo->tipo.rect.nombre,"_imax",bmodelo-
>tipo.rect.imax.valor,bmodelo->tipo.rect.imax.nombre);
      break;
    }
```

### A.3.4.5. Imprime_instancia

Tendremos que saber cómo hemos programado el modelo de *rectificador* en ambos lenguajes de programación para que el parser lo haga directamente. En primer lugar veamos como sería una instancia tanto en VHDL como en VERILOG del *rectificador*.

**VHDL**

```
nombre_instancia: ENTITY work.rectificador(funcion)
        GENERIC MAP (valor_vsal)
        PORT MAP (Reset,  nodo_entrada, nodo_salida, vsat, imax);
```

**VERILOG**

```
RectificadorM #(valor_vsal) nombre_instancia
        (Reset,  nodo_entrada, nodo_salida, vsat, imax);
```

En esta función habrá que añadirle un nuevo caso al switch. Se utilizarán una serie de instrucciones *fprintf*, que imprima la instancia para los distintos lenguajes y la función imprime_par, que se explicará en el apéndice C.

```
   case 't':{
      if(!tipo)
        fprintf(yyout,"\t%s: ENTITY work.rectificador(funcion)\n\t\tGENERIC MAP",busca-
>modelo.rect.nombre); /*Imprimir instancia en VHDL */
      else
        fprintf(yyout,"\tamplificadorM #"); /*Imprimir instancia en VERILOG */
      fprintf(yyout,"(%.10e)\n",busca->modelo.rect.mod->tipo.rect.vsal);
      if(!tipo)
        fprintf(yyout,"\t\tPORT MAP");
      else
        fprintf(yyout,"\t\t%s ",busca->modelo.rect.nombre);
      fprintf(yyout,"(Reset,%s,%s",busca->modelo.rect.entrada,busca->modelo.rect.salida);
    imprime_par(busca->modelo.rect.nombre_mod,"_vsat",busca->modelo.rect.mod-
>tipo.rect.vsat.nombre);
    imprime_par(busca->modelo.rect.nombre_mod,"_imax",busca->modelo.rect.mod-
>tipo.rect.imax.nombre);
    fprintf(yyout,");\n");
    break;
   }
```

# APÉNDICE B Generador de esquemáticos

## B.1. Descripción de la herramienta

La entrada de diseño basado en esquemáticos está limitada a describir un sistema en términos de una estructura creada por la interconexión de bloques básicos primitivos. Un modelo completo de HDL, puede ser instanciado en la descripción de otros diseños, al igual que en un esquemático se puede instanciar un símbolo para cada componente. La herramienta que se ha utilizado para la interconexión de los distintos bloques mediante captura de esquemáticos es el HDL – DESIGNER de Mentor.

Debido a las limitaciones de ambos lenguajes para variar los parámetros que se le pasan a cada uno de los bloques funcionales, los modelos reales de los bloques básicos del modulador fueron necesarios diseñarlos con más entradas de las necesarias, una entrada adicional por cada parámetro asociado al modelo real del bloque. En el caso especial de los integradores, serían además necesarias salidas adicionales correspondientes a las tensiones de los condensadores de muestreo. Estas tensiones de los condensadores de muestreo tendrían que conectarse como entradas al integrador anterior al que estuviera unido.

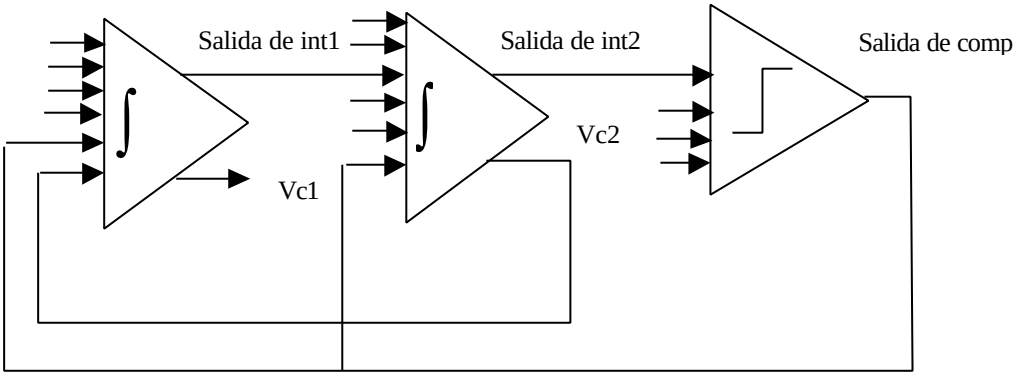



**Figura B.4. Interconexión de un modulador de lazo simple de segundo orden**

El símbolo generado para cada bloque, tendría las entradas y salidas necesarias más todas aquellas entradas asociadas a los parámetros. La interconexión de estos elementos para dar lugar a una arquitectura ΣΔ válida, daría como resultado un diagrama un tanto engorroso como se puede ver en la Figura B.4.

Para resolver esta dificultad, se ha propuesto una librería de bloques básicos alternativos, en la que sólo se tuviesen las entradas y salidas normales de cada bloque y el resto se pudiesen insertar como parámetros, de forma que la interconexión de estos elementos resultase más sencilla, como se puede ver en la Figura B.5.

Como se ha dicho antes los parámetros del modelo se tenían que introducir como entradas adicionales de los bloques, por lo que además ha sido necesaria la creación de un *"parser"* para poder transformar el fichero VHDL con las instancias de los bloques alternativos, a un fichero VHDL o VERILOG con las instancias de los bloques reales.

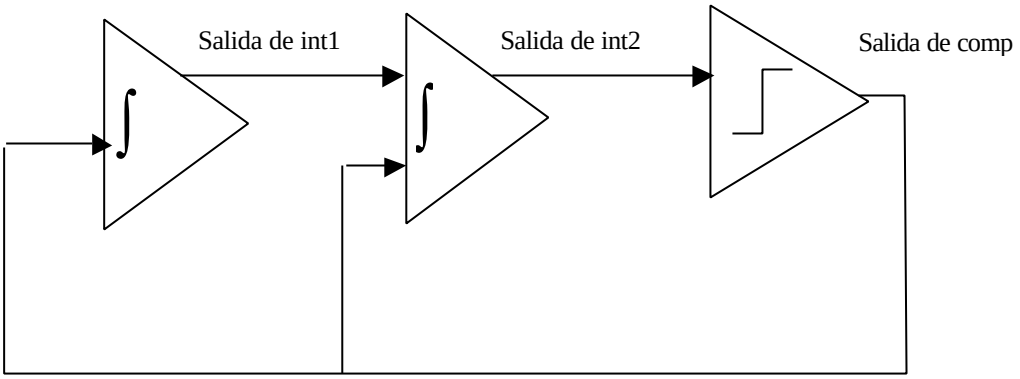



**Figura B.5. Interconexión de un modulador de lazo simple de segundo orden**

# B.2. Manual de Usuario

En la Figura B.6 se muestra un diagrama del proceso que hay que seguir para realizar una simulación. Este proceso consiste en los siguientes pasos:

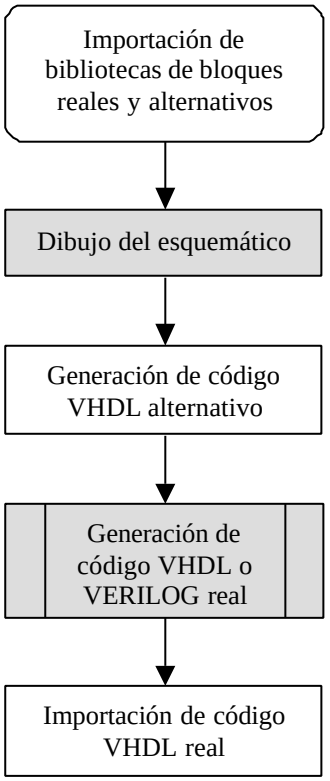



**Figura B.6. Diagrama de flujo de la herramienta**

- En primer lugar, es necesario importar las bibliotecas de bloques alternativos y las bibliotecas de bloques reales. Una vez hecho esto, no será necesario hacerlo más veces.

- En segundo lugar, se procederá a dibujar el circuito que queramos simular, con los bloques alternativos y a introducir todos los valores de los parámetros de los modelos.

- En tercer lugar, se generará el código VHDL correspondiente que se le pasará como entrada al parser SCHEMATIC.

- Por último se importará el código del fichero VHDL o VERILOG correspondiente a los bloques reales.

## B.2.2. Importación de bibliotecas de bloques reales y alternativos

En primer lugar se pulsará el botón de la ventana principal del programa (Figura B.7)  para mostrar el *HDL import Wizard*. Nos aparecerá una ventana como la de la Figura B.8.

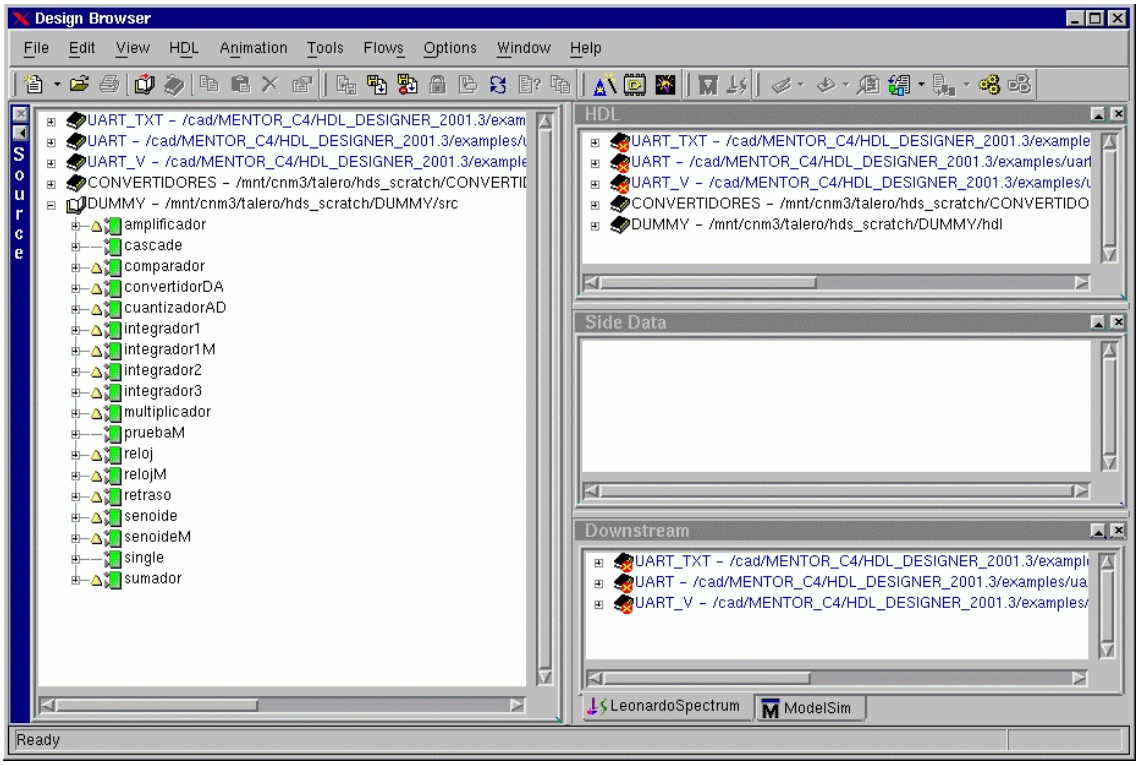
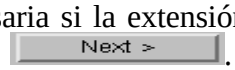



**Figura B.7. Ventana principal del programa**

Se seleccionará la opción *Specify HDL files*. La opción del lenguaje requerido sólo es necesaria si la extensión de los archivos es de un tipo distinto a *.vhd* o *.v*. Se pulsará luego 

.

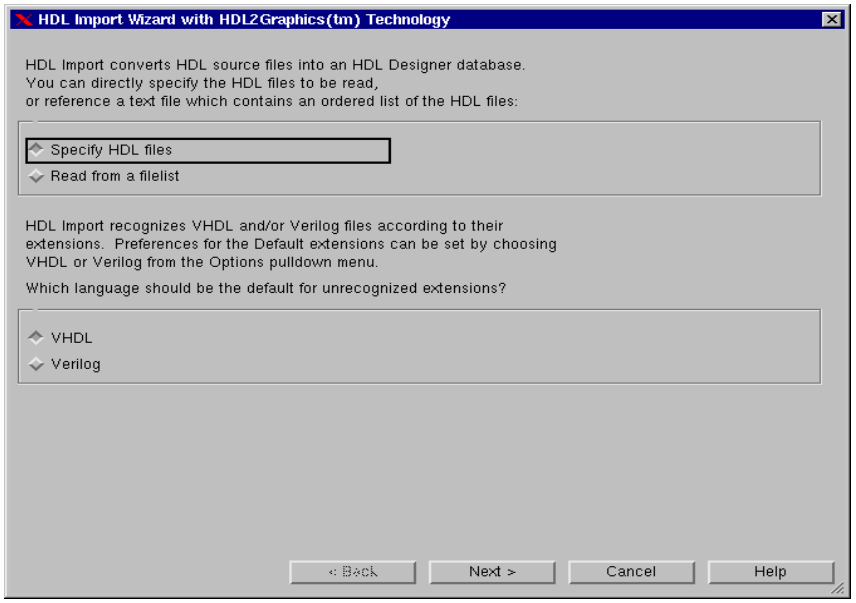



**Figura B.8. Primera ventana del HDL- Import Wizard**

En la segunda ventana (Figura B.9), usaremos el botón 

para localizar el directorio donde tenemos los archivos del código fuente de los bloques alternativos. Se nos mostrará una lista con todos los archivos de ese directorio.

