Sistemas Informáticos
Curso 2007-2008

Implementación de
Digital Audio BroadCasting en un DSP

Autores:

Miguel Sánchez Pecharromán.
Jesús Mascías Santa María.

Director:

Profesor Luis Piñuel Moreno.
Departamento de Arquitectura de Computadores y Automática.

Facultad de Informática
Universidad Complutense de Madrid

Índice de contenidos

Índice de tablas

Índice de ilustraciones

Palabras Clave

DAB

COOLFLUX

MICROPROCESADOR

DSP

BROADCASTING

RADIO

Cesión de derechos

Por la presente, los autores de este proyecto, autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código implementado y/o la documentación.

Firmado:

Miguel Sánchez Pecharromán Jesús Mascías Santamaría

Resumen

Dentro del contexto del apagón analógico, previsto en Europa para los primeros años de la segunda década del siglo XXI, los sistemas de transmisión de información digital por radiofrecuencia están en claro proceso de implantación y consolidación. Uno de los sistemas de radio digital más importantes es el Digital Audio Broadcasting (DAB). Las redes de emisión de DAB están operativas en la mayor parte de territorio europeo, y existe una oferta lo suficientemente amplia de sistemas de recepción de DAB para estaciones de radio domésticas y automovilísticas. Sin embargo, el mercado de receptores sufre una ausencia de dispositivos portátiles. Este gap tecnológico ha abierto una carrera entre los fabricantes de silicio para lograr un sistema de recepción de DAB que permita la portabilidad del equipo huésped, y con bajo requerimiento energético.

Los principales sistemas de radio digital han sido diseñados siguiendo un enfoque software, lo que permite tener hardware muy similar para diferentes estándares y ofrece la posibilidad de hacer frente a actualizaciones o adaptaciones sucesivas de los estándares de comunicación mediante demodulación/decodificación por software.

En este documento se presentará el desarrollo del software de un sistema de recepción de DAB para CoolFlux, un DSP de consumo ultraligero específico para dispositivos móviles de carga limitada. Para realizar dicho desarrollo se han utilizado diferentes técnicas de optimización combinadas siguiendo un enfoque holístico, adaptadas a las características del procesador. Algunas de estas técnicas serán descritas en este documento, así como las características fundamentales del sistema de comunicación y el procesador utilizados para su implementación.

El primer estudio de viabilidad ha ofrecido unos resultados satisfactorios en lo referente a los requerimientos de procesador que serían necesarios para la implementación, siendo posible realizar la recepción de DAB utilizando un único DSP.

Abstract

Within the context of the analogue switch-off expected to be completed for the first years of the second decade of XXI century in Europe, the RF digital data transmission systems are in process of introduction and consolidation. One of the most important digital radio systems is Digital Audio Broadcasting (DAB). The DAB broadcasting networks are already operational in most European territory, and there is a sufficiently broad range of systems for receiving DAB in domestic and automobile radio stations. However, the receivers market suffers a lack of portable devices. This technological gap has started a challenge between the silicon companies which goal is to achieve a DAB reception system that allows the portability of the host device with ultra-low power consumption.

The main digital radio systems have been designed following a software approach, which allows having very similar hardware for different standards and managing standard updates/adaptations efficiently through software demodulation/decoding.

This document will present the software developement of a DAB receiver for CoolFlux, an ultra-low power DSP core which targets mobile devices of limited charge. To make this development, different processor-adapted optimization techniques have been merged following an holistic approach. Some of these techniques will be outlined in this document, as well as the main features of the communication system and processor used for its implementation.

The first feasibility study has offered a satisfactory outcome regarding the processor requirements that would be necessary for the implementation, showing that is possible to develop DAB reception using one DSP.

Introducción

Este proyecto de sistemas informáticos ha consistido en el desarrollo de una aplicación de recepción de señal de radio digital, DAB (Digital Audio Broadcasting), para su inclusión a modo de funcionalidad software en la oferta de aplicaciones del procesador CoolFlux DSP.

CoolFlux DSP es una arquitectura especializada en el proceso de señales de audio. Este procesador forma parte de la oferta tecnológica de la compañía NXP Semiconductors, compañía holandesa que dispone de unas instalaciones de desarrollo de tecnología en el DSP valley de Leuven, lugar en el cual se ha desarrollado el proyecto

El interés de la compañía en la realización de este proyecto de sistemas informáticos responde a dos cuestiones principales:

En primer lugar, la metodología de desarrollo del procesador CoolFlux DSP por parte de la empresa. Esta metodología está basada en sucesivas iteraciones que se sirven del desarrollo de aplicaciones software de sistemas de comunicación reales para evaluar las prestaciones de dicho procesador y decidir así los objetivos y características a implementar y/o modificar en las siguientes generaciones del mismo. En este sentido, el sistema DAB es un sistema de telecomunicación real de audio y datos que podría servir perfectamente para testear la robustez y versatilidad de CoolFlux en el tratamiento de la señal. Otras aproximaciones similares siguiendo la misma política de desarrollo han sido la implementación de decodificadores de mp3 y la realización de una librería matemática que pudiera servir como soporte a futuros programadores de sistemas complejos.

En segundo lugar, el motivo principal del desarrollo: la aparición del sistema DAB (Digital Audio Broadcasting) en 1987, con el proyecto impulsado por la Unión Europea de Radiodifusión (UER) denominado Eureka 147, la cual derivó en un creciente interés por parte de compañías del rubro tecnológico en las posibilidades de esta tecnología y por tanto en la necesidad de abordar el diseño de nuevos equipos que permitieran la explotación de los servicios que este sistema de comunicación pudiera proporcionar. En 1995 se dio el siguiente paso para la explotación del sistema, cuando el European Telecommunication Standard Institute (ETSI) adoptó el sistema DAB como estándar único europeo [2]. A nivel mundial, las recomendaciones BS 11 14 y BS 11 30 de la International Telecommunications Union (ITU) sugieren la utilización de DAB como sistema para la difusión terrestre y por satélite, respectivamente.

Actualmente, DAB está entrando en la fase de implantación en varios países europeos, algunos de los cuales ya han programado la desconexión analógica para mediados de la segúnda década del siglo XXI. Existen multitud de proyectos en marcha, tanto en Europa como en otras partes del mundo, como puede apreciarse en la Ilustración 1.

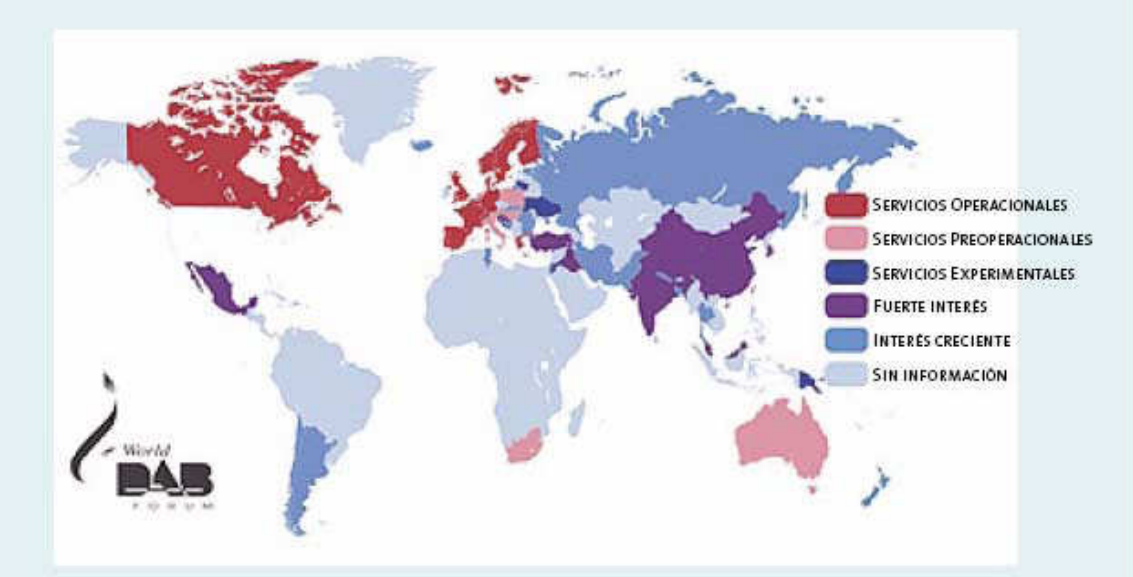


Ilustración 1: Situación del desarrollo del DAB en diferentes países del mundo. [15]

Debido al alto grado de desarrollo de infraestructuras de emisión de este tipo de señal de radio digital y a su madurez tecnológica, se han abierto nuevos horizontes en el mercado de los dispositivos digitales. Países como Francia (TDF), Reino Unido (BBC), Alemania (Deutsche Telekom) y la propia España tienen compañías dedicadas al establecimiento de una red de transmisión fiable, por lo que se prevee que en la mayoría de los países Europeos el sistema de comunicación DAB se imponga de aquí a unos años. No en vano, en la actualidad existen más de 400 servicios DAB en emisión en todo el mundo, los cuales alcanzan a una población de aproximadamente 230 millones de personas. [15] [16]

La definitiva aceptación e incipiente operatividad del sistema a nivel nacional e internacional motivará, más pronto que tarde, una estabilización definitiva de la demanda en el sector por parte de los usuarios, que desearán adquirir tanto equipos móviles de recepción de DAB (automóvil, radios de bolsillo) como equipos destinados a la recepción fija.

El mercado de los receptores ha sido hasta ahora ocupado por receptores estáticos o receptores destinados a su integración como accesorios de automóviles. Los receptores portátiles son los que presentan un mayor problema de diseño debido a la limitación que han de presentar tanto de tamaño como de consumo eléctrico (lo que repercute de forma drástica a su vez, también en el tamaño, debido a las baterías eléctricas); es en esto en lo que su implantación encuentra un problema a día de hoy no atendido; puesto que tienen un muy alto consumo en comparación con otras tecnologías portátiles, lo que hay que añadir ya a la de por sí alta diferencia en el precio del receptor. La ausencia de receptores a bajo coste se percibe como un factor crítico para la digitalización de la radio, por lo que la empresa que consiga integrar DAB en productos que ya gozan de cierto grado de competitividad en el mercado de los receptores móviles destinados al usuario, obtendrá una posición ventajosa en el momento en el que DAB se convierta en hegemónico.

Por tanto, el interés de la compañía en la implementación de DAB es doble. Por un lado encontramos la necesidad de hacer una primera aproximación al sistema para comprobar la viabilidad de su realización y un cálculo preliminar de los requerimientos tecnológicos que presenta la recepción. Interesa sobre todo su coste aproximado en términos de consumo de ciclos de procesador con el fin de su implementación y desarrollo como aplicación comercial. En segunda instancia, como test de de un sistema real útil para comprobar, depurar e incrementar el rendimiento y las prestaciones del DSP CoolFlux, descubrir errores y estudiar la posibilidad de incluir nuevas estructuras hardware para optimizar tanto el consumo de potencia del procesador como el rendimiento en ejecución de operaciones comunes a varios sistemas de comunicación que requieren tratamiento de audio, como lo es por ejemplo FFT (Fast Fourier Transform).

Este proyecto ha sido realizado íntegramente en las instalaciones de NXP en Leuven por parte de dos alumnos a los que se les asignó el mismo grupo de desarrollo. Para la realización del proyecto ha sido necesaria una alta dosis de trabajo en equipo de modo que, como podremos ver más adelante, las tareas a realizar una vez planificado el diseño eran semi-independientes, por lo que se repartieron entre los dos componentes del proyecto. A pesar de la independencia de las tareas comentadas, durante todo el proceso fue necesaria una comunicación continua entre los dos miembros del equipo de trabajo y el supervisor, el cual tuvo que tener conocimiento del proceso, avances y problemas que se iban encontrando. Este tercer y último participante del grupo de desarrollo en el proceso tenía la función de supervisar el camino que iba llevando el proyecto, pero además, tuvo dos roles muy importantes:

- el primero, el de resolver dudas y cuestiones. Como se ha explicado, era un proyecto innovador y con conceptos nuevos para los alumnos, por lo que la ayuda del tutor del proyecto en la empresa fue necesaria en varias ocasiones.

- además, ejercía de intercomunicador, pues sirvió de eslabón en la comunicación con otros grupos de trabajo cuando demandamos ciertos cambios o documentos.

Durante todo el tiempo de desarrollo del proyecto, siempre se tuvo en mente la necesidad de generar una buena documentación sobre el producto que se estaba creando. No era una documentación totalmente orientada al consumidor final, pues como ya se ha dicho, se trataba de un estudio de viabilidad para un futuro desarrollo. Por lo tanto esta documentación fue orientada a que un posible futuro programador entendiese el por qué de una función o atributo y supiese continuar o adaptar nuestra implementación.

El proyecto realizado consta de los siguientes cuatro capítulos.

El **primer capítulo** contiene una presentación de las características generales del sistema de radio digital DAB, así como una descripción de la trama de transmisión de datos y de la forma en que dichos datos son transmitidos en este estándar de comunicación.

El **segundo capítulo** consta de la presentación y descripción del procesador CoolFlux DSP. Se presentan sus características principales, su importancia en el panorama del Silicio y sus objetivos de diseño. A continuación se proporciona información detallada sobre su arquitectura y características técnicas, así como también una breve descripción de las herramientas de desarrollo existentes para dicho procesador.

En **el tercer capítulo** viene presentada la implementación realizada, partiendo del alcance del proyecto y las decisiones generales de diseño, para finalizar con un análisis exhaustivo de cada módulo desarrollado en sus vertientes de diseño, implementación y medidas de rendimiento.

En el **cuarto capítulo** se presenta el análisis de los resultados obtenidos en la implementación, centrándose en las medidas de rendimiento más importantes para el procesador CoolFlux, el número de ciclos de procesador y la memoria necesaria. También son presentadas las conclusiones derivadas de la realización del proyecto, y se comentan las dificultades encontradas.

Capítulo 1. Digital Audio Broadcasting

1.1 Introducción

La función elegida para ser implementada en el DSP es un sistema en desarrollo de radiodifusión de audio digital. Está orientado a la recepción móvil aunque también existen receptores domésticos para un uso estático. La difusión puede ser tanto terrestre como a través de satélites y además como se irá explicando, es un sistema capaz de incluir tanto audio como datos en su transmisión.

El sistema lleva funcionando en pruebas desde 1995, siendo los países pioneros Inglaterra y Suecia, más adelante se unió Alemania, y en 1996 comenzó España mediante Radio Nacional de España [16]. El proyecto se conoce mundialmente como EUREKA-147 donde se pretende que las diferentes partes involucradas en el proyecto (proveedores, administraciones públicas, empresas privadas, etc.) establezcan bases para conseguir una radiodifusión con calidad de CD.

El sistema DAB se basa en dos tecnologías fundamentales:

- **MUSICAM**: es un tipo de compresión de datos utilizado para establecer las capas del MPEG, el cual elimina partes del sonido que no sean necesarias, pues son solapadas por otras frecuencias, o serían obviadas por el oído humano.

- **COFDM**: La modulación por multiplexado por división de frecuencia ortogonal codificada es una ampliación de la tecnología OFDM, la cual utiliza varias portadoras simultáneamente siempre ortogonales pero modulada cada una de ellas en amplitud y fase. LA única diferencia entre la modulación utilizada en el DAB y la OFDM, es que la COFDM está diseñada añadiéndole una codificación del canal que permite la detección y corrección de errores. Estas técnicas aportan varios beneficios al nuevo sistema de radio como por ejemplo ayudar a evitar interferencias debidas a efectos multicamino, es decir, cuando una portadora llega al receptor por dos caminos distintos y con un retraso, esto, se puede proteger estableciendo un intervalo de guarda entre símbolos consecutivos.

A continuación se va a explicar la estructura utilizada para la transmisión de datos. Para realizar esta transmisión es necesario, utilizar datos para sincronizar cada portadora, para rellenar la información del sistema (tipo de codificación utilizada, protección, etc.). Además, es necesario, como es obvio, enviar tanto los datos como el audio que se desean enviar al receptor. Para cualquier ampliación de la información de este capítulo se recomienda la lectura del estándar de DAB cuyo enlace se puede encontrar en el punto [2] de la bibliografía.

1.2 Estructura de la transmisión

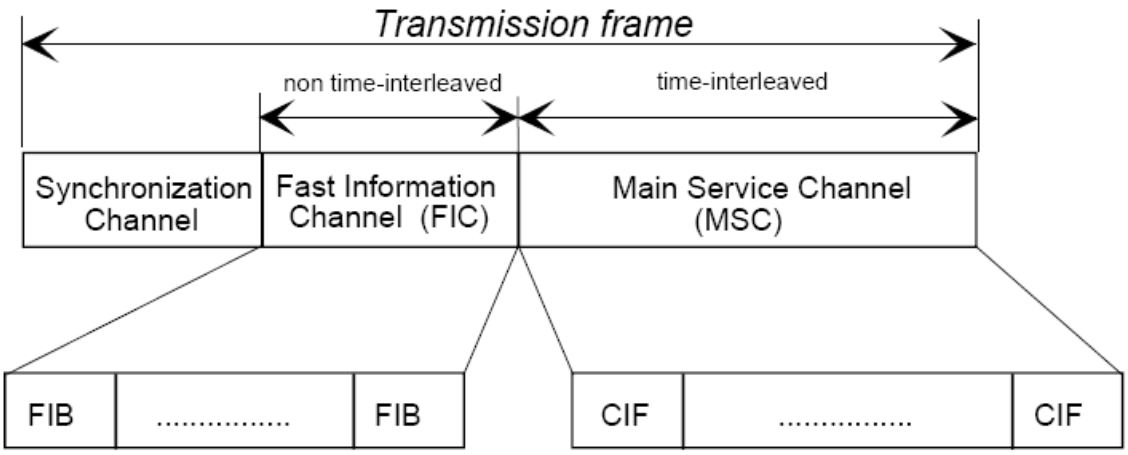


Ilustración 2: Estructura de la transmisión

Cada portadora usada en la transmisión, se divide en tres partes fundamentales:

- **Sychronization Channel** (SC): Es la parte usada para poder realizar la sincronización del canal. Esta sección está compuesta por los dos primeros símbolos OFDM recibidos en la transmisión. El primero se corresponde con el *Null Symbol* el cual es utilizado para conocer donde comienza la transmisión, dicho símbolo se detecta gracias a que su energía es cero (en ciertos casos podría contener cierta información, pero no será tratado en este documento). El segundo símbolo es el *Phase Referente Symbol*, donde se encuentra la información necesaria para la modulación diferencial del siguiente símbolo, pues éstos pueden llegar desfasados al receptor, y como explicaremos más adelante, el receptor se encarga de alinear para su futura descodificación.

- **Fast Information Channel** (FIC): Está compuesto por bloques del mismo tamaño, llamados *Fast Information Blocas* (FIB). En esta sección del sistema se encuentra la información correspondiente a la información del sistema, como por ejemplo, hora y fecha, configuración de multiplexación, información para una reconfiguración, número de subcanales, etc. Como veremos en un estudio en profundidad de esta parte de la estructura realizado más adelante, dependiendo del tipo de boques que contengan los FIB, variará la clase de información aportada por el FIC.

- **Main Service Channel** (MSC): La última parte de la estructura, se corresponde con la propia información que se desea transmitir. Ésta puede ser tanto de datos como de audio, pues como hemos comentado anteriormente, una de las ventajas del DAB, es la posibilidad de transmitir tanto audio como datos, además se puede transmitir esta información y hacerla corresponder con

diferentes subcanales. El MSC está construido a partir de *Common Interleaved Frames* (CIFs), y cada CIF ocupa 55296 bits. El número de CISS y de subcanales incluidos en el MSC variará dependiendo de ciertas circunstancias que se valorarán, como ya hemos comentado, en un estudio en profundidad realizado sobre el FIC y el MSC, que podemos encontrar a continuación.

1.2.1 Fast Information Channel

La razón de dedicar un apartado específico para el FIC y otro para el MSC, es la cantidad de información relevante que transmiten y, aunque la información del SC también es vital para la transmisión, su función se ha podido resumir brevemente. Sin embargo, el FIC, dependiendo del tipo de datos que contengan, así será su información.

En la siguiente tabla se puede observar que dependiendo de la duración de la transmisión, variará el número de FIBs y CIFs contenidos en la estructura, y por tanto, el tamaño del FIC y el MSC.

Tabla 1: Modos de transmisión.

Modo de transmisión	Duración de la transmisión	Número de FIBs	Número de CIFs
I	96 ms	12	4
II	24 ms	3	1
III	24 ms	4	1
IV	48 ms	6	2

Como ya hemos comentado, el FIC está compuesto por FIBs, bloques del mismo tamaño, 30 bytes más 16 bits de CRC. A su vez, estos FIBs están construidos a partir de FIGs y otras etiquetas.

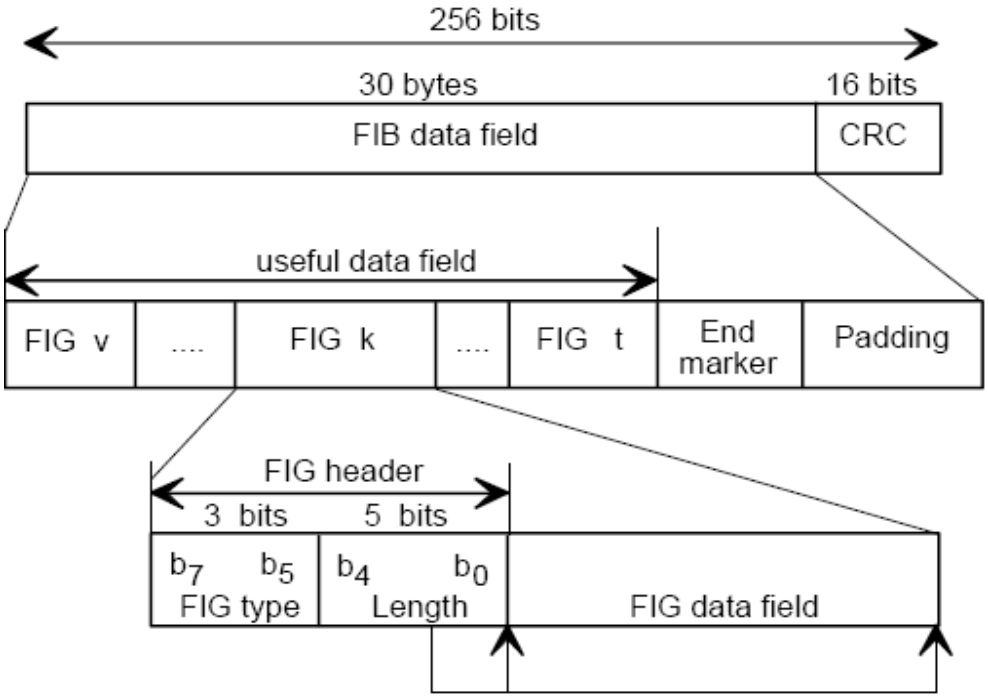


Ilustración 3: Estructura de FIB

En la ilustración 3, se puede observar como cada FIB está compuesto de varios FIGs, un campo de señalización del final de los campos FIG (*End Maker*, ocho 1's, que en realidad son un tipo especial de cabecera del FIG), y un campo llamado *Padding* utilizado para completar con ceros aquellos bits que no hayan sido utilizados por los FIG en caso de que no se utilicen tantos FIG como sean necesarios para completar el FIB.

Antes de empezar a estudiar la composición de un FIG, es importante resaltar la importancia del campo CRC, el cual es utilizado para confirmar, mediante redundancia, la correcta recepción de los datos del FIB.

Ahora sí, podemos explicar que el FIG tiene varios campos:

- **Tipo de FIG** (*FIG type*): 3 bits que sirven para identificar el tipo de datos que están contenidos en el FIG. Existen codificaciones específicas para referirse a cada tipo de FIG definidas en el Standard Europeo de DAB.

- **Tamaño** (*Length*): Los cinco bits que componen este campo representan la longitud en bytes del campo de datos del FIG. Está expresado en números binarios sin signo y los valores 0,30, y 31 están reservados. El campo anterior y este componen la cabecera del FIG, es decir, cuando estos dos campos son todo 1's, el sistema entiende que se trata del *End Maker* del FIB.

- **Campo de datos del FIG** (*FIG data field*): Este campo se puede resumir como el campo dónde se transmiten los datos de configuración del sistema. Aunque si sólo nos fijamos en este breve resumen, estaríamos obviando numerosos detalles importantes, por eso, a continuación se hace una

exposición detallada de algunos de los más relevantes tipos de datos que puede contener este campo dependiendo del tipo de FIG.

