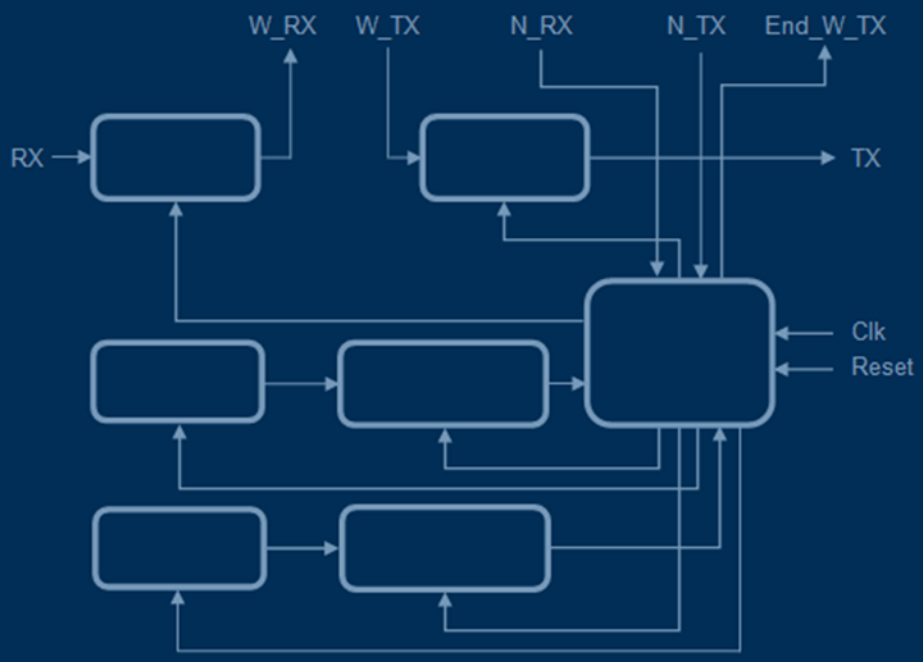
LaSalleOnLine
ENGINYERIES

RADIO DIGITAL
DISSEÑO Y TECNOLOGÍA

Apuntes

Carles Vilella i Parra



LA SALLE
ENGINYERIA I ARQUITECTURA
Universitat Ramon Llull

Creative Commons License Deed

Reconocimiento-No comercial- Sin obras derivadas 2.5 España

- **Usted es libre de:**

  Copiar, distribuir y comunicar públicamente la obra.

- **Bajo los siguientes condicionantes:**

  Reconocimiento.
  Se tiene que referenciar esta obra a Carles Vilella y Parra – Ingeniería La Salle (Semipresencial)

  No comercial.
  No se puede utilizar esta obra con fines comerciales.

  Sin obras derivadas.
  No se puede alterar, transformar o generar una obra derivada a partir de ésta.

- Cuando reutilicéis o distribuyáis la obra, tenéis que dejar bien claro los términos de la licencia de la obra.

- Alguna de estas condiciones puede no aplicarse si obtenéis el permiso del titular de los derechos de autor.

- No hay nada en esta licencia que menoscabe o restrinja los derechos morales del autor.

Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no quedan afectados por lo anterior

Esto es un resumen fácilmente legible del texto legal (la licencia completa) disponible en los idiomas siguientes:
Catalán Castellano Vasco Gallego

LaSalle OnLine ENGINYERIES
Universitat Ramon Llull

Temario

Índice de gráficos

Índice de ecuaciones

1. Introducción

1.1. Introducción a los sistemas digitales

En sus orígenes, los sistemas digitales se utilizaban básicamente para simular y verificar el correcto funcionamiento de sistemas analógico. Su tamaño y su coste así como la poca potencia que tenían, les hacían inviables para situaciones reales. Con el tiempo estos inconvenientes se fueron superando, de manera que los ingenieros empezaron a pensar en migrar algunas aplicaciones analógicas al dominio digital.

Con el fin de superar las limitaciones de velocidad de estos últimos se trabajaba en dos aspectos. Por un lado, se intensificaban las investigaciones para mejorar las arquitecturas *hardware*, haciéndolas más rápidas y eficientes. Este camino ha llevado al nacimiento de los procesadores digitales de la señal (DSP) y a otros dispositivos específicos como mezcladores digitales (DDC y DUC), delmadores y filtros.

Por otro lado, era necesario optimizar los algoritmos para que se ejecutasen con menos tiempo. Uno de los frutos más importantes de estos estudios fue el descubrimiento del algoritmo de la FFT, publicado en el año 1965, que calcula la transformada discreta de Fourier con un considerable ahorro de operaciones con respecto a implementaciones precedentes.

La evolución en estas dos vertientes de investigación, *hardware* y *software*, juntamente con la mejora de los procesos tecnológicos, han traído el dominio digital a ámbito impensables hace unos años. Un buen ejemplo, como veremos durante el curso, es la radio.

1.1.1. Digital vs Analógico

En el enseñamiento de la electrónica, ya desde los primeros cursos se hace una clara diferenciación entre las ramas analógica y digital. Asignaturas como Electrónica II, Servosistemas y Electrónica de Potencia pertenecen al primer grupo, mientras que Lógica y Ordenadores son del segundo.

La diferencia de enfoque de los dos grupos de asignaturas responde a la diferente concepción de la señal. Las señales analógicas tienen una variación continua. Aún siendo seres acotados, tienen un número infinito de valores significativos. En el caso de las señales digitales, la variación es discreta y el número de valores significativos es limitado.

Se debe enfatizar la importancia del adjetivo *significativo*. En el fondo, una señal digital también es analógica. La diferencia está en nuestra concepción. Aceptar que una señal es digital es equivalente a asimilar un número de valores infinito a uno solo. De estos últimos hay un número finito, y son precisamente estos los que para nosotros son significativos.

Desde este punto de vista, es por todos conocidos que la concepción digital es una simplificación de la concepción analógica. No obstante, es una simplificación que surge de la propia naturaleza humana. Sólo consideramos analógicos la mayoría de las señales que nos rodean: distancias, colores, tiempo,… Aún así, el hombre tiende a discretizarlos. En las disciplinas más precisas, raramente es necesaria una resolución superior a décimas de micra. Se ha demostrado empíricamente que para cuantificar un color sin pérdida de calidad apreciable hay suficiente con 24 bits. Raramente medimos tiempos inferiores a nanosegundos.

Esta simplificación, además, tiene connotaciones muy positivas en la sociedad. El adjetivo *digital* es sinónimo de calidad, y las ramas a las que abastece son cada vez más grandes. Cada vez más grandes, pero no todas. Aún hay ciertos campos reservados a la electrónica analógica: el filtrado antialiasing, la electrónica de potencia son, entre otras, disciplinas aún por conquistar. En las áreas de desarrollo comunes (la mayoría, por otro lado), la implementación digital se prefiere a la analógica para toda una serie de ventajas que a continuación analizaremos:

- Programabilidad. Con este término se hace referencia a la posibilidad que tienen las plataformas digitales de realizar diferentes tareas sin modificar el *hardware*. Así, en un sistema de filtrado digital podremos cambiar parámetros como la frecuencia de corte o la pendiente en la banda de transición modificando algunos parámetros del programa. incluso podemos ir más lejos y reprogramar el sistema para a que actúe como codificador o compresor de datos. Y todo esto sin cambio en la plataforma. Por contra, los sistemas analógicos son mucho más inflexibles a los cambios. En un filtro, por ejemplo, modificar la frecuencia de corte supondrá soldar y desoldar componentes. Si lo que pretendemos es reconvertir el funcionamiento con tal de que el sistema actúe, por ejemplo, como amplificador, la tarea será poco menos que imposible.

- Estabilidad. Los sistemas analógicos son muy sensibles a cambios de temperatura y humedad. Un amplificador tendrá, con toda seguridad, una respuesta diferente a cero grados o a cincuenta. Existen sistemas compensadores de estos efectos parásitos, pero su coste acostumbra a ser prohibitivo. Por contra, los sistemas digitales funcionan exactamente igual bajo cualquier condición meteorológica que esté dentro de los márgenes proporcionados por el fabricante.

- Repetitividad. A la hora de fabricar un producto con grandes series, los sistemas analógicos se ven afectados por las tolerancias en la

fabricación de los componentes. Así, cien filtros teóricamente idénticos tendrán con toda seguridad una respuesta diferente. Los sistemas digitales, en cambio, presentan un comportamiento totalmente repetitivo: un filtro tendrá la misma respuesta sea cual sea la plataforma que se implemente.

- Inmunidad al ruido. A causa de la propia naturaleza de las señales analógicas, el ruido les afecta siempre cambian su valor significativo. En las señales digitales, en algunos casos el ruido afectará el valor analógico pero no el significativo. Por ejemplo, si asociamos un 1 lógico a señales superiores a 2.5 voltios y un 0 lógico a valores inferiores, siempre y cuando el ruido no provoque un paso para lindar el valor significativo no cambiará.

- Implementación de algoritmos adaptativos. Entendemos por algoritmo adaptativo aquél que es capaz de modificar su tarea con tal de cumplir un objetivo. Un ejemplo típico es un adaptador de canal: obtiene información de la distorsión que en cada momento añade el canal, y filtra de manera adecuada para minimizar el error en recepción. Los algoritmos adaptativos son fácilmente implementables en una plataforma digital, mientras que son prácticamente inviables en sistemas analógicos.

- Facilidad de almacenamiento y compresión de datos. Aunque también existen medios de almacenamiento de datos analógicos (disco de vinilo, cinta de casete), ninguna de elles ofrece la facilidad y rapidez de los medios digitales. Además, los datos en formato digital pueden ser comprimidas mediante algoritmos diversos sin pérdida de calidad.

Finalmente, en lo que respecta a las ventajas, destacar que de la misma manera que hay tareas que sólo pueden implementarse de manera analógica, existen muchísimas que sólo se pueden realizar digitalmente: filtros de fase lineal o de pendiente abrupta son sólo un pequeño ejemplo.

Antes de acabar el presente apartado, comentaremos algunos de los inconvenientes inherentes a los sistemas digitales:

- Error de cuantificación. Por el hecho de identificar un conjunto infinito de valores con un conjunto finito, sufrimos una pérdida de información que recibe el número de error de cuantificación. Este error es propio de los sistemas digitales, limita el margen dinámico y no puede ser evitado; únicamente podremos evaluar sus efectos, y controlarlo aumentando o disminuyendo el paso de cuantificación.

- Precio. Los sistemas digitales que procesan señales analógicas necesitan un *hardware* mínimo para implementar cualquier proceso, por pequeño que sea. Son imprescindibles los conversores A/D y D/A así como algunos tipos de elementos de proceso (típicamente un microprocesador). Para aplicaciones muy sencillas el coste de este *hardware* mínimo será prohibitivo.

- Velocidad. Los sistemas digitales tienen un margen de frecuencias de funcionamiento muy inferior a los analógicos. Están limitados tanto por lo que son los conversores A/D y D/A (que presentan un compromiso entre frecuencia de muestreo y número de bits), como en lo que respecta a la velocidad de proceso.

1.1.2. Clasificación de los sistemas digitales computerizados

En este apartado intentaremos encuadrar los sistemas digitales que estudiaremos durante el curso mediante sucesivas clasificaciones que nos permitirán obtener una idea clara de sus características.

❖ Sistemas computerizados

Entendemos por sistemas computerizados aquéllos que son capaces de tomar decisiones; a menudo están basados en algún tipo de procesador o dispositivo similar. Una posible clasificación de los sistemas computerizados es la siguiente:

- Sistemas transformacionales. Calculan salidas en función de entradas y seguidamente se paran. Ejemplo: programas de simulación.

- Interactivos. Intercambian información con el exterior de tal manera que el sistema digital actúa como *master* y da servicios cuando estos están disponibles. Ejemplos: sistemas operativos, consultas a base de datos, etc.

- Reactivos. Reaccionan continuamente a estímulos exteriores generando otros estímulos. El tiempo de reacción está fijado por el entorno. Ejemplos: controles de máquinas, sistemas de procesado digital (ecualizadores, etc.), sistemas de radio, etc.

Los sistemas interactivos y reactivos a menudo son concurrentes: ejecutan una función mediante diversos procesos en paralelo que se comunican entre sí.

Los sistemas que estudiaremos durante el curso son un subgrupo de los reactivos.

❖ Sistemas reactivos

Los sistemas reactivos suelen responder, a grandes rasgos, al diagrama de bloques siguiente:

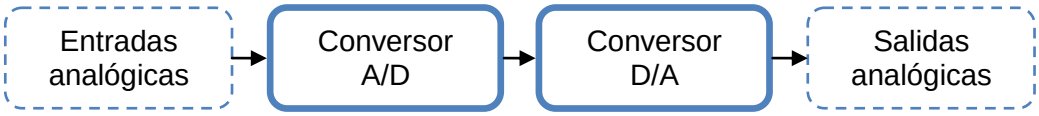


Figura 1. Diagrama de bloques de un sistema computerizado

Se trata de sistemas que reciben un flujo de datos de entrada y generan, como resultado, un flujo de datos de salida.

Los sistemas reactivos, al mismo tiempo, se pueden clasificar en los tipos siguientes:

- *Control-dominated*, o dominados por control. Generan salidas en respuesta a acontecimientos asíncronos de entrada. En este tipo de sistemas los datos disparan un algoritmo que no necesariamente requiere un procesado intensivo de la entrada. Por ejemplo, un controlador de ABS recibe como entradas el accionamiento del freno y la velocidad de la rueda. Ambas entradas controlan el funcionamiento del algoritmo, pero éstas no son procesadas de forma intensiva.

- *Data-flow*, o dominantes para datos. Generan continuamente valores de salida a partir de un procesado intensivo de los datos de entrada. Suelen ser síncronos. Por ejemplo, una radio recibe continuamente como entrada una señal de RF digitalizada que mezcla, desmodula, descodifica, etc. hasta generar una señal continua de datos o voz.

Durante el curso estudiaremos un tipo de sistemas encuadrados en los computerizados reactivos dominados por datos: los transceptores de radio digitales.

Los sistemas *data-flow* basan su funcionamiento en el procesado de unos datos de entrada. Para obtener las prestaciones deseadas no solo tendremos que hacer énfasis en el algoritmo sino que además tendrán mucha importancia las características de los datos sobre la velocidad, margen dinámica y resolución. Por ejemplo, en un sistema de radio típica es habitual toparse con requerimientos de margen dinámico del orden de 80 dB. Si no es así, la radio tendrá unas prestaciones inferiores a las mínimas (por ejemplo, podría no escucharse cuando nos encontremos lejos del trasmisor).

En oposición con este requerimiento, en los sistemas dominantes por control en general el factor importante en lo que respecta a la entrada es su existencia, no por sus características. Siguiendo con el ejemplo del

controlador ABS, el algoritmo se fijará sobre todo en la aparición de la señal de frenado; la cuantificación de la presión ejercida por el conductor, aun siendo también un dato relevante, lo será menos.

1.1.3. Metodologías de diseño

Los sistemas digitales computerizados, reactivos y dominados por datos tienen dos vertientes de diseño: la plataforma *hardware* por un lado y el *software* por otro. A continuación describiremos las dos metodologías de diseños que se suelen utilizar para desarrollar este tipo de aplicaciones.

❖ Metodología tradicional de diseño

Esta metodología es la que se ha utilizado siempre y aún está vigente para sistemas sencillos.

El proceso consiste en, partiendo de unas especificaciones iniciales, dar una primera solución *hardware* y *software*. A partir de aquí, se desarrollan por separado las dos vertientes hasta que se llega a un punto de integración. Seguidamente se testa el sistema y, si es necesario, se resuelven errores.

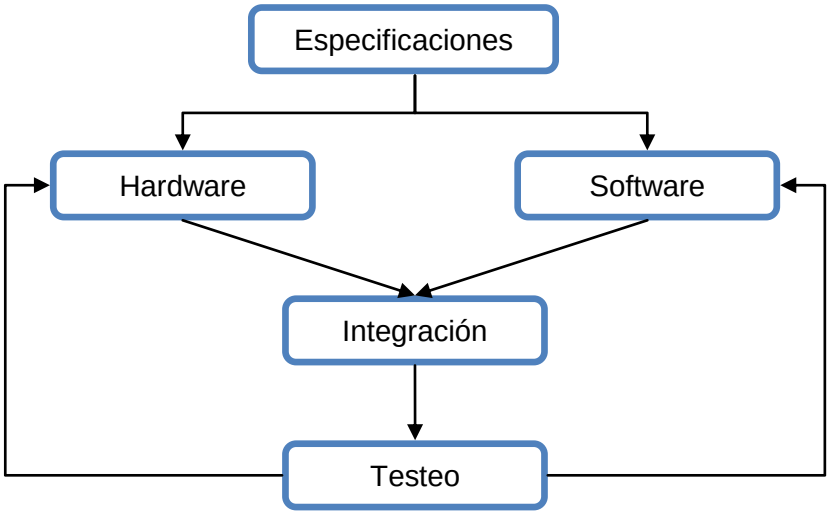


Figura 2. Proceso de diseño tradicional

Esta metodología presenta algunos inconvenientes:

- Dificultad para simular / verificar la corrección funcional del sistema como un todo ya que hace falta utilizar simuladores diferentes para cada plataforma: VHDL, C, ASM, etc.

- Los mecanismos de comunicación entre procesos no se pueden verificar hasta la etapa de integración, ya que las plataformas anteriores normalmente no pueden compartir información.

- Dificultad para simular / verificar la corrección temporal del sistema. ¿Se ejecutará el algoritmo en el tiempo especificado sobre el *hardware* escogido? Una vez más no se obtiene esta información hasta la etapa de integración

- Dificultad para optimizar las prestaciones del sistema. Una vez resuelto el *hardware*, únicamente se puede atacar la optimización intentando mejorar el código, ya que el rediseño del *hardware* suele ser costoso tanto en tiempo como en dinero.

❖ Metodología de codiseño

Una alternativa al proceso de diseño descrito en el punto anterior se denomina codiseño. Es una técnica que se está desarrollando actualmente, útil sobre todo para diseños complejos.

Consiste en, a partir de las especificaciones, diseñar la solución a nivel de diagrama de bloques funcional que integra *hardware* y *software*. Esta solución se especifica en un lenguaje único que permite simular / verificar el comportamiento global del sistema.

Una vez cumplido este paso, se escoge el *hardware* del sistema, que suele estar compuesto por un procesador y un dispositivo de lógica programable. Seguidamente se procede a la partición *hardware* / *software* del diseño sobre este dispositivo utilizando criterios temporales. Una vez decidido y simulada la partición, se mapea el código sobre los dispositivos.

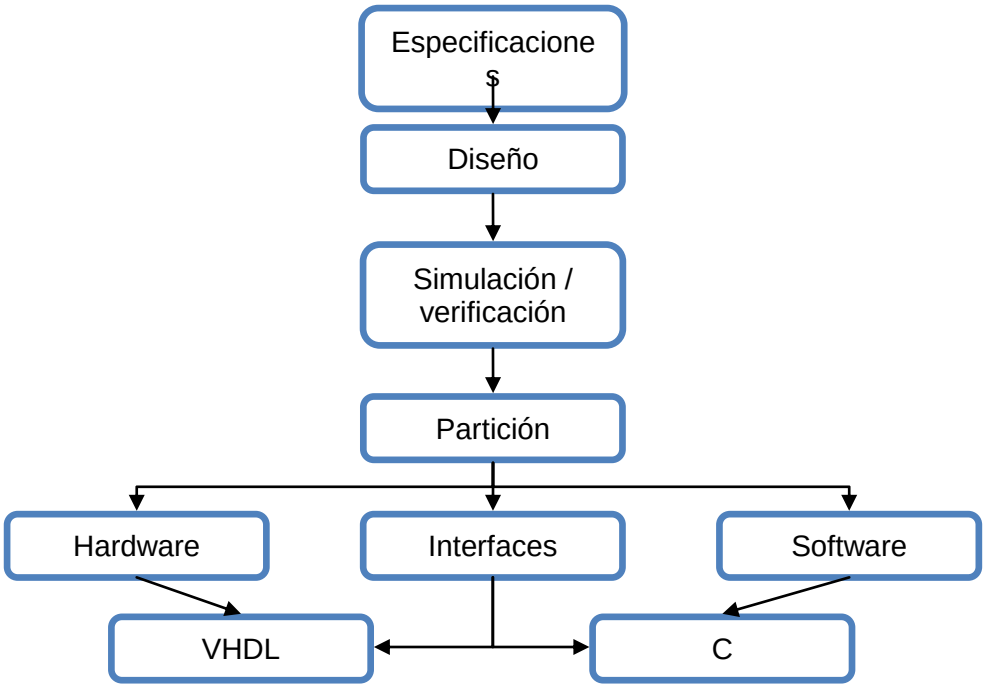


Figura 3. Codiseño

Esta metodología presenta la ventaja de retrasar la elección del *hardware* y el desarrollo de *software* hasta el último momento, cuando ya estemos seguros de la corrección del diseño.

Actualmente se está trabajando en diversos aspectos que reducirían considerablemente el tiempo de desarrollo, como ahora el mapeo y partición automática del *software*. Esto implicaría poder prescindir de la etapa de desarrollo en el lenguaje específico de cada plataforma.

1.2. Introducción a los sistemas de comunicación

En este documento expondremos algunos conceptos importantes relativos a los sistemas de comunicaciones, tanto en lo que se refiere a la terminología como a procedimientos, algoritmos y diagramas de bloques. El estudio, más que exhaustividad, busca ser intuitivo y que introduzca, lleve a cabo y facilite la comprensión del resto de la asignatura.

Nos referiremos básicamente a conceptos que después serán aplicados en el dominio digital. Además, hablaremos indistintamente de emisores o receptores, aunque a menudo pondremos más énfasis en los segundos ya que tienen una mayor dificultad de diseño.

1.2.1. Canales y bandas

En general los sistemas de comunicaciones (emisores y receptores) se diseñan para poder recibir cualquier canal de una banda de frecuencias. Por ejemplo, un receptor de FM ha de ser capaz de recibir cualquier canal de radio (de banda ancha aproximada de 200 Khz) de la banda FM *broadcast* (88 a 108 Mhz). Todos los canales que pertenecen a una banda determinada comparten características: modulación, niveles de potencia, etc.

Aunque en los sistemas analógicos podemos asociar un canal (un ancho de banda) a un único usuario, en los sistemas digitales esto no es del todo cierto. Por ejemplo, la banda GSM en el *downlink* (enlace entre estación base y terminal) está compuesta por 124 canales de 200 Khz de ancho de banda cada uno entre las frecuencias 935 y 960 Mhz. Cada canal lo comparten hasta 8 usuarios multiplexados en tiempo (TDMA). Otro ejemplo: para el sistema de tercera generación de telefonía móvil UMTS FDD, la banda del *uplink* contiene 12 canales de 5 Mhz cada uno entre las frecuencias 1920 y 1980 Mhz. Cada canal lo comparten un número variable de usuarios multiplexados en código (CDMA).

Los conceptos de banda y canal serán importantes ya que condicionarás las especificaciones de los componentes de los emisores y receptores: habitualmente los receptores tendrán que ser capaces de sintonizar cualquier canal de una determinada banda.

1.2.2. Dinámica de potencia

La sensibilidad máxima utilizable es una figura de mérito que corresponde a la mínima potencia de un canal que el receptor es capaz de desmodular con

una cualidad determinada (normalmente en los sistemas digitales medida en términos de BER). Asimismo, los receptores admiten en su entrada una potencia máxima, por encima de la cual el comportamiento de estos se degrada (a causa, a menudo, de distorsiones e intermodulaciones).

Un receptor se encontrará, por ejemplo, con señales muy pequeñas (cercanos a la sensibilidad máxima utilizable) en situaciones en que se encuentre muy lejos del emisor o el repetidor. Igualmente, se encontrará con señales muy grandes cuando esté muy cerca de estos.

La diferencia entre la máxima y la mínima potencia de un canal que el receptor es capaz de procesar con una cierta calidad recibe el nombre de dinámica de potencia o margen dinámico. El margen dinámico es uno de los parámetros más determinantes e influyentes en el diseño de sistemas de radio digital.

1.2.3. Modulación

La modulación es el proceso por el cual la información de una señal denominada modulador modifica alguno de los parámetros de otra señal denominada portador. La necesitad de aplicar una modulación para enviar información viene determinada por:

- Solo existe un medio de transmisión (el aire) que ha de ser compartido por diversos usuarios. Para ello se necesitará que las diferentes señales portadoras que se radien sean de frecuencias diferentes para que no interfieran los unos con los otros.

- Las señales de baja frecuencia (los moduladores suelen serlo) son difíciles de radiar. Es por esto que la información que contienen se introducen en otra señal de más alta frecuencia (el portador) y más fácil de ser trasmitidos al medio.

Como es bien sabido, existen tres tipos básicos de modulaciones: amplitud, frecuencia y fase. Algunas modulaciones digitales complejas combinan los tres métodos.

A partir de ahora nos convendrá dividir el proceso de modulación en dos subprocesos:

- La propiamente llamada modulación, consiste en modificar algunos parámetros del portador (amplitud, fase o frecuencia) según el modulador

- El traslado en frecuencia (mezcla), consistente en colocar a la señal portadora en la frecuencia a la cual se ha de radiar el canal (hacemos

hincapié en que la magnitud del traslado de frecuencia es independiente de la señal moduladora)

Esta subdivisión se aproxima mejor a la realidad en lo que respecta a los dispositivos que realizan cada tasca. Por ejemplo, en el dominio digital la modulación la llevará a cabo un procesador DSP mientras que para el traslado en frecuencia se suelen utilizar dispositivos específicos como un *Digital Down Converter*.

Cualquier señal portadora de una determinada frecuencia se puede expresar en función de sus componentes *I*, *Q*, proyecciones respectivamente sobre el *coseno* y el *sino* de ésta frecuencia. Estos componentes son ortogonales entre si. La manera que tenemos de expresar matemáticamente esta ortogonalidad es mediante números complejos. Así, cuando veamos un término del tipo $I + jQ$ hace referencia a dos señales moduladoras que modificarán respectivamente los términos *cosinos* y *sinos* de una misma frecuencia portadora, términos que se sumarán directamente antes de ser radiados al aire.

La representación I/Q es también muy útil en recepción, ya que permite separar las dos componentes ortogonales y analizarlas independientemente para extraer la señal moduladora. Estos componentes I/Q se suelen representar en el diagrama que lleva su nombre. Este diagrama es muy útil para analizar modulaciones de amplitud, fase o combinaciones de ambas.

1.2.4. Linealidad. Distorsión e intermodulación

Interpretado desde un punto de vista frecuencial, un sistema lineal es aquel que presenta a la salida los mismos componentes frecuenciales que en la entrada. Es decir, la salida y la entrada sólo se diferencian en las amplitudes y fases de las diversas frecuencias de las que está compuesta la señal.

En el diseño de emisores y receptores que intervienen tanto sistemas sobre papel lineales (amplificadores y filtros) como no lineales (mezcladotes). En la práctica, pero, todos los dispositivos tienen algún componente de no-linealidad. La diferencia está en el hecho de que en los mezcladores la no-linealidad es intencionada, mientras que en los amplificadores es indeseada. Los efectos frecuenciales de la no linealidad consisten en la aparición de distorsión e intermodulación:

- Distorsión es el fenómeno por el cual el sistema genera harmónicos de la señal de entrada. Así, si la entrada está compuesta por la frecuencia f_1, la salida puede contener componentes a $f_1, 2 \cdot f_1, 3 \cdot f_1, \ldots$

- Intermodulación es el fenómeno por el cual el sistema genera combinaciones de las componentes frecuenciales de entrada. Si ésta está compuesta por les frecuencias $f_1 i\, f_2$, según el orden de la intermodulación en la salida podemos encontrar $f_1 + f_2, f_1 - f_2, 2 \cdot f_1 + f_2, 2 \cdot f_1 - f_2, \ldots$

Cuando los sistemas que tenían que ser lineales presentan no-linealidad, las prestaciones de los transceptores empeoran. Especialmente degradante es la intermodulación de tercer orden ($2 \cdot f_1 - f_2, 2 \cdot f_2 - f_1$), ya que si f_1 y f_2 son las frecuencias centrales de dos canales adyacentes (por tanto, cercanos), los productos de intermodulación de tercer orden tienen un valor muy cercano al de los canales i, por lo tanto, se manifiestan como interferencia muy cercana a los canales.

1.2.5. Arquitectura de receptores analógicos

En este apartado describiremos la arquitectura, funcionamiento, ventajas e inconvenientes del receptor homodino y heterodino, paso previo al estudio de la configuración con el que nos basaremos durante todo el curso: el heterodino digital.

❖ Receptor homodino

El receptor homodino se caracteriza por la existencia de un único dispositivo mezclador. Su diagrama de bloques, muy simplificado, sería el siguiente:

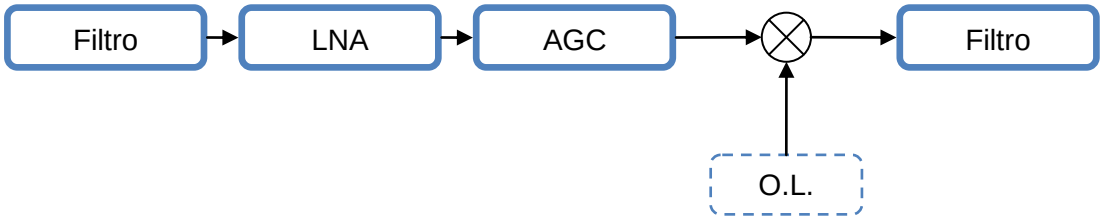


Figura 4. Diagrama de bloques de un receptor homodino

Los filtros aseguran la selectividad del receptor. Tienen como objetivo limitar la banda de ruido (recordemos que la potencia de ruido blanco viene dada por la expresión $K \cdot T \cdot B$, donde *B* es el ancho de banda) y rechazar señales de canales adyacentes.

El LNA (*Low Noise Amplifier*) es un amplificador de baja figura de ruido.
El AGC (*Automatic Gain Control*) es un amplificador de ganancia variable que asegura un nivel fijo en su salida independientemente del nivel de

entrada. Se caracteriza por un margen dinámico que determina la máxima variación de ganancia que puede aplicar.

El O.L. (oscilador local) está sintonizado a la frecuencia y fase del canal que nos interesa desmodular. El mezclador trasladará el canal de RF (*Radio Frequency*) a la banda base.

La principal ventaja de esta arquitectura es la simplicidad. Presenta, no obstante, diversos inconvenientes que en la práctica lo hacen inviable en la mayoría de los casos.

❖ Inconvenientes del receptor homodino

A continuación se enumeren algunos de los inconvenientes más importantes de la arquitectura de recepción homodina:

- La amplificación se lleva a cabo a una única frecuencia. Este hecho introduce riesgos de oscilaciones a causa de realimentaciones positivas entre amplificadores. Hay que tener en cuenta que la amplificación necesaria en un receptor es muy grande ya que los niveles de potencia de entrada pueden ser muy bajos.

- Dificultades para asegurar una buena selectividad. El objetivo del filtro de entrada es filtrar el canal o bien la banda si se trata de un receptor multicanal. Este filtrado presenta una gran dificultad de implementación cuando la frecuencia central es muy elevada. Por ejemplo, diseñar un filtro que a 900 Mhz seleccione un canal de 60 Khz de ancho de banda de manera que a 900.06 Mhz y 899.940 Mhz atenúe 60 dB requiere un factor de calidad de 10^7, no asumible en la mayoría de los casos.

- Las modificaciones necesarias para poder recibir modulaciones de fase son prácticamente irrealizables. En estos casos, hay que sintonizar dos mezcladores mediante dos O.L. de la misma frecuencia desfasados 90º:

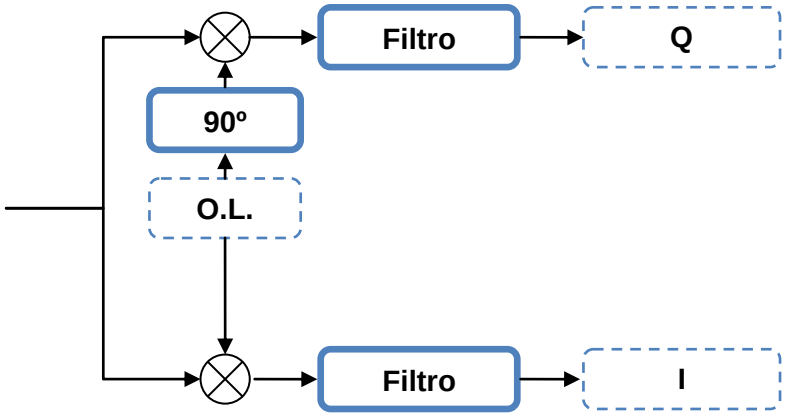


Figura 5. Mezclador en cuadratura

El problema surge cuando la frecuencia del canal a desmodular es elevada. En estos casos, el retrasador de 90º (un cuarto de periodo) ha de tener una precisión muy grande. Por ejemplo, un cuarto de periodo a 900 Mhz es 0.27 ns. Si la tolerancia del retrasador es significativa, en caso de modulaciones digitales el error en el diagrama de constelación I/Q provocará un BER elevado.

- Pérdida de sensibilidad causada por la distorsión de los amplificadores. Tal y como ya hemos comentado, los amplificadores sintonizados a RF han de tener una ganancia muy elevada. La ganancia elevada suele ir acompañada de no linealidades fuertes. Este factor, en conjunción con el limitado aislamiento del puerto de RF del mezclador, puede producir una degradación de las prestaciones del receptor en lo que se refiere a la mínima señal que éste puede recibir. Veámoslo:

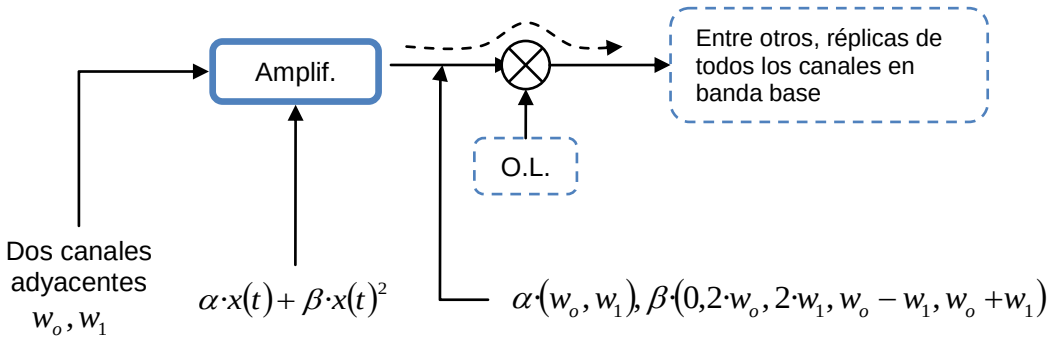


Figura 6. Efectos del amplificador y mezclador

Fijémonos en que la distorsión de segundo orden del amplificador genera réplicas de los dos canales en banda base (como si fuera un mezclador). Este efecto combinado con un limitado aislamiento del puerto de RF del mezclador genera interferencia de las réplicas sobre el canal deseado a la salida del mezclador.
Hemos de darnos cuenta los inconvenientes que hemos hecho patentes del receptor homodino vienen todos por causas de implementación. Sobre papel el receptor homodino es sencillo y perfecto. En la práctica, pero, en muchos casos es irrealizable.

❖ Receptor heterodino

El receptor heterodino nace para dar solución a los numerosos inconvenientes descorteados en el punto anterior. Su característica diferenciadora es la existencia de más de una etapa mezcladora. El traslado en frecuencia, por tanto, se lleva a cabo en diversas fases. Las frecuencias centrales de cada fase reciben el nombre de frecuencias intermedias.

Vemos a continuación el diagrama de bloques de un receptor heterodino de dos mezcladores:

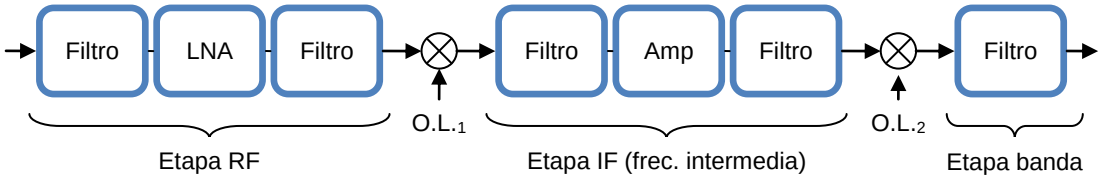


Figura 7. Diagrama de bloques de un receptor heterodino

❖ Etapa RF

El primer filtro tiene como objetivo limitar el ancho de banda de ruido. Es muy importante que no introduzca pérdidas en la banda de paso ya que en este punto el nivel de potencia de entrada es muy bajo.

El segundo filtro ha de asegurar un rechazo mínimo de la frecuencia imagen. La frecuencia imagen es aquella frecuencia simétrica de la del canal de entrada respecto del oscilador local: si f_{OL} es la frecuencia del O.L. y f_o es la frecuencia central del canal de interés, si suponemos $f_o > f_{OL}$ entonces la frecuencia intermedia es $f_{IF} = f_o - f_{OL}$ y la frecuencia imagen $f_{imatge} = f_{oL} - (f_o - f_{OL})$.

Si esta frecuencia imagen se encuentra en la entrada del mezclador, este la trasladará justamente a la frecuencia intermedia, producirá interferencia y degradará la sensibilidad del receptor. Es necesario, por lo tanto, introducir un filtro para limitar la amplitud de esta frecuencia imagen.

❖ Etapa IF

El tercer filtro elimina el producto de la mezcla que no interesa. Siguiendo con la notación anterior, el filtro tendría que eliminar los entornos de la frecuencia $f_o + f_{OL}$.

El cuarto filtro eliminará la posible distorsión del amplificador y la frecuencia imagen del siguiente proceso de mezcla (si no es el último).

El principal inconveniente del receptor heterodino es el problema de la frecuencia imagen. En comparación con el homodino, además, aumenta la complejidad.

En lo que se refiere a las ventajas, destacar que todos los inconvenientes redactados en el apartado anterior en lo que se refiere al receptor homodino se han solucionado:

- La amplificación está distribuida, como mínimo, a dos frecuencias. Por tanto, el riesgo de oscilaciones disminuye.

- El filtrado también se realiza a dos frecuencias. Además, en la etapa IF se pueden implementar filtros más selectos ya que están centrados en frecuencias más bajas.

- La desmodulación I/Q sólo es necesaria en la última etapa. En esta etapa la frecuencia es mucho más baja y, por lo tanto, la aleatoriedad del retrasador no es tan relevante.

- Los amplificadores no tendrán que ser de tanta ganancia. Por esto podemos enfatizar el conseguir una buena linealidad que no comporte los problemas descritos en el receptor homodino.

1.2.6. Receptor heterodino digital

En el punto anterior hemos visto cómo el receptor heterodino soluciona los problemas que presentaba el homodino. Si esto es realmente así, ¿es necesario un nuevo esquema?

En este apartado veremos las limitaciones del receptor heterodino analógico para introducir el que será el esquema que estudiaremos durante el curso: el receptor heterodino digital.

❖ Limitaciones del receptor heterodino analógico

El receptor heterodino visto hasta el momento presenta las limitaciones de cualquier sistema analógico; limitaciones que ya hemos comentado en apartados anteriores: sensibilidad al ruido, cambios con el tiempo y temperatura, dificultad para reprogramar y configurar su funcionamiento, etc. Si tenemos en cuenta la tendencia actual de los sistemas de comunicaciones, estos inconvenientes son especialmente graves.

Efectivamente, uno de los sectores que más impulso tiene actualmente es el de las comunicaciones móviles. Desde hace veinte años hasta la actualidad, diversos estándares se han ido sucediendo. La primera generación (AMPS, TACS) es analógica. La segunda generación ya migró a digital (GSM, IS-95, IS-54). La segunda generación y media (2.5G: GPRS, EDGE, HSCSD) y la tercera generación (UMTS, cdma2000) conviven en un mismo instantes temporal de manera que los terminales han de ser capaces de soportar todos estos estándares simultáneamente. La reconfigurabilidad de los terminales sólo es posible implementarla en el dominio digital, de manera que uno de los factores clave que explica el gran desarrollo que están disfrutando las comunicaciones móviles en la actualidad es la llegada de la tecnología digital en este ámbito.

Como bien sabemos, uno de los principales inconvenientes de los sistemas digitales recae en la dificultad para trabajar con frecuencias elevadas y grandes anchos de banda. Todo sistema digital está limitado en velocidad por los conversores A/D y D/A. Estos son, en efecto, el cuello de la botella de los transceptores de radio digital. No sólo por lo que respecta a la velocidad, sino también al número de bits ya que los receptores requieren un gran margen dinámico (sensibilidad muy baja).

Vistos los diferentes aspectos que influyen en la evolución del ámbito de las comunicaciones, podemos afirmar que la tendencia actual consiste en acercar la parte digital de un sistema de comunicaciones tanto en la antena como lo permita la tecnología. Ya hemos comentado que el cuello de botella es, sobretodo, el conversor A/D: los límites actuales están en frecuencias de muestreo alrededor de 100 MSPS y resolución de unos 16 bits. Con estos

parámetros, los transceptores de última generación pueden digitalizar la señal a partir de una frecuencia intermedia alrededor de 50 Mhz. Esto, como veremos durante el curso, permite una versatilidad alta a la hora de diseñar los sistemas de comunicaciones. Aún dependemos, no obstante, de una parte analógica que probablemente con el tiempo podemos ir reduciendo a la mínima expresión: filtros antialiasing y algún amplificador.

❖ Receptor heterodino digital

Antes de entrar a explicar el receptor heterodino digital introduciremos el impacto del significado que la palabra digital suele adoptar en el ámbito de los transceptores. A tal efecto, diferenciemos tres tipos de sistemas:

- Sistemas con interficie gráfica digital. Son transceptores analógicos que típicamente incluyen un microcontrolador que gestiona la interficie de usuario (teclado y *display*). Las prestaciones de estos sistemas son las mismas que la de los sistemas analógicos, ya que todo el proceso radio se implementa con esta última tecnología.

- Sistemas con procesado banda base digital. Este tipo de transceptores tienen un *front-end* de radio analógico que filtra, amplifica y traslada el canal de interés a la banda base (o a una frecuencia muy baja). Allí es digitalizado y procesado típicamente con un DSP. Estos sistemas se benefician de la tecnología digital en el sentido que permiten la implementación de algoritmos de codificación, protección de errores, ecualización y similares. La reconfigurabilidad, sin embargo, no es total ya que resulta complicado cambiar los parámetros de la cadena de recepción (p.e. el ancho de banda). Además están fuertemente sometidos a inconvenientes como tolerancias y derivadas de componente, etc.

  Las prestaciones de estos sistemas son mejores que la de los anteriores.

- Sistemas digitales. Este tipo de transceptores implementan en digital todos los bloques de los receptores que la tecnología permite. Actualmente este requisito se traduce en una conversión A/D, D/A alrededor de los 100 Mhz con una resolución de 16 bits. Incluyen filtrado y mezcla digital, factor que les confiriere una configurabilidad y prestaciones máximas.

El receptor heterodino digital pertenece al último grupo de los nombrados.

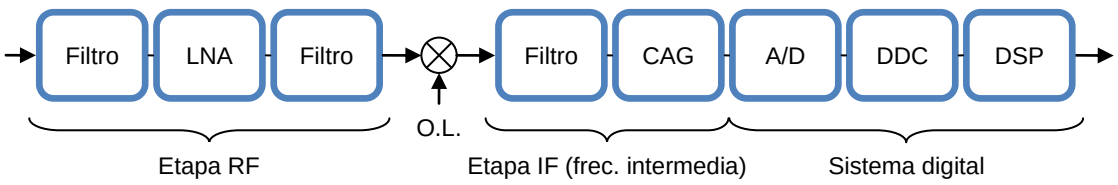


Figura 8. Diagrama de bloques de un receptor heterodino digital

La etapa de RF es idéntica a la del heterodino analógico.

La etapa de IF trabaja a una frecuencia analógica máxima inferior a 50 Mhz. Consiste en un filtro antialiasing y un control automático de ganancia que ajusta la potencia de salida al fondo de escala del conversor A/D.

El conversor A/D digitaliza la señal a una frecuencia de unos 100 MSPS (*Mega Sanchos per Second*) y 16 bits. Es un conversor con una linealidad muy buena con tal de asegurar una distorsión / intermodulación muy bajas.
El DDC (*Digital Down Converter*) es un dispositivo digital programable que filtra, mezcla y diezma la señal. Este dispositivo le confiere al receptor las prestaciones de configurabilidad de los parámetros radio: ancho de banda, frecuencia central, etc.

El DSP (*Digital Signal Procesor*) es un procesador específico que se encargará del procesado banda base: desmodulación, corrección de errores, sincronización de símbolo, etc.

Cabe decir que la arquitectura de la figura no es la única posible. Existe todo un abanico de dispositivos específicos que pueden ser integrados en la cadena digital de recepción (y en su caso de emisión): lógica programable (CPLD / FPGA), DUC (*Digital Up Converter*), filtros digitales, *costas loop*, etc
Durante el curso estudiaremos el comportamiento de todos estos dispositivos, arquitecturas y configuraciones alternativas, etc. haciendo especial énfasis en las relaciones entre los componentes para cumplir con unas determinadas prestaciones fijadas por unas especificaciones.

1.3. Los filtros analógicos

Los filtros analógicos son elementos clave en los sistemas digitales. Sus características determinan a menudo las prestaciones globales del sistema. Por esto es muy importante conocer bien la caracterización tanto a nivel temporal como frecuencial, tanto en lo que se refiere a la fase como a amplitud.
En este documento revisaremos en primer lugar la parametrización de los filtros analógicos y a continuación comentaremos las diferentes funciones y características que han de tener los diferentes filtros analógicos que aparecen en el diagrama de bloques de un heterodino digital.

1.3.1. Caracterización de filtros analógicos

Veamos seguidamente los parámetros más importantes que definen un filtro, tanto a nivel temporal como frecuencial. Pérdida de generalidad.

❖ Caracterización temporal

La manera más habitual de caracterizar un filtro en el dominio temporal es a través de su respuesta en el escalón unitario.

Figura 9. Respuesta de un filtro al escalón unitario

Los parámetros que caracterizan la respuesta al escalón de un filtro son los siguientes:

- Tiempo de subida: tiempo que tarda la señal de salida en completar el 90% del valor final.

- *Sobreamortigüamiento*: diferencia máxima, por exceso, entre la salida y la entrada.

- *Ringing*: porcentaje de oscilación de la salida alrededor de su valor final.

En la figura siguiente vemos estos parámetros representados sobre la respuesta a un escalón unitario.

Figura 10. Caracterización temporal de un filtro

❖ Caracterización frecuencial

La caracterización de un filtro en el dominio frecuencial se suele realizar mediante dos funciones, que son su respuesta frecuencial de amplitud y su respuesta frecuencial de fase.

La respuesta frecuencial de amplitud de un filtro pasa-bajo tiene la forma típica que se presenta en la figura 11. Los parámetros que la caracterizan son los siguientes:

Frecuencia de corte: frecuencia en la que la atenuación es de –3dB respecto al máximo de la banda de paso. Los filtros pasa banda tienen dos frecuencias de corte.

- Banda de paso: banda de frecuencias que van desde los 0 Hz hasta la frecuencia de corte. En los filtros paso banda se define como el margen de frecuencias comprendidas entre les dos frecuencias de corte.

- Rizada de la banda de paso: diferencia entre las atenuaciones máxima y mínima de la banda de paso; se mide en dB. Idealmente tendría que ser cero.

- Frecuencia de atenuación. Se define como aquella frecuencia en la cual el filtro presenta una atenuación determinada, que suele estar relacionada con algún parámetro que depende del resto del diseño (normalmente el margen dinámico). Podemos definir, entonces, la frecuencia de atenuación a 60 dB, o a 70 dB, o a cualquier otra atenuación que sea interesante para un diseño determinado.

- Banda de transición: margen de frecuencias comprendidas entre la frecuencia de corte y la de atenuación.

- Pendiente de la banda de transición: derivada de la atenuación respecto a la frecuencia; se mide en dB por década o por octava.

- Banda de atenuación: en un filtro de paso bajo, margen de frecuencias por encima de la frecuencia de atenuación.

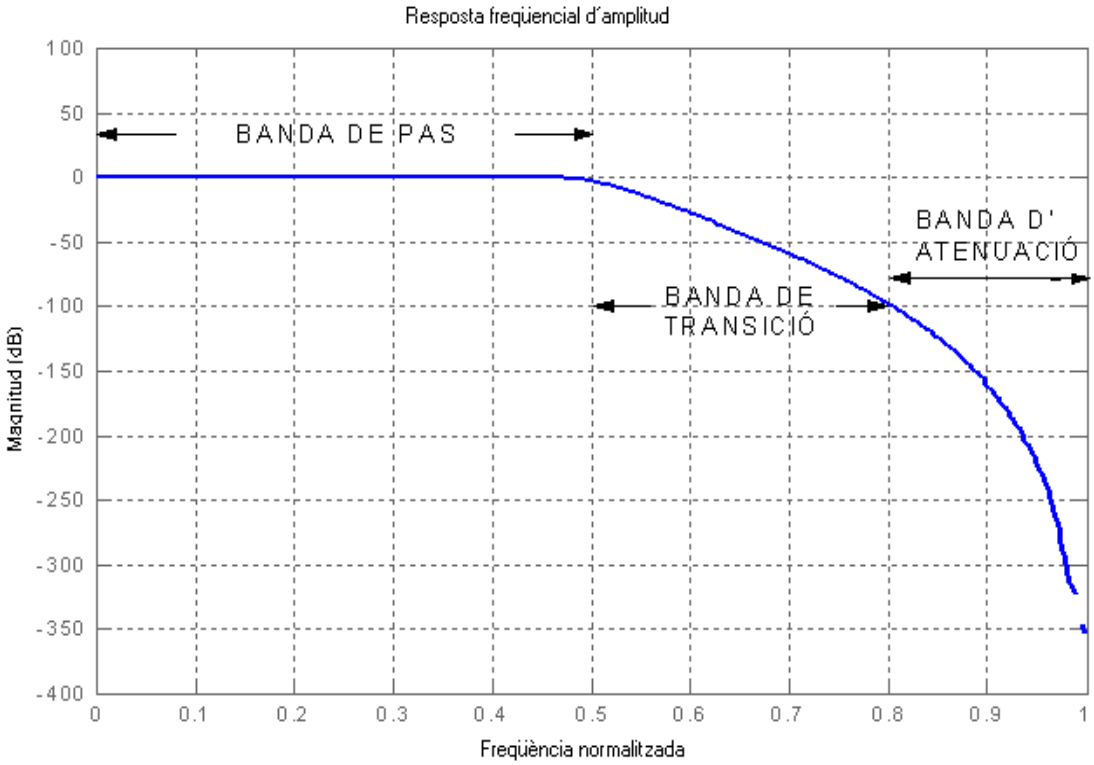


Figura 11. Respuesta frecuencial de amplitud de un filtro pasa-bajos

El parámetro más destacable que describe la respuesta de fase de un filtro es el retraso de grupo:

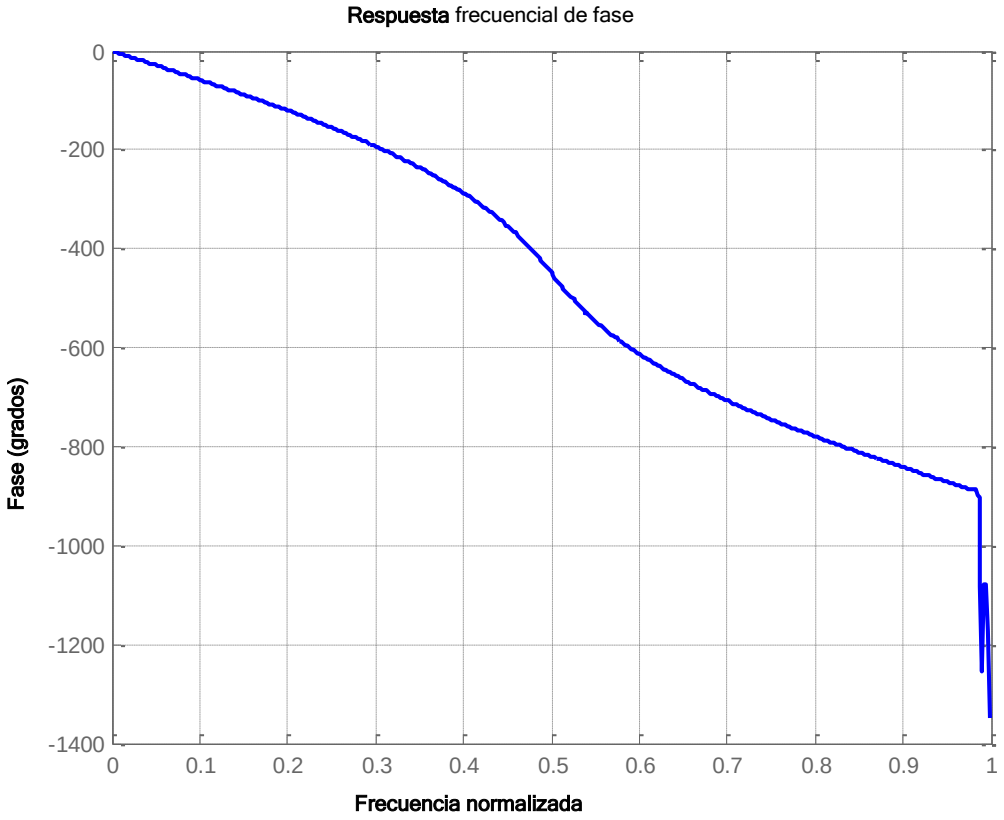


Figura 12. Respuesta en fase de un filtro de paso bajo

- Se calcula como la derivada de la respuesta de fase respecto a la frecuencia:

$$\tau(\omega) = -\frac{d\Phi(\omega)}{d\omega} \left[\frac{rad}{rad/sg} = sg \right]$$

- El retraso de grupo evaluado en una frecuencia determinada nos indica el retraso temporal que aquella frecuencia sufrirá al pasar por el filtro.

- Lo ideal es que el filtro retrase todas las frecuencias por igual, es decir, que el retraso de grupo sea constante; esto implicará que la respuesta de fase sea lineal:

$$\Phi(\omega) = k \cdot \omega \Rightarrow \tau(\omega) = -\frac{d\Phi(\omega)}{d\omega} = -k \text{ sg.}$$

Veremos que de cara al resto del curso, los parámetros que más nos interesarán de los filtros serán la frecuencia de corte (que en recepción

relacionaremos con el ancho de banda de la banda de canales de recepción), la frecuencia de atenuación (que relacionaremos con la frecuencia de muestreo de los conversores A/D) y la atenuación a la banda de atenuación (que relacionaremos con el margen dinámico del sistema).

1.3.2. Los filtros en el heterodino digital

Recordemos el esquema del receptor heterodino digital con tal de comentar las funciones y tecnologías de los filtros que aparecen.

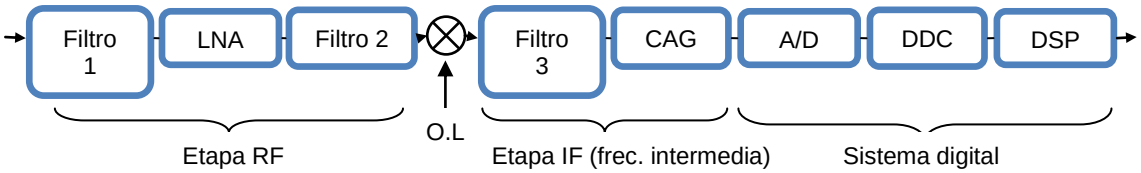


Figura 13. Receptor heterodino digital

El *Filtro 1* es el filtro de entrada. Sus funciones principales son:

- Limitar el ruido. Recordemos que la potencia de ruido blanco viene dada para $K \cdot T \cdot B$, donde B es el ancho de banda. Reduciendo B conseguiremos limitar la potencia de ruido y aumentar la sensibilidad y selectividad. El mínimo valor de B para un sistema multicanal será, en general, el de la banda.

- Eliminar la señal fuera de la banda de interés. Antes de amplificar la señal conviene atenuar al máximo todas las señales que no nos interesen. De esta manera se evita posibles saturaciones y distorsiones de los amplificadores.

Un requerimiento muy importante por el *Filtro 1* es que introduzca una mínima atenuación en la banda de paso, ya que en este punto del receptor la señal es muy débil y cualquier atenuación repercute en un empeoramiento de la sensibilidad.

Este filtro suele ser de bajo orden y se implementa habitualmente con componentes discretos (condensadores y bobinas).

El *Filtro 2* tiene las funciones siguientes:

- Eliminar los posibles harmónicos causados por la distorsión introducida por el LNA (*Low Noise Amplifier*).

- Asegurar una atenuación mínima de la frecuencia imagen del oscilador analógico que figura a continuación. Esta atenuación determina en gran medida la sensibilidad del receptor.

El *Filtro 2* se puede implementar utilizando la tecnología SAW (*Surface Acoustic Wave*). La atenuación en la banda de paso que presenta esta tecnología no es excesivamente grave ya que la señal en este punto ya se ha amplificado, al tiempo que el plano de la banda de paso y la abrupta banda de transición le hacen muy útil para este propósito.

El *Filtro 3* tiene como objetivos:

- Eliminar la frecuencia doble producida por el mezclador.

- Limitar el ancho de banda de la señal para que el control automático o de ganancia (CAG) trabaje correctamente.

La tecnología de filtros de cristal puede resultar adecuada en algunos casos (receptor monocanal). En la mayoría de las ocasiones, no obstante, conviene utilizar filtros que dejen pasar toda la banda para seleccionar el canal en concreto digitalmente (utilizando el mezclador digital DDC).

Algunas veces se pone un filtro entre el CAG y el A/D de alta velocidad para asegurar que se elimina el *aliasing*.

En caso de que el receptor (o emisor) necesite interaccionar en banda base con señales analógicas (p.e. voz), hay que incluir conversores A/D y D/A de baja velocidad con sus respectivos filtros *antialiasing* y reconstructores. Las dos opciones más frecuentemente usadas para resolver este aspecto son:

- Circuitos integrados que incluyen conversores de baja velocidad (hasta 50 KSPS), filtros y amplificadores. Reciben el nombre de AIC (*Analog Interface Circuit*) o CoDec (*Coder / Decoder*). Por ejemplo, el AD1847 de *Analog Devices* se utiliza en algunos módulos de evaluación de DSP.

- Circuitos integrados de filtros conmutados. Hay diferentes órdenes y diferentes técnicas (Txebixev, Bessel, etc.). Por ejemplo, LTC1063 de *Linear Technology*.

2. Conversores A/D y D/A

En este apartado estudiaremos principalmente la caracterización, prestaciones, técnicas y uso del conversor A/D en sistemas de radio digital. Lo haremos desde dos puntos de vista. En primer lugar discutiremos el conversor A/D desde un punto de vista teórico: fundamentos de funcionamiento (muestreo y cuantificación) y técnicas avanzadas de uso (*dithering*, *oversampling* y *undersampling*).

Seguidamente veremos las restricciones que la tecnología impone sobre las prestaciones de los conversores: errores de linealidad y restricciones de ancho de banda. Se completa el capítulo con un breve estudio sobre la función de interpolación realizada por los conversores D/A.

2.1. El conversor A/D ideal

Un conversor A/D es un componente que representa todos los valores analógicos dentro de un intervalo por un número limitado de códigos digitales, cada un de los cuales representa de manera exclusiva una fracción de la entrada.

Los conversores de *n* bits que utilizan un código binario estándar (que, por otro lado, son los más frecuentes), disponen de 2^n códigos digitales diferentes.
La función de transferencia caracteriza su comportamiento entrada – salida:

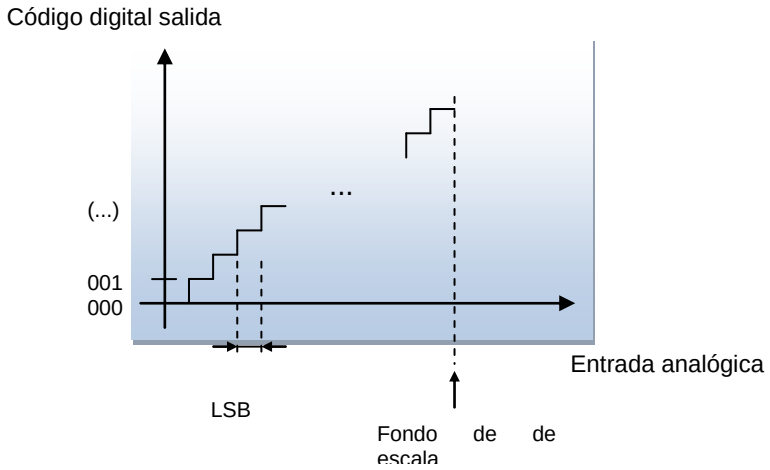


Figura 14. Función de transferencia de un conversor A/D

El fondo de escala (FSR: *Full-Scale Range*) es el máximo valor de la entrada analógica que el conversor A/D admite.

El LSB (*Least Significant Bit*, también abreviado comoΔ) define el ancho de la fracción de la entrada representada por un mismo código.

La ecuación siguiente relaciona los tres parámetros del conversor nombrados hasta ahora:

$$LSB = \frac{FSR}{2^n - 1}$$

Ecuación 1. LSB

A nivel de diagrama de bloques, nos interesará distinguir las funciones especificadas en la figura 2. El conversor, por tanto, simplifica el mundo analógico discretizando el tiempo y la amplitud. Esta doble simplificación se fundamenta en el *Teorema del Muestreo* y el *Modelo de Cuantificación*. A continuación revisaremos estos dos conceptos.