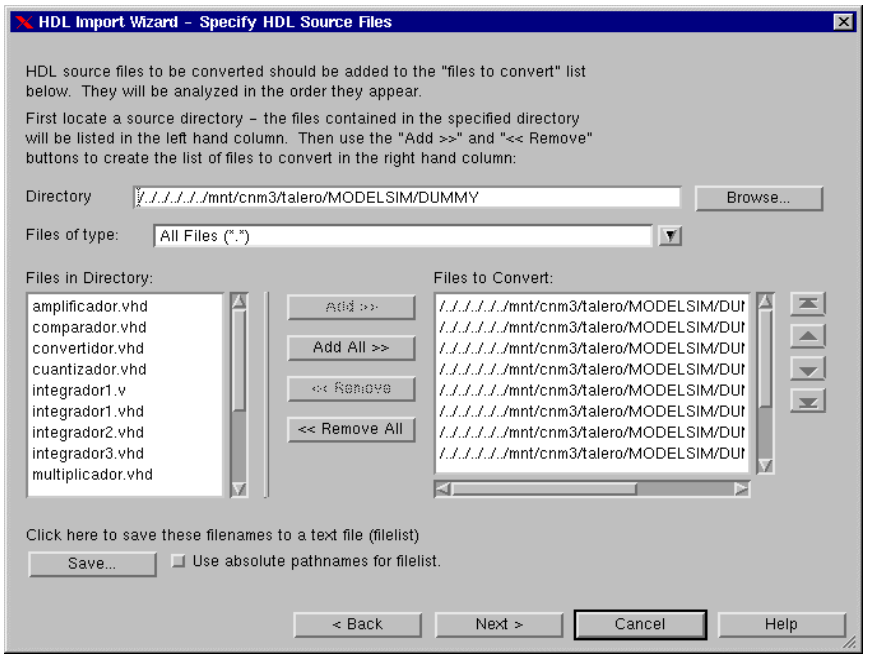



**Figura B.9. Segunda ventana del HDL - Import Wizard**

Pulsaremos el botón 

para añadir todos los ficheros que queramos importar a la lista de ficheros a convertir. Pulsaremos luego el botón 

En la tercera ventana (Figura B.10), nos permite escoger de una lista todas las unidades de diseño que hayan sido reconocidas como código HDL. Escogeremos la opción de importar todas las unidades y pulsaremos Next >

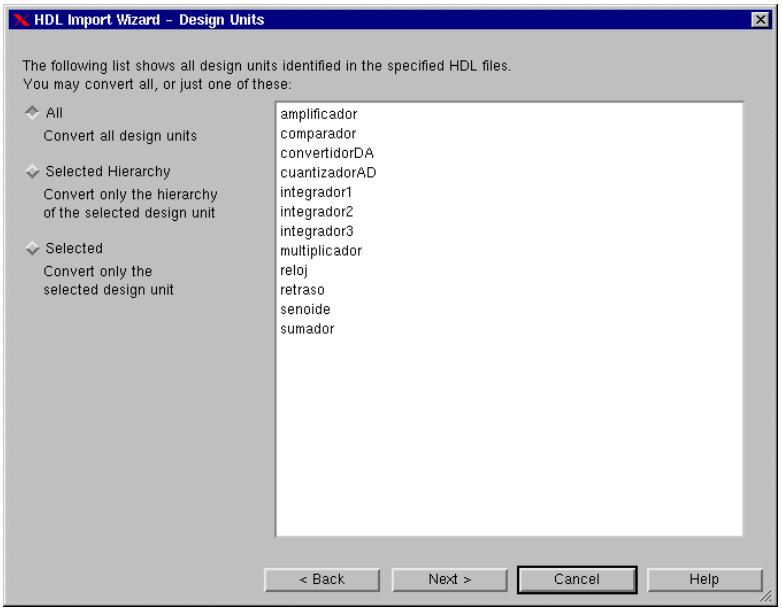



**Figura B.10. Tercera ventana del HDL- Import Wizard**

La cuarta ventana (Figura B.11), nos permite especificar la biblioteca donde asociaremos los distintos bloques, que para los bloques alternativos, la llamaremos *DUMMY* y *CONVERTIDORES*, para los bloques reales.

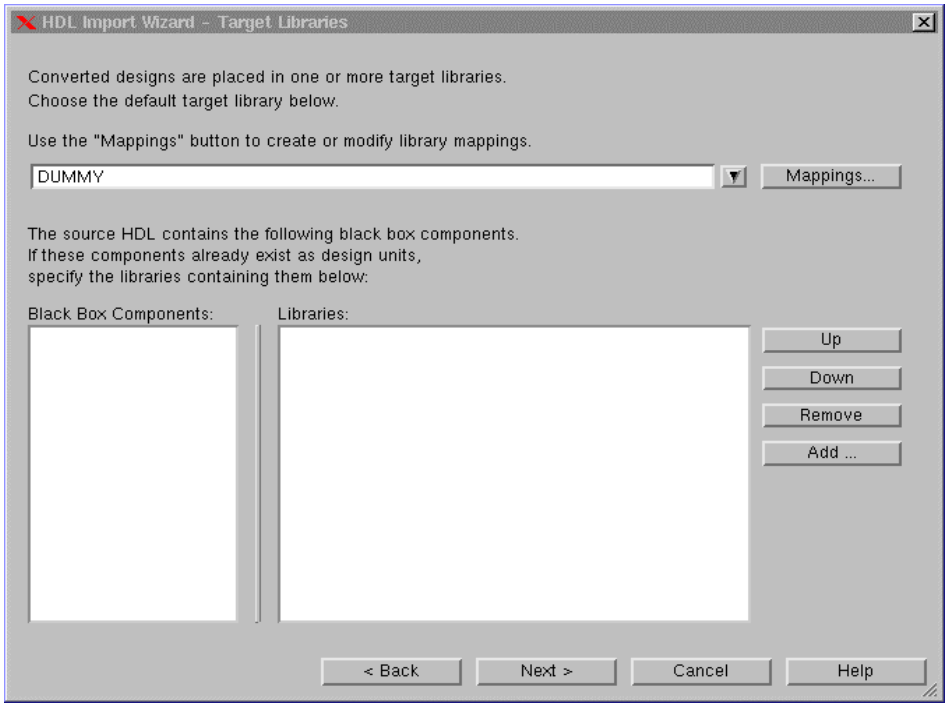



**Figura B.11. Cuarta ventana del HDL - Import Wizard**

La quinta ventana (Figura B.12), sirve para especificar el formato con el que queremos importar el diseño. En el apartado *Hierarchy Descriptions* se escogerá *HDL text* y en el *leaf-level descriptions*, *HDL text*. Las otras opciones servirán si queremos importar el código como diagrama de bloques o diagrama de flujo.

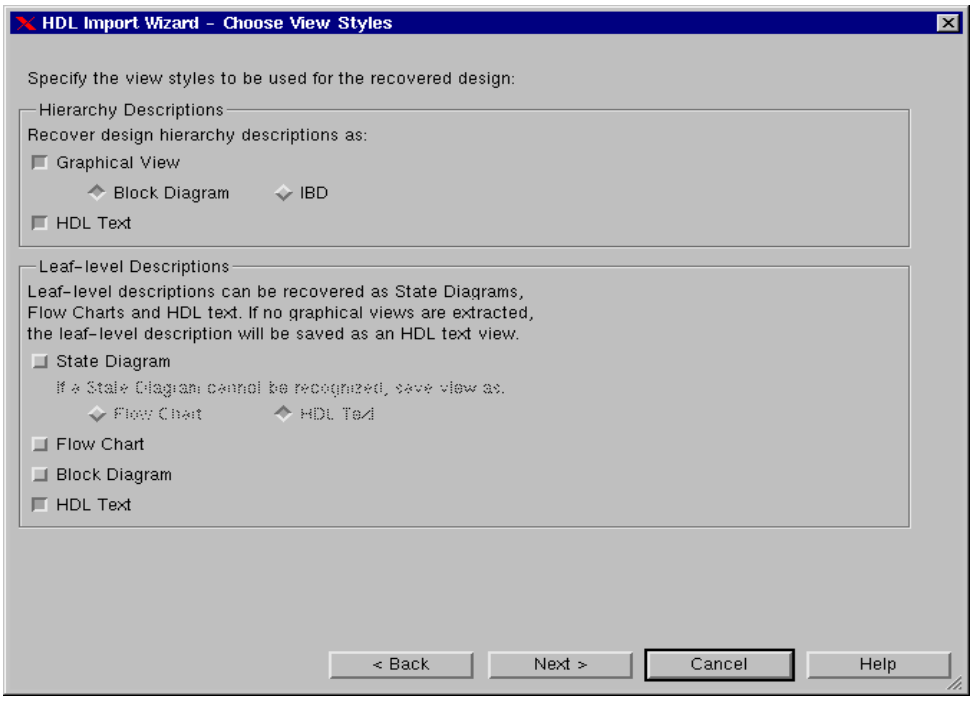



**Figura B.12. Quinta ventana del HDL - Import Wizard**

Las sexta y última ventana (Figura B.13), sirve para confirmar las opciones anteriores. Finalmente pulsaremos Finish para terminar con la importación.

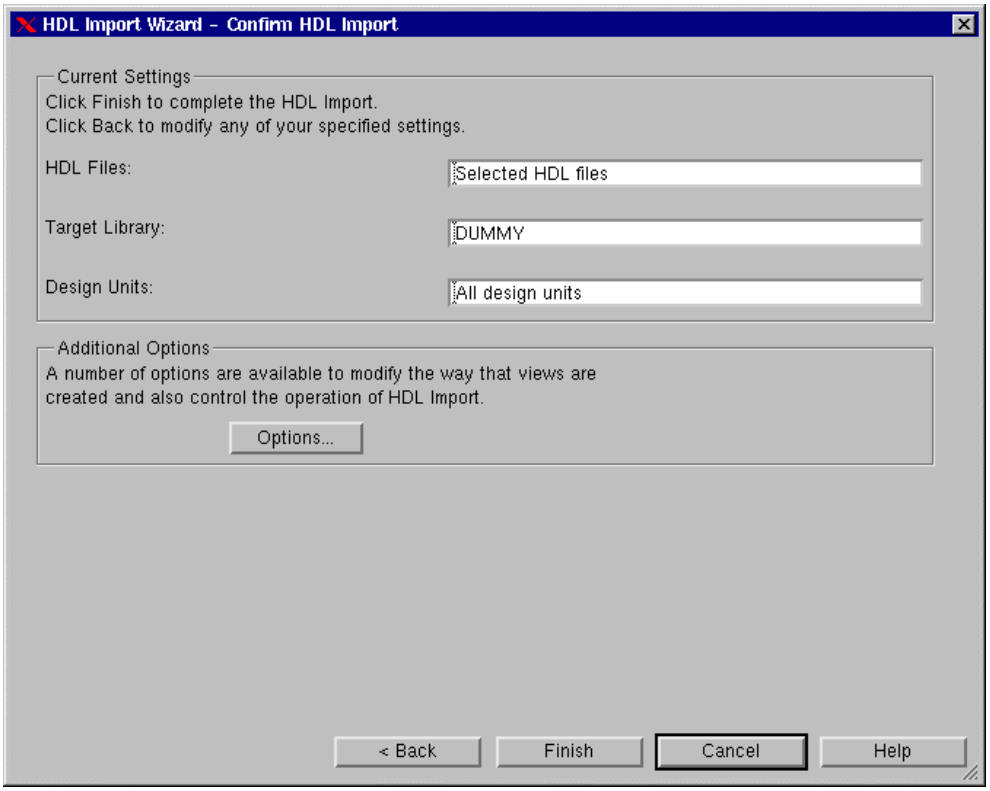



**Figura B.13. Sexta ventana del HDL - Import Wizard**

## B.2.3. Dibujo del esquemático

Una vez importados los bloques correspondientes a los distintos elementos, nos queda la siguiente vista en el *Design Browser* (Figura B.14).

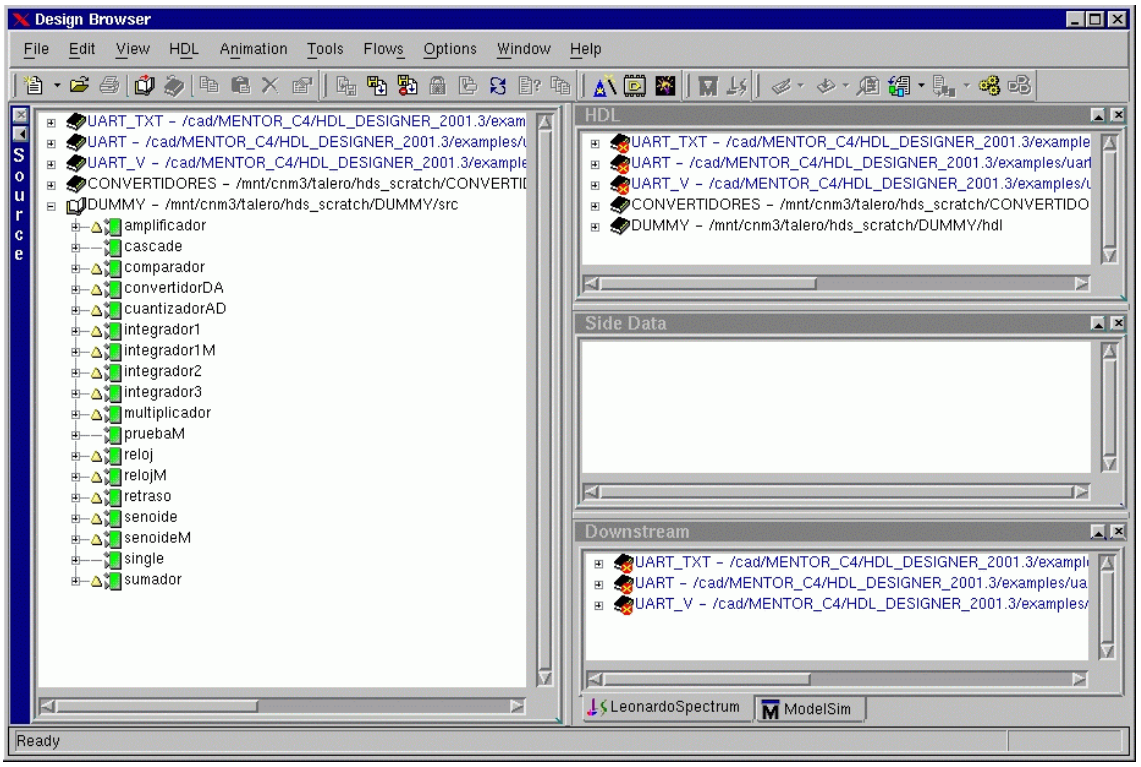



**Figura B.14. Ventana principal del HDL – DESIGNER**

Para dibujar una nueva arquitectura de modulador se escogerá del menú *File*, la opción *new- block diagram*. En la barra de herramientas escogeremos la opción de añadir componente  y nos saldrá una pantalla como sigue.

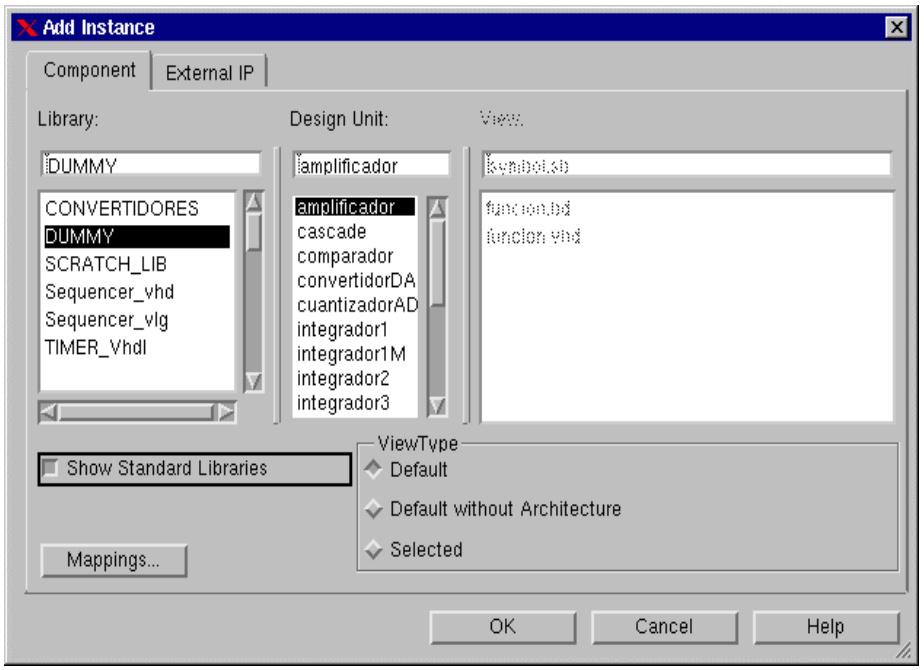



**Figura B.15. Ventana de esquemático**

Se seleccionará un elemento de la biblioteca *DUMMY* y se insertará en el esquemático. Para insertar otro tipo de elemento pulsaremos la tecla escape y realizaremos la misma operación.

Si algunos de los elementos tienen algunas de sus entradas conectadas a tierra, es necesario añadir un cable mediante el botón de la barra de herramientas  Se pulsará el botón derecho del ratón sobre el cable y se escogerá la opción *object properties*. En la pestaña *signals* se procederá a cambiar el nombre de la señal a *"gnd"*. Aunque no es necesario se podrían cambiar los nombres de los demás nodos del circuito de la misma manera.

Para modificar los parámetros de los elementos se pulsará el botón derecho del ratón y se escogerá la opción *object properties*. Nos saldrá un menú como el de la Figura B.16. y escogeremos la pestaña *generics*.

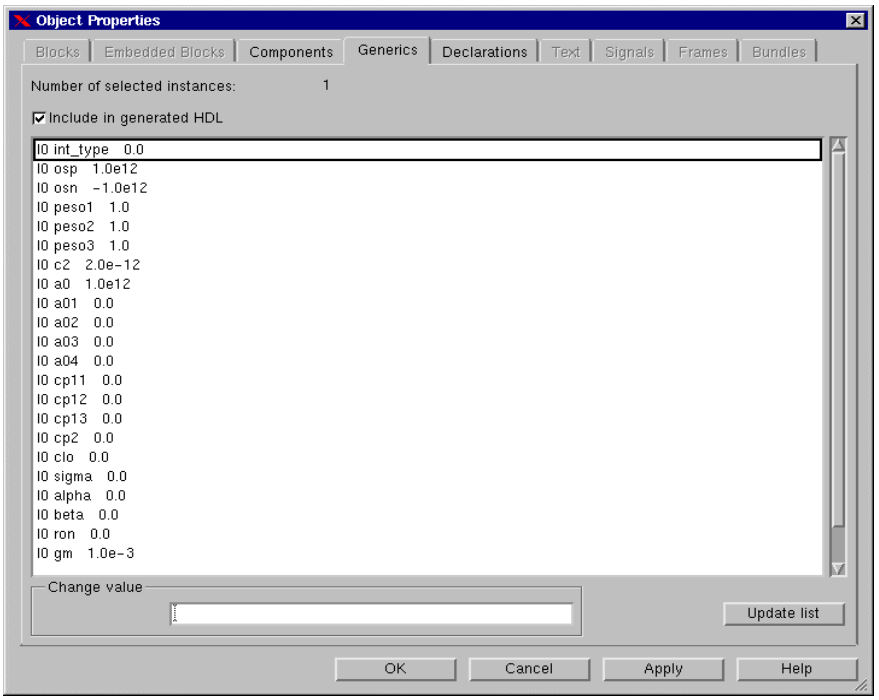



**Figura B.16. Ventana para cambiar las propiedades de los objetos**

Es obligatorio insertar un elemento llamado reloj en cualquier esquemático que vaya conectado a la entrada de la fuente. Esto es debido a que el parser SCHEMATIC necesita encontrar una instancia de reloj para generar el fichero VHDL con los bloques reales correspondientes. Todos los parámetros que se pueden encontrar en los modelos se corresponden con los dados en la Tabla A.1. A estos hay que añadirles los parámetros del elemento reloj que se pueden ver en la Tabla B.3.

| Bloque | Nombre | Descripción | Valor por defecto |
|---|---|---|---|
| Reloj | frecuencia | Frecuencia de muestreo | 2.5 MHz |
| | dc | Duty cycle | 0.5 |
| | ird | Reducción del tiempo de integración | 0 |
| | srd | Reducción del tiempo de muestreo | 0 |

**Tabla B.3. Parámetros de Reloj**

En la siguiente figura se puede ver el esquemático de un modulador single-loop de segundo orden.

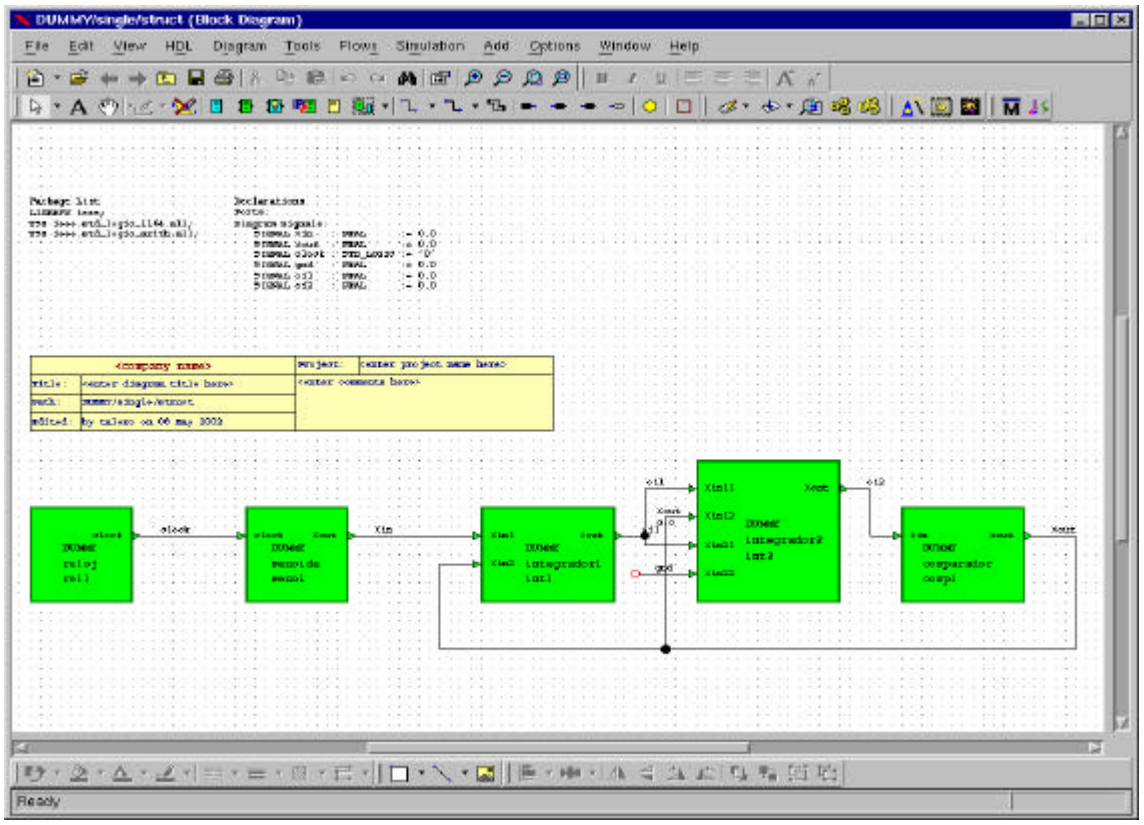



**Figura B.17. Modulador de lazo único de segundo orden**

## B.2.4. Funcionamiento del parser schematic

Primero tendremos que generar un fichero VHDL correspondiente a dicho esquemático. En la barra de herramientas se pulsará el botón o bien, se pulsará el botón derecho del ratón sobre el nombre del esquemático y se escogerá la opción *HDL - Generate - Single Level*.

A continuación pulsaremos el botón de la barra de herramientas o bien, se pulsará el botón derecho del ratón sobre el nombre del esquemático y se escogerá la opción *HDL - View generated HDL*. El archivo lo guardaremos en nuestro directorio de trabajo con extensión *.vhd*.

Para generar el fichero con el código VHDL O VERILOG real utilizaremos la siguiente línea de comando.

*% schematic nombre_de_archivo –formato*

Siendo formato *vhd* o *v* según sea VHDL o VERILOG el fichero final.

El fichero de salida será uno con el mismo nombre del de entrada pero con extensión *.par*. Será un fichero del tipo que se le pasa a VSIDES. Al ejecutar el programa se le preguntará al usuario cuál nodo entre todos los posibles, que coincide con los nombres que tienen los nodos del esquemático, quiere escoger como nodo de salida. Para ello se mostrará una lista con todos los nodos posibles. Seguidamente se preguntará el tipo de análisis que se quiere realizar entre los tres posibles: temporal, espectro de salida y SNR. Finalmente se preguntará al usuario cuántas muestras quiere generar de la salida y la razón de sobremuestreo.

Si continuamos con el ejemplo de la Figura B.17. la ejecución de schematic produciría lo siguiente, donde se ha escogido un número de muestras de 65536 y una razón de sobremuestreo de 256.