1.2.1.1 FIG tipo 0

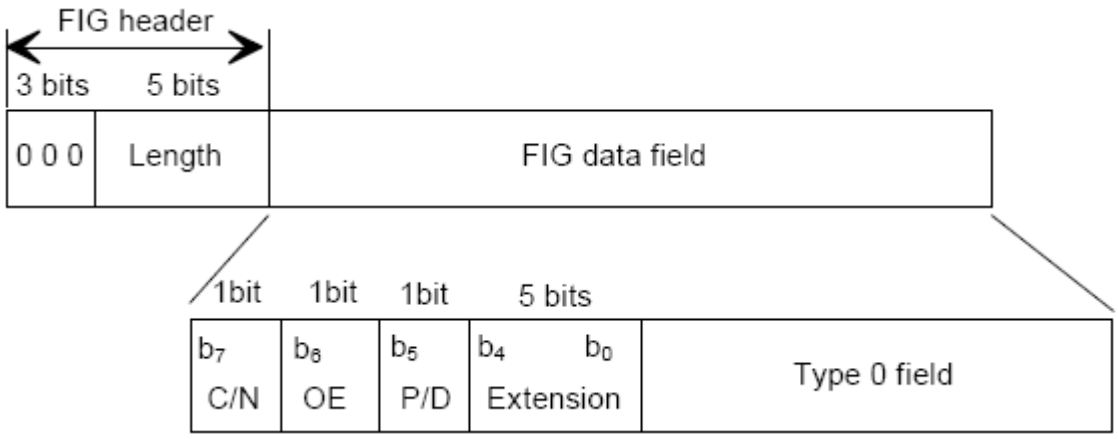


Ilustración 4: Estructura de FIG 0

Este tipo de FIG es utilizado para señalizar una inminente o futura reconfiguración de la multiplexación, hora y fecha, y otra información básica del sistema. En la figura se puede ver como la configuración de este tipo de FIG se corresponde con "000".

Veamos el significado de los campos que componen los datos del FIG:

- **C/N** (*Current / Next*): Mediante un cero, este campo especifica que la información enviada, se refiere a la actual configuración. Si el campo contiene el valor "1", toda la información contenida se corresponde con una futura reconfiguración. También puede extenderse esta información a conceptos relativos a una base de datos, pero estos son casos específicos no tratados en este proyecto.

- **OE** (*Other Ensemble*): Un bit que indica si la información contenida se refiere al sistema actual ("0"), o a otro (FM o AM). El significado de este campo, así como el de los demás del FIG *data field* puede tomar otros significados dependiendo del campo de la extensión, del cual hablaremos a continuación. De todos modos, es importante resaltar que toda esta información se puede ampliar con otros documentos, pues aquí sólo estamos tratando los casos específicos utilizados en nuestros proyectos.

- **P/D**: Si el valor de este campo es un "0", el significado que tomará es que el identificador del servicio se codificará mediante 16 bits, en caso contrario la codificación ocupará 32 bits.

- **Extensión**: Sirve para mediante 5 bits poder ampliar el significado de los datos del tipo del FIG, es decir, crear nuevos subtipos dentro de este FIG.

- **Type 0 field**: Campo concreto donde se encapsulará la información que se desea transmitir al sistema.

1.2.1.2 FIG tipo 1

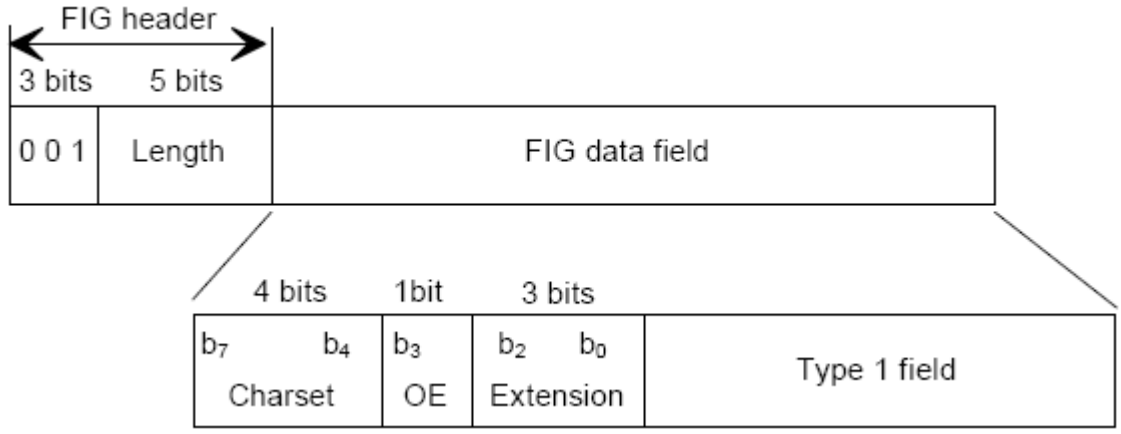


Ilustración 5: Estructura de FIG 1

Debido al parecido entre las estructuras de los diferentes tipos de FIG, sólo se van a mostrar en este documento el tipo 0 y 1. Sí es importante comentar que otros tipos como el 5 y el 6 sirven para transmitir información relativa al *Acceso Condicional* del sistema (opción a incorporar en la implementación del DAB), o al *Fast Information Data Channel* (FIDC, utilizado cuando se desea transmitir información, que debería ir incluida en el MSC, a través del FIC).

Se puede observar que la configuración utilizada para definir este tipo de FIG es "001". Este tipo define las etiquetas para mostrar y otra información relativa a ellas. Como se puede ver en la imagen, sus campos son:

- **Charset**: Se trata de 4 bits utilizados para identificar el grupo de caracteres utilizados para clasificar los caracteres contenidos dentro de los datos de este FIG.

- **OE**: Este bit tiene el mismo significado que en el FIG de tipo 0, es decir, indicar si la información contenida hace referencia a este o a otro sistema.

- **Extension**: Esta extensión de tres bits es usada para identificar el significado del FIG tipo 1 en casos concretos. Estas extensiones de momento no han sido definidas por lo que su significado queda reservado.

1.2.2 Main Service Channel

El Main Service Channel es la parte de datos que contiene, generalmente, el audio y los datos que llegarán finalmente al usuario. Como ya hemos dicho, está compuesto por unidades llamadas Common Interleaved Frames (CIFs), y cada una tiene un tamaño de 55296 bits. La unidad mínima direccionable dentro de un CIF se llama Capacity Unit (CU), y tiene un tamaño de 64 bits. Por lo tanto, con los datos anteriores, podemos decir, que cada CIF contiene 864 CUs, identificados por direcciones de la 0 a la 863.

En otro nivel de abstracción del MSC, éste está dividido en subcanales. Cada subcanal contiene un número entero de CUs, y al ser un número entero, cada CU ha de ser exclusivo para cada subcanal, es decir, dos subcanales no pueden compartir el mismo CU. Los datos transportados en el MSC son divididos en ráfagas de 24 ms correspondiéndose con el tamaño de datos del subcanal de cada CIF. Cada ráfaga de datos constituye un periodo lógico, y cada uno de éstos se corresponde con un CIF. Los sucesivos CIFs son identificados por un contador de CIFs.

Aunque no se va a entrar en detalle en este estudio del DAB debido a que no fue demasiado relevante a la hora de desarrollar el proyecto, es importante comentar y explicar brevemente que existen dos modos de transporte en el MSC:

- **Stream mode** (modo de flujo): Este modo permite a la aplicación recibir y repartir los datos de manera transparente desde el transmisor al receptor. Estos beneficios serán posibles siempre y cuando la tasa de datos esté fijada a un múltiplo de 8 Kbit/s. En este método, la aplicación también está capacitada para proveer información bajo demanda, o incluir un método de tratamiento asíncrono de datos para tasas más bajas. Otra particularidad de este modo es que sólo un componente de servicio puede ser transportado en cada subcanal.

- **Packet mode** (modo de paquetes): Como diferencia principal entre este modo y el anterior, podríamos resaltar que aquí si se permite que diferentes componentes de servicio sean transportados en el mismo subcanal. Por otra parte, como parecidos, en ambos modos la tasa de datos ha de ser múltiplo de 8 Kbit/s. Como particularidades de este método, cabe destacar que los datos pueden transportarse bien en grupos o bien en paquetes individuales. Existe un campo en la trama que indica cual de las dos opciones anteriores es la elegida para la transmisión. Cada paquete es identificado por una dirección. Esto permite que el envío de paquetes no sea ordenado, pues al recibir todos los paquetes se vuelve a construir la secuencia. Todos los paquetes tienen una longitud fija y existen cuatro longitudes diferentes permitidas que veremos más adelante. También existe la posibilidad de que paquetes con distintas longitudes sean enviados en el mismo subcanal, el cual está compuesto por varios paquetes. Aquí entra la labor de los paquetes de “padding” o relleno, son paquetes que se utilizan para completar la trama en caso de que no se quieran enviar suficientes paquetes, o las longitudes de éstos no sean suficientemente grandes. En la ilustración 6 se puede observar como todos los paquetes están compuestos por una cabecera, un campo de datos y un campo de CRC para comprobación de errores de transmisión.

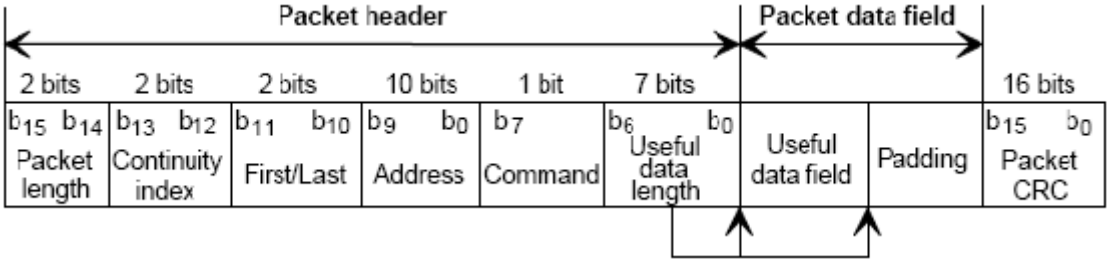


Ilustración 6: Estructura del MSC

Como ya se ha comentado (y se puede observar en el primer campo de la ilustración 6), la longitud de cada paquete es fija, y viene dada por el campo "Packet Length", que fija las posibles longitudes mediante 2 bits con la siguiente configuración.

Tabla 2: Configuración Packet Length del MSC.

Packet Length	Longitud del paquete (bytes)	Longitud del campo de datos (bytes)
0 0	24	19
0 1	48	43
1 0	72	67
1 1	96	91

Ya que la longitud de las cabeceras de los paquetes es siempre la misma, las diferentes longitudes de cada paquete sólo afectan al campo de datos del paquete.

El grupo de datos del MSC es transmitido en uno o varios paquetes que comparten la misma dirección (estos paquetes si deben ser recibidos en orden). En la ilustración 7 se muestra como el grupo de datos del MSC es transmitido en varios paquetes con la misma dirección j. Además, si recordamos la figura anterior, podemos ver el valor de la etiqueta "First/Last" en cada paquete ("First"=1 en el primero, "Last"=1 en el segundo).

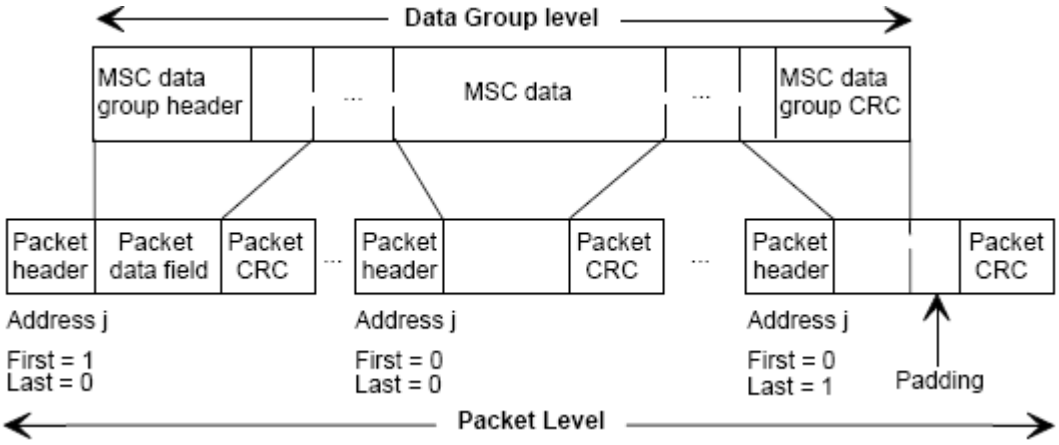


Ilustración 7: Composición del MSC.

Cada paquete individual tiene su propio CRC y aparte, el campo CRC del MSC está compuesto por paquetes que también contienen un CRC propio del paquete. Este CRC particular no comprueba la recepción general del MSC, sino la recepción particular del paquete.

Anteriormente se comentó que en ciertos casos es posible transportar información del FIC en el MSC, este particular caso es debido a un direccionamiento al Canal Auxiliar de Información (AIC), el cual utiliza el subcanal 63 y la dirección de paquete 1023. Para ello el grupo de datos del MSC particular debe cumplir ciertas características:

- Debe estar compuesto por FIG's.
- Debe poder transportar diferentes tipos de FIG.
- El grupo de datos debe estar definido como "Datos generales" ("General Data").
- Su tamaño máximo será de 512 bytes.

A continuación se puede observar un ejemplo de su estructura:

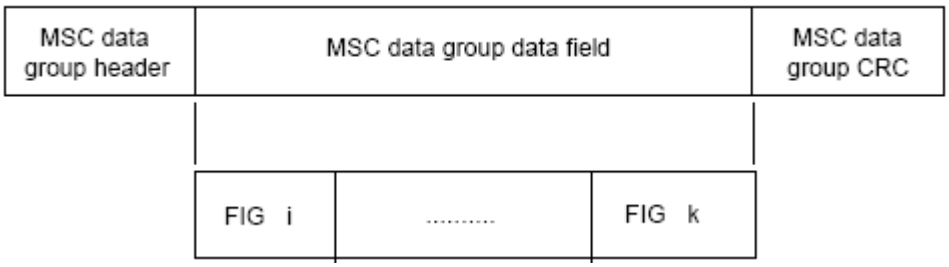


Ilustración 8:Estructura MSC.

1.3 Multiplex Configuration Information (MCI)

La manera más común de acceso de un usuario a los servicios que puede proporcionar DAB es mediante la selección de un servicio. Estos servicios pueden contener también componentes de otros servicios y se dividen en dos: servicios primarios y secundarios. Los servicios primarios, suelen ser los que transportan el audio, aunque se puede dar el caso de que sean datos. Los servicios secundarios son opcionales por lo que no siempre se presenta en una transmisión.

Los servicios están compuestos por componentes de servicio, que a su vez, están relacionados con subcanales. Estas relaciones son organizadas por el MCI cuyas principales funciones son:

- Organizar los subcanales en cuanto a su posición y tamaño en los CIF y a su protección de errores.

- Listar los servicios disponibles en el sistema.

- Establecer las conexiones entre componente de servicio y servicio.

- Establecer relaciones entre subcanales y componentes de servicio.

- Señalizar una reconfiguración de la multiplexación.

El tipo de FIG utilizado para codificar el MSI es el de tipo 0.

1.3.1 Organización de los subcanales

Esta es una de las principales funciones del MCI, y es codificada mediante un FIG tipo 0 con extensión 1. Para cada subcanal existe información específica como la dirección de comienzo (de 0 a 863 CU's), el tamaño o la protección de errores utilizada. Pueden ser direccionados hasta 64 subcanales usando valores de 0 a 64 para el identificador de subcanal. Existen dos maneras de señalizar el tamaño y la protección de errores utilizada para un subcanal que se explicarán en profundidad más adelante. De momento diremos que una manera es más larga (*Long form*) que la otra (*Short form*).

En la ilustración 9, se puede observar una trama del FIG utilizado para la organización de los subcanales y las dos opciones existentes para señalizar el tamaño y la protección de errores.

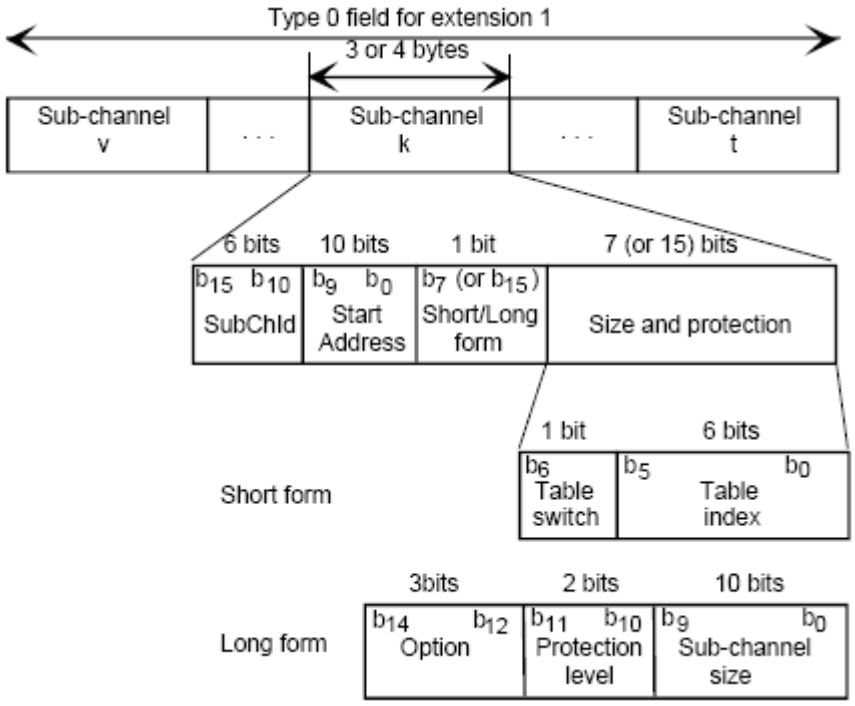


Ilustración 9: Estructura de los subcanales

Veamos el significado de cada campo para poder entender mejor la trama y su codificación:

- **SubChId**: Este campo se corresponde con el identificador utilizado para hacer referencia al subcanal.

- **Start Address**: Escrito como un número binario sin signo, utiliza 10 bits para hacer referencia a un CU (de 0 a 863), el cual será el CU de inicio del subcanal.

- **Short/ Long form**: Este flag indica el modo utilizado para señalizar el tamaño y el modo de protección de errores del subcanal. Cero se refiere a Short form y uno a Long form. Llegados a este punto, es necesario comenzar a explicar los dos modos de señalización.

  Short form: Esta forma se corresponde con el llamado *Unequal Error Protection* (UEP), orientado más a audio que a datos, aunque no se excluyen los segundos. Mediante este modo en la trama se indica la tabla a observar (mediante *Table Index* =0, el 1 está reservado para codificaciones futuras), y en dicha tabla se encuentra el tamaño del subcanal (en CU's), el nivel de protección, y el Bit Rate, tasa de bits (en Kbit/s). El campo encargado de indicar la fila a observar de la tabla es el *Table Index*.

  Long form: Si hablamos de esta opción, nos estamos refiriendo al *Equal Error Protection* (EEP). Este modo está orientado tanto a audio como a datos, de hecho existen ciertas tasas de bits para audio no definidas para el modo anterior que este modo complementa. Esta forma de señalización incluye ocho niveles de protección de datos separados en dos grupos (A y B). Para conocer cuál de los dos grupos es el utilizado en el subcanal se usa el campo Option (3 bits, '000' y '001'). Una vez elegido el modo de protección, el campo *Protection level*, especifica el nivel de protección dentro el grupo escogido; este campo está compuesto por dos bits que codifican los cuatro niveles posibles. Por último, se encuentra el campo *Sub-channel size* donde se puede conocer el tamaño del subcanal indicado en CU's (de 1 a 864). Existen tablas que relacionan la tasa de bits con el tamaño del subcanal [2].

1.3.2 Organización de los servicios

Este apartado define los servicios y componentes de servicios transportados en el sistema. Al igual que la organización de los subcanales, se trata de un FIG de tipo 0, pero con la diferencia de que en este caso las extensiones utilizadas son las 2, 3,4 y 7.

Cada servicio se corresponde con un Identificador de Servicio, el cual es único en el mundo al aplicarle un campo llamado Código de Extensión de País. Cada componente de servicio tiene un identificador único en el sistema.

La ilustración 10 muestra un esquema de la trama de definición de un servicio básico con sus componentes de servicio. Se trata de un FIG tipo 0 con extensión 2. La trama representa como cada servicio se divide en una cabecera y la definición de sus componentes de servicio. La cabecera incluye los identificadores necesarios del servicio y el número de componentes de servicio asociados. La siguiente parte de la definición, se corresponde con el particular componente el cual tendrá un identificador, una relación con el subcanal, tipo de datos, etc.

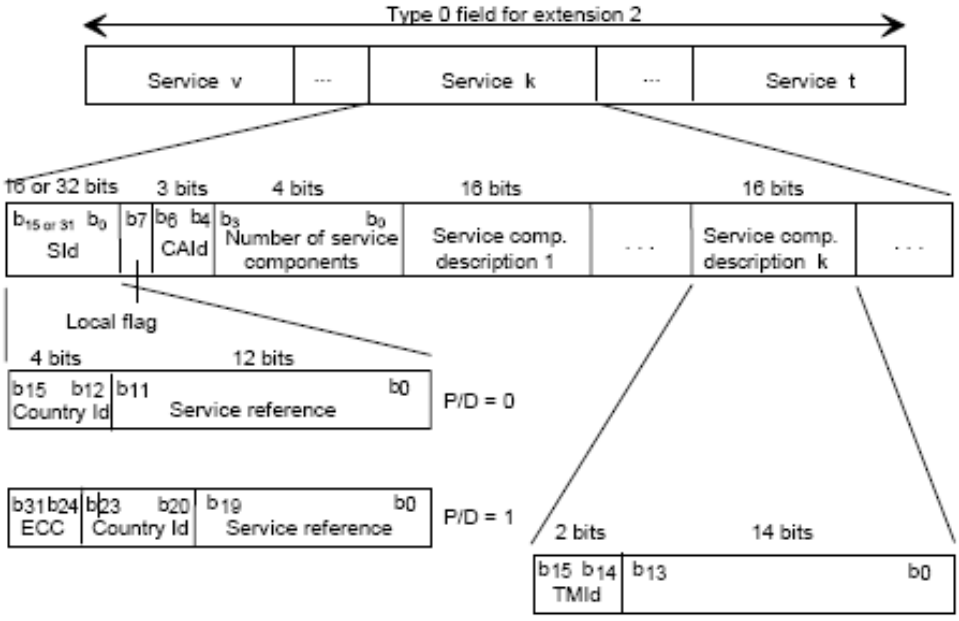


Ilustración 10: Estructura de los servicios

Puesto que este apartado de estudio de Digital Audio Broadcasting trata de una orientación necesaria para entender las funciones implementadas en el proyecto, no se va a entrar en más detalle sobre otras funciones del DAB. De nuevo, cualquier ampliación de información acerca de estas funciones se puede encontrar en el documento estándar de DAB [2] cuya referencia se incluye en la bibliografía.

1.4 Estructura general de DAB

Como último apartado de este estudio, es importante conocer también la estructura o esqueleto del DAB, es decir, como se divide su comportamiento. Hasta ahora, la visión que hemos tenido del DAB ha sido orientada a sus tramas de datos, sus posibles configuraciones y tipos de datos. Una vez conocemos sus funcionalidades, veamos cómo las lleva a cabo de una manera general para poder entender su posterior implementación, siempre desde el punto de vista del receptor, por lo que la parte del transmisor la vamos a obviar, ya que está fuera del objetivo, aunque a grandes rasgos, es el inverso del receptor.

Existe un esquema de diferenciación de módulos dentro del DAB que estudiaremos en profundidad al hablar del flujo de diseño, pero de momento vamos a intentar diferenciar a grandes rasgos las partes que pueden componer un receptor DAB.

Tratando de orientar esta diferenciación para explicar la parte del receptor en la que se centra el proyecto, podemos decir que existen tres partes:

- **Recepción de la señal**: Esta parte incluiría el llamado *Front End* de la aplicación donde se trataría la señal para luego procesarla en los sucesivos módulos. Esta primera parte no se diseñó durante el proyecto ya que en la empresa NXP tienen gran experiencia en tratamiento de señales y, por lo tanto, software óptimo para llevarlo a cabo.

- **Demodulación y demultiplexación**: Esta es la parte en la que se centró el proyecto. Una vez la señal está lista para ser tratada se le aplica la transformada de Fourier, se demodula, se le aplican métodos de recolocación de datos, se realiza una cuantificación de la señal y, se demultiplexa según sea información perteneciente al FIC o a un subcanal. Una vez se ha hecho la demultiplexación, el flujo de ejecución es particular para cada subcanal o para el FIC pero, a grandes rasgos, se puede decir que se le aplican diferentes métodos para la recolocación de datos utilizada para evitar posibles errores durante la transmisión.

- **Decodificación:** Este último apartado está fuera del objetivo del proyecto, es la parte donde se procede a decodificar los datos dependiendo de su codificación original (mp2, mp3, etc.). Estos módulos sólo se le aplican a los subcanales y no al FIC.