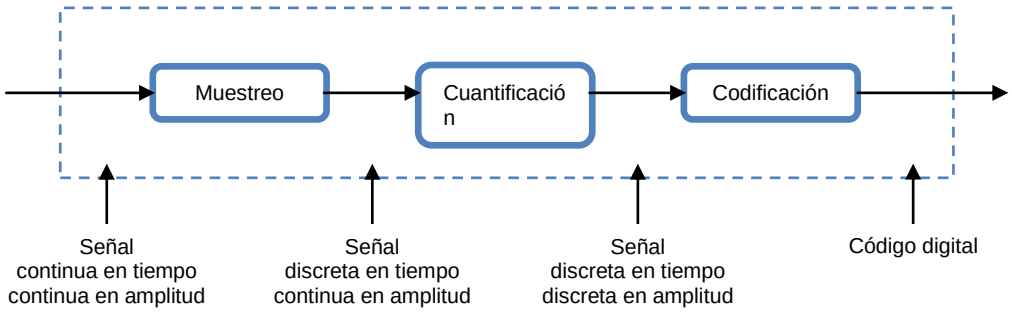


Figura 15. Diagrama funcional de un conversor A/D

2.1.1. Teorema del muestreo

El teorema del muestreo afirma que si la frecuencia más alta de una señal analógica es *B* y lo muestrearemos a una frecuencia $f_m \geq 2 \cdot B$, entonces la señal podrá ser recuperada exactamente a partir de sus muestras utilizando la función de interpolación / filtro:

$$g(t) = \frac{\sin(2 \cdot \pi \cdot B \cdot t)}{2 \cdot \pi \cdot B \cdot t} \leftrightarrow g(f) = \frac{1}{2 \cdot B} \Pi\left(\frac{f}{2 \cdot B}\right)$$

Ecuación 2. Interpolador ideal

El teorema impone, pues, dos restricciones: una sobre el proceso de conversión A/D y otra sobre el proceso de conversión D/A

- Sobre el A/D: una frecuencia mínima de muestreo, según la frecuencia máxima de la señal analógica de entrada. Con tal de asegurar

esta condición se suele utilizar un filtro analógico en la entrada del A/D, denominado *antialiasing*.

- Sobre el D/A: una función de interpolación concreta (Ecuación 2):

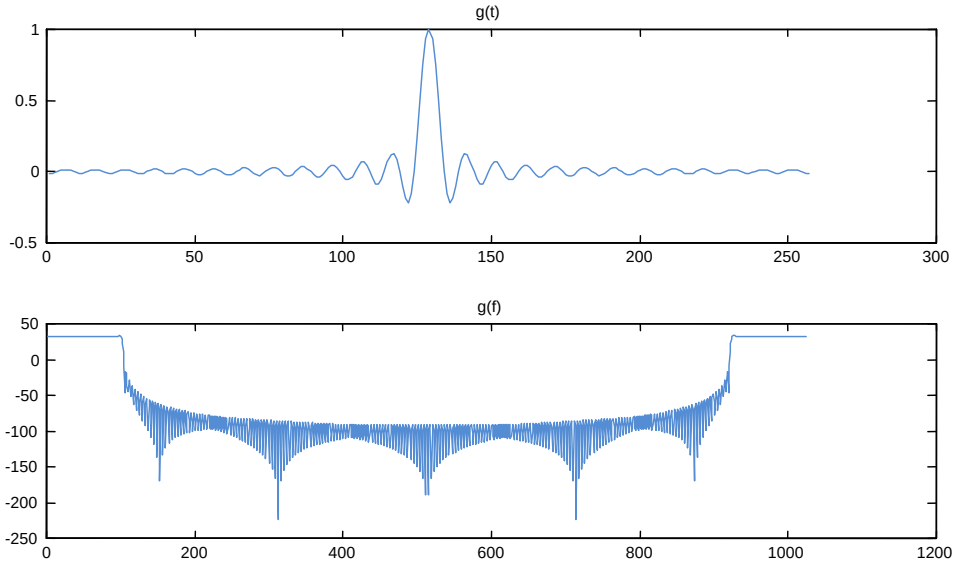


Figura 16. Función de interpolación ideal de un conversor D/A

Tanto si lo miramos temporalmente (respuesta impulsional infinita), como frecuencialmente (respuesta abrupta), esta función es irrealizable. En la práctica nos tendremos que conformar con una aproximación a base de dos filtros a la salida del D/A: un ecualizador y un paso bajo. Lo veremos con más detalle en apartados posteriores. Así pues, aunque teóricamente el proceso de muestreo no supone distorsión, en la práctica no seremos capaces de cumplir estrictamente los requisitos que éste impone. No obstante, esto no será, un grave inconveniente si aplicamos un conjunto de criterios que veremos más adelante.

❖ El filtro antialiasing

El teorema del muestreo impone una condición sobre la frecuencia de muestreo (f_m): se precisa que $f_m \geq 2 \cdot B$. Para asegurarla, se suele utilizar una configuración formada por un filtro de paso bajo (en radio normalmente es paso banda) denominado *antialiasing* antes del conversor A/D:

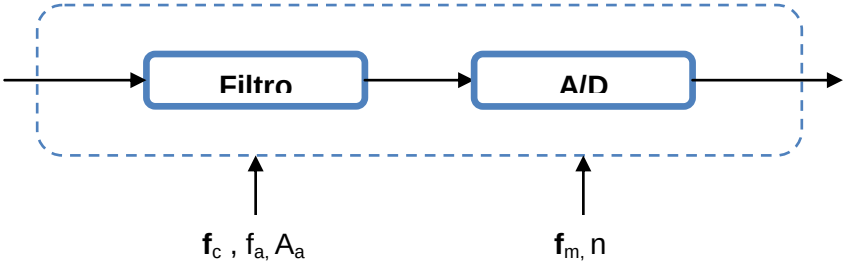


Figura 17. Filtro antialiasing y conversor A/D

Las características relevantes que a continuación relacionaremos son:

En lo que se refiere al filtro:

- Frecuencia de corte f_c. Indica la frecuencia de caída 3 dBs. Separa la banda de paso y la banda de transición. Si el filtro es paso banda tendremos que fijar dos frecuencias de corte (la inferior y la superior).

- Frecuencia de atenuación f_a. Marca el inicio de la banda de atenuación. La f_a es una frecuencia que depende del diseño en el que estemos involucrados: indica la frecuencia a la cual el filtro proporciona la atenuación deseada. Igualmente, para los filtros paso banda habrá dos frecuencias de atenuación, pero la más restrictiva (excepto si aplicamos submuestreo) será la superior.

- Atenuación mínima a la banda de atenuación (A_a).

En lo que se refiere al conversor A/D:

- Frecuencia de muestreo f_m.

- Número de bits n.

El filtro tiene que dejar pasar todo el ancho de banda de la señal sin atenuarlo. Para tal efecto fijaremos $f_c = f_{\max}$, donde $f_{\max}$ coincide con el ancho de banda (B) para señales banda base. Si el filtro es paso banda, las dos frecuencias de corte las fijaremos para que la banda de paso coincida con el ancho de banda de todos los canales (banda) de interés.

Además, el filtro tiene que rechazar todas las componentes frecuenciales por encima de $f_m/2$ para evitar que exista *aliasing* en la salida del A/D. En otras palabras, diseñaremos de manera que $f_a = f_m/2$ y exigiremos una atenuación suficiente a esta frecuencia para conseguir el objetivo que nos hemos marcado. ¿Cuál es esta atenuación suficiente? La respuesta está en el conversor A/D. Será necesario que el filtro atenúe cualquier componente frecuencial situada por encima de f_a un número de dBs con el que el A/D, en el peor de los casos, no se de cuenta de que existe: en otras palabras, por debajo de 1 LSB de variación.
El peor caso que se puede dar es una señal que no nos interesa a fondo de escala situado justo en la frecuencia de atenuación (f_a). Entonces la atenuación que deberá aplicar el filtro es:

$$A_a = \frac{FSR}{LSB} = \frac{FSR}{FSR/2^n - 1} \cong 2^n ; Atenuació\ (dB) \cong 6 \cdot n$$

Ecuación 3. Atenuación de un filtro antialiasing

La figura siguiente ilustra este concepto:

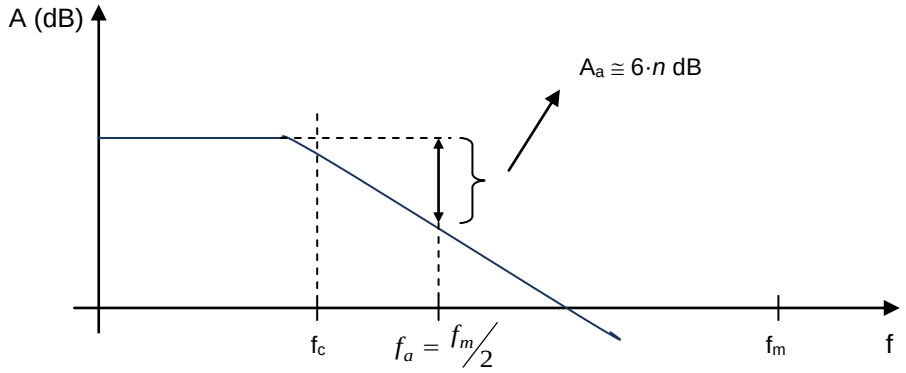


Figura 18. Características de un filtro antialiasing

Para fijar la frecuencia de atenuación hay otro criterio menos restrictivo. Fijaos en que si $f_a = f_m/2$ evitamos cualquier frecuencia que pueda generar *aliasing*. Ahora bien, podemos admitir *aliasing* por encima de f_c, ya que éste no caerá sobre la banda de interés sino sobre la de transición; después del A/D podremos filtrarlo digitalmente. En otras palabras, el *aliasing* que tenemos que eliminar sin falta es aquel que cae sobre la banda de interés, ya que es lo que no podremos filtrar después en digital. Este nuevo criterio (que usaremos en otras ocasiones durante el curso) lleva a fijar una frecuencia de muestreo $\frac{f_m}{2} = \frac{f_c + f_a}{2}$, donde f_c coincide con la frecuencia superior de la banda y f_a es la frecuencia a la cual el filtro atenúa A_a. Ahora la frecuencia de muestreo de el A/D será menor (o bien la pendiente del filtro será menos abrupta), pero por contra tendremos que poner un filtro digital para eliminar el posible *aliasing* por encima de $\frac{f_c}{f_m} = \frac{f_c}{f_c + f_a}$.

2.1.2. Error de cuantificación

La discretización de la amplitud que se lleva a cabo en el conversor A/D provoca un error denominado error de cuantificación. A modo de ejemplo en la figura 19 podemos observar las tres señales que intervienen en el proceso:

- La entrada al A/D, supuesta sinusoidal en un fondo de escala de 2 voltios

- La salida del A/D, cuantificado con 4 bits ($LSB = 2/15$)

- El error de cuantificación, requiere un valor máximo de $LSB/2$ porque se ha supuesto un conversor de redondeo.

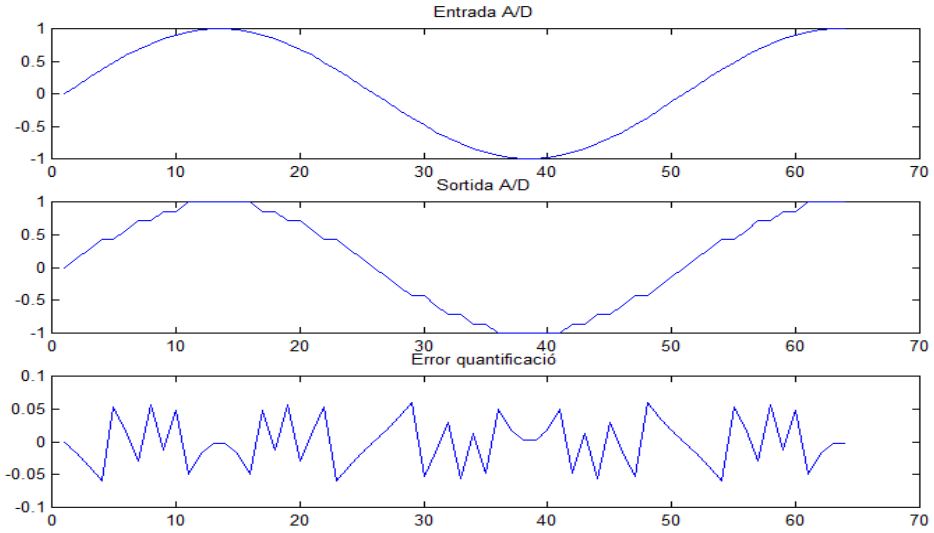


Figura 19. Error de cuantificación

El error de cuantificación es insalvable, y no hay ninguna teoría que nos permita cometerlo impunemente. En lugar de esto, nos conformaremos con conocer y limitar sus efectos.

❖ Modelo del error de cuantificación

A continuación expondremos un modelo que permite tratar el error de cuantificación como un ruido bajo ciertas condiciones. Este tratamiento nos permitirá encontrar una relación S/QN (leído *Signal to Quantization Noise*) que será el nuestro instrumento de trabajar.

Este modelo postula que si la señal en la entrada de un conversor A/D tiene una amplitud suficientemente grande como para ocupar, como mínimo, los niveles de cuantificación correspondientes a 4 ó 5 bits (suposición de "gran señal" respecto al LSB) entonces el error de cuantificación puede considerarse uniformemente distribuido entre $\left[-\frac{LSB}{2},\frac{LSB}{2}\right]$. En otras palabras, la probabilidad de cualquier valor del error dentro del intervalo es la misma.

Dado que el intervalo de probabilidad es simétrico, la media del error será cero

($\mu = 0$).

El cálculo de la variación nos dará la potencia del error:

$$QN = \int_{-\frac{\Delta}{2}}^{\frac{\Delta}{2}} (e-\mu)^2 \cdot p(e) \cdot de = \int_{-\frac{\Delta}{2}}^{\frac{\Delta}{2}} e^2 \cdot \frac{1}{\Delta} \cdot de = \frac{1}{\Delta} \cdot \int_{-\frac{\Delta}{2}}^{\frac{\Delta}{2}} e^2 \cdot de = \frac{1}{\Delta} \left[\frac{e^3}{3} \right]_{-\frac{\Delta}{2}}^{\frac{\Delta}{2}} = \frac{\Delta^2}{12}$$

Ecuación 4. Ruido de cuantificación

Remarquemos los puntos siguientes:

- Dado que el error de cuantificación lo desconocemos, intentemos modelarlo como un ruido. Este ruido quedará caracterizado con una media y una variación (potencia). Hay que tener claro, no obstante, que dado un caso concreto el error de cuantificación es determinante. Es el desconocimiento de la señal de entrada que hace que lo modelemos, bajo condiciones de "gran señal" como aleatorio.

- La potencia del error de cuantificación depende de dos factores: del fondo de escala y del número de bits:

$$QN = \frac{FSR^2}{12 \cdot \left(2^n - 1\right)^2}$$

Ecuación 5. Ruido de cuantificación

Con el fin de minimizar este error deberemos disminuir el fondo de escala (normalmente no será posible ya que el fabricante acostumbra a recomendar este parámetro), o bien aumentar el número de bits.

Para encontrar una relación S/QN será necesario a continuación encontrar la potencia de la señal de entrada. Supondremos una señal sinusoidal:

$$S = \frac{1}{T} \int_0^T \left(A \cdot \sin(w \cdot t)\right)^2 \cdot dt = \frac{A^2}{2}$$

Ecuación 6. Potencia de una señal sinusoidal

Ya estamos en disposición de encontrar la relación buscada:

$$S/QN = 10\cdot\log\left(\frac{A^2/2}{\Delta^2/12}\right) = 10\cdot\log\left(6\cdot\frac{A^2}{\Delta^2}\right)$$

Ecuación 7. Relación SQNR

Para poder simplificar la expresión anterior, supondremos la señal de entrada a fondo de escala: $2\cdot A = FSR$. Es importante darse cuenta de que ahora estamos suponiendo el mejor caso posible en lo que respecta a SQNR, ya que consideramos que la entrada tiene la máxima potencia.

Lo hacemos así en beneficio de obtener una expresión más simple:

$$S/QN = 10\cdot\log\left(6\cdot\frac{A^2}{\Delta^2}\right) = 10\cdot\log\left(6\cdot\frac{\left(\frac{FSR}{2}\right)^2}{\left(\frac{FSR}{2^n-1}\right)^2}\right) = 10\cdot\log\left(\frac{3}{2}\cdot\left(2^{2\cdot n}-1\right)\right) \cong 10\cdot\log\left(\frac{3}{2}\cdot 2^{2\cdot n}\right) = 1.76 + 6.02\cdot n$$

Ecuación 8. Relación SQNR

Hemos caracterizado, pues, el conversor A/D para "gran señal" como un sistema que discretiza la amplitud de la señal de entrada y que, además, introduce un ruido:

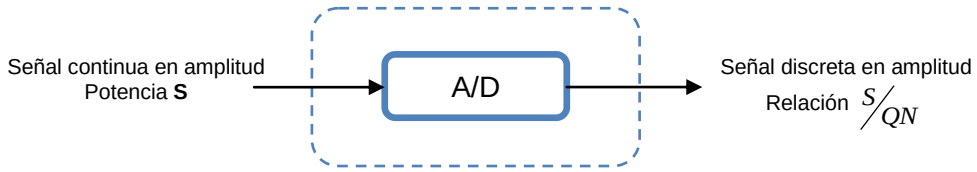


Figura 20. Conversor A/D y error de cuantificación

Es importante remarcar que la S/QN mejora 6 dB por cada bit adicional del sistema.

Fijaos en que el desarrollo anterior implica adoptar un modelo de conversor A/D lineal (respecto a la cuantificación), ya que la no linealidad (error de cuantificación) se modela como un ruido. Efectivamente, si en la entrada del A/D hay una señal cualquiera, en su salida aparece esta misma señal más un ruido (peri no hay distorsiones de la señal tal y como sería de esperar en caso de no linealidades).

Conviene remarcar que el modelo sólo se comporta bien para "gran señal". Veremos más adelante que para "pequeña señal" deberemos corregir el modelo.

A menudo se utiliza la relación $S/_{QN}$ como el margen dinámico del conversor A/D. Entendemos por margen dinámico la máxima diferencia de potencia entre dos señales simultaneas en la entrada del conversor que el A/D es capaz de captar. Efectivamente, si intentamos convertir dos señales una de las cuales está a fondo de escala del conversor y la otra tiene una potencia inferior a éste en un número de dBs igual al valor del $S/_{QN}$, en la salida del conversor la *QN* tendrá una potencia igual a la de la señal menor.

Hay que remarcar, no obstante, que si observamos, por ejemplo, el espectro de una señal sinusoidal cuantificada, la diferencia entre la potencia del pico frecuencial y el ruido de cuantificación será mucho más grande que la relación $S/_{QN}$. Esto se debe a que toda la potencia de la señal sinusoidal está concentrada en una frecuencia, mientras que la del ruido de cuantificación está uniformemente distribuida. Por tanto, la potencia para Herz del ruido de cuantificación (densidad espectral de potencia) es muy inferior a la potencia global.

Esta expresión del margen dinámico tiene especial interés en un análisis temporal ya que en este caso toda la potencia se manifiesta en cada instante de tiempo.

En general, en la entrada del conversor A/D no encontraremos una señal pura, sino que éste estará caracterizado por una relación $S/_{N}$. En este caso, en la salida del conversor este ruido se sumará al de cuantificación:

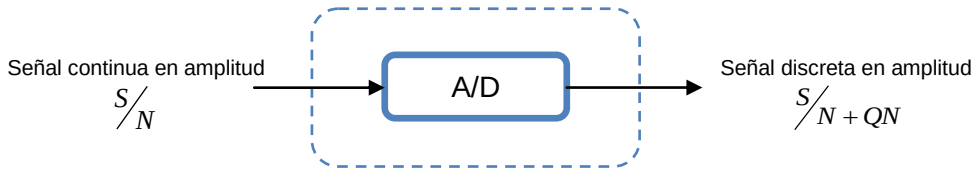


Figura 21. Conversor A/D, ruido y ruido de cuantificación

2.1.3. Dithering

El modelo anterior sólo es válido para "gran señal"; es decir, cuando la entrada ocupa como mínimo 4 ó 5 bits. ¿Qué pasa cuando no se da este caso? Para mostrarlo podemos realizar el experimento siguiente.
Utilicemos un conversor A/D para muestrear una señal del tipo $s(t)=\frac{1}{2}+\frac{A}{2}\cdot sin\left(\frac{1}{2}+2\cdot\pi\cdot f_0\cdot t\right)$ a una frecuencia de muestreo $f_m=12\cdot f_o$. Ajustemos la amplitud de la señal de entrada de manera que en un primer caso este ocupe únicamente 2 bits y en un segundo caso ocupe 8. Observemos la

señal de salida del A/D en formato temporal y frecuencial (FFT de 2048 muestras).
El resultado es el siguiente:

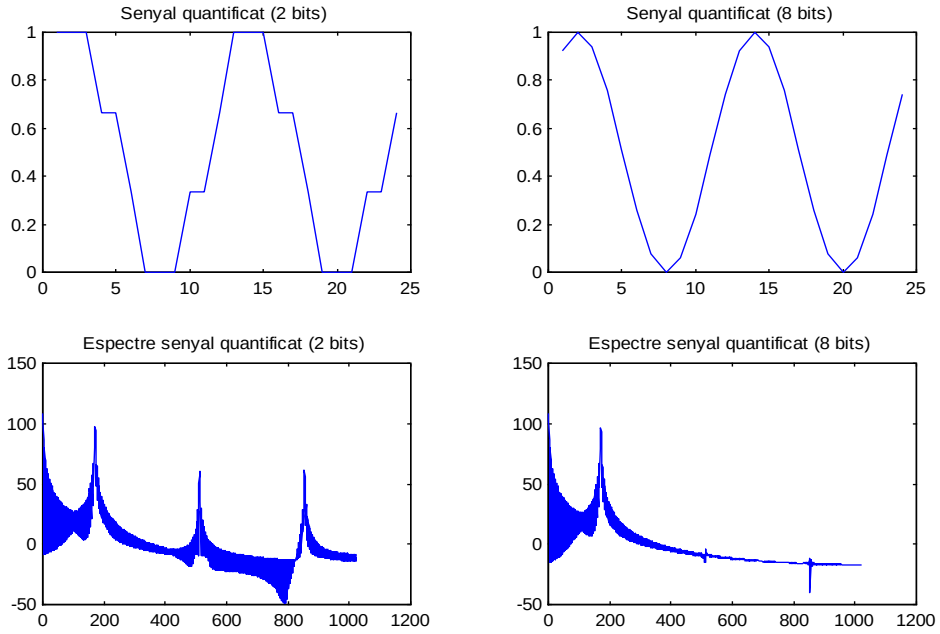


Figura 22. Error de cuantificación por señales pequeñas

En el caso del conversor de 8 bits, el espectro de la señal es el esperado: un tono a la frecuencia digital $1/12$ (muestra $2048/12 \cong 170$) y una componente de continua. En el caso del conversor de 2 bits aparecen, además, picos frecuenciales observados en las muestras múltiples impares de la anterior: $3 \cdot 2048/12$ y $5 \cdot 2048/12$. Estos tonos son consecuencia de dos factores:

- La señal de entrada está codificada con pocos niveles.
- La señal de entrada está sobremuestreda.

Fijémonos en que la coincidencia de estos dos factores provoca que la señal de salida contenga muy a menudo muestres consecutivas de igual valor. La conversión A/D da como resultado pulsos temporales de amplitud igual a los valores de cuantificación. Estos pulsos provocan la aparición de harmónicos en los múltiples impares de la frecuencia de la señal de entrada.

Si interpretamos el conversor A/D como un elemento que simplifica el mundo continuo en una aproximación discreta, en el caso que nos ocupa la aproximación no es fiel a la realidad y puede dar lugar a malas interpretaciones.
Con tal de resolver este problema podemos proceder de dos maneras.

La primera de ellas es bastante directa: aumentar la resolución del conversor A/D de manera que la señal de entrada ocupe 5 ó 6 bits: nos encontraremos pues en el caso estudiado en el apartado anterior.
La segunda manera resulta más interesante. Dado que el problema está causado por las muestras consecutivas de igual valor, podemos forzar cambios en la entrada para romper estas secuencias. Estos cambios los introduciremos añadiendo ruido a la entrada. Esta técnica recibe el nombre de *dithering*.

¿Qué amplitud deberá tener el ruido de entrada? Cualquier amplitud que rompa les secuencias de valores iguales será adecuada. Empíricamente se ha visto que un ruido de amplitud mediano en valor absoluto de $\sqrt{2}\cdot\Delta$ es adecuada.
La aplicación de un ruido en la entrada del conversor A/D tiene como efecto secundario la pérdida de margen dinámico. Cuantifiquémoslo en el caso que nos ocupa:

$$P_{dithering} = 2\cdot\Delta^2 \rightarrow \Delta P = 10\cdot\log\left(\frac{\Delta^2/12}{2\cdot\Delta^2}\right) = -13.8\,dB \cong -2\,bits$$

Ecuación 9. Pérdida de bits por culpa del dithering

Aplicando el ruido de amplitud mediano $\sqrt{2}\cdot\Delta$ el sistema pierde 2 bits, pero el espectro no lleva a confusión. Veámoslo en la gráfica siguiente:

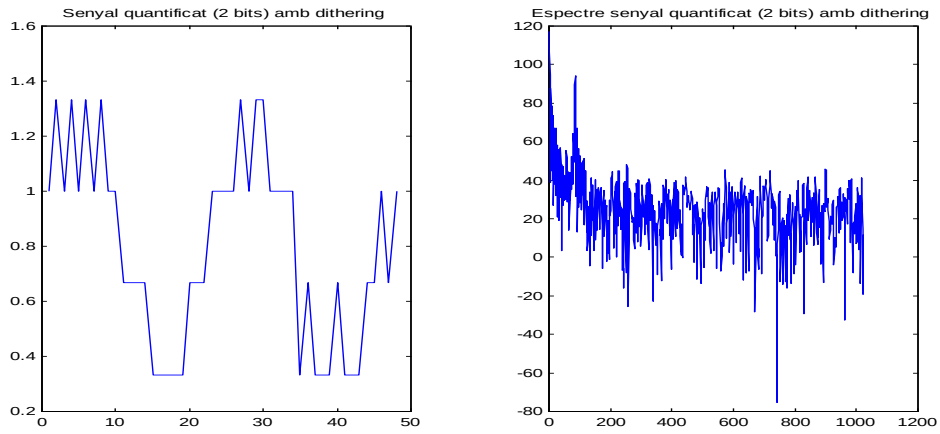


Figura 23. Efectos del *dithering*

El nivel de ruido global del sistema ha aumentado, pero el espectro es mucho más nítido.
Si requerimos la misma resolución de antes tendremos que aumentar en 2 el número de bits del sistema (recordemos que la solución 1 requería un aumento mayor de bits).

❖ Otros esquemas de dithering

El proceso descrito en el apartado anterior se puede implementar mediante el esquema siguiente:

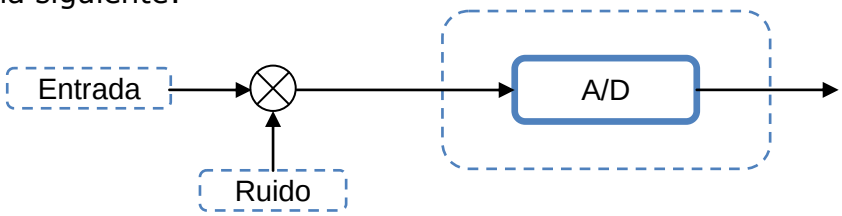


Figura 24. Esquema de aplicación de *dithering*, I

Este esquema tiene la ventaja de la simplicidad, pero tiene el inconveniente de la pérdida de margen dinámico (cuantificado en 13.8 dB por el caso estudiado en el apartado anterior).

Hay otras maneras de implementar el *dithering*. A continuación describiremos un par más.

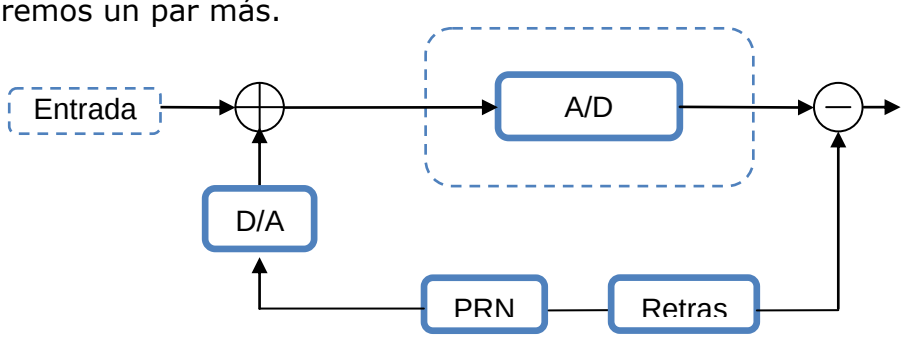


Figura 25. Esquema de aplicación de *dithering*, II

El esquema anterior basa su funcionamiento en la generación digital del ruido de manera que este ruido se resta de la señal una vez que ésta ha sido convertida por el A/D. Está compuesto por los elementos siguientes:

- PRN (*Pseudo Random Noise*): este módulo genera valores pseudo aleatorios digitalmente.

- El D/A convierte a analógico los números pseudo aleatorios generados por el PRN.

- El Retraso sincroniza las dos entradas del restador. Su presencia es necesaria para simular el retraso de conversión del A/D.

- El restador elimina el ruido de la muestra convertida (que contiene señal y ruido).

Este esquema presenta un mayor margen dinámico que el anterior ya que el ruido introducido se elimina digitalmente una vez que ya ha hecho su

trabajo. A pesar de esto, la recuperación de margen dinámico no es total, ya que el valor de ruido restado no tiene porqué corresponder con el valor de ruido cuantificado por el conversor A/D.

Como inconveniente, destacar la mayor complejidad de implementación debido sobre todo a la presencia del conversor D/A.

Otra posibilidad de implementación del *dithering* es la siguiente:

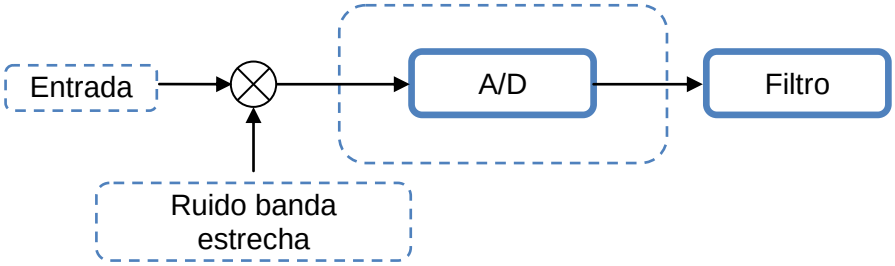


Figura 26. Esquema de aplicación de *dithering*, III

En este caso el ruido añadido a la entrada del conversor ha de tener un espectro que no pise el ancho de banda de la señal de entrada. Por tanto, si el conversor A/D muestrea por encima de dos golpes el ancho de banda total del sistema, el ruido podrá ser eliminado mediante un filtro digital. Veámoslo en la figura siguiente:

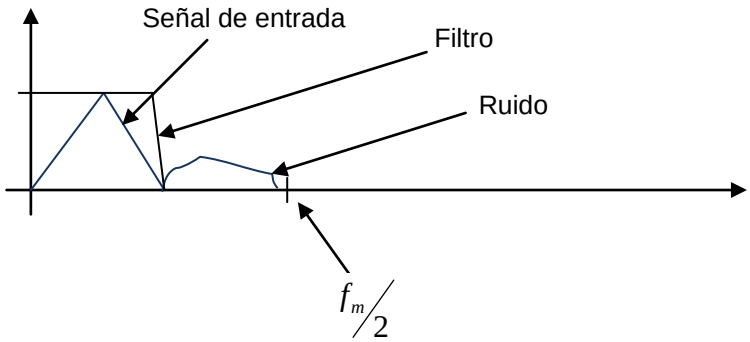


Figura 27. *Dithering*. Funcionamiento esquema III

Este esquema, igual que el anterior, tiene unes prestaciones mejores que el sistema básico ya que no se pierde margen dinámico del conversor. Por contra, en este caso hace falta un A/D que sea capaz de muestrear más rápido que el anterior. Además, el filtro tendrá que ser implementado en un *hardware* digital muy rápido.

2.1.4. Oversampling (sobremuestreo)

La técnica de *oversampling* consiste en muestrear una señal muy por encima de la frecuencia mínima que exige el teorema de Nyquist. Este sobre esfuerzo del A/D se verá compensado por un incremento del margen dinámico del sistema.

Para poder explicar la técnica, compararemos dos situaciones. En el primer caso, muestrearemos una señal con ancho de banda *B* a una frecuencia $f_{m1} = 2 \cdot B$, y en el segundo caso a una $f_{m2} = 4 \cdot 2 \cdot B$. En ambos casos el conversor será de *n* bits.

En la figura siguiente, el espectro superior corresponde al caso 1. Se ha marcado en azul el error de cuantificación correspondiente al conversor de *n* bits. Este error de cuantificación lo suponemos uniformemente distribuido. El espectro inferior corresponde al segundo caso, en el que la frecuencia de muestreo es cuatro veces superior. Igualmente, se ha marcado en azul le error de cuantificación.
Fijémonos en que la potencia total de error es la misma en los dos casos (sólo depende del fondo de escala y del número de bits del A/D), pero ahora está distribuida en un margen de frecuencias cuatro veces superior y, por tanto, la densidad espectral de potencia es cuatro veces inferior.

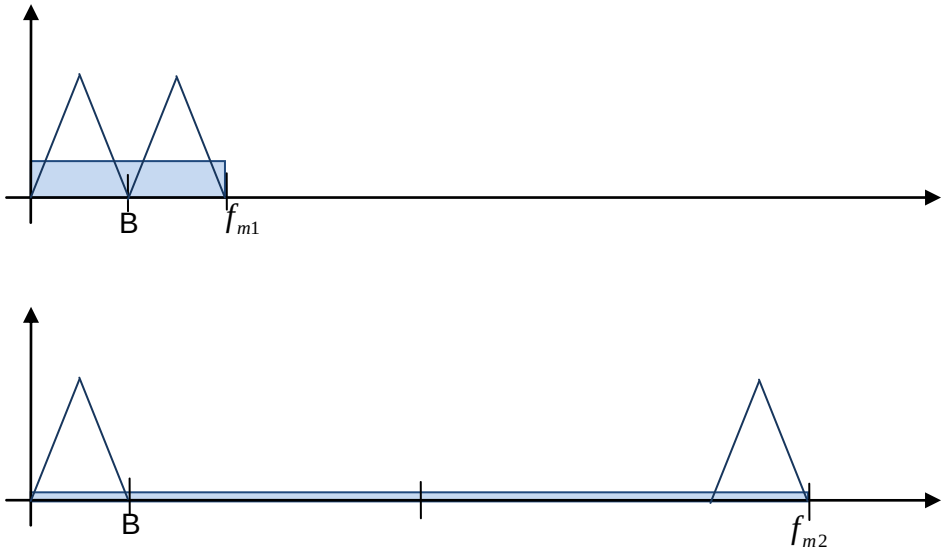


Figura 28. *Oversampling* y error de cuantificación.

En el segundo caso podemos proceder de la manera siguiente. Filtro de paso bajo con frecuencia de corte a *B*. La señal quedará intacta, mientras que el ruido por encima de *B* será atenuado (casi eliminado):

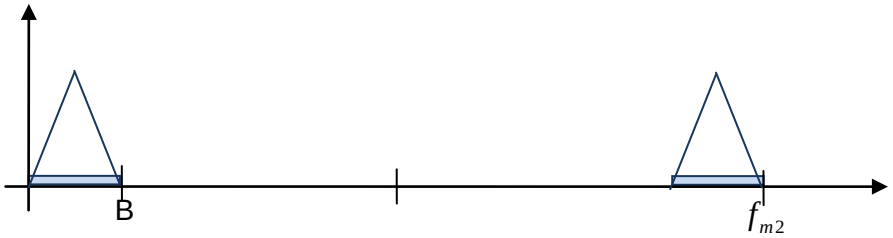


Figura 29. Filtrado de ruido durante el proceso de *oversampling*.

Seguidamente podemos diezmar en un factor cuatro para obtener el resultado final del proceso de *oversampling*:

Figura 30. Delmación en el proceso de o*versampling*

Si comparamos el espectro anterior con el espectro obtenido en el caso 1 observamos que hemos reducido la potencia de error de cuantificación a una cuarta parte del su valor. Hemos mejorado, pues, la relación S/QN en 6 dB, mejora equivalente a disponer de un conversor A/D de un bit más de resolución.
Si lo hacemos en el caso general en que procesamos la señal a una frecuencia de muestreo α golpes mayores a la requerida por *Nyquist*, podemos cuantificar la mejora experimentada en términos de números de bits:
El ruido de cuantificación en el segundo caso será

$$QN_2 = \frac{1}{\alpha}\cdot\frac{\Delta^2}{12} = \frac{FSR^2}{12\cdot\alpha\cdot 2^{2\cdot n}}$$

Ecuación 10. SQNR durante el proceso de oversampling, y

Para poder simplificar e interpretar la expresión escribimos α como a potencia de 2: $\alpha = 2^{\beta}$. Entonces:

$$QN_2 = \frac{FSR^2}{12\cdot 2^{\beta}\cdot 2^{2\cdot n}} = \frac{FSR^2}{12\cdot 2^{2\cdot\left(n+\frac{\beta}{2}\right)}} = \frac{FSR^2}{12\cdot 2^{2\cdot n'}}$$

Ecuación 11. SQNR durante el proceso de oversampling, II

Hemos llegado a una expresión de ruido de cuantificación tal y como la que hemos deducido en un apartado anterior, donde el número de bits del conversor viene dado por:

$$n' = n + \frac{\beta}{2}$$

Ecuación 12. Ganancia de bits durante el proceso

Donde β está directamente relacionada con el factor de sobremuestreo. Veámoslo en la tabla siguiente:

Factor sobremuestreo	B	Ganancia en bits	Ganancia en dB
4	2	1	6
8	3	1.5	9
16	4	2	12

Figura 31. Ganancia en bits y dB en el proceso de *oversampling*

Por tanto, mediante el proceso descrito podemos intercambiar frecuencia de muestreo por número de bits. Por ejemplo, es equivalente muestrear a f_0 y n bits, que hacerlo a $4 \cdot f_0$ y $(n-1)$ bits aplicando filtrado y delmación.

Propongamos como ejercicio demostrar que la ganancia en dB de la relación S/QN al sobremuestrear en un factor α viene dado por:

$$10 \cdot \log_{10}(\alpha)\,[dB]$$

Ecuación 13. Ganancia en dB durante el proceso

La ganancia en bits la podemos escribir también como:

$$\frac{10}{6} \cdot \log_{10}(\alpha)$$

Ecuación 14. Ganancia en número de bits durante el proceso

Para finalizar el apartado, proponemos dos cuestiones a reflexionar.

- ¿Cómo interpretar el hecho que el factor de sobremuestreo pueda producir un número de bits no entero?

- ¿Encontráis alguna situación en la que, por ejemplo, muestrear a f_0 y n bits no sea totalmente equivalente a hacerlo a $4 \cdot f_0$ y $(n-1)$?

2.1.5. Undersampling (submuestreo)

El proceso de muestreo se observa frecuencialmente en forma de creación de repeticiones en los múltiples de la frecuencia de muestreo. Si este proceso de replicación no provoca solapamientos, el muestreo no implica pérdida de información. Este razonamiento es equivalente a requerir una frecuencia de muestreo superior a dos veces el ancho de banda de la señal.

Normalmente origen espectral de las repeticiones se sitúa a la banda base (figura 32): el ancho de banda de la señal coincide con su frecuencia máxima ya que el espectro de la señal comienza a la frecuencia cero. Nada nos impide, pero, aplicar el mismo proceso a una señal paso banda (figura 33). En este caso, $B = f_{max} - f_{min}$, donde $f_{min} \neq 0$.

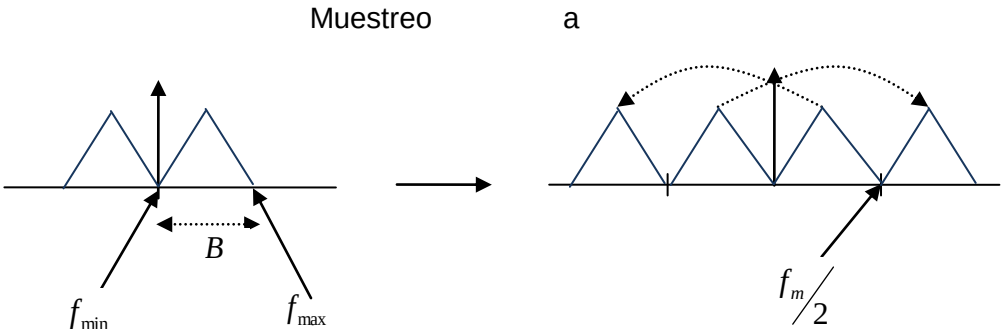


Figura 32. Muestreo de una señal banda base

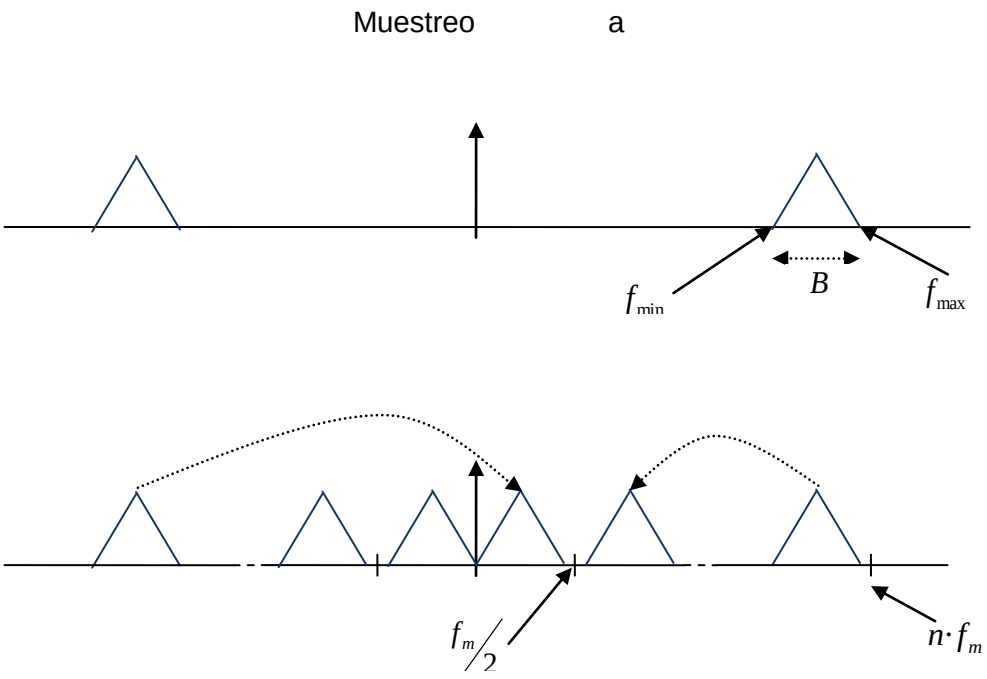


Figura 33. Muestreo de una señal pasa banda

El proceso de un*dersampling* consiste precisamente en esto: muestrear una señal paso banda de ancho de banda limitado a una frecuencia $f_m \geq 2 \cdot B$, donde $B = f_{\max} - f_{\min}$. De esta manera conseguimos no solamente discretizar la señal en tiempo y amplitud, sino a más trasladar el espectro en frecuencia. El conversor en cuestión, entonces, actúa también como mezclador.

Como criterio de diseño resulta aconsejable que una de las réplicas procedentes del proceso de muestreo acabe centrada a $f_m/4$. A tal efecto calcularemos la f_m siguiendo la secuencia siguiente de operaciones:

$$(1)\ f_m \geq 2 \cdot B\ ;\ (2)\ n \cdot f_m - \frac{f_m}{4} = IF_1\ ;\ (3)\ \overline{n} \cdot f_m - \frac{f_m}{4} = IF_1$$

Ecuación 15. Cálculo del undersampling

donde IF_1 es la frecuencia central de la señal analógica y n es el múltiplo útil de la frecuencia de muestreo.

Aclarémoslo: Con la ecuación (1) formulamos una primera aproximación de la frecuencia de muestreo (f_m). Con la ecuación (2) calculamos el múltiplo útil de la frecuencia de muestreo (n) de manera que una repetición de el espectro (una vez discretizado) se sitúe centrada a $f_m/4$. Este múltiple útil ha de ser entero, aunque el resultado de la Ecuación puede no serlo. En este caso, es necesario truncarlo al entero inferior ($\overline{n}$) y recalcular la frecuencia de muestreo (f_m) con el nuevo valor de $\overline{n}$ mediante la Ecuación (3).

El signo menos de las expresiones (2) y (3) de la Ecuación 15 provocará una inversión del espectro. Si no la deseamos, tendremos que usar el signo positivo, y el proceso de cálculo será análogo.
Hay que tener en cuenta una serie de consideraciones a la hora de trabajador con un sistema submuestreado:

- El proceso descrito en la figura 33 suponía un espectro vacío en todas partes menos en el ancho de banda de la señal de interés. En general este supuesto no será cierta: habrá aplicar un filtro *anti-aliasing* que en este caso será paso banda. Este filtro lo diseñaremos de manera que las frecuencias de corte coincidan con $f_{\min}$ y $f_{\max}$, y las frecuencias de atenuación con las frecuencias susceptibles de producir *aliasing* sobre la réplica banda base.

En estas frecuencias exigiremos que el filtro atenúe, como mínimo, una cantidad de dB igual a la relación SQNR del conversor A/D. Lo veremos más adelante en un ejemplo.

- El concepto de ancho de banda normalmente se define como el intervalo entre las dos frecuencias a las cuales la señal cae 3 dB. El ancho de banda entendido de esta manera implica que más allá del ancho de banda la señal contiene potencia que, submuestreando, es susceptible de producir *aliasing*. El filtro analógico *anti-aliasing* será el encargado de evitarlo.

A continuación veremos un ejemplo en el que se explica el proceso de diseño de un submuestreo.

Ejemplo:

Siendo 1 banda de frecuencias compuesta por 5 canales de 700 khz de BW (pasa banda), situados equidistantemente 1 Mhz. Decidimos utilizar un filtro de canal analógico y un conversor A/D de 6 bits como mezclador (submuestreo) con tal de bajar en frecuencia una de las bandas. La IF_1 en la que está centrada la banda de interés es de 20 Mhz.

a. Determina la frecuencia de muestreo del conversor A/D.
b. Determina las características frecuenciales deseables del filtro paso banda.

En primer lugar, destacar que la situación normal que abordaremos en esta asignatura no es la de un receptor monocanal como nos propone el enunciado. El ejercicio se propone únicamente con la intención de mostrar el proceso de diseño utilizando submuestreo.

Recordemos de la teoría en este apartado las condiciones sobre los parámetros del conversor,

$$f_m \geq 2 \cdot BW$$

$$n \cdot f_m - \frac{f_m}{4} = IF_1$$

donde,

f_m es la frecuencia de muestreo
BW es el ancho de banda (pasa banda) del canal
IF_1 es la frecuencia en la cual está centrado el canal

Apartado a.

Apliquémoslo en el caso planteado por el problema. El ancho de banda del canal es 700 khz. La mínima frecuencia de muestreo que podemos escoger

es 1.4 MSPS. Si la escogemos de un valor superior relajaremos las condiciones sobre el filtro pasa banda analógica. Fijémosla, por ejemplo, a 2 MSPS (Nota. Hubiera estado igualmente correcto tomar como primera aproximación 1.4 MSPS, e incrementar la frecuencia de muestreo final más adelante.
En cualquier caso, en algún momento del proceso hay que ser consciente de que una f_m muy cercana a 2· *BW* no es viable ya que requeriría una pendiente del filtro tendiendo a infinitos dB por década).

Seguidamente aplicamos la segunda Ecuación para encontrar el harmónico útil que llevará a término la bajada del canal.

$$n \cdot 2 - \frac{1}{2} = 20 \rightarrow n = 10.25 \rightarrow \bar{n} = 10$$

El valor obtenido a partir de la Ecuación ha sido 10.25. Como este número ha de ser, no obstante, entero, lo redondeamos a la baja a 10. A continuación se tendrá que reajustar la frecuencia de muestreo por $\bar{n}$=10,

$$10 \cdot f_m - \frac{f_m}{4} = 20 \rightarrow f_m = 2.05\, MSPS$$

Nota. Cualquier valor entero $\bar{n}$ que produzca una frecuencia de muestreo por encima de 2· *BW* es correcto. Así, si deseamos una f_m mayor de 2.05 MSPS y que cumpla las ecuaciones del submuestreo, podemos tomar $\bar{n}$=9, 8, 7, ...

Apartado b.
En las figuras siguientes podemos observar el proceso de submuestreo (por comodidad, sólo pintamos 3 canales: el de interés y los dos que lo rodean):

Antes del conversor A/D:

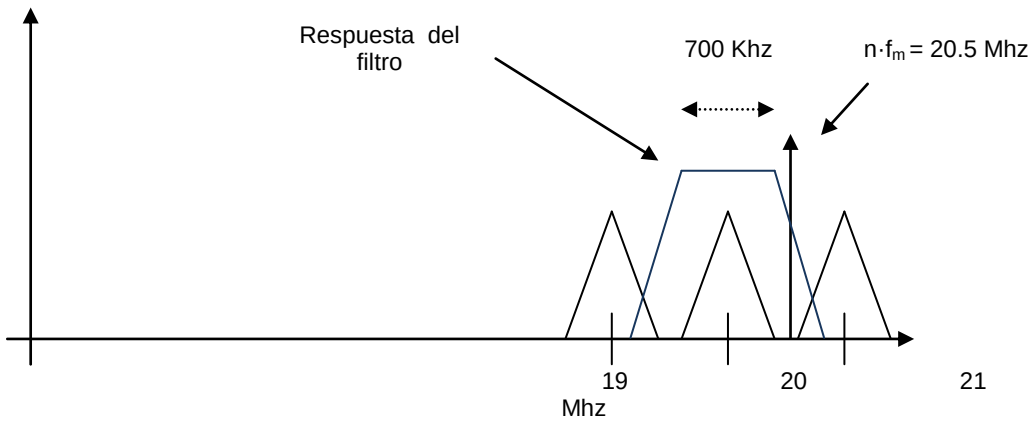


Figura 34. *Undersampling* y filtrado pasa banda

Después del conversor:

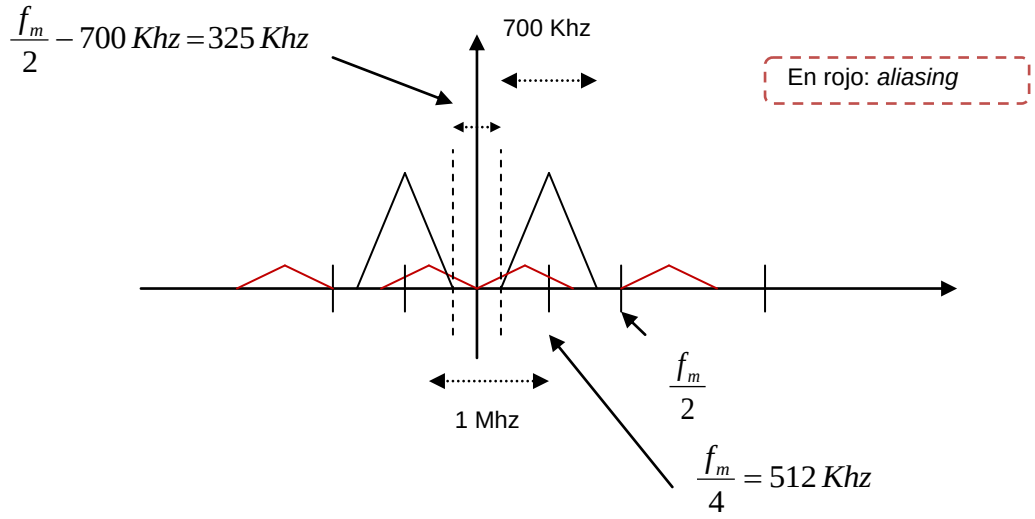


Figura 35. *Undersampling*. Cálculo características del filtro

Observemos cuáles tendrían que ser las características deseables del filtro analógico:

- Filtro centrado a 20 Mhz
- Ancho de banda igual al ancho de banda del canal: 700 Khz
- Banda de atenuación 325 Khz más allá de la frecuencia de corte
- Atenuación mínima en la banda de atenuación. Hay que tener en cuenta los puntos siguientes:

  - Queremos que los canales adyacentes no provoquen aliasing
  - Como el conversor A/D tiene 6 bits -> 36 dBs de margen dinámico.

  Por tanto, la atenuación mínima ha de ser de 36 dBs.
  Este requerimiento trasladado a pendiente en la banda de transición podemos cuantificarlo de la manera siguiente:

$$Pendent = \frac{(-3)-(-36)}{\log(20.35+0.325)-\log(20.35)} = \frac{33}{6.9\cdot10^{-3}} = 4787\,dBs/dec$$

Una pendiente, entonces, extraordinaria. Y eso que hemos trabajado con conversores de sólo 6 bits y hemos sido generosos a la hora de escoger la frecuencia de muestreo. Con tal de simplificar el filtro, tendríamos que aumentar la frecuencia de muestreo reduciendo el valor de $\bar{n}$.

2.2. El conversor A/D real

Los aspectos discutidos en el apartado anterior estudian el conversor desde un punto de vista ideal en el sentido que no tienen en cuenta consideraciones de implementación de estos dispositivos. Los parámetros tecnológicos del proceso de fabricación introducen defectos en el comportamiento del A/D que son relevantes y degradan las prestaciones de los conversores. En este apartado estudiaremos cuáles son estos defectos, cómo es cuantifican y qué consecuencias tienen.

2.2.1. Errores estáticos

Los errores estáticos son aquellos que tienen relevancia sea cual sea la frecuencia de la señal de entrada. Es decir, introducen defectos en el proceso de conversión que no son selectivos en frecuencia.

Todos estos errores se ponen de manifiesto al comparar la función de transferencia del conversor ideal y la de un conversor real. Veamos un ejemplo en la figura siguiente:

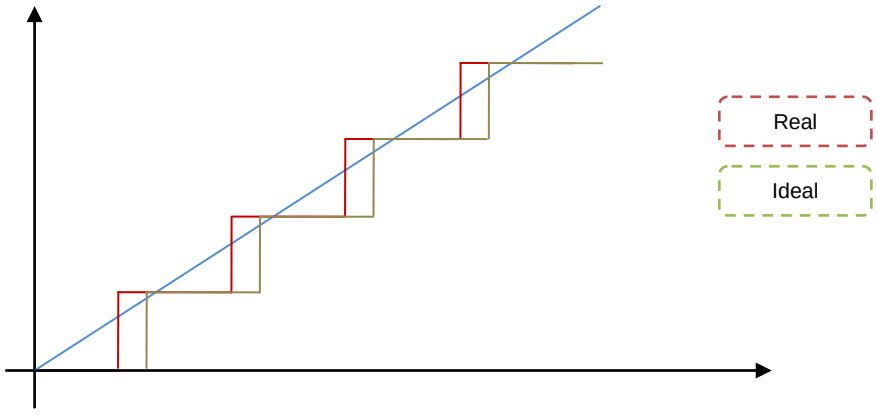


Figura 36. Función de transferencia ideal y función de transferencia real

Las diferencias entre las dos funciones de transferencia se pueden clasificar en cuatro tipos:

❖ Error de offset

El error de *offset* se da cuando la primera transición de la función de transferencia no se produce por una amplitud de la señal analógica de entrada de medio LSB (o 1 LSB si el conversor es de truncamiento).

Este error se puede corregir ajustando analógicamente el *offset* de la señal de entrada con el fin de hacer coincidir la primera transición por el valor de tensión analógica correcta.

❖ Error de ganancia

El error de ganancia es análogo al de *offset*, pero ahora no nos fijamos en la primera transición sino en la última. En este caso se puede corregir ajustando la ganancia del preamplificador de entrada.

❖ Error de linealidad diferencial (DNL)

La DNL (*Differential Non Linearity*) se produce cuando el ancho de algún intervalo de la función de transferencia difiere de 1 LSB. Este error no se puede corregir ya que puede afectar a muchos intervalos diferentes y la corrección se debería hacer por cada uno de ellos. El fabricante no da este parámetro por cada intervalo con error sino que se limita a dar un hito máximo de la desviación, expresada en LSB.

❖ Error de linealidad integral (INL)

La INL (*Integral Non Linearity*) se produce cuando los puntos de cambio entre intervalos de la función de transferencia real no coinciden con los de la función de transferencia ideal. Este error puede parecer a primera vista redundante respecto la DNL, pero no lo es. Fijémonos en que, por ejemplo, un conversor como la DNL de cada intervalo fuera de una decena parte de un LSB podría parecer bueno. No lo es, pero, si consideramos que al cabo de 100 intervalos, si la DNL es siempre positiva, produciría un error INL de 10 LSB.
La INL, por tanto, integra (suma y compensa) las DNL de cada intervalo. El error no se puede corregir tampoco. El fabricante también da una hito máxima de la desviación.

❖ Consecuencias y cuantificación de los errores

Ya hemos visto que, aunque tanto los errores de *offset* como de ganancia se pueden corregir, no pasa así los de DNL e INL. ¿Qué consecuencias tienen estos errores en lo que respecta a las prestaciones del conversor?

La función de transferencia ideal del conversor A/D es claramente no lineal: presenta discontinuidades cada LSB. Esta no linealidad se modela en forma de error (ruido) de cuantificación, de manera que el comportamiento del

conversor A/D ideal se suele modelar como una respuesta lineal más un ruido de cuantificación. La DNL y la INL introducen nuevas no linealidades en la función de transferencia del conversor que se manifiestan produciendo distorsión e intermodulación. Esto significa que si la señal de entrada a un A/D es un tono puro, a la salida aparecerá no sólo este tono, sino armónicos suyos producto de la DNL y la INL.

Los fabricantes acostumbran a dar información de la DNL y la INL a partir de sus consecuencias (distorsión) mediante los parámetros siguientes:

- **SFDR** (*Spurious Free Dynamic Range*). Parámetro que mide en la salida de una A/D la relación en dB entre el nivel de un todo a fondo de escala y el mayor de sus armónicos producidos por la no linealidad del conversor (DNL e INL).

- **ENOB** (*Effective Number of Bits*). La aparición de los armónicos puede provocar pérdidas de información en los bits de menor peso. Si consideramos todos los posibles efectos, podemos escribir:

$$SINAD = 6.02 \cdot ENOB + 1.76 \rightarrow ENOB = \frac{SINAD - 1{,}76}{6{,}02}$$

Ecuación 16. SFDR y ENOB

El SINAD incluye el ruido de cuantificación, el ruido de los conversores A/D, armónicos provocados por no linealidades, etc. Por tanto, el ENOB es una medida global del número de bits efectivos del conversor A/D teniendo en cuenta todos los factores del diseño.

- **RB** (*Resolution Bandwith*). Las no linealidades derivadas de la INL y DNL se agregan a altas frecuencias. Esto implica que el SFDR no es estático, sino que depende de la frecuencia de medida. El RB mide esta degradación de las prestaciones del conversor A/D definiendo un margen dinámico de resolución como el conjunto de frecuencias a las cuales la pérdida de resolución a causa de las no linealidades es inferior a medio bit (3 dB).

$$RB = f\big|_{ENOB=n-1/2}$$

Ecuación 17. RB

2.2.2. Errores dinámicos

Los errores dinámicos son aquellos cuyos efectos se hacen patentes como consecuencia de la variación de la señal de entrada. Establecen un límite de frecuencia máxima de la señal analógica de entrada para obtener unas prestaciones dadas. Veámoslo a continuación.

❖ Incertidumbre sobre el tiempo de obertura

El circuito de *sancho & hold* en la entrada de un conversor A/D se puede modelar con el esquema siguiente:

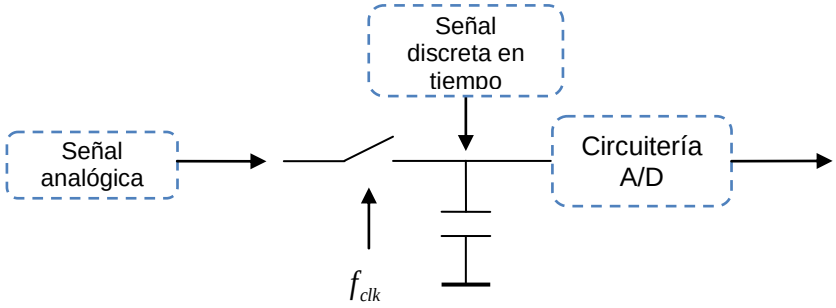


Figura 37. Circuito de *sancho & hold*

El circuito tiene dos estados:

- Estado de *sancho*. Con el interruptor cerrado, el condensador se carga en la tensión de entrada

- Estado de *hold*. Con el interruptor abierto el condensador mantiene el valor de tensión de entrada muestreado y la circuitería A/D lo cuantifica

En la gráfica siguiente podemos ver la evolución temporal de estas tres señales (estado del circuito, señal analógica y señal en la entrada de la circuitería del A/D).

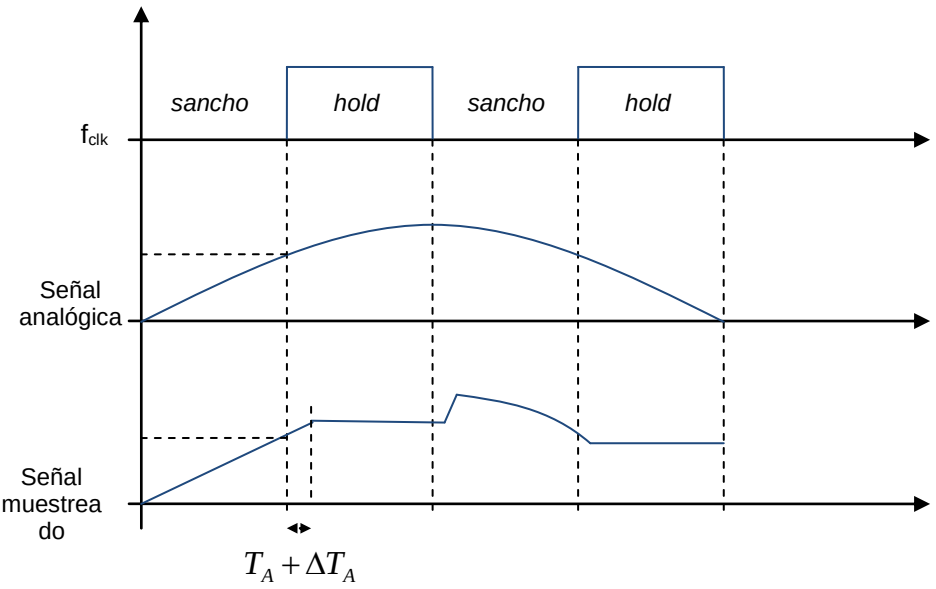


Figura 38. Funcionamiento del circuito *sancho & hold*

El detalle que nos interesa analizar a continuación es el momento de transición de *Sancho* a *Hold*. Este cambio de estado no es instantáneo. De hecho se suele desglosar en dos magnitudes:

- Tiempo de obertura (T_A). Corresponde al tiempo medio que tarda el interruptor en abrirse.