```
% schematic single_struct.vhd –vhd
Xout     oi2      Xin      oi1      gnd

Escoge un node de salida entre los anteriores: Xout
Tipo de análisis
0   Analisis temporal
1    Espectro de potencia
2    SNR
Número de muestras a generar: 65536
Tasa de sobremuestreo : 256
%
```

A continuación utilizaremos VSIDES sobre el fichero *.par* generado, para dar lugar a un fichero VHDL o VERILOG válido.

Hay otra manera de hacer esto automáticamente que es mediante el fichero script *change*. A este programa se le pasaría como fichero de entrada el fichero VHDL procedente del esquemático y daría como salida un fichero VHDL o VERILOG con el mismo nombre que el fichero de entrada y con extensión *.vhd* o *.v* según sea el formato. La sintaxis es

*% change nombre_de_archivo(sin extensión) –formato*

Donde formato es *vhd* o *v* según sea el fichero de salida

## B.2.5. Importación del fichero real

Una vez que se tenga el fichero real se procederá a importarlo de la misma manera que se ha dicho en el apartado 2.1. En vez de escoger como biblioteca de destino DUMMY se escogerá como biblioteca de destino CONVERTIDORES.

Otra opción hubiera sido simularlo directamente con MODELSIM para VERILOG o ADVANCE-MS para VHDL con el fichero script generado por VSIDES.

# B.3. Importación de nuevos modelos

En este apartado se indicará como incluir nuevos modelos de manera que el *parser schematic* los reconozca y coincida con lo que se le ha añadido al *parser vsides*. Los pasos a seguir para añadir un nuevo elemento pueden verse en la Figura B.18.

Como se ha visto en el apartado anterior, para obtener un fichero VHDL que se pueda simular a partir de la biblioteca de esquemáticos, hay que seguir los pasos del diagrama de flujo de la Figura B.6. Para ello, es necesario utilizar VSIDES, para llegar al fichero final, de manera que habrá que añadir un nuevo bloque a VSIDES como se ha explicado en el apéndice A. Además, se necesitará crear un bloque ficticio, que sirva como símbolo del bloque real y que tendremos que importar para utilizarlo en el HDL – DESIGNER.

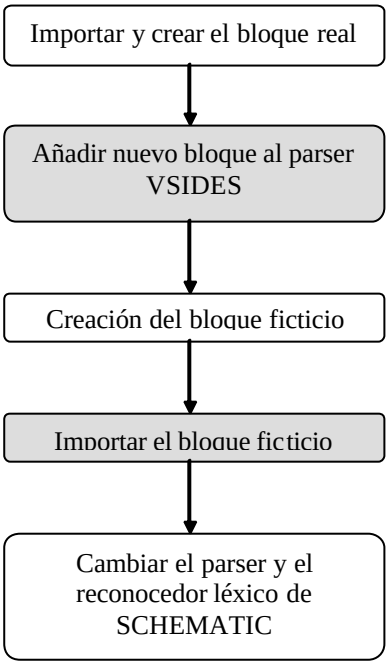



**Figura B.18. Diagrama de flujo para insertar un nuevo bloque**

## B.3.2. Importación de un bloque ficticio

Primero consideremos un ejemplo concreto como el mismo del apéndice A *rectificador*. Una vez programado el bloque en ambos lenguajes, se procederá a importarlo de la misma manera que en el apartado B.2.2. a la biblioteca CONVERTIDORES.

Se añadirá este nuevo elemento al parser *vsides* de la misma manera que se explica en el apéndice A.

Se programarán los bloques ficticios de forma que sus instancias se asemejen a las instancias de los bloques reales que para el ejemplo concreto eran del tipo,

**VHDL**

```
nombre_instancia: ENTITY work.rectificador(funcion)
        GENERIC MAP (valor_vsal)
        PORT MAP (Reset,  nodo_entrada, nodo_salida, vsat, imax);
```

**VERILOG**

```
RectificadorM #(valor_vsal) nombre_instancia
        (Reset,  nodo_entrada, nodo_salida, vsat, imax);
```

El bloque ficticio sería semejante al de VHDL, pero los parámetros del modelo que se le pasan como puertos de entrada a la entidad en la parte de *PORT MAP* se le pasarían como generics en la parte de *GENERIC MAP*. Estos GENERICS hay que inicializarlos al valor que tienen por defecto. La arquitectura del modelo quedaría vacía. La programación quedaría de la siguiente forma.

```
ENTITY rectificador IS
        GENERIC (vrect : REAL := 1.0;
                imax=1.0e12;
                vsat=1.0e12;
        PORT (  Xin1,Xin2: IN REAL;
                Xout : OUT REAL:= 0.0);
END  rectificador;
ARCHITECTURE funcion OF rectificador IS
BEGIN
END funcion;
```

A continuación si el bloque ficticio se programó fuera del HDL DESIGNER se procederá a importar el fichero de la misma manera que la descrita en el apartado B.2.2.

## B.3.3. Modificación del parser y del reconocedor léxico de SCHEMATIC

El parser de SCHEMATIC está preparado para que reconozca dentro de un fichero escrito en VHDL las instancias correspondientes a cada uno de los bloques básicos. El parser reconoce sintaxis distintas de las de VSIDES, aunque algunas reglas pueden ser parecidas a VSIDES. Para la modificación del parser y del reconocedor léxico de SCHEMATIC habrá que seguir estos pasos.

- En primer lugar, se necesitan incluir en el reconocedor léxico (**lexer2.l**), las nuevas palabras claves. En nuestro ejemplo será *rectificador*. Esta palabra clave tendrá que devolverle al parser un token llamado *RECTIFICADOR*.

```
rectificador {return RECTIFICADOR;}
```

- En segundo lugar, se necesitan incluir los token necesarios en el parser (**parser2.y**) correspondientes a las nuevas palabras claves, que serán en este caso *RECTIFICADOR*. Estos token asociados a cada uno de los elementos no tienen ningún valor asociado.

```
%token CUANTIZADOR
```

- En la regla "*elemento*" se deben incluir los token asociados a los nuevos modelos, en nuestro ejemplo, será RECTIFICADOR

```
elemento : SENOIDE | RELOJ | RETRASO | MULTIPLICADOR | AMPLIFICADOR | SUMADOR
| COMPARADOR | CUANTIZADOR | CONVERTIDOR | INTEGRADOR1 | INTEGRADOR2
| INTEGRADOR3 | RECTIFICADOR;
```

- En la regla "*instancia*" se debe incluir un token llamado de la misma manera que el bloque que queremos incluir, pero en minúsculas. En nuestro ejemplo sería, *rectificador*, cuyo token hay que definirlo como sigue

```
rectificador : NOMBRE ':' rect GENERIC MAP '(' parametro ',' parametro ',' parámetro ')'
PORT MAP '(' nodo ',' nodo ',' nodo ')' ';'
{fprintf(yyout,"rectificador %s %s %s %s %.10e modelo %s_mod;\n",$1,$22,$18
,$20,$7,$1);
fprintf(yyout,"modelo rectificador %s_mod vsat=%.10e imax=%.10e;\n",$1,$9,$11);
 anadir_lista($18);
 anadir_lista($20);
 anadir_lista($22);
}
```

- La función anadir_lista se explica en el Apéndice C. La regla asociada a cualquier bloque nuevo, en nuestro caso *rectificador*, necesita un token nuevo después de NOMBRE que se llame de la misma manera pero abreviada. Para cualquier bloque que añadamos hay que añadir una regla semejante que en este caso será:

```
rect: RECTIFICADOR
| ENTITY NOMBRE '.' RECTIFICADOR '(' NOMBRE ')'
;
```

# APÉNDICE C : Código C de VSIDES y de SCHEMATIC

## C.1. Código C de VSIDES

El parser *vsides* está compuesto por unas 1700 líneas de código C aproximadamente, más la parte del reconocedor léxico. Consta de las siguientes funciones,

- void genera_fichero_salida(char *, char *);
- void corresponde_modelo();
- void rellena_integradores();
- void anadir_elemento(char, char*, char *, char *, char *, char *, char *, char *, char *, char*, double, double, double);
- void anadir_nombre(struct lista **,char *);
- int anadir_lista(struct lista **,char *);
- void anadir_reloj();
- void anadir_modelo(char, char *);
- void anadir_parametro(char *, double, char *);
- void anadir_nodo(struct lista **, char *);
- int anadir_param();
- void rellena_monte( char *,int, int, int, int, int, int, double, double, double, double);
- void rellena_par(struct nombre *);
- void rellena_nombre(struct nombre *, double);
- void inicializa_modelo();
- void inicializa_elemento();
- void imprime_do_script();
- void imprime_fichero();
- void imprimir_nodo(char*);
- void imprimir_param(char *,char *, char *,double, char *);
- void imprime_par(char *,char *,char *);
- void imprime_monte();
- void imprime_cabecera();
- void imprime_elemento();
- void imprime_modelo()
- void imprime_instancias();
- void imprime_final();
- void borrar_lista(struct parametro **);
- main()

Estas funciones pasarán a explicarse a continuación,

## C.1.1. genera_fichero_salida

Esta función se le pasan como parámetros por valor, *fent*, que es el nombre del fichero de entrada de la línea de comando del parser y *formato,* que es el formato del fichero(*vhd* o *v*). Estos son los argumentos que se le pasan a la función *main* como *argv[1]* y *argv[2]*. En el caso de que el nombre del fichero de origen no sea válido, o bien, el formato no sea válido se terminará la ejecución del parser y se devolverá el control al sistema operativo.

Está función se encarga de generar el nombre de los ficheros de salida que son los siguientes.

- *".do"* Contiene un fichero que se ejecutará al arrancar el compilador correspondiente con el tiempo necesario para simular todos las muestras que se le especifiquen al programa.

- *".vhd"* Contiene el fichero VHDL con todas las instancias necesarias y el proceso adecuado para hacer análisis paramétricos

- *".v"* Contiene el fichero VERILOG con todas las instancias necesarias y el proceso adecuado para hacer análisis paramétricos

- *"sin extensión"* Contiene un ejecutable que compila el fichero correspondiente VHDL o VERILOG y arranca la herramienta de simulación correspondiente, ya sea ADVANCE-MS o MODELSIM y simula el número de muestras necesarias descritas en el fichero *".do"*.

  El código de la función es el siguiente

```
void genera_fichero_salida(char *fent,char *formato){
            int tam;
            int result;
            tam=strlen(fent)-4;
            if(!strcmp(fent+tam,".par")){
              if (!strcmp(formato,"-vhd")){
                 strncpy(outfile,fent,tam);
                 strcpy(outfile+tam,".vhd");
                 tipo=0;
              }
              else{
                if (!strcmp(formato,"-v")){
                   strncpy(outfile,fent,tam);
                   strcpy(outfile+tam,".v");
                   tipo=1;
                }
                else{
                  fprintf(stderr,"Opcion incorrecta:%s (-v,-vhd)\n",formato);
                  exit(1);
                }
              }
              strncpy(dofile,fent,tam);
              strcpy(dofile+tam,".do");
              strncpy(script,fent,tam);
              strcpy(script+tam,"");
            }
            else{
              fprintf(stderr,"Formato de fichero de entrada (%s) invalido
(.par)\n",fent);
              exit(1);
            }
}
```

## C.1.2. corresponde_modelo

Está función no toma ningún parámetro como argumento. Hace corresponder cada bloque del circuito a su modelo correspondiente de la sentencia del parser *vsides modelo*. En el caso de que al usuario se le haya olvidado definir el modelo, da error y se sale de la ejecución del programa devolviendo el control al sistema operativo.

Aquí tenemos parte del código de la función. Cada uno de los casos del *switch* es para cada uno de los tipos de bloques que se han incluido, esto es, integrador, comparador, CDA, cuantizador, y sumador. En este caso solo se muestra el caso para el integrador de tres ramas.

```
void corresponde_modelo(){
        int aux;
        struct modelo *busca_mod;
        struct elemento *busca_elem;
        for(busca_elem=primero;busca_elem!=NULL;busca_elem=busca_elem->siguiente){
          aux=1;
          switch(busca_elem->nombre){
            case '3' :{
              for(busca_mod=prim_mod;busca_mod!=NULL && aux;busca_mod=busca_mod->siguiente)
                if(busca_mod->nombre=='i'){
                  aux=strcmp(busca_elem->modelo.int3.nombre_mod,busca_mod->tipo.integ.nombre);
                  if(!aux)
                    busca_elem->modelo.int3.mod=busca_mod;
                }
              if(aux){
                fprintf(stderr,"No se encontro correspondencia con modelo %s\n",
busca_elem>modelo.int3.nombre_mod);
                exit(1);
              }
              break;
            }
          }
        }
}
```

## C.1.3. rellena_integradores

Esta función no tiene ningún parámetro que se le pase como argumento a la función. Busca en la lista de bloques del circuito todos los integradores que haya. Mira en cada integrador si tiene uno conectado a su salida y si es así, comprueba el valor de las capacidades de muestreo del siguiente integrador y se las introduce como parámetros al primero. También le asigna la tensión de los condensadores de muestreo del segundo integrador como entradas al primero. Para ello comprueba que algunos de los nodos de entrada del segundo integrador coincidan con el nodo de salida del primer integrador.

Aquí tenemos parte del código C correspondiente a esta función. Utiliza un puntero para localizar los integradores y otro para buscar los integradores conectados a él. Tiene que hacer esto para cada uno de los tipos de integradores, es decir, de una, dos o tres ramas.