Una vez visto a grandes rasgos el sistema, podemos empezar a estudiar y entender y la división en diferentes módulos de la parte del receptor que se implementó y además también veremos como aunque la división de trabajo se hiciese casi siempre en módulos independientes, era necesaria una comunicación constante dentro del equipo de trabajo debido a la fuerte interrelación de dichos módulos.

Capítulo 2. Procesador CoolFlux DSP

2.1. Introducción

El imparable avance en el campo de la electrónica, particularmente en las técnicas de fabricación de circuitos integrados, ha tenido un gran impacto en la industria de las comunicaciones. El rápido desarrollo de la tecnología de integración a gran escala (VLSI, Very Large Scale Integration) de circuitos electrónicos ha estimulado el desarrollo de computadores digitales más potentes, pequeños, rápidos y baratos, cuyo hardware responde a propósitos específicos. Esta tecnología ha hecho posible construir sistemas digitales altamente sofisticados, capaces de realizar funciones y tareas del procesado de señal digital que normalmente eran demasiado difíciles y/o caras con circuitería o sistemas de procesado de señales analógicas.

Los DSP's modernos son apropiados para su implementación bajo el criterio VLSI. Las grandes inversiones necesarias para diseñar un nuevo circuito integrado sólo pueden ser justificadas cuando el número de circuitos a fabricar es grande, o cuando los niveles necesarios de desempeño son tan altos que no pueden ser alcanzados con la tecnología existente. A menudo, ambos argumentos se unen la misma tendencia, particularmente en comunicaciones y aplicaciones dirigidas a los consumidores. Los avances en la tecnología de fabricación de circuitos integrados también abren nuevas áreas de desarrollo basadas en DSP, tales como sensores inteligentes, visión de robots y automatización, mientras pone las bases para continuar con el desarrollo tecnológico en áreas tradicionales del procesamiento de señal digital, tales como música, voz, radar, sonar, video, audio y comunicaciones.

Dentro del procesamiento de señal digital, en los últimos años se ha experimentado un direccionamiento de las tendencias del mercado tecnológico hacia las telecomunicaciones móviles de altas prestaciones, lo que ha provocado un creciente interés por arquitecturas empotradas de bajo consumo. La proliferación y popularidad de dispositivos portátiles hace que el bajo consumo unido a la alta productividad se haya vuelto imprescindible. Además, como consecuencia de la complejidad creciente de los protocolos de comunicación y del advenimiento de la era multimedia, se ha producido un espectacular aumento de los requerimientos computacionales para abordar funcionalidades cada día más demandadas. Por otro lado, cabe comentar que el ciclo de vida de los dispositivos móviles y multimedia se está reduciendo considerablemente debido a la presión del mercado, lo que provoca a su vez una reducción del tiempo de desarrollo de los procesadores. Esto significa que lograr la máxima eficiencia computacional con el mínimo consumo de energía y en el menor tiempo posible se ha convertido en la meta de varias compañías de desarrollo de DSPs.

Con la intención de dar respuesta a estas necesidades nace el proyecto CoolFlux, un DSP desarrollado en primera instancia por Philips PDSL (Philips Digital Systems Labs) en el año 2002, pasando a ser parte de las competencias del grupo Emerging Semiconductor Business Healthcare de NXP tras la escisión de esta última de su compañía nodriza, Philips.

En etapas iniciales de diseño el objetivo principal fue lograr un consumo energético reducido y hacerlo minimizando el tamaño del procesador e incluyendo todas las funcionalidades básicas de un DSP, cuyas aplicaciones objetivo se refieren principalmente al proceso de audio. Una combinación de técnicas a todos los niveles puede dar lugar a una solución óptima: esto se conoce como enfoque holístico, en el cual se aborda cada aspecto independientemente, no solo como una parte del todo ([6], [7]). Siguiendo este enfoque holístico, encontramos las siguientes decisiones de diseño:

- Optimización para un determinado dominio de aplicación. El procesador está diseñado específicamente para obtener el mejor rendimiento posible en aplicaciones objetivo, en este caso el proceso de señal de audio. Para ello, las anchuras de instrucciones y ruta de datos fueron elegidas para adaptarse mejor este tipo de procesamiento.

- Minimización del tamaño del procesador. Área reducida y menos puertas de tipo toggle (on/off), así como una distribución compacta, contribuyen a reducir el consumo de energía.

- Minimización del tamaño de la memoria y los accesos a ella. La memoria contribuye significativamente al área total del procesador y a su consumo de energía.

- Introducción de paralelismo. El procesador está dotado de unidades funcionales y buses de comunicación replicados para obtener paralelismo en ejecución, adquiriendo un compromiso entre tamaño del procesador y ciclos de ejecución.

- Optimización del ratio entre la máxima frecuencia de reloj y el número de ciclos requeridos para las aplicaciones objetivo. De manera que se aplican técnicas de escalado dinámico de voltaje.

- Programación en C. Siguiendo las tendencias actuales en el desarrollo de DSPs, el procesador no se programa en ensamblador como se solía hacer, sino que se hace en ANSI-C. Mejora la productividad siempre y cuando la acción combinada del compilador y de las librerías de subrutinas haga que el rendimiento sea equiparable al obtenido con programación en lenguaje ensamblador.

La reducción del ciclo de desarrollo de los procesadores comentada anteriormente hace que se planteen nuevos sistemas de diseño de software que sean altamente reutilizables en futuras generaciones del procesador. Por este motivo el diseño del procesador se abordó desde una perspectiva redireccionable, es decir, adaptable a nuevas especificaciones para el sistema. Para ello se utilizó el entorno de desarrollo de Target Compiler Technologies (TCT), el cual permite definir una arquitectura específica y genera un compilador C para la misma, el cual es utilizado para transformar los programas en C a código máquina, que puede entonces ser simulado y evaluado.

Para tener una noción clara de lo que la programación en C y el enfoque re-direccionable suponen en el desarrollo de CoolFlux, reproduciremos las palabras de Johan Van Ginderdeuren, manager del proyecto en Philips y posteriormente en NXP:

> *"We wanted our new DSP to outperform other solutions, not only in terms of power consumption and cost, but by offering very efficient C programmability at the same time. For audio applications, assembly programming has often been deemed mandatory to meet the ultra-low power requirements. However, the CoolFlux DSP product has shown us the key to compiler friendly low-power design, which is the use of retargetable compilation technology"* [18]

El entorno de desarrollo de software proporcionado por TCT, llamado "Checkmate for CoolFlux DSP, es una instancia específica de Chess/Checkers, el conjunto de herramientas re-direccionables de TCT para el diseño, la programación y verificación de procesadores. Las herramientas de Chess/Checkers soportan múltiples arquitecturas, desde pequeños microprocesadores a DSPs, procesadores VLIW, vectoriales e incluso ASICs programables.

Checkmate for CoolFlux contiene herramientas de programación y de verificación:

> El entorno de programación incluye el compilador C optimizado, un ensamblador/desensamblador y las herramientas de linkado, todo ello controlado desde un mismo IDE. El compilador C, llamado Chess, ofrece soporte para ANSI-C y de punto fijo para el procesador CoolFlux, y genera código máquina altamente eficiente que elimina la necesidad de programar en ensamblador para obtener el mejor rendimiento (aunque esta forma de programación también es posible en la herramienta). Chess es diferente de los compiladores de C convencionales como gcc: usa modelado basado en grafos y técnicas de optimización para proporcionar código altamente optimizado para arquitecturas especializadas que exhiben peculiaridades tales como pipelines complejos, estructuras de registro heterogéneas, unidades funcionales específicas y paralelismo a nivel de instrucción.
>
> El entorno de verificación consiste en un debugger gráfico, que puede ser conectado tanto a un simulador llamado Instruction Set Simulator (ISS) como al propio procesador vía interfaz JTAG.

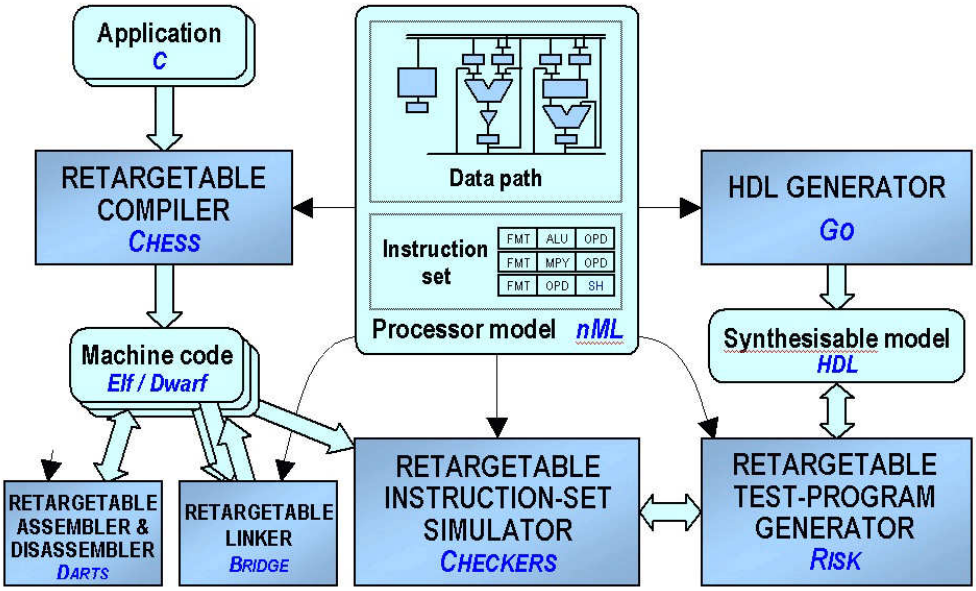


Ilustración 11: Estructura de Checkmate para CoolFlux DSP

El equipo de desarrollo de CoolFlux ha utilizado intensivamente este conjunto de herramientas para modelar el procesador, debido a la comodidad que supone a la hora de explorar y optimizar la arquitectura la posibilidad de contar con la información del compilador y del ISS. Con estas herramientas, la arquitectura del repertorio de instrucciones podía ser evaluada y ajustada a los requerimientos de aplicaciones que típicamente tendría que abordar el procesador, al mismo tiempo que el compilador podía ser evaluado y mejorado igualmente. Para obtener más información acerca de las herramientas de Target Compiler Technologies, ver [13].

2.2. Arquitectura

La última generación del CoolFlux es la llamada CoolFlux BSP, bajo la cual este proyecto fue finalizado, y de la cual pasamos a detallar las principales características. Para profundizar en el tema, ver [8].

2.2.1 Características generales

Para una arquitectura de bajo consumo energético los aspectos más importantes a tener en cuenta son la estructura del pipeline, la implementación de los decodificadores, la estructura del campo de registros y el paralelismo.

Como veremos con más detalle a lo largo de esta sección, CoolFlux reúne varias características que hacen de él un sistema de muy bajo consumo energético. He aquí algunos ejemplos preliminares:

- Dispone de un pipeline sencillo y bien balanceado en lo que a tiempo de ciclo de procesador se refiere, por lo que da libertad para aplicar técnicas de escalado de voltaje, las cuales redundan en una reducción cuadrática del consumo.

- Usa un decodificador de instrucciones segmentado, es decir, utiliza varios decodificadores para las diferentes secciones, en lugar de un único decodificador para toda la instrucción: estos decodificadores de sección se activan solamente cuando se necesitan, lo que supone un ahorro de energía a considerar.

- Los registros están situados siguiendo una política de localidad espacial con respecto a los recursos computacionales que los utilizan, complicando la gestión de los registros pero favoreciendo el menor consumo energético.

La mayor parte de las operaciones que se pueden llevar a cabo sobre la arquitectura pueden ser realizadas en modo sencillo o en modo empaquetado. El modo empaquetado comprende dos tipos de datos diferentes: complex y SIMD.

2.2.2 Descripción de la Arquitectura

La arquitectura hardware de CoolFlux BSP comprende una arquitectura Harvard, cuya memoria de programa tiene 32 bits de anchura (Memoria P) y dos memorias de datos de 24 bits (Memoria X y Memoria Y) de 16M palabras cada una. La ruta de datos de 24/56 bits dispone de dos multiplicadores de 24x24 bits (precedidos ambos por sumadores/restadores) y tres ALUs, además de dos unidades de redondeo, saturación y selección (unidades RSS).

Hay dos registros de 24 bits y dos acumuladores de 56 bits a ambos lados de la ruta de datos (Lado X y Lado Y). Dado que los dos MACs pueden operar en paralelo, ambos registros acumuladores pueden actuar como registros de destino simultáneamente. Hay dos buses de sistema, para mover datos, uno a cada lado del datapath, los cuales soportan movimientos sencillos y también movimientos paralelos. Un movimiento sencillo consiste en el transporte de un dato de 24 bits y se hace en un ciclo de reloj, mientras que un movimiento paralelo consiste en mover dos elementos de 24 bits usando los dos buses simultáneamente y, por tanto, en un mismo ciclo de reloj.

Además de la ruta de datos, hay unidades de generación de direcciones (AGUs), una para manejar los accesos a la memoria X y la memoria Y. En el lado X hay 8 registros de generación de dirección, mientras que en el lado Y tan solo hay 4. También hay 2 bloques de conversión de datos, puesto que hay varias anchuras de datos presentes en la arquitectura (24 bits, 56 bits).

CoolFlux BSP es una arquitectura de carga/almacenamiento, lo que supone que todos los operandos de memoria deben ser previamente cargados en registro antes de operar con ellos, así como todos los resultados almacenados en registros han de ser movidos explícitamente a memoria. Los registros están organizados en campos de registro cuyo nombre sirve para designar a cada registro individualmente mediante la adición de un índice. Todos los registros de un mismo campo de registros tienen la misma

anchura. Puesto que hay diferentes anchuras de datos para los diferentes tipos de registro, es necesario, como ya se ha comentado brevemente y como veremos a continuación en profundidad, que haya módulos específicos de conversión de datos de anchuras diferentes. Como ejemplo de esto cabe mencionar el caso de los acumuladores presentes en ambos lados del datapath: tienen una anchura de 56 bits lo que implica que no pueden ser transportados a través de los buses Xbus e Ybus en un solo ciclo de procesador. Estos registros son divididos en tres subregistros, cada uno de los cuales puede ser, por separado, origen y destino en operaciones de movimiento de datos. Sin embargo, no pueden ser utilizados en operaciones aritméticas.

Existen registros que no están organizados en campos de registros, como pueden ser los registros relacionados con el tratamiento de interrupciones (Ver sección Interrupciones), el registro de estado y el puntero de pila.

Por otro lado, algunos de los registros pueden ser utilizados en modo empaquetado, y su anchura se divide en dos partes para alojar dos datos distintos (por ejemplo, real e imaginario en el caso del tipo complex). Estos registros son siempre considerados ‘con signo’ cuando trabajan en modo empaquetado.

Los accesos a las memorias se realizan en un ciclo de reloj. De hecho, en un solo ciclo de procesador cada memoria puede ser accedida independientemente de manera que hasta 2 datos pueden ser leídos/cargados. También hay un modo de memoria especial en el que las dos memorias son combinadas para formar un único espacio de direcciones de datos de 48 bits: la memoria XY. Además, CoolFlux dispone de un sistema de E/S mapeado en memoria de 16Mword de 24 bits. Esto significa que, desde un punto de vista software, la E/S puede ser considerada como una memoria al uso.

La replicación de hardware supone que la arquitectura es capaz de ejecutar simultáneamente dos operaciones tipo MAC, dos transacciones de memoria y dos actualizaciones de puntero por ciclo, y por tanto es muy eficiente en ciclos de ejecución para aplicaciones computacionalmente intensivas. Puede observarse que la arquitectura CoolFlux muestra una simetría muy marcada. Sin embargo, las operaciones sobre la parte X de la misma están más elaboradas, relegando a la parte Y a una función de apoyo en operaciones aritméticas paralelas, y cargando la parte X con la mayor parte de los requerimientos computacionales.

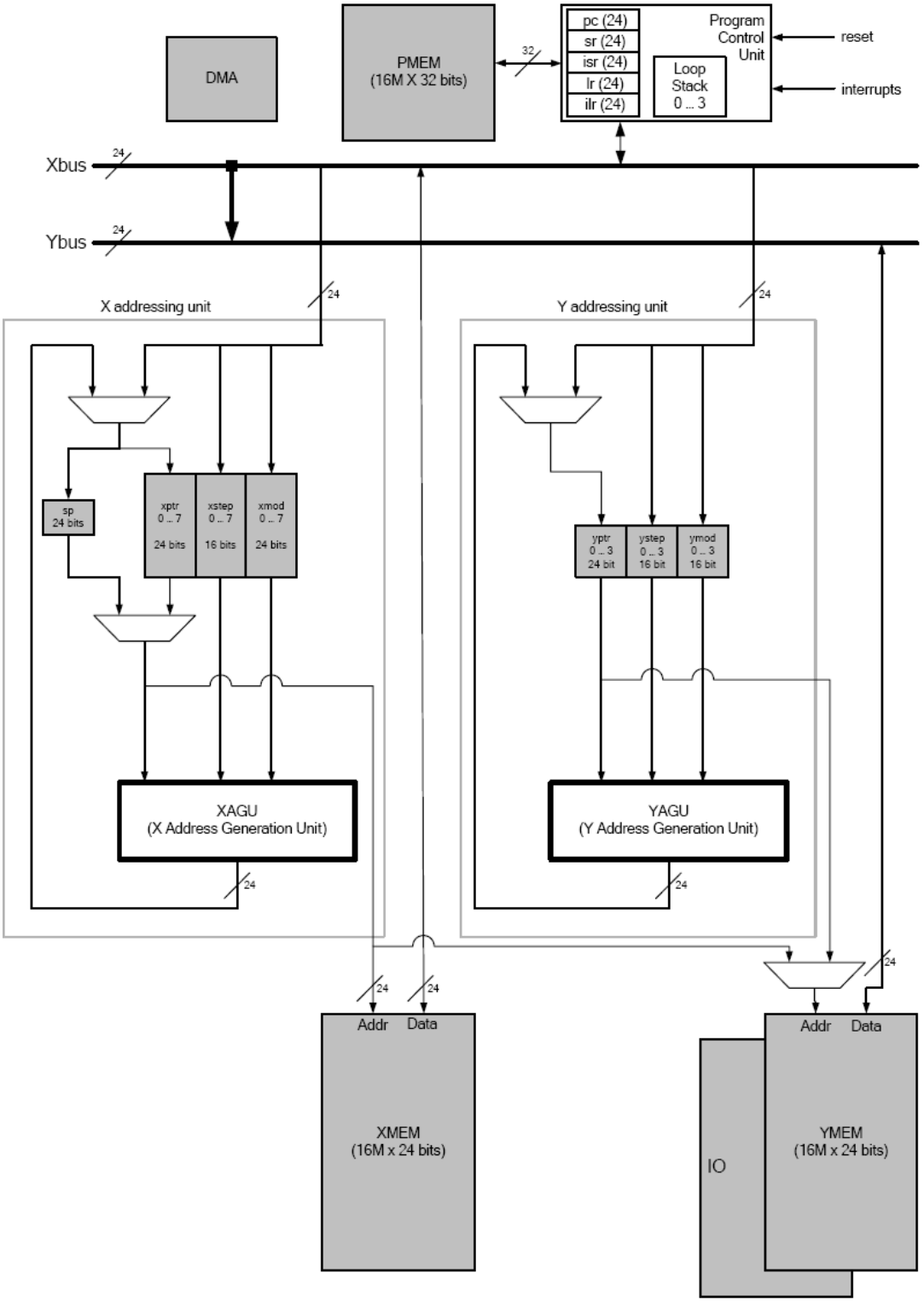


Ilustración 12: Unidades de direccionamiento, buses Xbus e Ybus, memorias, DMA y unidad de control de programa

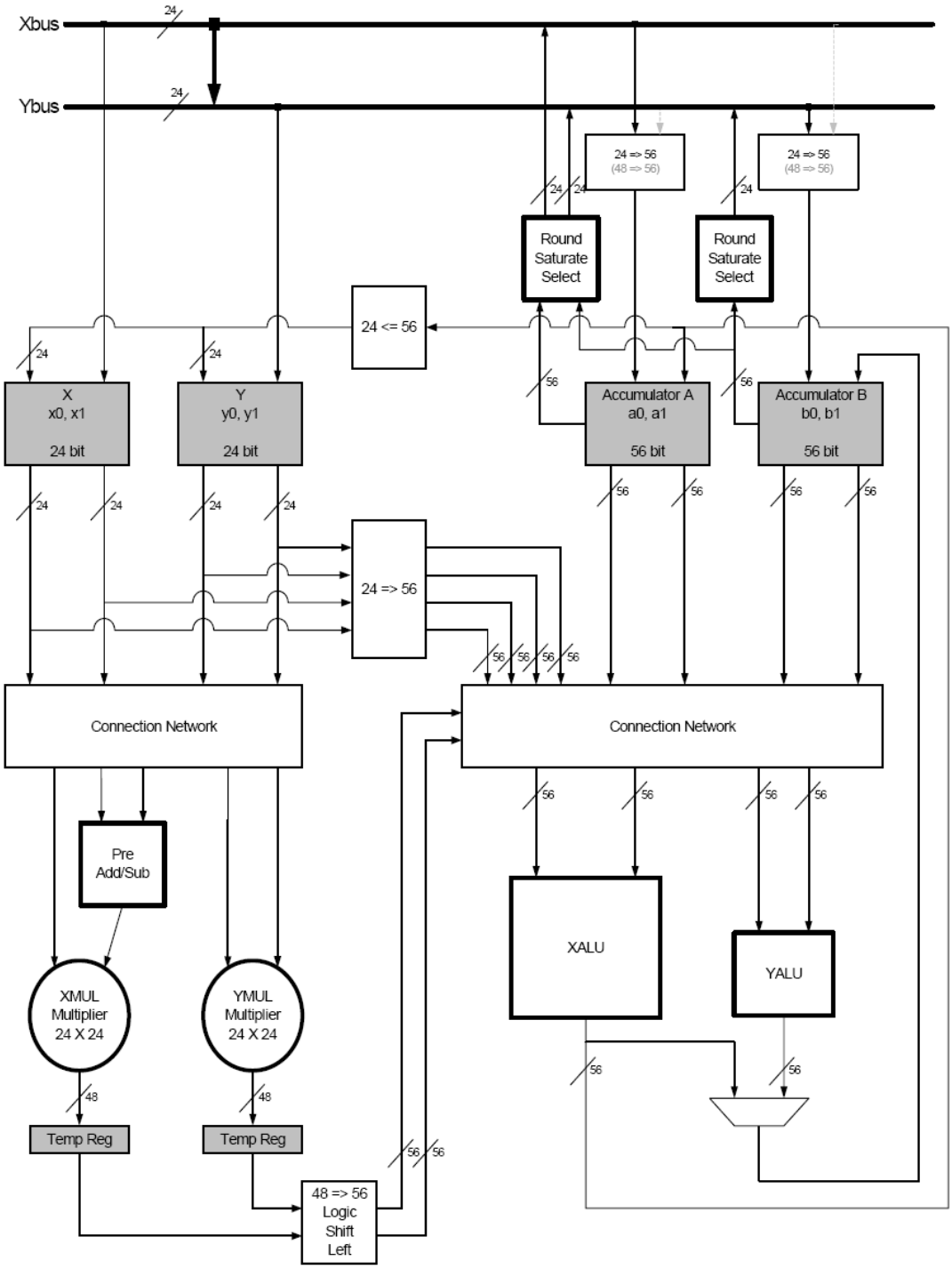


Ilustración 13: Ruta de datos, unidades RSS y buses de comunicación Xbus e Ybus

2.2.3 Repertorio de Instrucciones

En el desarrollo del repertorio de instrucciones se observa un compromiso entre el tamaño del código, tamaño del decodificador de instrucciones y la eficiencia del compilador para gestionar paralelismo a nivel de instrucción y las instrucciones de control.

Se trata de una arquitectura de 32 bits de instrucción con un conjunto de instrucciones tipo RISC combinado con un conjunto de instrucciones destinado a soportar el máximo paralelismo posible.

Los cuatro grupos principales de instrucciones son:

- el grupo de instrucciones long: utilizan los 32 bits de instrucción para gestionar operaciones de control y operaciones de aritmética de punteros sobre el puntero de pila.

- las instrucciones aritméticas/de movimiento: Combinan dos operaciones separadas. La primera parte es aritmética y puede consistir en una operación MAC, una operación ALU o ambas en paralelo. La segunda parte es completamente independiente a la primera y puede realizar transporte de datos sencillo o en paralelo (utilizando los dos buses). Pueden ser en modo escalar (AM) o en modo empaquetado (AMC).

- instrucciones de movimiento: optimización especial que combina 2 instrucciones AM escalares, para las cuales la parte aritmética es nop, en una instrucción de 2 movimientos escalares. Esta instrucción se ejecuta en 2 ciclos de procesador.