- Incertidumbre sobre el tiempo de obertura (ΔT_A). Cuantifica la variación máxima del tiempo de obertura del interruptor respecto al tiempo medio.

Por tanto, el retraso de obertura del interruptor tiene un margen de valores comprendidos entre $\left(T_A - \Delta T_A/2, T_A + \Delta T_A/2\right)$. Esta variabilidad debida a factores diversos atribuibles normalmente a ruido de *jitter* y similares, provoca que la frecuencia de muestreo no sea perfectamente periódica. La variación máxima entre dos tiempos de muestreo consecutivos será de ΔT_A.

Este factor sólo es grave a altas frecuencias (tiempo de muestreo pequeño). Para cuantificar el efecto que este parámetro tiene sobre las prestaciones del conversor se suele utilizar un modelo que razona de la manera siguiente: la variación del periodo de muestreo a causa de la incertidumbre sobre el tiempo de obertura no será importante siempre y cuando la variación del valor cuantificado causado por este efecto sea inferior a una cantidad dada (normalmente $\frac{1}{2}LSB$).

De aquí que la consecuencia directa de la incertidumbre sobre el tiempo de obertura sea limitar la máxima variación de la entrada durante la incertidumbre sobre el tiempo de obertura; dicho de otra manera, limitar la máxima frecuencia de entrada.
A continuación encontraremos una expresión que cuantifica las consecuencias de este parámetro. Procedemos de acuerdo con la gráfica siguiente:

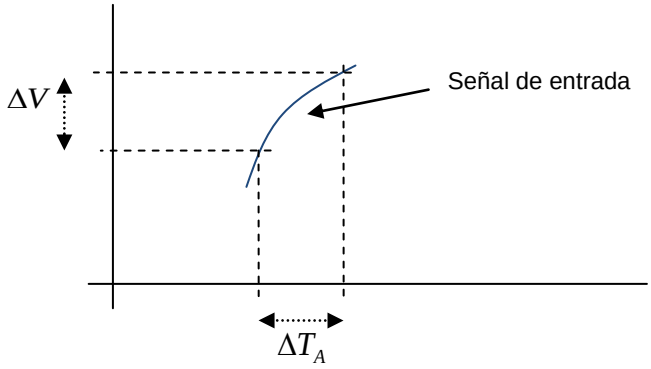


Figura 39. Modelo de cálculo del error de obertura

Sea una señal de entrada a fondo de escala: $2 \cdot A = FSR$. Lo introducimos en un conversor A/D de n bits y una incertidumbre sobre el tiempo de obertura de ΔT_A. Limitaremos la variación de la señal de entrada (ΔV) durante el tiempo ΔT_A a $\frac{1}{2} LSB$.

Utilizando la aproximación de la señal por su pendiente (ΔT_A es muy pequeño), encontramos:

$$\Delta V \leq \frac{1}{2} LSB \rightarrow 2 \cdot \pi \cdot f \cdot A \cdot \Delta T_A \leq \frac{1}{2} \frac{FSR}{2^n - 1} \rightarrow 2 \cdot \pi \cdot f \frac{FSR}{2} \cdot \Delta T_A \leq \frac{1}{2} \cdot \frac{FSR}{2^n}$$

$$f_{\max} \leq \frac{1}{\pi \cdot \Delta T_A \cdot 2^{n+1}}$$

Ecuación 18. Máxima frecuencia analógica

Llegamos, pues, a una expresión que nos da la máxima frecuencia de entrada que podemos introducir al conversor A/D si queremos que la variación del valor cuantificado por la incertidumbre sobre el tiempo de obertura sea, como máximo, de media LSB.

Hay que precisar, no obstante, que usar frecuencias de entrada inferiores a $f_{\max}$ nos asegura sólo que la variación de la señal de entrada (ΔV) durante la incertidumbre sobre el tiempo de obertura (ΔT_A) sea inferior $\frac{1}{2} LSB$. Esto,

no obstante, no implicará en todos los casos que el código digital obtenido después de la cuantificación sea el mismo que el que habría obtenido un conversor ideal. De hecho, esto no lo podremos asegurar por pequeño que sea el lindar de ΔV que fijemos; siempre existirá la posibilidad de cambio de código ya que el valor instantáneo de la tensión de entrada (V) puede estar arbitrariamente cercano al valor de transición de código. En cualquier caso, cuanto menor sea el lindar de ΔV menor será la frecuencia máxima admisible en la entrada del A/D y menor será, de media, la distorsión debida por la incertidumbre sobre el tiempo de obertura.

2.3. El conversor D/A

En este documento repasaremos algunos conceptos básicos relativos a los conversores D/A: su diagrama de bloques, la función de transferencia y sus características según el orden del interpolador. Seguidamente compararemos la respuesta y prestaciones de un conversor ideal respecto los reales así como las técnicas que se aplican para corregir el comportamiento de estos últimos y maximizar así sus prestaciones.

2.3.1. Definición

Un conversor digital a analógico (o DAC: *Digital to Analog Converter*) es un dispositivo que representa un número limitado de códigos digitales de entrada mediante un número idealmente igual de valores analógicos discretos de salida.

2.3.2. Diagrama de bloques

El diagrama de bloques de un D/A ideal se presenta en la siguiente figura:

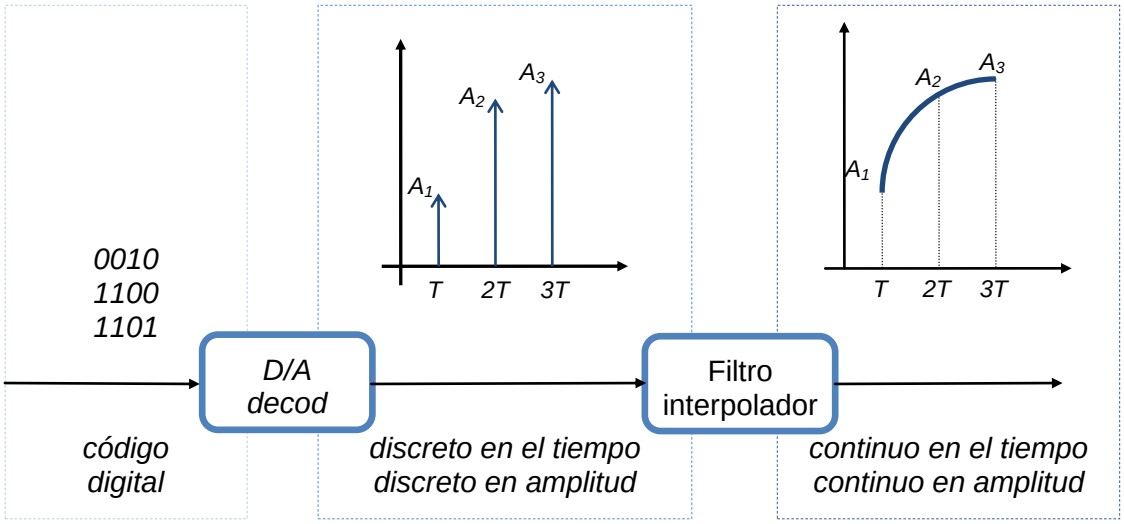


Figura 40. Diagrama de bloques de un conversor D/A

El bloque *D/A decod* es un bloque capaz de sacar, en su salida, un valor de tensión proporcional al código digital que tiene en su entrada (un descodificador). En su salida la señal es impulsional, discreta en tiempo.
El filtro interpolador es el encargado de darle continuidad temporal y de amplitud a la señal de salida del D/A.

El filtro interpolador por defecto de los conversores D/A reales es de orden 0: mantiene la señal constante entre instantes de muestreo. Podríamos diseñar otras órdenes de interpolador; por ejemplo la orden 1 une los valores convertidos mediante rectas, el orden 2 mediante parábolas, etc. El interpolador ideal, el requerido por el teorema del muestreo, es de orden infinito.

Evidentemente no habrá ninguna D/A real con este orden de interpolador, motivo por el cual habrá que añadir otros componentes (filtros) a la salida del D/A para complementar la respuesta global.

2.3.3. Función de transferencia

❖ El D/A decod

La función de transferencia del *D/A decod* ideal se presenta en la siguiente figura:

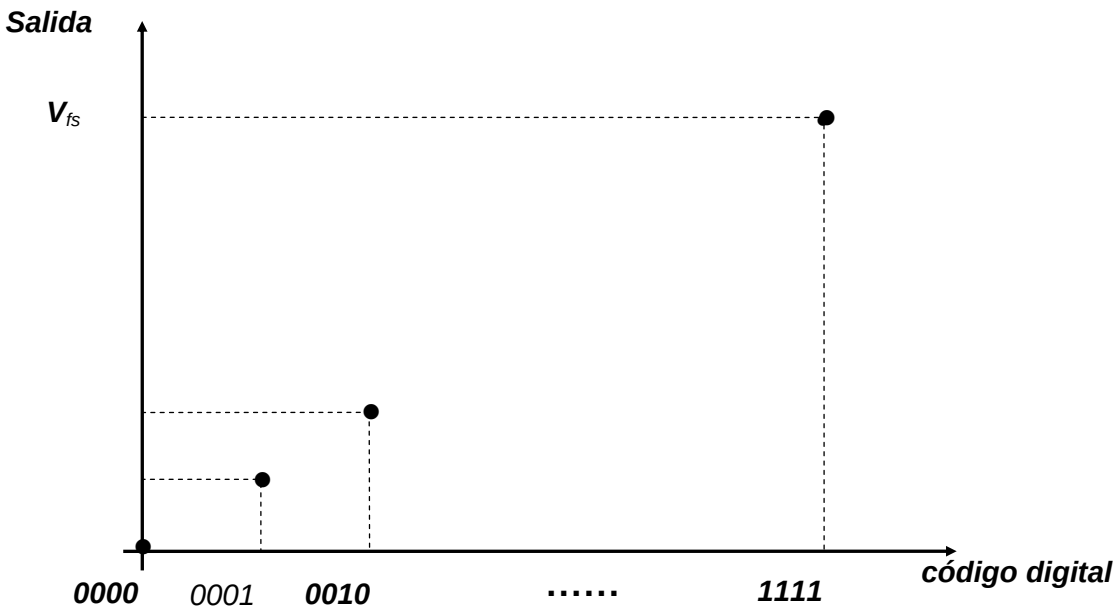


Figura 41. Función de transferencia de un conversor D/A

Como se puede apreciar, se trata de una función de transferencia completamente discreta, formada por puntos que asocian cada código digital de entrada con una tensión de salida. Esta función de transferencia sólo tiene sentido evaluarla en los instantes de conversión, momentos en los cuales el D/A dispone de un nuevo dato a procesar.

Las funciones de transferencia de los interpoladores las estudiaremos en los apartados siguientes.

2.3.4. El interpolador ideal

Como se ha comentado anteriormente, el conversor D/A ideal dispone de un filtro interpolador de orden∞, es decir, el propuesto por el teorema de Nyquist:

$$h(t) = \frac{\sin(2\pi Bt)}{2\pi Bt} \Leftrightarrow H(f) = \frac{1}{2B}\Pi\left(\frac{f}{2B}\right)$$

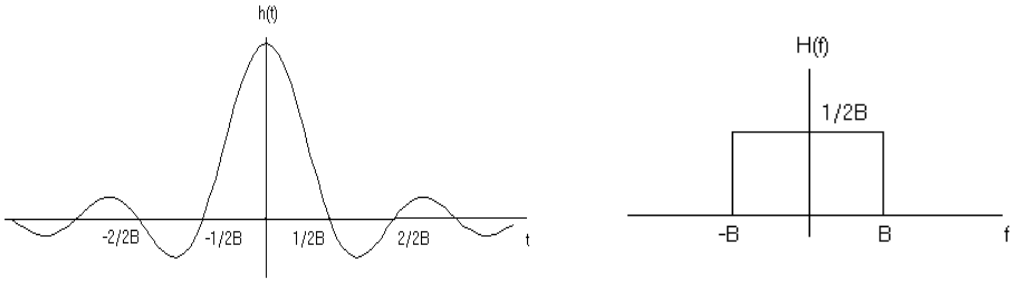


Figura 42. Interpolador ideal

El proceso de recuperación de la señal mediante el filtro interpolador de orden ∞ queda ilustrado, tanto en el dominio temporal como en el frecuencial, en la figura siguiente:

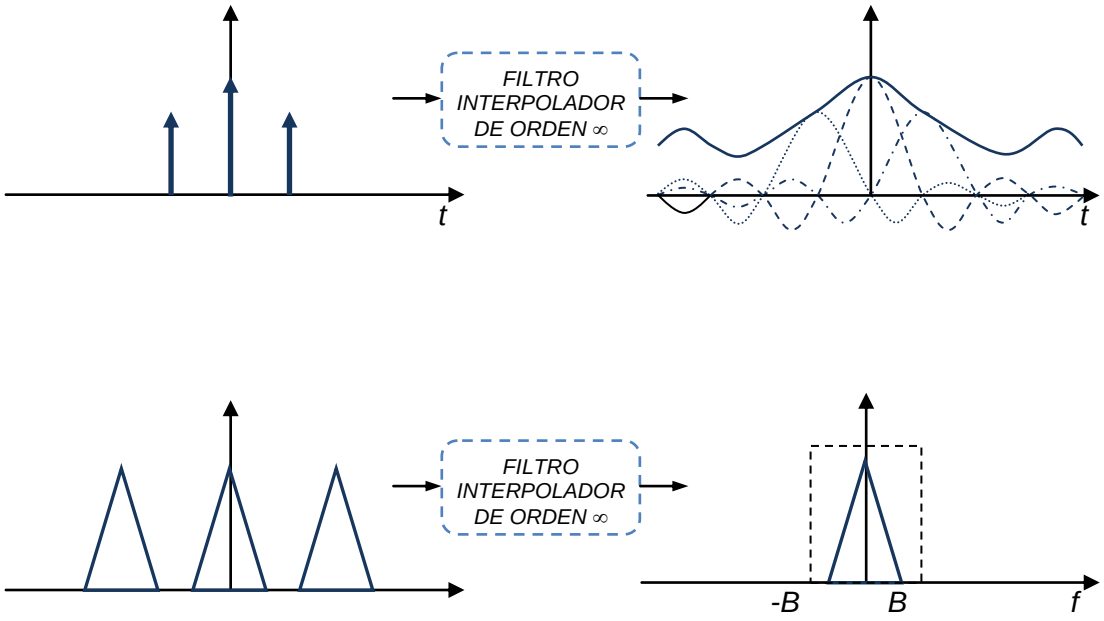


Figura 43. Respuesta temporal y frecuencial con interpolador de orden ∞

Como se puede apreciar, la reconstrucción que realiza este filtro es perfecta, y no introduce ninguna distorsión en la señal. Por desgracia un filtro interpolador de estas características es irrealizable, ya que es no causal y su pendiente espectral es infinitamente abrupta. Por tanto, en la práctica se deberá reducir el orden del filtro (habitualmente orden 0 ó 1).

2.3.5. Interpolador real. Orden 0

El interpolador de orden 0, también conocido como *cero-order hold* o *sancho & hold*, tiene una respuesta impulsional y una función de transferencia completamente opuestas a las del filtro interpolador de orden∞.

$$h(t) = \Pi\left(\frac{t - T/2}{T}\right) \Leftrightarrow H(f) = T \cdot \frac{\sin(\pi f T)}{\pi f T} \cdot e^{-j\pi f T}$$

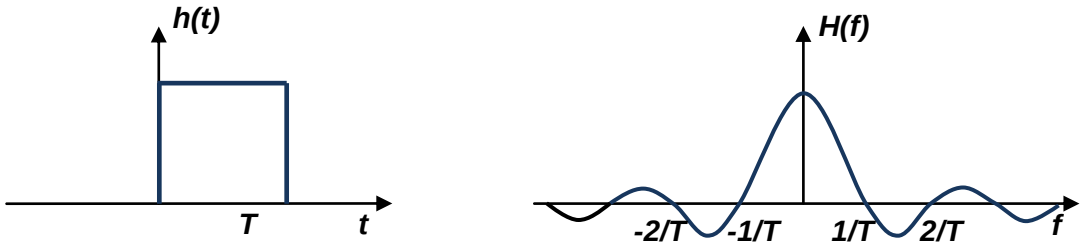


Figura 44. Interpolador de orden 0

El efecto, a nivel temporal, de la interpolación con *cero-order hold* se presenta en la siguiente figura:

Figura 45. Respuesta temporal del interpolador de orden 0

La transición entre pulsos adyacentes es brusca, lo que implica la aparición de información de alta frecuencia en la señal de salida. Este hecho se verifica en el análisis frecuencial de las señales implicadas en la interpolación de orden cero.

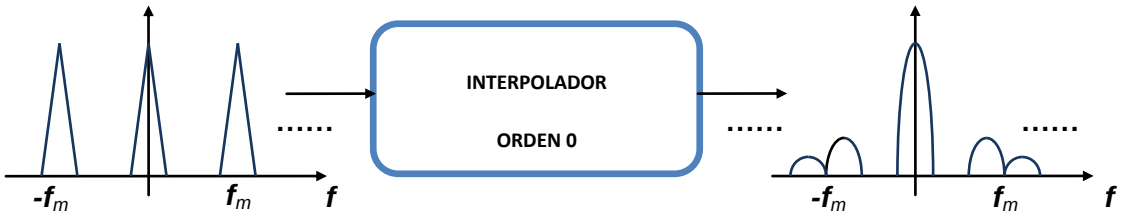


Figura 46. Respuesta frecuencial del interpolador de orden 0

En la figura anterior se puede ver cómo, además de añadir información de alta frecuencia, el *cero-order hold* introduce una distorsión del espectro de la señal de salida. Por tanto, será necesario añadir unos bloques posteriores al interpolador para eliminar la información de alta frecuencia y de compensar la distorsión introducida por éste.

Estos bloques serán:

- Un filtro pasa-bajos que atenúe los espurios de alta frecuencia situados en los múltiplos de f_m.

- Un filtro ecualizador del lóbulo principal, que tenga una respuesta opuesta en la *sinc* (conocida como corrector *sinx*/*x*).

Las respuestas frecuenciales de estos dos filtros son las siguientes:

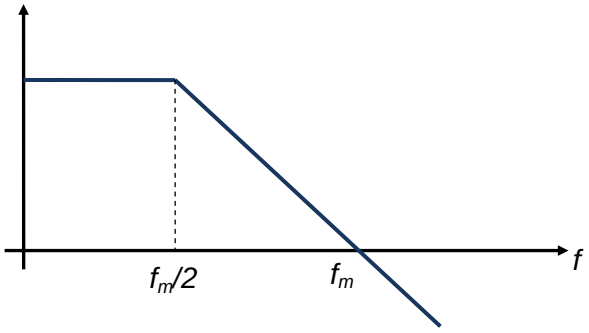


Figura 47. Paso bajo a la salida del D/A

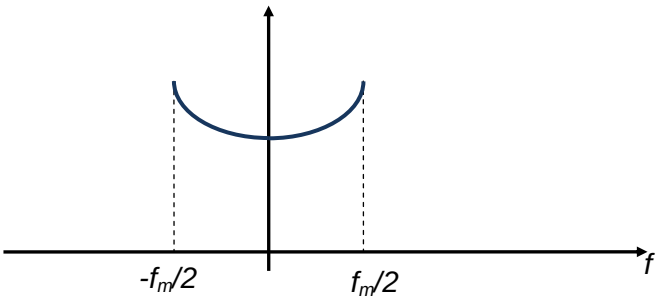


Figura 48. Ecualizador a la salida del D/A

Por tanto, la estructura definitiva del conversor D/A será la siguiente:

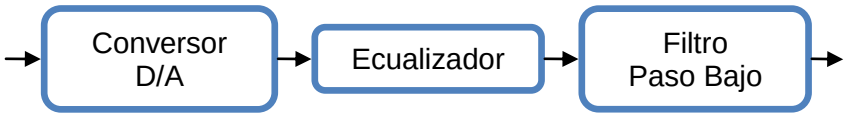


Figura 49. Conversor D/A y filtros

2.3.6. Interpolador real. Orden 1

El interpolador de orden 1, también conocido como *one-order hold*, tiene una respuesta impulsional y una función de transferencia de la forma siguiente:

$$h(t) = \Lambda\left(\frac{t-T}{T}\right) \Leftrightarrow H(f) = T \cdot \left(\frac{\sin(\pi f T)}{\pi f T}\right)^2 \cdot e^{-j2\pi f T}$$

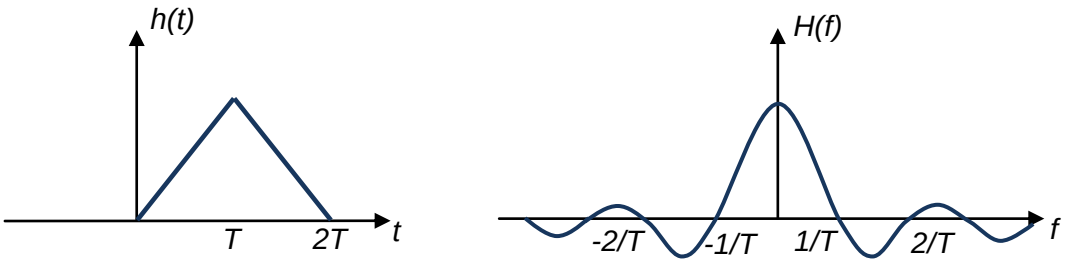


Figura 50. Interpolador de orden 1

El efecto, a nivel temporal, de la interpolación con *one-order hold* se presenta en la siguiente figura:

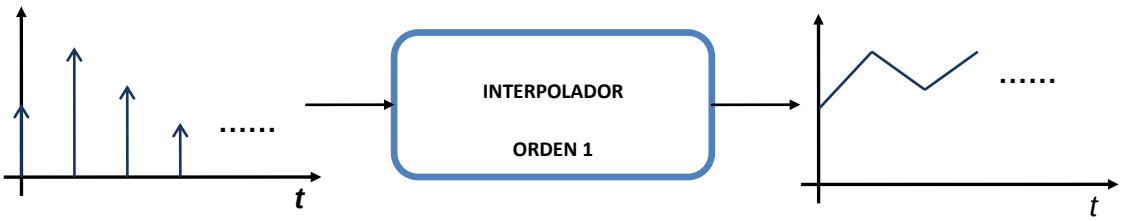


Figura 51. Respuesta temporal del interpolador de orden 1

Se puede apreciar que el interpolador de orden 1 interpola las muestras uniéndolas con rectas (polinomios de grado 1); por tanto, en comparación al *cero-order hold*, hay menos información de alta frecuencia en la señal de salida. Este hecho se puede comprobar analizando, a nivel frecuencial, las señales implicadas en la interpolación de orden 1.

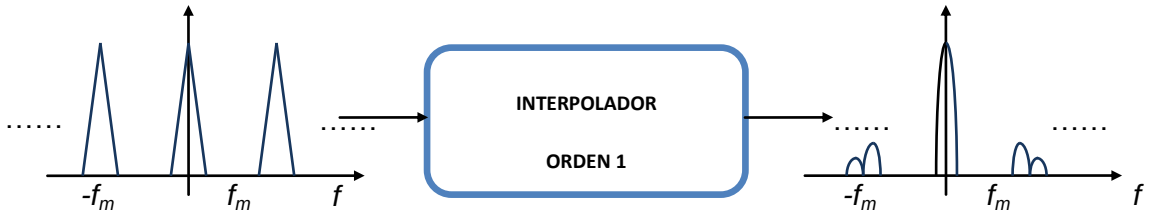


Figura 52. Respuesta frecuencial del interpolador de orden 1

En la figura anterior se puede ver cómo, además de añadir información de alta frecuencia, el *one-order hold* introduce una distorsión del espectro de la señal de salida, y que ésta es superior a la introducida por el interpolador de orden cero (esto se debe a que la *sinc* está elevada al cuadrado, y la atenuación que introduce es, por tanto, mayor).

Por este motivo también habrá que añadir unos bloques posteriores al interpolador para eliminar la información de alta frecuencia y de compensar la distorsión introducida por éste.

3. ASIC en el ámbito de la radio

En este apartado describiremos uno de los principales exponentes de los ASIC (*Application Specific Integrated Circuit*) en el ámbito de la radio: los mezcladores digitales. Concretamente hay ha de dos tipos:

- DDC (*Digital Down Converter*) / RSP (*Receive Signal Procesor*). Dispositivos que se utilicen en la etapa de recepción. Trasladan el espectro hacia frecuencias bajas.

- DUC (*Digital Up Converter*) / TSP (*Transmit Signal Procesor*). Se utilizan en emisión; por tanto a la salida la frecuencia es superior que en la entrada.

- Se utilizan en emisión; por tanto a la salida la frecuencia es superior que a la entrada.

Tanto el DDC como el DUC pertenecen, pues, a un tipo de dispositivos denominados ASIC (*Application Specific Integrated Circuit*). De entre los dispositivos digitales computerizados los ASIC son los menos programables, pero a la vez los más eficientes en términos de velocidad, consumo de potencia y área de silicio ocupada. Veremos durante el curso dos tipos más de dispositivos computerizados muy usados en radios digitales: los DSP y los PLD.

3.1. El DDC / RSP

3.1.1. Introducción

DDC o RSP son los dispositivos digitales que se utilizan en la cadena de recepción de un heterodino digital para seleccionar el canal de interés, filtrarlo y bajarlo típicamente a banda base en formato I/Q. Su diagrama funcional es el siguiente:

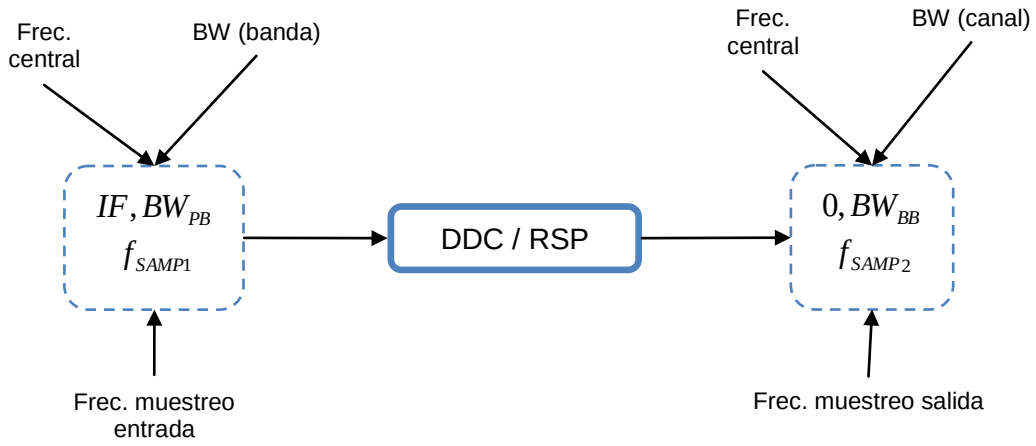


Figura 53. Diagrama funcional del DDC / RSP

En general en la entrada habrá una banda de frecuencias de ancho de banda BW_{banda} centrada en una frecuencia IF y muestreada a $f_{SAMP1} \geq 2 \cdot \left(IF + \frac{BW_{PB}}{2} \right)$

En general, nos interesará uno de los canales de la banda (de ancho de banda BW_{PB}). El DDC lo seleccionará, filtrará y bajará a banda base en formato I/Q. Además adaptará la frecuencia de muestreo de salida a las nuevas necesidades: $f_{SAMP2} \geq 2 \cdot BW_{BB}$. Para hacerlo aplicará una delmación (reducción de la frecuencia de muestreo) en factor M_{TOTAL}.

Ponemos un ejemplo: suponemos que en la entrada del DDC tenemos el *downlink* GSM en las condiciones siguientes:

- $BW_{PB} = 25\,Mhz$ (banda) compuesta por 124 canales de 200 KHz de ancho de banda.

- Frecuencia central: $IF = 15\,Mhz$ (suponed que hemos usado previamente un mezclador analógico)

- Frecuencia de muestreo: $f_{SAMP1} \geq 2 \cdot \left(IF + \frac{BW_{PB}}{2} \right) \rightarrow f_{SAMP1} = 60\,MSPS$

El DDC seleccionará un canal cualquiera de los 124 y lo bajará a banda base. Por tanto las condiciones en su salida serán:

- Ancho de banda: el de un canal en banda base $BW_{BB} = 100\,Khz$

- Frecuencia central: 0

- Frecuencia de muestreo: $f_{SAMP2} \geq 2 \cdot BW_{BB} \rightarrow f_{SAMP2} = 250\,KSPS$

En estas condiciones se observa que el DDC tendrá que aplicar una delmación en un factor de $M_{TOTAL} = \frac{f_{SAMP1}}{f_{SAMP2}} = \frac{60\,MSPS}{250\,KSPS} = 240$

El resto del estudio del DDC / RSP lo haremos basándonos en el dispositivo de *Analog Devices* AD6620.

3.1.2. Diagrama de bloques

El diagrama de bloques del AD6620 es el siguiente:

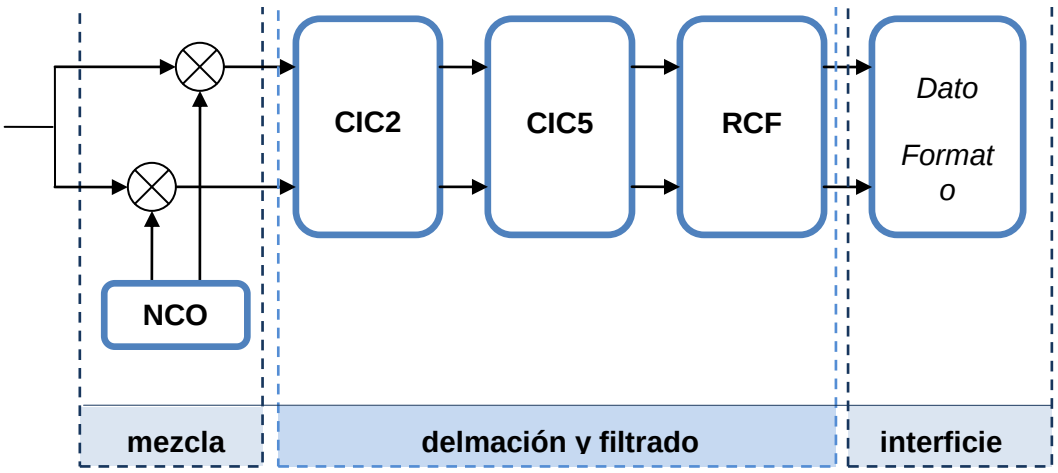


Figura 54. Diagrama de bloques del DDC / RSP

Diferenciemos tres partes:

- Proceso de mezcla o traslado en frecuencia. El NCO (*Numerically Controlled Oscillator*) es un oscilador digital que genera dos tonos desfasados 90º a la frecuencia central del canal que nos interesa. Los dos multiplicadores calculan las componentes I/Q.

- Delmación y filtrado. La delmación total se tiene que distribuir entre las tres etapas CIC2, CIC5 y RCF, de manera que $M_{TOTAL} = M_{CIC2} \cdot M_{CIC5} \cdot M_{RCF}$. Cada una de estas etapas tiene el diagrama de bloques siguiente:

Figura 55. Proceso de delmación

El motivo por el cual la delmación se distribuye en tres fases en vez de hacerlo todo en una es el siguiente. La frecuencia de muestreo en la entrada del DDC puede ser muy elevada (alrededor de 60 MSPS). A estas velocidades tecnológicamente no es posible implementar los filtros asociados a delmaciones muy grandes (y si fuera posible, su ejecución consumiría mucha energía), pero sí que es posible implementar filtros sencillos (asociados a delmaciones pequeñas).

Sucesivamente las etapas funcionan a frecuencias de muestreo más bajas y los filtros pueden ser más complejos.

Las características generales de cada etapa son:

- La etapa CIC2 (*Cascadable Intregrated Comb*, de orden 2) permite una delmación máxima de 16. El filtro es de coeficientes constantes.

- La etapa CIC5 (*Cascadable Intregrated Comb*, de orden 5) permite delmación hasta 32. El filtro es también de coeficientes constantes pero de un orden y prestaciones superiores.

- La etapa RCF (*Ram Coefficient Filter*) permite una reducción de la frecuencia de muestreo de hasta 32 e incluye un filtro con coeficientes totalmente programables.

Efectivamente, etapas sucesivas tienen mejores prestaciones ya que trabajan a frecuencias de muestreo inferiores.

- Finalmente la interficie adapta la salida a formatos de puerto serie o puerto paralelo.

3.1.3. Evolución de las frecuencias de muestreo

En la figura siguiente vemos como a partir de la primera etapa de delmación la frecuencia de muestreo se reduce sucesivamente:

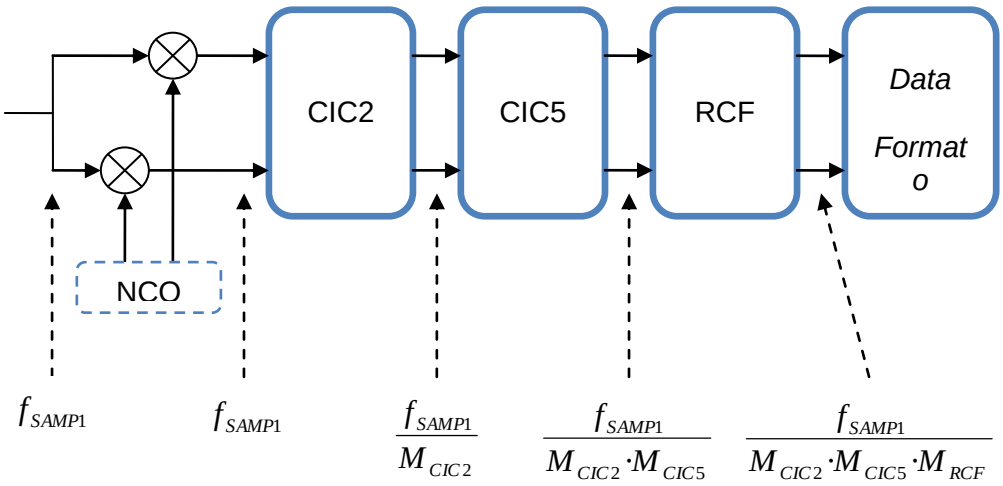


Figura 56. Evolución de las frecuencias de muestreo en el DDC / RSP

Un primer criterio a la hora de distribuir la delmación total entre las tres fases es de asignar tanta como sea posible a las primeras etapas ya que de esta manera las etapas posteriores trabajan a una frecuencia inferior y, por tanto, se optimiza el consumo de energía.

3.1.4. Evolución de los espectros

En la figura siguiente vemos un ejemplo de la evolución de los espectros en el DDC.
El mezclador traslada el canal de interés a banda base y las etapas delmadoras reducen la frecuencia de muestreo hasta ajustarla aproximadamente al doble del ancho de banda (banda base) del canal.

Tanto por lo que respecta al NCO como a cada etapa delmadora sólo hemos incluido un espectro, aunque como sabemos hay dos: uno por cada componente en cuadratura.

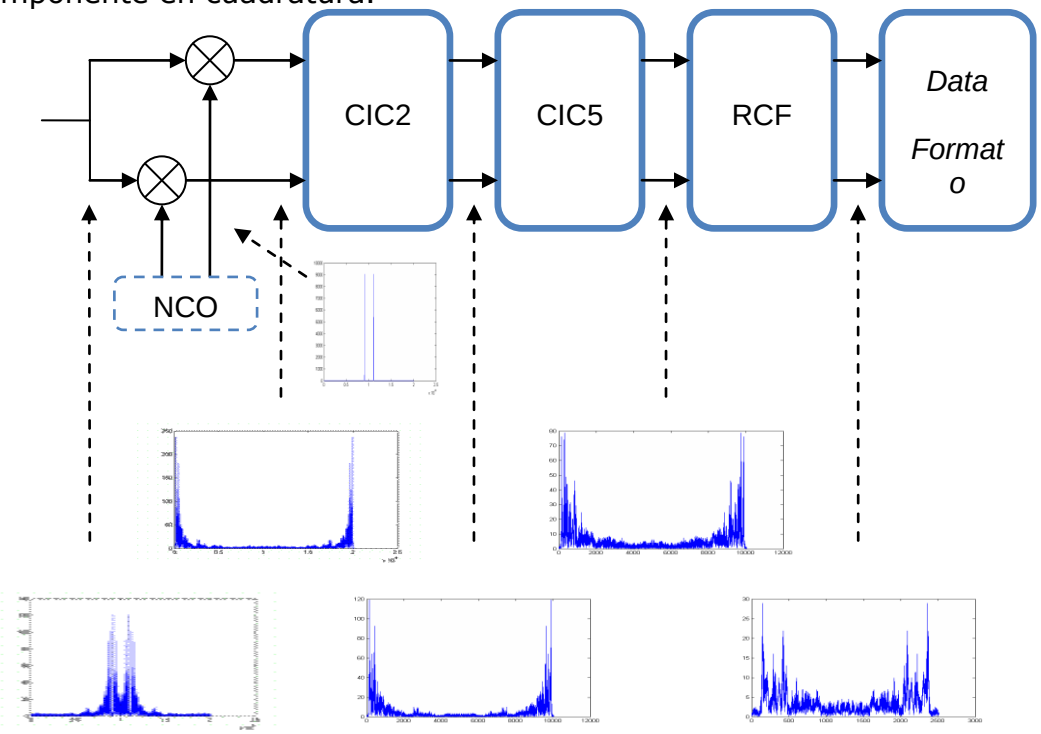


Figura 57. Evolución de los espectros en el DDC / RSP

3.1.5. Metodología de diseño

El funcionamiento del DDC viene determinado por los parámetros siguientes:

- Frecuencia central del oscilador local: f_{OL}
- Factor de delmación total y de cada etapa: $M_{TOTAL} = M_{CIC2} \cdot M_{CIC5} \cdot M_{RCF}$

La frecuencia central del oscilador local se fija igual que la frecuencia central del canal de interés. De esta manera conseguiremos trasladar el canal a la banda base.

El factor de delmación total se calcula como la relación entre las frecuencias de muestreo a la entrada y la salida del DDC:

$$M_{TOTAL} = \frac{f_{SAMP1}}{f_{SAMP2}}$$

Ecuación 19. Delmación total en un DDC

Con el fin de distribuir esta delmación entre les tres etapas hay que tener en cuenta los criterios siguientes:

- Intentaremos delmar al máximo en las primeras etapas para reducir el consumo total del dispositivo.

- Para las dos primeras etapas (CIC2, CIC5), existe un compromiso entre el ancho de banda del canal, el factor de delmación de la etapa y la pureza espectral requerida (*Alias Rejection*)

A continuación explicaremos este segundo criterio. Como ya sabemos, antes de delmar en un factor *M* conviene filtrar el espectro con un filtro paso bajo digital con frecuencia angular de corte $\frac{\pi}{M}$. Este filtro elimina todas las componentes frecuenciales que la delmación produce. A frecuencias de muestreo elevadas la tecnología no permite implementar estos filtros paso bajo ideales. En vez de esto se utiliza filtros *comb* que eliminan las fuentes de *aliasing* de una manera selectiva: el filtro sitúa ceros en aquellas frecuencias tales que al delmar producen *aliasing* en la banda base.

Veámoslo en un ejemplo. El filtro que aplica la etapa delmadora CIC2 en caso de un factor de delmación $M_{CIC2} = 4$ es:

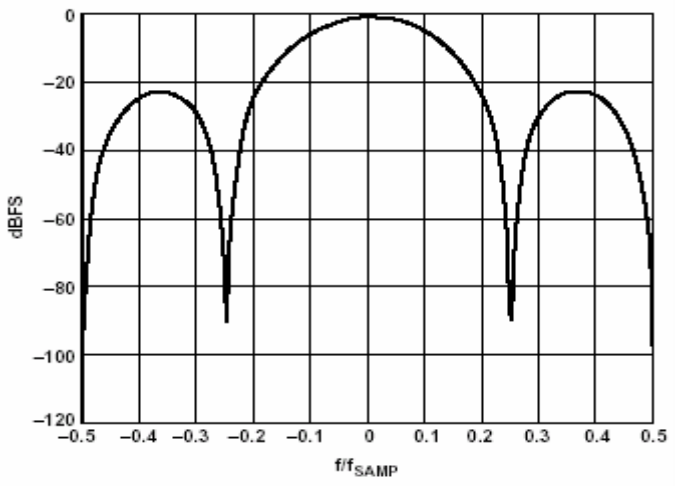


Figura 58. Respuesta en frecuencia de un filtro *comb.* (Font. *Datasheet* del AD6620 de Analog Devices)

Este filtro no tiene la respuesta esperada (paso bajo ideal con frecuencia de corte a $\frac{\pi}{4} \equiv \frac{0.5}{4}$). Por contra, el filtro presenta ceros en las frecuencias $f_1 = \frac{0.5}{2}, f_2 = \frac{0.5}{1}$.

¿Por qué? La solución a la pregunta pasa por remarcar que el filtro no pretende eliminar todo el *aliasing* producido por la delmación sino sólo aquél que va a parar a la banda base. La razón de hacerlo así es que sólo en la banda base es donde hay información de nuestro interés: el *aliasing* en otras frecuencias no nos importa. Desde esta nueva perspectiva, el análisis es sencillo.

Fijémonos en que al delmar 4 el espectro se expande en un factor 4 y, por tanto,

$$f \to M \cdot f \Rightarrow f_1 = \frac{0.5}{2} \to 4 \cdot \frac{0.5}{2} = 1 \to Banda\ Base$$
$$f \to M \cdot f \Rightarrow f_1 = \frac{0.5}{1} \to 4 \cdot \frac{0.5}{1} = 2 \to Banda\ Base$$

Ecuación 20. Proceso de delmación

Recuerda que la idea del rechazo selectivo de *aliasing* ya apareció en el diseño de los filtros *anti-aliasing*.

Un detalle más: la señal de interés tiene un ancho de banda diferente al de cero y, por tanto, no hay suficiente con eliminar estas frecuencias sino que además hay que eliminar los entornos de estas frecuencias. Con tal de cuantificar este parámetro, se define *Alias Rejection* como el nivel de rechazo que el filtro introduce en el entorno de las frecuencias de los ceros.

El *alias rejection* se suele igualar al margen dinámico del sistema. La idea es que no es necesario eliminar el *aliasing*; hay suficiente con que tenga un nivel inferior al margen dinámico (señal útil).

En les dos primeras etapas (CIC2 y CIC5) la relación entre los parámetros *alias rejection*, *BW* y factor de delmación (*M*) viene dada mediante tablas:

Table III. SSB CIC2 Alias Rejection Table (f_{SAMP} = 1)
Bandwidth Shown in Percentage of f_{SAMP}

M_{CIC2}	–50 dB	–60 dB	–70 dB	–80 dB	–90 dB	–100 dB
2	1.79	1.007	0.566	0.318	0.179	0.101
3	1.508	0.858	0.486	0.274	0.155	0.087
4	1.217	0.696	0.395	0.223	0.126	0.071
5	1.006	0.577	0.328	0.186	0.105	0.059
6	0.853	0.49	0.279	0.158	0.089	0.05
7	0.739	0.425	0.242	0.137	0.077	0.044
8	0.651	0.374	0.213	0.121	0.068	0.038
9	0.581	0.334	0.19	0.108	0.061	0.034
10	0.525	0.302	0.172	0.097	0.055	0.031
11	0.478	0.275	0.157	0.089	0.05	0.028
12	0.439	0.253	0.144	0.082	0.046	0.026
13	0.406	0.234	0.133	0.075	0.043	0.024
14	0.378	0.217	0.124	0.07	0.04	0.022
15	0.353	0.203	0.116	0.066	0.037	0.021
16	0.331	0.19	0.109	0.061	0.035	0.02

Figura 59. Tabla *Alias Rejection* por el CIC2 (Fuente. *Datasheet* de la AD6620 de Analog Devices)

En columnas, el *Alias Rejection* deseado (-50 dB, -60 dB, etc.)
En filas, factores de delmación (2, 3, 4, 5, etc.)

Cada valor de la tabla indica el porcentaje de ancho de banda normalizado a la frecuencia de muestreo que tendrá el rechazo de *aliasing* considerado por el valor de delmación escogido.

Por ejemplo, si el *Alias Rejection* es -70 dB, el factor de delmación 6 y la frecuencia de muestreo 60 MSPS, entonces el 0.279 % del ancho de banda tendrá la purezas espectral deseada; es decir,

$$0.279 = 100 \cdot \frac{BW}{f_{SAMP}} \rightarrow BW = 167.400\,Khz$$

A la fórmula anterior, f_{SAMP} es la frecuencia de muestreo a la entrada de la etapa y *BW* habría de ser superior o igual al ancho de banda (paso banda) del canal de interés para asegurar un rechazo de *aliasing* suficiente a todo el ancho de banda.

Existen tablas diferentes para la etapa CIC2 y CIC5. La segunda es menos restrictiva ya que el filtro es de mayor orden.

Resumiendo: fijados el ancho de banda del canal, el *Alias Rejection* y la frecuencia de muestreo por cada etapa, distribuiremos la delmación maximizándola a las primeras etapas y respetando el margen dinámico.

3.2. El DUC / TSP

El DUC / TSP es el dispositivo homólogo del explicado en el apartado anterior que se utiliza en la cadena de emisión. Su diagrama funcional es el siguiente:

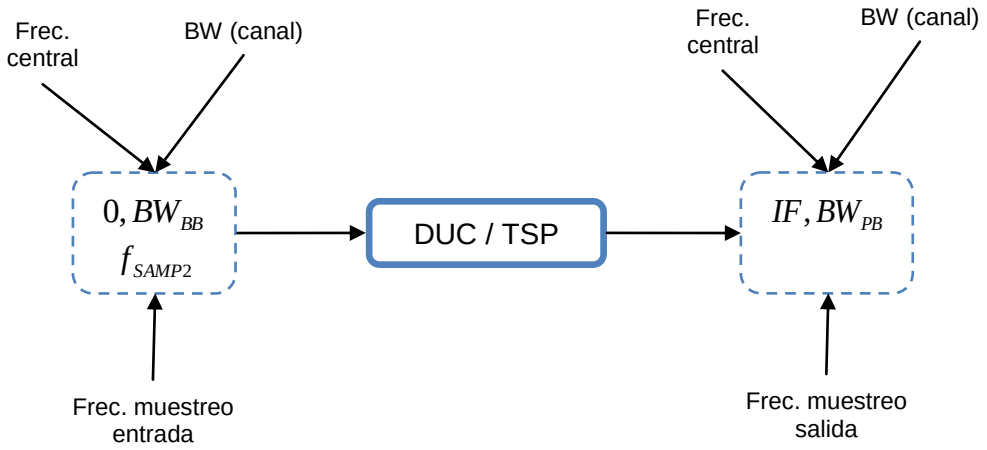


Figura 60. Diagrama funcional del DUC / TSP

Si lo comparamos con el diagrama del DDC observamos que es el mismo pero al revés. Ahora algún dispositivo computerizado habrá generado un canal en la banda base (admite formato I/Q) y el DUC / TSP lo trasladará a una frecuencia intermedia (o quizás la definitiva). La diferencia más importante entre el DUC y el DDC es que mientras el DDC ha de reducir la frecuencia de muestreo (y por lo tanto delmar), el DUC tiene que aumentarla (y por tanto interpolar).

El estudio del DUC y el del DDC es muy parecido y lo haremos remarcando las diferencias entre ambos. En este caso nos basaremos en el AD6622 de *Analog Devices*. El diagrama de bloques interno será:

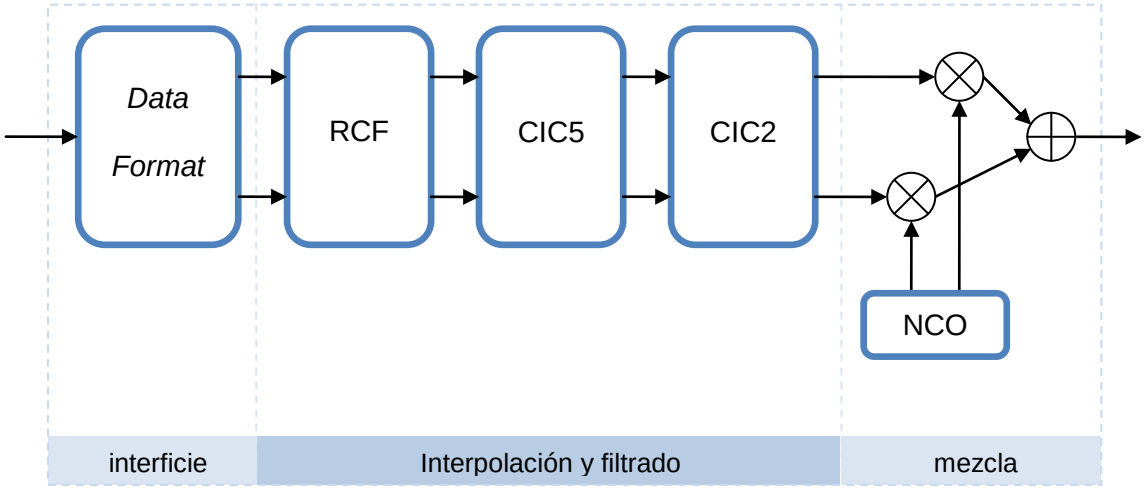


Figura 61. Diagrama de bloques del DUC / TSP

Las fases del proceso son idénticas pero en orden inverso. En conjunto es lógico:

- Primero hay que interpolar para aumentar la frecuencia de muestreo y poder hacer el traslado en frecuencia sin producir *aliasing*.

- Las etapas de interpolación más complejas (RCF) trabajan a frecuencias bajas (cerca de la entrada), y les más sencillas (CIC2) a frecuencias elevadas (cerca de la salida).

En este caso el dispositivo AD6622 incluye 4 canales independientes, de manera que el diagrama de bloques total contiene los bloques nombrados en figura anterior duplicados cuatro veces. Esto permite generar múltiples canales simultáneamente.

La metodología de diseño es idéntica al caso anterior. Ahora la estructura de cada bloque interpolador es:

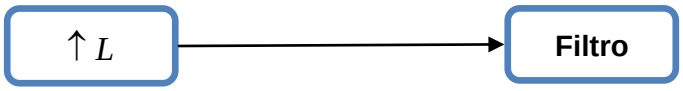


Figura 62. Proceso de interpolación

El filtro ideal es un paso bajo pero por los mismos motivos que al DDC no es aplicable tecnológicamente. La estructura de filtrado utilizada será tipo *comb*, con el consecuente problema del *alias rejection* (que ahora recibe el nombre de *image rejection*). Ahora situaremos los ceros del filtro coincidiendo con las frecuencias centrales de las imágenes de interpolación.

Veámoslo con un ejemplo en la figura siguiente:

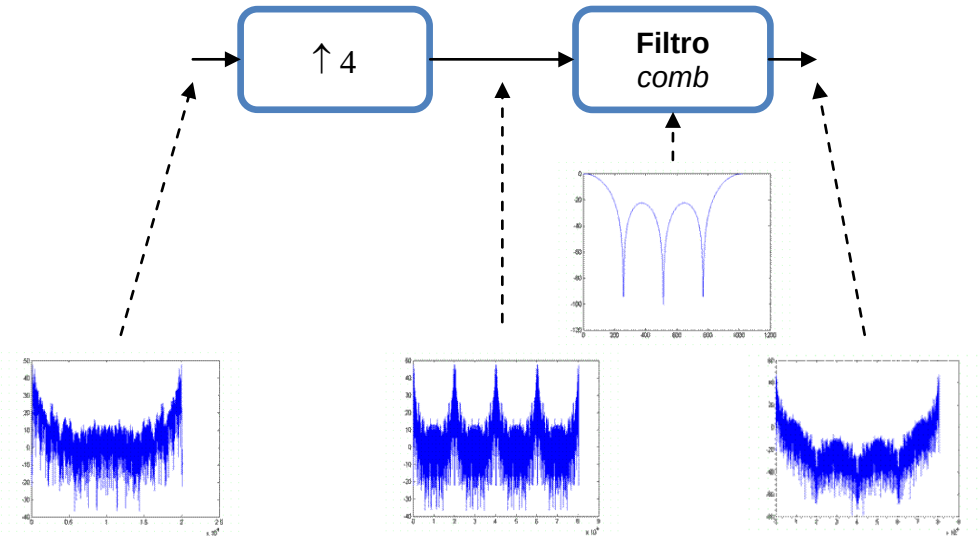


Figura 63. Espectros durante el proceso de interpolación

Con tal de distribuir la interpolación entre las tres etapas seguiremos los dos criterios conseguidos pero adaptados:

- Máxima interpolación a las etapas finales con el fin de minimizar el consumo del dispositivo.

- Distribución de la interpolación entre les tres etapas respectando el compromiso entre *image rejection*, factor de interpolación y *BW*. El fabricante suministra tablas que relacionan estos parámetros por la etapa CIC2 y CIC5.

4. Procesadores DSP

4.1. Bases del DSP

4.1.1. Introducción

Los orígenes del gran desarrollo que ha experimentado el procesado digital de la señal hay que buscarlos a mediados del siglo XX. En aquellos años los diseñadores de electrónica analógica vieron la necesidad de simular sus sistemas antes de construir los prototipos. El objetivo principal era abaratar costes de producción. Como las simulaciones eran muy lentas, se decidió atacar el problema de velocidad desde dos vertientes.
En una primera fase se hizo énfasis en el estudio y optimización de los algoritmos. Un resultado de estos esfuerzos fue la presentación de la FFT en el año 1965.

En una segunda fase se vio la necesidad de explorar la arquitectura sobre la que se ejecutaban las aplicaciones, adaptándolas a las características comunes que los algoritmos de procesado digital tenían. Así es como nación el concepto de DSP (siglas de *Digital Signal Procesor*).

Un DSP, a grandes rasgos, no es más que un procesador de propósito específico con una arquitectura optimizada para realizar cálculos matemáticos y trabajar en entornos con una carga de entrada salida muy intensa. Si concretamos un poco más sería necesario indicar que un DSP no realiza de manera óptima cualquier operación matemática. El diseño de estos procesadores se ha realizado pensando en las operaciones que se realizan más frecuentemente en las aplicaciones de procesado digital de la señal: estructuras suma y acumulación (filtros FIR, IIR, convulsiones, correlaciones, etc.) y análisis frecuencial (aderezamiento *bit-reversal*, acumulaciones, sumas, acceso intenso a memoria, etc.)

Estas dos vertientes de las aplicaciones DSP (optimización de algoritmos y arquitecturas específicas) son una constante en el ámbito. En cualquier situación para obtener las máximas prestaciones habrá que considerar ambos aspectos.

4.1.2. Análisis computacional de algoritmos

En este apartado realizaremos un pequeño estudio teórico donde introduciremos los conceptos de coste computacional y coste de ejecución de los algoritmos. A partir de estas definiciones explicaremos qué es un DSP. Durante todo el apartado nos basaremos en una aplicación ejemplo:

un banco de *M* filtros de *N* coeficientes cada uno, funcionando a f_m muestras por segundo.

Si pensamos en las instrucciones máquina (de un procesador cualquiera) que ejecutan una aplicación, observaremos que podemos clasificarlas en dos categorías:

- *Instrucciones propias*. Son aquellas instrucciones que realizan las operaciones matemáticas que aparecen en la definición del algoritmo.

  En caso de un filtro FIR, por ejemplo:

  $$y[n] = a_0 \cdot x[n] + a_1 \cdot x[n-1] + a_2 \cdot x[n-2] + \ldots + a_{N-1} \cdot x[n-(N-1)]$$

  Número de operaciones por filtro:

  (N-1) sumas y *N* multiplicaciones

  Si hay M bancos de filtros:

  M·(N-1) y *M·N* multiplicaciones

  Este recuento de operaciones matemáticas (ejecutadas por las instrucciones propias) recibe el nombre de coste computacional del algoritmo. A menudo se utiliza como coste base para dimensionar el *hardware* (aunque como veremos es una aproximación frecuentemente errónea).

  Así pues, las instrucciones propias son aquellas instrucciones máquina que ejecutan las operaciones matemáticas que forman el coste computacional del algoritmo.

- *Instrucciones parásitas*. Todo el resto de instrucciones que, por ejemplo, realizan las tareas de:

  - Control de flujo (bucles, saltos, *...)*. Los bucles, por ejemplo, requieren de un índice que a cada iteración se incrementa y se compara para tomar la decisión de continuar iterando o salir del bucle.
  - Cálculo de direcciones de memoria. En direcciones indirectas indexadas a menudo hay que hacer alguna suma o resta para calcular la dirección final.
  - Movimientos registro – memoria causada para que el procesador no permita que los operandos origen o destino de las operaciones matemáticas sean a memoria.
  - Gestión de la entrada / salida.
  - Comunicación multiprocesador.

- Etc.

En general, las instrucciones parásitos se pueden clasificar a la vez en:

- Instrucciones parásitos *independientes* del algoritmo o que no aumentan con la complejidad de éste. Por ejemplo:

  - Entrada y salida de la RSI
  - Inicialización de variables
  - Etc.

El coste de ejecución lo contabilizaremos con el término K_i ciclos de reloj por periodo de muestreo.

- Instrucciones parásitas *dependientes* de la complejidad del algoritmo (en el caso del banco de M filtros FIR de N coeficientes cada uno la complejidad del algoritmo es del orden de M y N). Por ejemplo:

  - Accesos a memoria
  - Gestión de bucles,
  - Gestión de *buffers* circulares,
  - Etc.

Este coste lo simbolizaremos con el término $M \cdot N \cdot K_d$ ciclos de reloj por periodo de muestreo.

En general, el coste de ejecución de las instrucciones parásitos depende de tres factores:

- Eficiencia del código. Está en las manos del programador generar un código que minimice este tipo de instrucciones optimizando el algoritmo.

- Eficiencia del compilador. En la mayoría de los casos las aplicaciones se generan en lenguajes de alto nivel (mejor portabilidad, menor tiempo de desarrollo, etc.). En estos casos la eficiencia del compilador será determinante ya que es él quien interpreta el código genera las instrucciones máquina que ejecuta el procesador.

- Eficiencia del procesador. Según el juego de instrucciones de que disponga el procesador, algunas de las tareas parásitos antes descritas generarán instrucciones máquina y otras no. Aquí es donde los procesadores DSP adquirieren su máxima expresión, tal y como veremos más adelante.

Una vez vista la clasificación podemos cuantificar el coste de ejecución por periodo de muestreo como:

$$propias + parásitos$$

En el ejemplo del banco de *M* filtros FIR de *N* coeficientes cada uno:

$$M\cdot(N\text{-}1)\ sumas + M\cdot N\ multiplicaciones + M\cdot N\cdot K_d + K_i$$

Este recuento recibe el nombre de coste de ejecución del algoritmo (por muestra). Corresponde al número de instrucciones totales (propias más parásitos) que se necesitan para ejecutarlo. El coste de ejecución es el parámetro que realmente se necesitaría para dimensionar el hardware.

Esta misma magnitud, contabilizada durante un segundo:

$$(M\cdot(N\text{-}1)\ sumas + M\cdot N\ multiplicaciones + M\cdot N\cdot K_d + K_i)\cdot f_m$$

Si suponemos, para simplificar, que todas las instrucciones tienen el mismo coste de ejecución (es decir, todas las instrucciones se ejecutan con el mismo número de ciclos de reloj):

$$\left(M\cdot N\cdot\left(2+K_d\right)+K_i\right)\cdot f_m$$

Las conclusiones de este análisis son las siguientes:

- El coste de las instrucciones parásitas dependientes del algoritmo tiene un peso importantísimo, del mismo orden o superior al de las instrucciones propias.

- Dimensionar a priori un *hardware* a partir de las operaciones propias es muy poco fiable.

- El coste de les instrucciones parásitos no está sólo en manos del programador: el compilador y el procesador tienen también un papel muy importante.

A partir de estas definiciones, ¿cómo podemos definir qué es un DSP? Visto el análisis expuesta hasta ahora, podemos afirmar que los DSPs intentan disminuir el coste de ejecución del programa desde dos puntos de vista:

- Minimizando (en algunos casos eliminando) la carga temporal de ejecución de las tareas parásitas. Los DSP incluyen *hardware* específico por resolver estas operaciones sin penalización de ciclos de reloj. Por ejemplo:

  - Accesos óptimos a *buffers* circulares
  - Gestión óptima de bucles
  - Instrucciones con ejecución condicional
  - Gestión óptima de la entrada / salida

En general, las instrucciones parásitos que se trataran de manera especial serán aquellas que aparecen frecuentemente en algoritmos de procesado digital del señal.

- Maximizar la relación ($^{operacions\ matemàtiques}/_{instrucció\ executada}$).

No sólo intentarán eliminar el coste de ejecución de tareas parásitos sino que, en algunos casos, permitirán agrupar diversas operaciones matemáticas en un ciclo de reloj. Un ejemplo de esto es la ejecución simultánea de una acumulación y un producto.

Estos dos factores permitirán diseñar el *hardware* a partir de las operaciones propias. Esta aproximación, no obstante, sólo será posible para aplicaciones que utilicen al máximo la funcionalidad y el juego de instrucciones de los algoritmos por los cuales ha sido diseñado el DSP. El ejemplo que hemos puesto en este apartado está dentro de este caso.

Si consideramos su implementación sobre un procesador DSP encontraremos un coste de ejecución bastante aproximado en la expresión:

$$(M \cdot N + K_i) \cdot f_m$$

4.1.3. Algunas características concretas comunes a los DSPs

En este apartado veremos en dos casos reales (TMS320C3x de *Texas Instruments* y ADSP-2106x de A*nalog Devices*) como quedan reflejados los conceptos vistos en el apartado anterior.

El análisis se hará comparativamente en cuatro aspectos: código en C, conversión a un código máquina típico y conversiones a los códigos máquina de los DSP en cuestión. El objetivo del estudio es ver reflejado en ahorro de coste (expresado en número de instrucciones) la implementación de diversas tareas que suelen llevar a cabo los algoritmos de procesado digital de la señal a tiempo real.

❖ Instrucciones condicionales

Con este término nos referimos a la ejecución condicional de una instrucción según el resultado de una comparación.

	Código	Coste
C	*if (a==b)* *b=100;* *(...)*	-
ensamblador típico	*cmp R0, R1* *jnz _continua* *ld 100, R1* *_continua:* *(...)*	**3 + penalización salto**
TMS320C3x	*CMPF R0, R1* *LDFz 0100h, R1*	**2**
ADSP-2106x	*COMP (R0, R1)* *IF EQ R1=0100h*	**2**

Figura 64. Comparativa. Instrucciones condicionales

Uno de los principales inconvenientes de la ejecución de instrucciones condicionales es el cambio en el flujo de ejecución: los saltos de programa en procesadores segmentados suelen provocar ciclos de *corte* que en esta comparativa se han expresado como *penalización de salto*.