```
void rellena_integradores(){
        struct elemento *busca1;
        struct elemento *busca2;
        for(busca1=primero;busca1!=NULL;busca1=busca1->siguiente)
          if(busca1->nombre=='1' || busca1->nombre=='2' || busca1->nombre=='3')
            for(busca2=primero;busca2!=NULL;busca2=busca2->siguiente){
              if(busca1->nombre=='1' && busca2->nombre=='1')
                if(strcmp(busca1->modelo.int1.nombre,busca2->modelo.int1.nombre))
                  if(!strcmp(busca1->modelo.int1.salida,busca2->modelo.int1.entrada1)
        | !strcmp(busca1->modelo.int1.salida,busca2->modelo.int1.entrada2)){
                    busca1->modelo.int1.vc1=busca2->modelo.int1.vcn;
                      busca1->modelo.int1.cn1=busca2->modelo.int1.mod->tipo.integ.c2*busca2-
>modelo.int1.peso;
          }
        }
    }
}
```

## C.1.4. anade_elemento

Esta función se le llama desde el parser cada vez que reconoce la sintaxis de un bloque. Se le llama con los siguientes parámetros

- *Tipo*: Es el tipo de elemento de entre todos los posibles: integrador de una, dos o tres ramas, comparador, DAC, cuantizador, sumador, amplificador, retraso, multiplicador, fuente.

- *Nombre*: Es el nombre de la instancia del elemento que se le ha pasado al parser

- Salida: Es el nombre del nodo de salida del elemento.

- *Entrada1*: Es el nombre de la entrada del elemento en el caso de que solo tenga una

- *Entrada2*: Es el nombre de la segunda entrada del elemento en el caso de que tenga dos. En el caso de que sólo tenga una este parámetro se pasará como NULL.

- *Entrada3,4,5,6*: En el caso de los integradores de dos y tres ramas tienen cuatro o seis entradas, coincidirán con los nombres de cada uno de los nodos y para los elementos que no tengan más entradas se pasarán como NULL.

- *Nmod*: Es el nombre asociado al modelo en la instancia del elemento. En el caso de que no la tuviera se pasaría como NULL

- *Peso1,2,3*: En el caso del integrador de tres ramas son los pesos de cada una de las ramas. Para los otros integradores, peso2 y 3 se pasan como cero. Lo mismo ocurre para el caso del sumador y el multiplicador, pero en este caso es peso3 el que se pasa como cero. Para el cuantizador y el DAC peso1 se pasa como el número de bits y los otros se pasan a cero. Para la fuente Peso 1 lleva el valor de la amplitud de la señal de entrada y Peso2 su frecuencia. Cuando no se utiliza Peso2 o Peso3 ambos parámetros se pasan como cero.

Esta función mete en una lista de *struct elemento* todos las instancias de elementos que localice e inicializa cada elemento con los valores que se le han pasado a la función. Cada elemento tiene un tipo de estructura, con todos los campos que necesita:

- integrador de una rama: *struct integrador1*
- integrador de dos ramas: *struct integrador2*
- integrador de tres ramas: *struct integrador3*
- comparador: *struct comparador*
- cuantizador: *struct cuantizador*
- CDA: *struct convertidor*
- Retraso: *struct retraso*
- Amplificador: *struct amplificador*
- Sumador: *struct sumador*
- Multiplicador: *struct multiplicador*

En el caso especial de los integradores, también se tienen que inicializar las entradas correspondientes a las tensiones en los condensadores de muestreo del integrador siguiente y los valores de los condensadores de muestreo del integrador siguiente, que por defecto en el caso de que no haya ninguno son cero.

```
void anadir_elemento(char tipo,char *nombre,char *salida,char *ent1,char *ent2,char *ent3,char
*ent4,char *ent5,char *ent6,char *nmod,double peso1,double peso2,double peso3){
        anadir_nombre(&pnombre,nombre);
        inicializa_elemento();
        switch(tipo){
         case '1' :{
          char cadena[20];
          ultimo->nombre='1';
          ultimo->modelo.int1.nombre=nombre;
          ultimo->modelo.int1.salida=salida;
          ultimo->modelo.int1.entrada1=ent1;
          ultimo->modelo.int1.entrada2=ent2;
          anadir_nodo(&sal,salida);
          anadir_lista(&ent,ent1);
          anadir_lista(&ent,ent2);
          ultimo->modelo.int1.peso=peso1;
          ultimo->modelo.int1.vc1="gnd";
          ultimo->modelo.int1.vc2="gnd";
          ultimo->modelo.int1.vc3="gnd";
          ultimo->modelo.int1.cn1=0;
          ultimo->modelo.int1.cn2=0;
          ultimo->modelo.int1.cn3=0;
          strncpy(cadena,nombre,strlen(nombre));
          strcpy(cadena+strlen(nombre),"_vcn");
          ultimo->modelo.int1.vcn=strdup(cadena);
          if(nmod!=NULL)
           ultimo->modelo.int1.nombre_mod=nmod;
          else{
           ultimo->modelo.int1.nombre_mod="defint";
           if(!integ)
            anadir_modelo('i',"defint");
           integ++;
          }
          break;
         }
        }
}
```

## C.1.5. anade_modelo

Esta función se le llama desde el parser cada vez que reconoce la sintaxis de un *modelo*. También se le llama desde la función *anade_elemento* si el bloque no tiene modelo asociado. Se le llama con lo siguientes parámetros.

- *C*: Es un carácter asociado a los elementos que tienen modelo asociado que sería:
  - C para los comparadores
  - S para los sumadores
  - A para los amplificadores
  - D para los CDA
  - Z para los cuantizadores
  - Default para los integradores

- *Nombre*: Es el nombre del modelo asociado. En el caso de que el elemento no lleve modelo asociado se le pondrá uno por defecto que se llamará *def_nombreelemento.*

Esta función mete en una lista de *struct modelo* todas las instancias de modelos que localice e inicializa cada modelo con los parámetros por defecto excepto en el caso de que en la instancia del modelo se le introduzcan parámetros distintos de los que tuviese por defecto.

```
void anadir_modelo(char c,char *nombre){
  anadir_nombre(&pmod,nombre);
  inicializa_modelo();
  switch (c){
    case 'c' :{
      ult_mod->nombre='c';
      ult_mod->tipo.comp.nombre=nombre;
      rellena_nombre(&ult_mod->tipo.comp.voff,0);
      rellena_nombre(&ult_mod->tipo.comp.vhis,0);
      ult_mod->tipo.comp.htype=0;
      if(strcmp(nombre,"defcomp"))
        for(upar=ppar;upar!=NULL;upar=upar->siguiente)
          if(!strcmp(upar->nombre,"voff"))
            rellena_par(&ult_mod->tipo.comp.voff);
          else if (!strcmp(upar->nombre,"vhis"))
              rellena_par(&ult_mod->tipo.comp.vhis);
            else if (!strcmp(upar->nombre,"htype"))
                ult_mod->tipo.comp.htype=(int)upar->valor;
                    else{
                fprintf(stderr,"%d: Tipo de parametro (%s) no corresponde con modelo de
comparador\n",lineno,upar->nombre);
                exit(1);
              }
      borrar_lista(&ppar);
      break;
    }
}
```

Cada modelo tiene un tipo de estructura conteniendo todos los parámetros que necesita:

- integrador: *struct mod_int*
- comparador: *struct mod_comp*
- cuantizador: *struct mod_cuant*
- CDA: *struct mod_conv*
- Sumador: *struct mod_sum*
- Amplificador: *struct mod_ampl*

En el caso de que en la instancia del modelo se le pase como parámetro alguno que no perteneciese a ese modelo o que no existiese, esta función daría error y devolvería el control al sistema operativo.

## C.1.6. anadir_reloj

A esta función se le llama desde el parser cuando se encuentra la instancia de reloj. Inicializa la estructura reloj a los parámetros por defecto y a la frecuencia de muestreo dada.

```
void anadir_reloj(){
  reloj.num++;
  rellena_nombre(&reloj.frec,1);
  rellena_nombre(&reloj.jitter,0);
  rellena_nombre(&reloj.dc,0.5);
  rellena_nombre(&reloj.ird,0);
  rellena_nombre(&reloj.srd,0);
  for(upar=ppar;upar!=NULL;upar=upar->siguiente)
    if(!strcmp(upar->nombre,"frec"))
      rellena_par(&reloj.frec);
    else if(!strcmp(upar->nombre,"jitter"))
        rellena_par(&reloj.jitter);
      else if(!strcmp(upar->nombre,"dc"))
          rellena_par(&reloj.dc);
        else if(!strcmp(upar->nombre,"ird"))
            rellena_par(&reloj.ird);
          else if(!strcmp(upar->nombre,"srd"))
              rellena_par(&reloj.srd);
            else{
              fprintf(stderr,"%d: Tipo de parametro (%s) no corresponde con modelo
de reloj\n",lineno,upar->nombre);
              exit(1);
            }
  borrar_lista(&ppar);
}
```

Da error si se introduce algún parámetro que no se corresponda con los parámetros asociados al reloj y devuelve el control al sistema operativo.

## C.1.7. anadir_parametro

Esta función la llama el parser cuando dentro de una instancia de modelo, localiza un parámetro. Introduce en una lista de *struct parámetro* todos los parámetros que se le introduzcan a ese modelo y después libera la lista después de que se llame a la función *anadir_modelo*, mediante la función *borrar_lista()*. Esta función libera de la memoria la lista de *struct parámetro*.

Se le llama con los siguientes parámetros:

- *Nombre*: es el tipo del parámetro asociado al modelo. Ej. Voff, vhis...
- *Valor*: es el valor del parámetro.
- *Npar*: es el nombre asociado al modelo para definirlo después en una sentencia "*param*". En el caso de que no lleve nombre asociado al parámetro, este parámetro debe pasarse como una cadena vacía.

```
void anadir_parametro(char *nombre,double valor,char *npar){
 if(ppar==NULL){
  ppar=(struct parametro *)malloc(sizeof(struct parametro));
  ppar->siguiente=NULL;
  upar=ppar;
  upar->siguiente=NULL;
 }
 else{
  if(ppar->siguiente==NULL){
    ppar->siguiente=(struct parametro *)malloc(sizeof(struct parametro));
    upar=ppar->siguiente;
    upar->siguiente=NULL;

  }
  else{
    upar->siguiente=(struct parametro *)malloc(sizeof(struct parametro));
    upar=upar->siguiente;
    upar->siguiente=NULL;
  }
 }
 upar->nombre=nombre;
 upar->valor=valor;
 upar->npar=npar;
 if(strcmp(upar->npar,""))
  anadir_lista(&pmonte,upar->npar);
 if(!strcmp(upar->nombre,"dc"))
  strcpy(dc,upar->npar);
 if(!strcmp(upar->nombre,"ird"))
  strcpy(ird,upar->npar);
 if(!strcmp(upar->nombre,"srd"))
  strcpy(srd,upar->npar);
}
```

## C.1.8. anadir_param

Esta función es llamada desde el parser cada vez que se encuentra una sentencia *param*. Introduce en una lista de *struct param* los parámetros definidos en cada sentencia del parser *param*.

Devuelve un cero en el caso de que lo haya podido realizar con éxito y un 1 en el caso que ya se haya definido ese parámetro antes en la sentencia *param*.

Se le llama con dos parámetros:

- *Nombre*: es el nombre con el que se ha definido al parámetro.
- *Valor*: Es el valor asociado a ese parámetro. En el caso de que la sentencia *param* sea del tipo de variables aleatorias, ya sea gaussiana, o bien, uniforme, se le pasará a la función la media de la distribución. En el caso de que sea un barrido *SWEEP* se le pasará el valor de comienzo.

```
int anadir_param(char *nombre,double valor){
  struct param *busca=pparam;
  int result=0;
  if (busca==NULL){
    pparam=(struct param *)malloc(sizeof(struct param));
    pparam->siguiente=NULL;
    pparam->nombre=nombre;
    pparam->valor=valor;
  }
  else{
    while(busca->siguiente!=NULL && strcmp(nombre,busca->nombre))
      busca=busca->siguiente;
    if(strcmp(nombre,busca->nombre)){
      busca->siguiente=(struct param *)malloc(sizeof(struct param));
      busca=busca->siguiente;
      busca->siguiente=NULL;
      busca->nombre=nombre;
      busca->valor=valor;
    }
    else result=1;
  }
  return result;
}
```

Otra función similar a esta es *anadir_lista,* que añade a una lista de *struct lista,* o bien, las etiquetas de los modelos, o bien, los nodos del circuito. Devuelve al igual que en el caso anterior cero, en el caso de que lo haya podido realizar con éxito y 1 en el caso de que la etiqueta o el nodo estén repetidos. En lugar del parámetro valor se utiliza un parámetro a *struct lista,* que es un puntero al comienzo de la lista en cuestión. (de etiquetas o de nodos).

## C.1.9. anadir_nodo

```
void anadir_nodo(struct lista **puntero,char *nombre){
  if(anadir_lista(puntero,nombre)){
    fprintf(stderr,"%d: Nodo de salida duplicado
:%s\n",lineno,nombre);
    exit(1);
  }
}
```

Esta función llama a la función *anadir_lista* explicada en el apartado anterior y en el caso de que haya algún nodo repetido en el circuito, da error diciendo el número de línea en el que se encuentra y devuelve el control al sistema operativo.

Se le llama con dos parámetros

- Nombre: Es el nombre del nodo del circuito
- Puntero: Es un puntero al comienzo de la lista de nodos

La función *anadir_nombre* es similar a está pero el puntero apunta al comienzo de la lista de etiquetas de modelos o etiquetas de elementos.

## C.1.10. rellena_nombre

La función rellena_nombre se le llama desde *anadir_modelo* y *anadir_reloj*. Inicializa los valores del un parámetro con el valor introducido en la instancia de modelo. Se le llama con los siguientes parámetros

- Valor: es el valor del parámetro
- Par: es el puntero al último parámetro de la lista *struct nombre*.

```
void rellena_nombre(struct nombre *par,double valor){
  (*par).nombre="";
  (*par).valor=valor;
}
```

## C.1.11. imprime_do_script

```
void imprime_do_script(){
  char entity[20];
  int i;
  FILE *iscript=fopen(script,"w");
  FILE *ido=fopen(dofile,"w");
  for(i=0;script[i]!='\0';i++)
    if((script[i]>=65 && script[i]<=90) || script[i]==95 || (script[i]>=48 && script[i]<=57))
      entity[i]=script[i];
    else if(script[i]>=97 && script[i]<=122)
        entity[i]=script[i]-32;
      else{
        fprintf(stderr,"caracter invalido en nombre de fichero : ");
        putchar(script[i]);
        fprintf(stderr,"\n");
        exit(1);}
  if(tipo)
    strcpy(entity+i,"_TESTM");
  else
    entity[i]='\0';
  if(!iscript){
    fprintf(stderr,"No se pudo abrir el fichero %s\n",script);
    exit(1);}
  if(!ido){
    fprintf(stderr,"No se pudo abrir el fichero %s\n",dofile);
    exit(1);}
  fprintf(ido,"run %d\n",(unsigned long
int)(monte.nparam*(nmuestras.npuntos+2)*periodo*1e9));
  fclose(ido);
  if(!tipo){
    fprintf(iscript,"vacom %s\n",outfile);
    fprintf(iscript,"vasim -c -do %s %s TEST\n",dofile,entity);}
  else{
    fprintf(iscript,"vlog -work CONVERTIDORES %s\n",outfile);
    fprintf(iscript,"vsim -c -do %s CONVERTIDORES.%s\n",dofile,entity);}
  fclose(iscript);
  if(chmod(script,488))
    fprintf(stderr,"WARNING: No se pudo cambiar el acceso al fichero: %s\n",script);
}
```

Esta función se le llama desde la función principal y es la encargada de generar el fichero script de salida y el fichero .do. En el caso de que no pudiera abrir alguno de los dos ficheros daría error y devolvería el control al sistema operativo. También da error si el nombre de fichero introducido por el usuario no esta entre los caracteres alfanuméricos o [-_].

En el fichero .do se encargará de escribir *"run tiempo_simulación"*. Mientras que en el fichero script escribirá los comandos necesarios para el arranque de las herramientas ADVANCE – MS o MODELSIM según queramos el código en VHDL o VERILOG respectivamente.

## C.1.12. imprime_fichero

Esta función tampoco tiene parámetros de entrada. Se encarga de generar el fichero de salida según sea para VHDL o VERILOG. Esta función llama a otras funciones auxiliares, que van escribiendo formando el fichero final por partes.

- imprime_cabecera: Imprime la primera parte del fichero VHDL o VERILOG, correspondiente a la declaración de bibliotecas y paquetes, definición de la entidad (en VHDL) y del módulo (en VERILOG) y definición de las constantes necesarias y de las señales comunes como la de reloj y la de reset.

- imprime_elemento: Sirve para declarar tanto en VHDL como en VERILOG las señales que forman los puertos del circuito.

- imprime_modelo: Sirva para declarar tanto en VHDL como en VERILOG, las señales que no forman los puertos del circuito, es decir, aquellas, que se necesitan para pasar los parámetros de segundo orden. En el caso de VERILOG se le llama una segunda ver para inicializar estos parámetros al valor que se ha definido en el fichero de entrada a VSIDES.

- imprime_instancia: Declara tanto en VHDL como en VERILOG, las instancias de los bloques básicos que componen el circuito.

- imprime_monte: En el caso de que en el fichero de entrada a VSIDES se haya definido la variación de un parámetro ya sea mediante una sentencia *sweep* o mediante una sentencia *monte*, esta función se encarga de añadir el código necesario, para que el simulador correspondiente lo tenga en cuenta

- imprime_final: Esta función se encarga de definir el proceso necesario para que los simuladores escriban los resultados temporales a un fichero.

- imprime_nodo: Esta función tiene como parámetro el nombre de un nodo del circuito. Declara un nodo según el lenguaje de programación

- imprime_par: Tiene como parámetros *nombre*, que es el nombre de un parámetro, *par*, que es un añadido a ese nombre y *npar*, que es un nombre alternativo como el definido en una sentencia *param* de VSIDES. Esta función imprime el nombre de un parámetro de la siguiente forma:

- nombre_par: En el caso de que npar sea la cadena vacía
- npar: Si npar es distinto de la cadena vacía

- imprime_param : Esta función declara un parámetro teniendo en cuenta lo explicado para la función imprime_par: Tiene como parámetros *nombre*, que es el nombre del parámetro, *valor*, que es el valor al que hay que inicializar el parámetro, *par* y *npar* que coinciden con lo explicado para la función anterior.