Asimismo, hay operaciones aritméticas condicionales que evitan usar el salto condicional con frecuencia, ahorrando ciclos de procesador.

Dado que el tamaño de instrucción es ligeramente superior a la media de tamaño de las instrucciones usadas en aplicaciones objetivo (experimentalmente, en torno a 25 bits de media), el compilador dispone de bastante libertad para organizar la ejecución de instrucciones de manera paralela y comprimir así el tamaño de código necesario para el mismo número de instrucciones.

Es importante resaltar que para disponer de un decodificador segmentado es necesario que la arquitectura del repertorio de instrucciones sea altamente ortogonal y modular.

2.2.4 Pipeline

CoolFlux tiene una estructura de pipeline sencilla y balanceada, contando con 3 etapas:

- Fetch (1 ciclo)
- Decode (1 ciclo)
- Execute (1 o 2 ciclos)

Este esquema básico se mantiene por varios motivos, entre los cuales subyace la reducción de consumo de energía que supone tener un pipeline de pocas etapas (aunque perdamos el rendimiento que nos ofrecería un pipeline más profundo). Además, un pipeline sencillo facilita la labor del compilador a la hora de encontrar la configuración de operaciones óptima.

2.2.5 Conversión de tipos: Unidades RSS

Como ya hemos comentado, las diferentes anchuras presentes en la arquitectura hacen que sea necesario tener módulos que se encarguen de convertir los datos y ajustarlos al formato de su destino.

Las operaciones de conversión de tipos se pueden clasificar en extensivas (de menos a más anchura) y limitantes (de más a menos anchura). Obviamente, las que presentarán más dificultad serán las que deban prescindir de ciertos bits (limitantes). Además, también se pueden clasificar en escalares y empaquetadas.

La conversión de tipos puede tener lugar en los siguientes sitios de la arquitectura:

- Desde y hacia los buses

Los buses tienen una anchura de 24 bits, por lo que los datos fuente son convertidos a 24 bits para su transporte y después son convertidos a la anchura de su destino, el cual puede ser en este apartado de 3, 8, 16 y 24 bits de anchura. Las operaciones extensivas son, por tanto, realizadas hacia el bus:

- sobre datos con signo se realizan mediante una extensión de signo añadiendo el número de bits necesario para completar la anchura de destino.

-sobre datos sin signo se añaden cero por el lado más significativo.

Las operaciones limitantes se llevan a cabo desde los buses hacia los registros de destino, y consisten en descartar los bits más significativos sobrantes.

Esto funciona así tanto para operaciones escalares como para empaquetadas.

- Desde y hacia sitios de 56 bits de anchura:

Este tipo de conversión puede llevarse a cabo en tres formas diferentes:

1. en los puertos de entrada y salida de las ALUs;
2. desde los buses al acumulador;
3. desde el acumulador a los buses.

Describiremos las políticas de conversión sin entrar en detalle, aunque éstas difieren en el número de bits extendidos y en el padding aplicado.

En el primer caso, la política para las entradas (de x a 56 bits) de las ALUs es extender el signo en un cierto número de bits y añadir ceros en la parte menos significativa del dato (padding). Para las salidas, descartar los posibles bits de overflow y los menos significativos (de 56 a X bits).

En el segundo caso la política es la misma que para las entradas de la ALU: cierta extensión de signo y cierto padding por el lado menos significativo hasta completar la anchura de 56 bits.

Para el tercer caso se requiere la participación de módulos dedicados, las unidades RSS, puesto que pasar de 56 bits a 24 bits puede suponer una pérdida importante de precisión si no se hace debidamente. Diferentes políticas de redondeo y saturación están a disposición del usuario para manejar los resultados de la manera que mejor le convenga.

2.2.6 Instrucciones multiciclo y delay slots

El hecho de que el pipeline sea tan sencillo tiene como consecuencia que las instrucciones de control necesiten varios ciclos de procesador para ejecutarse. Esto es debido a que modifican el contador de programa, y por tanto el pipeline ha de ser vaciado y rellenado de nuevas instrucciones.

La ilustración 14 muestra un cuadro de pipeline que se corresponde con la ejecución de instrucciones que no sean de control:

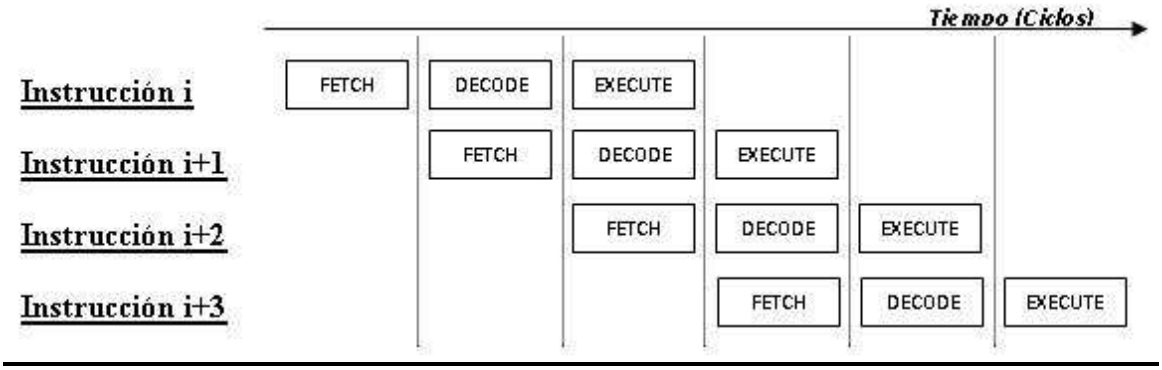


Ilustración 14: Pipeline de instrucciones no de control.

Como puede observarse, el pipeline está plenamente ocupado. Esta situación no se produce ante una instrucción de control, en la que el esquema de pre búsqueda de la siguiente instrucción podría fallar y ser necesaria la ejecución de otra instrucción diferente, en cuyo lanzamiento y decodificación se pierden ciclos. Cuando esto sucede, se dice que el pipeline está en parada.

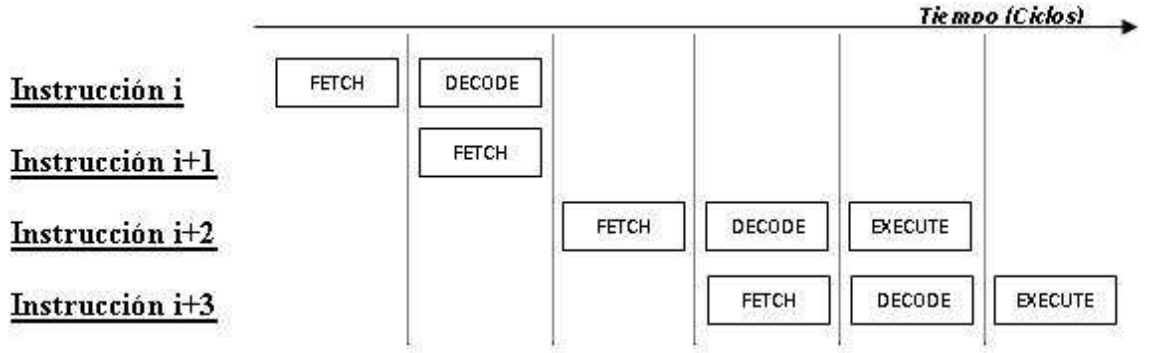


Ilustración 15: Pipeline con paradas.

La alternativa a considerar es, como siempre, aprovechar los delay slots y rellenarlos con instrucciones válidas o, en su defecto, instrucciones cuya ejecución no afecte ni al flujo ni al resultado del programa. Esta optimización no siempre es utilizable: a veces es necesario preservar la integridad de la ejecución del programa. En estos casos, sencillamente se vacía el pipeline y se buscan las instrucciones correctas.

2.2.7 Modos de direccionamiento

El direccionamiento está gobernado por las unidades de direccionamiento independientes para las memorias X e Y. Los tipos de direccionamiento soportados en CoolFlux son:

- direccionamiento directo
- direccionamiento indirecto con post-modificación
- direccionamiento indexado (con puntero normal o puntero de pila)

El direccionamiento indirecto es posible gracias a los dos campos de registros XPTR e YPTR localizados en las unidades de direccionamiento. Cuando una de las memorias es accedida a través de uno de estos punteros, el valor del puntero puede ser actualizado en paralelo con el acceso (para preparar un futuro acceso). Los cálculos para esta actualización del puntero son realizados en las unidades de generación de direcciones. El valor con el que calcular la post modificación se toma de los registros de XSTEP, YSTEP, XMOD e YMOD. De entre las posibles post-modificaciones, se destacan las más sencillas, como pueden ser:

- incrementos/decrementos en unidad
- incrementos/decrementos por 2
- incrementos/decrementos por valor del step register.

Por último, mencionaremos los modos de direccionamiento especiales, fundamentales para el tipo de tratamiento de señal que se pretende con esta arquitectura:

- bit reverse: utilizado en algoritmos específicos que utilizan diferentes etapas de las llamadas mariposas, como por ejemplo la FFT y el decodificador de Viterbi.

- buffer circular: define un buffer de tamaño programable en memoria y hace que las direcciones resultantes de la post modificación se correspondan con el inicio del buffer en caso de estar accediendo al final del mismo, asumiendo que la dirección actual pertenece al buffer y que el valor del offset no es mayor que el tamaño del buffer.

A través del puntero de pila se implementa el direccionamiento indexado, usando este puntero como registro base y calculando la dirección con un cierto desplazamiento. Esto proporciona soporte para mantener una estructura de pila.

Estas técnicas aseguran que los requerimientos de acceso a memoria sean minimizados así como también los ciclos de ejecución.

2.2.8 Bucles hardware

CoolFlux soporta hasta 4 bucles hardware anidados. Cuando una instrucción de bucle es ejecutada, la dirección de comienzo de bucle, la dirección de retorno de bucle (la cual es relativa a la dirección de comienzo y se calcula de la misma forma que los saltos relativos) y el número de veces que debe ejecutarse se introducen en la pila de bucles hardware, que tiene una profundidad de 4 posiciones. Cuando la dirección de retorno de la instrucción lanzada coincide con la dirección de retorno alojada en la cima de la pila, el contador de ejecuciones de bucle es decrementado, y cuando éste vale 0 es desapilado. Cuando no es igual a cero, el nuevo valor es guardado y el contador de programa se actualiza con la dirección de comienzo del bucle activo.

La instrucción de bucle hardware tiene 2 delay slots, por lo que la dirección de comienzo del bucle es el valor del contador de programa tras 2 ciclos de procesador.

El valor del contador de bucle varía de 1 a 4095 (12 bits) usando un valor inmediato y de 1 a 16777215 usando un contador de bucle calculado. El bucle siempre se ejecuta al menos una vez.

El mecanismo de bucles hardware tiene algunas limitaciones:

- las direcciones de retorno de todos los bucles presentes en la pila deben ser diferentes, porque si coinciden el compilador introduce instrucciones nop.

- las dos últimas instrucciones al final del bucle no deben ser instrucciones de salto para evitar paradas del pipe.

- las tres últimas instrucciones al final del bucle no deben ser instrucciones de salto condicional, salto relativo o instrucciones de retorno de subrutina, para evitar paradas del pipe.

Una de las características más interesantes de la estructura de gestión de los bucles hardware es que el valor del contador del bucle sólo se actualiza si el bucle no ha finalizado. Esto da una idea de la atención puesta en los detalles de diseño: cualquier ahorro es bienvenido.

2.2.9 Interrupciones

CoolFlux BSP soporta 3 interrupciones vectoriales y 4 interrupciones por software.

Cuando se produce una interrupción, el registro de estado *sr* se guarda en el registro *isr*, se deshabilitan las interrupciones y la dirección de retorno de interrupción se almacena en el registro *ilr*. Es entonces cuando el procesador salta a la dirección presente en la tabla de vectores de interrupción.

Las instrucciones en la tabla de vectores de interrupción deben ser *ireturn* (sin delay slots) o un salto incondicional a cierta dirección (sin saltos relativos).

Además de generar una interrupción hardware es posible atender interrupciones software. La instrucción toma un valor de 3 bits de anchura para indicar qué vector de interrupción debe ser invocado. Una interrupción software tarda 2 ciclos en ser atendida.

Dado que hay un solo registro *isr* y un solo registro *ilr*, y se produce la deshabilitación de las interrupciones al atender una de ellas, las interrupciones anidadas no son soportadas por el hardware.

Las interrupciones son habilitadas/deshabilitadas incidiendo sobre el campo *ie* del registro de estado:

sr.ie = ‘1’ | ‘0’

La habilitación de interrupciones (sr.ie = ‘1’) tarda 3 ciclos de procesador, sin delay slots, debido a que es posible que haya una interrupción pendiente que deba ser tratada tras esta instrucción, al activarse de nuevo las interrupciones, en lugar de la siguiente instrucción como era lo esperado. Para asegurar este comportamiento, el pipeline es vaciado tras la instrucción de habilitación de interrupciones. Por su parte, la deshabilitación tan solo toma 1 ciclo de procesador.

Una vez que la instrucción ha sido atendida y manejada, antes de producirse el retorno de interrupción es necesario reestablecer el registro de estado. La instrucción de retorno siempre tarda 3 ciclos, pero en algunos casos puede tener 2 delay slots factibles de rellenar con otras instrucciones.

2.2.10 Soporte para la integración en módulos multicore.

El procesador puede integrarse en un sistema de varios procesadores trabajando en paralelo. Pueden conectarse hasta 15 procesadores. Cada procesador tiene un número de índice propio, el cual es un pin de entrada dedicado. Es posible leer un número de identificación, el cual solo puede ser escrito en un registro AB. La parte baja del registro AB contendrá el index de un procesador determinado, mientras que la parte alta guardará la información correspondiente a la versión del procesador CoolFlux.

OPERATION TYPE: LONG MOVE
AB = dsp_id
EXAMPLES:
a0 = dsp_id;
b1 = dsp_id;

2.3 Enfoque software

A la hora de desarrollar software es importante tener en consideración las posibles optimizaciones que se puedan llevar a cabo y que deriven en un descenso del consumo de energía del procesador. Por ejemplo, la minimización del número de ciclos es esencial porque permite al procesador correr a una frecuencia menor, lo que a su vez permite un mayor escalado de la potencia. Asimismo, minimizar el uso de memoria reduce el tamaño necesario para la misma al tiempo que disminuye el número de ciclos de acceso a memoria, y ambas cosas derivan en un descenso del consumo de energía.

Por tanto, es fundamental tener en mente las posibles optimizaciones a realizar en el momento de desarrollar aplicaciones para CoolFlux. Estas optimizaciones se pueden clasificar en función de su dependencia con respecto a la arquitectura del procesador.

Optimizaciones independientes del procesador

Los accesos a memoria pueden reducirse explotando al máximo los datos que ya hayan sido leídos de memoria. Técnicas como la explotación de simetría de tablas, el proceso por bloques y el uso de buffers circulares responden a esta intención. Por supuesto, en este punto es necesario tener muy presente que la copia de datos de un buffer a otro debe ser reducida lo más posible, puesto que el tiempo consumido en ello es tiempo en el que no se procesan los datos, sino que tan solo se transportan.

En lo referente a la ejecución condicional, es preciso llevar este tipo de instrucciones al nivel más externo posible, sacándolas de los bucles internos y de las funciones, e incluso tratando de evitarlas.

La sobrecarga de llamadas a función también debe ser evitada en la medida de lo posible, y una función solamente será definida e invocada cuando realice una acción significativa. En esta misma línea, las funciones pequeñas es aconsejable programarlas inline.

Optimizaciones dependientes del procesador

El código debe estar estructurado con el fin de aprovechar al máximo las características arquitectónicas del procesador, aprovechando al máximo el paralelismo del datapath.

Para ello es importante tener en consideración el número de acumuladores y registros de puntero disponibles, evitando exigir su utilización simultánea en el código y sobre todo en interior de los bucles. Lo contrario derivaría en una sobrecarga de carga y almacenamiento de estos registros en memoria para satisfacer el algoritmo. Asimismo, el manejo de punteros aconseja utilizar patrones de acceso de 1, 2 o, en su defecto, constantes, para disminuir el número de ciclos necesarios.

Otra optimización a realizar es la de adaptar el código para indicar al compilador que dispone de funcionalidades que el procesador implementa mediante hardware específico.

El compilador interpreta directivas de compilación insertadas en el código de manera que puede optimizar el ensamblado. Por ejemplo, sabiendo que los datos pueden ser almacenados tanto en memoria X como en memoria Y, podemos implementar una función de proceso de datos para que tome los datos de una memoria y los escriba en otra, aprovechando así el paralelismo de acceso a los datos para reducir los ciclos de ejecución. Este almacenamiento se indica mediante la directiva ***chess_storage (memoria)***. El siguiente recuadro muestra ejemplos de almacenamiento en memoria:

```
int a[123];                              // este array se guardará en XMEM

fix chess_storage(YMEM) b[88];           // este array se guardará en YMEM

int chess_storage(IOMEM) c;              // la variable se guardará en IOMEM

int chess_storage(YMEM) * y_pointer;     // este puntero apuntará a YMEM
```

Mediante la directiva ***restrict*** podemos indicar al procesador que no hay dependencias de datos entre dos punteros diferentes ubicados en la misma memoria, y por tanto no tiene que protegerse ante la posibilidad de que existan dichas dependencias y esto redunda en menos ciclos de ejecución porque se accede a los datos de manera eficiente. El siguiente recuadro contiene un ejemplo de utilización de esta directiva.

```
int ScaleToPowerMedia(complex12 * s, complex12 * restrict scaleds, int size)
{...
complex12 ratio =  complex12(0.25, 0.0);
…
for (i=0;i<size;i++)
        scaleds[i] =  complex12( s[i] * ratio);
...
}
```

Otros ejemplos de directivas son las que permiten habilitar el software pipelining e indicar el número de bucles hardware anidados en una función, mostrados en el siguiente ejemplo:

```
int example(int * s, int chess_storage(YMEM) * scaleds, int size)
                        chess_loop_range(2, ) chess_prepare_for_pipelining
{
int ratio =  25;
for (i=0;i<size;i++)
        scaleds[i] =  (s[i]  *  ratio)<<3;
}
```

Para finalizar, cabe comentar que el compilador se ha revelado durante nuestra actividad con CoolFlux como un gran aliado, demostrando su alto nivel detectando estructuras en el código que podrían ser soportadas por el hardware del procesador. Esto facilita mucho la tarea de optimizar, porque la hace más transparente al programador.

2.4. Flujo de desarrollo de software

Ya hemos comentado que para CoolFlux se dispone de un conjunto de herramientas que permiten programar las aplicaciones en un LAN, concretamente en C.

El desarrollo de algoritmos para DSPs y para cualquier tipo de arquitectura, siempre según nuestra experiencia, suele abordarse en primera instancia, y por lo general, de un modo precipitado y difuso.

Sin embargo, una dosis de experiencia y un poco de sensatez aconsejan abordar esta tarea partiendo de un enfoque más cuidadoso. En primer lugar ha de priorizarse la corrección del algoritmo en términos teóricos, porque de nada sirve adaptar para cierta arquitectura un algoritmo incorrecto. En segundo lugar, a la hora de chequear la funcionalidad del algoritmo y corregirla en caso de ser necesario, es más fácil y sobre todo más rápido trabajar en un entorno [C++ en Windows o Linux] que sobre las herramientas de simulación del DSP proporcionadas por TCT, las cuales están orientadas a la optimización.

Es por ello que existen librerías específicas de CoolFlux que soportan tipos de datos, operaciones y funciones intrínsecas del sistema (como por ejemplo funcionalidades soportadas con hardware específico). Al incluir esta librería en el proyecto de C, se hace posible compilar código desarrollado para CoolFlux con cualquier compilador de C++ sobre Windows y Linux. Esto es lo que se conoce como *compilación nativa*.

Teniendo en cuenta todo esto, el flujo de diseño de algoritmos para CoolFlux sigue unas etapas que relatamos a continuación.

La primera tarea a abordar a la hora de desarrollar la implementación de cada algoritmo planteado es la de obtener una versión punto fijo en código C. Tras compilar y ejecutar en plataforma nativa y haber comprobado su correcta funcionalidad, podemos estar seguros de contar con un algoritmo correcto, pero obviamente no se dispone aún de una implementación exacta a nivel de bit de como la aplicación se ejecutara en CoolFlux BSP, dado que la longitud de las palabras de memoria puede no ser la misma y puede no tener en cuenta cuestiones tan importantes para CoolFlux como el truncamiento, la saturación, etc. a la hora de producir resultados.

Para obtener una simulación precisa, el primer paso es adaptar dicha implementación al entorno de aplicación de CoolFlux utilizando, para re-programar el algoritmo, los tipos de datos propios del DSP, así como las operaciones soportadas por el mismo y la sustitución de memoria dinámica por memoria estática.

En este punto es recomendable y hasta necesario el uso de sentencias condicionales de compilación para mantener sobre el mismo proyecto, y de manera paralela, las versiones destinadas a compilación nativa y la destinada a correr sobre CoolFlux BSP. Sobre esta última ya es posible aplicar optimizaciones de código para CoolFlux, como puede ser, por ejemplo, la utilización de directivas de almacenamiento específico de variables o buffers sobre la memoria Y.

Llegados a este punto se hacen test para comparar los resultados de ambas versiones (la de punto fijo en C y la de código soportado por CoolFlux) y se establece un criterio de aceptación del algoritmo para CoolFlux BSP, que en caso de no cumplirse obliga a reestructurar el código en la versión enfocada al DSP.

La compilación nativa nos aporta rapidez y eficiencia en el desarrollo, además de verosimilitud a nivel de bit. En cambio, no nos proporciona información sobre el desempeño del algoritmo sobre CoolFlux DSP, por lo que no tenemos información ni de los ciclos consumidos ni de la memoria necesaria para ejecutar el algoritmo.

Con esto se llega a la última parte del proceso de desarrollo, que consiste en ejecutar el algoritmo con las herramientas de simulación proporcionadas por TCT (Checkers). Una vez que hemos comprobado que los resultados obtenidos mediante esta simulación son idénticos a los obtenidos en ejecución nativa, podemos analizar la información del resultado final de la ejecución sobre la arquitectura objetivo que el simulador nos proporciona, con lo que es posible evaluar las prestaciones del algoritmo y tomar decisiones de diseño o de desarrollo en consecuencia.

Para profundizar en esta materia, ver [10].

Capítulo 3. Detalles de la implementación

La implementación del sistema de recepción de DAB se abordó según un patrón modular, dado que se observó por parte de la compañía que el receptor se podía organizar en una serie de módulos independientes. En las ilustraciones 16,17 y 18, se puede observar la conexión de dichos módulos y el orden en los que los vamos a ir estudiando.

En paralelo con este tipo de desarrollo, se mantuvo una aplicación integral en la que se iban acoplando los módulos según se iba terminando su diseño. A continuación explicaremos la forma en la que este trabajo fue organizado y llevado a cabo.