En el TMS320C3x casi todas las instrucciones se pueden ejecutar condicionalmente introduciendo un sufijo indicativo del resultado de la condición. En el ejemplo, el sufijo es *z* (cierto si el resultado de la comparación anterior es cero) y está asociada a la instrucción *LDF* (movimiento memoria – registro de un operando en coma flotante).

Por lo que respecta al ADSP-2106x, su código máquina es muy intuitivo. En este caso la ejecución condicional se realiza introduciendo la directiva *'IF flag instrucción'.*

En conjunto nos ahorramos una instrucción con la penalización correspondiente.

❖ Acceso óptimo a memoria y multiplicación y acumulación en un ciclo de reloj

	Código	Coste
C	*c=c+a(n++)*b(n++)*	-
Ensamblador típico	*load *IR1, R1* *add IR1, 1* *load *IR2, R2* *add IR2, 1* *mult R1, R2, R3* *add R0, R3*	**6**
TMS320C3x	*mpyf3 *ar0++(1), *ar1++(1), r0 ||* *addf3 r0, r2, r2*	**1**
ADSP-2106x	*f12=f0·f4, f8=f8+f12, f0=dm(i0, m0),* *f4=pm(i8, m8)*	**1**

Figura 65. Comparativa. Acceso óptimo a memoria. Multiplicación y acumulación.

La estructura estudiada en este apartado tiene mucha importancia ya que aparece en muchas aplicaciones de procesado digital de la señal: filtros, convulsiones, etc. Fijémonos en que esta instrucción requiere:

- Instrucciones propias: una suma y un producto
- Instrucciones parásitos: dos accesos a memoria y dos sumas asociadas al cálculo de las direcciones.

En el caso de los procesadores DSP toda la estructura se ejecuta con una única instrucción:

- Suma y producto paralelizados
- Accesos a memoria simultáneos
- Cálculo de las direcciones de memoria en unidades computacionales específicas

El indicador de paralelismo en el TMS320C3x es ||. Toda dirección puede ser *pre* / *post* incrementada / decrecida gracias a dos unidades computacionales y unos registros (AR0 – AR7: *auxiliary register*) asociados al cálculo de direcciones.
En el ADSP-2106x el indicativo de paralelismo es la coma (,).

La directiva *dm* permite el acceso a memoria de datos mientras que *pm* a la memoria de programa (la memoria de programa puede contener tanto código como datos). Estos accesos utilizan también unos registros especiales que son los *b* (dirección base del *buffer*), *e* (índice), *m* (modificadores de dirección), *l* (medida del *buffer*).

❖ Tratamiento óptimo de bucles (*cero overhead*)

	Código	Coste
C	*for (i=0;i<100;++i)* *c=c+a(i)*b(i)*	
Ensamblador típico	*load 100, r0* *_bucle:* *load *IR1, R1 // Apartado anterior* *add IR1, 1* *load *IR2, R2* *add IR2, 1* *mult R1, R2, R3* *add R0, R3* *sub 1, r0 // Gestión índice* *cmp r0, 0* *jnz _bucle* *_continuación:*	**1+100·(9+penalización salto)**
TMS320 C3x	*ldi 0100, rc* *rpts rc* *mpyf3 *ar0++(1), *ar1++(1)%, r0 || addf3 r0, r2, r2*	**1+100**
ADSP-2106x	*lcntr=100* *do macs until lce* *_macs: f12=f0·f4, f8=f8+f12, f0=dm(i0, m0), f4=pm(i8, m8)*	**1+100**

Figura 66. Comparativa. Tratamiento óptimo de bucles

Los algoritmos de procesado digital de la señal suelen tener una estructura iterativa construida mediante bucles. Los bucles tienen asociados índices (en este caso: *i*) que en cada iteración se tienen que incrementar y comparar con el fin de decidir salir del bucle o continuar iterando.

La familia del TMS320C3x incluye un mecanismo compuesto por un registro específico (*rc: repeat count*) t un par de instrucciones (*rpts: repeat single, rptb: repeat bloquek*) que implementan el bucle para *hardware* sin penalización de tiempo.

El ADSP-2106x incluye la directiva '*DO _etiqueta UNTIL LCE'* que tiene la misma función que en el caso anterior. En este caso el registro asociado al bucle es *lcntr* (*loop counter*). El procesador permite anidar de forma automática hasta 4 bucles.

❖ Dirección óptima de buffers circulares

	Código	Coste
C	*buffer [i++]=element;* *if (i==MAX_BUFFER)* *i=0;*	-
Ensamblador típico	*store r1, *ir1* *add 1, ir1* *cmp ir1, MAX_BUFFER* *jnz _continua* *load INICI_BUFFER, ir1* *_continua:*	**5 + penalización de salto (por cada acceso)**
TMS320C3x	*ldi MAX_BUFFER, bk* *ldi INICI_BUFFER, iro* *(...)* *sti r1, *ir0++(1)%*	**2+1**
ADSP-2106x	*l0 = MAX_BUFFER* *b0 = INICI_BUFFER* *(...)* *DM (i0, 1) = r1;*	**2+1**

Figura 67. Comparativa. Dirección de *buffers* circulares

Los *buffers* circulares (*FIFO: first in, first out*) son muy utilizados en aplicaciones de procesado digital de la señal a tiempo real que requieren memoria. Por ejemplo, en un filtro FIR de orden *N* hay que mantener una

memoria de les últimas *N* entradas; a cada ciclo de reloj hay que hacer sitio para la nueva entrada eliminando la más antigua.

La implementación de este tipo de *buffers* requiere de un puntero que recorra el vector de manera que a cada acceso haya que comparar con la medida del *buffer* y, si es necesario, volver a inicializarlo en la primera posición. Como vemos en la tabla, estas operaciones requieren de diversas instrucciones parásito.

En el TMS320C3x el acceso a *buffer* circular no tiene penalización de tiempo. Se indica mediante el sufijo *%*, previa inicialización del registro *bk* (*bloquek size*) a la medida del vector.

4.2. Implementación de filtros. Introducción

Un filtro es un sistema que permite cambiar de forma arbitraria y selectivamente en frecuencia, la amplitud y la fase de una señal. Los filtros se utilizan con objetivos muy diversos:

- Separar las componentes de una señal. Por ejemplo, en un receptor FSK para diferenciar las frecuencias y extraer la información
- Mejorar la calidad de una señal eliminando ruido o bien enfatizando / atenuando algunas componentes.
- Etc.

Los filtros se pueden implementar de forma analógica, mediante condensadores, resistencias, bobinas y elementos activos o bien de forma digital ejecutándolos sobre algún dispositivo programable.

Los filtros analógicos tienen diversos inconvenientes:

- Cambios con el tiempo, la temperatura, la humedad, etc.
- Dificultad para ajustarlos
- Dificultad para implementar filtros de orden grande
- Imposibilidad para implementar filtros adaptativos
- Etc.

Todos estos inconvenientes dejan de existir si los filtros se implementan en el dominio digital. Estos, no obstante, presentan también algunas limitaciones:

- Margen de frecuencias de funcionamiento limitado por el teorema del muestreo
- Margen dinámico limitado por el número de bits de los conversores y la aritmética

A pesar de todos estos inconvenientes la implementación digital suele ser la preferida en la mayoría de aplicaciones, siempre y cuando no requieran un ancho de banda o margen dinámico excesivos.

4.2.1. Sobre los filtros digitales

La respuesta frecuencial de un filtro digital, expresado en el dominio de la transformada *Z* viene dada por:

$$\frac{y(z)}{x(z)} = \frac{\sum_{n=0}^{N-1} d_n \cdot z^{-n}}{\sum_{n=0}^{M-1} c_n \cdot z^{-n}}$$

Ecuación 21. Filtro digital. Respuesta frecuencial

donde,

$x(z)$ es la entrada del filtro

$y(z)$ es la salida del filtro

c_n, d_n son los coeficientes del filtro

El denominador de la respuesta frecuencial es un polinomio en *z* de coeficientes c_n. Los coeficientes c_n determinan, pues, los puntos en que la respuesta frecuencial se hace infinita. Estos puntos reciben el nombre de *pulso* de la respuesta frecuencial. Así mismo, d_n fijan la forma del numerador y determinan los puntos en que ésta se hace cero: los ceros de la respuesta frecuencial.
Existen diversos métodos para diseñar los coeficientes del filtro para que tengan la respuesta frecuencial deseada: colocación de pulso y ceros, transformada bilineal, etc.
Hay dos tipos de filtros digitales:

- Filtros IIR (*Infinite Impulse Response*). Son aquellos que tienen, como mínimo, dos de los coeficientes c_n diferentes de cero. No tienen fase lineal, pueden ser inestables pero como contrapartida suelen requerir de pocos coeficientes con el fin de seguir respuestas de amplitud abruptas.

- Filtros FIR (*Finite Impulse Response*). Son aquellos que tienen el denominador igual a la unidad. Son, pues, siempre estables ya que la su respuesta frecuencial no tiene pulso. Se pueden diseñar para que tengan fase lineal. Requerimiento normalmente de muchos coeficientes para cumplir unas determinadas especificaciones.

La respuesta temporal de un filtro se encuentra anti-transformante *Z* la respuesta frecuencial:

$$y(n) = \sum_{i=0}^{N-1} \overline{d_i} \cdot x(n-i) - \sum_{j=1}^{M-1} \overline{c_j} \cdot y(n-j)$$

Ecuación 22. Filtro digital. Respuesta temporal

donde $\overline{d_i}, \overline{c_j}$ son los coeficientes del filtro normalizados respecto c_0.

La expresión anterior corresponde a un filtro IIR: fijémonos en que la salida en el instante actual depende de la entrada en el instante actual ($x(n)$), la entrada a los *N-1* instantes anteriores ($x(n-i)$) y las salidas en los instantes *M-1* anteriores ($y(n-j)$).

En el caso de un filtro FIR la expresión anterior se simplifica a:

$$y(n) = \sum_{i=0}^{N-1} d_i \cdot x(n-i)$$

Ecuación 23. Filtro FIR. Respuesta temporal

Fijémonos en que, como la salida sólo depende de las entradas, si éstas son finitas la salida será también siempre finita y el filtro siempre estable.

En los apartados siguientes nos entretendremos a estudiar la implementación de los filtros FIR. Veremos diferentes códigos aplicables a diferentes DSP y contabilizaremos el coste computacional en cada caso.

4.2.2. Implementación de filtros FIR

La implementación de los filtros FIR tiene gran importancia en el ámbito del procesado digital de la señal a tiempo real porque su misma estructura de cálculo aparece casi idéntica en otras operaciones como ahora la convulsión y la correlación.

Tomemos la expresión de la respuesta temporal de un filtro FIR (Ecuación 23), la reproducimos a continuación y analicémosla pensando en su implementación:

$$y(n) = \sum_{i=0}^{N-1} h(i) \cdot x(n-i) = h(0) \cdot x(n) + h(1) \cdot x(n-1) + \ldots + h(N-1) \cdot x(n-(N-1))$$

Ecuación 24. Filtro FIR. Respuesta temporal

Donde hemos cambiado el nombre de los coeficientes del filtro ya que la notación $h(i)$ se utiliza en muchos libros.

Fijémonos en primer lugar en las estructuras de memoria necesarias para ejecutar el filtro.

Tendremos que disponer de dos vectores de longitud *N*: uno para almacenar los coeficientes ($h(i)$) y otro para las entradas al filtro ($x(n-i)$).

El vector de coeficientes es estático: los coeficientes se calculen durante la fase de diseño y no varían durante la ejecución (variarían en el caso de un filtro adaptativo; caso que no consideraremos en este estudio).

El vector de entradas, en cambio, es dinámico. El valor $x(n)$ que aparece a la Ecuación 24 cambia a cada instante de muestreo, ya que $x(n)$ representa el dato actual. Veámoslo en la tabla siguiente en un ejemplo:

Instante muestreo	$x(n)$	$x(n-1)$	$x(n-2)$	$x(n-3)$	$x(n-4)$
T	5	3	7	2	1
T+1	0	5	3	7	2
T+2	1	0	5	3	7
T+3	2	1	0	5	3

Nuevo valor del instante T+1

Figura 68. Vector de entradas en un filtro FIR

El vector de entradas es, pues, un vector circular tipo FIFO (*First In, First Out*).

Las operaciones matemáticas que intervienen en el cálculo de la salida son productos y acumulaciones que podremos ejecutar dentro de una estructura interactiva de *N* repeticiones.

En la figura siguiente mostramos la implementación en ANSI C del filtro FIR:

```
0     #define N 10              // Orden del filtro

1     float h[N];               // Respuesta imp. colocada al revés
2     float x[N];               // Buffer de entrades
3     int index=0;              // índice del buffer circular


4     float filtro (float dada_in) {
5     int i;
6     float salida;

7     x[index++]=dada_in;       // Guardemos el último dato en el
buffer
8     if (index==N)             // Buffer circular ...
9        index=0;

10    salida=0
11    for (i=0;i<N;++i) {       // Calculamos la salida
12            salida+=h[i]*x[index++];
13            if (index==N)
14                index=0;
15        }

16    return (salida);
```

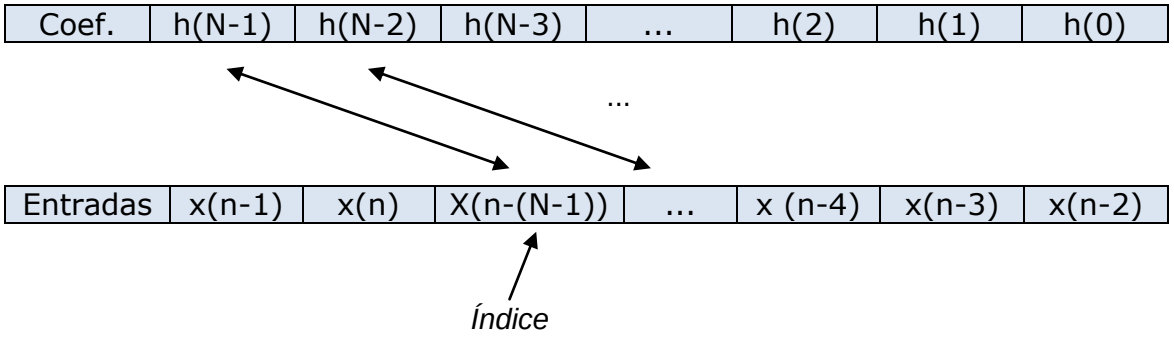
Figura 69. Implementación de un filtro FIR en ANSI C

La implementación del *buffer* circular (FIFO) requiere el mantenimiento de un *índice* que recorre el vector sin que salga de sus límites $[0, N-1]$.

Los coeficientes del filtro (respuesta impulsional) se colocan en orden inverso para optimizar el número de operaciones matemáticas a la hora de indexarlo.

Veámoslo en la figura siguiente:

Instante *T*:



Instante *T+1*:

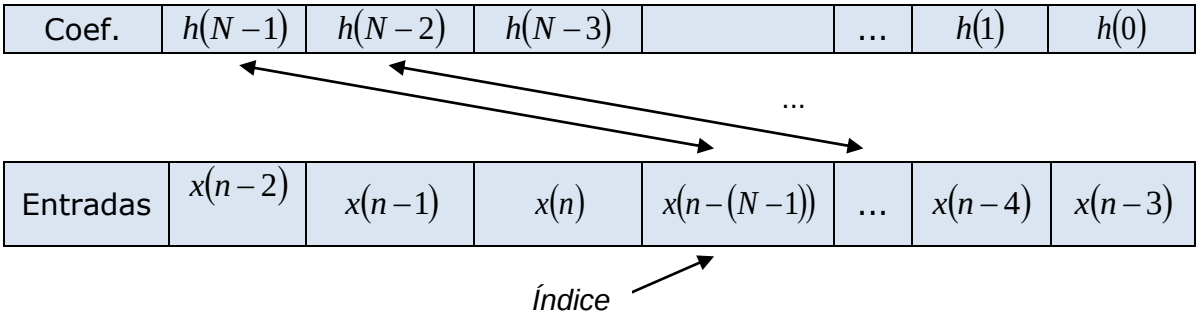


Figura 70. Ejecución de un FIR. Vectores de *coeficientes* y entradas

Fijémonos en que al almacenar el último dato obtenido en el vector de entradas siempre se sobrescribe el elemento más antiguo del vector e *índice* queda apuntando al nuevo elemento más antiguo.

4.2.3. Implementación sobre TMS320C3x

El código de la figura 69 puede ser directamente compilado con las herramientas de *Texas Instruments* para generar un ejecutable para el DSP TMS320C3x. En la figura 71 podemos observar el código ensamblador generado por el compilador. A partir de este código podremos hacer un recuento de instrucciones y calcular el coste de ejecución.

Sólo hemos reproducido las líneas de código relevantes. Fijémonos en que el bucle está compuesto entre las líneas 9 y 30. Algunas de las instrucciones están comentadas en el mismo código.

En total hay 18 instrucciones dentro de cada iteración, de las cuales sólo 2 corresponden a la suma y a la multiplicación correspondiente a la ejecución del filtro. El resto de instrucciones corresponden a accesos a memoria, gestión del *buffer* circular, gestión del índice de la iteración y otras instrucciones parásitos.
Si el filtro es de orden N (N coeficientes) y asumimos que todas las instrucciones tardan un ciclo de reloj, podemos aproximar el coste de ejecución del filtro para $18 \cdot N + K$ ciclos de reloj, donde K corresponde a la ejecución de las instrucciones previas y posteriores a la iteración.

El coste de ejecución obtenido no diferiría nada del que obtendríamos con cualquier otro procesador de características similares que no fuera DSP, ya que a la hora de generar el código en C de la figura 69Figura 69 no hemos tenido en cuenta que trabajábamos sobre DSP. En la figura 72 reproducimos la propuesta del fabricante para implementar un filtro FIR sobre el

TMS320C3x. Inicializaciones a parte, el cuerpo de la iteración son las líneas 30 y 31. A la línea 30 está el comando de control de iteración (*rpts*) y en la 31 se lleva a cabo en paralelo la multiplicación y la suma, así como los dos accesos a memoria (coeficientes y entradas), ambos implementados circularmente. El coste de ejecución de todo junto es, pues, $N+K$.

La clave de esta enorme mejora la encontramos en el aprovechamiento que se hace de todas las características del DSP que nos son útiles para reducir al máximo el número de ciclos de reloj.

```
*******************************************************
*    TMS320C30 C COMPILER     Version 4.70
*******************************************************

(...)             // Definición de constantes, ...

*******************************************************
* FUNCTION DEF : _filtro
*******************************************************
1     _filtro:
2           PUSH  FP
3           (...)       // 15 instrucciones manejando el stack
4     L1:
5           LDF   0.0,R2
6           STF   R2,*+FP(2)
7           LDI   0,R3
8           STI   R3,*+FP(1)
9     L2:                                       // Inicio del bucle
10          LDI   @_index,IR0
11          ADDI  1,IR0
12          STI   IR0,@_index
13          LDI   @CONST+0,AR0
14          SUBI  1,AR0
15          LDI   *+FP(1),IR1
16          LDI   @CONST+1,AR1
17          MPYF  *+AR0(IR0),*+AR1(IR1),R0      // Multiplicación
18          ADDF  *+FP(2),R0                    // Acumulación
19          STF   R0,*+FP(2)
20          LDI   @_index,R1                    // Buffer circular
21          CMPI  10,R1
22          BNZ   L4
23          LDI   0,R1
24          STI   R1,@_index
25    L4:
26          LDI   *+FP(1),R2
27          ADDI  1,R2                          //Gestión índice bucle
28          STI   R2,*+FP(1)
29          CMPI  10,R2
30          BLT   L2                            // Final del bucle
31    EPI0_1:
32          LDI   *-FP(1),R1
33          BD    R1
```

Figura 71. Filtro FIR. Código ensamblador para TMS320C3x, I

```
1     (...)                          // Constantes diversas
2     N       .set    32             // Orden del filtro

3             .sect   "Impulse_Response"
4     H       .float  1.0
5     (...)                          // Respuesta impulsional
6     X       .usect  "Input_Buffer", 32
7     (...)                          // Buffer de entradas

8           .data
9     DATOS_FIR:
10    HADDR       .word   H          // * HADDR = respuesta impulsional
11    XADDR       .word   X          // * XADDR = buffer de entradas
12    IN          .word   001000h
13    OUT         .word   001001h

14          .text
15    init:                          // Inicialització
16          LDP     DATOS_FIR
17          LDI     N, BK
17          LDI     @HADDR, AR0    // * AR0 : respuesta impulsional
18          LDI     @XADDR, AR1    // * AR1 : buffer datos de entrada
19          LDI     @IN, AR2       // * AR2 : dato recibido
20          LDI     @OUT, AR3 // * AR3 : dato a enviar
21          RETS

22    filter:                        // Filtro
23         (...)              // Manejamos el stack
24         LDP      DATOS_FIR
25         LDI      *AR2, R1  // Recuperamos el dato del A/D
26         FLOAT    R1, R1          // Pasemos el dato a coma flotante
27         STF      R1, *AR1++%     // Guardemos el dato en el buffer
28         LDF      0, R0
29         LDF      0, R2

30         RPTS     N-1        // El bucle
31         MPYF3    *AR0++%, *AR1++%, R0  || ADDF3 R0, R2, R2

32         ADDF     R0, R2          // última acumulación
33         FIX      R2, R2          //Pasemos el resultado a coma fija
34         STI      R2, *AR3

35         (...)              // Retorno
36     RETS
```

Figura 72. Filtro FIR. Código ensamblador para TMS320C3x, II

4.2.4. Implementación sobre ADSP-2106x

El coste de ejecución del código de la figura 72 ya es óptimo. Presenta, no obstante, la dificultad de tratarse de un código ensamblador. Nos interesaría disponer de herramientas que permitan obtener costes iguales o parecidos trabajando en C.

A continuación estudiaremos las herramientas que ofrece *Analog Devices* para implementar un filtro FIR de manera óptima en C: macros y funciones.

❖ Filtros FIR con macros

Si estudiamos el código de la figura 71 llegamos a la conclusión que gran parte de las instrucciones parásitos que llevan a un incremento tan grande del coste de ejecución se deben a los accesos a memoria, gestión del *buffer* circular.

Para ayudar al compilador de C a interpretar estos accesos y codificarlos de forma óptima existen unas macros para gestionar *buffers* circulares: *CIRC_BUFFER* se utiliza para declarar un *buffer* circular, *BASE y LENGTH* para inicializarlo y CIRC_WRITE y CIRC_READ para acceder.

En la figura siguiente podemos observar cómo queda el código del filtro FIR utilizando estos macros:

```
1     #define MAX_H 3200                    // Orden del filtro

2     CIRCULAR_BUFFER (int, 1, in_cb);    // Declaramos buffer
3     int in[MAX_H];                      // Buffer de entradas
4     int h[MAX_H];                 //Respuesta impulsional
5     (...)

6     main () {
7           (...)
8           BASE (in_cb) = in;            //Inicio buffer circular
9           LENGTH (in_cb) = MAX_H; //    Longitud    del    buffer
circular
10          (...)
11    }

12    void spr0_asserted( int sig_num )   // RSI del A/D
13    {
14          int i;
15          int salida, dada;

16          salida=0;
17          CIRC_WRITE  (in_cb,  1,  rx_buf[1],  dm);//  Escritura
buffer
18          for (i=0;i<MAX_H;++i) {             // Bucle

19          CIRC_READ (in_cb, 1, dada, dm);     // Lectura  buffer
20          salida+=dada*h[i];                  // Producto y suma
21          }

22          tx_buf[2] = (int) salida;   // Enviamos dato al D/A
23    }
```

Figura 73. Filtro FIR en C sobre ADSP2106x utilizando macros

En la línea 2 avisamos al compilador de la existencia de un *buffer* circular (*in_cb*), de *integers*, que utilizará el juego de registros 1 para acceder a memoria. En total existen 8 juegos de registros, aunque no todos son utilizables en C. En la línea 3 se reserva memoria para el vector.

En les líneas 8 y 9 inicializaremos el *buffer* circular indicando la posición inicial del vector y de su longitud: la asociada al *buffer* lógico (*in_cb*) con el *buffer* físico (*in*).

En las líneas 17 y 19 accedemos al *buffer* para escritura y lectura. Ambos accesos utilizan el mismo índice: ambos operan y seguidamente lo incrementan. El parámetro *dm* indica que el vector de memoria está situado en la zona de datos (*data memory*).

Si a continuación generamos el ensamblador correspondiente al código de la figura 73 obtenemos:

```
En algún lugar del main:

1     (...)
2     b1=_in;           //Dirección base del buffer de entradas
3     l1=3200;          // Longitud del buffer de entradas
4     (...)


5     _spr0_asserted:
!     FUNCTION PROLOGUE: spr0_asserted
!     rtrts protocol, params in registers, DM stack, doubles
are floats
6           modify(i7,-1);
7     !     saving registers:
8           (...)
9           r12=dm(_rx_buf+1);      // r12 = dato recibido l'A/D
10          mrf=0;
11          r2=_h;
12          dm(i1,1)=r12;           // CIRC_WRITE
13          (...)
14    _L$15:
15          lcntr = 3200,  do _L$16-1 until lce;
16          r4=dm(i4,m6);           // Respuesta impulsional
17          r2=dm(i1,1);            // Buffer de entrada
18          mrf=mrf+r2*r4 (ssi);    // Suma y multiplicación
19    _L$16:
10          (...)                   // Retorno de la función
```

Figura 74. Filtro FIR en ensamblador sobre ADSP2106x utilizando macros

Fijémonos en que la macro *CIRCULAR_BUFFER* no genera código; *BASE* inicializa el registro *b1*, en la línea 2; *LENGTH* inicializa el registro *l1*, en la línea 3; *CIRC_READ* y *CIRC_WRITE* generan accesos de memoria (líneas 12 y17 respectivamente) que serán circulares ya que el juego de registros 1 tiene el registro de longitud *l1* inicializado.

El compilador ya implementa la estructura iterativa de forma óptima utilizando la instrucción *do _fi until lce* de manera que las instrucciones parásitos asociadas al índice del *for* se han eliminado.

El coste de ejecución de esta implementación es $3 \cdot N + K$ ya que dentro del bucle hay tres instrucciones (líneas 16 a 18): el compilador no es capaz de paralelizar automáticamente los accesos a memoria con el producto y la suma. Aún así el resultado obtenido es mucho mejor que el de la figura 71.

❖ Filtros FIR con funciones

Además de los macros el fabricante ofrece una librería de funciones que implementan las tareas más comunes en procesado digital de la señal, entre ellas los filtros FIR. Si es posible utilizar estas funciones en nuestra aplicación, ésta será la opción más óptima desde un punto de vista de coste de ejecución.

En este caso el código quedaría así:

```
1     #define MAX_H 95          // Orden del filtro

2     float dm in[MAX_H];       // Buffer de entrades (a la dm)
3     float pm h[MAX_H];        // Respuesta impulsional (en la pm)

4     void spr0_asserted( int sig_num )
5     {
6     // Filtrat

7     tx_buf[2] = fir( rx_buf[1], &h[0], &in[0], (int) MAX_H );
8     }
```

Figura 75. Filtro FIR en C sobre ADSP2106x utilizando funciones

Y su traducción a ensamblador:

```
1     _spr0_asserted:
!     FUNCTION PROLOGUE: spr0_asserted
!     rtrts protocol, params in registers, DM stack, doubles are
2           floats
3           .def end_prologue;      .val  .;    .scl  109;  .endef;
4           (...)              // Preparación de la llamada (10 inst.)
5           cjump (pc, _fir) (DB)   // Llama a la función FIR
6           (...)              // Retorno de la función (10 inst.)
7           RFRAME;

8     _fir:
            (...)                    // 16 instrucciones
9           LCNTR=R0, DO pdconv UNTIL LCE;
10    pdconv:
11    F8=F2*F4,F12=F8+F12,F4=DM(dm_ptr,dm_1),F2=PM(pm_ptr,pm_1);

12          F8=F2*F4, F12=F8+F12; // Operaciones pendientes ...
13          F0=F8+F12;

14          (...)                    // Retorno (8 instrucciones)
15    .ENDSEG;
```

Figura 76. Filtro FIR en ensamblador sobre ADSP2106x utilizando funciones

En este caso el coste de ejecución es $N+K$: se han paralelizado la suma, producto y accesos a memoria circulares.

4.2.5. Conclusiones

Como conclusiones al estudio podemos nombrar:

- El programador en lenguaje de alto nivel está en manos del compilador. Hay que verificar el código ensamblador generado por el compilador para calcular el coste de ejecución y tomar las medidas oportunas. La programación en lenguaje de bajo nivel es cada vez más complicada a causa de la complejidad de las arquitecturas y ensambladores de los nuevos DSP.

- A la hora de programar una aplicación hay que tener en cuenta la plataforma sobre la que se ha de ejecutar. Aunque los compiladores de los DSP son compatibles ANSI, no utilizar les herramientas adicionales que el fabricante facilita al programador significa no poder sacarle el máximo rendimiento al procesador.

- Siempre que sea posible hay que consultar los manuales del fabricante (*Applications Handbook*) y estudiar sus propuestas de implementación.

4.3. Implementación de la a FFT

4.3.1. Introducción. Transformadas frecuenciales

El análisis en el dominio frecuencial es, juntamente con el filtrado, una de las aplicaciones más importantes en el ámbito de la ingeniería. La interpretación de señales temporales como suma de sinusoides a menudo simplifica los problemas y ofrece nuevas herramientas que permitan solucionar cuestiones complejas de manera muy fácil.

Las transformadas pueden ser interpretadas como un producto escalar entre una función, objeto del análisis, y una base de funciones.

Si el producto escalar se define en forma de integral, el dominio de la transformada será continuo (caso de *Laplace* y *Fourier*). Este tipo de análisis no es el más adecuado para trabajar en entornos digitales computerizados, donde todas las señales están muestreados.

El producto escalar definido en forma de sumatoria indica dominio temporal discreto. La transformada *Z* permite un análisis de todo el plano (frecuencia y atenuación), ya que la base de funciones es una exponencial compleja con parte real e imaginaria. La TFSD (*Transformada de Fourier de Sèries Discretes*) sólo permite análisis frecuencial (circulo unitario del plano *Z*), análisis que para la mayoría de las aplicaciones a tiempo real será la adecuada.

De cara a implementarla, la TFSD presenta dos dificultades: el dominio de la transformada tiene infinitos puntos y el recorrido es continuo en frecuencia.
La DFT (*Discrete Function Transform*) soluciona estos dos problemas. Será la transformada que se utilizará en las aplicaciones que requieran el análisis frecuencial, aunque no se implementará directamente sino mediante un algoritmo computacionalmente óptimo: la FFT (*Fast Fourier Transform*).

El cuadro de la página siguiente clasifica diferentes transformadas frecuenciales según el dominio y el alcance del análisis.

Transformadas según análisis		DOMINI	
		Continuo	Discreto
ABAST ANÁLISI	Plano complejo	Laplace $\int_0^{\infty} f(t) \cdot e^{-st}\ dt$	Z $\sum_{-\infty}^{\infty} f[n] \cdot z^{-n}$
	Eje frecuencial	Fourier $\int_{-\infty}^{\infty} f(t) \cdot e^{-jwt}\ dt$	TFSD $\sum_{-\infty}^{\infty} f[n] \cdot e^{-jwn}$

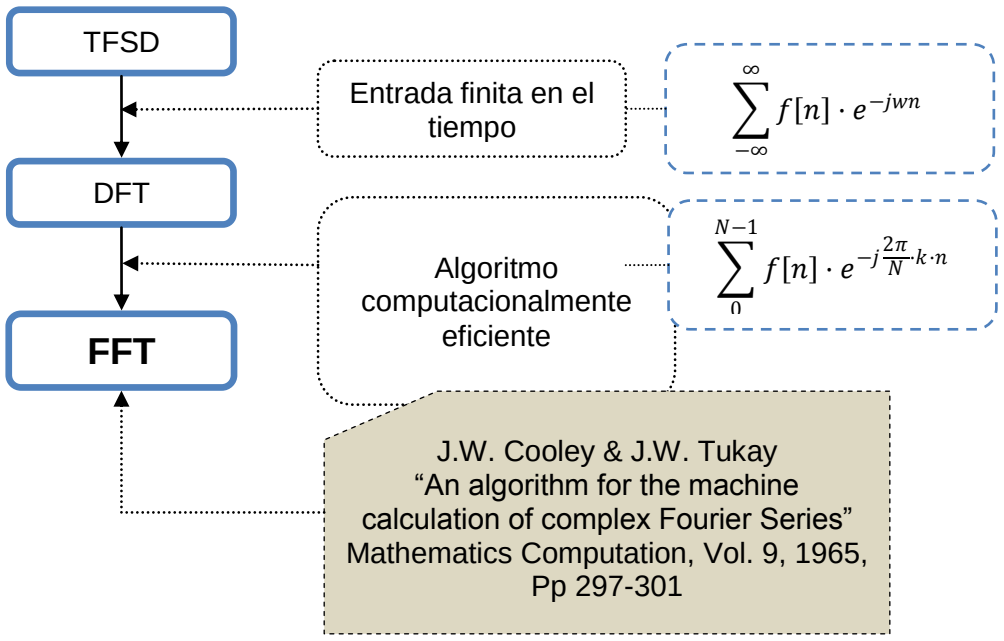


Figura 77. Clasificación de transformadas frecuenciales

4.3.2. Implementación directa de la DFT

Si tomamos la expresión de la DFT:

$$X(k)=\sum_{n=0}^{N-1}x[n]e^{-j\frac{2\cdot\pi}{N}\cdot k\cdot n}$$

Ecuación 25. DFT

y desarrollo el sumatorio llegan a la expresión matricial de la DFT:

$$\begin{pmatrix} e^{-j\cdot\frac{2\cdot\pi}{N}\cdot 0\cdot 0} & e^{-j\cdot\frac{2\cdot\pi}{N}\cdot 0\cdot 1} & (\ldots) & e^{-j\cdot\frac{2\cdot\pi}{N}\cdot 0\cdot(N-1)} \\ e^{-j\cdot\frac{2\cdot\pi}{N}\cdot 1\cdot 0} & e^{-j\cdot\frac{2\cdot\pi}{N}\cdot 1\cdot 1} & (\ldots) & e^{-j\cdot\frac{2\cdot\pi}{N}\cdot 1\cdot(N-1)} \\ (\ldots) & (\ldots) & (\ldots) & (\ldots) \\ e^{-j\cdot\frac{2\cdot\pi}{N}\cdot(N-1)\cdot 0} & e^{-j\cdot\frac{2\cdot\pi}{N}\cdot(N-1)\cdot 1} & (\ldots) & e^{-j\cdot\frac{2\cdot\pi}{N}\cdot(N-1)\cdot(N-1)} \end{pmatrix}\cdot\begin{pmatrix} x(0) \\ x(1) \\ (\ldots) \\ x(N-1) \end{pmatrix}=\begin{pmatrix} X(0) \\ X(1) \\ (\ldots) \\ X(N-1) \end{pmatrix}$$

Ecuación 26. Expresión matricial de la DFT

El cálculo de la transformada $X(k)$ de forma directa implica, pues, el producto de una matriz compleja NxN por un vector de datos temporales de longitud N. Este vector es real, pero a la hora de calcular el coste computacional lo consideraremos complejo para hacer extensivo el análisis también el caso de la transformada inversa.

La implementación de la DFT tiene, pues, un coste computacional de N^2 productos complejos y $N\cdot(N-1)$ sumas complejas, que equivalen, aproximadamente, a $4\cdot N^2$ productos reales y $4\cdot N^2$ sumas reales

- Nota: Este último cálculo es válido si la entrada a la DFT es un vector complejo. Se propone rehacerlo si el vector de entrada es real.

Observemos en la tabla siguiente el recuento de operaciones en el caso de una aplicación que calcula continuamente la DFT de vectores de datos que provienen de un flujo continuo de entrada.

Lo calculamos en tres casos de longitud de bloque diferentes: 512, 1024 y 2048.

Longitud DFT	*Nº DFT / seg.*	*Nº sumas / DFT*	*Nº prod. / DFT*	*Nº op. / seg.*
512	86	1048576	1048576	180.355.072
1024	43	4194304	4194304	360.710.144
2048	21,5	16777216	16777216	721.420.288

Figura 78. Recuento de operaciones de la DFT

Destacamos dos observaciones:

- El número de operaciones que requiere el cálculo de la DFT de forma directa es muy grande. Hay que tener en cuenta, además, que solo hemos contabilizado las operaciones matemáticas; un código real tendrá, probablemente, muchas otras instrucciones parásitas.

- El número de operaciones depende de la longitud de la DFT. Hay que tener en cuenta que la longitud de la DFT es un parámetro que se suele escoger en función de la resolución requerida (especificación o parámetro que puede cambiar el usuario); un cambio en este valor implica una variación muy grande en el número de operaciones a ejecutar por segundo y, por tanto, unos requerimientos muy diferentes por el procesador que implementa la transformada frecuencial (*hardware*). Esta dependencia *software – hardware* no es deseable.

Convendría disponer de un algoritmo que redujese al máximo el número de operaciones y, además, este fuese independiente de la longitud de le análisis.

4.3.3. La FFT

Existen dos variantes del algoritmo de la FFT: delmación en tiempo y delmación en frecuencia. A continuación explicaremos el primer.

En el cálculo de la DFT aparece de forma destacada la exponencial compleja. Como veremos a continuación, la exponencial compleja tiene dos propiedades que nos hacen pensar que el número de operaciones necesarias para calcular la DFT se puede reducir de manera importante:

- Propiedad de simetría:

$$e^{-j\cdot\frac{2\cdot\pi}{N}\cdot k} = -e^{-j\cdot\frac{2\cdot\pi}{N}\cdot\left(k+\frac{N}{2}\right)}$$

Ecuación 27. Propiedad de simetría

- Propiedad de periodicidad:

$$e^{-j\cdot\frac{2\cdot\pi}{N}\cdot k} = e^{-j\cdot\frac{2\cdot\pi}{N}\cdot(k+N)}$$

Ecuación 28. Propiedad de periodicidad

Veremos a continuación cómo reordenando adecuadamente los términos del sumatorio de la DFT y mediante sencillas manipulaciones algebraicas conseguiremos aprovechar las propiedades de simetría y periodicidad de la exponencial compleja para reducir drásticamente el número de operaciones matemáticas necesarias para ejecutar la DFT. Sin que esto implique pérdida de generalidad, explicaremos el proceso con un ejemplo: el cálculo de la DFT de 8 puntos.

La expresión para calcular una DFT de 8 puntos es:

$$x(k) = \sum_{n=0}^{7} x[n]\cdot e^{-j\cdot\frac{2\cdot\pi}{8}\cdot k\cdot n}$$

Ecuación 29. DFT-8

Recordemos que para implementar de forma directa esta expresión necesitamos ejecutar 64 productos complejos y 56 acumulaciones complejas.
Separemos el sumatorio en dos términos: uno que incluye los pares y otro para los impares:

$$\sum_{n=0}^{7}{}^{n\ parell} x[n]\cdot e^{-j\cdot\frac{2\cdot\pi}{8}\cdot k\cdot n} + \sum_{n=0}^{7}{}^{n\ \text{sen}\ ar} x[n]\cdot e^{-j\cdot\frac{2\cdot\pi}{8}\cdot k\cdot n}$$

Ecuación 30. DFT-8, descomposición I

Rescribir la Ecuación anterior, con un razonamiento de índice:

$$\sum_{m=0}^{3} x[2\cdot m]\cdot e^{-j\frac{2\cdot\pi}{8}\cdot k\cdot 2\cdot m} + \sum_{m=0}^{3} x[2\cdot m+1]\cdot e^{-j\frac{2\cdot\pi}{8}\cdot k\cdot(2\cdot m+1)}$$

Ecuación 31. DFT-8, descomposición II

Extraigamos el factor común al segundo sumatorio del término $e^{-j\frac{2\cdot\pi}{8}\cdot k}$:

$$\sum_{m=0}^{3} x[2\cdot m]\cdot e^{-j\frac{2\cdot\pi}{4}\cdot k\cdot m} + e^{-j\frac{2\cdot\pi}{8}\cdot k}\cdot\sum_{m=0}^{3} x[2\cdot m+1]\cdot e^{-j\frac{2\cdot\pi}{4}\cdot k\cdot m}$$

Ecuación 32. DFT's-4

Notemos que hemos llegado a una expresión que hemos descompuesto una DFT-8 en dos DFT-4 (un de los términos pares y otra de los términos impares de la secuencia original) más un término ($e^{-j\frac{2\cdot\pi}{8}\cdot k}$) que multiplica la segunda DFT-4. Si hacemos recuento de operaciones obtenemos $2\cdot 16+8=40$ productos y $2\cdot 12+8=32$ sumas. Por tanto, con unas sencillas manipulaciones hemos conseguido reducir el número de operaciones necesarias para calcular la DFT.

Conviene destacar que las dos DFT-4 que se han obtenido a partir de la descomposición de la expresión original tienen una particularidad: aunque el índice temporal *m* tiene un recorrido de 4 puntos (des de 0 hasta 3), el índice frecuencial *k* puede variar entre 0 y 7 (ya que la expresión proviene de una DFT-8).

Si repetimos el proceso,

$$\left(\underbrace{\sum_{m=0}^{1} x[2\cdot(2\cdot m)]\cdot e^{-j\frac{2\cdot\pi}{2}\cdot k\cdot m}}_{(A_{0,1})} + e^{-j\frac{2\cdot\pi}{4}\cdot k}\cdot\underbrace{\sum_{m=0}^{1} x[2\cdot(2\cdot m+1)]\cdot e^{-j\frac{2\cdot\pi}{2}\cdot k\cdot m}}_{(B_{0,1})}\right) +$$
$$e^{-j\frac{2\cdot\pi}{8}\cdot k}\cdot\left(\underbrace{\sum_{m=0}^{1} x[2\cdot(2\cdot m)+1]\cdot e^{-j\frac{2\cdot\pi}{2}\cdot k\cdot m}}_{(C_{0,1})} + e^{-j\frac{2\cdot\pi}{4}\cdot k}\cdot\underbrace{\sum_{m=0}^{1} x[2\cdot(2\cdot m+1)+1]\cdot e^{-j\frac{2\cdot\pi}{2}\cdot k\cdot m}}_{(D_{0,1})}\right)$$

Ecuación 33. DFT's-2, I

La expresión anterior contiene cuatro DFT-2, más algunos términos que ya conocemos ($e^{-j\frac{2\cdot\pi}{2}\cdot k}$, $e^{-j\frac{2\cdot\pi}{4}\cdot k}$, $e^{-j\frac{2\cdot\pi}{8}\cdot k}$).

Ara también las DFT-2 tienen una característica especial: el índice frecuencial *k* no toma los valores $[0,1]$, sino $[0,7]$.

Hasta el momento el proceso que hemos descrito modifica, de alguna manera, el orden de los datos de entrada. Mientras que la DFT-8 original toma los datos de forma consecutiva $[0,1,2,...,7]$, las cuatro DFT-2 operan, respectivamente, sobre los vectores $[0,4]$, $[2,6]$, $[1,5]$, $[3,7]$.

La transformación frecuencial aún no está calculada. Aún no hemos aplicado las propiedades de simetría ni periodicidad. Lo haremos a continuación.

Fijémonos en cada una de las cuatro DFT-2:

$$(A_{0,1}): \quad x(0)+x(4)\cdot e^{-j\frac{2\cdot\pi}{2}\cdot k} \quad (k: 0 \ldots 7)$$

$$(B_{0,1}): \quad x(2)+x(6)\cdot e^{-j\frac{2\cdot\pi}{2}\cdot k} \quad (k: 0 \ldots 7)$$

$$(C_{0,1}): \quad x(1)+x(5)\cdot e^{-j\frac{2\cdot\pi}{2}\cdot k} \quad (k: 0 \ldots 7)$$

$$(D_{0,1}): \quad x(3)+x(7)\cdot e^{-j\frac{2\cdot\pi}{2}\cdot k} \quad (k: 0 \ldots 7)$$

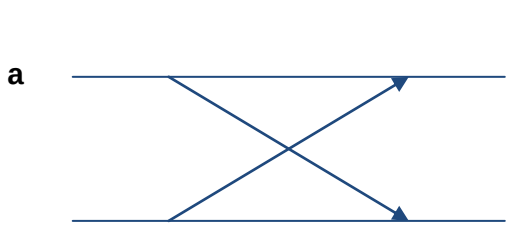
Ecuación 34. DFT's-2, II

Aunque el índice frecuencial *k*, como ya hemos visto, toma valores $[0,7]$, si aplicamos las propiedades ya nombradas observamos que la exponencial compleja no toma ocho valores diferentes ya que:

$$e^{-j\cdot\frac{2\cdot\pi}{2}\cdot 0} = e^{-j\cdot\frac{2\cdot\pi}{2}\cdot 2} = e^{-j\cdot\frac{2\cdot\pi}{2}\cdot 4} = e^{-j\cdot\frac{2\cdot\pi}{2}\cdot 6}$$
$$e^{-j\cdot\frac{2\cdot\pi}{2}\cdot 1} = e^{-j\cdot\frac{2\cdot\pi}{2}\cdot 3} = e^{-j\cdot\frac{2\cdot\pi}{2}\cdot 5} = e^{-j\cdot\frac{2\cdot\pi}{2}\cdot 7}$$
$$e^{-j\cdot\frac{2\cdot\pi}{2}\cdot 0} = -e^{-j\cdot\frac{2\cdot\pi}{2}\cdot 1}$$

Ecuación 35. Aplicación simetría y periodicidad

Por tanto, para calcular estas DFT-2 necesitamos muy pocas operaciones. Tan pocas como un producto complejo y dos sumas complejas. Gráficamente se suele representar mediante la estructura de cálculo "mariposa":



$W_2^{\,0}$

$a + b \cdot W_N^k$

$a - b \cdot W_N^k = a + b \cdot W_N^{k+\frac{N}{2}}$

***Figura* 79. Mariposa**

Donde hemos introducido la notación a menudo utilizada en los libros para referirse al exponencial complejo:

$$W_N^k = e^{-j\frac{2\cdot\pi}{N}\cdot k}$$

Ecuación 36. Notación por el 'exponencial complejo

Por tanto, todas las operaciones de la Ecuación se pueden resolver mediante 4 mariposas, de manera que sustituyendo a la Ecuación 33 obtenemos:

$$X(k) = \begin{pmatrix} (E_{0\ldots3}) \\ A_{0,1} + e^{-j\cdot\frac{2\cdot\pi}{4}\cdot k}\cdot B_{0,1} \end{pmatrix} + e^{-j\cdot\frac{2\cdot\pi}{8}\cdot k}\cdot\begin{pmatrix} (F_{0\ldots3}) \\ C_{0,1} + e^{-j\cdot\frac{2\cdot\pi}{4}\cdot k}\cdot D_{0,1} \end{pmatrix}$$

Ecuación 37. DFT's-4, recomposición, I

Donde $A_{0,1}$ son los dos resultados de aplicar la mariposa a los datos $x(0), x(4)$:

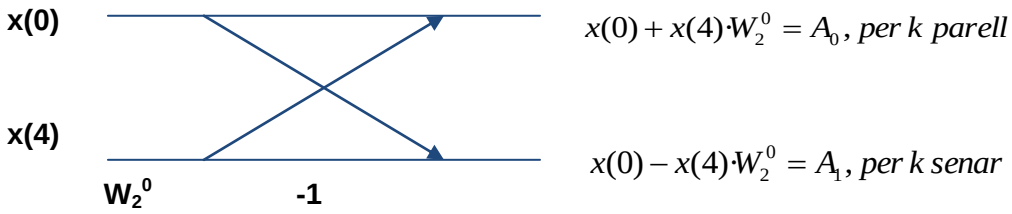


Figura 80. Mariposa aplicada a DFT-2

Análogamente para $B_{0,1}$, $C_{0,1}$, $D_{0,1}$.

Fijémonos en que el primer término de Ecuación 37, a medida que varía el índice *k* entre $[0,7]$, dará lugar a: $A_0 + W_4^0\cdot B_0$, $A_1 + W_4^1\cdot B_1$, $A_0 + W_4^2\cdot B_0$, $A_1 + W_4^3\cdot B_1$, $A_0 + W_4^4\cdot B_0$, $A_1 + W_4^5\cdot B_1$, $A_0 + W_4^6\cdot B_0$, $A_1 + W_4^7\cdot B_1$.

Los cuatro últimos términos, aplicando la propiedad de periodicidad, son idénticos a los cuatro primeros y a la vez estos, aplicando simetría, se pueden calcular mediante dos mariposas:

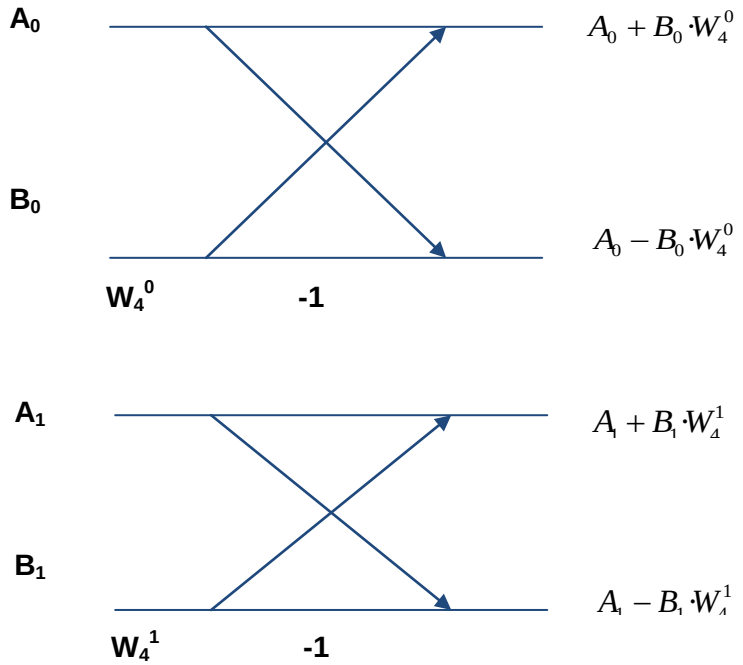


Figura 81. Mariposas de la segunda fase

En la literatura estas mariposas se suelen sobreponer de la manera siguiente:

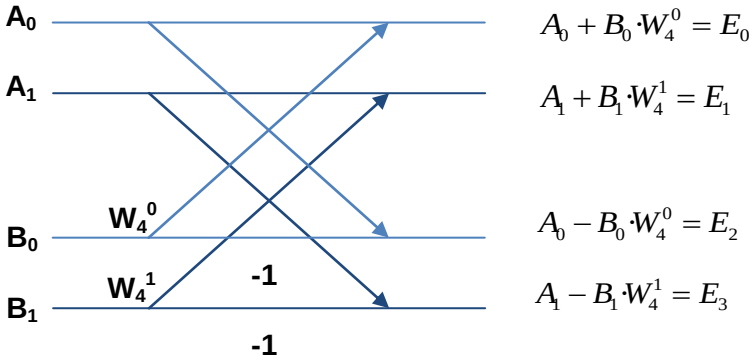


Figura 82. Mariposas de la segunda fase, agrupadas

Procediendo con el segundo término de la Ecuación 37 de manera idéntica a cómo lo acabamos de hacer con el primero, llegamos a:

$$X(k) = \begin{matrix} (X_{0..7}) \\ E_{0..3} + e^{-j\cdot\frac{2\cdot\pi}{8}\cdot k} \cdot F_{0..3} \end{matrix}$$

Ecuación 38. DFT-8, recomposición, II

Repitiendo el proceso veríamos que la Ecuación 38 se puede calcular también mediante cuatro mariposas de manera que finalmente obtendríamos ya el resultado final de la DFT calculado mediante el algoritmo de la FFT.

Vemos en la figura siguiente el proceso completo esquematizado mediante mariposas:

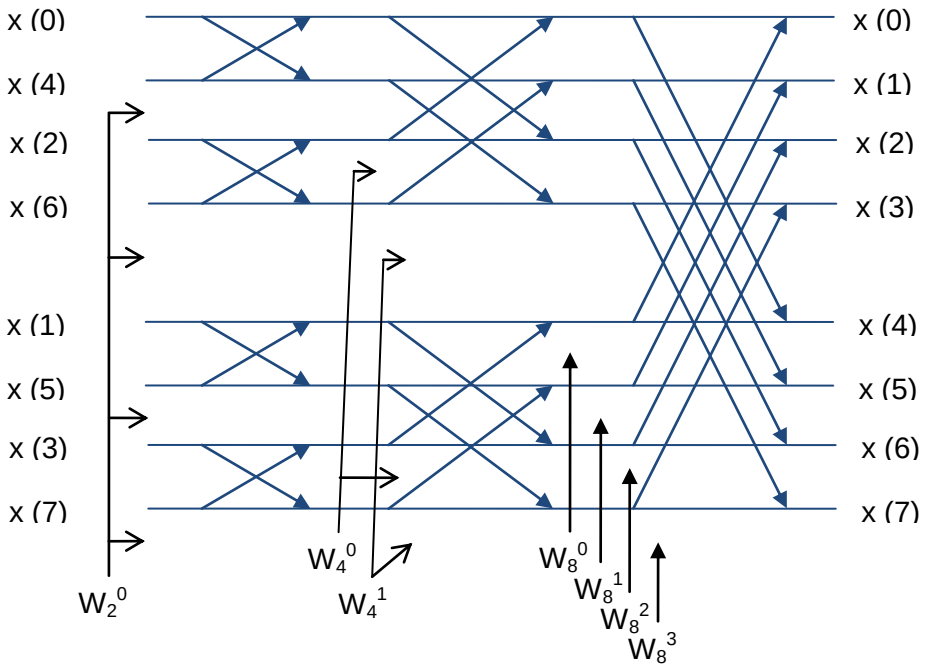


Figura 83. FFT-8

Vemos, pues, que el algoritmo de la FFT tiene dos pasos:

- Desorden de los datos de entrada según un algoritmo que se llama *bit-reversal* (lo explicaremos más adelante).

- Cálculo de la transformada frecuencial aplicando diversas estructuras mariposa.

4.3.4. Coste computacional de la FFT

Evaluaremos el coste computacional del algoritmo de la FFT generalizando los resultados obtenidos para la FFT-8.

Dado que la estructura computacional básica es la mariposa, procederemos calculando en primer lugar el número de mariposas necesarias para calcular

la FFT-N y seguidamente multiplicaremos por el número de operaciones reales que requiere el cálculo de ésta.
Las mariposas en una FFT se agrupan en fases. Fijémonos en que la FFT-8 tiene tres fases. En general, la FFT-N tendrá $\log_2(N)$ fases. Además, cada fase está compuesta por $N/2$ mariposas. Por tanto, el número total de mariposas de una FFT-N será:

$$N/2 \cdot \log_2(N)$$

Ecuación 39. Nº mariposas de una FFT-N

El número de operaciones reales diferentes de una mariposa es 4 productos y 6 sumas. En total, 10 operaciones matemáticas.

$$a + b \cdot W_N^K = \left(a_R + \left(b_R \cdot W_{N\ R}^K - b_I \cdot W_{N\ I}^K\right)\right) + j\left(a_I + \left(b_R \cdot W_{N\ I}^K + b_I \cdot W_{N\ R}^K\right)\right)$$
$$a - b \cdot W_N^K = \left(a_R - \left(b_R \cdot W_{N\ R}^K - b_I \cdot W_{N\ I}^K\right)\right) + j\left(a_I - \left(b_R \cdot W_{N\ I}^K + b_I \cdot W_{N\ R}^K\right)\right)$$

Ecuación 40. Ecuaciones de una mariposa

Por tanto, el coste computacional de una FFT-N expresado en número de operaciones reales es:

$$5 \cdot N \cdot \log_2(N)$$

Ecuación 41. Coste computaciones FFT-N

Convendría comprobar si hemos mejorado en los dos aspectos que nos propusieron al final del 4.3.2. Recordémoslo:

- Reducir drásticamente el número de operaciones

- Intentar que el número de operaciones por segundo que ejecuta un sistema que calcula la DFT de todos los datos de entrada sea independiente de la medida del bloque (*N*)

Verifiquémoslo reproduciendo el ejemplo de la figura 78, aplicándolo ahora a la FFT:

Longitud FFT	Nº FFT / seg.	Nº sumas / FFT	Nº prod. / FFT	Nº op. / seg.
512	86	9216	13284	1.935.000
1024	43	20480	30720	2.201.600
2048	21,5	45056	67584	2.421.760

Figura 84. Recuento de operaciones de la FFT

Fijémonos en que el primer objetivo se ha completado con creces. La reducción que se produce en el número de operaciones en el caso de una medida de bloques de 1024 muestras es en un factor 163,84.

En lo que se refiere al segundo objetivo se comprueba que mientras que en la implementación directa de la DFT (figura 78) el número de operaciones se doblaba al doblar la medida del bloque (*N*), mediante la FFT el coste sólo aumenta en un 10 %.

4.3.5. Implementación de la FFT sobre DSP

Hasta ahora hemos visto una mejora en un algoritmo (la FFT) que reduce de manera muy importante el coste computacional de un proceso (la DFT). Ahora bien, nos podemos beneficiar de las mejoras introducidas por la FFT implementándola en cualquier procesador. ¿Qué ofrece, además, un DSP para que sea óptimo ejecutar la FFT sobre este tipo de computadores? Lo veremos a continuación aplicándolo al *ADSP-2106x de Analog Devices*.

❖ Direccionamiento bit-reversal

Hemos comentado al final del apartado 4.3.3 que previo a la aplicación de la FFT hay que desordenar los datos de entrada según un algoritmo denominado *bit-reversal*.
Para calcular el orden *bit-reversal* hay que determinar en primer lugar la medida del bloque (*N*). Hagámoslo por $N = 8$.

La tabla siguiente indica el proceso:

Direccionamiento lineal	Expresado en binario	... invirtiendo dígitos ...	Direcc. *bit-reversal*
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Figura 85. Direccionamiento bit-reversal para un vector de 8 puntos

El ADSP-2106x (y la mayoría de los DSP) incluyen al juego de instrucciones el direccionamiento *bit-reversal*:

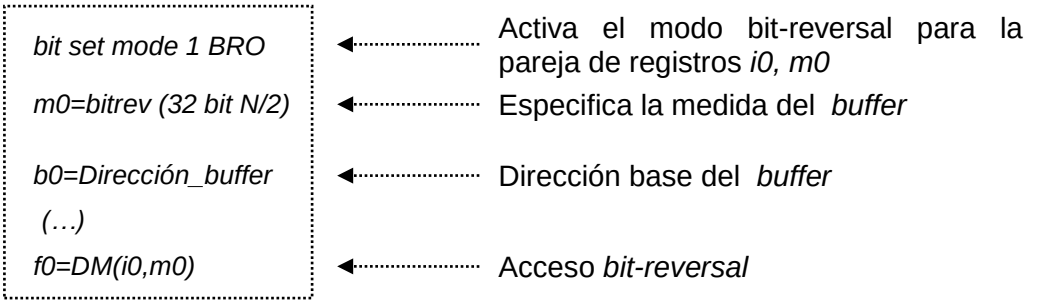


Figura 86. Direccionamiento *bit-reversal* con el ADSP-2106x

Una vez inicializado el *buffer* no hay ninguna penalización en el acceso *bit-reversal.* Implementar una FFT en un procesador DSP nos ahorra en tiempo correspondiente al desorden del vector de entradas que haría falta programar en cualquier otro computador.

❖ La mariposa

Tal y como hemos deducido en el apartado 4.3.4 la implementación de una mariposa requiere 10 operaciones matemáticas, además de los numerosos accesos a memoria. Vemos cómo resuelve esta estructura de cálculo el ADSP-2106x:

```
lcntr=r15,     do   end_bfly   until        /*Hacer una mariposa en cada grupo -2*/
               lce;
  f8=f1*f6,    f14=f11-f14,                          dm(i2, m0)=f10,   f9=pm(il1,m8);
  f11=f1*f7,   f3=f9+f14,              f9=f9-f14,     dm(i2,m0)=f13,    f7=pm(i8,m8);
  f14=f0*f6,   f12=f8+f12,                           f8=dm(i0,m0),     pm(il0,
                                                                       m10)=f9;
end_bfly:
  f12=f0*f7,   f13=f8+f12,             f10=f8-f12,   f6=dm(i0,m0),     pm(i10,m10)=f3;
```

Figura 87. Implementación de una mariposa en el ADSP-2106x

Identifiquemos las 10 operaciones (4 productos, 3 sumas y 3 restas) y 8 accesos a memoria que se ejecutan en sólo 4 ciclos de reloj.

Fijémonos en que en el segundo y cuarto ciclo hay 3 instrucciones matemáticas ejecutadas en un sólo ciclo: un producto, y una suma y una resta con los mismos operandos origen. (f9 y f14, f8 y f12).

4.4. Conceptos de aritmética de precisión finita

Con el término aritmética de precisión finita nos referiremos al formato y medida que utiliza un procesador para almacenar los datos. La aritmética de un procesador es un factor importante a la hora de escoger uno ya que determina significativamente su precio, velocidad asó como tiempo de desarrollo de las aplicaciones.

Según el formato de la aritmética diferenciaremos los procesadores de coma fija (ya sea entera o fraccional) y los de coma flotante. Estos términos hacen referencia al agente que gestiona la coma decimal: el propio programador o el procesador respectivamente.

La medida de la aritmética determina el número de bits que utiliza el procesador para almacenar los números. Los valores típicos son 16, 24 ó 32 bits.

Las aritméticas de precisión finita se caracterizan por una determinada resolución (mínimo número representable, positivo y diferente de cero) y un margen dinámico (cociente, normalmente expresado en dB, entre el máximo valor representable por la aritmética y la resolución). Cuando una aritmética es con signo, se suele especificar.

En operar con una aritmética que presenta una resolución y margen dinámico determinados existe el riesgo de que se produzca ya sea un *overflow* (el resultado de una operación es superior al máximo valor representable por la aritmética) o bien *underflow* (el resultado de la operación tiene una resolución mayor a la que ofrece la aritmética). Ambos factores provocaran una desviación en el funcionamiento esperado del algoritmo: el *overflow* provoca resultados erróneos en orden de magnitud y el *underflow* resultados poco precisos.