```
void imprime_fichero(){
 struct elemento *busca;
 struct modelo *bmodelo;
 struct param *bparam;
 yyout=fopen(outfile,"w");
 if(!yyout){
  fprintf(stderr,"No se pudo abrir el fichero %s\n",outfile);
  exit(1);}
 imprime_cabecera();
 imprime_elemento();
 if(!anadir_lista(&pnodo,output.nombre)){
  fprintf(stderr,"Nodo de sentecia output(%s)no se corresponde con ningun nodo\n",output.nombre);
  exit(1);}
 imprime_modelo();
 for(bparam=pparam;bparam!=NULL;bparam=bparam->siguiente)
  if(strcmp(bparam->nombre,reloj.frec.nombre))
   if(!tipo) fprintf(yyout,"\tSIGNAL %s : REAL := %.10e;\n",bparam->nombre,bparam->valor);
   else fprintf(yyout,"\treg  [63 : 0] %s;\n",bparam->nombre);
  else{
   if(!tipo) fprintf(yyout,"\tCONSTANT Periodo : REAL := %.10e;\n",1/bparam->valor);
   else fprintf(yyout,"\tparameter real Periodo = %.10e;\n",1/bparam->valor);
   periodo=1/bparam->valor;}
 if(!tipo){
  fprintf(yyout,"BEGIN\n");
  fprintf(yyout,"\tclock<= NOT clock AFTER %.10f ns;\n",periodo/2e-9);}
 else  fprintf(yyout,"\n");
 imprime_instancias();vez=1;
 if(tipo){
  fprintf(yyout,"\n\tinitial begin\n");
  fprintf(yyout,"\t\taux=1;\n\t\taux_s=1;\n\t\taux_e=1;\n");
  fprintf(yyout,"\t\tseedg=1;\n\t\tseedu=1;\n");
  fprintf(yyout,"\t\tstop=0;\n\t\tnum=0;\n");
  fprintf(yyout,"\t\tReset=1;\n\t\tclock=0;\n");
  fprintf(yyout,"\t\tgnd = $realtobits(0.0);\n");
fprintf(yyout,"\t\tPPDEC=%.10e;\n\t\tNPARAM=%.10e;\n",(double)monte.ppdec,(double)monte.nparam);
  fprintf(yyout,"\t\tNPUNTOS=%.10e;\n",(double)nmuestras.npuntos);
  imprimir_param("SIGNAL","dc","",reloj.dc.valor,reloj.dc.nombre);
  imprimir_param("SIGNAL","ird","",reloj.ird.valor,reloj.ird.nombre);
  imprimir_param("SIGNAL","srd","",reloj.srd.valor,reloj.srd.nombre);
  imprime_modelo();
  for(bparam=pparam;bparam!=NULL;bparam=bparam->siguiente)
   if(strcmp(bparam->nombre,reloj.frec.nombre))
        fprintf(yyout,"\t\t%s = $realtobits(%.10e);\n",bparam->nombre,bparam>valor);
  fprintf(yyout,"\n\t\tforever #%.10f clock=!clock;\n",periodo/2e-9);
  fprintf(yyout,"\tend\n\talways @(posedge clock) begin\n");  }
 imprime_final();
 fclose(yyout);}
```

# APÉNDICE D PLI y encapsulación de código C

## D.1. Uso de PLI

### D.1.1. Procedimiento genérico

VERILOG PLI (Interfaz de lenguaje de programación) provee un mecanismo para definir system tasks y funciones que comunican al simulador a través de un interfaz de procedimientos en C. Con MODELSIM se puede implementar todas las PLI definidas en el estándar IEEE 1364.

En este proyecto las PLI se han usado para encapsular las funciones matemáticas que VERILOG no tiene como seno, logaritmo, exponencial…, así como dos funciones para la escritura de datos a un fichero. En este apartado se procederá a describir el procedimiento genérico para encapsular cualquier PLI que queramos, particularizándolo para las funciones que hemos dicho anteriormente.

Para poder usar una función PLI con MODELSIM hay que seguir el procedimiento mostrado en la Figura D.19.

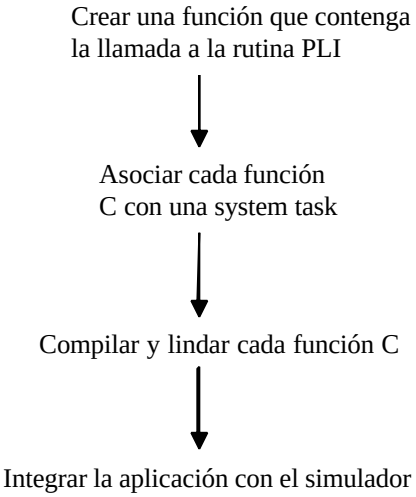



**Figura D.19. Diagrama de flujo para la incorporación de funciones PLI**

### D.1.2. Creación de una función C y asociación con una system task

Una vez escrito el código C que queremos ejecutar, se tiene que asociar a una función PLI. Hay que tener en cuenta que este fichero C, tiene que tener incluidos todos los archivos de cabecera necesarios, además de el archivo <<*veriuser.h*>>.

Para asociar la rutina en C con una system task hay que editar el array *veriusertfs*. Parte de la información que se introduce en el array es el nombre de la nueva system task y el nombre de la rutina C que corresponde a dicha system task.

La estructura veriusertfs se define como sigue:

```
s_tfcell veriusertfs[]={
{type,data,checktf(),sizetf(),calltf(),misctf(),"$tfname"},
{0}
}
```

- *Type*: El primer campo de la estructura, puede ser lo siguiente
  - *Usertask*, que es una system task que no devuelva ningún valor
  - *Userfunction*, que es una system task que puede devolver cualquier valor
  - *Userrealfuncion* que es una system task que devuelve un número real

- *Data*: Este valor se le pasa a la rutina definida en C.

- *Checktf*: puntero a la rutina que sirve para comprobar los parámetros de la systmem task. Si no tiene ninguna rutina asociada se le pasa como cero

- *Sizetf*: Este es un puntero a la rutina definida por el usuario que devuelve el tamaño en bits del dato devuelto por la system task.

- *Calltf*: Puntero a la rutina principal que VERILOG llama a una system task

- *Misctf*: Puntero a una rutina opcional

- *Tfname*: Nombre de la system task llamada desde VERILOG

La estructura tiene que tener tantas entradas como system task quiera definir el usuario. Así mismo tiene que acabar con una entrada vacía {0}. La estructura veriusertfs definida para llamar a las rutinas necesarias para este proyecto tiene diez entradas, una para cada una de las funciones necesarias. Utiliza como rutina sizetf(), una que devuelve 64 que es el tamaño del dato devuelto en cada una de las llamadas. Se han definido tres rutinas para comprobar el tamaño y el tipo de datos, que en todos los casos son reales. Cada una de las rutinas comprueba si se le pasa uno, dos o tres números reales y son: math_chech1, math_check2 y math_check3. La estructura queda como sigue:

```
s_tfcell veriusertfs[] = {
   {userrealfunction, 0, math_check1, math_size, exp_call,   0, "$exp"},
   {userrealfunction, 0, math_check1, math_size, log_call,   0, "$log"},
   {userrealfunction, 0, math_check1, math_size, log10_call, 0, "$log10"},
   {userrealfunction, 0, math_check1, math_size, sin_call,   0, "$sin"},
   {userrealfunction, 0, math_check1, math_size, sqrt_call,  0, "$sqrt"},
   {userrealfunction, 0, math_check2, math_size, pow_call,   0, "$pow"},
   {userrealfunction, 0, math_check1, math_size, abs_call,   0, "$abs"},
   {userrealfunction, 0, math_check1, math_size, sign_call,  0, "$sign"},
   {userrealfunction, 0, math_check2, math_size, copiar_call, 0,"$copiar"},
   {userrealfunction, 0, math_check3, math_size, escribir_call,0,"$escribir"},
   {0} /* final entry must be 0 */
};
```

## D.1.3. Compilado y linkado de la función C.

Una vez que tenemos el archivo con extensión .c hay que proceder a compilarlo y linkarlo. MODELSIM usa llamadas dinámicas al sistema operativo para cargar las aplicaciones PLI cuando el simulador carga un diseño, por lo tanto las aplicaciones deben ser compiladas y linkadas para que se carguen dinámicamente en el sistema operativo que se este utilizando que en nuestro caso es SOLARIS.

El archivo donde se han definido todas las rutinas necesarias es pli.c Para compilarlo y linkarlo hay que realizar lo siguiente

cc –c –I/cad/MENTOR_C4/AMS_SIMULATION_00.Q4/modeltech/include pli.c
ld –G –o pli.so pli.o

## D.1.4. Integrar la aplicación con el simulador

Cada aplicación PLI debe registrar sus system tasks proveyendo al simulador del nombre de cada system task y las rutinas asociadas de llamada.

Existen tres métodos para que MODELSIM utilice las aplicaciones PLI:

- Una lista en la entrada veriuser del archivo modelsim.ini
  Ej. Veriuser = pli.so

- Establecer una lista en la variable de entorno PLIOBJS
  Ej. % setenv PLIOBJS “pli.so”

- Usar la opción –pli para arrancar el simulador
  Ej. –pli pli.so

Los diversos métodos para especificar las aplicaciones PLI se pueden usar simultáneamente.

# D.2. Encapsulación de código C.

## D.2.1. Procedimiento genérico

ADVANCE – MS permite encapsular funciones C para que se puedan invocar desde una simulación. Para que esto sea posible el código C tiene que tener las mismas propiedades que un subprograma en VHDL.

- El comportamiento del subprograma es el mismo para la clase establecida de parámetros
- No se hacen suposiciones sobre el número de llamadas que se hacen a la función
- El valor de los parámetros puede ser independiente de las llamadas anteriores a la función
- No es posible obtener objetos complejos de VHDL como SIGNALS.

En este proyecto la encapsulación de código C se ha tenido que utilizar para crear dos funciones de escritura de datos a ficheros. Se procederá a explicar el procedimiento genérico para encapsular cualquier función mediante una función de código C llamada vhdl_escribir cuyo prototipo es:

double vhdl_escribir(double *,double,double);

El procedimiento para encapsular una función C puede verse en la Figura D.20.

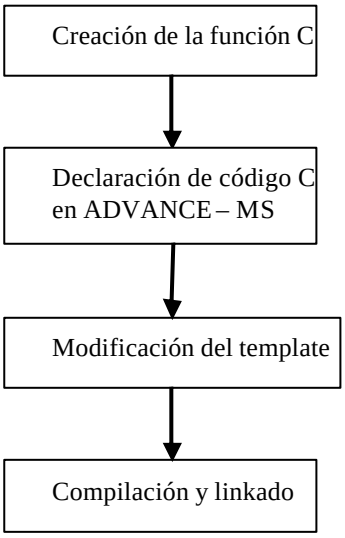



**Figura D.20. Diagrama de flujo para encapsular funciones C en ADVANCE – MS**

## D.2.2. Creación y declaración de código C

Una vez que se ha creado la función C, tenemos que declararla de forma que se pueda usar en ADVANCE – MS. La declaración de funciones C sólo esta restringida a paquetes (PACKAGES). En estos paquetes hay que definir tres cosas.

Estos paquetes necesitan la especificación de dos atributos, la ruta del archivo de cabecera que contiene el prototipo de la función C que queremos encapsular y la ruta del archivo objeto del fichero C en cuestión.

La declaración de la función VHDL correspondiente a la función C que queremos encapsular. En nuestro caso es vhdl_escribir

Como atributo de la función VHDL hay que especificar el nombre de la función C que queremos llamar. En nuestro caso es escribir_c.

Esto quedaría como sigue

```
PACKAGE encapsula_escr IS
        ATTRIBUTE foreign OF encapsula_escr : PACKAGE IS "ADMS
:h@<user_path>prototipos_ex_c.h o@<user_path>/escribir.o";
        FUNCTION vhdl_escribir(dato:IN REAL_VECTOR;nparam:IN REAL;iter:IN
REAL)RETURN REAL;
        ATTRIBUTE foreign OF vhdl_escribir : FUNCTION IS "ADMS :escribir_c";
end encapsula_escr;
```

## D.2.3. Modificación del template

El template hace uso del fichero <<*macro.h*>> que es donde se encuentran todas las macros necesarias para establecer la correspondencia de tipos entre los de VHDL – AMS y los de código C. Hay que modificar cuatro partes:

a. La declaración del archivo del prototipo de la función C a encapsular
b. Una predeclaración correspondiente a la función VHDL
c. La tabla de los nombres de la imagen de la función C de VHDL
d. El cuerpo de la imagen de la función C de VHDL

En la Figura D.21 se puede ver el témplate modificado correspondiente a la función escribir_c.

## D.2.4. Compilación y linkado

El archivo resultante *.so* debe de copiarse dentro de la biblioteca de trabajo en el directorio igual a *ss5*. En nuestro caso el template de ha renombrado a *template_escr.c* que tiene como código objeto *template_escr.o* La función de código C asociada se llama *escribir.c* La secuencia de comandos necesaria para compilarlo y lindarlo se muestra a continuación.

```
gcc -c -g -fPIC c_encapsulation/escribir.c -o c_encapsulation/escribir.o
gcc -fPIC -ansi -g -DNDEBUG -I $anacad/adms/$admsver/include -D__EXTENSIONS__ -
D__EXT_VHDLA__ -DVHDLANALOG -D__EXT_ELDO__ -I..
c_encapsulation/template_escr.c -c -o c_encapsulation/template_escr.o
gcc -Wl,-G -nostdlib  c_encapsulation/escribir.o c_encapsulation/template_escr.o -o
./CONVERTIDORES /ss5/ENCAPSULA_ESCR.so –lgcc
```

```
#include "macro.h"
/*****************************************/
/*        MANUAL MODIFICATIONS         */
/*****************************************/
/* the file containing the prototype of the C encapsulated functions  has to be included*/
#include "prototipos_ex_c.h"
/*********************************/
/* END OF MANUAL MODIFICATIONS */
/*********************************/
extern PROC_CAST current_active_process;
RG_DECL;
VPTR_DECL;
LOOP_I_DECL;
LOOP_DRIVER_DECL;
static void elab_desent();
static int dblines[0]={};
/*****************************************/
/*        MANUAL MODIFICATIONS         */
/*****************************************/
/* To each VHDL-AMS function "Funci_VHDL", correspond a C predeclaration */
static double vhdl_escribir ();
/* Table of the names of C image of the VHDL-AMS functions */
CODE_SUPRGS_DECL=
{(CODE_CAST)vhdl_escribir};
/*********************************/
/* END OF MANUAL MODIFICATIONS */
/*********************************/
PKGI_DECL(0);
ELB_CODE_CAST elab_funcs=
{elab_desent,0,(void*)code_suprgs,NULL};
static void elab_desent(int* ENVIR_CAST,int selector){
switch(selector){
case -1: /* Analog Code and Subprogram Definition */
break;
case 0: /* block: design entity */
ALLOC_RG;
break;
default:
/* Error Management */
break;
}
}
/*****************************************/
/*        MANUAL MODIFICATIONS         */
/*****************************************/
/* body of the C image of the VHDL-AMS functions*/
static double vhdl_escribir(FRGN_PARAM_CON_UARR(0),
FRGN_PARAM_CON_REAL(1),FRGN_PARAM_CON_REAL(2),PKG_DECL)
{
        FRGN_DECL_CON_ARR_REAL(dato,0);
        FRGN_DECL_CON_REAL(nparam,1);
        FRGN_DECL_CON_REAL(iter,2);
        return escribir_c(dato,nparam,iter);
} /* end vhdl_escribir */
/*********************************/
/* END OF MANUAL MODIFICATIONS */
/*********************************/
```

**Figura D.21. Template modificado para la función escribir**

# APÉNDICE E Código VHDL y VERILOG de los bloques básicos

## E.1. Código VHDL del integrador

A continuación se presenta el código VHDL del integrador de tres ramas. El código VHDL del integrador de dos ramas y de una rama es semejante al de tres.