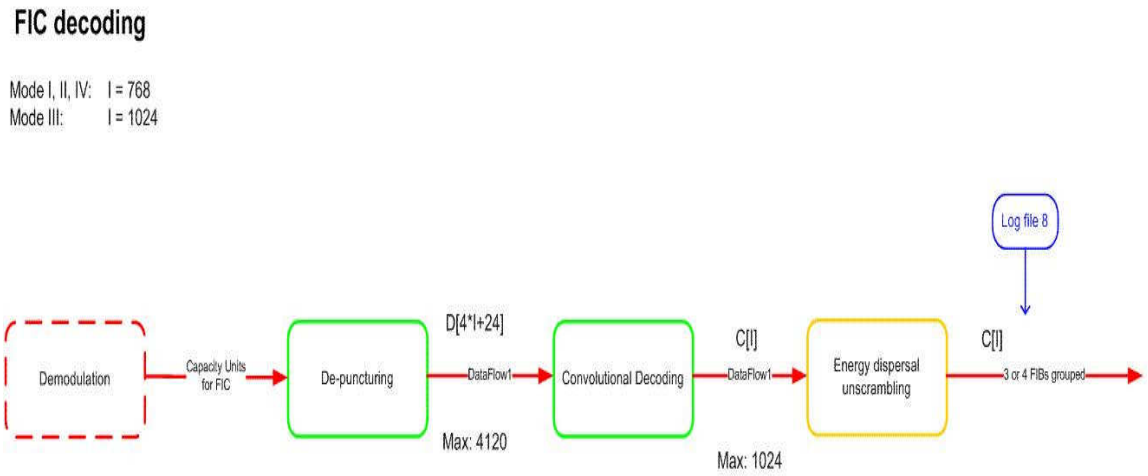
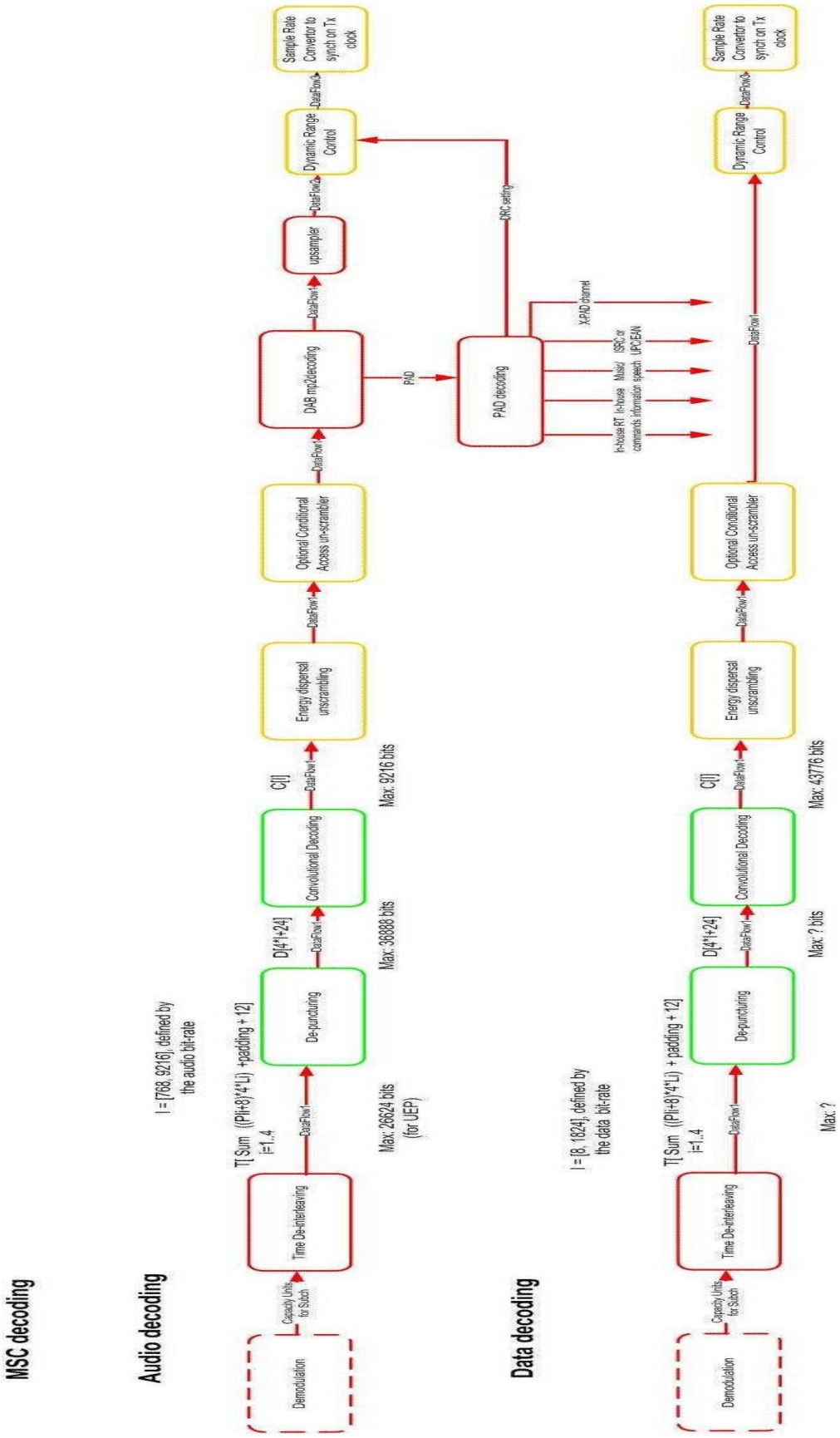
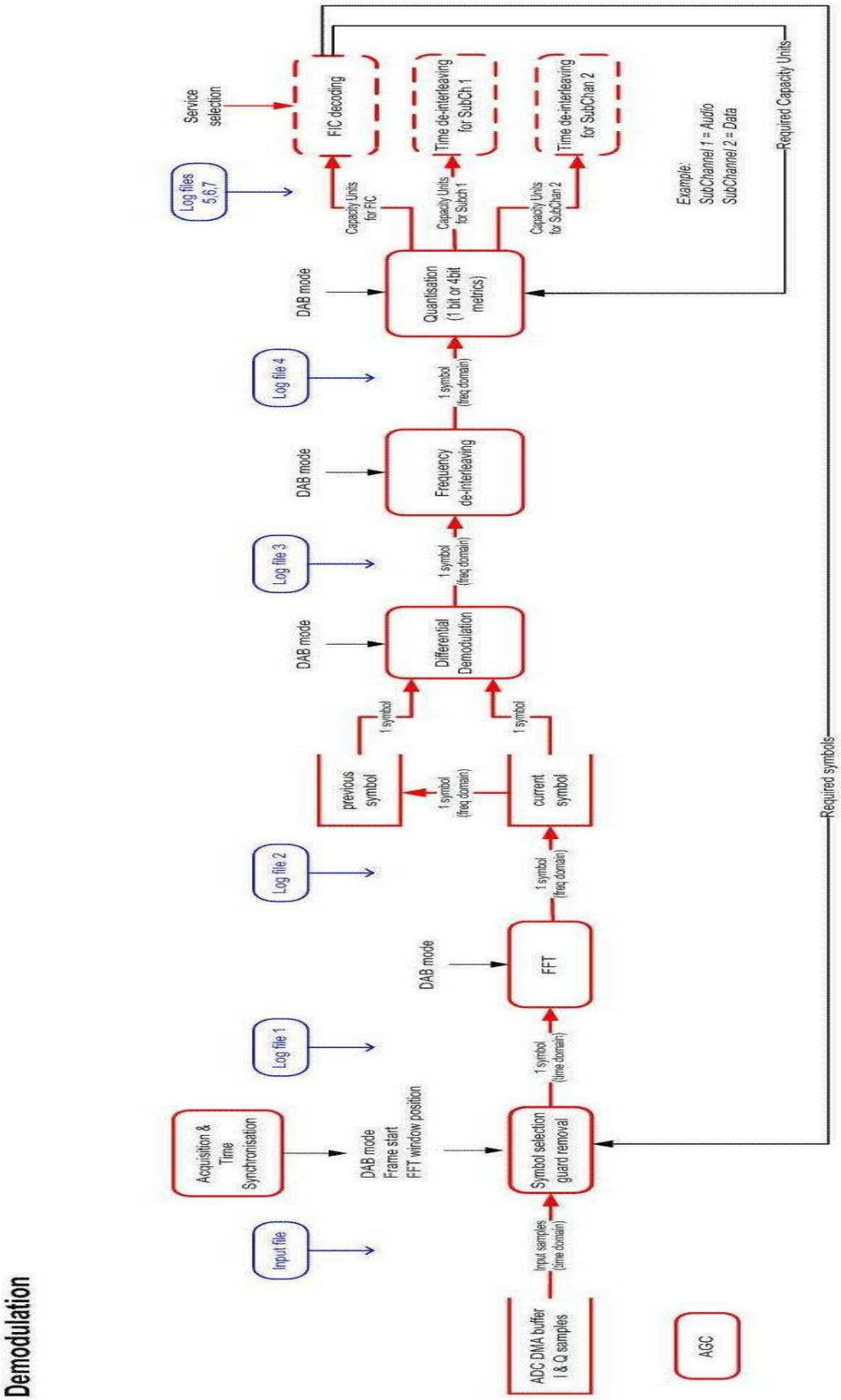


Ilustración 16: División en módulos del FIC decoding.


MSC decoding
Audio decoding
Data decoding
Demodulation
Capacity Units for Subch
Time De-interleaving
DataFlow
TI[Sum (I*Ni+8)*4*Li) +padding + 12]
i=1..4
I = [768, 9216] defined by the audio bit-rate
Max: 26624 bits (for UEP)
De-puncturing
D[4*I+24]
Max: 36888 bits
Convolutional Decoding
C[I]
Max: 9216 bits
Energy dispersal unscrambling
Optional Conditional Access un-scrambler
DAB mp2decoding
PAD
PAD decoding
CRC setting
X-PAD channel
ISRC or UPC/EAN
Music
In-house
In-house RT comments, information, speech
upsampler
Dynamic Range Control
Sample Rate Conviertor to synch on Tx clock
I = [8, 1824] defined by the data bit-rate
Max: ?
Max: 7 bits
Max: 43776 bits


Demodulation
Input file
Input samples (time domain)
ADC DMA buffer I & Q samples
AGC
Acquisition & Time Synchronisation
DAB mode
Frame start
FFT window position
Log file 1
Symbol selection guard removal
1 symbol (time domain)
DAB mode
FFT
Log file 2
1 symbol (freq domain)
current symbol
1 symbol
previous symbol
1 symbol
1 symbol (freq domain)
DAB mode
Differential Demodulation
Log file 3
1 symbol (freq domain)
DAB mode
Frequency de-interleaving
Log file 4
1 symbol (freq domain)
DAB mode
Quantisation (1 bit or 4bit metrics)
Log files 5,6,7
Service selection
FIC decoding
Capacity Units for FIC
Time de-interleaving for SubCh 1
Capacity Units for SubCh 1
Time de-interleaving for SubChan 2
Capacity Units for SubChan 2
Example:
SubChannel 1 = Audio
SubChannel 2 = Data
Required Capacity Units
Required symbols

3.1 Estudio general

Lo primero que se observó al abordar el diseño de los módulos de recepción de DAB es que dichos módulos necesitan cierta información para gestionar adecuadamente las señales que les llegan. Visto que mucha de esta información es compartida por varios de los módulos, en las primeras etapas de planificación se decidió incluir toda esta información es una estructura de ámbito global, de la cual hubiera una sola copia en memoria y fuera visible para todos los módulos. Esta estructura además contendría otros campos o estructuras para almacenar la información extraída del FIC necesaria para el procesamiento del MSC. La primera aproximación realizada en lo que a requerimientos de esta estructura se refiere no fue demasiado satisfactoria, por lo que se decidió dotarla de una serie de requisitos mínimos observados en primera instancia y después seguir un diseño incremental: cada módulo añadiría a la estructura lo que le fuera necesario, mezclando todo en la versión integrada del sistema. Al final, dentro de esta estructura figuraban las variables de entorno necesarias, los buffers de datos necesarios para el sistema y también las estructuras que soportan información decodificada del FIC.

Un aspecto fundamental a tratar es la forma en la que se unen todos los módulos, es decir, la que guía el comportamiento del sistema. Es necesario decidir cuándo y cómo actuará cada módulo. Para ello se diseñó un programa principal encargado de activar y mantener la recepción. Varias funciones estructurales atienden a este programa, las cuales describimos a continuación.

La primera función del programa principal es la recepción de muestras, llevada a cabo mediante una función llamada processSamples, la cual toma como argumento un buffer de muestras y un cierto tamaño de buffer y copia dichas muestras en un buffer de entrada al sistema. Dicho buffer en realidad consiste en un buffer circular, de anchura doble al tamaño de la ventana FFT (nº de muestras por cada símbolo OFDM recibido). El motivo de la inclusión de este tipo de buffer es el propio diseño del sistema. Si disponemos de k muestras ya recibidas (k<tamaño ventana FFT) y recibimos n muestras más, hay que tener en cuenta que es posible que k + n sea mayor que el tamaño de la ventana FFT y por tanto el número de muestras del sistema exceda lo necesario para lanzar el procesamiento de un símbolo OFDM. Para dotar a la aplicación de robustez ante esta eventualidad, la función processSamples escribe las muestras recibidas en el buffer circular, actualiza el puntero de escritura y después comprueba si dicho puntero de escritura ha pasado la mitad del buffer doble o por el contrario ha vuelto al principio del buffer. Si se da alguna de esas dos circunstancias se escalan los datos recibidos mediante la función de control automático de ganancia (AGC), y tras esto se invoca la función processSymbol (), encargada de procesar un símbolo OFDM.

La función processSymbol se encarga de gestionar el orden de las llamadas a los módulos que forman el canal de demodulación, los cuales serán explicados a continuación. Hay que destacar que con la llamada al último de éstos módulos, quantization, finaliza la función de gestión en processSymbol, puesto que a partir de aquí dicho módulo toma el control de las operaciones.

Otra importante aportación a nivel global para conseguir un correcto comportamiento del sistema fue la elaboración de una estructura y unas funciones que

implementasen el uso de memoria dinámica en el procesador. Esto fue debido a la imposibilidad de utilizar dicho tipo de utilización de memoria en el DSP y a la necesidad de reservar espacio variable para los buffers del Subcanal, cuyo tamaño no es fijo. Dicho sistema básico de memoria dinámica consistía en la declaración de dos punteros: uno a la posición inicial de la memoria que se iba a utilizar de memoria dinámica, y otro para fijar la última posición accesible, declarados a nivel global y externo a la aplicación de recepción. A su vez, se crearon funciones para poder reservar espacios de memoria dinámica (aunque de manera muy rudimentaria, sin políticas de ubicación de bloques), y funciones que realizasen estas reservas alineadas a un entero pasado por parámetro. Las funciones de liberación de memoria no se implementaron por lo que no se podían reutilizar en ningún caso posiciones previamente reservadas durante la ejecución. Un punto importante fue la necesidad de ampliar el número de bits utilizados para direccionar posiciones en la memoria, pues con los que teníamos en ese momento, no se permitía reservar espacio para tamaños grandes. Gracias a esta mejora, ampliando los registros a 23 bits, actualmente se pueden direccionar 8388608 palabras para ser reservadas en el Heap.

3.2 Estudio Modular

En este apartado se presentan los diferentes módulos realizados para su integración en el receptor DAB.

Antes de comenzar con dicha presentación, es oportuno comentar que uno de los módulos del sistema no fue diseñado por ninguno de los dos estudiantes firmantes del proyecto. Dicho módulo, el módulo encargado de realidad la FFT, fue desarrollado y optimizado con anterioridad por personal de la compañía, por lo que su implementación no era necesaria en ningún caso.

Sin embargo, a pesar de no haber tenido que diseñarlo ni implementarlo, fue necesario familiarizarse con su funcionamiento y con su razón de ser. Sin entrar a describirlo en profundidad, comentaremos que el módulo FFT implementa la transformada rápida de Fourier, caso particular de la Transformada Discreta de Fourier en el que la cardinalidad de los datos a tratar es de un valor potencia de 2, para el cual existen algoritmos muy rápidos y eficientes (de ahí el nombre, Fast Fourier Transform). La función de este módulo en el DAB es modificar la señal de manera que los datos transportados en un símbolo OFDM en el dominio del tiempo pasen a estar en el dominio de la frecuencia. [Completable → Ortogonalidad de las portadoras da lugar a poder utilizar FFT.]

Como ya hemos comentado, en NXP existía una versión muy optimizada que consumía 23 millones de ciclos de procesador por segundo. Debido a la existencia de dicha versión, ya testeada y validada, el estudio del rendimiento de la FFT que usáramos en el sistema de recepción de DAB no era un objetivo. Por esta razón se utilizó una versión de la FFT previa a la optimización que dio lugar a la última versión antes comentada.

3.2.1 Automatic Gain Control

3.2.1.1 Introducción

En el procesamiento de la señal es posible que al operar con ella los resultados excedan los límites de representación del procesador y por tanto se pierda información por el camino. Este módulo define el procedimiento para escalar la señal en las portadoras de un símbolo OFDM con el fin de mantenerla en un rango dinámico apropiado para CoolFlux a través del canal de demodulación. Esto se lleva a cabo multiplicando cada portadora presente en un símbolo OFDM por un cierto factor. Para calcular este factor se diseñaron y aplicaron diversos procedimientos, todos ellos basados en la siguiente descripción del escalado a realizar:

Símbolo OFDM = (X_i , i = 0..N – 1)

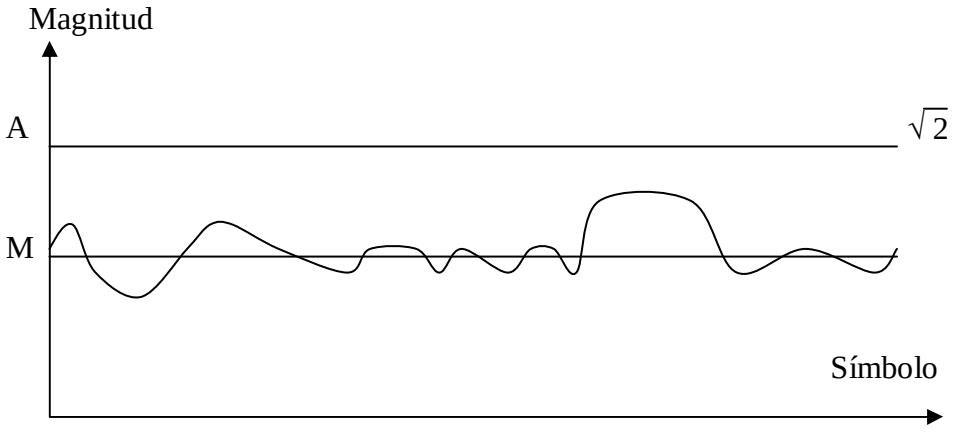
Procedimiento de escalado de la portadora i = $\mathbf{X_i * A / M}$,

donde

A = $\mathbf{\Sigma_i}$ **(** $\sqrt{2}$ **) / N** para i = 0..N – 1, $\sqrt{2}$ el valor máximo representable y esperado para la magnitud de la portadora.

M = $\mathbf{\Sigma_i}$ **(Magnitud$_i$) / N** para i = 0..N – 1, la energía del símbolo OFDM.

N = tamaño de la ventana FFT.



El primer paso es calcular la energía media del símbolo y compararla con el máximo valor energético representable en CoolFlux, previamente obtenido para un tamaño determinado de la ventana FFT (esto es, el tamaño del símbolo OFDM para el modo de recepción DAB). En función del resultado de dicha comparación se realizan dos procedimientos diferentes pero ambos destinados a calcular el factor con el cual multiplicar cada portadora del símbolo con el fin de mantener el valor de dichas portadoras dentro del rango dinámico deseado.

3.2.1.2 Desarrollo de la implementación

Para este módulo se desarrollaron tres funciones diferentes, las cuales son versiones del mismo algoritmo de escalado. La diferencia entre estas tres funciones estriba en el modo de calcular la energía del símbolo OFDM. La primera de todas, una función meramente descriptiva del algoritmo, se realizó con el fin de comprobar la corrección del algoritmo y la utilidad del mismo y por consiguiente, según lo explicado en XXX, se desarrolló únicamente una versión para la plataforma WIN32, es decir, para compilación nativa. Esta función calcula la energía del símbolo utilizando el algoritmo clásico de obtención de la magnitud de un par IQ: $(Re^2 + Im^2)^{1/2}$. La segunda función implementada usa el algoritmo CORDIC para calcular la magnitud de cada par IQ. La tercera utiliza una aproximación, max(|Re|,|Im|) + min(|Re|,|Im|)/2, para calcular dicha magnitud del símbolo. En este punto es menester comentar que se estudió la posibilidad de aproximar la magnitud utilizando polinomios de Taylor, pero tal punto fue descartado rápidamente por la complejidad de la función obtenida o, mejor dicho, sus requerimientos en lo que a ciclos de procesador se refiere.

Las entradas del módulo son el símbolo OFDM a escalar, un buffer de complex12 de tamaño dinámicamente fijado al tamaño de la ventana FFT, así como el propio tamaño de la ventana FFT. Además, un parámetro que almacena el valor de la máxima magnitud representable en CoolFlux. La salida es asimismo un buffer de complex12 del mismo tamaño que el de entrada.

Tras evaluar las prestaciones de cada una de las funciones, se observó que la que mejor rendimiento y más fiabilidad ofrecía era la versión que utiliza el algoritmo de CORDIC. Esto es así porque en CoolFlux existe hardware específico que realiza cada etapa CORDIC en 1 ciclo de procesador, por lo que se decidió aprovechar esta circunstancia.

En la línea de la descripción anterior, describiremos el procedimiento de escalado seguido en la implementación de esta función con CORDIC:

Escalado: $\mathbf{X_i * A' / M'}$, donde:

> $\mathbf{A' = \Sigma_i (\sqrt{2}) / 4}$ para i = 0..N – 1, $\sqrt{2}$ el máximo valor esperado para Magnitud (dividido por 4 por razones de seguridad, con el fin de evitar que el resultado exceda el máximo representable se salvan 2 bits de precisión).
>
> $\mathbf{M' = \Sigma_i (Magnitud)}$ para i = 0..N – 1, la energía del símbolo OFDM
>
> **N =** size of the FFT window.
>
> dado que

$$\mathbf{X_i * A / M = \frac{X_i * (\Sigma_i (\sqrt{2}) * N}{4 * N} \quad \frac{}{\Sigma_i (Magnitud)} = X_i * A' / M'}$$

Dado que para la correcta realización de este algoritmo ha sido necesario hilar muy fino en lo que al manejo de magnitudes en CoolFlux se refiere, pasamos a describirlo con un poco más de detalle.

El símbolo de entrada es almacenado en el buffer 's'. En primer lugar, mediante un bucle se calcula la acumulación de las magnitudes de cada portadora (cada valor complex12) del símbolo OFDM, el cual se almacena en un buffer complex12 y es pasado como argumento a la función que realiza la primera etapa CORDIC. El resultado del algoritmo de CORDIC de 4 etapas (se hicieron pruebas y con este número de etapas se obtuvo precisión suficiente) se almacena en una variable auxiliar, cordicAux.

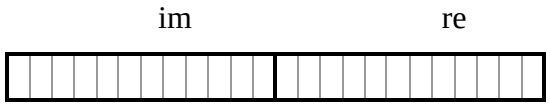


Ilustración 19: Estructura complex12

→ ETAPAS DEL ALGORITMO DE CORDIC ←

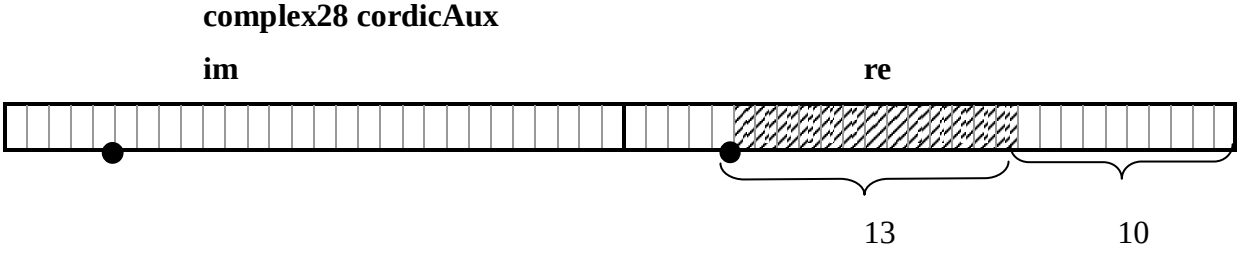


Ilustración 20: Estructura complex28.

El resultado de las etapas CORDIC, esto es, la magnitud del par IQ (del valor complex12 tratado en esta vuelta del bucle), representa la energía transportada en cada portadora. Como resultado del algoritmo de CORDIC nos interesa solamente los 13 bits más significativos de la parte real de la variable complex28 (48 bits, CORDIC devuelve su resultado en este formato), por lo que es necesario realizar un desplazamiento de 10 bits a la derecha para obtenerlo. Dos bits más se añaden a este desplazamiento por motivos de seguridad, para evitar que el resultado sea mayor que el máximo representable: se salvan dos bits de precisión (tal y como se hizo en el cálculo de A').

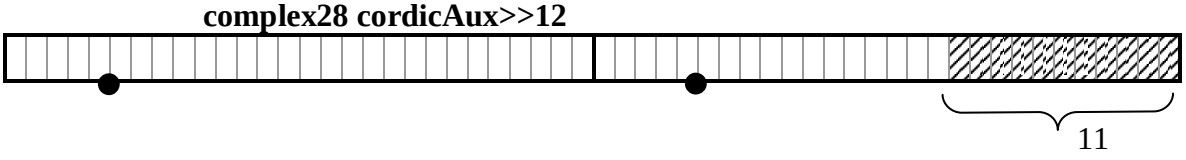


Ilustración 21: Complex28 Shift a la derecha 12.

Los resultados de las etapas CORDIC para cada portadora son acumulados en una variable acc(56 bits – para el acumulador), energyAcc. Los 11 bits de resultado almacenados en la variable cordicAux en cada iteración del bucle se convierten, en el peor de los casos, en 22 bits de resultado acumulados en acc al final del proceso de cálculo de la energía del símbolo. Esto es así dado que el tamaño máximo de un símbolo OFDM es 2048 valores complex12, y por tanto el máximo número de sumas a realizar, por lo tanto necesitamos 11 bits más.

Esta variable energyAcc se convierte a formato lfix (48 bits) con el fin de poder realizar la división, la cual no es soportada en CoolFlux para valores complex, simd o acc. Antes de realizar esta conversión es necesario desplazar 16 posiciones a la derecha, porque las variables de tipo acc tienen 8 bits de overflow y 48 para un valor con signo, y la división solo está soportada para valores de 32 bits con signo.