Los dos factores (*overflow* y *underflow*) requerirán replantear el correcto funcionamiento del algoritmo en la fase de implementación. Efectivamente, en el proceso de desarrollo de una aplicación de procesado digital de señal podemos diferenciar dos etapas:

- Durante el diseño de la aplicación hay que determinar el algoritmo que permitirá cumplir las especificaciones. En esta fase se hará incidencia en la estructura del *software*, variables que se utilizarán, costes computacionales, etc. Se suele suponer que se trabaja con una aritmética de precisión infinita (o casi – infinita); es a decir, números reales.

- Durante la implementación del algoritmo nos enfrentamos con un procesador concreto, con una aritmética de precisión finita. Habrá que verificar que el algoritmo diseñado continúa siendo funcionalmente correcto.

En general esto equivaldrá a asegurar que no es produce *overflows* y que los *underflows* no malmeten el margen dinámico del sistema.

Es importante remarcar que, en general, es concentraran los esfuerzos para evitar el *overflow*. El *underflow* suelen ser un riesgo que se pasa por alto: hay que ser consciente de los efectos que provoca pero no suele ser necesario evitarlos.

En los apartados siguientes analizaremos las aritméticas de precisión finita en lo que respecta a formato, resolución, margen dinámico y comportamiento respecte la suma y el producto. A continuación estudiaremos el efecto del *overflow* así como algunas técnicas para combatirlo (escalado). Lo haremos aplicado al algoritmo de la FFT. Así mismo analizaremos los efectos del *underflow* sobre la implementación de filtros. Todo esto para poder diseñar algoritmos teniendo en cuenta desde un principio el factor aritmética.

4.4.1. Conceptos básicos

Un número cualquiera *x* se puede expresar, de forma genérica, como

$$x = b \cdot 2^{e}$$

Ecuación 42. Número binario

donde *b* recibe el nombre de mantisa y *e* exponente. Tanto *b* como *e* se pueden expresar con signo (a menudo en complemento a dos: CA2) o bien sin signo. Los diferentes formatos de aritmética distribuyen sus bits en parte para códigoficar la mantisa y en parte para codificar el exponente. Cada uno de estos bits tendrá asignado un peso determinado, expresado como una potencia de dos.

Por ejemplo, sería posible definir un formato de 24 bits de manera que 8 de ellos codificasen el exponente ($e_0 ... e_7$) y 16 la mantisa ($b_0 ... b_{15}$). Suponemos también que el formato que estamos proponiendo sólo permite números positivos, los pesos del exponente son potencias crecientes positivas y los pesos de la mantisa potencian decrecientes negativas. Entonces, dado un valor determinado de los 24 bits, el número representado sería:

$$x = \left(b_0 \cdot 2^{-1} + b_1 \cdot 2^{-2} ... + b_{15} \cdot 2^{-16}\right) 2^{\left(e_0 2^0 + ... + e_7 \cdot 2^7\right)}$$

Ecuación 43. Ejemplo de aritmética

Fácilmente podemos calcular la resolución y margen dinámico de este formato:

- Resolución: $2^{-16}\cdot 2^{0}=2^{-16}$, donde se han asignado tanto a la mantisa como a los exponentes valores de manera que el número obtenido sea el menor posible (diferente de cero)
- Máximo número: $\left(2^{-1}+2^{-2}+\ldots+2^{-16}\right)\cdot 2^{\left(2^{0}+2^{1}+\ldots+2^{7}\right)}\approx 2^{255}$
- Margen dinámico: $2^{255}/2^{-16}=2^{271}$

4.4.2. Aritmética de coma fija

Los formatos de aritmética fija de coma fija asignan todos los bits para codificar la mantisa.

En general, se suele utilizar la notación A.B para indicar el número de bits que tienen asignados pesos mayores o igual a cero (A) y el número de bits con pesos negativos (B).

En caso de formatos que permiten números negativos, el bit de signo se contabiliza como bit con peso mayor o igual a cero
Así, los formatos de coma fija se clasifican en:

- Entera. Todos los pesos son mayores o iguales a cero ($B=0$)
- Fraccional. Algún peso es negativo($B\neq 0$)

Por ejemplo, un formato de coma fija con signo de tipos 3.13 reserva un bit para el signo, 2 bits para la parte entera y 13 per la parte decimal.

Podríamos calcular la resolución y el margen dinámico y obtendríamos:

- Resolución: 2^{-13}
- Máximo número: $\left(2^{1}+2^{0}+2^{-1}+\ldots+2^{-13}\right)\approx 4$
- Margen dinámico: $2^{2}/2^{-13}=2^{15}$

Fijémonos en que el margen dinámico que hemos encontrado sólo considera los números positivos. Si tenemos en cuenta también los negativos (el formato lo soporta), el margen dinámico total seria el doble del encontrado: $2^{15}\cdot 2=2^{16}$.

En general, y si no se explicita lo contrario, entenderemos que un formato de coma fraccional tiene el formato 1.N-1 (o bien 0.N si sólo permite números positivos), de forma que margen de valores representable será entre -1 y 1 (o bien entre 0 y 1 si es con signo).

❖ Aritmética de coma fija entera

Sea un formato de coma fija entera de *N* bits. Es fácil calcular su resolución (*1*) y su margen dinámico (2^N). ¿Cómo se comporta este tipo de aritmética ante las operaciones de suma y producto? Tendremos que evaluar, en el peor caso, si existe riesgo de que se produzca *overflow / underflow.* Como es de esperar, el peor caso cuando estudiemos el *overflow* es el de operandos origen grandes y cuando estudiemos el *underflow* operandos origen pequeños. Veámoslo.

❖ *Riesgo de overflow en la suma.*

$$y = \left(2^{N-1} + 2^{N-2} + \ldots + 2^0\right) + \left(2^{N-1} + 2^{N-.2} + \ldots + 2^0\right) = \left(2^N + 2^{N-1} + \ldots + 2^1\right)$$

Ecuación 44. Aritmética coma fija entera. Overflow en la suma

Vemos que en el peor caso en sumar dos números de *N* bits el resultado está codificando con *N+1* bits. El riesgo de o*verflow*, por tanto, existe.

❖ *Riesgo de underflow en la suma.*

$$y = \left(2^0\right) + \left(2^0\right) = \left(2^1\right)$$

Ecuación 45. Aritmética coma fija entera. Underflow en la suma

Ahora el peor caso es los menores números posibles expresables con *N* bits. Vemos que el resultado es expresable con la aritmética estudiada y, por tanto, el riesgo de un*derflow* en la suma no existe.

❖ *Riesgo de overflow en el producto*

$$y = \left(2^{N-1} + 2^{N-2} + \ldots + 2^0\right)\left(2^{N-1} + 2^{N-.2} + \ldots + 2^0\right) = \left(2^{2\cdot(N-1)} + \ldots + 2^0\right)$$

Ecuación 46. Aritmética coma fija entera. Overflow en el producto

El resultado del producto de dos números de N bits no se puede expresar con N bits. Por tanto el riesgo de *overflow* en el producto existe.

❖ *Riesgo de underflow en el producto*

$$y = (2^0)\cdot(2^0) = (2^0)$$

Ecuación 47. Aritmética coma fija entera. Underflow en el producto

Ahora el peor es los menores números posibles expresables con *N* bits. Vemos que el resultado es expresable con la aritmética estudiada y, por tanto, el riesgo de un*derflow* en el producto no existe.

En la tabla siguiente resumimos los resultados obtenidos:

	Suma	**Producto**
Overflow	Sí	Sí
Underflow	No	No

Figura 88. Tabla de riesgos de la aritmética de coma fija entera

❖ Aritmética de coma fija fraccional

Sea un formato de coma fija fraccional de *N* bits, *A.B*. El razonamiento siguiente lo haremos suponiendo el caso más habitual en que *A=1*, *B=N-1*, es decir, cuando sólo es pueden representar números en el intervalo *[-1, 1]*.

La resolución será 2^{-B} y su margen dinámico (2^N). Dejemos como ejercicio calcular los riesgos de *overflow* y *underflow* en la suma y el producto. En la tabla siguiente resumimos los resultados que deberíamos obtener:

	Suma	**Producto**
Overflow	Sí	No
Underflow	No	Sí

Figura 89. Tabla de riesgos de la aritmética de coma fija fraccional

Si comparamos la coma fija fraccional con la entera vemos que la coma fija fraccional elimina el riesgo de o*verflow* en el producto pero introduce el de un*derflow*. o

Este intercambio de riesgos resulta muy beneficioso a la hora de implementar algoritmos ya que el error de *overflow* es más grave que el de un*derflow* (recordemos que el *overflow* provoca errores de un orden de magnitud mucho más grande que el *underflow*)

4.4.3. Aritmética de coma flotante

La aritmética de coma flotante reserva un conjunto de bits para codificar la mantisa y otros para codificar el exponente. Uno de los formatos más utilizados es el siguiente:

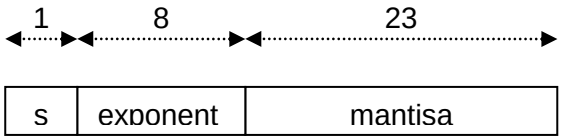


Figura 90. Formato de coma flotante

Donde:

s es el signo de la mantisa

exponente son los 8 bits en CA2 (pesos positivos) que codifican el exponente

mantisa son los 23 bits (pesos negativos) que codifican la mantisa

Calculemos la resolución y el margen dinámico:

- Resolución: $2^{-128} \cdot 2^{-23} = 2^{-151}$
- Máximo número representable: $2^{127} \cdot \left(2^{-1} + 2^{-2} + \ldots + 2^{-23}\right) \approx 2^{127}$
- Margen dinámico: $2^{127} \Big/ 2^{-151} = 2^{278}$

Como se trata de un formato con signo para la mantisa, el margen dinámico total se debería multiplicar por dos.

Dejemos como ejercicio evaluar los riesgos de *overflow* y *underflow* en la suma y el producto. Vemos los resultados en la tabla siguiente:

	Suma	**Producto**
Overflow	Raramente	Raramente
Underflow	Sí	Sí

Figura 91. Tabla de riesgos de la aritmética de coma flotante

Fijémonos en que la coma flotante casi elimina el riesgo de *overflow* ya que, si bien es un riesgo que existe, en la realidad es prácticamente imposible que se dé.

Así, mientras que cuando trabajamos con procesadores de coma fija tendremos que vigilar para evitar los riesgos de *overflow*, a la hora de programar con coma flotante este peligro prácticamente desaparece. De hecho, la coma flotante es la mejor aproximación a la aritmética de precisión infinita de la que disponemos.

4.4.4. Tabla resumen de riesgos

En la tabla siguiente recogemos los resultados obtenidos hasta el momento.

	Suma		Producto	
	Overflow	***Underflow***	***Overflow***	***Underflow***
Coma fija entera	Sí	No	Sí	No
Coma fija fraccional (*1. N-1*)	Sí	No	No	Sí
Coma flotante	Raramente	Sí	Raramente	Sí

Figura 92. Tabla de riesgos resumen

Es importante tener bien presente la tabla anterior ya que el análisis para evitar el escalado requiere conocer en que operaciones hay riesgo de que se produzca *overflow* para poder a continuación poner los medios necesarios para evitarlo.

4.4.5. El overflow. Técnicas de escalado

Como ya ha quedado dicho, el *overflow* es una consecuencia de trabajar con coma fija de precisión finita. El *overflow*, o saturación del margen dinámico de la aritmética, produce errores en los resultados de las operaciones matemáticas; errores que pueden ser de gran magnitud. En este caso, los algoritmos que sobre papel funcionaban correctamente dejan de hacerlo cuando se implementan.

Para afrontar este problema hay que llevar a cabo un análisis del algoritmo, posterior a haber verificado su corrección en la fase de diseño, y previa a la implementación. En caso de observar peligro de *overflow*, hay que proceder aplicando la técnica de escalado que convenga más a nuestro diseño. El diagrama de bloques siguiente refleja este procedimiento:

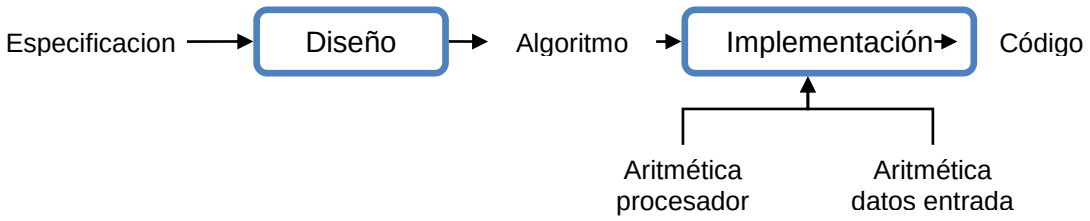


Figura 93. Metodología de diseño y análisis de *overflow*

La metodología de análisis del *overflow* incluye los pasos siguientes:

1. Determinación del número de bits con que están codificados los datos de entrada (*n*). El formato suele ser coma fija, ya que estos datos suelen provenir de un conversor A/D o similar.

2. Determinación del número de bits (*N*) y formato de la aritmética. En función del formato habrá que detectar los casos (operaciones) en las que se puede producir *overflow,* según la tabla vista en el apartado 1.4.

3. Análisis del algoritmo a implementar. Los algoritmos de procesado digital de la señal que suelen ser objeto de implementación en el ámbito de estudio de esta asignatura suelen ser algoritmos iterativos que repiten un número determinado de veces una serie de operaciones básicas. El análisis del algoritmo consiste en reducirlo a una secuencia de operaciones básicas que a continuación habrá que caracterizar desde una vertiente aritmética como una caja negra.

Esta caracterización se realizará analizando en el peor caso de los datos de entrada (máximo número posible), el número de bits que se generan en la salida.

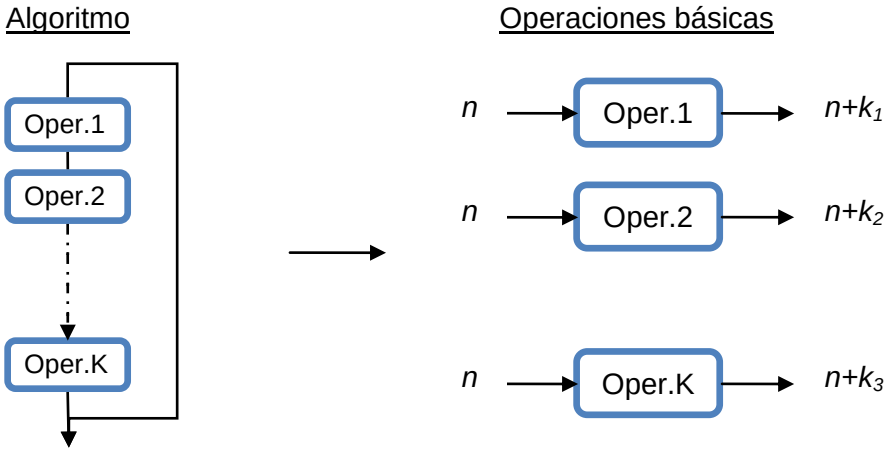


Figura 94. Descomposición en operaciones básicas

Aunque el diagrama anterior supone un caso genérico con un número de operaciones básicas cualquier (*k*), por algoritmos sencillos el valor de *k* ser á uno o dos.

Una operación básica no significa necesariamente una suma o bien una multiplicación. Por ejemplo, en el algoritmo de la FFT podemos identificar una operación básica a una mariposa.

Por un mismo algoritmo no hay una única solución de descomposición en operaciones básicas. De hecho, cuantas más operaciones básicas descomponemos el algoritmo, peores serán los resultados que se derivaran del análisis

4. Dada la medida de los datos de entrada, la secuencia iterada de operaciones básicas y la aritmética del procesador habrá que ver si hay o no riesgo de *overflow*. En caso de que exista se tendrá que evaluar la necesidad de implementar alguna técnica de escalado. Veremos tres:

   a. Escalado incondicional
   b. Escalado incondicional por fase
   c. Escalado condicional por fase

A continuación aplicaremos esta metodología a un caso concreto que además nos servirá para explicar los tres tipos de escalado nombrados.

❖ Aplicación en la FFT.

Para poder desarrollar la metodología expuesta en el punto anterior tendremos que fijar unas condiciones de trabajo, analizar el algoritmo y aplicar una técnica de escalado, si hace falta.

1. Formato de los datos de entrada: conversor A/D de 16 bits (coma fija fraccional)

2. Aritmética del procesador: coma fija fraccional de 24 bits. De acuerdo con el cuadro anterior habrá que vigilar con las sumas.

3. Algoritmo: FFT-256. La figura siguiente es un diagrama de bloques de la FFT-256:

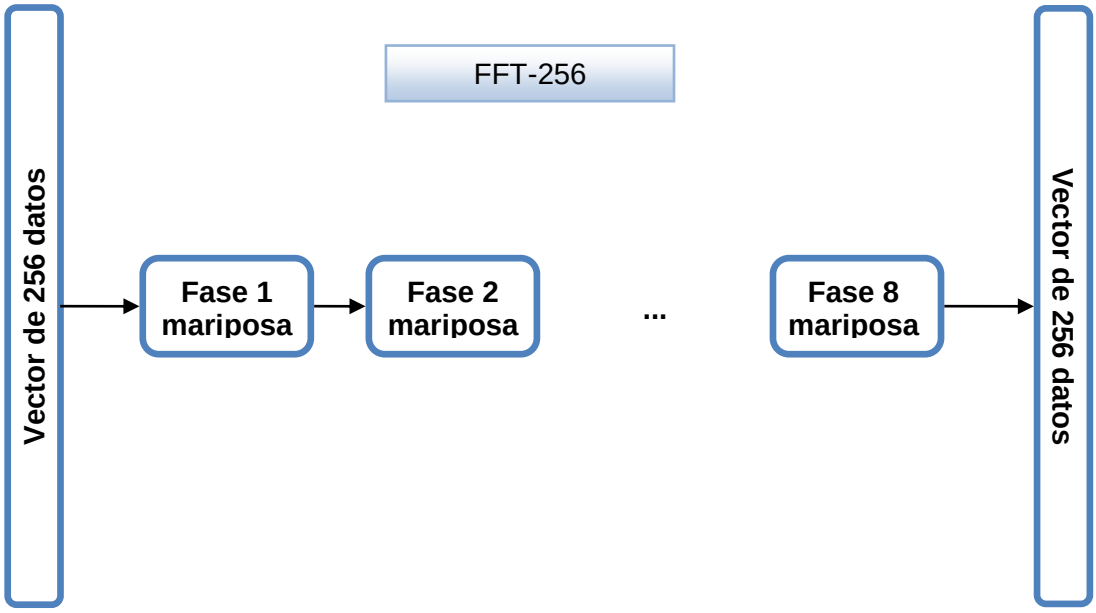


Figura 95. FFT-256

Démonos cuenta de que cada dada del vector de entradas pasará a través de 8 mariposas. Por lo tanto, desde el punto de vista de cada dato, podemos reducir la FFT-256 en la ejecución consecutiva de 8 estructures mariposa.

A continuación procederemos al estudio de las ecuaciones de una mariposa para determinar el posible incremento en el número de bits de la salida respecto la entrada.

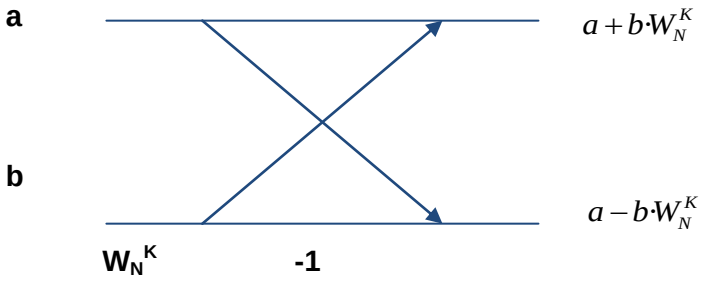


Figura 96. Diagrama bloques de una mariposa

Dado que los valores de *a*, *b* y W_N^K son complejos en el casi general, habrá que desarrollar las ecuaciones de salida en función de las entradas:

$$a+b\cdot W_N^K = \left(a_R + \left(b_R\cdot W_{N\ R}^K - b_I\cdot W_{N\ I}^K\right)\right) + j\left(a_I + \left(b_R\cdot W_{N\ I}^K + b_I\cdot W_{N\ R}^K\right)\right)$$
$$a-b\cdot W_N^K = \left(a_R - \left(b_R\cdot W_{N\ R}^K - b_I\cdot W_{N\ I}^K\right)\right) + j\left(a_I - \left(b_R\cdot W_{N\ I}^K + b_I\cdot W_{N\ R}^K\right)\right)$$

Ecuación 48. Ecuaciones de una mariposa

Las dos salidas tienen la misma estructura de operaciones; por tanto, será suficiente analizar una de las dos. Es más, la parte real e imaginaria también tienen la misma secuencia de operaciones; bastará con estudiar sólo una. Fijémonos, pues, en que sólo en la parte real de $a+b\cdot W_N^K$.

Recordemos que como trabajamos con coma fija fraccional, los problemas los podemos tener al sumar. Fijémonos en el diagrama siguiente:

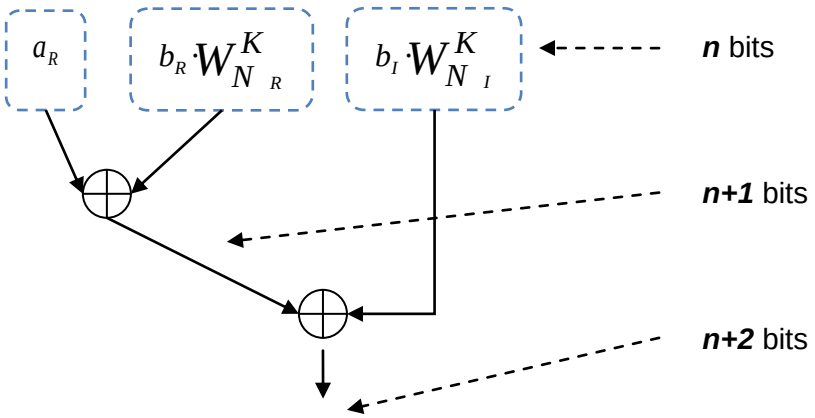


Figura 97. Análisis aritmético de una mariposa

A considerar los puntos siguientes:

a. Hemos sustituido la resta por una suma: los efectos producidos por ambas operaciones son idénticos (sólo varia el valor del peor caso)

b. La suma de los dos primeros términos requiere un bit más. La suma de este resultado de *n+1* bits (procedente de la suma de dos peores casos de *n* bits) con un peor caso de *n* bits requiere también de un bit más. Se deja como ejercicio demostrarlo.

Así pues, des de un punto de vista aritmética podemos caracterizar la mariposa como:

Figura 98. Mariposa des de un punto de vista aritmético

4. A continuación será necesario juntarlo todo. Fijémonos en que en el punto anterior hemos deducido que una mariposa, en el peor caso, puede provocar un aumento de dos bits en la codificación de los datos. Teniendo en cuenta los puntos siguientes:

    a. Datos de entrada codificados con 16 bits

    b. Cada dada ejecuta 8 mariposas

    c. Cada mariposa requiere, en el peor caso, de 2 bits extras

    d. El procesador es de 24 bits

    Llegamos a la conclusión que:

$$16 + 8 * 2 = 32 \geq 24$$

    i, por tanto, la ejecución de la FFT sobre el procesador en cuestión puede provocar *overflows*.

    Para combatir el *overflow* existen las técnicas de escalado.

El escalado consiste en la división de alguno de los datos que intervienen como operandos origen en una operación para que la ejecución de esta no provoque *overflow*.

Veámoslo en la figura siguiente:

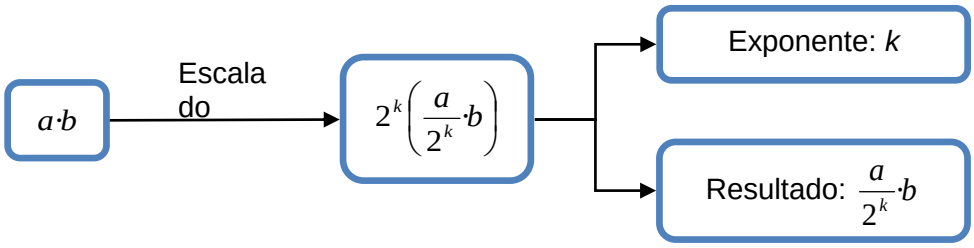


Figura 99. Descripción del proceso de escalado

El escalado consiste, pues, en implementar un exponente para *software*; implementar una especie de coma flotante para *software*. Los factores de escalado (2^k en la figura) suelen ser potencias de dos ya que las divisiones se pueden implementar con un coste muy bajo mediante desplazamientos.

Una vez razonado que en el ejemplo que estamos estudiando requerimos escalado, y visto en qué consiste éste, explicaremos a continuación tres estrategias o técnicas que se pueden adoptar.

a. **Escalado incondicional**. El escalado incondicional parte de los supuestos siguientes:

   i. Consideraremos el peor caso de datos de entrada (*incondicional*)
   ii. Analizaremos el algoritmo como un todo. Escalaremos sólo una vez, al principio del algoritmo.

   El razonamiento es el siguiente: los resultados de las operaciones ejecutadas a cualquier algoritmo no pueden tener más de 24 bits. Dado que la FFT-256 requiere la ejecución consecutiva de 8 mariposas, y cada mariposa requiere 2 bits extras, en total necesitamos 16 bits extras (bits de guarda) para no tener riesgo de *overflow*.

   Los datos de entrada no podrán estar, pues, codificadas con más de $24-16=8$ bits. Como el conversor A/D suministra 16 bits, habrá que escalar 8 bits la entrada. Disminuimos, pues, la resolución y el margen dinámico de entrada.

Veámoslo en la figura siguiente:

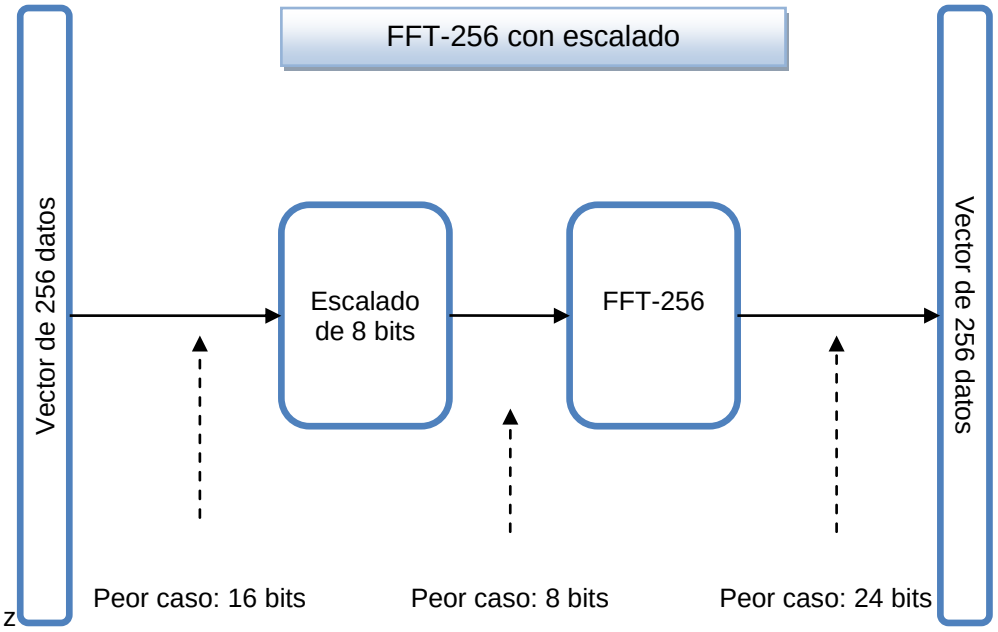


Figura 100. FFT-256 con escalado incondicional

Este método tiene como principal ventaja la sencillez y bajo coste de ejecución. El inconveniente más importante es la pérdida de margen dinámico y resolución de entrada que puede comportar el no cumplimiento de las especificaciones del sistema.

b. **Escalado incondicional por fase**. Los supuestos, en este caso, son:

   i. Consideremos el peor caso de los datos de entrada.

   ii. Dividimos el algoritmo en fases y nos planteamos la necesidad de l escalado fase a fase.

   El método aplicado al ejemplo seguiría el procedimiento siguiente. La FFT-256 se puede dividir en 8 fase de una mariposa. Dado que les datos de entrada están codificados con 16 bits y disponemos de 8 bits de guarda ($24-16=8$), las 4 primeras fases se pueden ejecutar sin riesgo de *overflow*. A partir de la quinta fase y hasta la octava necesitaremos escala dos bits en la entrada de cada mariposa para evitar *overflow*.

   Fijémonos en que este método requiere más coste de ejecución pero como contrapartida ofrece un mayor aprovechamiento del margen dinámico de entrada (figura siguiente). Esta manera de proceder nos permite definir un parámetro que

denominaremos resolución media de la entrada que pondera en cada fase el número de bits reales de la entrada que utiliza. En este caso,

$$R_{mitja} = \frac{16+16+16+16+14+12+10+8}{8} = 13.5 bits$$

Ecuación 49. Resolución media

Figura 101. FFT-256 con escalado incondicional por fase

c. **Escalado condicional por fase**. En este caso, los supuestos serán:

   i. No consideraremos los peores casos sino que miraremos los valores que nos llegan a tiempo de ejecución.
   ii. Dividimos el algoritmo en fases y nos planteamos la necesidad del escalado fase a fase.

Ahora, pues, no podremos determinar a priori los puntos en que se tendrá que escalar. Tendremos que comprobar antes de cada fase si es necesario escalar. Esta comprobación es dura al término mediante instrucciones (código) que verifiquen los valores de los bits altos de los datos de entrada (en nuestro caso, los dos bits más significativos).

Este método es normalmente el que requiere de un coste computacional más grande. En contrapartida ofrece el mayor aprovechamiento posible del margen dinámico de entrada.

Esquemáticamente:

Figura 102. FFT-256 con escalado condicional por fase

En este caso la resolución media no se puede calcular ya que depende de los valores de los datos de entrada.

Conviene comentar que en el algoritmo de la FFT cada fase no es sólo una mariposa, sino $N/2$ mariposas. Al aplicar un escalado condicional por fase, la decisión de escalar hay que tomarla si para cualquier dato de entrada de estas $N/2$ mariposas existe la necesidad de hacerlo. Es decir, en cada fase hay que escalar todas las entradas o bien ninguna. No podemos hacérselo a unas sí y a otras no ya que entonces daríamos más peso a unos valores que a otros.

Tal y como hemos visto, la aplicación de una u otra técnica de escalado se tendrá que llevar a cabo según un compromiso prestaciones / tiempo y coste de ejecución con la restricción de unas especificaciones de margen dinámico dadas.

4.4.6. El underflow

El *underflow* se produce cuando el resultad de una operación matemática tiene más resolución que la que permite la aritmética del procesador. Ponemos un ejemplo. Supongamos una aritmética de coma fija fraccional con signo de 8 bits tipo 1.7. Sabemos (punto 1.2.2) que existe riesgo de un*derflow* en el producto.
Veámoslo:

Format 1.7.

Resolución 2^{-7}		=	**0.0078125**
0.0000001	≡	2^{-7} =	**0.0078125**
0.0000001	≡	2^{-7} =	**0.0078125**
0.0000000000001	≡	2^{-14} =	**0.00006103 515625**
0.0000000	≡		0

underflow

Figura 103. Ejemplo del riesgo de un*derflow*

El *underflow* es, pues, un error pequeño que en la práctica se puede modelar como n ruido adicional al sistema. Una manera de entenderlo es considerar que una vez realizada la operación se cuantifica el resultado con el número de bits de la aritmética del procesador.

¿Cómo se manifiestan en la práctica los efectos del *underflow*? Lo analizaremos sobre un ejemplo: los filtros.

❖ Aplicación a los filtros

Los efectos del *underflow* en las aplicaciones basadas en filtros se manifiestan en dos momentos de la secuencia de diseño e implementación:

a. Cuantificación de los coeficientes (diseño). Efectivamente. En el momento del diseño del filtro se suele trabajar con coeficientes expresados en coma flotante. Estos coeficientes forman una respuesta impulsional con las características deseadas (pulso y ceros en los escogidos). Una vez tomamos estos coeficientes para incluirlos en el código implementado hay que expresarlos en el formato de la aritmética del procesador que, a menudo, es diferente. Los coeficientes, por tanto, varían de valor y en consecuencia la posición de pulso y ceros y respuesta del filtro serán diferentes a las deseadas.

Veámoslo con ecuaciones.
Respuesta impulsional diseñada:

$$H(z) = \frac{\sum_{k=0}^{M-1} a_k \cdot z^{-k}}{1 + \sum_{k=1}^{N-1} b_k \cdot z^{-k}}$$

Ceros

Pulso: $p_1, p_2, \ldots$

Figura 104. Respuesta Z de un filtro IIR

Cuantificación de los coeficientes:

$$a_k \to \overline{a_k} \Rightarrow \Delta a_k = a_k - \overline{a_k}$$
$$b_k \to \overline{b_k} \Rightarrow \Delta b_k = b_k - \overline{b_k} \Rightarrow p_i \to \overline{p_i} \Rightarrow \Delta p_i = p_i - \overline{p_i}$$

Esta variación de los coeficientes modificará la posición de pulso y ceros. Pongamos más énfasis en los pulsos ya que su variación es más crítica: estamos pensando en los próximos pulsos en el círculo unitario que pueden ir a parar fuera del círculo y afectar a la estabilidad del filtro.

La expresión siguiente nos indica la sensibilidad de la posición de los pulsos de un filtro en función de la variación de sus coeficientes:

$$\Delta p_i = -\sum_{K=1}^{n} \frac{p_i^{N-K}}{\prod_{\substack{l=1 \\ l \neq i}}^{N} (p_i - p_l)} \cdot \Delta a_k$$

Ecuación 50. Sensibilidad de los pulsos en la aritmética

Podemos ver que la variación de la posición de los pulsos depende de la variación de los coeficientes pero también de la distancia entre los pulsos del sistema. Es decir, si hemos diseñado un filtro cualquiera de orden *N* que contiene dos pulsos muy cercanos, su respuesta impulsional será muy sensible a la cuantificación de los coeficientes.

Por tanto si implementamos este mismo filtro de orden N como $N/2$ filtros de orden 2 en cascada, la respuesta frecuencial será idéntica en los dos casos pero la sensibilidad a variaciones de los coeficientes será menor en el segundo caso ya que los dos pulsos próximos que antes nos podían causar problemas ahora pertenecerán a dos filtros diferentes (y, evidentemente, unos filtros no afecten a los otros).

4.5. Gestión de la entrada / salida

En este documento comentaremos los mecanismos que el ADSP-2106x de *Analog Devices* pone a disposición del programador para gestionar de forma óptima la entrada / salida.

Recordemos que de acuerdo con el análisis visto en capítulos anteriores las instrucciones que codifican tareas de gestión de la entrada / salida pueden ser consideradas como parásitos y, por tanto, los procesadores DSP ofrecerán mecanismos para realizarles sin penalización de tiempo.

4.5.1. Introducción

Los sistemas que estudiamos son del tipo:

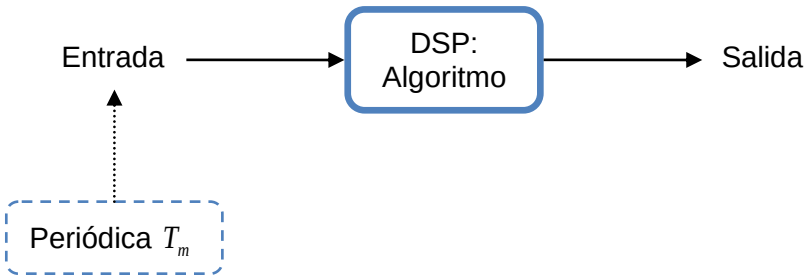


Figura 105. Diagrama de bloques I/O de un algoritmo

La comunicación de los nuevos datos en el DSP puede ser catalogada como rápida: el dispositivo generador de entradas avisa mediante una interrupción y seguidamente cede el dato sin tiempo de espera remarcables. La carga importante para el algoritmo no es el *hand-shaking* con el dispositivo sino la necesidad de entrar y salir de la RSI.

Según el comportamiento del algoritmo respecto a la entrada / salida podemos clasificarlos en los tipos siguientes:

- Algoritmos que generan una salida para cada nuevo dato de entrada. Por ejemplo, los filtros FIR. El hilo de ejecución de este tipo de algoritmos es:

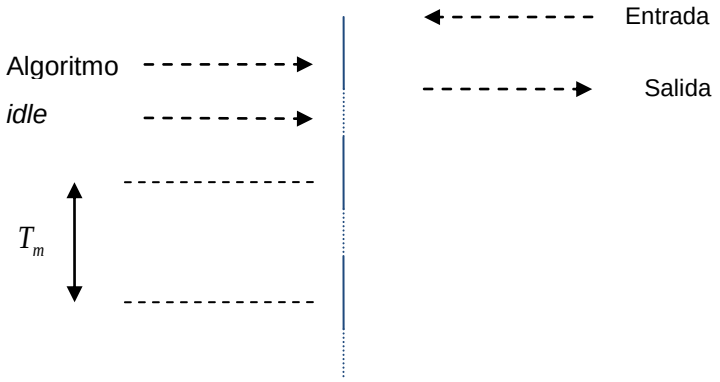
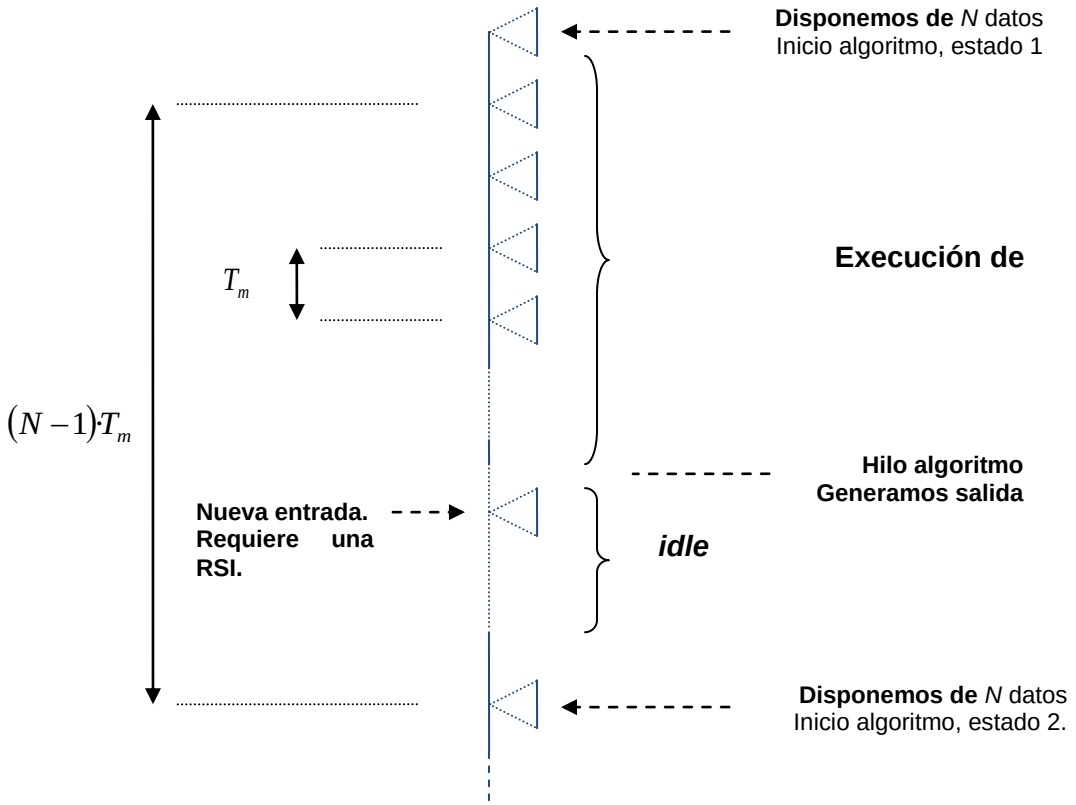


Figura 106. Hilo de ejecución de algoritmos que generan una salida por cada entrada

Cada nueva entrada se pide una RSI (Rutina de Servicio de Interrupción) que adquiere el dato del dispositivo, ejecuta un algoritmo y genera la salida. El tiempo que transcurre entre la generación de la salida y la nueva interrupción corresponderá a un tiempo de *idle*. La relación en tanto por ciento entre el tiempo de ejecución y período de muestreo corresponde al porcentaje de ocupación de la CPU.

En este tipo de algoritmos la entrada / salida no resulta una carga demasiado pesada, ya que el requerimiento de a la RSI es necesario de todas maneras.

- Algoritmos que se aplican por cada bloque de *N* datos de entrada. Por



ejemplo, la FFT. El hilo de ejecución de este tipo de algoritmos es:

Figura 107. Hilo de ejecución de los algoritmos que trabajan sobre bloques de datos

El tiempo de ejecución de los algoritmos que trabajan sobre bloques de datos suele ser grande en comparación con el periodo de muestreo. Esto implica que simultáneamente a la ejecución del algoritmo se reciben nuevos datos. Estet paralelismo se suele resolver mediante estructuras doble *buffer*: se definen dos estados:

- Estado 1, durante el cual se procesa el *buffer* 1 y se llena el *buffer* 2.
- Estado 2, durante el cual se procesa el *buffer* 2 y se llena el *buffer* 1.

La ejecución del algoritmo se lleva a cabo alternando los dos estados.
Así pues, en este segundo tipo de algoritmos la ejecución del proceso se ve interrumpida repetidamente por la recepción de nuevos datos. Una gestión óptima de la entrada / salida resultaría muy ventajosa ya que ahorraría los

requerimientos a la RSI. El objetivo sería generar una interrupción no por cada dato, sino cada *N* datos.

4.5.2. La DMA del ADSP-2106x

La DMA del ADSP-2106x releva al *core* del DSP de la gestión de la entrada / salida. Es capaz de realizar diversos tipos de transferencias, según el origen y el destino de los datos. Cada transferencia tiene asociado un canal:

- Memoria – memoria
- Memoria – puerto serie
- Memoria – *link port*
- Etc.

Los movimientos que más nos interesarán a nosotros por la tarea que nos ocupa son entre puerto serie – memoria (entrada) y memoria – puerto serie (salida). El interés deriva del hecho de que la mayoría de periféricos conectados a DSP lo hacen a través de puerto serie síncrono.

Hay cuatro canales relacionados con los puertos serie síncronos:

- Canal 0: puerto serie 0, recepción
- Canal 1: puerto serie 1, recepción
- Canal 2: puerto serie 0, transmisión
- Canal 3: puerto serie 1, transmisión

La DMA tiene dos modos de funcionamiento:

- Modo de transferencia única. Consiste en programar la DMA para realizar una única tarea, una sola vez.

- Modo de transferencia encadenada. Permite programar la DMA para realizar una o diversas tareas, repetidas veces.

El segundo modo de funcionamiento es muy útil para gestionar entradas / salidas periódicas ya que la tarea de recepción de datos se tiene que repetir cada período de muestreo.

Para programar un canal de la DMA asociado a un puerto serie síncrono hay que actuar sobre cuatro registros:

- *II* (*Internal Index*): dirección del bloque de memoria que participa en la transferencia (ya sea como origen o destino)

- *IM (Internal modifier)*: modificador de dirección en accesos consecutivos a la memoria

- *C (Count)*: medida de la transferencia

- *CP (Chain Pointer)*: para transferencias encadenadas, puntero en el TCB (*Transfer Control Bloquek*) que especifica el orden siguiente.

❖ Modo de transferencia única

En el modo de transferencia única sólo hay que programar los registros antes nombrados.

Por ejemplo, supongamos que queremos enviar por el puerto serie síncrono 0 un vector de 10 datos que tenemos en memoria, en un *buffer*. Habrá que programar el canal 2 de la manera siguiente:

Tal y como hemos comentado, la utilidad de este modo es muy limitada, ya que si quisiésemos utilizarlo para recibir datos periódicos tendríamos que reprogramar el canal cada vez.

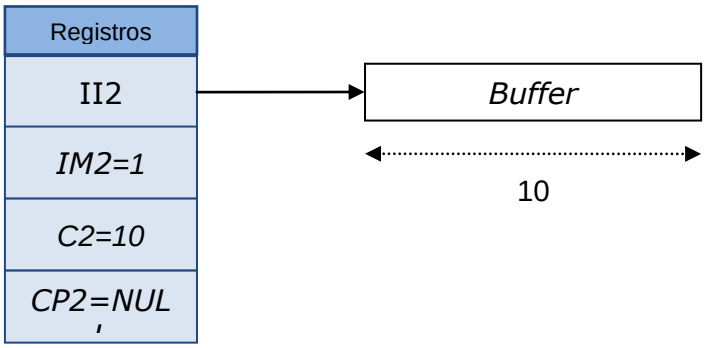


Figura 108. Programación de la DMA en el modo de transferencia única

❖ Modo de transferencia encadenada

En el modo de transferencia encadenada es posible programar de una sola vez uno o más órdenes diferentes que se ejecutarán una o diversas veces. Para hacerlo es necesario crear una estructura de datos dinámica basada en TCBs (*Transfer Control Bloquek*) encadenados. Los campos más importantes de un TCB son *II*, *IM*, *C* y *CP*; es decir, los necesarios para programar una DMA.

Por ejemplo, supongamos que queremos crear una estructura doble *buffer* para calcular la FFT-1024 a partir de los datos procedentes del puerto serie síncrono 0. Lo podríamos hacer de manera siguiente:

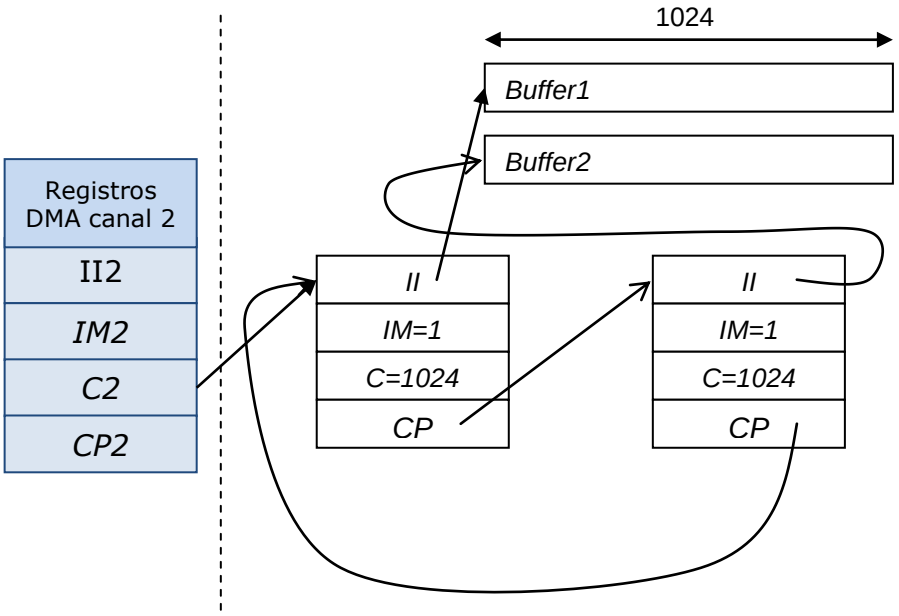


Figura 109. Programación de la DMA en el modo de transferencia encadenada

Al indicarle a la DMA que queremos utilizar el modo de transferencia encadenada (activando el bit SCHEN), se busca una estructura TCB apuntada por el CP (*Chain Pointer*) y se copia la información que en ella hay contenida sobre los registros de la DMA. Cuando se acaba la tarea de entrada / salida, se genera una interrupción para avisar (opcional), se busca el siguiente TCB apuntado por el CP y se procede así hasta que el CP vale NULO.

Este modo es muy útil ya que en el ejemplo de la figura la DMA ahorra 1023 interrupciones: sólo se genera una interrupción que indica la recepción de un bloque completo de 1024 muestras.

4.6. Sistemas multi-procesador

Algunas aplicaciones, normalmente a causa de la gran capacidad computacional que comportan, no pueden ser ejecutadas sobre un único procesador. En estos casos hay que del algoritmo en diversas entidades y ejecutar cada entidad en un procesador diferente. Aunque la segmentación se lleve a cabo siguiendo criterios de máxima autonomía de cada parte, será necesario un intercambio de información; es decir, interacción entre los diversos procesadores.

En estos casos es necesario un protocolo de compartición de datos de manera que se garanticen propiedades de exclusión mutua, justicia, versatilidad, etc. Y todo esto minimizando el número de ciclos utilizados para que se consuma el mínimo tiempo posible en la comunicación entre procesadores.
En este documento introduciremos diversos esquemas de compartición de datos entre procesadores. Estos esquemas presentan características complementarias respecto versatilidad, de manera que según la aplicación que se desee implementar, unos serán más adecuados que otros. Veremos también cómo estos diferentes esquemas se pueden utilizar simultáneamente para obtener las ventajas de todos ellos a la vez.

Los mecanismos que explicaremos son los que ofrece el DSP ADSP-2106x de *Analog Devices*. Será un ejemplo más de cómo los procesadores DSP incorporan *hardware* que ejecuta eficientemente tareas parásitas del algoritmo.

4.6.1. Comunicaciones punto a punto

Existe un tipo de algoritmos tales que una vez partidos sólo existe comunicación entre sub-bloques circularmente consecutivos de manera que el bloque i requiere acceso de lectura al $i-1$ y de escritura al $i+1$. En estos casos no suele haber problemas de exclusión mutua ni necesidad de árbitros. La ejecución de estos algoritmos en entornos multiprocesador resulta óptima si se utilizan esquemas de compartición de datos punto a punto del tipo *data-flow* tal y como veremos más adelante.

Por ejemplo, podríamos pensar en ejecutar una $FFT-N$ en un entorno multiprocesador. La forma más sencilla de particionar la aplicación sería pensar en ejecutar cada fase de la FFT sobre un procesador diferente. En total tendríamos $\log_2(N)$ fases. Fijaos en que el resultado de la primera fase es la entrada de la segunda fase y así sucesivamente. En figura 110 podemos ver un esquema de la solución propuesta.

Para implementar la comunicación punto a punto entre procesadores, el ADSP-2106x ofrece dos alternativas: puerto serie síncrono o *link port*. El puerto serie síncrono es un puerto bidireccional que permite transferir información serie. Destaca por la elevada velocidad en comparación con los asíncronos (posible gracias a la transmisión del reloj) y para la necesidad de interconectar pocos pins (ya que la transmisión es bit a bit).

El *link port* permite transferencias de cuatro bits en cada ciclo de reloj. Es, por tanto, más rápido que el puerto serie síncrono pero requiere interconexión de más líneas.

El principal inconveniente de las comunicaciones punto a punto es la poca versatilidad que presentan. Si, por ejemplo, el proceso i requiriese ocasionalmente comunicarse con el $i+2$ tendría que hacerlo a través del procesador $i+1$, con la consiguiente lentitud en la comunicación y pérdida de ciclos de reloj en $i+1$.

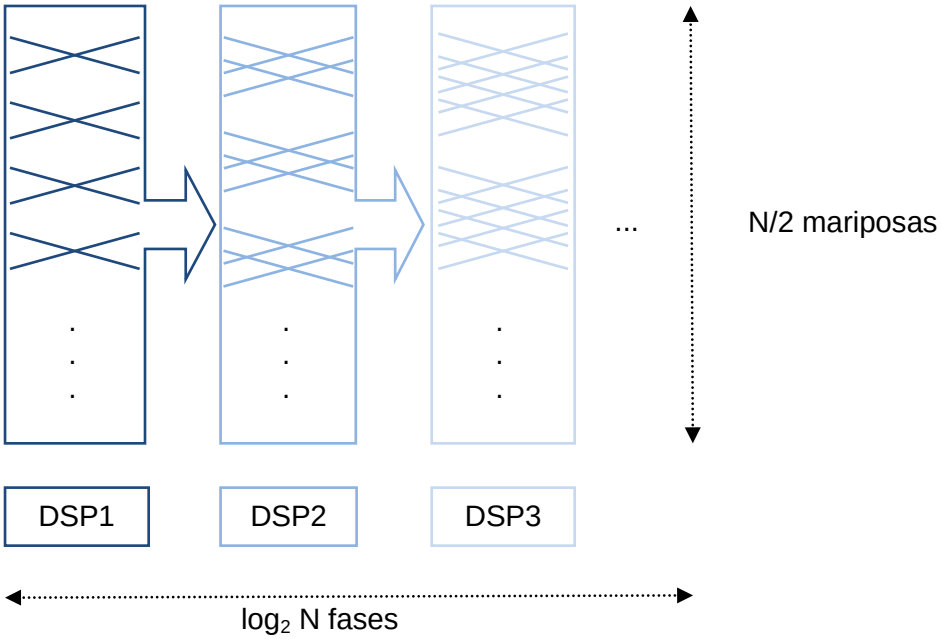


Figura 110. Distribución de una aplicación en un sistema *data-flow*

4.6.2. Compartición de bus

Cuando la aplicación que pretendemos segmentar no presenta una comunicación entre procesos tal y como la descrita en el apartado anterior, el esquema *data-flow* resulta poco adecuado (aunque la implementación también es posible si disponemos los procesadores en anillo *token ring*, con un identificador para cada uno de ellos y un protocolo que permita el paso de la información para aquellos procesadores que no son el destino de la petición).

En estos casos un esquema de compartición de bus suele ser más adecuado (para seguir con la analogía telemática, una compartición de bus correspondería con una red *Ethernet*).

En la figura 111 reproducimos un diagrama de bloques típico.

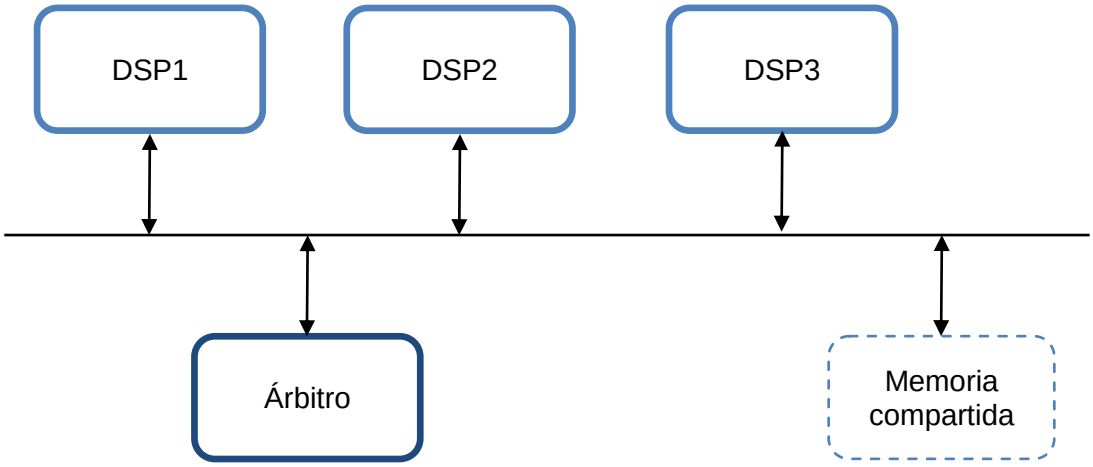


Figura 111. Compartición de bus

Tal y como su nombre indica, los diferentes procesadores involucrados en el sistema tendrán acceso de lectura y escritura a un bus compartido. Típicamente este bus estará formado por unas direcciones, unos datos, señales de control de acceso a dispositivo (*chip select*, *read*, *write*,...) y unas señales de control de acceso al bus (*bus request*, *bus acknowledge*, etc.).

Para garantizar que en cada momento sólo un procesador accede al bus es necesario un árbitro que distribuya los permisos y asegure exclusión mutua, justicia, etc. Además, para compartir datos a menudo es necesario una memoria externa a la cual todos los procesadores puedan acceder.

Este esquema presenta cómo sacar ventaja a la versatilidad: cualquier procesador se puede comunicar con cualquier otro. Los principales inconvenientes son la necesidad de una lógica externa (árbitro) y la incertidumbre en la duración de la comunicación (según el número de procesadores que en cada momento pretenden acceder al bus).

4.6.3. Compartición de bus basada en el ADSP-2106x

En este apartado describiremos los mecanismos que ofrece el procesador ADSP-2106x de Analog Devices para conectar diversos DSP siguiendo un esquema de compartición de bus de una forma altamente eficiente.

❖ Identificación de procesadores

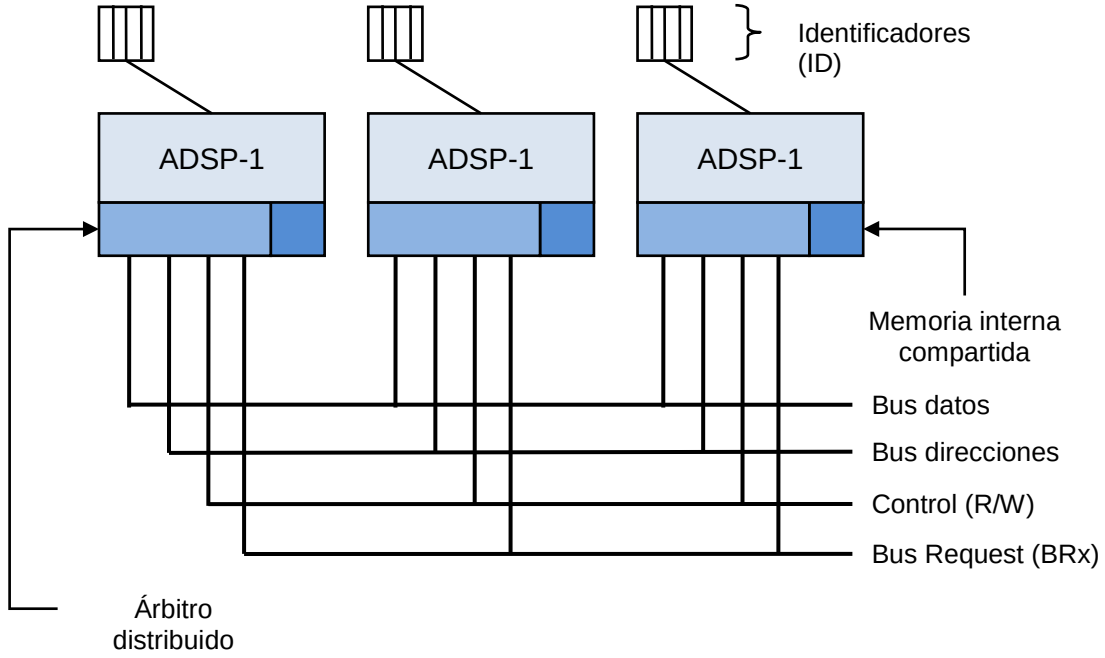
El esquema propuesto que describiremos a continuación permite la interconexión de hasta 6 procesadores ADSP-2106x. Con el, fin de

identificarse, cada DSP tendrá asignada una dirección única mediante los pins ID0-2. Los posibles valores que se pueden adoptar son entre 1 y 6. Si un DSP está sólo hay que asignarle el valor 1. Estas direcciones, como veremos más adelante, que tienen importancia a la hora de asignar prioridades de acceso y compartir datos a través de la memoria interna.

❖ Características del bus

En la figura 112 podemos ver un esquema de tres ADSP-2106x que comparte un bus.

Figura



112. Compartición de bus ADSP-2106x

El bus consta de cuatro categorías de señales:

- Bus de datos de 48 bits, correspondientes a la anchura de la memoria compartida (más adelante hablaremos sobre ello)
- Bus de direcciones de 32 bits.
- Señales de control de lectura y escritura
- Señales de gestión de la compartición del bus:

  - Petición de bus (BRx: *bus request*, uno por cada dirección posible entre 1 y 6). El procesador i puede activar su señal BRi y

escuchar el del resto. Es la única señal de esta categoría que se explicita en la figura 110.

- Tipos de prioridad (RPBA: *rotation priority bus arbitration*).
- Prioridad de acceso *core* inverso DMA (CPA: *core priority access*)

❖ Árbitro distribuido

En la figura 110 podemos ver que no hay un árbitro externo, sino que cada ADSP-2106x incluye un *hardware* de árbitro distribuido. Este es uno de las grandes ventajas de la implementación de una arquitectura de compartición de bus mediante procesadores ADSP-2106x.

La idea consiste en que a cada ciclo los procesadores que quieren acceder al bus activan su señal BR correspondiente. Dado un sistema de prioridades basado en la dirección de cada DSP y vistas las señales BR activados, todos los procesadores saben cuál de los procesadores solicitados tiene el derecho a acceder al bus en ese ciclo. Recordemos que cada procesador puede ver las señales BR de todos los otros.

❖ Compartición de memoria

En la figura 110 tampoco hay una memoria externa compartida porque los diferentes ADSP-2106x son capaces de compartir su memoria interna con el resto de procesadores. Esta característica es posible gracias al mapa de memoria del DSP que reproducimos en la figura 113.

Fijémonos en que, para cada ADSP, entre la dirección *0x00080000* y *0x0037FFFF*, en saltos de *0x00080000* hay mapeada la memoria interna de los procesadores con identificador entre $ID = 001$ y $ID = 110$. Por tanto, el procesador $ID = 010$, por ejemplo, puede acceder a la memoria interna del procesador con $ID = 001$ lanzando un acceso a memoria en el margen de direcciones entre *0x00080000* y *0x00100000.* Este acceso no dura necesariamente un ciclo de reloj ya que cal canalizarlo a través del bus multiprocesador (petición de bus mediante la activación de *BR2*, obtención del bus y acceso al DSP con $ID = 001$).

Un par de recortes más. Un procesador puede acceder a su propia memoria direccionando entre *0x00000000* y *0x0007FFFF* o bien a través del espacio multiprocesador. Además, un DSP puede escribir a una determinada dirección de todos los demás procesadores utilizando el área de memoria *broadcast* comprendida entre *0x00380000* y *0x003FFFFF*.

❖ Prioridades

Sólo un ADSP puede acceder al bus en cada ciclo. Si hay dos o más DSP que activan el *BR* en el mismo ciclo, ¿cuál de los dos tendrá prioridad de acceso? La respuesta a esta pregunta depende del valor del pin RPBA (*rotation priority bus arbitration*)

- Si $RPBA = 0$ se utiliza un esquema de prioridad fija. En este caso tiene prioridad de acceso el ADSP con el *ID* menor. Como ambos procesadores saben de las peticiones (*BR*) de los otros ADSP, en el ciclo siguiente sólo accederá al bus aquél DSP con el *ID* más bajo.