```
// Entity Integrador de tres rama

// ENTRADAS Xin11,Xin21,Xin22,Xin12,xin31,Xin32,clock,Reset
//        COMO PARAMETROS
//                A0      Ganancia en DC del OTA
//                A01,A02,A03,A04      Parametros de la ganancia no lineal en tanto porciento de la
ganancia total
//                Cp1,Cp2         Capacidades parasitas
//                Ron     Resistencia de llave
//                INPSDDensisdad espectral de ruido a la entrada
//                alpha,beta coeficientes no lineales de los condensadores en tanto por uno
//                sigma desviacion en el peso en tanto por uno
// SALIDAS Xout,Xvc1
// ERRORES IMPLEMENTADOS
//        ganancia finita y no lineal, condensadores no lineales
//        error de settling, error de desapareamiento, ruido termico
// PARAMETROS DE ENTRADA
//                VDD   Tension de rail positivo de entrada
//                VSS   Tension de rail negativo de entrada
//                INT_TYPE =1 REAL
//                C2      Capacidad de integracion
//                gm      Transconductancia del amplificador
//                w       Peso de la rama
//                dc      ciclo de trabajo
//                ird     reduccion de integracion
//                srd     reduccion de muestro
//                TS      Periodo de reloj
// VARIABLES LOCALES
//                Vai,Vas,Va1,Va2,Va_ant       Tensiones en la pata negativa del integrador
//                        debida al settling
//                vc11,vc12,vc21,vc22  Tensiones de los condensadores en las fases de muestreo e
integracion
//                Ceqs,Ceqi,Cprima Capacidades para calcular el settling
//                Vout,Vin1,Vin2 Variables auxiliares que almacenan los valores de salida y entrada
//                Vant    Almacena el valor anterior de la salida
//                C1      Variable del condensador de muestreo
```

```
//              A0_nl  Ganancia no lineal
//              A0_nl_var Varianza de la ganancia no lineal
//              iter     Valor de la iteraccion
//              S1d     PSD de ruido de llave phi1d
//              S1      PSD de ruido de llave phi1
//              Sop     PSD de ruido del OPAMP
//              S2      PSD de ruido de llave phi2
//              BW1d  Ancho de banda equivalente de phi1d
//              BW1    Ancho de banda equivalente de phi1
//              BWop  Ancho de banda equivalente del OPAMP
//              BW2    Ancho de banda equivalente de phi2
//              Taui,Taus       Tiempo efectivo de muestro y de integracion
//              Vruido Tension equivalente de ruido
//              seed1,seed2     Semilla para la generacion de variables aleatorias

moduleintegrador3M(Reset,clock,Xin11,Xin12,Xin21,Xin22,Xin31,Xin32,
                        Xvcn1,Xvcn2,Xvcn3,Xout,Xvc1,Xvc2,Xvc3,A0p,A01p,A02p,A03p,A04p,
                        Cp11p,Cp12p,Cp13p,Cp2p,Clop,sigmap,alphap,betap,Ronp,gmp,
                        INPSDp,Imaxp,dcp,irdp,srdp);
        input Reset,clock;
        input   [63 : 0] Xin11,Xin12,Xin21,Xin22,Xin31,Xin32,Xvcn1,Xvcn2,Xvcn3;
        input [63 : 0] A0p,A01p,A02p,A03p,A04p,Cp11p,Cp12p,Cp13p,Cp2p,Clop;
        input [63 : 0] sigmap,alphap,betap,Ronp,gmp,INPSDp,Imaxp,dcp;
        input [63 : 0] irdp,srdp;
        output [63 : 0] Xout,Xvc1,Xvc2,Xvc3;
        reg [63 : 0 ] Xout,Xvc1,Xvc2,Xvc3;
        parameter int_type=1.0,osp=1.0e12,osn=-1.0e12,Peso1=1.0,Peso2=1.0,Peso3=1.0;
        parameter C2=2.0e-12,Periodo=4.0e-7,Cn1=0.0,Cn2=0.0,Cn3=0.0;
        real vc11a,vc11b,vc21,vc12a,vc12b,vc11c,vc12c,vc22,A0_nl,A0_nl_var;
        real iter,C11,C12,C13,Vout_ant,Taui,Taus,S11d,S11,S12d,S12,S13d,S13;
        real Sop,S2,BW11d,BW11,BW12d,BW12,BW13d,BW13,BWop,BW2,delta1,delta2;
        real delta3,vruido1a,vruido2a,vruido1b,vruido2b,vruido3a,vruido3b;
        real vout_v,Vai,Vas,Va1,Va2,Va_ant,Ceqs,Ceqi,Cprima,aux,ts,ti,W1,W2,W3;
        real A0,A01,A02,A03,A04,Cp11,Cp12,Cp13,Cp2,Clo,sigma,alpha;
        real    beta,Ron,gm,INPSD,Imax,dc,ird,srd;
        real i,j,media,std;
        integer seedi;
        initial begin
                seedi=1;
                C11=Peso1*C2;
                C12=Peso2*C2;
                C13=Peso3*C2;
                Vai=0.0;
                Vas=0.0;
                Va1=0.0;
                Va2=0.0;
                vc22=0.0;
                vc21=0.0;
                Xout=$realtobits(0.0);
                Xvc1=$realtobits(0.0);
```

```
        Xvc2=$realtobits(0.0);
        Xvc3=$realtobits(0.0);
        aux=1.0;
        A0=$bitstoreal(A0p);
        A01=$bitstoreal(A01p);
        A02=$bitstoreal(A02p);
        A03=$bitstoreal(A04p);
        A04=$bitstoreal(A04p);
        Cp11=$bitstoreal(Cp11p);
        Cp12=$bitstoreal(Cp12p);
        Cp13=$bitstoreal(Cp13p);
        Cp2=$bitstoreal(Cp2p);
        Clo=$bitstoreal(Clop);
        sigma=$bitstoreal(sigmap);
        alpha=$bitstoreal(alphap);
        beta=$bitstoreal(betap);
        Ron=$bitstoreal(Ronp);
        gm=$bitstoreal(gmp);
        INPSD=$bitstoreal(INPSDp);
        Imax=$bitstoreal(Imaxp);
        dc=$bitstoreal(dcp);
        ird=$bitstoreal(irdp);
        srd=$bitstoreal(srdp);
end
always @(clock)
        if(Reset) begin
                Xout=$realtobits(0.0);
                Xvc1=$realtobits(0.0);
                Xvc2=$realtobits(0.0);
                Xvc3=$realtobits(0.0);
                vc22 =0.0;
                Vai = 0.0;
                Vas = 0.0;
                Va1= 0.0;
                Va2 = 0.0;
                vc21 = 0.0;
                aux=1.0;
                A0=$bitstoreal(A0p);
                A01=$bitstoreal(A01p);
                A02=$bitstoreal(A02p);
                A03=$bitstoreal(A04p);
                A04=$bitstoreal(A04p);
                Cp11=$bitstoreal(Cp11p);
                Cp12=$bitstoreal(Cp12p);
                Cp13=$bitstoreal(Cp13p);
                Cp2=$bitstoreal(Cp2p);
                Clo=$bitstoreal(Clop);
                sigma=$bitstoreal(sigmap);
                alpha=$bitstoreal(alphap);
                beta=$bitstoreal(betap);
```

```
                Ron=$bitstoreal(Ronp);
                gm=$bitstoreal(gmp);
                INPSD=$bitstoreal(INPSDp);
                Imax=$bitstoreal(Imaxp);
                dc=$bitstoreal(dcp);
                ird=$bitstoreal(irdp);
                srd=$bitstoreal(srdp);
        end
        else begin
                if (aux) begin
                        Taui = Periodo*(1.0 -dc)-ird;
                        Taus = Periodo*dc-srd;
                        Ceqs= Cp2+(Clo+Cn1+Cn2+Cn3)*(1.0+Cp2/C2);
                        Ceqi=
Cp2+C11+C12+C13+Clo*(1.0+Cp2/C2+Peso1+Peso2+Peso3);
                        Cprima= Cp2+Clo*(1.0+Cp2/C2);
                        if (Ron> 0.0) begin
                                BW11d= 1.0 /(2.0 *Ron*(2.0*C11+Cp11));
                                BW12d= 1.0 /(2.0 *Ron*(2.0*C12+Cp12));
                                BW13d= 1.0 /(2.0 *Ron*(2.0*C13+Cp13));
                                BW11=(C11+Cp11)/(2.0 *Ron*C11*(2.0 *C11+Cp11));
                                BW12=(C12+Cp12)/(2.0 *Ron*C12*(2.0 *C12+Cp12));
                                BW13=(C13+Cp13)/(2.0 *Ron*C13*(2.0 *C13+Cp13));
                                BW2=
0.25*(Clo*Cp2+C2*Cp2+2.0*(C11+C12+C13)*gm*Ron*C2+Clo*C2)/(((C2+Clo+2.0*gm*Ron*
C2)*(C11+C12+C13)+Clo*C2+Clo*Cp2+C2*Cp2)*Ron*(C11+C12+C13));
                        end
                        else begin BW11d= 0.0;BW12d= 0.0;BW11= 0.0;BW12=
0.0;BW13d= 0.0;BW13= 0.0;BW2= 0.0; end
                        BWop= 0.5
*gm*C2/((C2+Clo+2.0*gm*Ron*C2)*(C11+C12+C13)+Clo*(C2+Cp2));
                        S11d= 8.32e-21
*Ron*(BW11d*Periodo*((Taui/Periodo)*(Taui/Periodo))+Taus/Periodo);
                        S11= 8.32e-21 *Ron*(BW11
*Periodo*((Taui/Periodo)*(Taui/Periodo))+Taus/Periodo);
                        S12d= 8.32e-21
*Ron*(BW12d*Periodo*((Taui/Periodo)*(Taui/Periodo))+Taus/Periodo);
                        S12= 8.32e-21 *Ron*(BW12
*Periodo*((Taui/Periodo)*(Taui/Periodo))+Taus/Periodo);
                        S13d= 8.32e-21
*Ron*(BW13d*Periodo*((Taui/Periodo)*(Taui/Periodo))+Taus/Periodo);
                        S13= 8.32e-21 *Ron*(BW13
*Periodo*((Taui/Periodo)*(Taui/Periodo))+Taus/Periodo);

        Sop=INPSD*INPSD*(BWop*Periodo*((Taus/Periodo)*(Taus/Periodo))+Taui/Periodo);
                        S2= 16.64e-21 *Ron*(BW2
*Periodo*((Taus/Periodo)*(Taus/Periodo))+Taui/Periodo);
                        delta1=$sqrt(12.0 *(S11d+S11+Sop+S2)/Periodo);
                        delta2=$sqrt(12.0 *(S12d+S12+Sop+S2)/Periodo);
                        delta3=$sqrt(12.0 *(S13d+S13+Sop+S2)/Periodo);
```

```
                        j=1.0;
                        media=Peso1;
                        std=sigma;
                        for(i=0.0;media<=$abs(5000000.0) && std
<=$abs(5000000.0);i=i+1.0) begin
                                media=media*10.0;
                                std=std*10.0;
                                j=j*10;
                        end

        W1=$itor($dist_normal(seedi,$rtoi(media/10.0),$rtoi(std/10.0)))/(j/10.0);
                        j=1.0;
                        media=Peso2;
                        std=sigma;
                        for(i=0.0;media<=$abs(5000000.0) && std
<=$abs(5000000.0);i=i+1.0) begin
                                media=media*10.0;
                                std=std*10.0;
                                j=j*10;
                        end

        W2=$itor($dist_normal(seedi,$rtoi(media/10.0),$rtoi(std/10.0)))/(j/10.0);
                        j=1.0;
                        media=Peso3;
                        std=sigma;
                        for(i=0.0;media<=$abs(5000000.0) && std
<=$abs(5000000.0);i=i+1.0) begin
                                media=media*10.0;
                                std=std*10.0;
                                j=j*10;
                        end

        W3=$itor($dist_normal(seedi,$rtoi(media/10.0),$rtoi(std/10.0)))/(j/10.0);
                        aux=0.0;
                end
                if (clock) begin
                        if(Ceqs>0.0 && int_type) begin
                                Vas= Va2-Cn1/Ceqs*(vc22-$bitstoreal(Xvcn1))-
Cn2/Ceqs*(vc22-$bitstoreal(Xvcn2))-Cn3/Ceqs*(vc22-$bitstoreal(Xvcn3));
                                if ($abs(Vas)<=Imax/gm)
                                        Va1=$exp(-gm*Periodo/(2.0*Ceqs))*Vas;
                                else begin
                                        ts=$abs(Vas)*Ceqs/Imax-Ceqs/gm;
                                        if (ts<Periodo/2.0)
                                                Va1=$exp(-gm/Ceqs*(Periodo/2.0-
ts))*$sign(Vas)*Imax/gm;
                                        else
                                                Va1=Vas-$sign(Vas)*Imax/Ceqs*Periodo/2.0;
                                end
                        end
```

```
                                vc21=vc22+(1.0+Cp2/C2)*(Va1-Va2);
                                Xout=$realtobits(vc21);
                        end
                        else begin
                                vruido1a=$itor($dist_uniform(seedi,-
5000000,5000000))*delta1/10000000;
                                vruido2a=$itor($dist_uniform(seedi,-
5000000,5000000))*delta2/10000000;
                                vruido1b=$itor($dist_uniform(seedi,-
5000000,5000000))*delta1/10000000;
                                vruido2b=$itor($dist_uniform(seedi,-
5000000,5000000))*delta2/10000000;
                                vruido3a=$itor($dist_uniform(seedi,-
5000000,5000000))*delta3/10000000;
                                vruido3b=$itor($dist_uniform(seedi,-
5000000,5000000))*delta3/10000000;
                                if (Ron> 0.0) begin
                                        vc11a = (1.0-$exp(-1.0*Periodo/(2.0*Ron*C11)))*(-1.0 *
$bitstoreal(Xin11)*(1.0+alpha/2.0*
$bitstoreal(Xin11)+beta/3.0*$bitstoreal(Xin11)*$bitstoreal(Xin11)) + vruido1a);
                                        vc11b = (1.0-$exp(-1.0*Periodo/(2.0*Ron*C12)))*(-1.0 *
$bitstoreal(Xin21)*(1.0+alpha/2.0*
$bitstoreal(Xin21)+beta/3.0*$bitstoreal(Xin21)*$bitstoreal(Xin21)) + vruido1b);
                                        vc12a = (1.0-$exp(-1.0*Periodo/(2.0*Ron*C11)))*(-1.0 *
$bitstoreal(Xin12)*(1.0+alpha/2.0*
$bitstoreal(Xin12)+beta/3.0*$bitstoreal(Xin12)*$bitstoreal(Xin12)) + vruido2a);
                                        vc12b = (1.0-$exp(-1.0*Periodo/(2.0*Ron*C12)))*(-1.0 *
$bitstoreal(Xin22)*(1.0+alpha/2.0*
$bitstoreal(Xin22)+beta/3.0*$bitstoreal(Xin22)*$bitstoreal(Xin22)) + vruido2b);
                                        vc11c = (1.0-$exp(-1.0*Periodo/(2.0*Ron*C13)))*(-1.0 *
$bitstoreal(Xin31)*(1.0+alpha/2.0*
$bitstoreal(Xin31)+beta/3.0*$bitstoreal(Xin31)*$bitstoreal(Xin31)) + vruido3a);
                                        vc12c = (1.0-$exp(-1.0*Periodo/(2.0*Ron*C13)))*(-1.0 *
$bitstoreal(Xin32)*(1.0+alpha/2.0*
$bitstoreal(Xin32)+beta/3.0*$bitstoreal(Xin32)*$bitstoreal(Xin32)) + vruido3a);
                                end
                                else begin
                                        vc11a = -1.0*$bitstoreal(Xin11)*(1.0+alpha/2.0*
$bitstoreal(Xin11)+beta/3.0*$bitstoreal(Xin11)*$bitstoreal(Xin11)) + vruido1a;
                                        vc11b = -1.0*$bitstoreal(Xin21)*(1.0+alpha/2.0*
$bitstoreal(Xin21)+beta/3.0*$bitstoreal(Xin21)*$bitstoreal(Xin21)) + vruido1b;
                                        vc12a = -1.0*$bitstoreal(Xin12)*(1.0+alpha/2.0*
$bitstoreal(Xin12)+beta/3.0*$bitstoreal(Xin12)*$bitstoreal(Xin12)) + vruido2a;
                                        vc12b = -1.0*$bitstoreal(Xin22)*(1.0+alpha/2.0*
$bitstoreal(Xin22)+beta/3.0*$bitstoreal(Xin22)*$bitstoreal(Xin22)) + vruido2b;
                                        vc11c = -1.0*$bitstoreal(Xin31)*(1.0+alpha/2.0*
$bitstoreal(Xin31)+beta/3.0*$bitstoreal(Xin31)*$bitstoreal(Xin31)) + vruido3a;
                                        vc12c = -1.0*$bitstoreal(Xin32)*(1.0+alpha/2.0*
$bitstoreal(Xin32)+beta/3.0*$bitstoreal(Xin32)*$bitstoreal(Xin32)) + vruido3b;
                                end
```