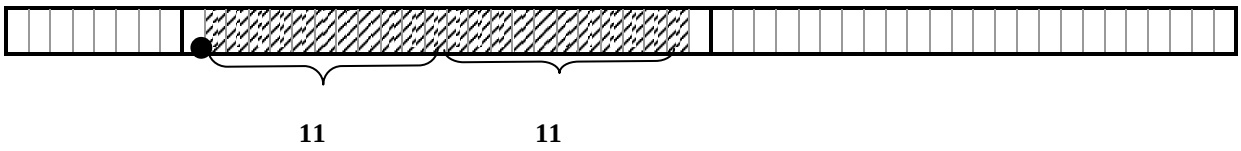


Ilustración 22: Energy Acc.

Ilustración 23: Lfix media.

Una vez calculada la energía del símbolo, es almacenada en la variable media, en la que se hace una comprobación condicional para discriminar el procedimiento de cálculo del factor de escalado a seguir: A' > M':

1. MaxMedia > Media

En este caso es necesario calcular el número de bits de diferencia entre los dos bits más significativos de ambos valores, llamado shift, ya que si MaxMedia es mayor que Media esto significa que el resultado de su división será mayor que 1, lo cual no puede ser representado en CoolFlux para el tipo de datos lfix. Además, resulta interesante trabajar con ambos valores ajustados a 32 bits de precisión para realizar la división, por lo que tenemos que desplazar media 'shift' posiciones:

Ilustración 24: MaxMedia

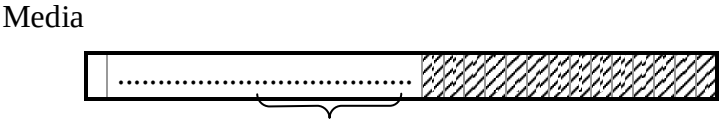
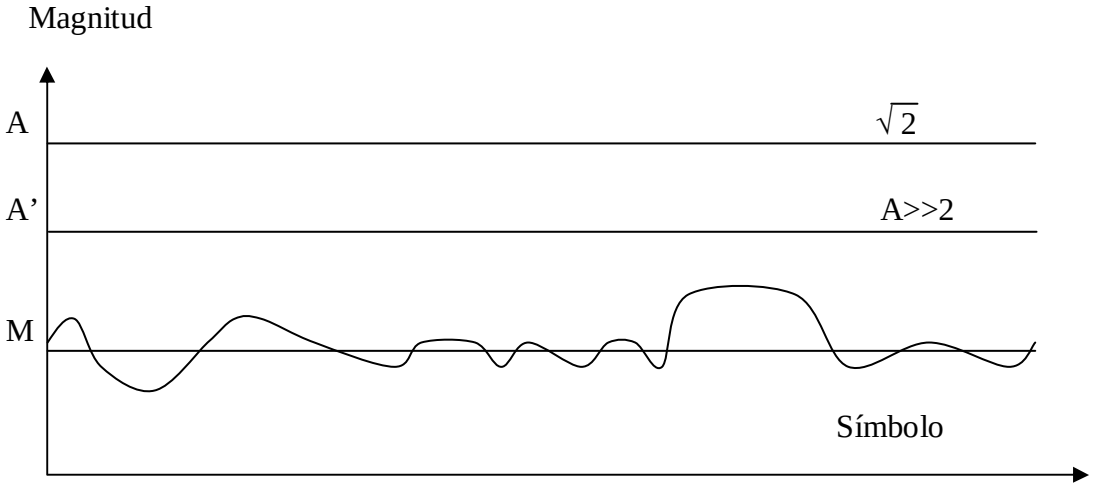


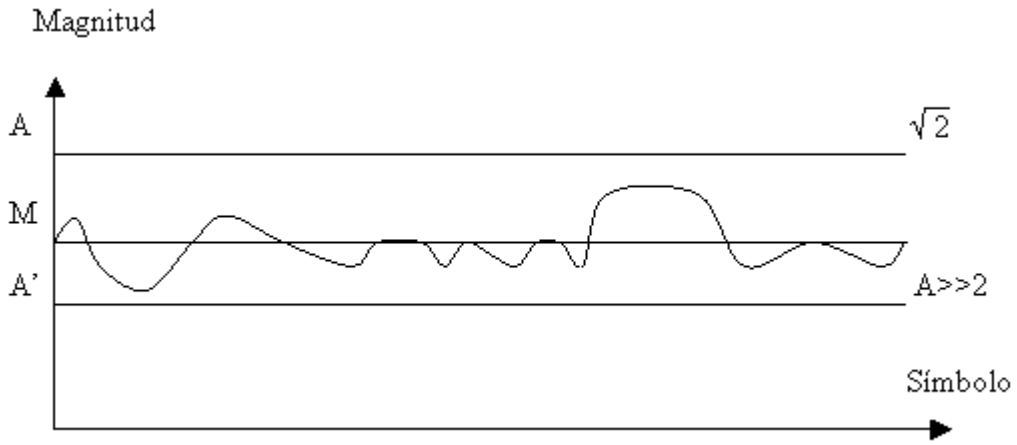
Ilustración 25: Media.



El factor de escalado es el resultado de la división de ambos valores. Tras realizar el escalado, es decir, la multiplicación de cada portadora por el factor, es necesario compensar el desplazamiento realizado anteriormente, por lo que se debe hacer un desplazamiento en sentido inverso a cada portadora.

2. MaxMedia <= Media

En este caso el resultado no será mayor que 1, por lo que no hay que hacer desplazamientos: tan solo calcular el factor de escalado mediante la división y multiplicarlo a cada portadora.



Una cosa que aún no se ha comentado es el factor de corrección que es necesario multiplicar después de las etapas CORDIC para cada portadora. Este factor se calcula siguiendo una función definida en el artículo "A survey of CORDIC algorithms for FPGA based computers¨, de Ray Andraka. Con el fin de determinar el número ideal de etapas CORDIC a utilizar se desarrollo una función que calculara dicho factor de corrección en función del número de etapas.

3.2.1.3 Mejoras realizadas

No se han llevado a cabo mejoras específicas para este módulo, quedando pendiente para futuros desarrollos de la aplicación.

3.2.1.4 Rendimiento parcial del módulo y pruebas

La siguiente tabla muestra los resultados obtenidos al ejecutar el programa en el simulador de CoolFlux.

Tabla 3: Rendimiento Power Media

Función	Ciclos por llamada	Palabras en PMEM	Palabras en XMEM
Scale_To_Power_Media	22567 para el modo DAB II	67	11 + Ventana_FFT * 2

3.2.2 Modulo Demodulador diferencial

3.2.2.1 Introducción

La información en la modulación OFDM es transmitida mediante un número determinado de portadoras ortogonales, cada una de ellas modulada utilizando D-QPSK (Differential Quadrature Phase Shift Keying). Esta codificación utiliza cambios en la fase de dos símbolos consecutivos para transportar la información, concretamente dos bits por símbolo DQPSK (π/4-shift D-QPSK).

Este modulo lleva a cabo la demodulación diferencial de un símbolo OFDM. Esto significa que calcula la diferencia de fase entre los símbolos DQPSK de cada portadora en los dos últimos símbolos OFDM recibidos: el anterior y el actual. Esta información obtenida será utilizada para obtener los bits codificados en la transmisión en un módulo posterior. Es importante remarcar que por cada portadora de un símbolo OFDM se transmite un dibit, de manera que el número de bits totales transmitidos en un símbolo OFDM es el doble del número de portadoras.

Para la correcta implementación de este sistema de modulación se introdujo en la trama bit el ya comentado PRS (Phase Reference Symbol) cuya misión es ser utilizado junto al primer símbolo OFDM del FIC como referencia para el cálculo de la diferencia de fase inicial, correspondiente a dicho primer símbolo del FIC.

3.2.2.2 Desarrollo de la implementación

Dado que cada portadora transporta dos bits, en el estándar de DAB se especifica que las portadoras serán representadas utilizando números complejos. Para nuestra implementación,[, como ya se ha comentado,] un símbolo OFDM consiste en un buffer de elementos del tipo complex12 de cardinalidad variable en función del modo DAB en el que estemos recibiendo, y por tanto de tamaño dinámicamente establecido.

Las entradas del módulo son el último símbolo OFDM recibido y el actual, mientras que la salida es también un buffer de complex12 que contiene la diferencia de fase entre estos dos símbolos. Además, es necesario almacenar el símbolo OFDM actual para utilizarlo en el procesamiento del siguiente símbolo.

El cálculo de la diferencia de fase se realiza multiplicando el símbolo actual por el conjugado del símbolo anterior, siguiendo la fórmula siguiente:

k-ésimo símbolo DQPSK del símbolo OFMD I-1 => $Z_{I-1,k}$
k-ésimo símbolo DQPSK del símbolo OFMD I => $Z_{I,k}$
k-ésimo símbolo QPSK del símbolo OFMD I => $Y_{I,k}$

$$Y_{I,k} = Z_{I,k} \times \text{conjugado}(Z_{I-1,k})$$

Para el cálculo del conjugado existe una función específica en las librerías de CoolFlux, y el procesador soporta la multiplicación de datos del tipo complex12 en un solo ciclo.

3.2.2.3 Mejoras realizadas

Para optimizar esta función se decidió almacenar el resultado en memoria Y, y los operandos en memoria X, para permitir la escritura en memoria de manera concurrente con la lectura de operandos para la siguiente operación. Además, se llevo a cabo una estrategia de loop-unrolling manual así como una inclusión de la directiva chess_prepare_for_pipelining la cual busca en el código la posibilidad de hacer software pipelining.

3.2.2.4 Rendimiento parcial del módulo y pruebas

La siguiente tabla muestra los resultados obtenidos al ejecutar el programa en el simulador de CoolFlux. Como puede observarse, es claro que la versión que utiliza YMEM es más eficiente en lo que a ciclos de procesador se refiere.

Tabla 4: Rendimiento Demodulador Diferencial.

Función	Ciclos por llamada	Palabras en PMEM	Palabras en XMEM	Palabras en YMEM
Demodulador diferencial XMEM	8 + [(nº portadoras – 1) * 3]	11	4 + Tamaño ventana FFT *3	0
Demodulador diferencial YMEM	8 + [(nº portadoras / 2) * 5]	13	4 + Tamaño ventana FFT *2	Tamaño ventana FFT

3.2.3 Frecuency de-interleaving

3.2.3.1 Introducción

Se trata de un módulo diseñado para evitar errores de transmisión en el dominio de la frecuencia debidos a multicaminos en la transmisión, interferencias de portadoras, etc. Lo que trata es de reordenar las portadoras en el receptor, ya que se realizó una ordenación previa de ellas en el transmisor para aleatorizar el burst de error. Lo que se consigue con el Frequency Interleaving al separar las portadoras es que como a su llegada al receptor los errores están contiguos, si se realiza el Frequency de-interleaving, estos errores se separan. En la ilustración 26 se puede comprobar cómo a la izquierda se ha realizado el de-interleaving, haciendo que los errores que antes estaban contiguos en la recepción, ahora estén separados.

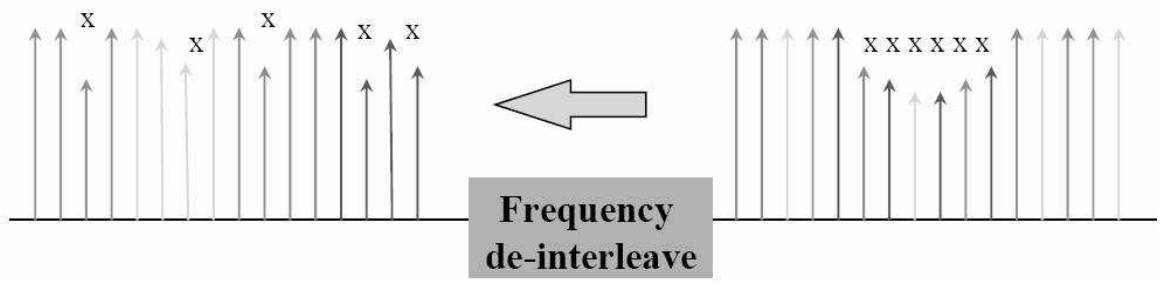


Ilustración 26: Frequency de-Interleaving.

Por supuesto, esta reordenación de las portadoras no es aleatoria sino que se basa en unas tablas especificadas en el estándar de DAB, mediante las cuales se pueden dispersar los errores. Este módulo trabaja en el dominio de la frecuencia como bien hemos dicho por lo que recibe datos de tipo complejo y, al no realizar ninguna conversión de datos, devuelve el mismo tipo. El llamamiento a este código se realiza después de la demodulación diferencial y antes de la cuantización como se puede observar en la ilustración 27.

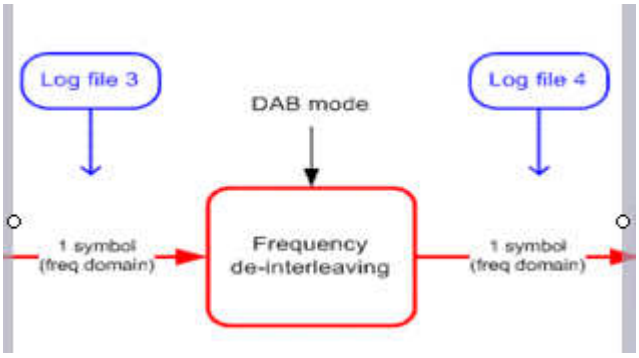


Ilustración 27: Módulo Frequency de-Interleaving.

3.2.3.2 Desarrollo de la implementación

En el desarrollo de este módulo hubo que tener en cuenta que dependerá del tipo de DAB que se esté tratando ya que dependiendo de ello, habrá más o menos portadoras. Por ello, hubo que implementar cuatro tablas de reordenación de portadoras. La lectura de estas tablas fue implementada de dos maneras atendiendo al compromiso de reducir el tiempo de ejecución sin olvidar el consumo de memoria. Estos dos métodos son: lectura de las tablas desde un archivo de texto externo, donde se reduce el consumo de memoria pero aumenta el tiempo de ejecución y, integradas en la memoria ROM del procesador, donde se reduce el tiempo de memoria pero se aumenta el tiempo de ejecución. Se permite la decisión del usuario a elegir uno de los dos métodos para valorar cual es el más conveniente para su caso. Para nuestro proyecto, puesto que la lectura de estas tablas se realiza una vez y ya deben quedar en memoria, pareció más útil la integración de dichas tablas en la memoria ROM.

Si continuamos hablando de esta reordenación de portadoras, es importante resaltar que una parte importante de este módulo es el reordenarlas de tal modo que sólo son válidas las situadas en el centro del símbolo, de tal manera que así se garantizaba la ausencia de interferencia entre símbolos. Este no es el objetivo principal del módulo, pero si es muy importante para la recepción ya que los demás módulos ignoraran los extremos de los símbolos.

En el código de este módulo, se puede observar como también se implementó la función de Frequency Interleaving. Esto fue posible gracias a que la complejidad no radicaba en la propia función sino en el manejo de las tablas, y así nos permitió realizar las pruebas de funcionamiento de manera más sencilla. Gracias a esta función conseguimos simular una ordenación previa la transmisión de la trama y su posterior reordenación en la función que realmente nos importaba el Frequency De-Interleaving.

Los datos utilizados para la entrada y la salida, como ya hemos comentado, eran de tipo complejo, y se utilizó el tipo propio de Coolflux *Complex12*. Este tipo de datos fue suficiente para nuestra configuración ya que su tamaño nos permitía mantener en una misma variable tanto la parte real como la imaginaria. Sin embargo, las tablas no era de tipo Complex12, sino de tipo entero que representaba los índices de las portadoras en el símbolo; de este modo la función leía los índices de las tablas y reordenaba los elementos (de tipo *Complex12*) en base a dichos índices.

3.2.3.3 Correcciones

Tras realizar toda la implementación y comprobar su correcto funcionamiento bajo las dos plataformas posibles (*Microsoft Visual Studio* y *Coolflux*), comenzó la optimización del rendimiento. Para poder mejorar el rendimiento hubo que configurar el sistema bajo todas las posibilidades de alojamiento en memoria de los datos y realizar un estudio para ver cuál era la configuración óptima. De esta manera, se descubrió que existía dos configuraciones que podían implantarse sin diferencia en cuanto a ciclos de ejecución pero, como se ha insistido a lo largo de este documento, la decisión de dónde almacenar los datos no era independiente de éste módulo, había que consultar las

configuraciones óptimas del anterior y el siguiente. Resultó que el módulo que se sitúa a continuación de éste, necesitaba que los datos de entrada fuesen alojados en memoria X, por lo tanto, esto coincidía con una de nuestras configuraciones óptimas: datos de entrada en memoria Y, datos de salida en memoria X, y las tablas en memoria Y. Además, hubo la suerte de que la mejor configuración para el módulo predecesor, era que sus datos de salida se alojasen en la memoria Y.

3.2.3.4 Rendimiento parcial del módulo y pruebas

Una vez elegida la configuración final del alojamiento de datos de la función, se establecieron una serie de filtros y testeadores para realizar las pruebas. Estos filtros tratan de leer los datos de entrada como enteros, y los transforma a complejos para poder ser enviados al módulo. El filtro de salida, transforma el complejo en un double para poder ser testeada la pérdida de precisión. Es claro que en este caso no puede haber pérdida de precisión pues no se hacen conversiones de tipos de datos en la función, pero para confirmarlo se implementó un programa de test para complex12, este programa comprobaba los datos de entrada con su correspondiente salida, y confirmaba que no se había perdido ningún decimal en la ejecución.

Además de estas pruebas, se realizaron test bajo Coolflux para comprobar el rendimiento en ciclos y memoria del módulo. El resultado de este estudio queda reflejado en las siguientes tablas:

Tabla 5: Rendimiento Frequency de-Interleaving por módulos.

Modo de DAB	Número de portadoras	Número de ciclos por llamada
Modo I	2048	6157
Modo II	512	1549
Modo III	256	781
Modo IV	1024	3085

En esta primera tabla se puede observar el incremento del número de ciclos, utilizados para la ejecución, al aumentar el número de portadoras a tratar. A continuación se muestra el consumo de memoria

Tabla 6: Rendimiento de memoria Frequency de-Interleaving.

		Palabras (3 bytes)	
XMEM	Stack	1024	Máximo consumo + 3 words
	Statics	13072	Definidas en cada instancia
	Literals	0	Constantes del programa
	Tables	0	Tablas en memoria, definidas una vez
YMEM	Statics	10000	Variables estáticas, buffers, y tablas.
	Tables	3841	Tablas en memoria, definidas una vez

3.2.4 Quantization

3.2.4.1 Introducción

Este módulo hace las funciones de engarce entre el canal de demodulación y el proceso de decodificación de la información transmitida propiamente dicho, es decir, el que se realiza una vez que abandonan el dominio de la frecuencia los datos han sido transformados a formato bit.

Como todas las anteriores, esta función es invocada una vez por símbolo OFDM. Como ya sabemos, las tramas de transmisión constan de símbolos OFDM correspondientes al FIC y al MSC. La primera labor del quantization es discriminar los de un tipo y los del otro. Después, los trata de diferente manera debido a las diferencias existentes entre el tratamiento que se le da ambos tipos. En cualquier caso, también se encarga de realizar el de-mapping de (mediante un algoritmo de soft-decision) de los símbolos OFDM recibidos, los cuales se corresponden con la diferencia de fase calculada por el Demodulador Diferencial y reordenada por el Frequency De-interleaving, y de llamar a los siguientes módulos para continuar con el procesamiento cuando sea menester. Y aquí entramos en otra de las funciones importantes del quantization: el Block partitioning. De esta manera se conoce a la forma de estructurar la información para su posterior procesamiento, y el quantization se encarga de ir rellenando estas estructuras a medida que los datos le van llegando símbolo a símbolo.

Con respecto al FIC, el quantization sirve como pila de datos hasta que se ha recibido el FIC por completo. Para ello hace uso de los contadores almacenados en la estructura DABinfo correspondientes al estado del quantization para este tipo de símbolos OFDM. Lo primero que hace tras comprobar que el símbolo actual corresponde al FIC, es realizar el de-mapping. Como resultado obtiene un buffer de bits cuyo tamaño se corresponde con el doble del número de portadoras activas en un símbolo. Después, comprueba si el número de símbolos recibidos se corresponde con el esperado para el FIC en el modo de recepción DAB actual. Si es el caso, envía el buffer de datos al siguiente módulo para el FIC (De-puncturing) y deja en la estructura de datos la información de que el siguiente símbolo que llegue pertenecerá al MSC.

El proceso es diferente para el MSC. El MSC porta un número determinado de símbolos, los cuales pueden pertenecer a un mismo subcanal o varios subcanales de transmisión de datos, incluso un mismo símbolo puede portar información correspondiente a dos subcanales distintos. Además, no todos los subcanales están activos en todo momento, por lo que solamente los símbolos que se correspondan con subcanales activos han de ser procesados. He aquí la principal diferencia en el procesamiento del MSC con respecto al procesamiento del FIC: el volumen de datos a tratar es variable, no fijo. Por tanto, la estructura del tratamiento de del MSC es bastante más complicada que la referente al FIC.

3.2.4.2 Desarrollo de la implementación

El módulo de quantization no fue desarrollado de la misma manera que todos los demás. No fue desarrollado de forma independiente a todos los demás módulos y, una vez finalizado y testeado, integrado en el receptor de DAB. Debido a sus múltiples relaciones con otros módulos y a su papel de pila de datos, su diseño se abordó desde el propio receptor, puesto que era necesario que tuviera acceso a los diferentes módulos con los que interactúa.

De-mapping

En primer lugar se realizó la implementación de dos versiones diferentes de de-mapping. El de-mapping es un proceso mediante el cual se evalúan la diferencia de fase existente entre las portadoras, calculada en el Demodulador Diferencial, y en función de esa evaluación se reconstruye el dibit enviado en la transmisión y codificado en cada par IQ recibido por el quantization.

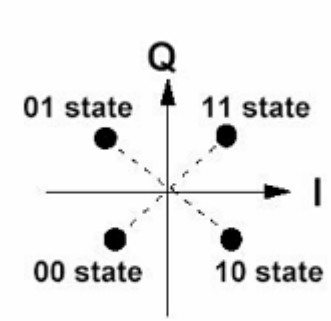


Ilustración 28: Modulación QPSK-4

Para la realización del de-mapping se desarrollaron dos algoritmos diferentes, conocidos por hard-decision y soft-decision debido al refinamiento a la hora de tomar la decisión.

El algoritmo de hard-decision asume que un bit solo puede ser 0 o 1, sin preocuparse de que más tarde hay está prevista la acción de un decodificador convolucional de Viterbi para reconstruir la señal. Por tanto, asigna 1 o 0 en función de que el valor de la componente imaginaria o real en una determinada portadora sea positiva o negativa.

Este algoritmo fue abandonado por el de soft-decision, el cual toma los 4 bits más significativos de la parte real de una portadora y se los asigna como valor probabilistico a un bit, de manera que cuanto más próximo esté un número a –1 más probabilidad tendrá de ser negativo y viceversa. Así, hace lo mismo con la parte imaginaria, y como resultado obtiene un dibit cuyo rango es {-7, -6...6, 7}.

Este dibit obtenido por ambos procesos de de-mapping para cada portadora no se almacena consecutivamente, sino que las partes reales de las portadoras activas se

corresponden con la primera mitad de la secuencia resultante, y las partes imaginarias se mapean en la segunda mitad.

Para distinguir si el símbolo que llega pertenece al FIC o al MSC se ha utilizado un flag que se inicializa a FICreceiving = true y después de recibir el FIC completo se inicializa con el valor que indica que es está recibiendo MSC, es decir, FICreceiving = false.

Tratamiento del FIC

El tratamiento del FIC depende del modo de recepción de datos DAB en el que estamos trabajando. Este modo de datos nos indica el número de símbolos OFDM que forman el FIC, así como el número de bits de información transportados en cada símbolo y por ende en el FIC completo. Tras el de-mapping de un símbolo, se actualiza el puntero de escritura en el buffer de bits del FIC para la siguiente recepción, y se procede a comprobar si ya se ha recibido el FIC completo. Si es este el caso, llega el momento de hacer una llamada a una función que se encarga de procesar el FIC para cada modo. Estas 4 funciones, una para cada modo, se acceden a través de una array de punteros a función. Lo que hace cada una de estas funciones es gestionar las llamadas a los módulos siguientes para el proceso de la información, comenzando por el módulo De-puncturing, siguiendo por el Decodificador de Viterbi y finalizando el en Energy Dispersal UnScrambler, por lo que marcan el nivel de tratamiento de datos. Desde esta función se deberá invocar al FIC decoder una vez que esté implementado (Ver sección 5.4.1 Aspectos no finalizados: Fuera de Objetivo)

Tratamiento del MSC

Este tratamiento es mucho más complicado que el anterior, por las razones anteriormente comentadas. En primer lugar, es menester comentar que el sistema está dotado de una lista de subcanales activos, aunque más vale decir una cola de subcanales activos. El primer elemento de la cola es el subcanal activo del que se espera recibir los datos en las primeros símbolos OFDM del MSC. Es decir, los primeros símbolos OFDM que sea menester procesar pertenecerán a dicho subcanal activo. Una vez recopilados todos los datos pertenecientes al subcanal, se llevará a cabo su procesamiento mediante la invocación de una función destinada a procesar un Subcanal, processSubchannel.