- Si $RPBA = 1$ se utiliza un esquema de prioridad rotativa. En el caso general, dada la numeración de direcciones desde 1 hasta 6 la prioridad en el próximo acceso se asigna al ADSP con el *ID* más próximo al siguiente del que tiene el bus en el acceso actual (donde entendemos que el siguiente en el sentido circular: el siguiente del 6 es el 1).

El modo de prioridad rotativa de media asigna la misma prioridad a todos los procesadores. Además, es puede asegurar que si un procesador pide el bus en un tiempo finito se le asignará. Esta propiedad no se puede asegurar en un sistema de prioridad fija ya que si el ADSP con $ID = 001$ quiere siempre el bus los otros ADSP nunca tendrá acceso.

En una aplicación determinada habrá que escoger uno u otro modo de prioridad según las necesidades: si todos los procesos son igualmente críticos habrá que escoger el sistema rotativo; si, en cambio, un ADSP ejecuta una aplicación mucho más crítica que los otros habrá que escoger un sistema fijo.

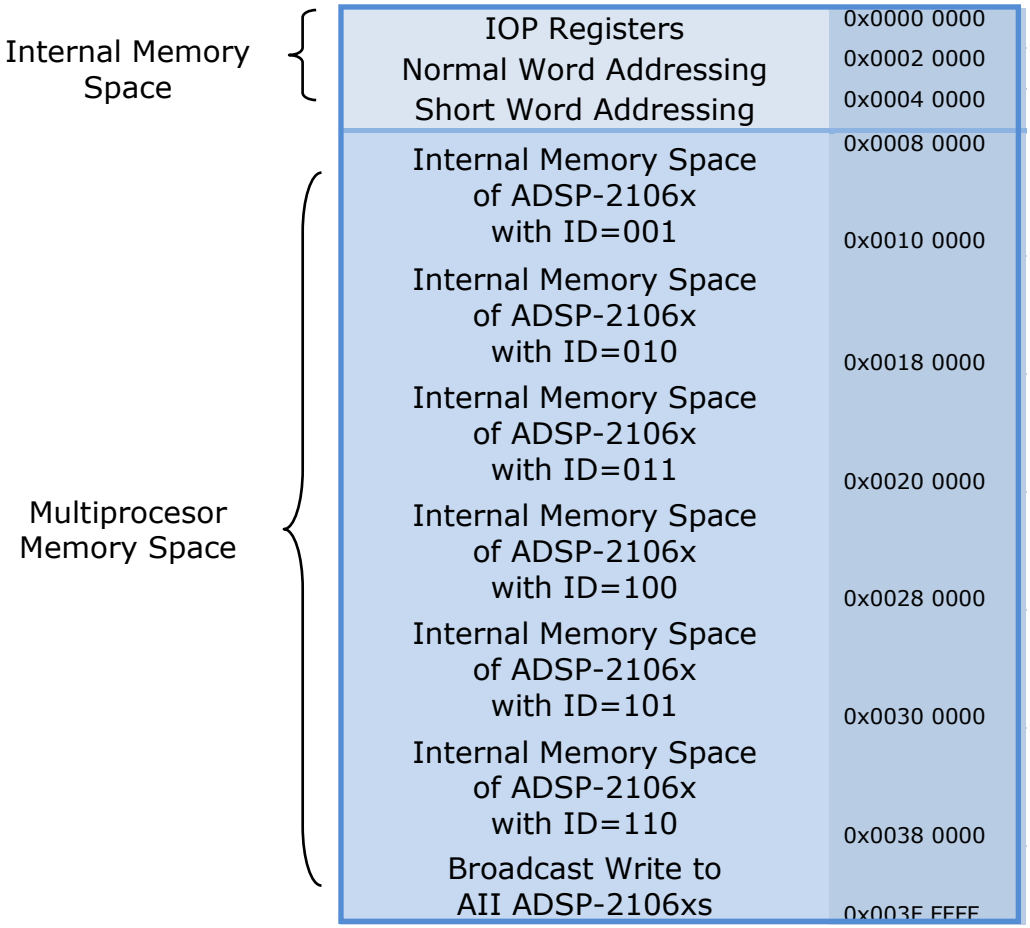


Figura 113. Mapa de memoria del ADSP-2106x

❖ Otros mecanismos

Tal y como hemos visto en el apartado anterior, el sistema de prioridad fija puede hacer que un procesador no pueda coger nunca el bus y se quede bloqueado para siempre a la espera del acceso (en la realidad esto difícilmente pasará; lo que sí puede pasar a menudo es que el procesador con *ID* más alto tarde muchos ciclos en poder acceder).

En este apartado explicaremos dos mecanismos que permiten poner límites al sistema de prioridad fija de manera que la mayor prioridad del

procesador con *ID* menor se pueda compaginar asegurando el acceso de los otros ADSP en un tiempo razonable.

- Mecanismo CPA (*core priority access*). Para utilizar este mecanismo todos los CPA de los ADSP estén conectados entre sí. El CPA es un pin que tiene dos posibles estados: alta impedancia y actividad (cero lógico). Un procesador activará el CPA cuando solicita el acceso al bus a causa de la ejecución de una instrucción, mientras que le dejará en alta impedancia si la petición provee de un acceso DMA. Un procesador más prioritario (con ID bajo) liberará el bus si su acceso es DMA y el pin CPA está activado (un procesador con ID mayor quiere hacer un acceso proveniente de código).

- Damos, por tanto, prioridad a accesos de código enfrente accesos DMA (el concepto subyacente a este mecanismo es que las transferencias de DMA suelen ser poco prioritarios).

4.7. Evaluación del rendimiento

En aplicaciones de media pequeña o mediana la metodología de diseño que más se utiliza es la tradicional: a partir de unas especificaciones se define un *hardware* (por ejemplo, un DSP y periféricos) que sea capaz de ejecutar a tiempo real el *software* previsto y a partir de aquí los dos caminos trabajan de una forma más o menos independiente hasta el momento en que se prueba y retoca el diseño final.

Esta aproximación requiere disponer de una forma de evaluar la potencia de cálculo de diferentes procesadores. Esta medida del rendimiento es la que nos ha de permitir determinar si ese *hardware* es o no adiente para nuestros propósitos. Ha de permitir, además, comparar diferentes procesadores entre sí.

Resulta obvio remarcar que la única manera segura de determinar si un procesador será capaz de ejecutar a tiempo real un software es probarlo (excepto casos triviales). Dado que el coste económico y temporal de este método es muy elevado, los fabricantes ofrecen una serie de índice que intenten aproximarse a este caso ideal pero que, como veremos, pequen de diversos inconvenientes. Estos índices son los MIPS, MOPS, MFLOPS y similares.

4.7.1. MIPS

MIPS son las siglas de *Million Instructions per Second*. Mesuren, tal como indica su nombre:

$$MIPS = \frac{Instruccions\ executades\ en\ un\ segon}{10^6}$$

Ecuación 51. MIPS

Este indicador, que a primera vista parecería bueno, presenta el inconveniente de ser una medida que no permite comparar procesadores que tengan juegos de instrucciones diferentes. Por ejemplo, si pretenden comparar un procesador CISC con un de RISC, la mesura saldrá muy favorable al procesador RISC ya que éstas, típicamente, tienen un juego de instrucciones sencillas que se ejecutan muy rápido y por contra los CISC tienen instrucciones muy potentes que se ejecutan lentas. La medida no implicará que en un caso general el RISC sea mejor que el CISC.

En el caso de procesadores con instrucciones que tengan tiempo de ejecución diferentes, el fabricante a la hora de hacer la medida de los MIPS utiliza las instrucciones con menor tiempo de ejecución para obtener una medida de MIPS máxima. Por tanto, la comparación del número de

instrucciones del algoritmo propio con el número de MIPS del procesador tampoco nos asegura que la ejecución a tiempo real sea posible ya que nuestro algoritmo no sólo utilizará instrucciones con tiempo de ejecución mínimo.
Los MIPS, en general, tienen poca utilidad. Relacionando esta cantidad con la frecuencia de reloj podemos saber el tiempo de ejecución de la instrucción más rápida. Por ejemplo, un procesador de 40 Mhz y 40 MIPS ejecuta la instrucción más rápida con un ciclo de reloj, mientras que una de 40 Mhz y 20 MIPS necesita dos.

4.7.2. MOPS

MOPS (*Million Operation per Second*) es un indicativo del número de operaciones que un procesador es capaz de realizar por segundo. Por operación se entiende no sólo las matemáticas, sino también los accesos a memoria, accesos DMA, etc.

El fabricante acostumbra a medir los MOPS ejecutando un programa que repite de forma indefinida la instrucción que optimiza el compromiso $operacions / temps\ d'execució$ y paralelamente a esta ejecución todos los periféricos (DMA, puertos serie, etc.) funcionando a pleno rendimiento.

La medida es, por lo tanto, de utilidad limitada ya que una aplicación convencional utilizará instrucciones que ejecuten menos operaciones y no todos los periféricos funcionaran simultáneamente. Los MOPS si que dan, no obstante, una medida de la complejidad de la arquitectura: un procesador con mucho más MOPS que MIPS indicará que tiene el *hardware* necesario para ejecutar muchas operaciones en paralelo.

Por ejemplo, la versión de 80 Mhz del TMS320C3x tiene 40 MIPS y 440 MOPS. De esta lectura en pueden extraer las conclusiones siguientes:

- La instrucción más rápida del TMS320C3x tarda 2 ciclos de reloj a ejecutarse. De hecho, este es el tiempo que tardan todas las instrucciones.

- La instrucción más potente del TMS320C3x permite la ejecución de 11 operaciones (incluidas operaciones matemáticas, accesos a memoria, transferencias de DMA y puertos serie, etc.)

4.7.3. MFLOPS

MFLOPS (*Million Floating-Point Operations per Second*) mide el número de operaciones matemáticas en coma flotante que es capaz de ejecutar un determinado procesador en un segundo. Démonos cuenta de que las operaciones en coma flotante están siempre asociadas a instrucciones propias del algoritmo. Por tanto los MFLOPS son en cierta medida comparables al cálculo del número de operaciones matemáticas que requiere un algoritmo, cálculo que se obtiene en la fase de diseño de éste.

Por ejemplo, una *FFT-N* ejecutada sin encabalgamiento sobre los datos de entrada provenientes de un conversor A/D de frecuencia de muestreo f_m requiere $5 \cdot f_m \cdot \log_2(N)$ operaciones matemáticas por segundo. Esta medida ha de ser siempre inferior al número de MFLOPS del procesador que tenga que ejecutar esta aplicación en tiempo real. En principio no hay que tener en cuenta las operaciones parásitas ya que la medida de los MFLOPS tampoco las considera.

El único inconveniente de esta lectura es el hecho de que el fabricante da esta medida utilizando siempre la instrucción que más operaciones en coma flotante ejecuta. Es decir, en el mejor caso para el... Nuestro algoritmo seguro que requerirá otras instrucciones que ejecutan menos operaciones en coma flotante.

4.7.4. Benchmark

Los *benchmarks* son medidas sobre el tiempo de ejecución de aplicaciones típicas. Es decir, no consideremos aplicaciones irreales sino que nos fijemos en algoritmos como ahora *FIR tap*, FFT-*1024* y una división. El fabricante codifica estos algoritmos sobre su procesador de la manera más óptima posible y seguidamente mide el tiempo de ejecución.

5. PLD

5.1. Introducción

5.1.1. El compromiso flexibilidad – eficiencia

Un factor importante a tener en cuenta a la hora de escoger dispositivos computerizados es la flexibilidad enfrente de la eficiencia. Flexibilidad es un concepto relacionado con la programabilidad del dispositivo, la capacidad que tiene de ejecutar tareas diferentes. La eficiencia la evaluaremos en términos de consumo, área de silicio ocupada y frecuencia de reloj máxima a la que puede trabajar
En un extremo de este compromiso están los ASIC (*Application Specific Integrated Circuit*), que contienen una circuitería especialmente diseñada y optimizada para realizar una determinada tarea. Esta especialización favorece que los ASIC presenten el mínimo consumo y la máxima frecuencia de reloj posibles por aquella aplicación, con el mínimo área de silicio ocupada. Por contra sólo pueden llevar a cabo el algoritmo por el cual han estado diseñados; si el algoritmo se ha de mejorar o queda obsoleto, el dispositivo se ha de tirar y rehacerlo ya que la pastilla de silicio de un ASIC no se puede reprogramar. Justamente por este motivo, los costes de desarrollo son grandes en tiempo y dinero.

Los ASIC son, pues, muy eficientes pero poco flexibles. Los DDC y DUC son ejemplos de ASIC: los dos dispositivos están diseñados para mezclar, filtrar (CIC y FIR) y delmar o interpolar (respectivamente). Cada proceso es configurable: frecuencia del oscilador local, factor de delmación / interpolación de cada etapa y coeficientes del filtro FIR entre otros parámetros. La programabilidad no va, no obstante, más allá: no es posible cambiar de forma substancial el comportamiento de un ASIC.

En el otro extremo del compromiso flexibilidad – eficiencia tenemos los los procesadores en general y los DSP (*Digital Signal Procesor*) en particular. Pueden ejecutar un número determinado y limitado de operaciones aritméticas y de control en cualquier orden, de forma secuencial. Cuando las operaciones requeridas por el algoritmo no se corresponden directamente con el juego de instrucciones del procesador o bien cuando el algoritmo permite elevados grados de paralelismo, estos dispositivos son poco eficientes y ofrecen un rendimiento pobre. Los DSP son, pues, muy flexibles pero poco eficientes. Es importante notar que la flexibilidad de un procesador proviene del software: posibilidad que tiene el programador de escoger las instrucciones y el orden en que se ejecutaran en el algoritmo. El hardware del DSP (número de multiplicadores, sumadores, memoria, buses, etc.) viene fijado de fábrica.

En un punto medio entre ASICs y DSPs están los dispositivos de lógica programable (PLD: *Programmable Logic Device*). Están formados por un

hardware flexible que puede estructurarse de acuerdo con la organización y flujo de datos de un algoritmo concreto. En cada caso la configurabilidad de estos dispositivos permite ejecutar las operaciones precisas que el algoritmo requiera. El diseñador tiene un control total del proceso ya que la reprogramabilidad del dispositivo no requiere ningún proceso de fabricación. Un PLD ve de fábrica con unos recursos hardware determinados: bloques de funcionalidad muy genérica (multiplexores, biestables, etc.) y redes de interconexión entre estos bloques. El programador, a partir de estos bloques sencillos, tiene la capacidad de formar estructuras hardware de más alto nivel (multiplicadores, memoria, filtros, etc.) según las necesidades del algoritmo. Por tanto, la flexibilidad de un PLD ve determinada por la posibilidad de reconfigurar el hardware.

Uno de los dispositivos de lógica programable que más ha contribuido a el éxito de esta familia es la FPGA (*Field Programmable Gate Array*). En un principio las FPGA se utilizaban sólo en la fase de prototipage de un ASIC aprovechando la ventaja que en estos momentos del diseño representa la posibilidad de ser reprogramado. No se pensaba en la FPGA como dispositivo comercial a causa del elevado consumo y a la baja velocidad de reloj que permitía. Avances tecnológicos han minimizado estas desventajas y actualmente se considera que estas dispositivos son una alternativa eficiente para algoritmos que permitan segmentación y paralelismo y trabajen preferentemente en aritmética de coma fija.

En el futuro se considera que los procesadores (DSP) continuaran siendo la mejor alternativa para implementar algoritmos complicados en el sentido de requerir muchas construcciones del tipo *if – then – lose*, o bien cuando los requerimientos computacionales de estos algoritmos permiten la ejecución directa y óptima sobre el juego de instrucciones del procesador. Hay que remarcar que una ventaja importante de los sistemas basados en procesadores es la facilidad de desarrollo de la aplicación y el menor *time – to – market* de las tres opciones consideradas en este apartado.

Hay otros casos en los que el uso de DSP no es adecuado. Cuando la complejidad computacional o las frecuencias de reloj son muy elevadas o bien existe una importante limitación en el consumo de potencia o bien se han de fabricar largas tiradas del producto, el uso de ASIC o FPGA suele ser más conveniente.
Los ASIC suelen ser más rápidos y consumir menos que las FPGA. Además, los ASIC suelen presentar una densidad de integración mayor.
Por contra, el tiempo y coste de desarrollo son muy menores en un sistema basado en FPGA que en ASIC. Además, les FPGA permiten cambios tanto durante la fase de diseño como actualizaciones futuras.

Con el tiempo se observa que las mejorías tecnológicas están minimizando los desventajas de les FPGA enfrente de los ASIC, al mismo tiempo que la situación del mercado tecnológico de consumo que requiere rápidos desarrollos y actualizaciones continúas encuentra en la lógica programable

un campo de trabajo muy adecuado. El diseño de ASIC parece quedar relegados a aquellos productos de mucha tirada en que un ahorro de b, consumo o precio final unitario justifique el largo y costes desarrollo.

5.1.2. ... y el consumo de potencia

Un aspecto importante a remarcar es la correlación directa que existe entre la programabilidad (flexibilidad) y el consumo de potencia. Diversos estudios han demostrado que la conmutación de una puerta *nand* en una FPGA consume entre 7 y 10 veces más que en un ASIC. No es necesario decir que la ejecución de una instrucción *nand* en un DSP requiere un consumo mucho más elevado aún. La programabilidad comporta consumo de potencia adicional.

El consumo de potencia es especialmente crítico en el diseño de aplicaciones móviles en que alargar la autonomía de la batería es un aspecto clave. Justamente la telefonía móvil requiere ahorro energético y máxima configurabilidad a la vez, ya que es uno de los ámbitos donde la adaptabilidad y reconfigurabilidad son más necesarias. En un sistema de radio real hay que utilizar una combinación adecuada de dispositivos para reducir el consumo de potencia.

En el ámbito de los DSP es donde más esfuerzos se hacen para fomentar el ahorro energético. Reducir la complejidad excesiva de las instrucciones, utilizar preferentemente dispositivos internos a externos (por ejemplo, las memorias), emplear diversos modos de bajo consumo que desactiven diferentes partes del dispositivo, reducir la tensión de alimentación, etc. Son estrategias que hay que tener en cuenta para tal limitar el consumo energética de los diseños.

5.1.3. PLD in the software radio.

Con el término *software radio* hacemos referencia a la nueva tecnología que permite diseñar de forma eficiente sistemas de radio flexibles, multi-servicio, multi-estándar, reconfigurables y reprogramables para software. En gran medida, hablar de *software radio* es parlar de telefonía móvil. Desde su aparición comercial hace más de veinte años se han desarrollado estándares cada vez más complejos que ofrecen una creciente oferta de funcionalidad. Las sucesivas generaciones no hacen desaparecer a las anteriores de forma inmediata, sino que durante un período de tiempo conviven.

Los teléfonos móviles se tienen que diseñar para que acepten diversos estándares y diversas bandas frecuenciales. Las necesidades de

reconfigurabilidad y adaptabilidad son actualmente un requisito de diseño imprescindible.

De forma simplificada, podríamos diferenciar cuatro tecnologías o combinaciones de tecnologías que tienen un papel relevante en este tipo de diseños.

En primer lugar la aparición de una nueva generación de procesadores DSP que operan a elevadas frecuencias de reloj, con una medida reducida y modos de funcionamiento de bajo consumo. No obstante, no pueden, resolver el diseño por sí solos ya que no son capaces de procesar de forma suficiente señales en bandas de Mhz. Incluso si tecnológicamente fuese viable hacerlo, el consumo y disipación de calor de estos dispositivos haría inútil cualquier esfuerzo en esta dirección.

Una segunda alternativa ya considerada desde hace tiempo es la combinación de un DSP con diversos ASIC, donde estos últimos implementan las funciones de más alta frecuencia propias de cada modo de funcionamiento. Es una solución que además de factible en lo que se refiere a velocidades de proceso, es aceptable desde un punto de vista de consumo energético. Sin embargo, tiene algunos inconvenientes. Para soportar diversos modos a menudo serán necesarios diversos ASIC que incrementarán la medida y consumo del diseño. Además, un sistema diseñado así es poco flexible para permitir nuevas bandas frecuenciales introducidas una vez el producto ya esté en el mercado. Si el número de modos y bandes a soportar es grande, la complejidad del sistema crece muy rápidamente.

La tercera posibilidad recibe el nombre de hardware parametrizado. Consiste en diseñar ASICs o procesadores para realizar una subtarea no asociada a un estándar en concreto, sino común a diferentes estándares. Por ejemplo, un ASIC que implemente un filtro FIR con coeficientes configurables o bien un procesador microprogramado en el que el microcódigo es diseño ad hoc para una función específica. El sistema final estará formado por estos dispositivos que se tendrán que reconfigurar según el modo de funcionamiento deseado.

El principal inconveniente que se han encontrado las compañías que trabajan con esta metodología es el elevado coste temporal de desarrollo del producto final, en gran parte causado por la dificultad de ajustar correctamente los diferentes módulos para que el diseño soporte la toda la funcionalidad deseada.

La cuarta alternativa es el uso de la lógica programable. Además de todas los ventajas ya nombradas que ofrecen estos dispositivos, la posibilidad que tienen de reprogramación dinámica permite que el área de silicio del diseño no tenga que ser directamente proporcional al número de modos de funcionamiento deseados: el dispositivo en cuestión puede reinicializarse en

cualquier momento a partir de una memoria para cambiar el su modo de operación, de manera que con la capacidad necesaria para soportar el modo más complejo no hay suficiente para soportarlos todos.

5.1.4. Tipo de PLD

Los primeros dispositivos no especializados que permiten implementar funciones lógicas son las memorias (PROM). Efectivamente, cualquier Ecuación lógica de *N* entradas y *M* salidas se puede ejecutar sobre una memoria con *N* líneas de direcciones y una medida de palabra de *M* bits. Será necesario, por lo tanto, una memoria de $M \cdot 2^N$ bits. Uno de los principales inconvenientes de esta implementación consiste en que la memoria descodifica todas las entradas para calcular cada salida, mientras que esto no es necesario en muchos casos. La medida de la memoria será a menudo ineficientemente gran.

Veámoslo en un ejemplo. Supongamos que hemos de calcular las ecuaciones lógicas siguientes:

$$y = a + b \cdot c$$
$$z = a \cdot b + c$$

Ecuación 52. Funciones lógicas

Tenemos tres entradas y dos salidas. Por los tanto necesitaremos una memoria de 8 posiciones de dos bits cada una (en total, pues, $2 \cdot 2^3 = 16$ *bits*). Supongamos que añadimos una nueva salida al sistema:

$$w = a \cdot c$$

Ecuación 53. Función lógica adicional

Si queremos usar la misma memoria de antes, habrá que incrementar su capacidad en 8 bits: de 16 a 24 ($3 \cdot 2^3 = 24$ *bits*). La implementación será poco eficiente ya que el cálculo de *w* no requiere descodificar el valor de *b*: para calcular *w* hay suficiente con consultar el valor de *a y c* y, por tanto, la mitad de la memoria adicional (4 posiciones de las 8 añadidas) es redundante.
Este método para implementar funciones lógicas es a menudo poco óptimo en términos de cantidad de lógica empleada. A continuación veremos otras arquitecturas especialmente diseñadas para este fin.

El primer dispositivo especialmente diseñado para implementar circuitos lógicos programables fue el PLA (*Programmable Logic Array*). Está formado por dos niveles de puertas lógicas: un primer nivel de puertas *and*

configurables y un segundo nivel de puertas *or* configurables. El dispositivo permite que cualquier combinación de los pins de entrada (o los negados) sea multiplicado (*and*) y que cualquier combinación de salida del plano de *and* sea sumado (*or*) para generar la salida final. Por tanto el dispositivo es muy adecuado para generar sumas de productos.

Los primeros PLA fueron fabricados por Philips a principio de los años 70. En aquel momento tenían como principal defecto la dificultad de fabricación (elevados costes) y la baja velocidad de funcionamiento, ambos problemas causados por la alta configurabilidad (a dos niveles: *and y or*) que el dispositivo ofrece.

Para solucionar este problema se desarrollaron las PAL (*Programmable Array Logic*), que reducían la configurabilidad al nivel de *and*; el nivel de *or* seria fijo. Esta reducción de prestaciones se compensa con una mayor oferta de dispositivos que se diferencian entre si por el número de pins de entrada / salida, el número de entradas de cada puerta *or*, etc. El producto final es más económico y más rápido.

Además los dispositivos PAL suelen disponer biestables a la salida de las puertas *or* para de permitir la implementación de sistemas secuenciales. Destaquemos por la gran popularidad que han conseguido los dispositivos de AMD 16R8 y 22V10.

El conjunto de dispositivos PLA, PAL y similares suelen recibir el nombre de SPLD (*Simple Programmable-Logic Devices*). En general (y sobretodo para los dispositivos con estructura PAL), sus principales características son el bajo coste y la elevada velocidad a la que pueden trabajar.

Con el tiempo se intentó incrementar la capacidad de los SPLD. Ahora bien, el crecimiento directo de la arquitectura tal y como estaba planteada estaba muy limitado ya que la complejidad del plano programable (*and* en les PAL) aumentaba exponencialmente con el número de entradas. Para mejorar las prestaciones se tomó una vía alternativa: combinar diversos SPLD dentro de uno mismo integrado mediante interconexiones programables. El tipo de dispositivos que utilizan esta filosofía reciben el nombre de CPLD (*Complex Programmable Logic Device*).

Algunos CPLD están formados de hasta50 SPLD. Altera fue el primer fabricante en aplicar esta idea. Los SPLD están interconectados con una red continua, tipo *cross connect*. A medida que el número de SPLD dentro de un CPLD se incrementa, la complejidad de la red aumenta muchísimo. El crecimiento mucho más allá de 50 SPLD dentro de un CPLD resultó ser también difícil.

Para obtener mayores densidades de integración se tomó un camino diferente. Las FPGA (*Field Programmable Gate Array*) están formadas por un grupo de celdas lógicas que pueden ser interconectadas entre sí. Las

celdas lógicas suelen ser más sencillas que las SPLD. Se pueden combinar entre sí a través de una matriz de interconexión programable.

5.1.5. Clasificación según la granularidad

En la línea iniciada en el apartado 1.4, una manera de distinguir y clasificar los dispositivos de lógica programable es según la medida de los bloques lógicos o celdas (granularidad). Hay dos tipos principales:

- Granularidad media. Corresponde a los dispositivos FPGA. El contenido del bloque lógico depende del fabricante. En general, suele contener LUT (*look-up table*: memoria de *n* entradas y una salida que permite implementar cualquier función lógica de *n* entradas con un retraso fix) o multiplexores. A la salida de la celda suele haber un biestable. Las entradas y salidas de los diferentes bloques se pueden combinar a través de una matriz jerarquizada de conexiones.

- Granularidad basta (*coarse*). Corresponde a los dispositivos CPLD. El bloque lógico es un SPLD: grupo de puertas *and* / *or*. Típicamente tienen entre 8 y 10 entradas, y 3 o 4 salidas. Estas entradas y salidas se pueden unir mediante un grupo de interconexión programable.

Existen también dispositivos de granularidad fina (denominados *sea of gates*) formados por un grupo de puertas *nand* interconectadas mediante capas de mecorte adicional. Es precisamente la conectividad la que limita el crecimiento de este tipo de arquitecturas, ya que el coste de unir las puertas es muy grande comparado con su funcionalidad.

5.1.6. Clasificación según la tecnología

La tecnología en que está implementado el dispositivo de lógica programable determina diversos factores:

- Admite sólo una programación o bien diversas
- Admite programación *in-circuit* o *out of circuit*.
- Programación volátil o no volátil

Destaquemos las tecnologías siguientes:

- *Fuse*. Tecnología bipolar que se utilizaba en los primeros dispositivos PLA. Es no volátil y sólo permite una programación. Está obsoleta.

- *EPROM*. Tecnología CMOS no volátil, reprogramable diversas veces *out of circuit* (requiere chorro ultraviolado para borrar el contenido y elevadas tensiones de alimentación para grabar).

- *EEPROM*. Tecnología CMOS no volátil, reprogramable diversas veces *in circuit*.

- SRAM. Tecnología CMOS volátil, reprogramable diversas veces a través de una memoria externa (normalmente serie). La telecarga se suele efectuar cada vez que se conecta la alimentación y suele tardar milisegundos.

- *Antifuse*. Tecnología CMOS no volátil y programable una sola vez. Una celda *Antifuse* tiene una medida y un orden de magnitud inferior a la SRAM.

Los dispositivos CPLD suelen ser EPROM, EEPROM o SRAM. Las FPGA suelen utilizar SRAM o *Antifuse*.

5.1.7. CPLD vs FPGA

A continuación daremos unos criterios generales para diferenciar los dispositivos CPLD y FPGA. Lo haremos desde tres puntos de vista:

- Red de interconexión
- Elementos lógicos
- Proceso de diseño

Hay que decir, no obstante, que los fabricantes tienden a mezclar las dos estructuras de manera que estas distinciones son cada vez menos claras.

En capítulos posteriores veremos otras diferencias y profundizaremos en éstas.

❖ Red de interconexión

Un CPLD está formado por diversos SPLD que pueden combinarse a través de redes de interconexión que reciben nombres diferentes según los fabricantes:

- *Programmable interconnect array*, para la familia MAX7000 de Altera.
- *Central switch matrix*, para la familia Mach d'AMD.
- *Global routing pool*, para las familias pLSI y ispLSI de Lattice.

- *Programmable interconnect matrix*, para las Flash370 de Cypress.
- Etc.

En todos los casos, las redes de interconexión son tipo *cross – connect* que permiten conectar de forma directa cualquier entrada / salida de los SPLD entre ellos o bien con pins externos. Este tipo de red asegura un retraso mínimo de propagación de la señal. Además, este retraso es constante sea cual sea el par de pins unidos (estén cerca o lejos). Una red de este estilo es posible porque el número de celdas (SPLDs) que contiene un CPLD es relativamente bajo (recordemos que son dispositivos de granularidad basta).

Dada la gran cantidad de celdas que suelen integrar les FPGA (granularidad media) no se pueden utilizar redes de interconexión directa como en los CPLD ya que la complejidad de estas redes sería enorme. En lugar de esto, una FPGA está formada un grupo bidimensional de celdas lógicas interconectadas por líneas horizontales y verticales de diferente longitud: desde aquéllas pensadas para conectar celdas adyacentes, hasta otras que van de punta a punta de pastilla (la jerarquía de longitudes varía según el fabricante y el dispositivo). Existen conmutadores programables que permiten unir las entradas o salidas de las celdas a las líneas o bien líneas entre si. Cualquier conexión entre dos celdas recorrerá tramos de diferentes longitudes y atravesará un número de conmutadores diferente. Por tanto, el retraso de propagación no es predecible a priori ya que depende del ruteado. Este es uno de los principales inconvenientes de las FPGA frente a las CPLD. La estructura de FPGA explicada en el párrafo anterior corresponde a los dispositivos de los fabricantes Xilinx y Actel.

Otros fabricantes (Altera, familia FLEX10k y AT&T, familia ORCA2) han propuesto soluciones de interconexión para FPGA a medio camino con las de los CPLD. Por ejemplo, en el caso de Altera, una pastilla está formada por diversos grupos de 8 bloques lógicos (celdas) agrupado. Cada grupo tiene interconexión local. Una red adicional (*FastTrack*) con líneas de longitud única permite la interconexión de bloques lógicos. Más adelante lo estudiaremos con más detalle.

❖ Elementos o bloques lógicos

En el caso de los CPLD, los bloques lógicos tienen una arquitectura de SPLD (PAL y similares). Habitualmente están formados por un grupo programable de puertas *and*, un grupo fijo de puertas *or* que a veces tienen un número de entradas variable (suele recibir el nombre de *product-term distributor* o *product-term alugarator*) y un conjunto de biestables de salida (denominado macrocelda excepto en el caso de Altera en el que macrocelda es el nombre que recibe el bloque lógico entero).

Algunos parámetros a tener en cuenta a la hora de escoger un dispositivo CPLD es el número de entradas y salidas y el número de biestables de cada bloque lógico. Por ejemplo, si queremos implementar un algoritmo que trabaje sobre valore de 8 bits conviene que bloque lógico tenga como mínimo 8 entradas, 8 salidas y 8 biestables. Si no es así el rendimiento del CPLD bajará mucho.
Les FPGA tienen una estructura de bloque lógico más sencilla (Xilinx la denomina CLB: *Configurable Logic Bloquek*). El primer sub-bloque de un CLB suele estar basado en LUTs (*Look- Up Table*) de *n* entradas y una salida que permiten implementar cualquier función lógica de *n* entradas, o bien ser utilizadas directamente como memorias de 2^n x 1 bit. A continuación multiplexores permiten dirigir la/es salida/es del la/es LUT/s hacia biestables.

❖ Proceso de diseño

Para diseñar y programar un dispositivo de los que estamos considerando se utilizan herramientas de CAD, ya sean específicas de cada fabricante o bien independientes que permitan trabajar con cualquiera de ellos.

El proceso de diseño es ligeramente diferente según nos dispongamos a utilizar un dispositivo tipo CPLD o bien FPGA. En el caso del CPLD, en general habrá que seguir los pasos siguientes:

- *Desing entry*. El diseñador determina el comportamiento del dispositivo, ya sea mediante esquemáticos, ecuaciones lógicas o lenguajes de alto nivel (VHDL, Verilog, ABEL, etc.). A partir de esta especificación las herramientas de CAD suelen permitir simular el comportamiento del diseño.

- *Logic optimization*. Hay que transformar la especificación del diseño en ecuaciones lógicas optimizadas.

- *Device fitting*. Consiste en determinar el hardware que se utilizará y las conexiones que habrá que activar

- *Simulation.* Una vez determinado el uso del hardware podremos simular el comportamiento temporal del diseño. Si no cumpliésemos con las especificaciones, habría que retocar el diseño o bien usar un dispositivo más potente.

- *Configuration.* Grabación del dispositivo.

Si escogemos FPGA, la mayoría de los pasos son análogos. Hay, no obstante, un incremento de complejidad en el *Device fitting*, que suele subdividirse en tres procesos:

- *Map.* Consiste en especificar las ecuaciones lógicas optimizadas en función de los recursos lógicos del dispositivo en cuestión (LUT y similares).

- *Placement.* Escoger los bloques lógicos concretos que se usarán para implementar el diseño.

- *Routing.* Determinar los caminos de conexión entre bloques lógicos.

De todos los pasos descritos, sólo el primer (*Design entry*) es a cargo del programador. El resto de procesos son automáticos, aunque pueden ser configurados introduciendo condicionantes y restricciones. En el caso de las FPGA, el tiempo de proceso de la fase de *routing* puede ser significativamente costoso en tiempo.

❖ Comparación. Ventajas e inconvenientes

De los CPLDs destacamos:

- Fácil migración de diseños realizados sobre SPLD

- Adquieren velocidades muy elevadas.

- El retraso de interconexión entre elementos lógicos es predecible. Esto facilita y acelera el diseño.

- Muy adecuados para implementar algoritmos que hagan un uso intensivo de *and y or y* no requieran un gran número de biestables (por ejemplo, máquinas de estados)

De les FPGAs destacamos:

- Mayor densidad de integración

- Mayor aprovechamiento de los recursos lógicos

La naturaleza de la red de interconexión provoca:

- Retrasos impredecibles que dependen del ruteado

- En diseños grandes, el ruteado es muy complejo y suele tardar mucho tiempo.

5.2. Estudio de mercado. FPGA de Altera

A continuación describiremos las familias de FPGA de Altera. Veremos que este fabricante se caracteriza por intentar aprovechar los beneficios de la distribución de elementos lógicos de capacidad media en un grupo bidimensional e interconectarlos por redes que presenten un retraso predecible. La elevada complejidad de estas redes reduce la densidad de integración de puertas lógicas en comparación con otros fabricantes.

Analizaremos cada familia desde diferentes puntos de vista:

- Jerarquía, estructura y densidad de la funcionalidad lógica
- Tipo, modos y densidad de la memoria
- Tipo y jerarquía de la red de interconexión
- Otras prestaciones

Siempre consideraremos el elemento más potente y rápido de cada familia. Las fuentes de todas las figuras son los *Datasheet* de los dispositivos correspondientes.

5.2.1. FLEX6000

FLEX son las siglas de (*Flexible Logic Element matriX*). El dispositivo está formado por un grupo bidimensional de bloques lógicos denominados LAB (*Logic Array Bloquek*) envueltos de recursos de interconexión. Cada LAB está formado por 10 LE (*Logic Element*). En total, la capacidad equivalente máxima es de unas 24000 puertas.

Un LAB está dimensionado para poder abarcar un bloque lógico de complejidad media (p.e. un multiplexor o un contador).

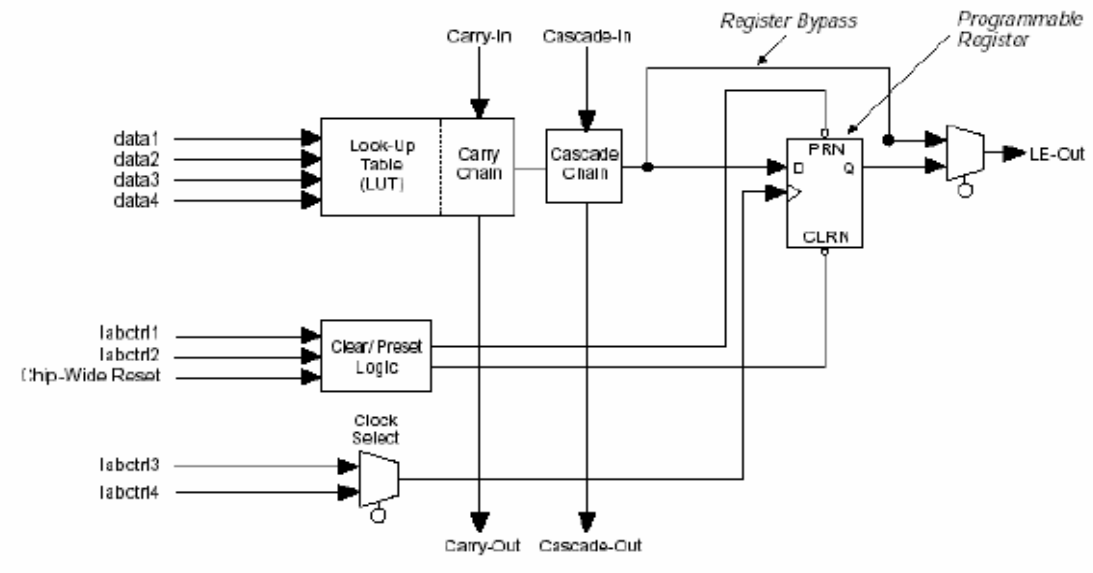


Figura 114. LE (FLEX6000)

El LE está basado en una *4-input LUT y* un biestable, tal y como se aprecia en la figura 112. La *4-input LUT* es una memoria que permite aplicar cualquier función lógica de cuatro entradas y una salida. El biestable permite implementar sistemas secuenciales. Los *carry-in y carry-out* son conexiones de bajo retraso entre LE que facilitan la implementación de bloques aritméticos basados en la propagación de *carry*. Los *cascade-in y cascade-out* son también conexiones de bajo retraso entre LE que facilitan la implementación de funciones lógicas de más de cuatro entradas. Los *labctrl1*, ..., *labctrl4* son comunes a todos los LE de un mismo LAB y controlan la funcionalidad del biestable de salida.
Un LEI se puede configurar de tres formas:

- El modo normal (ver figura 115) es útil para implementar funciones lógicas de propósito general (típicamente requerirá conexiones de diversas LE en cascada y, por tanto, los *cascade-in y cascade-out* tendrán mucha relevancia).

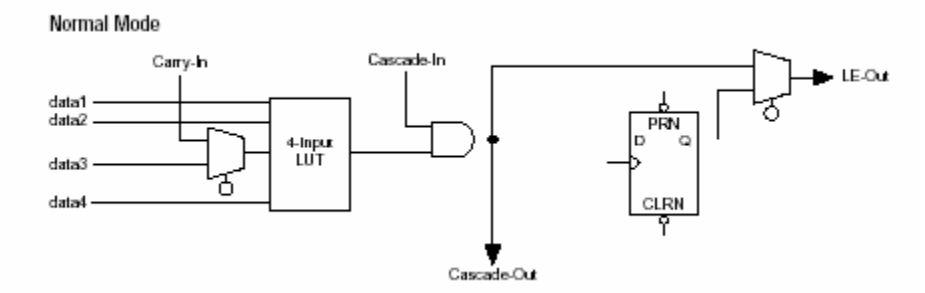


Figura 115. LE configurado en modo normal

- El modo aritmética (figura 116) facilita la implementación sumadores, restadores, comparadores, etc. La *4-input LUT* se divide en dos *3-input LUT.* En el caso de un sumador, por ejemplo, una de ellas se usará para calcular la suma de los dos bits asociados al LE y la otra para calcular el *carry* de la etapa. Este *carry* se propagará con un retraso mínimo hacia el LE del bit de siguiente peso.

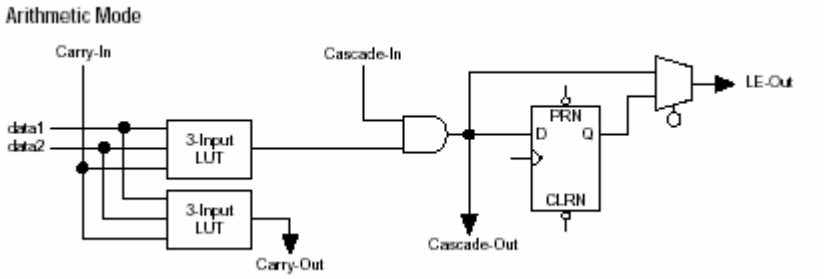


Figura 116. LE configurado en modo aritmético

- El modo contador (figura 117) se usa para implementar contadores con diferentes funcionalidades: *enable*, *up-down*, etc. Ahora también la *4-input LUT* se divide en dos *3-input LUT*; una de ellas calculará el *carry* cap en la siguiente etapa y la otra el bit actual según la configuración del contador determinada mediante *data1 y data2*.

El software de diseño configura cada LE en el modo apropiado.
En lo que respecta a la memoria, este dispositivo no dispone de elementos de almacenamiento específicos: sólo tiene memoria distribuida que provee de la posibilidad de configurar cada LE como una memoria de 16x1.

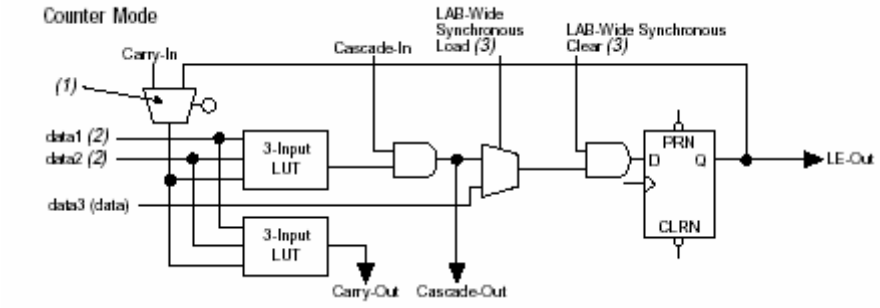


Figura 117. LE configurado en modo contador

La red de interconexión es jerárquica. Hay recursos de corto, medio y largo alcance (de más rápidos a más lentos respectivamente):

- Entre LEs de un mismo LAB, o pertenecientes a LABs de una misma media fila: *carry-chain* (útil para funciones aritméticas. Retraso de 0.1 ns.)

- Entre LEs de un mismo LAB, o pertenecientes a LABs de una misma media fila: *cascade-chain* (útil para funciones lógicas de muchas entradas. Retraso de 0.5 ns.)

- Entre LE de un mismo LAB o LAB adyacentes: *LAB local interconnect* (figura 118).

- Entre LE de diferentes LAB: red de interconexión global *FastTrack* (por filas o columnas) (figura 118)

  - Líneas que van de punta a punta de dispositivo
  - Líneas de longitud mitad

La red FastTrack no está basada en matrices de interconexión, sino que permite conexiones punto a punto a través de líneas dedicadas y multiplexoras. El retraso de la red es predecible ya que el número de multiplexores que se atravesarán para ir de un punto a otro está perfectamente definido. Este tipo de red, con ligueras variaciones, es común a todos los dispositivos FPGA de Altera.

Un *benchmark* destacado por este dispositivo: *un 8-bit 16-tap parallel FIR* ocupa 599 LE y se puede implementar a una frecuencia de reloj de 94 MSPS sobre el dispositivo más rápido de la familia.

5.2.2. FLEX10k

La principal novedad de esta familia respecto al anterior es que incluye bloques de memoria dedicada denominados EAB (*Embedded Array Bloquek*).
La capacidad lógica máxima (*logic array*) es de 1520 LAB. Cada LAB está formada por 8 LE (en número de LE por LAB cambia con esta familia). En total, capacidad equivalente de unas 250.000 puertas y hasta470 pins de I/O.
Cada LE está basado en una *4-input LUT y* un biestable. Se pueden configurar con Cuatro modos diferentes, aunque la funcionalidad es muy parecida a la de la familia FLEX6000: ahora existen dos modos de contadores, uno con el control *up/down y enable y* el otro *up/down y clear*.

Los EAB se pueden utilizar para implementar funciones lógicas complejas o diversos tipos de memorias:

- Función lógica de 8 entradas y 8 salidas (se graba en el momento de configuración y se utiliza para sólo lectura). Por ejemplo, un multiplicador de 4 x 4 bits.

- Memoria (256x8, 512x4, 1024x2, 2048x1)

  - Memoria RAM de dos puertos (dos lecturas o dos escrituras)
  - Memorias RAM de un puerto
  - Memorias FIFO

Se pueden combinar diversos EAB para formar memorias más grandes.

La conectividad entre elementos lógicos es idéntica a la Flex6000. Los retrasos de *carry y cascade* son un poco mayores (0.2 ns y 0.7 ns respectivamente). Ahora la posibilidad de conexión de estas pins de forma rápida están en toda a la fila (recordemos que en el caso de Flex6000 sólo a la mitad de la fila).

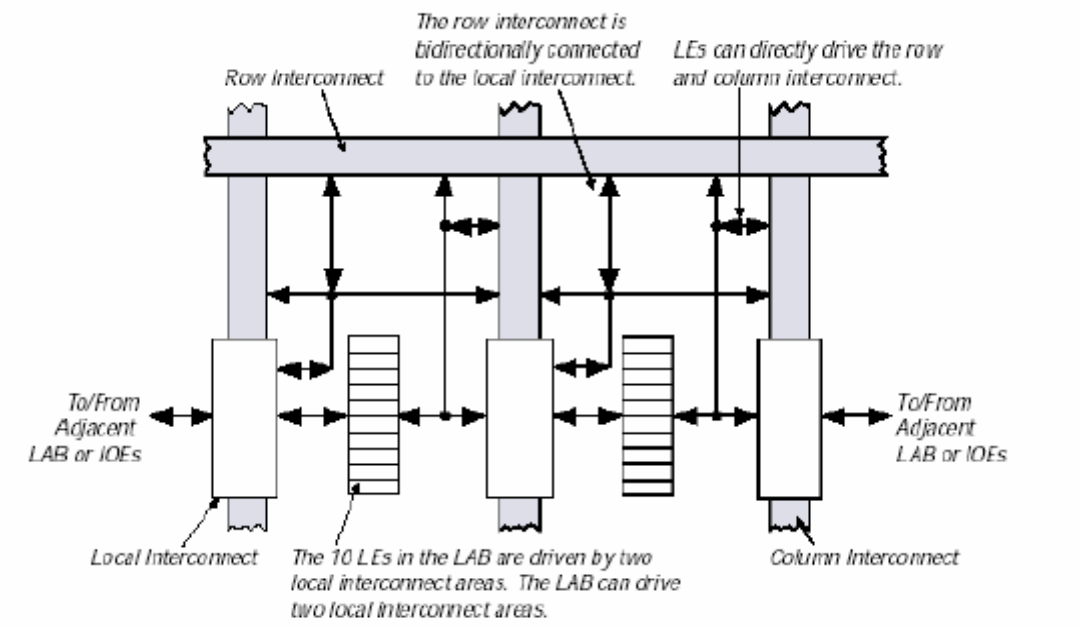


Figura 118. Red de interconexión (FLEX6000)

5.2.3. ACEX1k

La familia ACEX1k ofrece dispositivos de bajo coste y capacidad media. No introduce grandes novedades; sólo el incremento de la capacidad en algunos aspectos. La capacidad lógica máxima es de 624 LAB (4992 LE), equivalentes aproximadamente a 100.000 puertas.

Contiene además 12 EAB, cada EAB de 4096 bits. Por tanto se pueden configurar para implementar funciones lógicas de hasta 8 entradas y 16 salidas. La red de interconexión es muy parecida a la de la familia FLEX10k. Ahora los retrasos de *carrychain* y *cascade-chain* son de 0.2 ns y 0.6 ns respectivamente.

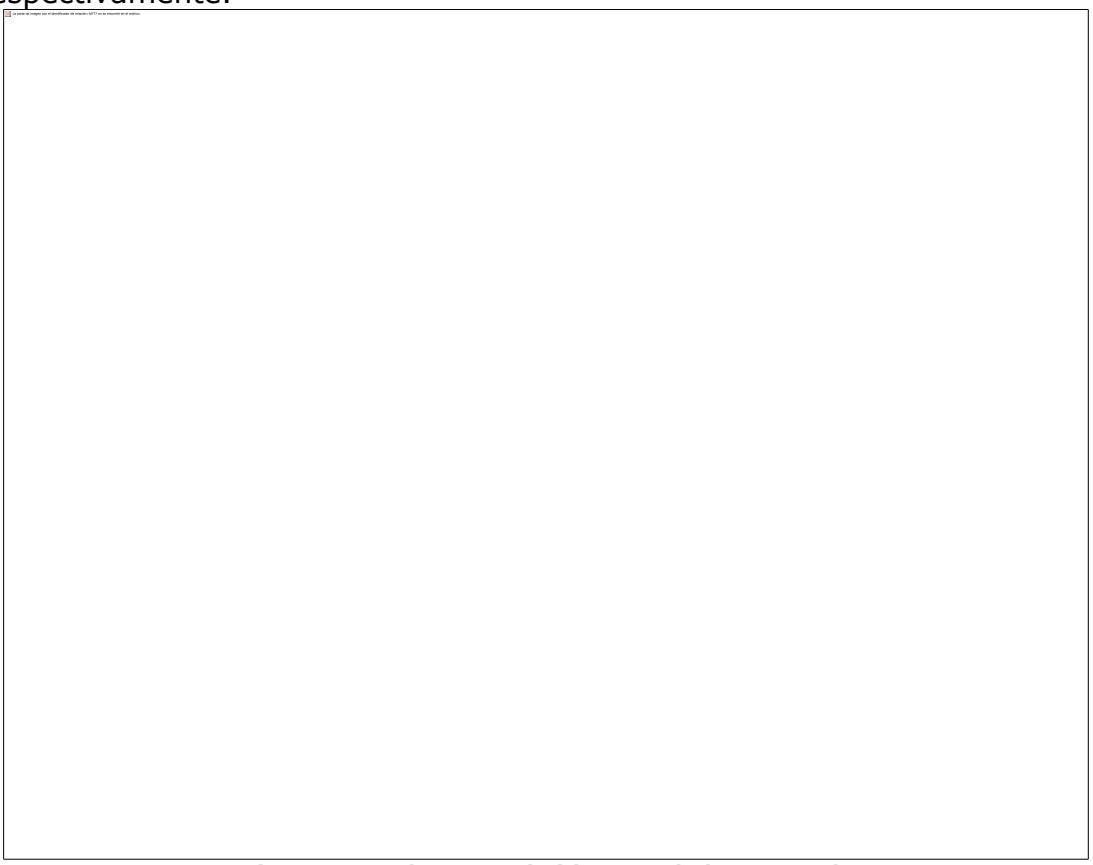


Figura 119. Diagrama de bloques de la FLEX10k

5.2.4. Mercury

Capacidad lógica máxima de 1440 LAB (14400 LE), equivalentes por lo aproximadamente a 350.000 puertas lógicas. Un LAB, por tanto, vuelve a estar formado para 10 LE.

El LE continúa basado en una *4-input LUT y* un biestable. Ahora, no obstante, entre otras mejoras incorpora soporte para multiplicadores y más señales de control comunes a todos los LE de un LAB.

Además, podemos utilizar la *4-input LUT y* el biestable de un LE de forma independiente (hay una salida de la LUT accesible y otra del registro). Esta posibilidad permite aprovechar mejor los recursos del dispositivo.
Cada LE tiene tres modos de funcionamiento:

- Normal. Funciones lógicas y combinacionales.

- Aritmética. Sumas y comparaciones. La *4-input LUT* se configura como 4 *2-input LUT* (dos para calcular una suma de 1 bit con los dos posibles valores de *carry*, y dos más para calcular los *carrys*)

- Multiplicador. Se utiliza para implementar multiplicadores rápidos de hasta16x16 bits.

Los bloques de memoria cambian de nombre: ara ESB (*embedded system bloqueks*). Permiten implementar diferentes tipos de memorias:

- *Quad port* (dos lecturas y dos escrituras)

- *True dual port* (dos lecturas, dos escrituras o una lectura y una escritura)

- *Dual port* (una lectura y una escritura)

- Un port (RAM, ROM, FIFO y CAM)

Cada ESBse puede configurar como 256x 16, 512x8, 1024x4, 2048x2 o 4096x1. Los bloques, además, son divisibles: se puede configurar para implementar, por ejemplo, 2 bloques de 128x16.

Figura 120. LE (Mercury)

La red de interconexión se denomina MultiLevel FastTrack Interconnect structure.

Como siempre, Altera la diseña para introducir retrasos predecibles.
En un primer nivel, hay dos tipos de líneas: regulares y priorizadas. También se diferencian los recursos de conectividad de fila y columna. Veámoslo.
Recursos de fila. Interconectan LAB, EAB o IOB de una fila. Tipo:

- Conexiones que atraviesan todo el dispositivo, de izquierda a derecha.

- Conexiones prioritarias que atraviesan todo el dispositivo, de izquierda a derecha

- RapidLAB, que comprenden 10 LAB desde una LAB central (la estructura de la figura se repite por cada LAB)

Figura 121. Red de interconexión (Mercury)

Recursos de columna. Ídem. Tipo:

- Conexiones que atraviesan todo el dispositivo, de izquierda a derecha

- Conexiones prioritarias que atraviesan todo el dispositivo, de izquierda a derecha

- *Leap lines* que rutean filas adyacentes de EAB y LAB.

Otros recursos de conectividad:

- FastLUT: señales entre un LE y el adyacente inferior del mismo LAB (a la salida de la LUT, antes de los flip-flops)

- Conectividad local: señales entre LE de un mismo LAB o adyacentes. Los LE parejos se pueden conectar con el LAB situado a la izquierda y los impares del LAB situado a la derecha

Otra novedad importante consiste en que la FPGA permite reconfigurabilidad dinámica: tiene capacidad para cambiar su comportamiento en tiempo de ejecución cargando una nueva configuración ya sea a través de una memoria externa o gobernado por un procesador.

El dispositivo también incorpora hardware específico para implementar puertos de comunicación serie de hasta1.25 Gbps.

5.2.5. APEX20K

Este dispositivo introduce una nueva organización jerárquica en el *array* de elementos lógicos: el MegaLAB. Un MegaLAB es una agrupación de 16 LAB, 1 ESB y una *MegaLAB interconnect*. En el dispositivo mayor de la familia (EP20K1000C) cada MegaLAB contiene 24 LAB. La capacidad lógica máxima es de 38400 LE, equivalente a 1.000.000 de portes. Los LE son muy parecidos a los miembros anteriores de la familia.

Cada ESB (*Embedded System Bloquek*) contiene 2048 bits de memoria. La capacidad máxima de memoria es de 327.680 bits. Cada ESB se puede configurar como RAM, ROM, FIFO o CAM.

Los ESB también se pueden configurar en el *product-term mode*. Entonces, cada ESB implementa *16 macrocell*. En la figura 120 podemos ver el diagrama de bloques de una *macrocell* (contiene dos *product-term* que se pueden combinar con *or* o *xor*, seguidos de un registre). Este tipo de estructuras son óptimas para implementar descodificación de direcciones,

diagramas de estados complejos o funciones lógicas de muchas entradas. De hecho, son estructuras parecidas a las PLD.

La combinación de arquitecturas basadas en LUT y las basadas en *product-terms* en un mismo dispositivo permite optimizar muchos diseños que requieren ambos perfiles de *hardware*.

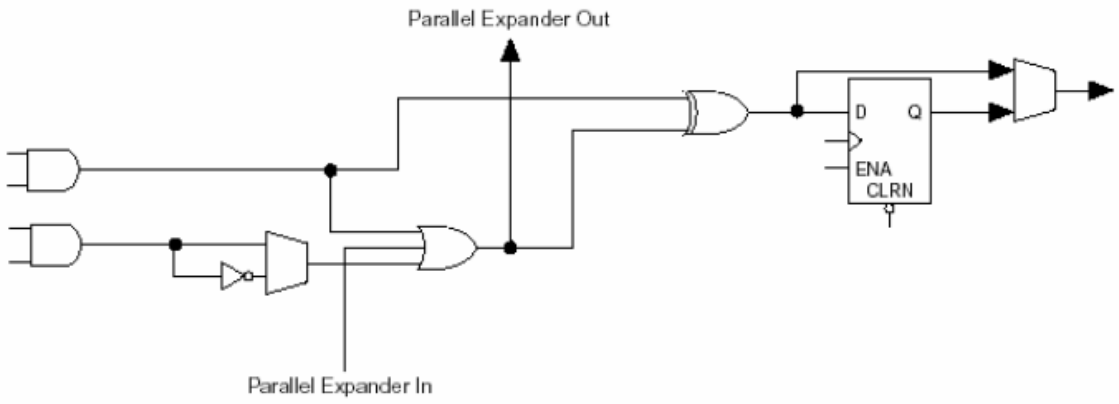


Figura 122. *Product-term macrocell (APEX20k)*

En lo que se refiere a la red de interconexión, distinguimos cuatro jerarquías:

- *Carry y cascade*, entre LE de un LAB y todos los LAB de un MegaLAB
- *Local interconnect*, entre LAB adyacentes de un MegaLAB, y hacia IOB
- *MegaLAB interconnect*, entre LAB de un MegaLAB
- *FastTrack interconnect*. Filas y columnas que abrazan todo el dispositivo

5.2.6. APEX-II

Es un incremento cuantitativo del dispositivo anterior. La FPGA se organiza en MegaLABs. Cada MegaLAB contiene 16 o 24 LAB, 1 ESB y una *MegaLAB interconnect*. Cada LAB tiene 10 LE. La capacidad máxima es de 67200 LE y 280 ESB correspondientes a 1146880 bits. Los LE tienen una estructura similar a la de otras familias. La red de interconexión (*MegaLAB interconnect y FastTrack interconnect*) es también similar a otras familias.

El dispositivo también permite implementaciones basadas en arquitecturas *product-term* que se implementan sobre bloques ESB.

5.2.7. Stratix y Stratix GX

La familia de Stratix es un salto tanto cuantitativo como cualitativo. Se incrementa tanto la capacidad lógica como la de memoria. Además, incluye bloques multiplicadores y sumadores (DSP), memoria con nuevos modos de configurabilidad, canales de comunicación serie de alta velocidad, etc. (ver figura 123). La capacidad lógica máxima (Stratix GX) es de 41250 LE. Cada LAB contiene 10 LE (ver figura 124) con una estructura muy similar a la de las familias anteriores a Apex. Como novedad destacamos dos nuevas posibilidades de encadenamiento de señales (además del *carry-chain*):

- El *LUT chain* permite encadenar LUT de LE del mismo LAB sin pasar por el registro de salida. Facilita la implementación de funciones lógicas de muchas entradas.

- *Register chain* que permite encadenar registros de LE del mismo LAB sin tener que por el LUT. Facilita la implementación de *shift-registers*.

Los dos mecanismos permiten usar el LUT y el flip-flop de un mismo LE para tareas que no tengan nada que ver. Por lo tanto, la eficiencia de la arquitectura en lo que se refiere a las posibilidades de utilización de recursos mejora.

Cada LE tiene dos modos de funcionamiento:

- Normal, por implementar funciones lógicas y combinacionales en general

- *Dynamic arithmetic* para implementar sumadores, contadores, comparadores, generadores de paridad

Por lo que respecta a la memoria, también hay novedades. La nueva arquitectura, denominada *TriMatrix Memory,* contiene tres tipos de bloques de RAM: M512, M4k, M-RAM, que tienen las posibilidades siguientes:

- True dual-port (M4, M-RAM), dual-port y single-port

- FIFO, ROM

- Subdivisibles en dos memorias

- Soporten modo de paridad a nivel de byte

- Modo de shift-register (útil para filtros, correlaciones, generador de secuencias pseudo-aleatorias, etc.)

De bloques M512 (512 bits) hay muchos distribuidos por todo el dispositivo. De M4k (4096 bits) hay menos. M-RAM (512 Kbit) hay pocos. Los M512 son los más rápidos y los M-RAM los más lentos (ver figura 123).

Otra novedad son los bloques DSP. Bloques DSP. Un *DSP bloquek* permite implementar:

- 8 multiplicadores 9x9 bits
- 4 multiplicadores 18x18 bits
- 1 multiplicador 36x36 bits

Cada *DSP bloque k* contiene también sumadores / acumuladores:

- 4 sumadores 9+9 bits
- 2 sumadores 18+18 bits

Cada *DSP bloque k* puede funcionar en los modos siguientes:

- Multiplicador
- Multiplicador y acumulador (2 x (1 multiplicadores + 1 acumulador) de 18 bits)
- Multiplicador y sumador (2 x (2 multiplicadores + 1 sumador) de 18 bits). Muy útil para resolver con un *DSP bloquek* una multiplicación compleja.
- 4 multiplicadores y un sumador (4 x multiplicador de 18x18 y suma de los resultados). Implementa un filtro FIR de 4 taps en un *DSP-bloquek*.

La red de interconexión recibe el nombre de *MultiTrack interconnect structure with DirectDrive technology*. Está compuesta por filas y columnas de líneas de interconexión que tienen una longitud determinada. El retraso, como siempre, es predecible: no se basa en matrices de conexiones sino que son conexiones punto a punto que se pueden activar o desactivar mediante multiplexores.

Recursos de fila:

- Interconexiones de enlace directo entre LAB y otros bloques adyacentes
- Interconexiones R4 que incluyen:
  - Cuatro LAB adyacentes o
  - Tres LAB adyacentes y un bloque M512 adyacente

- Dos LAB adyacentes y un bloque M4k adyacente
- Dos LAB adyacentes y un bloque DSP

Cada LAB tiene dos interconexiones R4 disponibles: una hacia la derecha y otra hacia la izquierda.