```
                        if(Ceqs>0.0 && int_type) begin
                                Vai= (1.0+Clo/C2)/Ceqi*(C11*($bitstoreal(Xin12)-
$bitstoreal(Xin11))+C12*($bitstoreal(Xin22)-$bitstoreal(Xin21))+C13*($bitstoreal(Xin32)-
$bitstoreal(Xin31)))+Cprima/Ceqi*Va1;
                                if ($abs(Vai)<= Imax/gm)
                                        Va2=$exp(-gm*Periodo/(Ceqi*2.0))*Vai;
                                else begin
                                        ti=$abs(Vai)*Ceqi/Imax-Ceqi/gm;
                                        if (ti<=Periodo/2.0)
                                                Va2=$exp(-gm/Ceqi*(Periodo/2.0-
ti))*$sign(Vai)*Imax/gm;
                                        else
                                                Va2=Vai-$sign(Vai)*Imax/Ceqi*Periodo/2.0;
                                end
                        end
                        vc22 = (vc21*(A0+1.0) + W1*A0*(vc12a-vc11a)+W2*A0*(vc12b-
vc11b)+W3*A0*(vc12c-
vc11c))/(A0+1.0+W1+W2+W3)+(1.0+Cp2/C2+Peso1+Peso2+Peso3)*Va2+(1.0+Cp2/C2)*Va1;
                        A0_nl=A0;A0_nl_var=0.0;iter=0.0;
                        for(iter=0; $abs((A0_nl-A0_nl_var)/A0_nl)>=0.01;iter=iter+1.0)
                                if( iter <=50.0 ) begin
                                        A0_nl_var=A0_nl;

        A0_nl=A0+A0*A01*vc22+A0*A02*vc22*vc22+A0*A03*vc22*vc22*vc22+A0*A04*vc2
2*vc22*vc22*vc22;
                                        vc22 = (vc21*(A0_nl+1.0) + W1*A0_nl*(vc12a-
vc11a)+W2*A0_nl*(vc12b-vc11b)+W3*A0_nl*(vc12c-
vc11c))/(A0_nl+1.0+W1+W2+W3)+(1.0+Cp2/C2+Peso1+Peso2+Peso2)*Va2+(1.0+Cp2/C2)*Va1
;
                                        if (vc22>= osp)
                                                vc22 = osp;
                                        else if (vc22 <= osn)
                                                vc22= osn;
                                end
                        Xout=$realtobits(vc22);
                        Xvc1=$realtobits(-($bitstoreal(Xin12)+Va2));
                        Xvc2=$realtobits(-($bitstoreal(Xin22)+Va2));
                        Xvc3=$realtobits(-($bitstoreal(Xin32)+Va2));
                end
        end
endmodule
```

# E.2. Código VERILOG del integrador

```
// Entity Integrador de tres rama
// ENTRADAS Xin11,Xin21,Xin22,Xin12,xin31,Xin32,clock,Reset
//        COMO PARAMETROS
//                Peso1,Peso2,Peso3 Pesos de las ramas
//                A0      Ganancia en DC del OTA
//                A01,A02,A03,A04     Parametros de la ganancia no lineal en tanto porciento de la ganancia total
//                gm transconductancia del amplficador
//                Cp1,Cp2         Capacidades parasitas
//                Ron     Resistencia de llave
//                INPSD Densisdad espectral de ruido a la entrada
//                alpha,beta coeficientes no lineales de los condensadores en tanto por uno
//                sigma desviacion en el peso en tanto por uno
// SALIDAS Xout,Xvc1
// ERRORES IMPLEMENTADOS
//        ganancia finita y no lineal, condensadores no lineales
//        error de settling, error de desapareamiento, ruido termico
// PARAMETROS DE ENTRADA
//                VDD   Tension de rail positivo de entrada
//                VSS   Tension de rail negativo de entrada
//                INT_TYPE =1 REAL
//                C2      Capacidad de integracion
//                dc      ciclo de trabajo
//                ird     reduccion de integracion
//                srd     reduccion de muestro
//                TS      Periodo de reloj
// VARIABLES LOCALES
//                Vai,Vas,Va1,Va2,Va_ant      Tensiones en la pata negativa del integrador
//                        debida al settling
//                vc11,vc12,vc21,vc22 Tensiones de los condensadores en las fases de muestreo e integracion
//                Ceqs,Ceqi,Cprima Capacidades para calculer el settling
//                Vout,Vin1,Vin2 Variables auxiliares que almacenan los valores de salida y entrada
//                Vant    Almacena el valor anterior de la salida
//                C1      Variable del condensador de muestreo
//                A0_nl   Ganancia no lineal
//                A0_nl_var Varianza de la ganancia no lineal
//                iter    Valor de la iteraccion
//                S1d     PSD de ruido de llave phi1d
//                S1      PSD de ruido de llave phi1
//                Sop     PSD de ruido del OPAMP
//                S2      PSD de ruido de llave phi2
//                BW1d    Ancho de banda equivalente de phi1d
//                BW1     Ancho de banda equivalente de phi1
//                BWop    Ancho de banda equivalente del OPAMP
//                BW2     Ancho de banda equivalente de phi2
//                Taui,Taus       Tiempo efectivo de muestro y de integracion
//                Vruido Tension equivalente de ruido
```

```
//              seed1,seed2     Semilla para la generacion de variables aleatorias

moduleintegrador3M(Reset,clock,Xin11,Xin12,Xin21,Xin22,Xin31,Xin32,Xvcn1,
                        Xvcn2,Xvcn3,Xout,Xvc1,Xvc2,Xvc3,Peso1p,Peso2p,Peso3p,
                        A0p,A01p,A02p,A03p,A04p,Cp11p,Cp12p,Cp13p,Cp2p,Clop,
                        sigmap,alphap,betap,Ronp,gmp,INPSDp,Imaxp,dcp,irdp,srdp);
        input Reset,clock;
        input   [63 : 0] Xin11,Xin12,Xin21,Xin22,Xin31,Xin32,Xvcn1,Xvcn2,Xvcn3;
        input [63 : 0] Peso1p,Peso2p,Peso3p,A0p,A01p,A02p,A03p,A04p,Cp11p;
        input [63 : 0] Cp12p,Cp13p,Cp2p,Clop,sigmap,alphap,betap,Ronp,gmp;
        input [63 : 0]INPSDp,Imaxp,dcp,irdp,srdp;
        output [63 : 0] Xout,Xvc1,Xvc2,Xvc3;
        reg [63 : 0 ] Xout,Xvc1,Xvc2,Xvc3;
        parameter int_type=1.0,osp=1.0e12,osn=-1.0e12;
        parameter C2=2.0e-12,Periodo=4.0e-7,Cn1=0.0,Cn2=0.0,Cn3=0.0;
        real vc11a,vc11b,vc21,vc12a,vc12b,vc11c,vc12c,vc22,A0_nl,A0_nl_var;
        real iter,C11,C12,C13,Vout_ant,Taui,Taus,S11d,S11,S12d,S12,S13d,S13;
        real Sop,S2,BW11d,BW11,BW12d,BW12,BW13d,BW13,BWop,BW2,delta1,delta2;
        real delta3,vruido1a,vruido2a,vruido1b,vruido2b,vruido3a,vruido3b;
        real vout_v,Vai,Vas,Va1,Va2,Va_ant,Ceqs,Ceqi,Cprima,aux,ts,ti,W1,W2,W3;
        real Peso1,Peso2,Peso3,A0,A01,A02,A03,A04,Cp11,Cp12,Cp13,Cp2,Clo;
        real    sigma,alpha,beta,Ron,gm,INPSD,Imax,dc,ird,srd;
        real i,j,media,std;
        integer seedi;
        initial begin
                seedi=1;
                Vai=0.0;
                Vas=0.0;
                Va1=0.0;
                Va2=0.0;
                vc22=0.0;
                vc21=0.0;
                Xout=$realtobits(0.0);
                Xvc1=$realtobits(0.0);
                Xvc2=$realtobits(0.0);
                Xvc3=$realtobits(0.0);
                aux=1.0;
                Peso1=$bitstoreal(Peso1p);
                Peso2=$bitstoreal(Peso2p);
                Peso3=$bitstoreal(Peso3p);
                A0=$bitstoreal(A0p);
                A01=$bitstoreal(A01p);
                A02=$bitstoreal(A02p);
                A03=$bitstoreal(A04p);
                A04=$bitstoreal(A04p);
                Cp11=$bitstoreal(Cp11p);
                Cp12=$bitstoreal(Cp12p);
                Cp13=$bitstoreal(Cp13p);
                Cp2=$bitstoreal(Cp2p);
                Clo=$bitstoreal(Clop);
```

```
                sigma=$bitstoreal(sigmap);
                alpha=$bitstoreal(alphap);
                beta=$bitstoreal(betap);
                Ron=$bitstoreal(Ronp);
                gm=$bitstoreal(gmp);
                INPSD=$bitstoreal(INPSDp);
                Imax=$bitstoreal(Imaxp);
                dc=$bitstoreal(dcp);
                ird=$bitstoreal(irdp);
                srd=$bitstoreal(srdp);
                C11=Peso1*C2;
                C12=Peso2*C2;
                C13=Peso3*C2;
        end
        always @(posedge clock)
// Sincronizacion en cada semiperiodo de reloj
                if(Reset) begin
// Inicializacion de parametros
                        Xout=$realtobits(0.0);
                        Xvc1=$realtobits(0.0);
                        Xvc2=$realtobits(0.0);
                        Xvc3=$realtobits(0.0);
                        vc22 =0.0;
                        Vai = 0.0;
                        Vas = 0.0;
                        Va1= 0.0;
                        Va2 = 0.0;
                        vc21 = 0.0;
                        aux=1.0;
                        Peso1=$bitstoreal(Peso1p);
                        Peso2=$bitstoreal(Peso2p);
                        Peso3=$bitstoreal(Peso3p);
                        A0=$bitstoreal(A0p);
                        A01=$bitstoreal(A01p);
                        A02=$bitstoreal(A02p);
                        A03=$bitstoreal(A04p);
                        A04=$bitstoreal(A04p);
                        Cp11=$bitstoreal(Cp11p);
                        Cp12=$bitstoreal(Cp12p);
                        Cp13=$bitstoreal(Cp13p);
                        Cp2=$bitstoreal(Cp2p);
                        Clo=$bitstoreal(Clop);
                        sigma=$bitstoreal(sigmap);
                        alpha=$bitstoreal(alphap);
                        beta=$bitstoreal(betap);
                        Ron=$bitstoreal(Ronp);
                        gm=$bitstoreal(gmp);
                        INPSD=$bitstoreal(INPSDp);
                        Imax=$bitstoreal(Imaxp);
                        dc=$bitstoreal(dcp);
```

```
                ird=$bitstoreal(irdp);
                srd=$bitstoreal(srdp);
        end
        else begin
                if (aux) begin
// Inicializacion de parametros
                        W1=Peso1;
                        W2=Peso2;
                        W3=Peso3;
                        Taui = Periodo*(1.0 -dc)-ird;
                        Taus = Periodo*dc-srd;
                        Ceqs= Cp2+(Clo+Cn1+Cn2+Cn3)*(1.0+Cp2/C2);
                        Ceqi=
Cp2+C11+C12+C13+Clo*(1.0+Cp2/C2+Peso1+Peso2+Peso3);
                        Cprima= Cp2+Clo*(1.0+Cp2/C2);
// Calculo de la densidad espectral de ruido termico
                        if (Ron> 0.0) begin
                                BW11d= 1.0 /(2.0 *Ron*(2.0*C11+Cp11));
                                BW12d= 1.0 /(2.0 *Ron*(2.0*C12+Cp12));
                                BW13d= 1.0 /(2.0 *Ron*(2.0*C13+Cp13));
                                BW11=(C11+Cp11)/(2.0 *Ron*C11*(2.0 *C11+Cp11));
                                BW12=(C12+Cp12)/(2.0 *Ron*C12*(2.0 *C12+Cp12));
                                BW13=(C13+Cp13)/(2.0 *Ron*C13*(2.0 *C13+Cp13));
                                BW2=
0.25*(Clo*Cp2+C2*Cp2+2.0*(C11+C12+C13)*gm*Ron*C2+Clo*C2)/(((C2+Clo+2.0*gm*Ron*
C2)*(C11+C12+C13)+Clo*C2+Clo*Cp2+C2*Cp2)*Ron*(C11+C12+C13));
                        end
                        else begin BW11d= 0.0;BW12d= 0.0;BW11= 0.0;BW12=
0.0;BW13d= 0.0;BW13= 0.0;BW2= 0.0; end
                        BWop=                                   0.5
*gm*C2/((C2+Clo+2.0*gm*Ron*C2)*(C11+C12+C13)+Clo*(C2+Cp2));
                        S11d=                                   8.32e-21
*Ron*(BW11d*Periodo*((Taui/Periodo)*(Taui/Periodo))+Taus/Periodo);
                        S11=                    8.32e-21                *Ron*(BW11
*Periodo*((Taui/Periodo)*(Taui/Periodo))+Taus/Periodo);
                        S12d=                                   8.32e-21
*Ron*(BW12d*Periodo*((Taui/Periodo)*(Taui/Periodo))+Taus/Periodo);
                        S12=                    8.32e-21                *Ron*(BW12
*Periodo*((Taui/Periodo)*(Taui/Periodo))+Taus/Periodo);
                        S13d=                                   8.32e-21
*Ron*(BW13d*Periodo*((Taui/Periodo)*(Taui/Periodo))+Taus/Periodo);
                        S13=                    8.32e-21                *Ron*(BW13
*Periodo*((Taui/Periodo)*(Taui/Periodo))+Taus/Periodo);

        Sop=INPSD*INPSD*(BWop*Periodo*((Taus/Periodo)*(Taus/Periodo))+Taui/Periodo);
                        S2=                     16.64e-21               *Ron*(BW2
*Periodo*((Taus/Periodo)*(Taus/Periodo))+Taui/Periodo);
                        delta1=$sqrt(12.0 *(S11d+S11+Sop+S2)/Periodo);
                        delta2=$sqrt(12.0 *(S12d+S12+Sop+S2)/Periodo);
                        delta3=$sqrt(12.0 *(S13d+S13+Sop+S2)/Periodo);
```

```
                aux=0.0;
            end
            if (clock) begin
// Fase de muestreo
                if(Ceqs>0.0 && int_type) begin
// Calculo del settlin de la fase de muestreo
                    Vas=        Va2-Cn1/Ceqs*(vc22-$bitstoreal(Xvcn1))-
Cn2/Ceqs*(vc22-$bitstoreal(Xvcn2))-Cn3/Ceqs*(vc22-$bitstoreal(Xvcn3));
                    if ($abs(Vas)<=Imax/gm)
                        Va1=$exp(-gm*Periodo/(2.0*Ceqs))*Vas;
                    else begin
                        ts=$abs(Vas)*Ceqs/Imax-Ceqs/gm;
                        if (ts<Periodo/2.0)
                            Va1=$exp(-gm/Ceqs*(Periodo/2.0-
ts))*$sign(Vas)*Imax/gm;
                        else
                            Va1=Vas-$sign(Vas)*Imax/Ceqs*Periodo/2.0;
                    end
                end
// Ecuacion de balance para la fase de muestreo
                    vc21=vc22+(1.0+Cp2/C2)*(Va1-Va2);
                    Xout=$realtobits(vc21);
            end
            else begin
// Fase de integracion
// Calculo de la tension equivalente de ruido para cada rama
                vruido1a=$itor($dist_uniform(seedi,-
5000000,5000000))*delta1/10000000;
                vruido2a=$itor($dist_uniform(seedi,-
5000000,5000000))*delta2/10000000;
                vruido1b=$itor($dist_uniform(seedi,-
5000000,5000000))*delta1/10000000;
                vruido2b=$itor($dist_uniform(seedi,-
5000000,5000000))*delta2/10000000;
                vruido3a=$itor($dist_uniform(seedi,-
5000000,5000000))*delta3/10000000;
                vruido3b=$itor($dist_uniform(seedi,-
5000000,5000000))*delta3/10000000;
// Calculo de las tensiones de los condensadores de muestreo
// Incluyendo la no linealidad y el la resistencia en on
                if (Ron> 0.0) begin
                    vc11a   =   (1.0-$exp(-1.0*Periodo/(2.0*Ron*C11)))*(-1.0   *
$bitstoreal(Xin11)*(1.0+alpha/2.0*
$bitstoreal(Xin11)+beta/3.0*$bitstoreal(Xin11)*$bitstoreal(Xin11)) + vruido1a);
                    vc11b   =   (1.0-$exp(-1.0*Periodo/(2.0*Ron*C12)))*(-1.0   *
$bitstoreal(Xin21)*(1.0+alpha/2.0*
$bitstoreal(Xin21)+beta/3.0*$bitstoreal(Xin21)*$bitstoreal(Xin21)) + vruido1b);
                    vc12a   =   (1.0-$exp(-1.0*Periodo/(2.0*Ron*C11)))*(-1.0   *
$bitstoreal(Xin12)*(1.0+alpha/2.0*
$bitstoreal(Xin12)+beta/3.0*$bitstoreal(Xin12)*$bitstoreal(Xin12)) + vruido2a);
```