Para esto es necesario recopilar la información en primera instancia. Esto es complicado debido a que, como hemos comentado, un subcanal puede transportar su información en varios símbolos OFDM y dentro de un mismo símbolo OFDM se puede transportar información referente a dos subcanales distintos. Por tanto es necesario una estructura que utilice la información de la composición del MSC proporcionada por la decodificación del FIC para realizar esta criba en los datos que nos permite recopilar únicamente los datos del primer subcanal activo:

Para la recopilación de la información de una manera correcta se utilizan las etiquetas ya vistas de dirección de comienzo de subcanal y tamaño del mismo. Además, se tienen diversos punteros necesarios para realizar el recorrido por la trama, y contadores como el número de CIF procesados. Si la dirección final del subcanal que queremos procesar no está en el CIF actual (está en alguno sucesivo), se tramita todo el CIF sin

problema. Sin embargo, si esta dirección está dentro del mismo CIF, sólo se deben tratar los bits hasta dicha posición. Además, es importante tener en cuenta que no se van a tratar todos los subcanales que se reciban, sólo deben ser tratados los que se indiquen en la lista de subcanales activos, y por esta razón, es posible que se tengan que descartar CIFs enteros o parte de ellos sin realizar ninguna acción sobre ellos, sólo adelantar los punteros de recorrido y los correspondientes contadores.

3.2.4.3 Rendimiento parcial del módulo y pruebas

Para este módulo no se han tomado medidas de rendimiento, puesto que no se ha dispuesto en ningún momento de tramas de transmisión reales DAB para producir el testeo.

3.2.5 Time De-Interleaving

3.2.5.1 Introducción

Esta función tiene como objetivo dispersar los errores producidos por el ruido de impulso (ruido urbano, por ejemplo). Este módulo trabajo en el dominio del tiempo y gracias a él, mejora la recepción móvil del DAB. A diferencia del Frequency de-interleaving, en este caso no depende del modo de DAB sobre el que se esté trabajando ya que ya se ha hecho la cuantificación. Debido a que ahora trabajamos con tramas de bits, el tamaño de las tramas con las que trabajemos dependerá de la protección de datos utilizada, a más protección más número de bits a recolocar. Sin embargo, y debido a la necesidad de implementar de manera general el sistema, la tabla creada es de un tamaño tal que sea suficiente para todos los tipos de protección posible.

Esta tabla de la que hablamos es una manera de almacenar los datos antes de ser enviado al siguiente módulo. Como ya hemos dicho, se trata de una reordenación de bits en el tiempo, es decir, según nos llega una trama de bits, los recolocamos, según unas tablas de desplazamiento, en sus respectivas nuevas tramas. Por esta razón, se necesita recibir 16 tramas para poder generar una válida. Esta idea queda más clara si observamos la tabla 7, en la que se refleja la manera de reordenar los bits:

Tabla 7: Configuración Time de-Interleaving.

$R(i_r/16)$	$r'(r, i_r)$
0	r
1	r-8
2	r-4
3	r-12
4	r-2
5	r-10
6	r-6
7	r-14
8	r-1
9	r-9
10	r-5
11	r-13
12	r-3
13	r-11
14	r-7
15	r-15

Esta tabla significa que el primer bit que recibamos de nuestra trama se corresponderá con el bit de la trama actual, el segundo bit con el de la trama que recibimos hace ocho iteraciones, el siguiente con la trama que recibimos hace cuatro ciclos, y así sucesivamente. Además, es de destacar que este proceso de realiza de manera cíclica a lo largo de una trama, es decir, como todas las tramas tienen un tamaño múltiplo

de 16, éstas se dividen (conceptualmente) en subtramas de 16 bits para ser tratadas por esta función.

Por lo tanto, la manera de comportarse de este módulo consiste en ir recibiendo tramas de bits y, según las tablas de desplazamiento, recolocar los bits en una tabla auxiliar en la que se representan todos los bits de 16 tramas diferentes. La manera de devolver un resultado es mediante un puntero que señala la fila que en esa iteración acaba de ser completada (la que se comenzó a rellenar 16 iteraciones antes).

También es importante que ahora que conocemos el funcionamiento de este módulo, entendamos por qué no es implementado para tratar las tramas pertenecientes al FIC. Queda claro que el FIC es una parte de la transmisión que debe ser descodificada rápidamente y esto impide que pueda haber un módulo por el que tenga que estar esperando tanto tiempo. Sin embargo, es un módulo importante para evitar que demasiados errores de transmisión estén juntos y es útil para el MSC. Además, el FIC contrarresta estos errores de transmisión mediante otros mecanismos, como por ejemplo utilizar mayor nivel de protección de datos. La idea de separar los bits que puedan estar dañados debido a la transmisión tiene como objetivo que el módulo del Viterbi sea capaz de restaurar dichos errores. Si al Viterbi le llegan muchos bits con errores juntos, éste será incapaz de restaurar la trama original que se debería haber recibido. Un ejemplo de esta separación de errores se puede comprobar en la ilustración 29 donde en la parte de la izquierda no se realiza Time De-interleaving y en la derecha sí. Las tramas en este ejemplo se corresponden con las columnas, y las diferentes iteraciones con las filas:

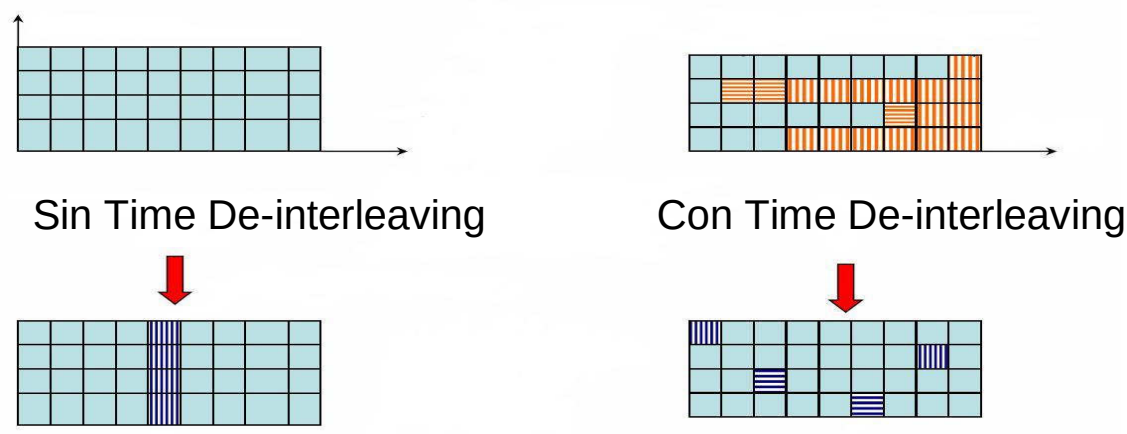


Ilustración 29: Time de-Interleaving.

3.2.5.2 Desarrollo de la implementación

Para poder llevar a cabo la implementación de éste módulo fue necesario, como ya hemos comentado, la creación de una tabla intermedia que se adaptase a todos los niveles posibles de protección. Esto significa que la tabla debía tener un tamaño de 16 por el tamaño de la trama que fuésemos a trazar. En un principio para recolocar todos los bits, se creó una tabla general de desplazamiento, pero al tener que depender del nivel de protección de datos, esta no era una manera eficaz de implementar el código. Por ello, se

pensó en utilizar el concepto de circular buffer que tiene implementado Coolflux, pues de este modo, cuando se llegase al final de la tabla intermedia, el puntero volvería a apuntar al principio de la tabla; recordemos que para cada trama se realizaba la recolocación de los bits un número de veces igual al tamaño de la trama partido por 16, y por esa razón se tenía que estar continuamente volviendo al principio de la tabla ya que las filas representaban las diferentes tramas. Una vez implementamos esta idea, pudimos comprobar cómo el comportamiento del módulo en las pruebas no era el esperado pues devolvía resultados extraños. En ese momento nos dimos cuenta de que el tamaño de los punteros utilizados para realizar el circular buffer no era suficiente para abarcar todo el tamaño de la tabla que queríamos implementar.

Una vez encontramos este problema, y ante la imposibilidad de cambiar el tamaño de los punteros, decidimos implementar las tablas de desplazamiento de tal modo que modificasen el puntero automáticamente para volver al principio una vez se llegaba al final, y así evitar escrituras fuera de memoria. Finalmente, se generaron 16 tablas de desplazamiento, una por cada iteración necesaria, y una tabla general que llamaba a la tabla de desplazamiento a la que le tocase actuar según el número de iteración en el que se encontraba el módulo.

Además, hay que recordar que este módulo está después de la multiplexación de los datos según los canales. Esto hace que el módulo pueda ser distinto dependiendo del subcanal pues, como ya hemos visto, no todos los subcanales tienen el mismo nivel de protección. Estas razones llevan a que se implementase una función llamada preparación del interleaving que debía ser llamada al crear cada subcanal de tal manera que se crease la tabla intermedia y se inicializasen las tablas de desplazamiento dependiendo del subcanal.

Una parte que no se trató del time de-interleaving en este proyecto fue la posibilidad de una reconfiguración del sistema. Según el estándar de DAB, si surge una reconfiguración del sistema, el cambio en el nivel de protección de datos es el que afecta a este módulo, el time de-interleaving debe acomodar el tamaño de su tabla intermedia a estos cambios, y además, tardaría quince iteraciones en retomar el nuevo funcionamiento. Sin embargo, estos posibles cambios no han sido considerados debido a los problemas que suponía el tratamiento de memoria dinámica en el DSP, de hecho la memoria dinámica era inexistente hasta que implementamos la función simuladora ya comentada.

3.2.5.3 Correcciones

Además de los cambios ya comentados en la implementación de las tablas de desplazamiento, se realizaron algunos cambios en el alojamiento de los datos en memoria. Primero se hicieron todas las pruebas posibles con la colocación de las tablas de desplazamiento y la salida ya que la entrada estaba establecida, por el módulo anterior, en memoria X. Este estudio no sólo incluye las tablas, intermedia y de desplazamiento, también incluye los punteros hacia esas tablas. Esto hizo que la cantidad de pruebas fuese bastante grande.

Por otra parte, aunque hemos comentado que el sistema finalmente no quedó preparado para reconfiguraciones del tipo de protección de datos, si se implementó una

función a modo de prueba en la que se cambiaban éstos parámetros. La razón de que se implementase esta función pero al final no fuese integrada en el proyecto general, fueron los problemas que se tuvieron para el tratamiento de memoria dinámica en el DSP. Estos problemas fueron que había que reservar el tamaño fijo para la tabla intermedia, y al no poder liberar memoria, ni realizar ampliaciones sobre una reserva ya creada, el tamaño de esa tabla no podía variar. De todos modos, la función a la que nos referimos en este punto era llamada "preparación del interleaving" y era llamada siempre antes de llamar al time de-interleaving. Esta función tenía en cuenta si se había producido un cambio en el tipo de protección de datos, si había que reducir o ampliar la tabla intermedia, cuántos ciclos habían pasado desde la última reconfiguración, etc.

3.2.5.4 Rendimiento parcial del módulo y pruebas

Como ya hemos explicado, se realizaron diversos estudios para encontrar la mejor configuración de memoria de este módulo. Estas pruebas nos llevaron a deducir que la mejor manera de alojar los datos era con la tabla intermedia (datos de salida) en memoria X, y con las tablas de desplazamiento en memoria Y. En cuanto a los punteros que guían las tablas de desplazamiento, también se alojaron en memoria Y.

Es importante destacar que para este módulo no se generaron test para comprobar la pérdida de precisión ya que era imposible al trabajar solamente con números enteros y no realizar ninguna operación de cambio de tipo sobre ellos. Además, tampoco se crearon funciones para filtrar los datos como en otros módulos se había hecho por la misma razón de que aquí ya se estaba trabajando directamente con números enteros.

A continuación se muestra la tabla 8, donde se puede comprobar el consumo de ciclos del módulo con un tamaño de entrada de 1024 elementos, y una tabla con el correspondiente consumo de memoria:

Tabla 8: Rendimiento en ciclos Time de-Interleaving.

Función	Tamaño de la entrada	Ciclos por llamada
TimeDeInterleaver	1024	1576

Tabla 9 Rendimiento en memoria Time de-Interleaving.

		Palabras (3 bytes)	
XMEM	Stack	1024	Máximo consumo + 3 words
	Statics	10015	Definidas en cada instancia
	Literals	0	Constantes del programa
	Tables	4096	Tablas en memoria, definidas una vez
YMEM	Statics	10000	Variables estáticas, buffers, y tablas.

En esta última tabla se puede comprobar cómo el consumo en la memoria Y de las tablas de desplazamiento se produce de una manera estática. Es decir, una vez inicializadas estas tablas, no vuelven a cambiar sus valores.

3.2.6 DePuncturing

3.2.6.1 Introducción

Con el fin de proteger los datos frente a errores en la transmisión, la señal es codificada convolucionalmente en el proceso de transmisión. La codificación aplicada en DAB conlleva un incremento del 400% en el ancho de banda necesario para transmitir (enviar y recibir) la misma cantidad de información. Para paliar este efecto sobre el ancho de banda, algunos bits son eliminados, por lo que no se transmite entera la palabra codificada resultante del proceso de codificación convolucional. Este procedimiento se conoce como puncturing, y gracias a él es posible dotar al sistema de transmisión de múltiples niveles de protección de la señal, puesto que cuanto más queramos proteger una señal menos bits dejaremos de transmitir (el decodificador convolucional tendrá un mejor rendimiento al reconstruir la señal emitida original), y viceversa.

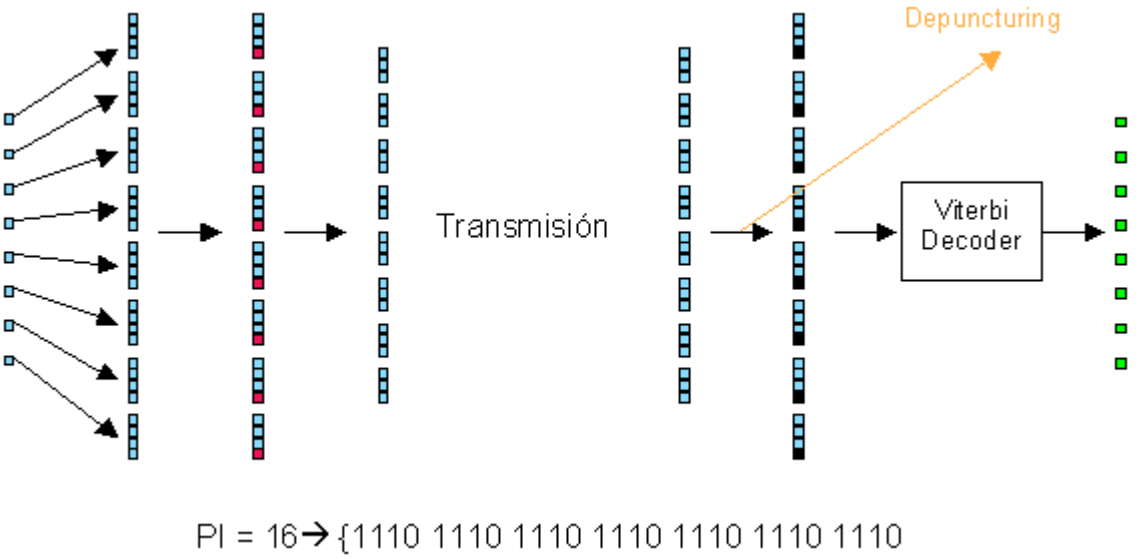


Ilustración 30: DePuncturing

Así pues, algunos bits son eliminados de la señal de transmisión de acuerdo a un patrón de puncturing. Dado que se pueden definir diversos niveles de protección en función de los bits que eliminemos, en el estándar de DAB quedan definidos perfiles de protección específicos, los cuales son combinaciones de diferentes patrones de puncturing, definidos también en el estándar, aplicados a una cantidad de datos mayor.

El proceso básico de puncturing se realiza sobre sub-bloques de 32 bits. Estos sub-bloques están enmarcados dentro de bloques de 128 bits, los cuales a su vez forman parte de la palabra codificada (a la que llamaremos codeword de aquí en adelante).

El número de bloques de 128 bits que tiene un determinado codeword viene dado por el número total de bits que puede ser transmitido en una misma trama de transmisión

para el FIC y para el MSC, y por tanto depende del número de bits transmitidos en cada símbolo OFDM, esto es, depende del modo de recepción de DAB en el que estemos recibiendo. A continuación se describe el proceso de puncturing en DAB.

Un codeword se compone de 4I + 24 bits. Los primeros 4*I bits (u_0, u_1, u_2,..., u_{4I-1}) del codeword U son divididos en bloques de 128 bits, cada uno de los cuales es a su vez sub-dividido en 4 sub-bloques de 32 bits. Todos los sub-bloques pertenecientes a un mismo bloque de 128 bits son sometidos al proceso de puncturing con el mismo patrón, el cual se identifica mediante un índice de punturing, PI (Puncturing Index). Los índices de puncturing se corresponden con vectores de puncturing, V_{PI}, los cuales se organizan como sigue:

$$V_{PI} = (V_{PI,0}\ ,\ V_{PI,1}\ ,\ V_{PI,i}\ ,\ V_{PI,31}\ )$$

El elemento i-ésimo de cada sub-bloque de bits es procesado de acuerdo al valor del elemento i-ésimo en el V_{PI}, de la siguiente forma:

- Si $V_{PI,i} = 0$, el bit correspondiente es eliminado del sub-bloque y por tanto no es transmitido.
- Si $V_{PI,i} = 1$, el bit correspondiente es mantenido en el sub-bloque y por tanto transmitido.

La tabla 10 muestra los vectores de índices en función del PI.

Tabla 10: Configuración de-Puncturing.

PI	(VPI, 0,... ... , VPI, 31)
1	1100 1000 1000 1000 1000 1000 1000 1000
2	1100 1000 1000 1000 1100 1000 1000 1000
3	1100 1000 1100 1000 1100 1000 1000 1000
4	1100 1000 1100 1000 1100 1000 1100 1000
5	1100 1100 1100 1000 1100 1000 1100 1000
6	1100 1100 1100 1000 1100 1100 1100 1000
7	1100 1100 1100 1100 1100 1100 1100 1000
8	1100 1100 1100 1100 1100 1100 1100 1100
9	1110 1100 1100 1100 1100 1100 1100 1100
10	1110 1100 1100 1100 1110 1100 1100 1100
11	1110 1100 1110 1100 1110 1100 1100 1100
12	1110 1100 1110 1100 1110 1100 1110 1100
13	1110 1110 1110 1100 1110 1100 1110 1100
14	1110 1110 1110 1100 1110 1110 1110 1100
15	1110 1110 1110 1110 1110 1110 1110 1100
16	1110 1110 1110 1110 1110 1110 1110 1110
17	1111 1110 1110 1110 1110 1110 1110 1110
18	1111 1110 1110 1110 1111 1110 1110 1110
19	1111 1110 1111 1110 1111 1110 1110 1110
20	1111 1110 1111 1110 1111 1110 1111 1110
21	1111 1111 1111 1110 1111 1110 1111 1110
22	1111 1111 1111 1110 1111 1111 1111 1110
23	1111 1111 1111 1111 1111 1111 1111 1110
24	1111 1111 1111 1111 1111 1111 1111 1111

Los últimos 24 bits del codeword tienen su propio vector de índices para el proceso de puncturing:

VT = (1100 1100 1100 1100 1100 1100)

Los 12 bits resultantes se conocen como bits de cola, y son añadidos al final de los bits resultantes del proceso de puncturing, después de los bits correspondientes a cada bloque de 128 bits.

En función del esquema de puncturing, es posible que sea necesario añadir ceros al final de esta secuencia de bits resultante, con el fin de que el número de bits sea múltiplo de 64 bits.

En el receptor, se tiene que realizar el proceso inverso, y la señal original tiene que ser reconstruida en su estructura con el fin de que el decodificador convolucional sea

capaz de reconstruir la señal original. Para ello, el receptor utiliza el mismo perfil de protección aplicado en el transmisor a un conjunto de datos y ubica un bit '0' en cada lugar correspondiente a un bit eliminado en el proceso de puncturing. Este proceso es conocido como De-puncturing.

Esquemas de protección

El proceso de de-puncturing se aplica de manera diferente al FIC y al MSC. Esto es así porque en el caso del FIC no se hace time de-interleaving, y por tanto se realiza el de-puncturing inmediatamente después del proceso de mapeo y cuantización.

Para el de-puncturing del FIC se juntan todos los bits provenientes de los símbolos OFDM que forman dicho FIC y se envían al módulo de de-puncturing como una secuencia de bits unificada. Estas secuencias tienen una configuración o patrón de puncturing definida, en función del número de bits que forman la secuencia unificada:

El codeword está dividido en L bloques de 128 bits ($L = L_1 + L_2$), y cada conjunto de bloques se codifica con un determinado PI.

Tabla 11: Codeword dependiente del modo de transmisión.

Modos DAB	bits de información real en el FIC	bits de información codificada en FIC	bits de información tras puncturing	(L1, PI1)	(L2, PI2)
I, II, IV	768	3072	2304	(21,16)	(3,15)
III	1024	4096	3072	(29,16)	(3,16)

Esto es así porque en el caso del FIC no es crítico el nivel de protección de la información y sí la rapidez de procesamiento de la misma, por lo que se reduce el número de bits transmitidos a cambio de una mayor velocidad de transmisión.

En lo referente al MSC, hay varios niveles de protección definidos, dentro de dos políticas de protección distintas UEP (Unequal Error Protection) y EEP(Equal Error Protection). El denominador común de todos ellos es que realizan el de-puncturing de varios bloques consecutivos de 128 bits con el mismo PI. A continuación se muestran las tablas correspondientes a los diversos niveles de protección dentro de ambas políticas:

Tabla 12: Perfiles de Protección UEP

Audio bit rate (kbit/s)	P	L_1	L_2	L_3	L_4	PI_1	PI_2	PI_3	PI_4	number of padding bits
32	5	3	4	17	0	5	3	2	-	0
32	4	3	3	18	0	11	6	5	-	0
32	3	3	4	14	3	15	9	6	8	0
32	2	3	4	14	3	22	13	8	13	0
32	1	3	5	13	3	24	17	12	17	4
48	5	4	3	26	3	5	4	2	3	0
48	4	3	4	26	3	9	6	4	6	0
48	3	3	4	26	3	15	10	6	9	4
48	2	3	4	26	3	24	14	8	15	0
48	1	3	5	25	3	24	18	13	18	0
56	5	6	10	23	3	5	4	2	3	0
56	4	6	10	23	3	9	6	4	5	0
56	3	6	12	21	3	16	7	6	9	0
56	2	6	10	23	3	23	13	8	13	8
64	5	6	9	31	2	5	3	2	3	0
64	4	6	9	33	0	11	6	5	-	0
64	3	6	12	27	3	16	8	6	9	0
64	2	6	10	29	3	23	13	8	13	8
64	1	6	11	28	3	24	18	12	18	4
80	5	6	10	41	3	6	3	2	3	0
80	4	6	10	41	3	11	6	5	6	0
80	3	6	11	40	3	16	8	6	7	0
80	2	6	10	41	3	23	13	8	13	8
80	1	6	10	41	3	24	17	12	18	4
96	5	7	9	53	3	5	4	2	4	0
96	4	7	10	52	3	9	6	4	6	0
96	3	6	12	51	3	16	9	6	10	4
96	2	6	10	53	3	22	12	9	12	0
96	1	6	13	50	3	24	18	13	19	0
112	5	14	17	50	3	5	4	2	5	0
112	4	11	21	49	3	9	6	4	8	0
112	3	11	23	47	3	16	8	6	9	0
112	2	11	21	49	3	23	12	9	14	4
128	5	12	19	62	3	5	3	2	4	0
128	4	11	21	61	3	11	6	5	7	0
128	3	11	22	60	3	16	9	6	10	4
128	2	11	21	61	3	22	12	9	14	0
128	1	11	20	62	3	24	17	13	19	8
160	5	11	19	87	3	5	4	2	4	0
160	4	11	23	83	3	11	6	5	9	0
160	3	11	24	82	3	16	8	6	11	0
160	2	11	21	85	3	22	11	9	13	0
160	1	11	22	84	3	24	18	12	19	0
192	5	11	20	110	3	6	4	2	5	0
192	4	11	22	108	3	10	6	4	9	0
192	3	11	24	106	3	16	10	6	11	0
192	2	11	20	110	3	22	13	9	13	8
192	1	11	21	109	3	24	20	13	24	0
224	5	12	22	131	3	8	6	2	6	4
224	4	12	26	127	3	12	8	4	11	0
224	3	11	20	134	3	16	10	7	9	0
224	2	11	22	132	3	24	16	10	15	0
224	1	11	24	130	3	24	20	12	20	4
256	5	11	24	154	3	6	5	2	5	0
256	4	11	24	154	3	12	9	5	10	4
256	3	11	27	151	3	16	10	7	10	0
256	2	11	22	156	3	24	14	10	13	8
256	1	11	26	152	3	24	19	14	18	4
320	5	11	26	200	3	8	5	2	6	4
320	4	11	25	201	3	13	9	5	10	8
320	2	11	26	200	3	24	17	9	17	0
384	5	11	27	247	3	8	6	2	7	0
384	3	11	24	250	3	16	9	7	10	4
384	1	12	28	245	3	24	20	14	23	8

Como puede verse, hay varios niveles de protección para una misma tasa de bits transmitidos. El codeword es dividido en L bloques de 128 bits ($L = L_1 + L_2 + L_3 + L_4$). El depuncturing se realiza de la siguiente manera:

1. L_1 bloques con PI_1
2. L_2 bloques con PI_2
3. L_3 bloques con PI_3
4. L_4 bloques con PI_4
5. De-puncturing de los bits de cola
6. Se ignoran los bits de padding.