- Interconexiones R8 que abrazan ocho bloques adyacentes
- Interconexiones R24 que abrazan el ancho del dispositivo

Recursos de columna:

- *LUT chain* entre LE de un LAB
- *Register chain* entre LE de un LAB
- Interconexiones *C4* (abrazan cuatro bloques arriba o abajo)
- Interconexiones *C8* (abrazan ocho bloques arriba o abajo)
- Interconexiones *C16* (abrazan dieciséis bloques arriba o abajo)

A continuación podemos observar el diagrama de bloques del Statrix:

Figura 123. Diagrama de bloques (Stratix)

5.2.8. Cyclone

Es una simplificación (bajo coste) del Stratix (sólo tiene bloques de memoria RAM4k, no tiene DSP, etc.)

5.2.9. Stratix II

Los Stratix II son las FPGA de última generación de Altera. Están basadas en una nueva arquitectura de elemento lógico básico: el ALM (*Adaptive logic module*). Un ALM permite 8 entradas (sólo hasta 6 diferentes) y 4 salidas (dos síncronas y dos asíncronas). Contiene dos LUT de medida (entre 3 y 6 entradas cada una) y modo de funcionamiento configurable. La flexibilidad del ALM es el principal punto fuerte de la propuesta.

La capacidad máxima es equivalente a 180.000 LE de los de antes, contiene hasta9 Mbits de memoria jerarquizada igual que las anteriores Stratix (*TriMatrix*), 96 DPS bloques y 1173 pins de entrada salida.
Según el fabricante, el Stratix II puede implementar FFT-1K a frecuencias de reloj de 274 Mhz.

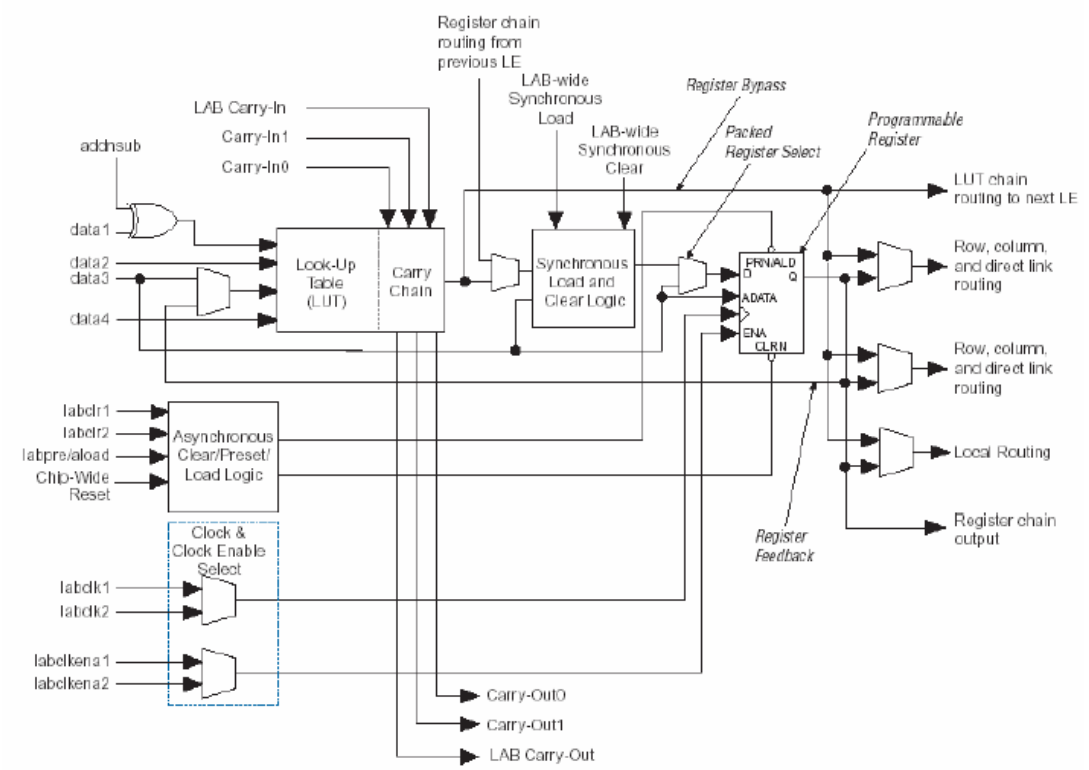


Figura 124. LE (Stratix)

5.3. Estudio de mercado. FPGA de Xilinx

A continuación describiremos las familias de FPGA de Xilinx. La estrategia de este fabricante consiste en utilizar una red de interconexión basada en matrices de manera que el ruteado de pistas condiciona el retraso de propagación de las señales. En contrapartida, la simplicidad de esta aproximación en contraste con la de Altera favorece una mayor densidad de integración de portes.

Analizaremos cada familia desde diferentes puntos de vista:

- Jerarquía, estructura y densidad de la funcionalidad lógica
- Tipo, modos y densidad de la memoria
- Tipo y jerarquía de la red de interconexión
- Otras prestaciones

Siempre consideraremos el elemento más potente y rápido de cada familia. Las fuentes de todas las figuras son los *Datasheet* de los dispositivos correspondientes.

5.3.1. Spartan, /XL

La arquitectura interna de esta familia está formada por un *array* de hasta28x28 (784) CLB (*Configurable Logic Bloquek*), equivalentes a unas 40.000 puertas lógicas. Un CLB está formado por dos *4-input LUT*, una *3-input LUT*, multiplexores y dos biestables.

Un CLB permite implementar:

- 2 x Función lógica de 4 variables, 1 x Función lógica de 3 variables
- 1 x Función lógica de 5 variables
- 1 x Función lógica de 4 variables, 1 x Alguna función lógica de 6 variables
- 1 x Alguna función lógica de 9 variables

La red de interconexión está basada en líneas de diferentes longitudes que se cruzan en nudos denominados PSM (*Programmable Switch Matrix*). Los PSM permiten unir cualquier línea horizontal con cualquier vertical mediante transistores. Con tal de unir dos CLBs distantes el número de PSM que se

atravesarán dependerá de cada ruteado. El retraso extremo a extremo será impredecible a priori ya que depende en gran medida del número de PSM atravesado.

Hay líneas de interconexión de tres longitudes diferentes:

- Simple: conectan un CLB con una PSM. Sirven por conectividad de corto alcance, ya que si se utilizan para largo alcance han de pasar muchas PSM y cada vez que se pasa por una PSM hay un retraso adicional
- Doble: antes de entrar a una PSM pasan de largo dos CLB
- Larga: van de punta a punta del dispositivo sin pasar por PSM

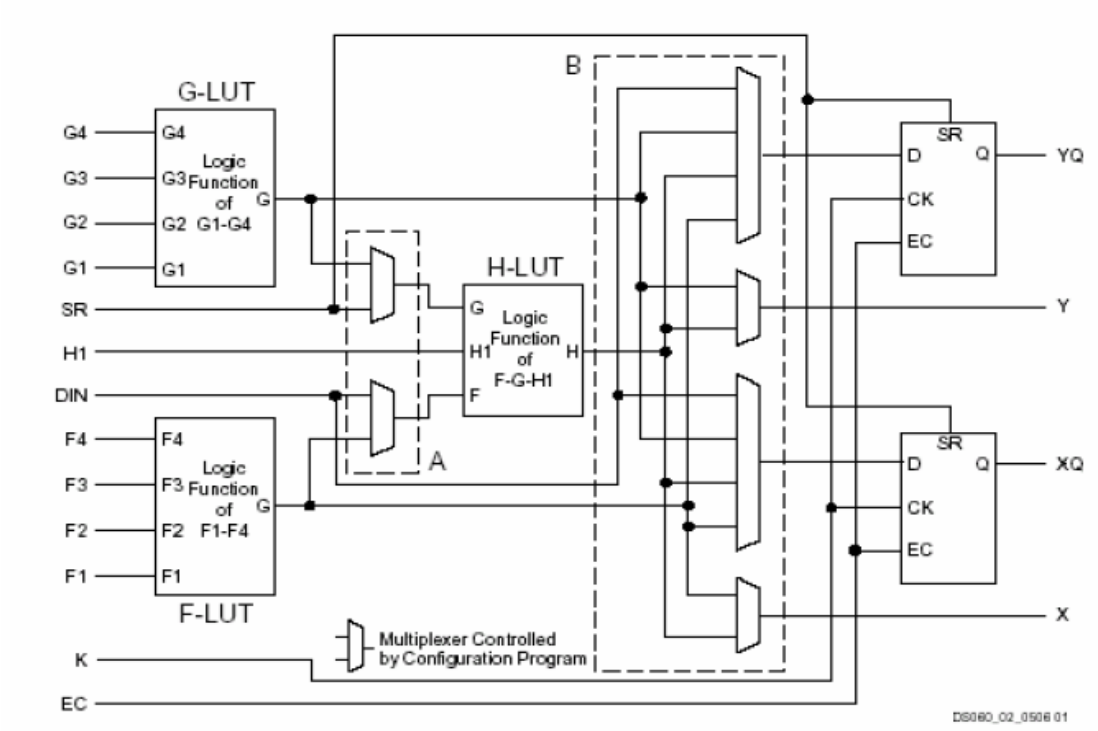


Figura 125. CLB (Spartan)

La FPGA no tiene memoria concentrada. La capacidad máxima de la memoria distribuida es de 25088 bits (32 bits para CLB, ya que contiene dos LUT de cuatro entradas).

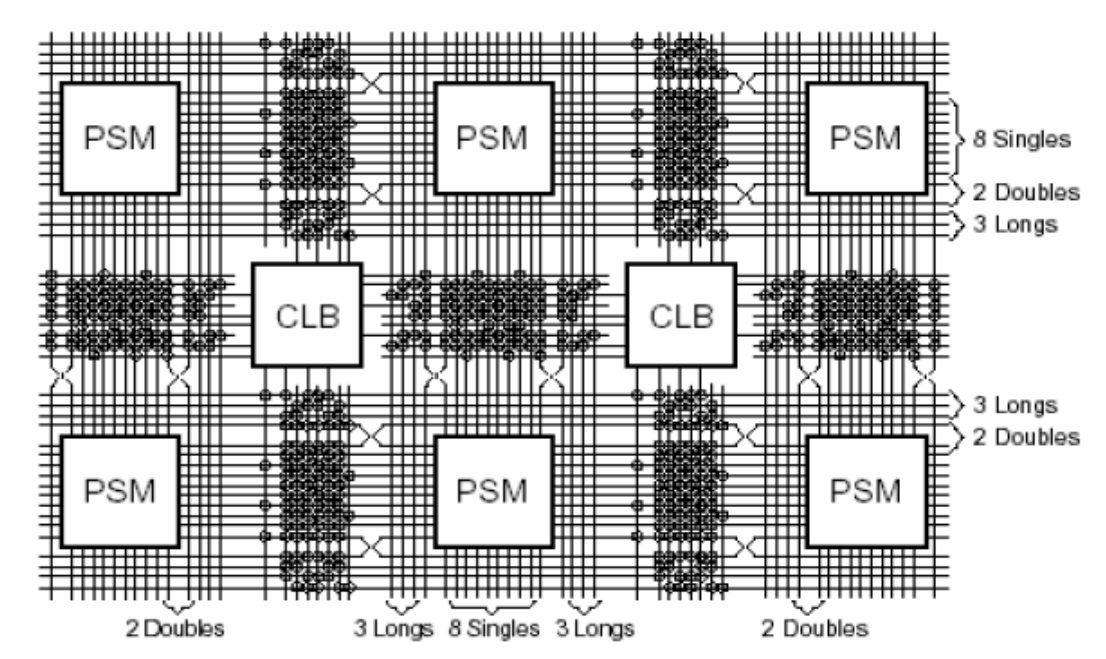


Figura 126. Matrices de conexión (Spartan)

5.3.2. Spartan II

Esta familia introduce como una de las principales novedades los bloques de memoria dedicada (hasta 14 bancos de 4096 bits), gestión de reloj mediante DLL (*Delay Locked Loop*) y soporte para una gran variedad de estándares de I/O.

La capacidad lógica máxima es de 1323 CLB, correspondientes a 5292 LC (*Logic Cell*). Cada LC contiene una *4-input LUT*, lógica de *carry y* un biestable (ver figura 127). Dos LC forman un *slice*.

Una LUT se puede configurar para generar una función lógica cualquiera de cuatro entradas, una memoria 16x1 bit, un *shift register* de 1 bit y 16 desplazamientos o bien para ejecutar funciones aritméticas (dispone de lógica para implementación rápida de *carry*).

Además de la memoria distribuida de les LUT, la familia introduce una máxima de 14 *bloquek RAM* que permiten hasta56 kbit de capacidad.

Los nudos centrales de la red de interconexión entre líneas horizontales y verticales se denominan GRM (*General Routing Matrix*). Hay dos jerarquías de conectividad:

- Local, subdividida en:
  - Conexiones entre los componentes de cada CLB y la GRM
  - Realimentaciones de otra velocidad entre componentes de un CLB
  - Conexiones directas entre componentes de CLB adyacentes sin pasar por ningún GRM

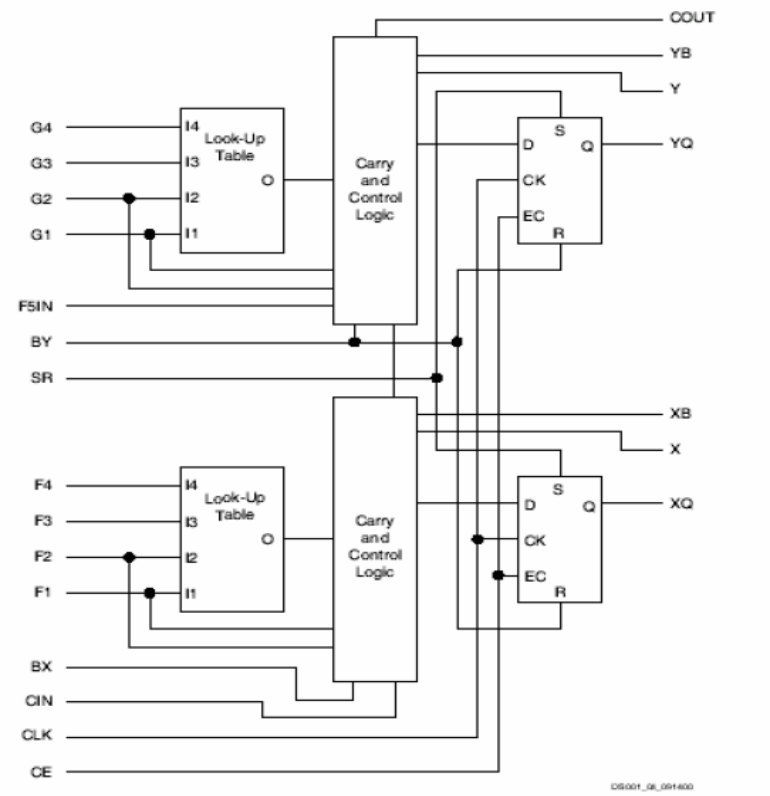


Figura 127. Un *slice* (Dos LC) (Spartan II)

- De propósito general, subdividida en:
  - Conexiones entre líneas horizontales y verticales dentro de cada GRM
  - Simples, que conectan GRMs adyacentes
  - Hex, que conectan GRM separados 6 bloques, accesibles por lectura e el punto medio.
  - Largas, de punta a punta, tanto a lo ancho como a lo alto.

5.3.3. Spartan IIE

Muy parecida a Spartan II, con más capacidad de memoria (hasta a 288Kbit).

5.3.4. Virtex

La capacidad lógica máxima es de 6144 CLBs distribuido en un array de 64x96. El número de LC equivalentes es de 27648.

Por lo que se refiere a la arquitectura lógica no hay demasiados cambios. Cada CLB contiene 4 LC organizados en dos *slices*. Cada LC incorpora una *4-input LUT*, generador de *carry y* un biestable. Además cada CLB contiene lógica para combinar las salidas de cada LUT.

Por esto a la hora de contabilizar LC cada CLB equivale a 4.5 LC.
Cada *slice* puede operar como a generador de funciones lógicas, memoria RAM (32x1 *single-port* o 2x16x1 *dual-port*) o 16-bit *shift register*.

El dispositivo incorpora además hasta32 bloques de memoria *SelectRAM* hasta una capacidad total de 128kbit.

La red de interconexión y de distribución de reloj (DLL) es igual a la del Spartan II.

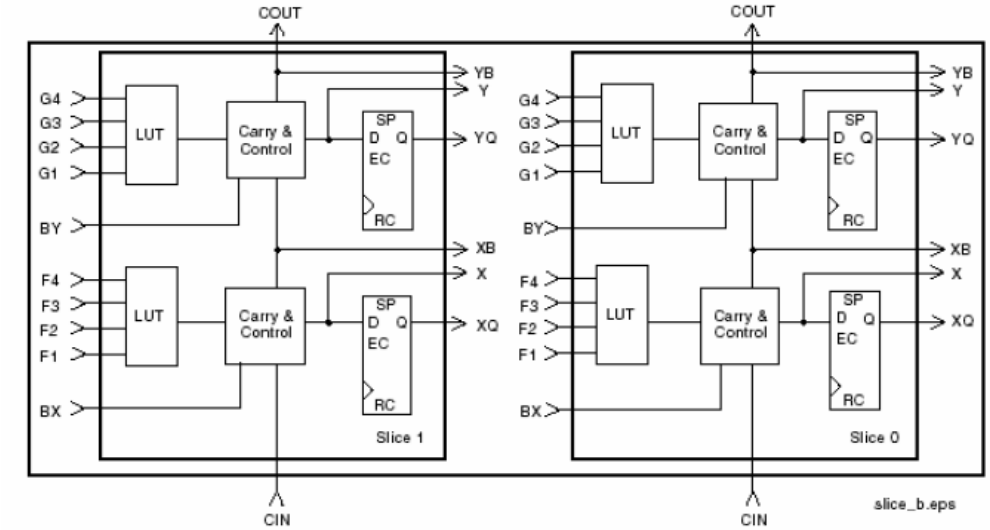


Figura 128. CLB (Virtex)

5.3.5. Virtex E

Muy parecida a Virtex con mayor capacidad. Por ejemplo, hasta16224 CLB y 832 Kbits de memoria concentrada

5.3.6. Spartan 3

Dispositivos que funcionen con una tecnología de 1.2v, con IOB que soporten gran variedad de estándares de alta velocidad.

La capacidad lógica máxima es de 8320 CLBs. Cada CLB está compuesta por Cuatro *slices*, y cada *slice* por dos *4-input* LUT. En total, pues, 8 *4-input LUT* por CLB. Los *slices* están agrupados en parejas: *left-hand SLICEM y righ-hand SLICEL*. La primera pareja puede implementar funciones lógicas, memoria distribuida o *shift-registers*. La segunda pareja sólo puede implementar funciones lógicas.

Cada CLB equivale a 9 LC. Por tanto, la capacidad lógica máxima es de 74880 LC. La capacidad de memoria distribuida es de 64 bits por CLB (4x16x1). En total, por lo tanto, 520 kbit. Además hay un máximo de 104 bloques de memoria dedicados de 18 kbit cada uno (en total 1.872 kbit), configurables como ROM, RAM o *dual-port* RAM.

El dispositivo incorpora hasta104 multiplicadores de 18x18 bits (CA2 con salida de 35 bits). Cada multiplicador está fuertemente acoplado a un bloque de RAM de manera que la transferencia de datos entre los dos dispositivos es óptima.

Otro aspecto a destacar son los DCM (*Digital Clock Manager*). Tienen las funciones siguientes:

- Modo DLL (*Delay-Locked Loop*), elimina el *clock skew* entre dos relojes (un de referencia y otrol).

- Modo DFS (*Digital Frequency Synthesizer*), multiplicador y divisor de frecuencias (generador de reloj).

- Modo PS (*phase shifter*), shiftador de fase.

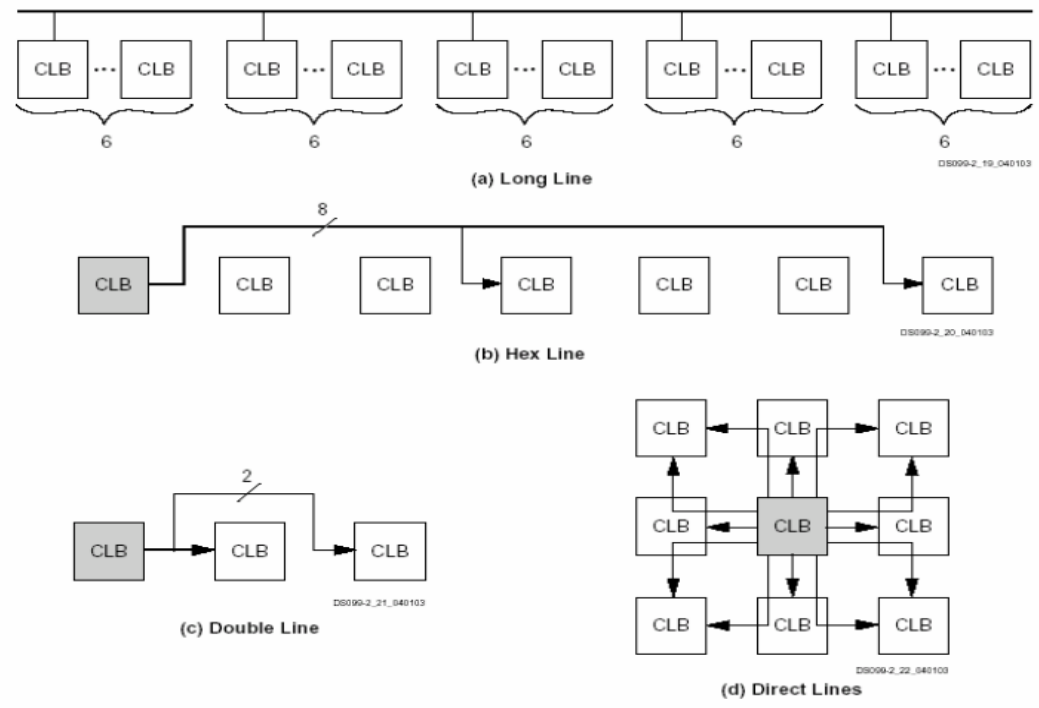


Figura 129. Tipo de líneas (Spartan 3)

En lo que se refiere a la red de interconexión, distinguimos:

- Red específica para rutear reloj: (*Global Clock Network*)
- Para otras señales, cuatro tipos de líneas
  - Directas: conectan CLB adyacentes
  - Dobles: conectan CLB separados 2 bloques
  - Hex: conectan CLB separados 6 bloques, accesibles en el punto medio
  - Largas: conectan CLB separados por 6 bloques (parar señales de otra frecuencia)

5.3.7. Virtex II

Arquitectura parecida a la de los Spartan 3. Por un lado la capacidad lógica, como siempre distribuida en un array del CLB (1 CLB equivale a 4 *slices; hasta* 11648 CLB en un array de 112 x 104), por otro, la memoria concentrada en bloques SelectRAM (hasta 168 bloques de 18kbit, dando un total de 3024 kbit configurable con *singleport*, *dual-port y* diversas anchuras de bus), multiplicadores de 18 x 18 bits asociados a cada bloque SelectRAM y hasta12 gestores de reloj DCM. El dispositivo mayor tiene hasta1108 pins d'I/O.

5.3.8. Virtex II Pro

Esta familia amplia las capacidades cuantitativas de la Virtex II:

- Capacidad lógica de hasta13904 CLBs
- Hasta 556 bloques de 18 kbit de SelectRAM (10.008 kbit)
- Hasta 556 multiplicadores de 18x18 bits

Además, cada dispositivo lleva hasta4 *IBM PowerPC RISC procesors*:

- 300 Mhz con cinco niveles de segmentación
- 32 registros de propósito general de 32-bits
- Unidad multiplicadora / divisora
- *Cache* de datos y de código

5.4. HDL

5.4.1. Introducción

Los dispositivos de lógica programable de poca capacidad se solían configurar a través de lenguajes sencillos que permitan introducir ecuaciones combinacionales o secuenciales.

Estas ecuaciones se sintetizaban por software para crear un archivo que programaba el PLD.

Para dispositivos con una capacidad mucho mayor (tipo CPLD o FPGA) esta metodología es inadecuada ya que la generación de ecuaciones en diseños complejos es complicada y puede llevar a cometer errores fácilmente. En estos casos una primera alternativa es el uso de esquemáticos que permiten, entre otras ventajas, la visualización gráfica del diseño y la aplicación de una metodología modular y jerárquica. El uso de esquemáticos, no obstante, tiene también importantes inconvenientes:

- Reflejan más la estructura de la implementación que el funcionamiento general. Para este motivo podemos afirmar que:
  - Diseños muy complejos se convierten en poco comprensibles en lo que se refiere al funcionamiento general. Se considera que los esquemáticos de un diseño con una complejidad superior a unas 600 puertas son difíciles de entender.
  - El mantenimiento, cambios y ampliación del diseño pueden ser complicados para aplicaciones muy complejas.
  - Los esquemáticos no son autoexplicativos; a menudo requieren comentarios sobre los circuitos, comentarios que pueden convertirse en un punto débil para la falta o variedad de interpretaciones a que pueden dar lugar.

- Los paquetes software que soportan la entrada de esquemáticos suelen ser propietarios y vinculados a fabricantes, de forma que la exportación de los diseños es complicada.

Una segunda alternativa es la utilización de un lenguaje estándar de descripción de hardware (HDL: *Hardware Description Language*). Los lenguajes HDL permiten especificar hardware directamente a partir de los requerimientos con un esfuerzo de traducción mínimo.

Un lenguaje HDL conviene que soporte:

- Diseño jerárquico y modular.

- Potencia, con construcciones de alto nivel que implementen complicadas funciones lógicas de una manera sencilla para el programador.

- Flexibilidad. Ha de permitir el uso de librerías y la reutilización de componentes.

- Sea portátil y, por tanto, no esté ligado a ninguna arquitectura concreta ni fabricante. Es necesario que permita la posibilidad de iniciar un diseño sin haber elegido el dispositivo que lo ejecutará y ofrezca intrínsecamente la posibilidad de mapear sobre diferentes arquitecturas para optimizar la implementación.

- Esté bien definido y sea autoexplicativo (no requiera de un exceso de aclaraciones)

- Facilite el mantenimiento y ampliación del diseño.

- Minimize el *time-to-market*. Como todos los lenguajes de alto nivel, al no requerir del conocimiento específico de cada arquitectura permitan utilizar nuevos dispositivos con un coste temporal mínimo.

Existen diversos lenguajes con estas características. Dos de los más importantes son VHDL y Verilog.

VHDL (*Very High speed integrated circuit Hardware Description Language*) es un producto creado para el departamento de defensa de los Estados Unidos durante las décadas de los setenta y ochenta. El año 1987 fue adoptado como estándar (IEEE 1076-87) y posteriormente revisado y ampliado (IEEE 1076-93). Inicialmente VHDL estaba pensado para ser un lenguaje estructurado que permitiese documentar y simular hardware digital.

Posteriores esfuerzos en el ámbito de la investigación permitieron utilizar el lenguaje no sólo para las tareas antes nombradas, sino además para sintetizar el diseño sobre dispositivos de mercado.

Por otra parte, Verilog fue creado por Gateway1 en el año 1983. Originalmente fue, pues, un lenguaje propietario pensado para optimizar el *time-to-market* de los diseños basados en lógica programable. Posteriormente, el 1990 se convirtió en un estándar abierto y el 1995 el estándar IEEE 1364.

VHDL es un lenguaje más completo y que permite un mayor número de construcciones de alto nivel que Verilog. La simulación de sistemas complejos con VHDL suele ser lenta y requiere ordenadores con muchos recursos de procesador y memoria. Muchos sectores del mercado piensan

1 El 1989 Cadence compró los derechos de Verilog a Gateway

que es un lenguaje creado en los despachos, que no ofrece las prestaciones que los diseñadores de sistemas requieren. Por otro lado, Verilog es un lenguaje pensado y utilizado desde hace mucho por diseñadores, más sencillo y fácil de aprender y más rápido a la hora de simular.

Los lenguajes de descripción de hardware también tienen algunos inconvenientes:

- Se pierde información del diseño a nivel de puerta. De hecho este inconveniente es una consecuencia directa de las características exigidas a los lenguajes HDL: trabaja en un nivel de abstracción superior repercute en una pérdida de control sobre la implementación. Aún así, la mayoría de las herramientas de síntesis ofrecen cierto control de este aspecto permitiendo, por ejemplo, optimizar el diseño según un criterio de velocidad o espacio.

- La implementación lógica creada por las herramientas de síntesis suele ser poco eficiente, ya que las técnicas que apliquen son generales, nunca específicas de cada diseño.

- La optimización del diseño depende en gran medida de las herramientas de síntesis que se utilicen.

5.4.2. Proceso de diseño

Dividiremos el proceso de diseño en un lenguaje HDL en seis pasos que a continuación explicaremos.

❖ Definición de requerimientos

Es un paso común en cualquier proceso de diseño en el que no hay que hacer mucho énfasis.

❖ Descripción del diseño en HDL

En un primer momento tendremos que escoger la metodología de diseño que se utilizará. Para aplicaciones complejas las aproximaciones *top-down* o *bottom-up* suelen ser las más adecuados, mientras que diseños planos suelen ser óptimos en casos sencillos.

La metodología *top-down* afronta la solución al problema definiendo en primer lugar la estructura del nivel superior, es decir el bloque funcional principal y su entrada / salida. A partir de aquí se subdivide esta funcionalidad en módulos más pequeños sucesivamente hasta llegar a bloques básicos que habrá que definir e implementar. Por lo tanto, hay que

definir primero funcionalidades e interficies y finalmente implementar código.
La metodología *bottom-up* comienza implementando los módulos más pequeños, que se unirán entre sí para formar niveles jerárquicos superiores hasta llegar al módulo principal que cumplirá las especificaciones. En este caso, pues, es la implementación la que lleva a la definición de interficies (al revés que *top-down*); se afronta el problema generando código.

La modularidad inherente a las dos metodologías permite distribuir la tarea entre diferentes ingenieros y acelerar así el tiempo de desarrollo de la aplicación. Abusar de niveles jerárquicos no es recomendable ya que va en detrimento de la comprensión del diseño.

La codificación de cada módulo se llevará a cabo típicamente en un lenguaje HDL. Para generar un código eficiente hay que conocer cómo funcionan las herramientas de síntesis y ser consciente de que el código una vez sintetizado da lugar a hardware a medida. Por lo tanto, algunas prácticas habituales en la programación son muy poco eficientes en HDL.

❖ Simulación

La simulación a nivel funcional del código VHDL puede ahorrar mucho tiempo a la hora de completar diseños complexos. Además, si se utiliza una aproximación modular ese podrán simular bloques por separado y garantizar la funcionalidad de cada módulo de forma aislada.

Por diseños sencillos la simulación funcional es un paso que se suele obviar ya que el proceso de simulación del *layout* (posterior a la síntesis) da información no sólo funcional sino también temporal y no será excesivamente costosa en tiempo. Ahora bien, para diseños complejos la simulación es un proceso que permite detectar y corregir errores en una fase muy inicial del proyecto, sin peder tiempo en costosos procesos de síntesis.

❖ Síntesis, optimización y mapeo sobre la arquitectura

La síntesis consiste en la transformación del diseño codificado en HDL en una especificación de más bajo nivel (a menudo *netlist* para FPGA y ecuaciones lógicas para CPLD). El proceso de síntesis es específico de cada dispositivo: si se genera un *netlist*, el fichero de interconexiones es fuertemente dependiente de los recursos de cada arquitectura, mientras que si se genera un fichero con ecuaciones booleanas, el tipo de puertas lógicas utilizadas para especificarlas dependerá de las que contenga el dispositivo en cuestión.

El paso siguiente consistirá en optimizar la especificación en ecuaciones booleanas para adaptarla de la mejor forma posible a los recursos disponibles. Por ejemplo, la optimización por CPLD suele consistir en reducir las ecuaciones lógicas a un número mínimo de sumas de productos. En el caso de FPGA, conviene factorizar las ecuaciones en términos que se adapten a la estructura de una celda lógica.

Finalmente, el mapeo toma las ecuaciones sintetizadas y optimizadas y las ajusta a los recursos de una arquitectura en concreto. Diferenciaremos dos casos según si se trata de CPLD o FPGA:

- En el caso de CPLD recibe el nombre de *fitting*. Se evalúan nuevamente diferentes expresiones de las ecuaciones lógicas para señales activas de '0' o por '1', sistemas síncronos basados en *reset* o *preset*, etc. para adaptarlas de forma óptima a los recursos disponibles. Se escogen los recursos lógicos en función de estos resultados y de las directivas especificadas por el diseñador (p.e. asignación de pins) y se intenta el ruteado entre los recursos hasta que se consiga el objetivo.

- En el caso de FPGA recibe el nombre de *Place and Route*. El proceso de *Place and Route* tiene un impacto determinante en las prestaciones de velocidad del diseño. Partiendo de un *Placing* inicial condicionado entre otros factores por las directivas del diseñador, se ajusta iterativamente previniendo conseguir unas determinadas prestaciones. A continuación se inicia el *Routing* global y de pistes de reloj y posteriormente el *Routing* local.

Los diferentes software que hay en el mercado utilizan algoritmos de síntesis, optimización y ruteado propietarios. Es por esto que los resultados pueden ser muy diferentes de un caso a otro.

❖ Simulación del layout

La simulación de *layout* permite verificar funcional y temporalmente el diseño sobre el dispositivo escogido. En caso de que las prestaciones temporales no sean las adecuadas hay diversas alternativas:

- Resintetizar el diseño con directivas adicionales
- Revisar el código HDL e intentar optimizarlo
- Utilizar el mismo dispositivo pero con un *speedgrade* diferente
- Cambiar de dispositivo

❖ Programar el dispositivo

Una vez verificadas todas las prestaciones sólo habrá que programar el dispositivo a partir del fichero que el proceso especificado a 4.2.4 genera.

5.5. Lenguaje VHDL

En este apartado explicaremos los fundamentos de la programación en lenguaje VHDL. Para más detalles sobre la sintaxis, consultar la bibliografía. En primer lugar introduciremos el modelo de programación para dispositivos de lógica programable, comparándolo con el modelo orientado a procesador. A continuación describiremos aspectos básicos de la sintaxis del lenguaje VHDL.

5.5.1. Modelo de programación orientado a procesador

A continuación enumeramos algunos aspectos relevantes referentes al diseño e implementación de una aplicación software para ser ejecutada sobre un procesador.

El algoritmo a implementar se suele subdividir en funciones. Cada función implementa una tarea concreta a partir de unas entradas (parámetros de la función) y genera una o diversas salidas (variables de retorno). Esta estructuración modular en funciones no se efectúa con el objetivo de acelerar ni optimizar la ejecución del algoritmo, sino en beneficio de la claridad del código.
La ejecución del programa se inicia por la primera instrucción de una función especial denominada *main* (pienso en un programa en C). A partir de aquí existe un hilo único de ejecución marcado por un registro denominado normalmente *program counter*. Por tanto, la ejecución de la aplicación consiste en la secuenciación de una serie de instrucciones organizadas lógicamente dentro de funciones. Remarcamos que en cada momento, sólo una instrucción se ejecuta (en procesadores segmentados o súper escalar se pueden ejecutar diversas instrucciones simultáneamente; en cualquier caso, no obstante, serán instrucciones que en la definición del código son consecutivas).

5.5.2. Modelo de programación orientado a PLD

El modelo de programación definido en el apartado anterior no es válido para programar dispositivos de lógica programable. A continuación definimos el paradigma de programación de PLDs.

La aplicación se divide en dos partes. Por un lado, la entidad (*entity*) y por otro, la arquitectura (*architecture*). La entidad es la definición del número y tipo de entradas y salidas; es decir, la forma de relacionarse del módulo con el resto de módulos. La arquitectura, asociada a una entidad, define la forma concreta en que las salidas se relacionan con las entradas; es decir, la implementación propia del algoritmo. Por una determinada entidad

pueden existir diferentes tipos de arquitecturas (diversas implementaciones).

Dentro de la arquitectura, la aplicación se organiza en procesos y señales. El proceso es la implementación de una parte del algoritmo. La señal es el mecanismo de comunicación entre procesos o entre procesos y módulos externos a la arquitectura. Un proceso tiene asociada una lista de sensibilidad (*sensitivity list*) formada por el conjunto de señales de entrada al proceso.

Todos los procesos se ejecutan simultáneamente. No existe un único hilo de ejecución, sino que hay tantos como procesos tenga la arquitectura. Esto es posible ya que cada proceso se mapeará sobre una zona diferente de los dispositivos, y tendrá un hardware específico para ser ejecutado.

La secuencia de operaciones que se llevan a cabo para ejecutar un proceso es la siguiente:

1. Por defecto, todos los procesos están "dormidos"; en *standby* a la espera que alguno de las señales de la *sensitivity list* cambie.

2. Cuando este cambio se produce, se registran los valores de todas las señales de entrada al proceso y se dispara una ejecución de este proceso. Por tanto, cualquier cambio de las señales de entrada a un proceso que se produce posteriormente al inicio de la ejecución de este no tendrá efecto hasta la próxima ejecución del proceso.

3. Las sentencias dentro de un proceso se ejecutan de forma consecutiva, desde la primera hasta la última

4. Las señales modificadas dentro del proceso no adoptarán los nuevos valores hasta la finalización de la ejecución del proceso. Por tanto, después de la ejecución de la última tendencia del proceso, se actualizarán los valores de las señales modificadas dentro del proceso, y este pasará a estado *standby* a la espera de un nuevo cambio en alguna señal de la *sensitivity list.*

Remarquemos que aunque la organización en procesos de los programas orientados a PLD puede parecer análoga a la organización en función de los programas orientados a procesadores, y que las variables parecen señales, existen diferencias fundamentales en la forma en que se ejecutan. Remarquemos una vez más estas diferencias:

1. Las funciones no se pueden ejecutar en paralelo. Cada función se ejecuta sólo cuando se requiere. Por otro lado, los procesos se pueden ejecutar en paralelo. Un proceso inicia su ejecución cuando alguno de las señales de su *sensitivity list* cambia. Puede darse el

caso de que en un momento dado todos los procesos se ejecuten simultáneamente, y otro momento en que todos estén en *standby*.

2. Los valores de las variables de los programas orientados a procesadores se actualizan inmediatamente después de la ejecución de la instrucción afectada, mientras que el cambio en el valor de las señales:

    a. Si son de entrada a un proceso, el cambio no afecta durante la ejecución del proceso.
    b. Si son de salida, el cambio no se produce hasta el final de la ejecución de todo el proceso (¡no de la sentencia!)

5.5.3. Declaración de la entidad

La declaración de la entidad es equivalente a la descripción del símbolo esquemático. Aporta información, pues, de la conexión del componente con el resto del sistema.
Por ejemplo,

```
library ieee;
use ieee.std_logic_1164.all;
entity eqcomp4 is
        generic (size : integer := 4);
        port (
        a, b : in std_logic_vector (size-1 downto 0);
        ecuals : out std_logic);
end eqcomp4;
```

Código 1. Declaración de la entidad de un comparador de 4 bits

define un comparador de *size* bits (cuatro en este caso).
Declarar todas las entidades de un diseño en una fase inicial de un proyecto facilita la posterior repartición de la codificación de cada módulo entre diversos ingenieros.

Dentro de la entidad se pueden definir parámetros (en este caso *size*) que permiten configurar de forma explícita el módulo (*size es* la medida en bits de las señales de entrada *a y b*).

Cada señal de entrada o salida ha de ser definida como ***puerto*** que a la vez puede ser de entrada (***in***), salida (***out***), salida realimentada al circuito (***buffer***; las herramientas de síntesis no suelen permitir este modo) o bien entrada / salida (***inout***).

Cada señal puede contener tipos diferentes de datos que hay que especificar. Por ejemplo: *bit*, o *integer* son tipos admitidos por el IEEE 1076-87 y 1076-93.
El estándar IEEE 1164 define un nuevo tipo básico (*std_ulogic*) muy utilizado por simulación y síntesis. Este tipo define un sistema lógico de 9 posibles valores: además del '0' y '1', la alta impedancia ('Z'), el *don't care* ('-') y otros. Para usar *std_ulogic y* sus derivados hay que incluir el *package* correspondiente a la cabecera del fichero que contiene las declaraciones (ver código 1).

A partir de cada tipo básico se pueden construir arrays (buses). Por ejemplo, el odi 1, *std_logic_vector es* un tipo predefinido: un vector de *n* elementos *std_logic* (en este caso *n=size=4, y* el bit más significativo será el que tiene un índice mayor). Se pueden crear nuevos tipos además de los ya existentes con la palabra clave ***type***.

5.5.4. Cuerpo de la arquitectura

Cada cuerpo de la arquitectura está asociado con una declaración de la entidad. De hecho, el cuerpo de la arquitectura describe los contenidos de la entidad. Hay tres estilos para implementar la arquitectura: comportamiento (*behavioral*), flujo de datos (*dataflow*) o estructural (*estructural*). Estos tres estilos permiten diferentes niveles de abstracción: desde la utilización de construcciones de muy alto nivel hasta la definición del comportamiento a nivel de puerta.

❖ Behavioral

El estilo *behavioral* corresponde a la descripción del comportamiento de la entidad en forma algorítmica. De los tres estilos éste es el de más alto nivel y como tal presenta la ventaja de permitir la máxima inteligibilidad y concentración de esfuerzo en el comportamiento a modelar y el inconveniente de encontrarse muy alejado de la estructura lógica que acabará implementando la función.

```
architecture behavioral of eqcomp4 is
begin
        comp : process (a, b)
        begin
                if a=b then
                        ecuals <='1';
                lose
                        ecuals <= '0';
                end if;
        end process comp;
```

```
end behavioral;
```

Codi 2. Cuerpo de la arquitectura de un comparador de 4 bits (*behavioral*)

El *keyword* ***architecture*** permite relacionar el cuerpo de la arquitectura con la declaración de la entidad (*eqcomp4*). El cuerpo de la arquitectura en sí contiene el ***process*** (algoritmo) *comp*, que se ejecutará cuando alguno de las señales especificadas en la *sensitivity list* (en este cas *a* o *b*) cambie. A continuación del ***begin***, una secuencia de comandos que definen la forma en que las salidas reaccionan a las entradas. El orden de los comandos es importante (más adelante veremos cómo se ejecutan estos comandos). Un ***end*** del proceso y de la arquitectura delimita el final del cuerpo.
Una arquitectura puede contener diversos procesos que se ejecutarán parleramente. Ahora bien, dentro de un proceso los comandos se ejecutan secuencialmente.

Los procesos que no tienen *sensitivity list* se ejecutan indefinidamente a no ser que se utilice el comando *wait* en cualquiera de sus formas (*wait for*, *wait until*, etc.).

Aunque se pueden poner los *wait*s en cualquier punto del proceso, se recomienda hacerlo al principio o al final. Si se utiliza *sensitivity list* el proceso no debería de tener ningún *wait*.

❖ Dataflow

Se utiliza el estilo *dataflow* cuando se especifica la forma en que las salidas responderán a las entradas sin utilizar una secuencia de comandos. En la práctica, *behavioral* está basado en la utilización de procesos mientras que *dataflow* no. El estilo *dataflow* se utiliza preferentemente cuando el comportamiento de la arquitectura se puede especificar mediante ecuaciones sencillas, estructuras condicionales (*when – lose*) o de selección (*with – select – when*).

En general, cuando se necesite especificar algoritmos complejos, anidados será más fácil utilizar secuencias de comandos (*behavioral*).

```
architecture dataflow of eqcomp4 is
begin
        ecuals <= '1' when (a=b) lose '0';
end dataflow;
```

Código 3. cuerpo de la arquitectura de un comparador de 4 bits (*dataflow*)

Un aspecto muy importante a tener en cuenta es que los comandos dentro de un proceso (*process*) se ejecutan de forma secuencial, mientras que

fuera del proceso (*dataflow*) todos los comandos se ejecutan concurrentemente, en paralelo (ver 5.7.1).

❖ Structural

El Código 4 corresponde a la arquitectura de comparador especificada en estilo estructural.

```
use work.gatespkg.all;
architecture struct of eqcomp4 is
        signal x : std_logic_vector (0 to 3)
begin
        u0 : xnor2 port map (a(0), b(0), x(0));
        u1 : xnor2 port map (a(1), b(1), x(1));
        u2 : xnor2 port map (a(2), b(2), x(2));
        u3 : xnor2 port map (a(3), b(3), x(3));
        u4 : and4 port map (x(0), x(1), x(2), x(3), ecuals);
end struct;
```

Codi 4. Cuerpo de la arquitectura de un comparador de 4 bits (*structural*)

componentes que implementarán el diseño. En este caso, las puertas lógicas *xnor2 y and4* (contenidas en la librería *gatespkg*) se combinan a través de las entradas (*a*, *b*), variables locales (*x*) y salidas (*ecuals*) de la forma que explícitamente se indica en el cuerpo de la arquitectura. El estilo estructural es jerárquico en el sentido que dentro de *eqcomp4* se utilizan otros componentes (*xnor2 y and4*) que han sido definidos en otros ficheros (*gatespkg*) y ahora se enlazan para formar el diseño.

Si comparamos el Código 4 con los anteriores se observa que el estilo estructural tiene forma de *netlist y* que, por tanto, ofrece un control muy grande sobre la implementación física.

Permite, pues, optimizar a nivel de puerta. Para grandes diseños, no obstante, hay que utilizar profusamente los niveles jerárquicos que aunque facilitan la simulación dificultan la comprensión general del algoritmo y alarga el tiempo de desarrollo.

Otro inconveniente del estilo estructural es la dificultad para soportar de forma automática módulos configurables mediante parámetros definidos en la declaración de entidad (en este caso, *size.* Ver Código 1). Si cambiamos el valor de *size*, habrá que añadir o quitar líneas de código a Código 4.

❖ Estilos e implementación

Seguidamente habría que preguntarse si la elección del estilo de programación condiciona la optimización de la implementación. Para diseños pequeños, los resultados son muy similares. Ahora bien, para diseños complejos las diferencias pueden ser grandes, principalmente por tres factores:

- La especificación lógica escrita por el programador puede no ser óptima desde el punto de vista de utilización de recursos lógicos, y la herramienta de programación puede no saber simplificarla de la forma más adecuada.

- La herramienta de síntesis, optimización y mapeo puede no utilizar óptimamente los recursos de la arquitectura. Por ejemplo, habrá que sintetizar un mismo código en forma de suma de productos para dispositivos CPLD o bien módulos basados en LUT para dispositivos FPGA.

- La herramienta de mapeo puede no ser óptima a la hora de escoger el emplazamiento de los recursos lógicos.

Por tanto, especificaciones diferentes pueden dar lugar a implementaciones funcionalmente idénticas, pero diferentes en lo que respecta al uso de recursos y prestaciones temporales.

5.5.5. Categorías, tipo de datos y atributos

En el apartado anterior hemos visto que para iniciar un diseño en primer lugar hay que cal cuáles son sus entradas y salidas (declaración de la entidad) y seguidamente especificar cómo se relacionarán a nivel de comportamiento las salidas con las entradas (cuerpo de la arquitectura). Además, hemos visto que el cuerpo de la arquitectura se puede introducir siguiendo diferentes filosofías que ofrecen mayor o menor abstracción al mismo tiempo que mayor o menor control de la implementación.

A continuación nos centraremos en aspectos de sintaxis del lenguaje VHDL referentes a la definición y tipo de datos que éste ofrece.

❖ Categorías de datos

Destacamos tres categorías de datos: constantes, señales y variables:

- ***Constantes***

Utilizar constantes facilita la lectura y comprensión del código. A continuación un ejemplo de la declaración de una constante:

constante *anchura : integer := 8;*

El alcance de la constante se extiende al interior de la construcción donde se declare (ya sea un *package*, una entidad, una arquitectura, o un proceso).

Para diferenciar los objetos definidos como constantes de los definidos dentro del bloque *generic* de la declaración de la entidad (ver Código 1) se suele usar el convenio siguiente: los valores definidos como *generic* pueden ser cambiados con el fin de configurar el código sin necesidad de retocarlo, mientras que los definidos como constante son totalmente estáticos; si se cambian hay que retocar el código; se usen únicamente para facilitar la comprensibilidad del programa.

- ***Señales***

Las señales en VHDL son los objetos básicos de comunicación entre módulos. Representan entradas o salidas a dispositivos o procesos. Se definen de la forma siguiente:

signal *comptador : bit_vector (3* ***downto*** *0);*

Las señales declaradas como puertos (dentro de la declaración de la entidad, ver Código 1) tienen asignado un modo (entrada, salida, etc.). Por contra, los declarados dentro de la arquitectura son de lectura y escritura y se pueden usar para pasar valores entre procesos o para comunicación con el exterior.

Para asignar valores a señales se utiliza el símbolo '<='. En general, la asignación no es inmediata. Por ejemplo, cuando se utilizan señales dentro de procesos, la actualización del su valor no se hace hasta que se suspende el proceso (ver 5.7.1).

Los *signals* no se pueden declarar dentro de los procesos.

- ***Variables***

Las variables sólo se pueden definir localmente en procesos, funciones o procedimientos.

No representan conexiones entre dispositivos ni elementos de memoria, sino que se suelen usar para modelar y simular o bien para almacenar cálculos intermedios.

A continuación un ejemplo de declaración de una variable:

variable *resultado : std_logic := '0';*

La asignación de un valor a una variable se realiza mediante el símbolo ':='. A diferencia de las señales, esta asignación es inmediata.

No se recomienda utilizar variables en un diseño que se tenga que sintetizar ya que la síntesis de variables no está bien especificada.

❖ Tipo de datos

Cada una de las categorías anteriores puede declararse con tipo de datos diferentes: escalares (enumerado, *integer*, real y físico) y compuestos (*array y record*).

Veámoslo a continuación.

- ***Enumerado***

Un tipo enumerado puede contener una lista de valores posibles predefinidos.
Algunos ejemplos de la definición de tipo y declaración y uso de variables:

```
type boolean is (FALSE, TRUE);
type bit is ('0', '1')
type verdura is (col, espinac, bleda);
(...)
signal encert : boolean;
signal caixa : verdura;
(...)
caixa <= bleda;
```

Existen algunos enumerados ya definidos: *boolean y bit* (IEEE 1076) y *std_ulogic y std_logic* (IEEE 1164).

- ***Integer***

El tipo integer permite valores numéricos enteros dentro del intervalo $\left[-\left(2^{31}-1\right),\left(2^{31}-1\right)\right]$. Cuando se defina una señal o una variable de este tipo hay que especificar su rango para optimizar el uso del hardware,

***variable** a : **integer range** -255 to 255;*

- ***Real o floating***

El tipo real permite valores numéricos con coma flotante dentro del intervalo $[1e-38,1e38]$ como mínimo (depende de la arquitectura).

- ***Físico***

El único tipo físico definido es el tiempo (*time*). Se utiliza sólo para medir el tiempo en simulaciones (no tiene ningún sentido en síntesis).El intervalo mínimo respecto a la unidad por defecto (en segundos) es el de *integers*.

- ***Array***

Un dato de tipo *array* contiene múltiples elementos del mismo tipo básico. Los standards IEEE 1076 y IEEE 1164 definen tres:

```
type bit_vector is array (natural range <>) of bit;
type std_ulogic_vector is array (natural range <>) of std_ulogic;
type std_logic_vector is array (natural range <>) of std_logic;
```

La medida del *array* se especifica cuando se declara la variable:

```
signal a : std_logic_vector (3 downto 0);
```

- ***Record***

Un *record* o estructura contiene diversos elementos de tipo diferentes. Se definen, declaran y usan de la forma siguiente:

```
type orden is record
tipo : boolean;
valor : integer;
end record;
(...)
signal a : orden;
(...)
a.tipo <= FALSE;
```

❖ Atributos

Los atributos permiten obtener información adicional de señales.

Los tipos escalar tienen los atributos *'left, 'right, 'high, 'low, 'length.* En el caso de un *integer*, por ejemplo, *'left* retorna el valor más a la izquierda

(menor si se ha definido ascendente, el mayor si está definido descendentr), *'right* el valor más a la derecha, *'high* el máximo valor que puede representar, *'low* el mínimo y *'length* retorna el número de elementos de un *array*.

Otro atributo importante es 'event, que evalúa cierto cuando la señal sobre el que se aplica ha cambiado. Se suele asociar a señales de reloj con estructuras parecidas a:

__if__ (CLK__'event and__ CLK=1) __then__ -- para detectar un flanco de subida
(...)

Hay que tener en cuenta que *CLK'event* evaluará cierto para cualquier transición del señal CLK (incluidas 'X' a '1', 'U' a '0', etc.). Si queremos detectar únicamente flancos de subidada o bajada sobre señales std_logic podemos usar las funciones *rising_edge y falling_edge*.

❖ Operadores

- ***Operadores y asignación condicional***

  A menudo los comandos que escribiremos en VHDL responderán a la estructura siguiente:

  salida <= función (entradas)

  Es decir, las entradas combinadas mediante operadores generarán las salidas. En este apartado veremos los operadores más importantes de VHDL.

- ***Operadores lógicos***

  Operan sobre tipo de datos *bit*, *boolean* o *bit_vector*.
  Son *and, or, nand, nor, xor, xnor y not*.
  Los operadores lógicos en VHDL no tienen orden de procedencia: se ejecutan siempre de izquierda a derecha. Se pueden utilizar paréntesis para modificar esta regla.

- ***Operadores numéricos***

  Operan sobre tipo *integer* o *real* (también sobre *std_logic* con o sin signo). Los operandos han de ser del mismo tipo (no hay *cast* automáticos). También se puede utilizar el tipo *time*: en el caso de la suma y la resta todos dos operandos han de ser *time*, mientras que en caso de producto o división uno ha de ser *time y* la otra *integer* o *real*.

Son *+, -, *, /, mod, rem, ** y abs*. La división (/), módulo (*mod*), residuo (*rem*), exponencial (**) y valor absoluto (*abs*) raramente se utilizan por diseños que se tengan que sintetizar.

- ***Operadores relacionales***
  Se utilizan para comparar valores de cualquier tipo. El resultado siempre es *boolean*.
  Son =, /= (diferente), <, <=, >, >=.

- ***Operadores de shiftat***
  Se utilizan sobre *arrays* de *bit* o *boolean*.
  Los clasificamos en tres subtipos:

  - Shiftado lógico, izquierda (*sll*) o derecha (*srl*).
  - Shiftado aritmético, a izquierda (*sla*) o derecha (*sra*)
  - Rotación, a izquierda (*rol*) o a derecha (*ror*)

- ***Operadores de concatenación***
  El operador (&) permite crear un *array* a partir de la concatenación de otros arrays.

5.5.6. Construcciones dataflow

❖ with – select – when

La construcción siguiente no es un operador. Se usa en algoritmos *dataflow* para asignar selectivamente a una *señal_destino* según el valor de una *señal_selección*.

```
with señal_selección select
señal_destí <= valor1 when valor1_señal_selección,
señal_destí <= valor2 when valor2_señal_selección,
...
señal_destí <= valor1 when others;
```

Es conveniente que la última asignación complete el universo de posibilidades de valores de *señal_selección* con la directiva *when others*. Hay que Cal tener en cuenta que si *señal_selección* se ha definido *std_logic* además de '0' y '1' puede valer también alta impedancia ('Z'), *don't care* ('-') y otros. Si no contemplamos estas posibilidades dentro de la construcción, *señal_destino* se tendrá que implementar con un elemento de memoria asíncrona realimentada por los casos no considerados. Acabando la construcción con una asignación *when others* evitaremos este problema.
En resumen, hay que asignar un valor a la señal para todos los posibles casos de la condición ya que dejar casos sin considerar implica el uso de memoria que puede llevar a contigüidades. Problema.

Esta misma norma de estilo es válida para cualquier otra construcción condicional que implique asignaciones (*when –lose*, *if – then – lose*, etc.)

❖ when – lose

De naturaleza muy similar al anterior, la construcción *when – lose* permite una asignación condicional en algoritmos *dataflow*:

```
Señal_destino <= valor1 when condición1 lose
valor2 when condición2 lose
...
valorx;
```

5.5.7. Control de flujo

Las construcciones siguientes se usan dentro de procesos (especificaciones *behavioral*).

❖ if - then – lose

Se utiliza la sintaxis siguiente:

```
if condición1 then
        acción si condición1 evalua cierto;
losif condición2 then
        acción si condición 2 evalua cierto;
        (...)
lose
        o bien;
end if;
```

Código5. *if – then – lose*

La construcción es similar a *when – lose*.

❖ case – when

Se utiliza la sintaxis siguiente:

```
case señal is
        when valor1 => acción1;
        when valor2 => acción2;
        (...)
        when others => acción de otro modo;
end case;
```

Código 6.. *case – when*

La construcción es similar a *with – select - when*.

❖ for - loop

La estructura de interacción *for – loop* ejecutará una secuencia de comandos un número determinado de veces. La sintaxis es:

```
for (variable) in (interval) loop
        comanda1;
        (...)
        comandaN;
end loop;
```

Codi 7. *for –loop*

No hay que declarar previamente la variable de la iteración

El intervalo se suele dar utilizando las palabras clave *to* (0 *to* 3) o bien *downto* (4 *downto* 1).

El comando *exitse puede* utilizarse para salir del bucle en cualquier momento.

❖ while – loop

Se utiliza para iterar un número indeterminado de veces, limitado por una condición de mantenimiento:

```
while condición loop
        comanda1;
        (...)
        comandaN;
end loop;
```

Codi 8. *while – loop*

❖ wait until

Permite parar la ejecución de un proceso hasta que evalúe una condición. Esta construcción hay que utilizarla como alternativa (no simultáneamente) a una *sensitivity list*. Para diseños que se tengan que sintetizar se recomienda que sólo haya instrucciones *wait - until* al principio o al final del proceso.

```
wait until condición;
```

Código 9. *wait until*

5.5.8. Consideraciones generales

❖ Sobre los procesos

Los procesos (*process*) pueden estar o bien en espera o bien ejecutándose. Se inicia la ejecución cuando se produce un cambio (event) en alguna de las señales de su *ensitivity list*. Por lo tanto, secuencialmente se evalúan los comandos hasta llegar al último. Aunque los simuladores permiten parar un proceso en cualquier punto de la secuencia de comandos utilizando un *wait* (por ejemplo, *wait sg*, on *sg es* una señal cualquiera), la mayoría de los sintetizadores sólo soportan el uso de esta comando al principio o al final del proceso.

Los comandos de asignación no son efectivos hasta que se han ejecutado todas las del proceso al que pertenecen.

Veámoslo con los tres ejemplos independientes siguientes,

```
c1: process (a, b, c)
begin
        x <= a and b;
        y <= x and c;
end process

c2: process (a, b, c)
begin
        y <= x and c;
        x <= a and b;
end process

c3: process (a, b, c)
begin
        y <= a and b and c;
end process
```

Codi 10. Código ejemplo para mostrar el uso de señales en procesos

Los procesos *c1 y c2* donen el mismo resultado, diferente de *c3* (hay que tener en cuenta que a *c1 y c2* el valor de *x* no se actualiza hasta el final de cada proceso respectivamente).

❖ Programación behavioral y structural

La construcción de una arquitectura a base de muchos procesos (*behavioral*) puede resultar complicada si estos procesos son sencillos. Consideremos, por ejemplo, el código siguiente:

```
architecture esta of cualquiera is
signal Int1, Int2 : bit;
begin
        porta1: process (A, B)
        begin
                Int1 <= A or B;
        end process porta1;

        porta2: process (A, C)
        begin
                Int2 <= A and C;
        end process porta2;

        porta3: process (Int1, Int2)
        begin
                Out <= Int1 xor Int2;
        end process porta3;
end architecture esta;
```

Código 11. Arquitectura con muchos procesos sencillos (*behavioral*)

VHDL permite simplificar este diseño de la forma siguiente:

```
architecture esta of cualquiera is
signal Int1, Int2 : bit;
begin
        Int1 <= A or B;
        Int2 <= A and C;
        Out <= Int1 xor Int2;
end architecture esta;
```

Código 12. código simplificado (*structural*)

Ambos códigos son idénticos a nivel funcional. Las asignaciones situadas dentro de una arquitectura fuera de los procesos se denominan *concurrent signal assignoments y* se ejecutan en paralelo entre sí y con los procesos.

La *sensitivity list* dlos *concurrent signal assignments* contiene de forma implícita todas las señales a la derecha de la asignación.
Hay otras estructuras que se ejecutan siguiendo la misma filosofía:

- Conditional signal assignment

  a <= b ***when*** *c='1'* ***lose*** *d;*

- Selected signal assignment

  with *mode* ***select***
  a <= b ***when*** *'00',*
  b or c ***when*** *'11',*
  '0' ***when others****;*

❖ Función de resolución

Conviene evitar múltiples asignaciones a una misma variable a no ser que el tipo de datos de la variable asignada disponga de funciones de resolución (*resolution function*).
Por ejemplo,

y <= a ***and*** *b;*
y <= a ***or*** *b;*

Una función de resolución asignaría a *y* un valor lógico '0' o '1' si las dos expresiones evalúan el '0' o '1' respectivamente. En cambio, si evalúan diferente, *y* contendría el valor 'X' (indeterminado).

Otros valores metalógicos interesantes son el *don't care* '-' y alta impedancia 'Z'. El *don't care* se utiliza para de permitir al sintetizador simplificar una determinada expresión.

En la práctica, *don't care* valdrá '0' o '1' según le convenga al optimizador. La alta impedancia se utiliza para referir el tercer estado de un *buffer tri-state*.

❖ Librerías y packages

Cuando se requiere utilizar alguna funcionalidad no definida en los estándares de VHDL hay que recorrer al uso de librerías. Para tal efecto se utilizan los comandos *library y use*.

Library hace accesible la librería que contiene la funcionalidad que se quierel utilizar. Una librería está formada por diversos *packages*, y cada *package* por diversos ítems.

El comando *library* por sí solo no permite utilizar su contenido. Esto se consigue mediante *use*. La sintaxis de este comando es:

```
use library_name.package_name.item
```

Si no se quiere explicitar el *item* puede usarse el comando *all*.

```
library ieee;
use ieee.std_logic_1164.all;
```

Algunos de los *packages más* utilizados son el *Standard* (no hay que declararlo; el compilador lo utiliza directamente), *textIO* para manipular textos (sólo se usa para simulación o modelado; hay que incluir con *use Std.TextIO.all*) y el ya comentado *std_logic_1164* (no es un IEEE-1076, pero es un estándar IEEE por sí solo; hay que incluirlo con *use IEEE.Std.TextIO.all*).