```
                                vc12b = (1.0-$exp(-1.0*Periodo/(2.0*Ron*C12)))*(-1.0 *
$bitstoreal(Xin22)*(1.0+alpha/2.0*
$bitstoreal(Xin22)+beta/3.0*$bitstoreal(Xin22)*$bitstoreal(Xin22)) + vruido2b);
                                vc11c = (1.0-$exp(-1.0*Periodo/(2.0*Ron*C13)))*(-1.0 *
$bitstoreal(Xin31)*(1.0+alpha/2.0*
$bitstoreal(Xin31)+beta/3.0*$bitstoreal(Xin31)*$bitstoreal(Xin31)) + vruido3a);
                                vc12c = (1.0-$exp(-1.0*Periodo/(2.0*Ron*C13)))*(-1.0 *
$bitstoreal(Xin32)*(1.0+alpha/2.0*
$bitstoreal(Xin32)+beta/3.0*$bitstoreal(Xin32)*$bitstoreal(Xin32)) + vruido3a);
                        end
                        else begin
                                vc11a = -1.0*$bitstoreal(Xin11)*(1.0+alpha/2.0*
$bitstoreal(Xin11)+beta/3.0*$bitstoreal(Xin11)*$bitstoreal(Xin11)) + vruido1a;
                                vc11b = -1.0*$bitstoreal(Xin21)*(1.0+alpha/2.0*
$bitstoreal(Xin21)+beta/3.0*$bitstoreal(Xin21)*$bitstoreal(Xin21)) + vruido1b;
                                vc12a = -1.0*$bitstoreal(Xin12)*(1.0+alpha/2.0*
$bitstoreal(Xin12)+beta/3.0*$bitstoreal(Xin12)*$bitstoreal(Xin12)) + vruido2a;
                                vc12b = -1.0*$bitstoreal(Xin22)*(1.0+alpha/2.0*
$bitstoreal(Xin22)+beta/3.0*$bitstoreal(Xin22)*$bitstoreal(Xin22)) + vruido2b;
                                vc11c = -1.0*$bitstoreal(Xin31)*(1.0+alpha/2.0*
$bitstoreal(Xin31)+beta/3.0*$bitstoreal(Xin31)*$bitstoreal(Xin31)) + vruido3a;
                                vc12c = -1.0*$bitstoreal(Xin32)*(1.0+alpha/2.0*
$bitstoreal(Xin32)+beta/3.0*$bitstoreal(Xin32)*$bitstoreal(Xin32)) + vruido3b;
                        end
                        if(Ceqs>0.0 && int_type) begin
// Calculo del settling para la fase de integracion
                                Vai= (1.0+Clo/C2)/Ceqi*(C11*($bitstoreal(Xin12)-
$bitstoreal(Xin11))+C12*($bitstoreal(Xin22)-$bitstoreal(Xin21))+C13*($bitstoreal(Xin32)-
$bitstoreal(Xin31)))+Cprima/Ceqi*Va1;
                                if ($abs(Vai)<= Imax/gm)
                                        Va2=$exp(-gm*Periodo/(Ceqi*2.0))*Vai;
                                else begin
                                        ti=$abs(Vai)*Ceqi/Imax-Ceqi/gm;
                                        if (ti<=Periodo/2.0)
                                                Va2=$exp(-gm/Ceqi*(Periodo/2.0-
ti))*$sign(Vai)*Imax/gm;
                                        else
                                                Va2=Vai-$sign(Vai)*Imax/Ceqi*Periodo/2.0;
                                end
                        end
// Ecuacion de balance para la fase de integracion
                        vc22 = (vc21*(A0+1.0) + W1*A0*(vc12a-vc11a)+W2*A0*(vc12b-
vc11b)+W3*A0*(vc12c-
vc11c))/(A0+1.0+W1+W2+W3)+(1.0+Cp2/C2+Peso1+Peso2+Peso3)*Va2+(1.0+Cp2/C2)*Va1;
                        A0_nl=A0;A0_nl_var=0.0;iter=0.0;
                        for(iter=0; $abs((A0_nl-A0_nl_var)/A0_nl)>=0.01;iter=iter+1.0)
                                if( iter <=50.0 ) begin
// Calculo de la ganancia no lineal
                                        A0_nl_var=A0_nl;
```

```
        A0_nl=A0+A0*A01*vc22+A0*A02*vc22*vc22+A0*A03*vc22*vc22*vc22+A0*A04*vc22*vc22*vc22*vc22;
                                                vc22 = (vc21*(A0_nl+1.0) + W1*A0_nl*(vc12a-vc11a)+W2*A0_nl*(vc12b-vc11b)+W3*A0_nl*(vc12c-
vc11c))/(A0_nl+1.0+W1+W2+W3)+(1.0+Cp2/C2+Peso1+Peso2+Peso2)*Va2+(1.0+Cp2/C2)*Va1;
// Limitacion de la tension de salida del integrador
                                                if (vc22>= osp)
                                                        vc22 = osp;
                                                else if (vc22 <= osn)
                                                        vc22= osn;
                                        end
// Asignacion a las salidas la tension de los condensadores de muestreo
                                Xout=$realtobits(vc22);
                                Xvc1=$realtobits(-($bitstoreal(Xin12)+Va2));
                                Xvc2=$realtobits(-($bitstoreal(Xin22)+Va2));
                                Xvc3=$realtobits(-($bitstoreal(Xin32)+Va2));
                        end
                end
endmodule
```

# REFERENCIAS

[1] R. Stee'le: "Delta Modulation System", Wiley, New Cork 1975.

[2] J. C. Candy and G. C. Temes, (Editors): "Oversampling Delta – Sigma Converters". IEEE Press, 1992.

[3] H. Inose, Y. Yasuda and J.Murakami:"A Telemetering System By code Modulation - $\Delta - \Sigma$ Modulation", IRE Transactions on Space Electronics and Telemetry, Vol.8, pp.204-209.Septiembre, 1962.

[4] C. A. Leme: Oversampled Interfaces for IC Sensors, ETH Press, Zurich, 1993.

[5]B.P. Del Signore D. A. Perth, N. S. Sooch and E. j. Swanson: "A Monolithic 20-b Delta – Sigma A/D Converter", IEE Journal of Solid – State Circuits, Vol. 25, pp.1311 – 1317, December 1990.

[6]B. Brandt and B. A. Wooley: "A 50 – MHz Multibit $\Sigma\Delta$ Modulator for 12-v 2-MHz A/D Conversion", IEEE Journal of Solid – State Circuits, Col. 26, pp. 1746 – 1756, December 1991.

[7]R. T. Baird and T. S. Fiez: "A Low Oversampling Ratio 14-b 500-kHz DS ADC with a Self – Calibrated Multibit DAC", IEEE Journal of Solid – State Circuits, Vol. 23. pp. 1298 – 1308, December 1988.

[8]F. Op'T eybde, G. Yin and W. Sansen: " A CMOS Fourth – Order 14b 500K Sample/s Sigma – Delta ADC converter", in Proc. of IEEE International Solid – Satate Circuits Converence, pp. 62-63, 1991.

[9]G. Yin and W. Sansen : "A High – Frecuency and High – Resolutions Fourth – Order $\Sigma\Delta$ A/D Converter in Bi-CMOS technology", IEEE Journal of Solid – State Circuits, Vol 29, pp. 857-865, August 1994.

[10]B. E. Boser and B. A. Wooley: "The Design of Sigma – Delta Modulation Analog-to-Digital Converters", IEEE Journal of Solid – State Circuits, Vol. 23, pp,1298 – 1308, December 1988.

[11]H. Baher and E. Afifi : "Novel Fouth – Order Sigma- Delta converter" electronics Letters, Vol. 28, pp. 1437-1438, july 1992.

[12]L. R. Carley: " A Noise-Ghaping Coder topology for 15+ Bit converters". IEEE Journal of solid – State Circuits, Vol 24, pp. 267-273, April 1989.

[13]K. C. H. Chao et al. : » A Higher Order Topology for Interpolative Modulators for Oversampling A/D Converters », IEEE Transactions on Circuits and Systems, Vol 37, pp. 309-318, March 1990.

[14]T. Hayashi et al.:"A Multi-Stage Delta – Sigma Moduldator without Double Integration Loop", in Proc. of IEEE International Solid – State Circuits Conference, pp. 182-183, February 1986.

[15]W. L. Lee and C.G.Sodini:"ATopology for Higher Order Interpolative Coders", in Proc. of IEEE International Symposium on Circuits and systems, pp. 459 – 462, 1987.

[16]B. H. Leung and S.Sutarja:"Multi – bit ΣΔ A/D Converter Incorporating a Novel class off Dynamic Element Matching Techniques", IEEE Transactions on Circuit and systems – II, Vol 39, pp 35-51, January 1992.

[17]H. Leopold, G. Winkler, P. O'Leary, K. Ilzer and J. Jernej:"A Monolitic 20 bit Analog to Digital Converter", IEEE Journal of Solid – State Circuits, Vol. 26, July 1991.

[18]O. Nys and r. Henderson: "A monolithic 19bit 800Hz Low- Power Multbit Sigma Delta CMOS ADC using Data Weighted Averaging", in Proc. of European Solid – Satate Circuits Conference, pp. 252 – 255. 1996.

[19]B. Brandt, d. W. wingard and B. A. Wooley:"Second Order Sigma – Delta Modulation for Digital – Audio Signal Acquisition", IEEE Journal of Solid – State circuits, Vol 23. pp. 618 – 627, aprl 1991.

[20]P. Ferguson, Jr. Et al.:"An 19b 20KHZ Dual ΣΔ A/D Converter", in Proc. of IEEE International Solid-State Circuits Conference, pp 68-69, February 1991.

[21]L. Le toumelin et al.:"A 5-V CMOS Line controller with 16b Audio converters", IEEE Journal of Solid – State Circuits, Vol 27, pp 332-341, March 1992.

[22]G. M Win F Stubbe and W.Sansen "A 16bit 320 KH< CMOS A/D Converter using 2 – Stage 3rd – Order ΣΔ Noise-Shaping",IEEE Journal of Solid- Satate Circuits, Vol 28. pp 640-647, June 1993.

[23]Z-Y Chang, D MAcq, D. Haspeslagh, P. Spruyt and B. Goffart: " A CMOSAnalog Front-End Circuit for an FDM-Based ADSL System", IEEE Journal of Solid- State Circuits, Vol. 30, pp. 1449-1456, April 1995.

[24]M.W. Hauser and R. W. Brodersen: " Circuit and Technology Considerations for MOS Delta – Sigma A/D Converters", in Proc. of IEEE International symposium on circuits and Systems, May 1986.

[25]A. Yukawa: "Constraints Analysys for Oversampling A-to-D Converter Structures on VLSI Implementation", in Proc. of IEEE International Symposium on Circuits and Systems, pp. 467-472,1987.

[26]S.R. Norsworthy, R.Schereider and G. C. Temes, (Editors): "Delta – Sigma Data Converters:Theory, Design and Simulation", IEEE Press, New York 1997.

[27]W.Bennett: “Spectra of Quantized Signals”,Bell Syst. Tech. J., Vol 27, pp 446-472, Juky 1948.

[28]Fernando Medeiro, Angel Pérez – Verdú and Angel Rodríguez-Vazquez “Top – Down Design of High – Performance Sigma – Delta Modulators” Kluwer Academia Publishers1999.

[29]J.C. Candy: “ A Use of Double Integration in Sigma – Delta Modulation”:, IEEE Transactions on Communications, Vol 33, pp. 249 – 258, March 1985.

[30]R. W. Adams and R.Schreider: “Stability Theory for Sigma – Delta Modulators”, Chapter 4 in the book “Delta – sigma Data Converters: Theory, Design and Simulation (s. R. Norswhorthy, R.Schreider and G. C. Temes, Editors)”,pp 141 – 163, IEEE Press, New York 1997.

[31] W. Chou, P. Wong and R. Gray: “Multi – Stage Sigma – Delta Modulation”, IEEE Transactions on Information Theory, Vol 35, pp. 784 – 796, July 1989.

[32] L. Longo and M. A. Copeland: “A 13-bit ISDN- band ADC using Two Stage Third – Order Noise Shaping”, in Proc. Of IEEE Custom Integrated Circuit Conference, pp.316-317, 1996.

[33] Y. Matsuya et al.:”A 16-bit Oversampling A-to-D Conversion Technology Using Triple – Integration Noise Shaping”, IEEE Journal of Solid State Circuits, Vol. 22, pp 921 – 929. December 1987.

[34] M. Rebeschini et al.:”A 16 – b 160 kHz CMOD A/D Converter using Sigma – Delta Modulation », IEEE Journal of Solid State Circuits, Vol 25, pp 431-440 , April 1990.

[35] G. M. Yin and W. Sansen:”A High – Frecuency and High Resolution Fourth – Order Sigma – Delta A/D Converter in Bi – Cmos Technology”, IEEE Journal of Solid – State Circuits, Vol 29, pp. 857 – 865, August 1994.

[36]J. J. Paulos, G. T. Brauns, M. B. Steer and S. H. Ardalan:”Improved Signal-to-Noise Ratio Using Tri-Level Delta-Sigma Modulation”, in Proc. Of IEEE International Symposium on Circuits and Systems, pp. 463-466, 1987.

[37]L. R. Carley, R. Schreider and G. C. Temes:”Delta – Sigma ADCs with Multibit Internal Converters”, Chapter 8 in the book “Delta – Sigma Data Converters: Theory, Design and Simulation (S. R. Norswhorthy, R. Schreier and G. C. Temes, Editors)”, pp.244-281, IEEE Press, New Work 1997.

[38]R. W. Adams et al.:”Theory and Practical Implementation of a Fifth – Order Sigma – Delta A/D Converter”, Journal of Audio Engeneering Society, Vol 39, pp. 515-528, Juky 1991.

[39]R. J. van de Plasshe: “A Sigma – Delta Converter as and A/D Converter”, IEEE Transactions on Circuits and Systems, Vol 25, pp. 510 – 514, July 1978.

[40]K.Martin and A. S. Sedra: “Strays-insensitive Switched-Capacitor Filters for a PCM Voice Codec,”, Electronic Letters, Vol 19, pp. 365-366, June 1979

[41]W. Sansen et al. “Transient Analysis of Charge Transfer in SC Filters:Gain Error and Distorsion”, IEEE Journal of Solid-State Circuits, Vol.22, pp. 268-276, April 1987.

[42]R.del Río, F.Medeiro, J.M. de la Rosa, B.Pérez Verdú and A. Rodríguez Vázquez. “Reliable Análisis od Settling Errors in SC Integrators-Application to High-Speed Low-Power Delta-Sigma Modulators Design”. Pp 727-732. DCIS. Palma de Mallorca. Noviembre de 1999

[43]P. R. Gray and R. G. Meyer: “Analysus and Design of Analog Integrated Circuits (3$^{er}$ Edition)”, Wiley 1993

[44]M. Sarhang – Nejad, G. C. Temes : » A High – Resolution $\Sigma\Delta$ ADC with Digital Correction and Relaxed Amplidiers Requeriments”, IEEE Journal od Solid – State Circuits, Vol 28, pp 648-660, June 1993.

[45]F. Chen And B. Leung: “ A Multi – Bit $\Sigma\Delta$ DAC with Dynamic Element Matching Techniques”, in Proc. Of IEEE Custom Integrated Circuits Conference, pp. 16.2.1.-16.2.4, May 1992

[46]R. T. Baird and T. S. Fiez: “Improved $\Sigma\Delta$ DAC Lineaity Using Data Weighted Averaging”, in Proc. Of IEEE International Symposium on Circuits and Systems, pp 13-16, May 1995

[47]L. E. Nagel:”SPICE2:A Computer Program to Simulate Semiconductor Circuits”, ERL-M520, University of California, Berkeley, 1975

[48]K. Suyama et al.: “Simulation of Mixed Switched-Capacitor/Digital Networks with Signal-Driven Switches”, IEEE JOrunal of Solid-State Circuits, Vol, 25, pp 1403-1413, December 1990

[49]V. F. Dias et al.: “Design Tools for Oversampling Data Converters: Needs and Solutions”. Microelectronics Journal, Vol. 13, pp. 641-650. 1992.

[50]C. H. Wolff and L. Carley: “Simulation of a Sigma – Delta Modulators Using Behavioral Models”, in Proc. Of IEEE International Symposium on Circuits and Systems, pp. 376-379, 1990.

[51]Meta Software Inc.: “HSPICE User Manual”, 1988

[52]Anacad Computer System: ELDO: Electrical Circuit Simulator, Ulm, Germany, 1991

[53]Saber User's Guide, Analog. INC., Beaverton, OR.1987.

[54]Sudhatar Yalamanchili: "VHDL Starter's Guide", Prentice Hall, 1998

[55]Michael d. Cileth: "Modeling, Synthesis, and Rapid Prototyping with the VERILOG HDL" Prentice Hall, 1993

[56]Dan Fiztpatrick, Ira Miller: "Analog Behavioral Modeling Using the VERILOG – A language" Kluwer Academic Publisher, 1998

[57]V. F. Dias, G. Palmisano, P. O'Leary and F. Maloberti: "Fundamental Limitations of Switched – Capacitor Sigma – Delta Modulators", IEE Proceedings – G, Vol.139, pp.27-32, February 1992

[58]C. Gobet and A. Knob: "Noise Analysis of Switched Capacitor Networks", IEEE Transactions on Circuits and Systems, Vol. 30 ç. January, 1983.

[59]A. Yukawa: "Constraints Analysys for Oversampling A-to-D Converter Structurees on VLSI Implementation", in Proc. Of IEEE International Symposium on Circuits and Systems, pp. 467-472, 1987.

[60]E. Okan Brigman : "The fast Fourier Transform and its applications", Prentice Hall, 1975

[61]Mentror Graphics: "ADVance MS User'S Manual", Document 315000 for Software Version v1.3_1.1

[62]Mentror Graphics: "ADVance MS User'S Manual", Document 315000 for Software Version v1.3_1.1

[63]Mentor Graphics: "Start Here Guide for the HDL DESIGNER Series", Software Version 2001.5, 10 Septiembre 2001

[64]Javier Moreno Reina: "SDTOOLBOX: Una herramienta para la simulación de moduladores sigma – delta en el entorno de Matlab/Simulink", Universidad de Sevilla, 2002