El usuario puede crear sus propios *items*, almacenarlos en librerías propias y usarlos en otros diseños (de la misma forma que puede hacerlo cuando programa en otros lenguajes como el C). Este mecanismo permite reaprovechar software y ganar en modularidad y claridad del diseño. Diseño basado en máquinas de estados síncronos.

5.6. Implementación de máquinas de estados síncronos

5.6.1. Aplicación a la UART

En este apartado explicaremos cómo diseñar una máquina de estados síncrona en VHDL a partir de un ejemplo concreto: el diseño de un puerto serie asíncrono (UART). El ejemplo también nos servirá para presentar una posible metodología de diseño de aplicaciones usando el modelo de programación orientado a hardware.

5.6.2. Especificaciones

Un puerto serie asíncrono permite la comunicación de datos (en este caso palabras de 8 bits) a través de sólo dos líneas de comunicaciones (una por recepción y otra por transmisión). Es un sistema lento pero que requiere un número mínimo de señales. El protocolo típico se describe a continuación:

- Por defecto la línea de comunicación está a '1'. es el estado de reposo.

- La comunicación empieza con un *start bit* que consiste en la bajada de la línea a '0' durante, normalmente, un tiempo de bit.

- A continuación se envían los 8 bits de la palabra; cada uno dura un tiempo de bit. En este diseño supondremos que el primer bit que se envía es el de más peso (*Little endian*)

- La comunicación finaliza manteniendo la línea a '1' durante un tiempo de bit (*stop bit*). Una vez ha transcurrido este tiempo, se puede iniciar la transmisión de una nueva palabra.

El inverso del tiempo de bit es la frecuencia de envío de datos del puerto serie. Diseñaremos una UART de forma que la frecuencia del puerto sea configurable, con un margen de valores del orden de Khz.

5.6.3. Diseño

Planteemos el diagrama de bloques simplificado siguiente. Es un diagrama de bloques orientado al diseño de hardware; es por esto que aparecen componentes como contadores, comparadores, registros, etc. Al plantearlo

estamos pensando en la implementación propia del lenguaje VHDL: una arquitectura formada por un conjunto de procesos que se ejecutan en paralelo.
Algunos de los bloques especificados en la figura 130 acabarán en procesos; otro no. Lo veremos más adelante. En cualquier caso, cuanto más detallado sea el diagrama más sencillo será después la implementación.

A la figura 130 presentamos un diagrama completo por emisión y recepción. A continuación, no obstante, sólo implementaremos la parte de emisión, dejando como ejercicio la de recepción.

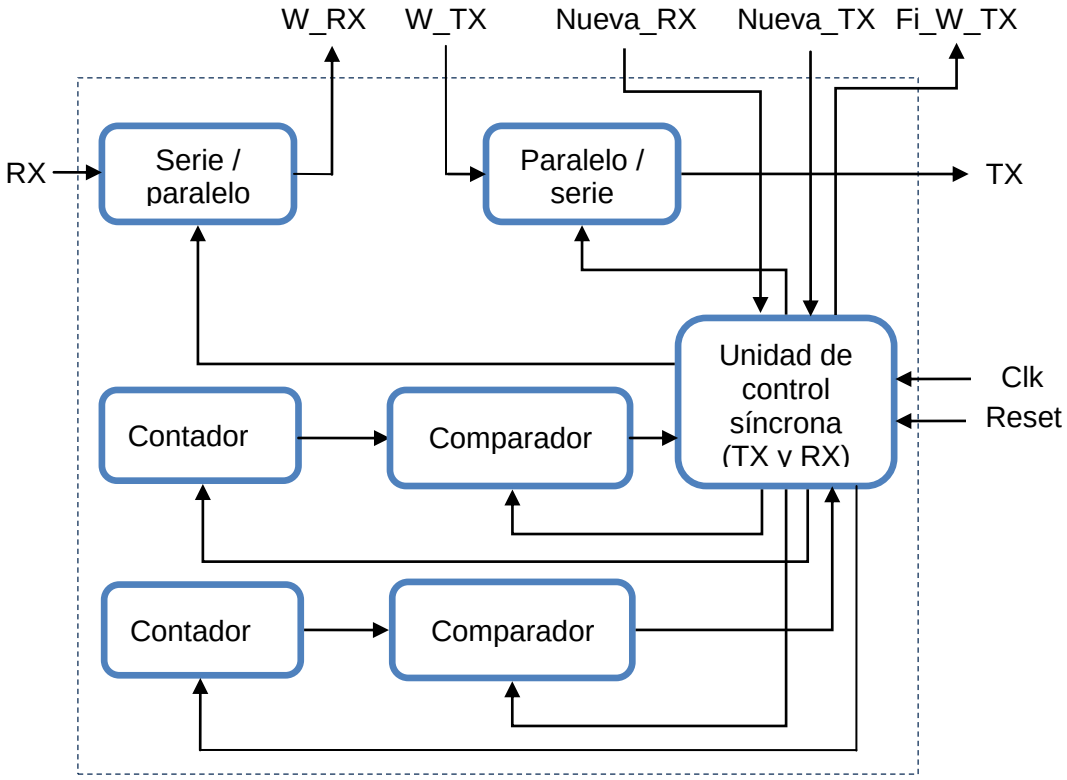


Figura 130. Diagrama de bloques de la UART

En recepción, un registro paraleliza los datos recibido por la línea del puerto serie RX. En su salida, W_RX (*word_RX*: 8 bits) contiene la última palabra recibida; la recepción se señalizará con lal señal Nova_RX
En transmisión, un registro serializa la palabra a enviar (W_TX: 8 bits) señalizada con la señal Nova_TX (entrada al módulo que indica la presencia de una nueva palabra a enviar) hacia la línea de transmisión del puerto serie (TX). Una vez enviado el dato, el módulo avisa de que el proceso ha finalizado (Fi_W_TX).

Un comptador y un comparador servirán para fijar el tiempo de bit, en función del reloj del sistema. Otra pareja contador / comparador permitirán contar el número de bits recibido.

Una máquina de estados síncrona controlará todo el proceso: indicará cuándo ha de contar cada contador, consultará las salidas de los comparadores, activará las señales de control serie y paralelo de los registros, etc. En definitiva, secuenciará el uso del hardware para conseguir la función deseada.

Recalcamos que con el término 'síncrona' aplicado a la máquina de estados hacemos referencia a que los cambios de estado que se producirán simultáneamente con los flancos de subida del reloj del sistema.

Veamos un ejemplo del diagrama de estados para la máquina síncrona que secuencia el proceso de emisión (figura 131). Por cada estado, apuntamos lo que se hará y los motivos de las transiciones a otros estados.

Tengamos presente que las posibilidades a la hora de diseñar esta máquina son muchas. Aquí presentamos una. En cualquier caso, el número de estados y de iteraciones fijará la optimización de la implementación en lo que respecta al número de ciclos de reloj que serán necesarios para finalizar el proceso.

Lo veremos con más claridad en diseños posteriores (filtros y similares)

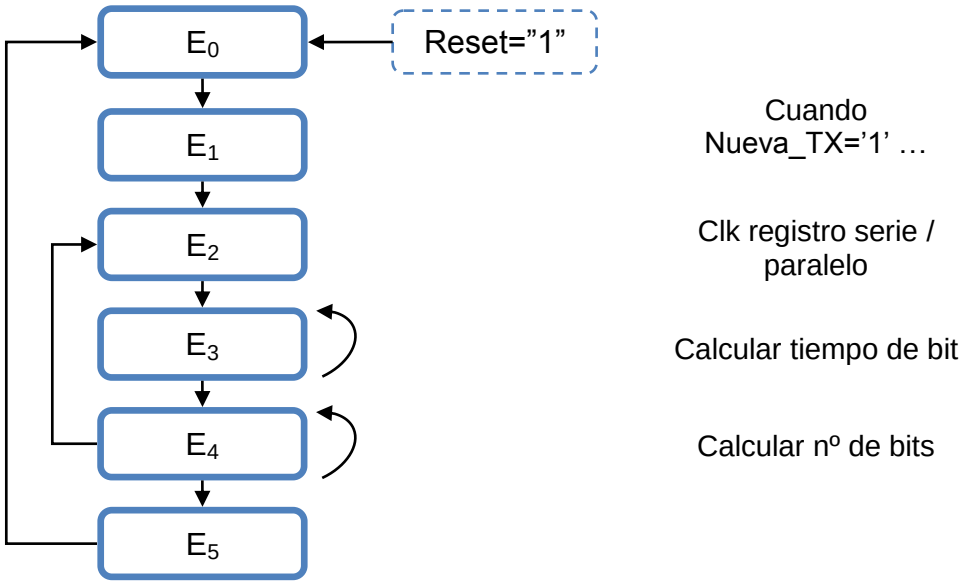


Figura 131. Máquina de estados simplificada de la UART en emisión

5.6.4. Implementación del módulo de emisión

Diseñaremos por separado el módulo de emisión y el de recepción. Veamos en primer lugar la declaración de la entidad, de acuerdo con la figura 130.

Para fijar la frecuencia de bit de la transmisión serie definiremos un parámetro (*generic*) *n* que contendrá el cociente entre la frecuencia de reloj del sistema (en este caso 60 Mhz) y la frecuencia de bit.
Por una frecuencia de bit de 9600 bps, el valor de n es 6250 ('1100001101010' en binario).

Veamos, a continuación, como declara la entidad el módulo emisor de la UART:

```
entity TX_UART is
generic (n:std_logic_vector (15 downto 0):="0001100001101010");
        port(
                clk:            in STD_LOGIC;
                reset:                  in STD_LOGIC;

                W_TX:           in STD_LOGIC_VECTOR(7 downto 0);
                nova_TX:        in STD_LOGIC;
                fi_W_TX:        out STD_LOGIC;

                TX:             out STD_LOGIC
            );
end TX_UART;
```

Código 13. Declaración de la entidad del módulo emisor de la UART

Organizaremos la arquitectura de la forma siguiente. Un contador de 16 bits para fijar la frecuencia de bit (permitirá una velocidad serie mínima alrededor de de 900 bps); un contador de 4 bits para llevar el control del bit que se está serializando en cada momento; un registro paralelo serie; una máquina de estados para gestionarlo todo.

En comparación con el diagrama de bloques de la figura 128, remarcar que los comparadores no los implementaremos como procesos separados en la arquitectura, sino que usaremos las sentencias de VHDL para implementarlos (esto mismo lo hubiésemos podido hacer con los contadores; mezclaremos las dos técnicas para poderlas exponer).

Veamos la implementación de un contador de 16 bits que cuenta períodos de reloj dentro de un tiempo de bit (el contador que cuenta el número de bits sería muy similar a este, pero con un *enable*):

```
signal q_compt16: std_logic_vector (15 downto 0);
signal clr_compt16: std_logic;

compt16: process(clk, clr_compt16)
begin
        if clr_compt16 = '1' then
                q_compt16 <= (others => '0');
        losif clk'event and clk='1'  then
                q_compt16 <= q_compt16 + 1;
```

```
        end if;
end process;
```

Código 14. Implementación de un contador de 16 bits

Las señales *q_compt16* (la salida del contador) y *clr_compt16* (el reset del contador) los gestionará la unidad de control. Fijémonos en que en esta implementación el reset (*clr_compt16*) es prioritario sobre el orden de contar.

Para implementar un *enable* síncrono, tendríamos que modificar el código de la forma siguiente:

```
(...)
signal ena_compt16: std_logic;
(...)
losif clk'event and clk='1'  then
        if ena_compt16 = '1' then
                q_compt16 <= q_compt16 + 1;
        end if;
end if;
(...)
```

Codi 15. Modificación del código para añadir una señal de *enable* síncrono

Planteemos a continuación la implementación de un registre paralelo / serie adaptado a las necesidades de la transmisión serie:

```
signal mem: std_logic_vector (10 downto 0);
signal ena_s: std_logic;
signal reset_s:std_logic;

tx_ps_reg_p: process (clk, nova_TX)
begin
        if clk'event and clk='1' then
                if nova_TX = '1' then
                        mem <= '1' & W_TX (7 downto 0) & "01";
                end if;
        end if;
end process;

tx_ps_reg_s: process (clk, reset_s, ena_s)
variable i:integer range 0 to 10;
begin
        if reset_s = '1' then
                TX <= '1';
                i := 0;
        losif clk'event and clk='1' then
                if ena_s = '1' then
                        TX <= mem(i);
                        i := i+1;
                end if;
```

```
        end if;
    end process;
```

Código 16. Implementación de un registre paralelo / serie, adaptado.

La memoria del registre (señal *mem*) no es de 8 bits sino 11, para almacenar el estado de reposo de la línea ('1'),el *start bit* ('0'), los ocho bits a enviar y el *stop bit* ('1'). La implementación está organizada en dos procesos: el primero registra la señal a enviar y el segundo serializa este dato. A destacar el uso de una variable (*i*) al segundo proceso para indexar el bit a serializar en cada momento.

Veamos a continuación la implementación de la máquina de estados síncrona que secuenciará la ejecución de los anteriores procesos.

```
type ESTATS is (E0, E1, E2, E3, E4, E5);
signal estat:ESTATS;
signal próximo_estado:ESTATS;

-- Señales de control del contador de 4 bits: reset, enable, salida
signal clr_compt4: std_logic;
signal ena_compt4: std_logic;
signal q_compt4: std_logic_vector (3 downto 0);

-- Las señales de control del contador de 16 bits y el registro están
definidas  en los códigos correspondientes (Código 14 y Código 16)

begin

tx_Maq_C: process (reset, estat, nueva_TX, q_compt4, q_compt16)
begin
    if reset='1' then
        próximo_estado <= E0;
    lose
        clr_compt4 <= '0';
        ena_compt4 <= '0'
        clr_compt16 <= '1';
        ena_s <= '0';
        reset_s <= '0';
        fi_W_TX <='0';
        case estado is
            when E0 =>
                clr_compt4 <= '1';
                reset_s <= '1';
                if nova_TX='1' then
                    próximo_estado <= E1;
                lose
                    próximo_estado <= E0;
```

```
                        end if;
                    when E1 =>
                        próximo_estado <= E2;
                    when E2 =>
                        ena_compt4 <= '1';
                        ena_s <= '1';
                        próximo_estado <= E3;
                    when E3 =>
                        clr_compt16 <= '0';
                        if q_compt16 = n then
                            próximo_estado <= E4;
                        lose
                            próximo_estado <= E3;
                        end if;
                    when E4 =>
                        if q_compt4 = "1010" then
                            próximo_estado <= E5;
                        lose
                            próximo_estado <= E2;
                        end if;
                    when E5 =>
                        fi_W_TX <= '1';
                        próximo_estado <= E0;
                end case;
        end if;
    end process;

    tx_Maq_S: process (clk)
    begin
        if clk'event and clk=1' then
            estado <= próximo_estado;
        end if;
    end process;
```

Codi 17. Implementación de la máquina de estados síncrona de la UART

La enumeración de estados posibles de la máquina se implementa creando un tipo *ESTATS* que contiene los estados posibles (en este caso, desde E0hastaE5). A continuación hay que crear dos señales de este tipo: una de ellas marcará el estado de la máquina (*estat*) y la otra la próxima transición de la máquina (*próximo_estado*). Hacerlo así asegura una máxima sincronía entre el reloj del sistema y las transiciones de la máquina.

La máquina de estados se implementa con dos procesos.

Uno de ellos es de naturaleza combinacional (*tx_Maq_C*). Tiene las funciones siguientes:

- Fija la próxima transición actualizando la señal *próximo_estado* según el valor de las entradas (*nueva_TX*, *q_compt4*, etc.)

- Fija el valor de las señales de salida de la máquina de estados (*clr_compt4*, *ena_compt4*, etc.) en función del estado actual (*estat*).

El otro proceso es síncrono (*tx_Maq_S*). Su única función es cambiar el estado (*estado <= próximo_estado*) síncronamente con el flanco de subida del reloj.

5.7. Implementación de filtros FIR

5.7.1. Introducción

A la hora de plantear la implementación de un algoritmo cualquiera sobre un dispositivo de lógica programable existirán, en general, dos posibles enfoques:

- Intentar minimizar la cantidad de hardware empleado. En este caso los recursos se reaprovecharán. La implementación suele estar basada en una máquina de estados síncrona que secuencia el uso de los recursos. Las iteraciones serán a menudo necesarias.

- Intentar maximizar la frecuencia a la cual puede trabajar el módulo. Entonces, paralelizar la ejecución será en general necesario. La cantidad de hardware usado para ejecutar la aplicación será grande.

El primer enfoque corresponde a filosofía de procesador. En un procesador, los recursos hardware son, en general, pocos. Si nos fijamos en los aritméticos, podrían ser por ejemplo un sumador y un multiplicador. Si planteamos la implementación de un filtro FIR de *N taps*, el sumador y el multiplicador se tendrán que reutilizar para cada *tap.* La máquina de estados síncrona es la unidad de control, que secuencia el uso del sumador y del multiplicador según las instrucciones del programa. Si trasladamos esta misma idea a un dispositivo de lógica programable, definiremos un sumador, un multiplicador y una máquina de estados síncrona específica para ejecutar la aplicación de filtrado (específica a diferencia de la unidad de control del procesador, que es genérica). En general, para calcular la salida a partir de la entrada habrán de pasar N periodos de reloj, pero la complejidad del hardware es unitaria (independiente de N).

El segundo enfoque permite aprovechar al máximo las posibilidades de paralelismo de los dispositivos de lógica programable. Siguiendo con el mismo ejemplo, un filtro FIR de N *taps* necesita N multiplicadores y N-1 sumadores. Definámoslos en paralelo y ejecutemos todas las operaciones del filtro en un único ciclo de reloj. Ahora, no obstante, la complejidad del hardware será de orden N.
En la tabla siguiente resumimos estos conceptos aplicados al filtro FIR de N taps. El término de complejidad espacial es equivalente a la cantidad de hardware que hay que emplear.

Implementación de un filtro FIR de N *taps*		
Enfoque	**Complejidad espacial**	**Complejidad temporal**
Secuencial	unitaria	orden N
Paralelo	orden N	unitaria

Tabla 1. Resumen características enfoques secuenciales y paralelos

5.7.2. Filtro FIR

A continuación aplicaremos las ideas presentadas en el apartado anterior al diseño e implementación de un filtro FIR. Comenzaremos definiendo la entidad, que puede ser la misma para los dos tipos de implementaciones.

```
entity FIR_S is
        generic (M:integer:=8);      -- Número de etapas del filtro
        port (
                reset: in std_logic;
                clk: in std_logic;
                x: in std_logic_vector (9 downto 0);
                nova_x: in std_logic;
                y: out std_logic_vector (12 downto 0);
                nova_y: out std_logic);
end FIR_S;
```

Codi 18. Definición de la entidad de un filtro FIR

El número de etapas (*taps*) del filtro lo definimos utilizando el comando *generic*. El número de bits, no obstante, lo fijamos. En este caso 10 bits en la entrada y 13 en la salida. Remarcamos que, como implícitamente usaremos formato de coma fija fraccional, para evitar el *overflow* tendremos que añadir un número de bits de guarda que depende del número de iteraciones a través de la fórmula $\log_2(M)$, donde M es el número de iteraciones. Para un valor de $M=8$, el número de bits de guarda es 3, y por esto la salida (y) tiene tres bits más que la entrada (x). Éste es el motivo por el cual no definimos N como *generic*: la dependencia logarítmica del número de bits de la salida con M no la podríamos especificar.

Por otro lado, ya estamos familiarizados con el protocolo de paso de datos entre módulos: combinación de las señales (x, y) y las activaciones (*nova_x*, *nova_y*).

❖ Implementación secuencial

Recordemos que, de forma esquemática, la implementación de un filtro FIR, en C, se especifica como:

```
acumulado=0;
        for (i=0;i<M;++i)
                acumulado=acumulado+coeficientee(i)*entrada(i);
```

Código 19. Especificación de un filtro FIR en C

Para implementar esto mismo en VHDL, usando un enfoque secuencial, hay que reflexionar sobre los recursos hardware que hay implícitamente en esta especificación. En la figura siguiente presentamos un diagrama de bloques simplificado de estos recursos.
Será necesaria una memoria tipo FIFO para las entradas, una memoria estática tipo ROM para los coeficientes, un multiplicador, un acumulador y la máquina de estados que secuenciará las operaciones. Ya podemos intuir que, además, probablemente serán necesarios contadores y comparadores para iterar M golpes sobre los recursos aritméticos.

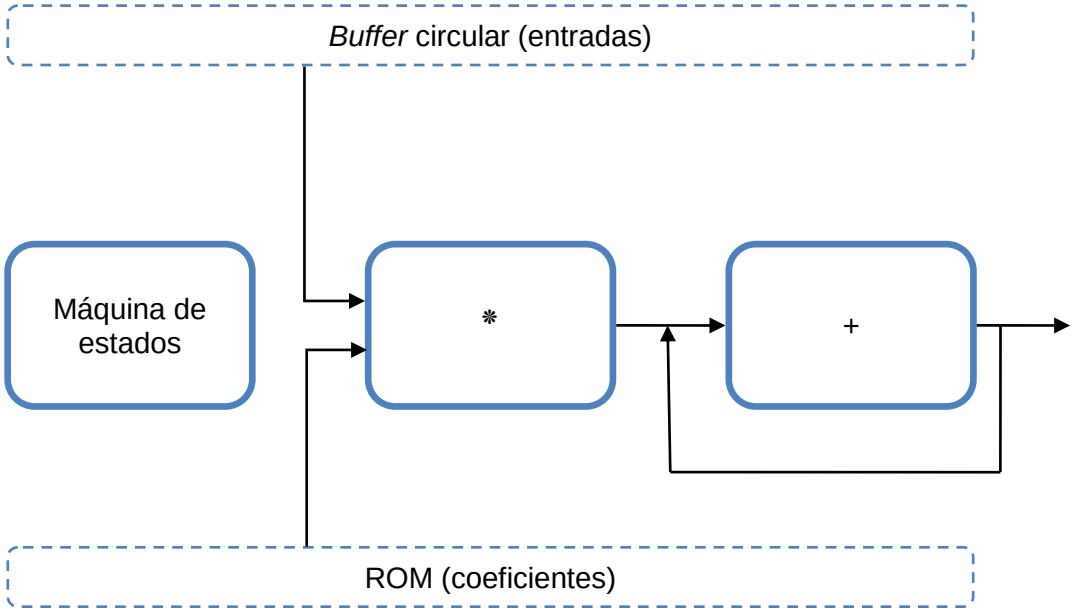


Figura 132. Diagrama de bloques de la implementación secuencial

❖ Implementación paralela

La forma de la implementación paralela del filtro FIR se puede deducir a partir del Código 19 expandiendo la iteración:

```
acumulado=acumulado+coeficiente(0)*entrada(0);
acumulado=acumulado+coeficiente(1)*entrada(1);
acumulado=acumulado+coeficiente(2)*entrada(2);
(...)
acumulado=acumulado+coeficiente(M-2)*entrada(M-2);
acumulado=acumulado+coeficiente(M-1)*entrada(M-1);
```

Codi 20. Especificación de un filtro FIR en C. Iteración expandida.

A partir de esta especificación observamos que:

- Los productos no están acoplados entre si. Es decir, se pueden implementar independientemente: para calcular *coeficiente(1)*entrada(1)* no hace falta disponer del resultado de *coeficiente(0)*entrada(0).*

- La operación suma es asociativa. Es decir, no es necesario acumular en primer lugar el resultado de *coeficiente(0)*entrada(0)*, en segundo lugar *coeficiente(1)*entrada(1)*, etc., sino que podemos agrupar los sumandos como queramos.

Teniendo en cuenta estos puntos, planteemos el diagrama de bloques siguiente:

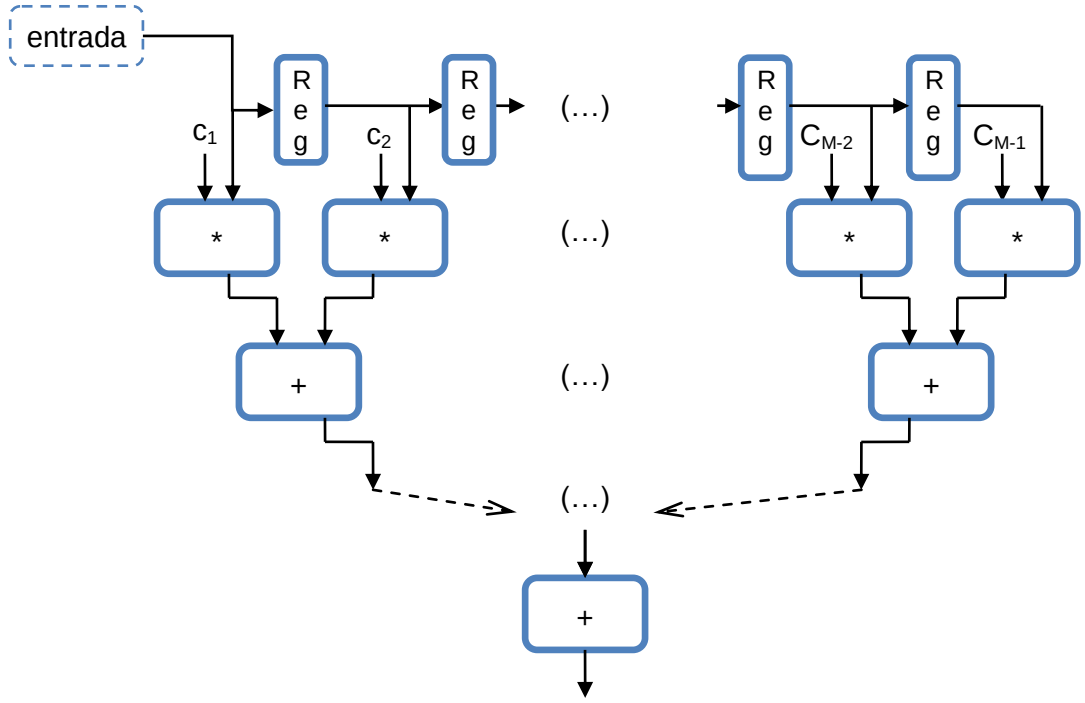


Figura 133. Diagrama de bloques de la implementación paralela

Consta de tres partes. La primera, un conjunto de registros encadenados que harán la función de memoria FIFO. La segunda, un banco de multiplicadores, que calcularán los productos de las entradas y los coeficientes del filtro. Un tercer bloque, un banco de sumadores de dos entradas en cascada.

Todos los módulos pueden funcionar en paralelo, activados por la misma señal de reloj y *enable*; por tanto, a cada ciclo de reloj se generará un valor de salida, aunque existirá una latencia[2] en el cálculo del resultado. Esta

[2] Entendemos por latencia el tiempo que transcurre entre la llegada de un dato y la generación de la salida correspondiente. Se suele expresar en ciclo de reloj y suele depender del número de fases en que se divide la implementación de un algoritmo.

latencia, si el número de coeficientes del filtro se potencia de 2, será de $1+\log_2(M)$; el 1 corresponde al banco de multiplicadores y $\log_2(M)$ al banco de sumadores de dos entradas.

Ésta no es la única implementación paralela posible: si disponemos de sumadores de más entradas, el número de etapas de suma será menor.

5.7.3. Implementación. Ejercicio

A continuación se propone como ejercicio la implementación de un filtro FIR siguiendo los enfoques estudiados en el documento.

Antes de continuar se aconseja estudiar los dos apartados al final del documento, que versa sobre la implementación de sumas y productos y sobre la reutilización de arquitecturas en forma de componentes.

Para el filtro FIR secuencial, se aconseja seguir los pasos siguientes:

- Estudia bien la definición de la entidad y el diagrama de bloques propuesto más arriba.

- Implementa una memoria de sólo lectura (*read enable*) para almacenar los coeficientes.

- Implementa una memoria de lectura/escritura (*read* y *write enable*) para almacenar los datos de entrada. Consulta el documento sobre la implementación de filtro FIR sobre DSP para recordar la funcionalidad de los *buffers* circulares.

- Implementa un multiplicador síncrono, controlado por una señal de activación (*enable*) y una de reset.

- Implementa un acumulador síncrono, controlado por una señal de activación y una de reset.
- Será necesario un contador para iterar según el número de coeficientes. Impleméntalo.

- Diseña el diagrama de estados de la máquina síncrona que controle todo el hardware.

Reflexiona sobre los beneficios de la implementación síncrona de todos los módulos. Intenta pensar en multiplicadores y sumadores no governados por señales de reloj. ¿Cómo lo harías?

Por el filtro FIR paralelo, se aconseja seguir los pasos siguientes:

- Estudia bien la definición de le entidad y el diagrama de bloques propuesto más arriba.

- Implementa los dos módulos siguientes: uno para el multiplicador y registro, y otro para el sumador de dos entradas. Ambos, síncronos y con señal de activación. Conviene definir una entidad / arquitectura por cada módulo. Así, en la construcción del filtro paralelo se podrá reaprovechar cada módulo.

- Implementa una entidad / arquitectura donde se usen los módulos anteriores.

Reflexiona sobre la conveniencia de definir módulos independientes como entidades / arquitecturas separadas, en vez de hacerlo como procesos diferentes dentro de una misma arquitectura.

Compara los códigos de las dos implementaciones. Fíjate en la complejidad y facilidad de programación de los dos enfoques.

5.7.4. Implementación VHDL secuencial de un filtro

A continuación incluimos un código VHDL que sigue el enfoque secuencial a la hora de calcular la salida del filtro.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNOD.ALL;

entity FIR_Seq is
   generic (M:integer:=8);  -- Número de etapas del filtro
   port ( reset : in std_logic;
          clk : in std_logic;
          x : in std_logic_vector(9 downto 0);
          nova_x : in std_logic;
          y : out std_logic_vector(12 downto 0);
          nova_y : out std_logic);
end FIR_Seq;

architecture Behavioral of FIR_Seq is

-- Señales relacionadas con el registro de las entradas
signal r_x:std_logic_vector (9 downto 0);

-- Señales relacionadas con la máquina de estados
```

```
type ESTATS is (E0, E1, E2, E3, E4, E5, E6, E7);
signal estat:ESTATS;
signal próximo_estado:ESTATS;

-- Señales relacionadas con la memoria de coeficientes
type V is array (0 to M-1) of std_logic_vector (9 downto 0);
signal coef_re:std_logic;
signal coef_out:std_logic_vector (9 downto 0);
-- Memoria de coeficientes, con los sus valores.
signal       coef_mem:V:=("0111111111",       "0111111111",       "0111111111",
"0111111111",  "0111111111",  "0111111111",  "0111111111",  "0111111111",
"1000000000",  "1000000000",  "1000000000",  "1000000000",  "1000000000",
"1000000000", "1000000000", "1000000000");

-- Señales relacionadas con la FIFO
signal fifo_we:std_logic;
signal fifo_re:std_logic;
signal fifo_in:std_logic_vector (9 downto 0);
signal fifo_out:std_logic_vector (9 downto 0);
-- Memoria que se usará para almacenar las señales de entrada
signal fifo_mem:V:=("0000000000", "0000000000", "0000000000", "0000000000",
"0000000000",  "0000000000",  "0000000000",  "0000000000",  "0000000000",
"0000000000",  "0000000000",  "0000000000",  "0000000000",  "0000000000",
"0000000000", "0000000000");

-- Señales relacionadas con la multiplicación
signal mult_out:std_logic_vector (19 downto 0);
signal mult_e:std_logic;
signal mult_r:std_logic;

-- Señales relacionadas con la suma
signal add_out:std_logic_vector (12 downto 0);
signal add_e:std_logic;
signal add_r:std_logic;

-- Señales relacionadas con el contador
signal counter_ce:std_logic;
signal counter_r:std_logic;
signal counter_out:integer range 0 to M;

begin

--------------------
-- Registro de la nueva entrada
reg_x: process (clk, reset, nova_x)
begin
if (reset='1') then
        r_x <= (others => '0');
losif (clk'event and clk='1') then
        if (nova_x = '1') then
                r_x <= x;
        end if;
end if;
```

```
end process;

fifo_in <= r_x;

--------------------
fifo: process (clk, reset, fifo_we, fifo_re, fifo_in)
variable i:integer range 0 to M;
begin
if (reset='1') then
        i:=0;
losif (clk'event and clk='1') then
        if (fifo_re='1' or fifo_we='1') then
                if fifo_re='1' then
                        fifo_out <= fifo_mem (i);
                losif fifo_we='1' then
                        fifo_mem (i) <= fifo_in;
                end if;
                i:=i+1;
                if (i=M) then
                        i:=0;
                end if;
        end if;
end if;
end process;

--------------------
coef: process (clk, reset, coef_re)
variable i:integer range 0 to M;
begin
if (reset='1') then
        i:=0;
losif (clk'event and clk='1') then
        if (coef_re='1') then
                coef_out <= coef_mem (i);
                i:=i+1;
                if (i=M) then
                        i:=0;
                end if;
        end if;
end if;
end process;

--------------------
mult: process (clk, mult_r, mult_e)
begin
if (mult_r='1') then
        mult_out <= (others => '0');
losif (clk'event and clk='1') then
        if (mult_e = '1') then
                mult_out <= fifo_out*coef_out;
        end if;
end if;
end process;
```

```
--------------------
add: process (clk, add_r, add_e)
begin
if (add_r='1') then
        add_out <= (others => '0');
losif (clk'event and clk='1') then
        if (add_e = '1') then
                add_out <= add_out + (mult_out(18) & mult_out(18) &
mult_out(18) & mult_out(18 downto 9));
        end if;
end if;
end process;

y <= add_out;

--------------------
counter: process (clk, counter_r, counter_ce)
begin
if (counter_r = '1') then
        counter_out <= 0;
losif (clk'event and clk='1') then
        if (counter_ce = '1') then
                counter_out <= counter_out + 1;
        end if;
end if;
end process;

--------------------
Maq_comb: process (estat, nova_x)
begin
if (reset = '1') then
        próximo_estado <= E0;
lose
        fifo_we <= '0';
        fifo_re <= '0';
        coef_re <= '0';
        counter_ce <= '0';
        counter_r <= '0';
        mult_r <= '0';
        mult_e <= '0';
        add_r <= '0';
        add_e <= '0';
        nova_y <= '0';
        case estat is
                when E0 =>
                        counter_r <='1';
                        mult_r <= '1';
                        add_r <= '1';
                        if (nova_x = '1') then
                                próximo_estado <= E1;
                        end if;
                when E1 =>
```

```
                        fifo_we <= '1';
                        próximo_estado <= E2;
                when E2 =>
                        fifo_re <= '1';
                        coef_re <= '1';
                        counter_ce <= '1';
                        próximo_estado <= E3;
                when E3 =>
                        mult_e <= '1';
                        add_e <= '1';
                        if (counter_out = M) then
                                próximo_estado <= E4;
                        lose
                                próximo_estado <= E2;
                        end if;
                when E4 =>
                        add_e <= '1';
                        próximo_estado <= E5;
                when E5 =>
                        próximo_estado <= E6;
                when E6 =>
                        nova_y <= '1';
                        próximo_estado <= E7;
                when E7 =>
                        próximo_estado <= E0;
        end case;
end if;
end process;

--------------------
Maq_sinc: process (clk)
begin
if (clk'event and clk='1') then
        estat <= próximo_estado;
end if;
end process;

end Behavioral;
```

Codi 21. Implementación secuencial de un filtro FIR

5.7.5. Implementación VHDL paralela de un filtro FIR

La implementación está organizada en tres ficheros, cada uno corresponde a tres entidades:

- Fir_add2: implementa un sumador síncrono de dos entradas

- Fir_tap: implementa un registro, que formará parte de la FIFO, y un multiplicador síncrono de un coeficiente para un valor de entrada

- Fir_par: filtro FIR paralelo de dos coeficientes, usando fir_tap y fir_add2.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNOD.ALL;
```

❖ Entidad FIR_add2:

```
entity FIR_add2 is
    Port ( clk : in std_logic;
           reset : in std_logic;
           enable : in std_logic;
           x1 : in std_logic_vector(9 downto 0);
           x2 : in std_logic_vector(9 downto 0);
           y : out std_logic_vector(10 downto 0));
end FIR_add2;

architecture Behavioral of fir_add2 is

begin

process (clk, reset, enable, x1, x2)
begin
if (reset = '1') then
  y <= (others => '0');
losif (clk'event and clk='1') then
  if (enable = '1') then
          y <= (x1(9) &  x1) + (x2(9) & x2);
  end if;
end if;
end process;

end Behavioral;
```

❖ Entidad mult:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNOD.ALL;
```

```
entity FIR_tap is
    generic (coef:std_logic_vector (9 downto 0):="0000000000");
    Port ( clk : in std_logic;
           reset : in std_logic;
           enable : in std_logic;
           x : in std_logic_vector(9 downto 0);
           reg_out : out std_logic_vector(9 downto 0);
             mult_out: out std_logic_vector (9 downto 0));
end FIR_tap;

architecture Behavioral of FIR_tap is
signal mult_temp:std_logic_vector (19 downto 0);

begin

r: process (clk, reset, enable, x)
begin
if (reset = '1') then
  reg_out <= (others => '0');
losif (clk'event and clk='1') then
  if (enable = '1') then
           reg_out <= x;
  end if;
end if;
end process;

mult: process (clk, reset, enable, x)
begin
if (reset = '1') then
  mult_temp <= (others => '0');
losif (clk'event and clk='1') then
  if (enable = '1') then
           mult_temp <= x*coef;
  end if;
end if;
end process;

mult_out <= mult_temp (18 downto 9);

end Behavioral;
```

❖ Entidad FIR_par:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNOD.ALL;

entity FIR_par is
    Port ( clk : in std_logic;
```

```
        reset : in std_logic;
        x : in std_logic_vector(9 downto 0);
        nova_x : in std_logic;
        y : out std_logic_vector(10 downto 0);
        nova_y : out std_logic);
end FIR_par;

architecture Behavioral of FIR_par is

component FIR_tap
   generic (coef:std_logic_vector (9 downto 0));
   Port ( clk : in std_logic;
        reset : in std_logic;
        enable : in std_logic;
        x : in std_logic_vector(9 downto 0);
        reg_out : out std_logic_vector(9 downto 0);
          mult_out: out std_logic_vector (9 downto 0));
end component;

component fir_add2
  Port ( clk : in std_logic;
        reset : in std_logic;
        enable : in std_logic;
        x1 : in std_logic_vector(9 downto 0);
        x2 : in std_logic_vector(9 downto 0);
        y : out std_logic_vector(10 downto 0));
end component;

signal reg_out1: std_logic_vector (9 downto 0);
signal mult_out1: std_logic_vector (9 downto 0);
signal mult_out2: std_logic_vector (9 downto 0);
signal temp_reg_out2: std_logic_vector (9 downto 0);
signal temp_y: std_logic_vector (10 downto 0);

begin

tap1: FIR_tap
  generic map ("0111111111")
  port map (
         clk => clk,
       reset => reset,
       enable => nova_x,
       x => x,
       reg_out => reg_out1,
       mult_out => mult_out1);

tap2: FIR_tap
  generic map ("0111111111")
  port map (
         clk => clk,
       reset => reset,
       enable => nova_x,
       x => reg_out1,
```

```
        reg_out => temp_reg_out2,
        mult_out => mult_out2);

add2_1: FIR_add2
  port map (
          clk => clk,
        reset => reset,
        enable => nova_x,
        x1 => mult_out1,
        x2 => mult_out2,
        y => temp_y);

y <= temp_y (10 downto 0);
nova_y <= nova_x;
end Behavioral;
```

5.8. Sobre la implementación de funciones aritméticas

Las funciones aritméticas suma y producto se pueden implementar directamente a partir de comandos VHDL. A continuación describiremos el comportamiento de estos comandos VHDL cuando se utiliza la herramienta de síntesis XST de Xilinx (consultarlo por otras herramientas).

Supondremos que tanto los valores de entrada como los de salida se interpretan en complemento a dos (CA2) con signo.

5.8.1. La suma

La suma de dos sumandos de *N* bits (*a* y *b*) genera un resultado de *N* bits. En caso de *overflow*, el resultado de la suma será:

$-2^N + a + b$, si $a + b > 2^{N-1} - 1$
$2^N + a + b$ si $a + b < -2^{N-1}$

Para evitar el *overflow* tendremos que definir una señal resultante con un bit adicional, y aplicar extensión de signo a los sumandos (ver código siguiente).

```
signal a : std_logic_vector(N-1 downto 0);
signal b : std_logic_vector(N-1 downto 0);
signal r : std_logic_vector (N downto 0);

r <= (a(N-1) & a) + (b(N-1) & b);
```

Codi 22. Ejemplo de suma en VHDL (CA2 con extensión de signo). Sin escalado

Si no queremos expresar el resultado con un bit más, tendremos que escalar. Tenemos dos opciones. La primera consiste en escalar los sumandos antes de operar (Codi 23); en este caso, el bit de menos peso de los sumandos no tendrá ningún efecto sobre el resultado de la suma.

```
signal a : std_logic_vector(N-1 downto 0);
signal b : std_logic_vector(N-1 downto 0);
signal r : std_logic_vector (N-1 downto 0);

r <= (a(N-1) & a(N-1 downto 1)) + (b(N-1) & b(N-1 downto 1));
```

Codi 23. Ejemplo de suma en VHDL (CA2 con extensión de signo). Escalando los sumandos

La segunda opción consiste en escalar el resultado de la suma (Código 24). En este caso el resultado de la suma si se verá afectado por los bits de menos peso de los sumandos cuando los dos valgan '1' (la generación de carry afectará el resultado de la suma), pero no cuando sólo uno de los dos valga cero.
En ambos casos hay que ser consciente de que hemos dividido el resultado entre 2. Hay un exponente implícito con valor 1 que hemos de tener en cuenta.

```
signal a : std_logic_vector(N-1 downto 0);
signal b : std_logic_vector(N-1 downto 0);
signal r_temp : std_logic_vector (N downto 0);
signal r : std_logic_vector (N-1 downto 0);

r_temp <= (a(N-1) & a) + (b(N-1) & b);
r <= r_temp(N downto 1);
```

Código 24. Ejemplo de suma en VHDL (CA2 con extensión de signo). Escalando el resultado

Es obvio que las mejores prestaciones las obtendremos conservando todos los bits del resultado (Codi 22), y que puestos a escoger, el Código 24 es mejor que el Codi 23.

5.8.2. El producto

El producto de dos valores de *N* bits (*a* y *b*) genera un resultado de *2·N* bits. Ahora bien, de estos *2·N* bits, sólo los *2·N-1* bits de menos peos tienen importancia, ya que el bit de más peso es un bit redundante del signo.

```
signal a : std_logic_vector(N-1 downto 0);
signal b : std_logic_vector(N-1 downto 0);
signal r_temp:std_logic_vector (2*N-1 downto 0)
signal r : std_logic_vector (2*N-2 downto 0);

r_temp <= a*b;
r <= r_temp (2*N-2 downto 0);
```

Código 25. Ejemplo de producto en VHDL

Las mismas ideas de escalado, vistas en la operación suma, se pueden aplicar en este caso.

5.9. Sobre la definición de componentes

En este apartado se explica cómo reaprovechar diseños (entidades / arquitecturas) en lenguaje VHDL. Considerad la implementación de una arquitectura cualquiera, correspondiente a la entidad *X* siguiente:

```
entity X  is
   generic (c:std_logic:='0');
   port ( clk : in std_logic;
          reset : in std_logic;
          x : in std_logic_vector(7 downto 0);
          y : out std_logic_vector(7 downto 0);
    );
end X;
```

Código 26. Una entidad cualquiera

Suponed que queréis utilizar esta entidad / arquitectura *X* en otro diseño *Y*. tendréis que proceder de la forma siguiente.

1. Dentro de la arquitectura de *Y* (fuera del *begin - end*) habrá que declarar la entidad *X* como componente.

2. Dentro de la arquitectura de *Y* (dentro del *begin - end*) habrá que definir cada uso de la entidad *X*: mapeo de las señales de entrada / salida (Observad que estas señales habrá que definirlas como tales previamente)

```
architecture arch_Y of Y is
component X
        generic (c:std_logic);
        port (clk : in std_logic;
                reset : in std_logic;
                x : in std_logic_vector(7 downto 0);
                    y : out std_logic_vector(7 downto 0));
        end component;
(...)
signal clk: std_logic; -- Si no está definido en la entidad
signal reset: std_logic; -- Si no está definido en la entidad
signal x1: std_logic_vector (7 downto 0);
signal y1: std_logic_vector (7 downto 0);
begin
        X1: X
                generic map ('0')
                port map (
                        clk => clk,
                        reset => reset,
                        x => x1,
                        y => y1);
        X2: X
```

generic map ...

5.10. Implementación de filtros CIC

5.10.1. Introducción

En este documento trataremos la implementación de filtros asociados a delmadores e interpoladores, de tipo CIC (*cascaded integrator-comb*). Estos filtros ya los estudiamos como parte integrante de los mezcladores digitales (DDC y DUC); allí hicimos énfasis en las prestaciones, mientras que ahora nos centraremos en aspectos de implementación usando VHDL.

Recordemos el lugar que ocupen los filtros según si el sistema es delmador o interpolador:

Figura 134. Papel de los filtros en los procesos de delmación e interpolación

En el caso de delmación, el filtro actúa como a antialiasing, antes de la reducción de la frecuencia de muestreo. Asociado a un interpolador, después de incrementar la frecuencia de muestreo el filtro atenúa las imágenes.
A partir de ahora nos centraremos en la delmación, sabiendo que el sistema interpolador es dual. La función de transferencia de un filtro CIC, expresada en el dominio de la transformada Z es:

$$H(z)=\left(\frac{1-z^{-M}}{1-z^{-1}}\right)^{N}$$

El parámetro *M* es el factor de delmación de la etapa asociada. El orden del filtro (*N*) determina la atenuación alrededor de los ceros. En la figura siguiente se reproduce la respuesta en frecuencia de un filtro CIC con parámetros *N=3*, *M=3*.

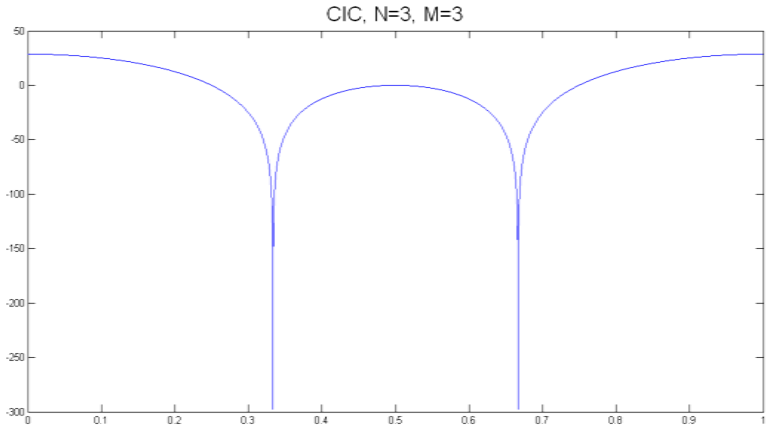


Figura 135. Respuesta en frecuencia de un filtro CIC3, asociado a una delmación de 3

A partir de la fórmula de la suma de una serie geométrica podemos obtener otra representación de estos filtros:

$$H(z)=\left(\frac{1-z^{-M}}{1-z^{-1}}\right)^{N}=\left(1+z^{-1}+\ldots+z^{-M+1}\right)^{N}$$

Observemos que:

- El filtro tiene una naturaleza FIR (todo ceros)

- El filtro tiene *M-1* ceros equiespaciados sobre el circulo unitario, a las frecuencias:

$$Zero_k = 1_{k\cdot\frac{2\cdot\pi}{M}} = \cos\left(k\cdot\frac{2\cdot\pi}{M}\right)+j\cdot\sin\left(k\cdot\frac{2\cdot\pi}{M}\right),\ k\in[1,M-1]$$

- El filtro tiene una ganancia en continua de M^N, que expresado en bits corresponde a $\lceil N\cdot\log_2(M)\rceil$ bits (el operador $\lceil\ \rceil$ corresponde al redondeo al entero superior). Este parámetro nos forzará, a la hora de implementarlo, a tomar decisiones en lo que respecta al escalado.

5.10.2. Implementaciones

A continuación describiremos tres implementaciones diferentes, siempre teniendo en mente que para la aplicación que suelen tener estos filtros (DDC), serán necesarias arquitecturas paralelas para maximizar la frecuencia de ejecución.

❖ Basada en integradores y derivadores, con delmador al final.

Descomponemos la expresión sumada del filtro CIC de la forma siguiente:

$$H(z)=\left(\frac{1-z^{-M}}{1-z^{-1}}\right)^{N}=\left(\frac{1}{1-z^{-1}}\right)\cdot\left(\frac{1}{1-z^{-1}}\right)\cdot\ldots\left(\frac{1}{1-z^{-1}}\right)\cdot\left(1-z^{-M}\right)\cdot\left(1-z^{-M}\right)\cdot\ldots\left(1-z^{-M}\right)$$

La ejecución del filtro consiste, pues, en el producto en frecuencia de *N* módulos integradores ($\left(\frac{1}{1-z^{-1}}\right)$) y *N* módulos derivadores ($\left(1-z^{-M}\right)$). Una vez aplicado el filtro, se aplicará la delmación en un factor *M*.
La implementación de un integrador es muy sencilla:

$$H_{INT}(z)=\frac{1}{1-z^{-1}}\rightarrow X(z)=Y(z)\cdot\left(1-z^{-1}\right)\rightarrow y(n)=y(n-1)+x(n)$$

El diagrama de bloques será:

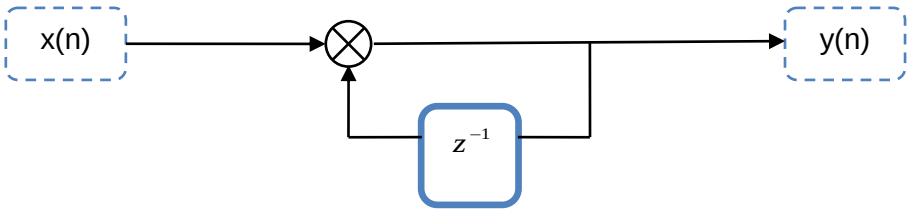


Figura 136. Diagrama de bloques de un integrador

Para implementar el integrador será suficiente con un registro (retrasador) y una sumador / restador. Veámoslo en el código siguiente:

```
signal x:std_logic_vector (Nin-1 downto 0);
signal t_y1:std_logic_vector (Nin+2 downto 0);

F1: process (clk, reset, x)
begin
if reset = '1' then
        t_y1 <= (others => '0');
losif clk'event and clk='1' then
        if (nova_x = '1') then
                t_y1 <= t_y1 + ( x(Nin-1) & x(Nin-1) & x(Nin-1) & x);
        end if;
end if;
end process;
```

Código 27. Un integrador unitario, sin escalado, en VHDL

Fijaos en que:

- El módulo es síncrono, con *reset* asíncrono, y con *enable* de operación (*nova_x*)

- La entrada (*x*) tiene N_{in} bits. La salida (*t_y1*) tiene 3 bits más. Consultad el apartado siguiente, referente al escalado en los filtros CIC.

- El acumulador, de $N_{in}+3$ bits, opera sobre un dato N_{in} bits, previa extensión de signo.

- La memoria *y(n-1)* es implícita al proceso.

Igualmente podemos repetir el proceso por el derivador, llegando a la respuesta impulsional y diagrama de bloques siguientes:

$$H_{DER}(z) = 1 - z^{-M} \rightarrow y(n) = x(n) - x(n-M)$$

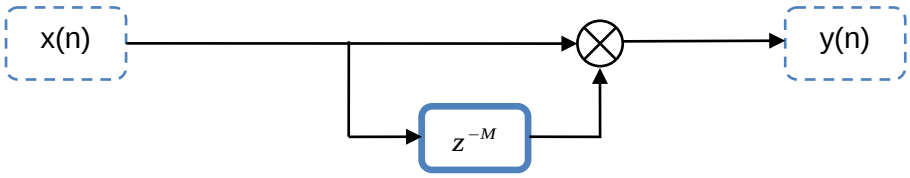


Figura 137. Diagrama de bloques de un derivador

Los recursos necesarios serán un sumador / restador y un retardante de *M* muestras (puede implementarse con un banco de *M* registros en serie, o bien una memoria FIFO, etc.)

La implementación paralela del filtro requiere de *2·N* sumadores / restadores. El coste computacional por segundo (número de operaciones matemáticas que se ejecutan cada segundo) será de $f_s \cdot 2 \cdot N$. Remarquemos que el número de recursos usado es indicativo del espacio que ocupa el diseño, mientras que el coste computacional por segundo está relacionado con el consumo de potencia.

❖ Basada en integradores y derivadores, con delmador en el medio.

Para presentar la idea en que se basa la alternativa de implementación de este apartado, analizaremos la evolución temporal de la salida del derivador y del delmador, para *N=1* y *M=2*. Veámoslo en la tabla siguiente:

Tiempo	Salida integrador	Salida derivador	Salida delmador
T	x_k	x_k-x_{k-2}	x_k-x_{k-2}
2·T	x_{k+1}	x_{k+1}-x_{k-1}	-
3·T	x_{k+2}	x_{k+2}-x_k	x_{k+2}-x_k
4·T	x_{k+3}	x_{k+3}-x_{k+1}	-
5·T	x_{k+4}	x_{k+4}-x_{k+2}	x_{k+4}-x_{k+2}
6·T	x_{k+5}	x_{k+5}-x_{k+3}	-
7·T	x_{k+6}	x_{k+6}-x_{k+4}	x_{k+6}-x_{k+4}

Fijémonos en la salida del delmador; los valores impares de la entrada del derivador (x_{k+1}, x_{k+3}, etc) no afectan nunca a la salida; de hecho, el derivador no tiene porqué calcularlos.

Es por esto que se propone la alternativa siguiente de implementación:

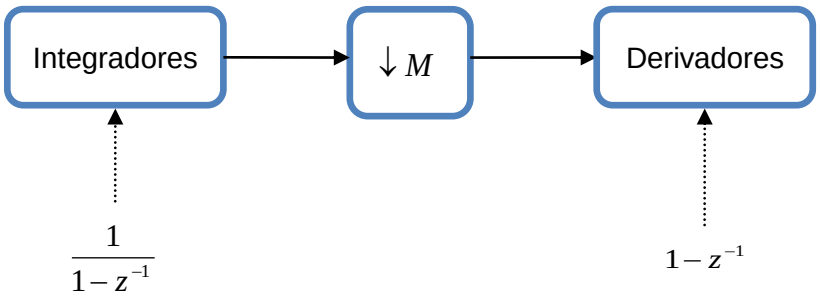


Figura 138. Implementación con el delmador entre integradores y derivadores

Es importante remarcar que el retraso asociado a los derivadores ahora es *1* en vez de *M* (el delmador ya descarta *M-1* muestras de cada *M*, de forma que muestras separadas por *M* instante antes que el delmador, son consecutivas después del delmador). En la tabla siguiente reproducimos la evolución temporal de las señales con la nueva arquitectura (para el caso particular *N=1* y *M=2*).

Se observa que los resultados son idénticos.

Tiempo	Salida integrador	Salida delmador	Salida derivador
T	x_k	x_k	x_k-x_{k-2}
2·T	x_{k+1}	-	-
3·T	x_{k+2}	x_{k+2}	x_{k+2}-x_k
4·T	x_{k+3}	-	-
5·T	x_{k+4}	x_{k+4}	x_{k+4}-x_{k+2}
6·T	x_{k+5}	-	-
7·T	x_{k+6}	x_{k+6}	x_{k+6}-x_{k+4}

Los recursos aritméticos necesarios para implementar paralelamente el filtro CIC con delmador en el medio es de 2·N sumadores / restadores (igual que la anterior implementación), pero el coste computacional por segundo ahora es menor:

$$f_s \cdot N + \frac{f_s}{M} \cdot N = f_s \cdot N \cdot \left(1 + \frac{1}{M}\right)$$

ya que la rama de derivadores funciona a una frecuencia menor. Por tanto, esta alternativa tiene un consumo de potencia menor a la anterior.

Vamos a continuación el código de un derivador con retraso unitario:

```
signal x:std_logic_vector (Nin-1 downto 0);
signal M1:std_logic_vector (Nin-1 downto 0);
signal t_y1:std_logic_vector (Nin-1 downto 0);

F1: process (clk, reset, x)
begin
if reset = '1' then
        t_y1 <= (others => '0');
losif clk'event and clk='1' then
        if (nova_x = '1') then
                t_y1 <= (x(Nin-1) & x(Nin-1 downto 1))
                        – (M1 (Nin-1) & M1(Nin-1 downto 1));
                M1 <= x;
        end if;
end if;
end process;
```

Codi 28. Un derivador unitario, con escalado incondicional, en VHDL

En este caso remarcar que el módulo escala los operandos origen a la entrada, de forma incondicional. El resultado del derivador (*t_y1*) tiene el mismo número de bits que la entrada.

❖ Basada en filtro FIR

Si tomamos la expresión alternativa del filtro CIC, obtenemos:

$$H(z)=\left(1+z^{-1}+\ldots+z^{-M+1}\right)^{N}=\left(1+z^{-1}+\ldots+z^{-M+1}\right)\cdot\ldots\left(1+z^{-1}+\ldots+z^{-M+1}\right)$$

Podemos implementar, pues, el filtro CIC como la concatenación de *N* filtros FIR de M coeficientes cada uno. Todos los coeficientes son 1, por tanto no son necesarios multiplicadores. El diagrama de bloques se explicita a continuación:

Figura 139. Implementación alternativa, como concatenación de filtro FIR

Después del último filtro podríamos el delmador.

El número de recursos necesarios para implementar el filtro con esta arquitectura es de $(M-1)\cdot N$ sumadores / restadores. Por tanto, para cualquier factor de delmación superior a 3 el número de recursos necesarios

es mayor a las dos alternativas anteriores. El coste computacional por segundo será de $f_s \cdot (M-1) \cdot N$; superior a la primera alternativa si *M>3* y superior a la segunda alternativa si *M>2.*

A continuación presentamos un proceso VHDL que implementa un filtro FIR con *M=3*. Una implementación completa requeriría la concatenación de *N* procesos como el siguiente:

```
signal x:std_logic_vector (Nin-1 downto 0);
signal M1_1:std_logic_vector (Nin-1 downto 0);
signal M1_2:std_logic_vector (Nin-1 downto 0);
signal t_y1:std_logic_vector (Nin+1 downto 0);

F1: process (clk, reset, x)
begin
if reset = '1' then
        M1_1 <= (others => '0');
        M1_2 <= (others => '0');
        t_y1 <= (others => '0');
losif clk'event and clk='1' then
        if (nova_x = '1') then
                t_y1<=(x(Nin1)&x(Nin1)&x)
                        +(M1_1(Nin1)&M1_1(Nin1)&M1_1)
                        +(M1_2(Nin-1)&M1_2(Nin-1)&M1_2);
                M1_1 <= x;
                M1_2 <= M1_1;
        end if;
end if;
end process;
```

Codi 29. Ejemplo de un filtro FIR paralelo, de tres coeficientes unitarios

Fijaos en que:

- El módulo es síncrono, con reset asíncrono, y con *enable* de operación (*nova_x*)

- La entrada (*x*) tiene N_{in} bits. La salida (*t_y1*) tiene 2 bits más ($\lceil \log_2(3) \rceil$); por tanto esta implementación no escala: opera con la máxima resolución y conserva todos los bits a la salida.

- Los sumadores, de $N_{in}+2$ bits, operan sobre datos de N_{in} bits, previa extensión de signo.

- La memoria del filtro se implementa mediante las señales *M1_1* y *M1_2*, utilizando los registros implícitos a los procesos.

5.10.3. El escalado en los filtros CIC

A continuación se explican los requerimientos en lo que respecta al número de bits de las implementaciones de filtros CIC con estructuras formadas por la concatenación de integradores y derivadores.

Discutimos, pues, la implementación de un filtro CIC a partir de la Ecuación:

$$H(z)=\left(\frac{1-z^{-M}}{1-z^{-1}}\right)^{N}=\left(\frac{1}{1-z^{-1}}\right)\cdot\left(\frac{1}{1-z^{-1}}\right)\cdot\ldots\cdot\left(\frac{1}{1-z^{-1}}\right)\cdot\left(1-z^{-M}\right)\cdot\left(1-z^{-M}\right)\cdot\ldots\cdot\left(1-z^{-M}\right)$$

Fijáos en que el filtro H(z)se puede reescribir como un FIR (ved el apartado anterior). Por tanto, es necesariamente estable (su salida no crece indefinidamente, sea cual sea su entrada).

La implementación a partir de la concatenación de integradores y derivadores, no obstante, presenta un problema. Un módulo integrador es inestable: $y(n)=y(n-1)+x(n)$ crece indefinidamente si la señal de entrada es siempre positiva. Surge entonces la pregunta: ¿con cuántos bits hay que implementar el integrador para asegurar que, globalmente, no se produzca overflow en el filtro?

Se puede demostrar (mirad la referencia al final del apartado) que si se implementa la estructura CIC sobre una aritmética CA2 hay suficiente con que todas les etapas (derivadores e integradores) del filtro se implementen con $ceil(n+N\cdot log_2 M)$ bits, donde *n* es el número de bits en la entrada del filtro, *N* es el orden del filtro y *M* es la delmación asociada al filtro. El operador ceil (·) genera el número natural inmediatamente superior (o igual)

Ejemplo:

Sea un filtro CIC de orden 4 asociado a una etapa delmadora de orden 5. El filtro recibe los datos de un conversor A/D de 10 bits. Determina el número de etapas del filtro y el número de bits con que hay que implementar cada etapa per asegurar que la salida del filtro sea correcta. Trabajemos con CA2.

Solución:

Hagámoslo suponiendo una implementación basada en integradores y derivadores, con el delmador en el medio.

Entonces, el número de etapas del filtro será: 4 integradores, seguidos de 1 delmador de orden 5 y seguido de 4 derivadores.

Los integradores y derivadores hay que implementarlos usando:

$$ceil(n + N \cdot log_2 M) = ceil(10 + 4 \cdot log_2 5)) = 20 \text{ bits}$$

Si lo hacemos así, la salida del filtro (después del último derivador) será correcta, aunque muy probablemente se habrá producido overflow en etapas intermedias.

El razonamiento aquí expuesto no es generalizable a otros algoritmos. Da una demostración ad hoc para esta estructura concreta.

Referencia:

Eugene B. Hogenauer, An Economical Class of Digital Filters for Decimation and Interpolation, IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-29, No. 2, April 1981