Este esquema de protección está destinado a datos de audio.

El caso del esquema de protección EEP es diferente. Depende de si la tasa de bits del nivel de protección es múltiplo de 8 o de 32. En cualquier caso, el esquema es similar al de UEP: el codeword es dividido en L bloques de 128 bits ($L = L_1 + L_2$). El depuncturing se realiza de la siguiente manera:

1. L_1 bloques con PI_1
2. L_2 bloques con PI_2
3. De-puncturing de los bits de cola
4. Se ignoran los bits de padding.

Tabla 13: Perfiles de protección EEP para tasas de bits múltiplos de 8

Data bit rate (kbit/s)	*P*	L_1	L_2	PI_1	PI_2
8n	4-A	4n-3	2n+3	3	2
8n	3-A	6n-3	3	8	7
8	2-A	5	1	13	12
8n (n>1)		2n-3	4n+3	14	13
8n	1-A	6n-3	3	24	23

Tabla 14: Perfiles de protección EEP para tasas de bits múltiplos de 32

Data bit rate (kbit/s)	*P*	L_1	L_2	PI_1	PI_2
32n	4-B	24n-3	3	2	1
32n	3-B	24n-3	3	4	3
32n	2-B	24n-3	3	6	5
32n	1-B	24n-3	3	10	9

Este esquema está destinado tanto a transmisión de audio como de datos.

3.2.6.2 Desarrollo de la implementación

Como se ha indicado anteriormente, el proceso básico de punturing se realiza sobre vectores de 32 bits en función de un índice de puncturing, PI, que indica el tratamiento que debe darse a la secuencia.

En la misma línea, el PI nos proporciona información de cómo debe realizarse el de-puncturing: han de tomarse 8+PI bits de entrada para producir 32 bits de salida por sub-bloque. Por tanto, para cada bloque serán 4*(8+PI) bits de entrada para reconstruir 128 bits.

Para cada tipo de de-puncturing que nos marca un PI, se desarrolló una función independiente que tratara un sub-bloque cada vez, la cual fuera invocada desde funciones a más alto nivel que trataran independientemente el de-puncturing para el FIC y para el MSC en sus versiones UEP y EEP.

Así pues se desarrollaron en total 28 funciones para el procesamiento del de-puncturing:

- 24 funciones a nivel de sub-bloque (tratando un número n de sub-bloques)
- 1 función para el tratamiento de los bits de cola
- Una función para el tratamiento del FIC
- Dos funciones para el tratamiento MSC, una para cada sistema de protección(EEP, UEP)

Tanto las funciones de nivel superior destinadas al tratamiento del FIC o de MSC, como las funciones de tratamiento de n sub-bloques. reciben un puntero a buffer de entrada y devuelven un puntero a buffer de salida. Por tanto, fue necesario declarar localmente un puntero a buffer, inicializarlo con el valor del parámetro de entrada, y después ir actualizándolo en los valores pertinentes a medida que se leen/producen los datos de entrada/salida (en el caso de las funciones de sub-bloque, a nivel de bucle; en el caso de las demás, tras cada llamada a función de tratamiento de n sub-bloques).

Durante el desarrollo del módulo se mantuvieron dos versiones diferentes de todas las funciones: una que trabajaba únicamente sobre XMEM y otra que escribía los resultados en YMEM. Esta práctica se llevó a cabo en algunos de los módulos para proporcionar flexibilidad al diseño cuando aún no se habían tomado ciertas decisiones estructurales acerca de la ubicación de los buffers de datos sitos entre dos módulos acoplados.

3.2.6.3 Mejoras realizadas

Empaquetado de los datos

En primera instancia los bits de salida fueron codificados como enteros, al igual que los de entrada. Sin embargo, cuando se abordó la implementación del decodificador de Viterbi se observó que era necesario disponer de los datos empaquetados en simd12 para poder hacer uso del hardware de viterbi específico. Por motivos de acoplamiento de ambos módulos, fue necesario empaquetar en simd12 los datos de salida del de-puncturing, almacenando por tanto dos bits en una misma palabra. Esto produjo una mejora de rendimiento significativa en el de-puncturing, además de un importante ahorro de espacio en memoria necesario para alojar los datos de salida. Para realizar el empaquetamiento de acuerdo a las necesidades se creó un nuevo tipo de datos isimd12 el cual funciona con números enteros solamente. Su funcionamiento es completamente análogo al simd12 pero con la salvedad de que solo admite datos enteros. Una nueva funcionalidad añadida a CoolFlux.

Estructura funcional

En un primer momento las funciones de de-puncturing eran atómicas a nivel de sub-bloque, es decir, procesaban un sub-bloque por cada llamada a función. Según se fue conociendo la estructura del sistema de protección de datos en DAB se observó que se producían muchas llamadas consecutivas a una misma función, en el interior de un bucle for. Para evitar la sobrecarga producida por las llamadas consecutivas a función, se cambió el esquema y se introdujo el bucle for dentro de la función, recibiendo los límites del bucle como parámetros en dichas funciones.

Array de punteros a función

Para evitar la sobrecarga de evaluación condicional en el acceso a la función pertinente para un determinado PI, y aprovechando la estructura común de todas las funciones de de-puncturing a nivel de sub-bloque, se implementó un array de punteros a función que direccionaba a la función adecuada según el valor del PI.

Reestructurado de bucles de sub-bloque

Como se puede observar en la tabla XXX, para ciertos vectores de índices se repiten patrones cada 4, 8 o 16 bits. En estos casos, se procedió a aumentar el contador del bucle multiplicándolo por 8, 4 y 2 respectivamente, de manera que se procesara el único patrón presente. Esto derivó en una reducción del tamaño de PMEM necesario para las funciones en las que se pudo llevar a cabo, pero no significó ninguna ganancia en ciclos de procesador.

3.2.6.4 Rendimiento parcial del módulo y pruebas

La tabla 15 muestra los resultados obtenidos al ejecutar el programa en el simulador de CoolFlux.

Tabla 15: Rendimiento De-Puncturing.

Función	Ciclos por llamada	Palabras en PMEM	Palabras en XMEM	Palabras en YMEM
De_Puncturing_PI_1	25	23	9	16
De_Puncturing_PI_2	28	16	10	16
De_Puncturing_PI_3	28	25	11	16
De_Puncturing_PI_4	30	12	12	16
De_Puncturing_PI_5	30	27	13	16
De_Puncturing_PI_6	32	18	14	16
De_Puncturing_PI_7	32	29	15	16
De_Puncturing_PI_8	34	10	16	16
De_Puncturing_PI_9	35	30	17	16
De_Puncturing_PI_10	38	19	18	16
De_Puncturing_PI_11	39	32	19	16
De_Puncturing_PI_12	40	13	20	16
De_Puncturing_PI_13	43	34	21	16
De_Puncturing_PI_14	46	23	22	16
De_Puncturing_PI_15	46	36	23	16
De_Puncturing_PI_16	48	9	24	16
De_Puncturing_PI_17	48	38	25	16
De_Puncturing_PI_18	50	22	26	16
De_Puncturing_PI_19	50	38	27	16
De_Puncturing_PI_20	52	13	28	16
De_Puncturing_PI_21	52	39	29	16
De_Puncturing_PI_22	54	22	30	16
De_Puncturing_PI_23	54	39	31	16
De_Puncturing_PI_24	55	39	32	16
De_Puncturing_PI_LastMotherCodeword	27	23	12	12
De_Puncturing_YMEM_FIC_Codeword DABmode I,II, IV DABmode III	 3903 5183		 2304 3096	 1548 2060
MSCGenericDe_Puncturing_YMEM_UEP (Depende del nivel de protección)	Caso mejor: 2135 Caso peor: 43222			
MSCGenericDe_Puncturing_YMEM_EEP (Depende del nivel de protección)	Caso mejor: 572 Caso peor: 41547			

3.2.7 Decodificación Convolucional

3.2.7.1 Introducción

Este módulo también es conocido como Viterbi ya que es la técnica que utiliza para llevar a cabo su función. A diferencia de otros módulos, este no trata de evitar errores en la transmisión sino arreglar errores que ya se han producido. A grandes rasgos podemos decir que lo que trata ésta función es de recomponer la trama original de bits que tenía que haber llegado.

La manera de conseguir su objetivo se basa en que en el transmisor existe un módulo que por cada bit, genera otros cuatro, bien, pues gracias a tener los bits repetidos, el módulo puede deducir con que bit se corresponden aunque en alguno de ellos se haya producido un error. En este punto debemos hacer un inciso para destacar la función del Time De-Interleaving a la hora de dispersar los errores. Imaginemos que no se separasen los errores y el bit que intentamos recomponer se corresponda con cuatro contiguos en los que se produjo un error; sería imposible acertar a la hora de deducir el bit original.

El paso de un bit a cuatro en el transmisor se hace basándose en unas ecuaciones que producen el llamado código convolucional. Gracias a esas ecuaciones, podemos conocer los posibles bits que pueden generar las diferentes configuraciones de los cuatro bits. Éste no es el único modulo que reduce el número de bits recibidos en la trama, pero si es el único que debe deducirlos. Por ejemplo, como acabamos de ver, el módulo anterior tiene una función parecida a éste, pero la reducción de bits la hace bajo el conocimiento de unas tablas establecidas. Podemos ver la situación del módulo y sus entradas/salidas en la ilustración 31.

Ilustración 31: Módulo Decodificador convolucional.

En esta figura se puede apreciar como varía el tamaño máximo de los datos de entrada y de salida. De hecho, si nos fijamos en la parte de arriba de la figura, podemos ver que la entrada se corresponde con un tamaño igual a 4*I+24, y la salida con un tamaño igual a I. Esto se debe a que no sólo se reduce en cuatro el tamaño, sino que además, los últimos 24 bits de la trama son considerados de relleno.

Una vez tenemos unas nociones de éste módulo, vamos a intentar comprender cómo realiza sus tareas. Para poder conseguir su objetivo, la función de Viterbi se basa en la estructura conocida como diagrama de Trellis. En este diagrama, se van anotando los

diferentes paquetes de bits que se van recibiendo y cuáles son sus posibles correspondencias con el bit original. Este diagrama es como una tabla en el que el número de filas dependerá de la constante de restricción (constraint length) utilizada en la codificación (es un dato conocido mediante la información procedente del FIC), en nuestro caso al tener un constraint length de 7, el número de filas será igual a $2^{7-1}= 64$, de manera general la fórmula para conocer el número de columnas (delay states) es escrita como 2^{k-1}, conociendo k como el valor del constraint lenght; y tantas columnas como recepciones de 4 bits se produzcan, es decir, el tamaño de la trama recibida partido por cuatro. Por tanto, las transiciones de una columna a otra, de un estado a otro, se producirán por codificaciones compuestas por cuatro bits. Para poder entender mejor el concepto del diagrama de Trellis, podemos ver la ilustración 32 en la que se muestra un ejemplo de este diagrama para un code-rate de ½, un constraint length de 3 (por lo que tiene 4 filas), y un tamaño de entrada de 16 bits.

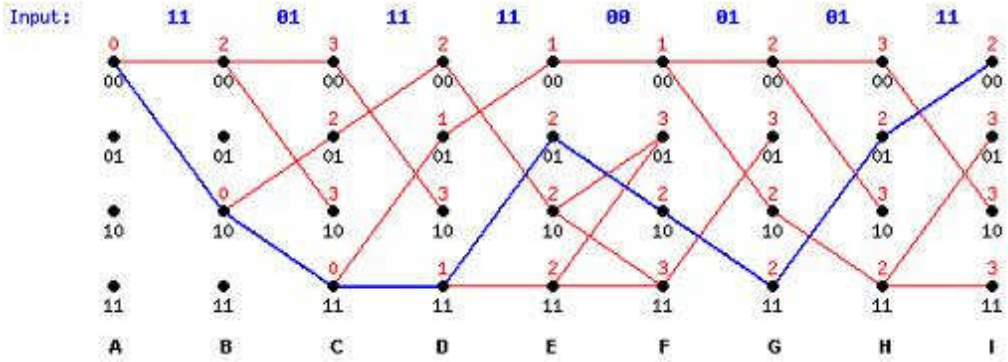


Ilustración 32: Diagrama de Trellis.

La figura anterior muestra como el diagrama siempre empieza en el estado cero (fila cero), y por cada dos bits que recibe tiene dos posibles caminos, que el bit original fuese un cero o un uno. Una vez deduce los caminos posibles para esos dos bits, calcula la distancia con el estado al que lleva cada uno de esos caminos. En el caso de los dos primeros bits se recibe un '11', esto produce dos destino que son '00' y '10'; una vez calcula la distancia entre la entrada y los posibles objetivos, realiza una evaluación para comprobar cuál de los caminos es más probable. Estas "notas" que se le adjudican a los diferentes estados a lo largo de la ejecución, son guardadas, pues serán útiles en el siguiente paso del proceso. Una vez se han recibido todos los bits, los posibles caminos de cada recepción están marcados en rojo, se hace la valoración final de la última columna, y la que tenga mayor "nota", será elegida como comienzo de la vuelta atrás. Esta vuelta atrás lo que hace es recomponer la trama final recorriendo todo el diagrama desde el final al principio. Para guiarse en su recorrido, utiliza las notas que se fueron guardando previamente, que son las que se utilizaron para elegir uno de los dos caminos posibles en su momento. En azul queda marcado en la figura el camino final escogido después de la vuelta atrás.

En resumen, podemos decir que el algoritmo del Viterbi se basa en tres pasos: construcción del diagrama de Trellis paso a paso, evaluación de cada camino generado, y vuelta atrás con la elección del camino final.

3.2.7.2 Desarrollo de la implementación

Como se puede deducir de la introducción previa, el código que se implementó tiene un nivel de complejidad bastante alto. De hecho, como se verá en el siguiente apartado hubo que hacer diversos cambios en el procesador para poder implantar este módulo. A la hora de plantear su implementación, se dividió el trabajo en dos partes claramente diferenciadas, la construcción de los posibles caminos hasta el final, y la segunda, la vuelta atrás reconstruyendo la trama final.

Para esta primera parte del código se crearon dos funciones fundamentales para lograr el objetivo de la función general. Una primera función calcula la distancia entre los bits recibidos y los esperados según las cuatro codificaciones posibles que se deberían recibir '0000', '1001', '0010', '1011'. De este modo se consigue puntuar cada estado de los 64 posibles para poder saber cuál es el más probable. De hecho la inicialización de la puntuación de la primera etapa no es trivial, antes de empezar a ejecutar se establecen puntuaciones negativas a todos los estados excepto al estado cero al que se le da una puntuación neutral.

Una vez se han calculado las distancias, el siguiente paso es calcular los posibles caminos para llegar a los estados correspondientes de la próxima etapa. Para esta llevar a cabo su tarea, existen dos funciones específicamente implementadas para ello (viterbi_add y viterbi_sub). Estas funciones aprovechan la proximidad de los estados contiguos para calcular las transiciones, por cada estado se realiza una suma y una resta (correspondiéndose con la posibilidad de que el bit real sea un uno o un cero). Además, estas sumas y restas se realizan simultáneamente para dos estados contiguos gracias a las funciones ya mencionadas, si el estado se corresponde con los de la parte superior de la tabla, el camino superior se corresponde con la suma, el inferior con la resta, y viceversa.

Una vez se han ejecutado 72 veces las funciones previas, comienza la vuelta atrás. La razón de ejecutar exactamente 72 veces las funciones anteriores es debido a que al ir sumándose las puntuaciones de los estados, a partir de la ejecución 72, estas puntuaciones (en ciertos casos), superan el rango sobre el que puede trabajar el tipo de datos utilizado para las puntuaciones. Por eso, lo que hace el código es ejecutar hasta el máximo de iteraciones posibles (72), y en ese momento comenzar la vuelta atrás. Una vez se quiere volver a llamar a la primera parte del proceso para continuar tratando los bits restantes, se escoge la máxima puntuación de la última etapa y a todos los estados de dicha etapa se les resta este dato. De este modo, todas las puntuaciones comenzarán siendo negativas excepto la que era máxima, que ahora será neutra.

El segundo gran bloque del código es la vuelta atrás (trace back), esta parte se compone de dos funciones, la inicialización de los datos necesarios para desarrollar su trabajo, y la propia función. Los datos de inicialización, aparte del mejor estado de la última fase y el vector de caminos que ha escrito la función de calcular caminos, constan de una estructura que incluye una máscara y un bit que indican el estado al que pertenece y si ese estado está en la mitad superior o inferior de la tabla (0 y 1 respectivamente).

Una vez se tienen los datos necesarios, se puede ejecutar la función de la vuelta atrás la cual va leyendo el vector de caminos recibidos y eligiendo uno de ellos, una vez sabe cual es el camino elegido, lo escribe en el array de salida y se posiciona en el estado correspondiente de la etapa anterior para volver a comenzar el proceso. Una vez llega a la primera etapa (relativa, ya que se ejecutan 72 etapas como máximo cada vez), devuelve la trama de bits final.

3.2.7.3 Correcciones

En este apartado es importante recordar que se tuvieron que implementar funciones específicas del procesador para poder optimizar este módulo. Sin embargo, existieron problemas al desarrollar estas ampliaciones en el procesador. Precisamente para poder incluir estas nuevas mejoras, el proveedor del software de desarrollo nos proporcionó una nueva versión.

Como habíamos estado realizando hasta el momento, se hizo una versión previa del código bajo *Microsoft Visual Studio,* y una vez comprobado su buen funcionamiento sin funciones específicas del procesador, se le añadieron éstas. Al añadirlas nos dimos cuenta de que el comportamiento no era el mismo, no daba los resultados correctamente, y para comprobar el problema tuvimos que repasar el código de las nuevas librerías. En ese momento fue cuando pudimos comprobar que se habían producido errores en su implementación. El problema de descubrir errores en las librerías era que podíamos solventarlos directamente nosotros pero trabajar con ellas solamente bajo *Microsoft Visual Studio,* ya que para hacerlo bajo Coolflux debíamos mandar las mejoras al proveedor externo de software y esperar a que nos lo devolviesen ya corregido. Esta última versión del software no pudo llegar a tiempo durante nuestra estancia en la empresa y por eso tuvimos que trabajar sin utilizar el software propio del DSP.

Estas son las razones de que no se pudiesen hacer diferentes pruebas para ver el comportamiento del módulo con distintas configuraciones de memoria ya que bajo Coolflux el código no llegaba a compilar. Sin embargo, si se pudieron realizar pruebas para comprobar su funcionamiento (aunque no su optimización). Estas pruebas constan de dos funciones de inicialización en las que se les pasa un código y ellas generan su correspondiente codeword. Hay que decir que aunque se pueda suponer que estas tramas con las que trabaja el Viterbi son de ceros y unos, esto no es así. Todos los datos con los que trabaja el receptor de DAB desde el módulo de cuantización son valores enteros comprendidos entre -7 y 7, donde -7 se corresponde con uno y 7 con cero; los valores intermedios se corresponden con la probabilidad de ser un 7 o un -7. Eso sí, la salida del viterbi proporciona una trama compuesta por ceros y unos.

Las nuevas funciones por las que hubo problemas con las librerías del DSP son muy importantes para la optimización de este módulo ya que lo que hacen es aprovechar la cercanía entre estados contiguos para realizar tres operaciones de una manera óptima. Estas tres operaciones son conocidas como “mariposas” y lo que hacen es realizar la suma del primer estado con las distancias y la resta con el segundo estado (en el caso del viterbi_add, en el viterbi_sub sería al contrario).

3.2.7.4 Rendimiento parcial del módulo y pruebas

Como ya hemos comentado, no se consiguió a tiempo realizar la actualización del nuevo software corregido. Por ello, las pruebas que se realizaron sirvieron para comprobar el correcto funcionamiento del módulo, pero no su óptimo comportamiento en cuanto a consumo de ciclos y memoria. Además de las pruebas, tampoco fue posible, como es lógico, poder calcular el mejor rendimiento de memoria y las correspondientes tablas del estudio.

3.2.8 Energy Dispersal Unscrambling: Dispersión de Energía

3.2.8.1 Introducción

La dispersión de energía es una operación que realiza el complemento selectivo y determinista de los bits de la transmisión, con el fin de reducir la posibilidad de que patrones sistemáticos den como resultado una regularidad no deseada en la señal trasmitida. Para asegurar una dispersión apropiada de la energía de la señal transmitida los bits de entrada del módulo son "aleatorizados" mediante una suma modulo-2 cuyos parámetros son los propios bits de entrada y una secuencia binaria pseudo-aleatoria, PRBS (Pseudo-Random Binary Sequence).

Por consiguiente, es necesario hacer alguna operación para reconstruir la señal original. Esta operación es la misma que se realiza en el transmisor: sumar modulo-2 los bits aleatorizados y la secuencia PRBS. La adición modulo-2 equivale a una XOR a nivel de bit. Esta operación fue seleccionada en el estándar debido a sus propiedades: (a XOR b) XOR b = a.

La secuencia PRBS está definida como la salida de un LFSR (Linear Feedback Shift Register) que utiliza el siguiente polinomio: P(X)= X^9 + X^5 + 1. La entrada de este LFSR es una palabra de 9 bits todos inicializados a 1, llamada GIW (Global Init Word).

3.2.8.2 Desarrollo de la implementación

Se realizaron dos implementaciones para este módulo. La primera de ellas trata un bit en cada palabra de 24 bits de CoolFlux, a la cual se ha llamado "non24packed". Para esta versión se tenía un buffer de entrada, un buffer de salida y un tamaño de secuencia de bits a tratar como parámetros, además de la secuencia PRBS.

En relación con esta última, es necesario decir que se inicializaba en la función initDAB en su máxima extensión posible y se alojaba en la memoria Y. Esto se hacía para evitar generar la misma secuencia una y otra vez cada vez que hiciera falta hacer un Unscrambling. De esta forma, se genera la secuencia una vez y se utiliza todas las veces que sean necesarias. Una función específica se ocupa de generar la secuencia a partir del polinomio antes indicado.

Dado que la implementación anterior es claramente ineficiente (puesto que lo que se pretendía a esta altura de la recepción es tratar la secuencia de bits reconstruida y utilizar 24 bits para tratar uno solo resulta ineficiente) se buscó y logró una implementación mejor, en la cual se empaquetaban 24 bits de la secuencia en una misma palabra de CoolFlux, para la que se realizaba la XOR. Esta nueva versión mejoró el rendimiento del procedimiento de Unscrambling, dado que realiza XOR a nivel de bit (suma módulo-2) sobre 24 bits en cada ciclo, en lugar de hacer un bit por ciclo como hacía en la implementación anterior.

Por otro lado, necesitamos tener los datos empaquetados antes de llamar a este módulo, por lo que se pone la responsabilidad de dicho empaquetado sobre los hombros del módulo previo en el sistema DAB (Decodificador de Viterbi). Además, la generación de la secuencia PRBS tiene un rendimiento peor en la versión empaquetada, puesto que hay que solapar desplazamientos y operaciones XOR. Considerando que esta función está destinada a ejecutarse una sola vez para la recepción, podemos obviar este incremento de coste en ciclos de procesador y concentrarnos en sus ventajas: menos ocupación de la memoria Y (resultado de empaquetar la secuencia PRBS) y ganancia de ciclos en el procedimiento de dispersión de energía. En relación con esto, para observar la ganancia real en ciclos de procesador es necesario hacer una comparación de módulos acoplados (Viterbi – Unscrambler) dado que el Viterbi Decoder debe empaquetar los datos, lo cual supone un incremento del número de ciclos. Para esto es necesario saber que esta versión optimizada del Unscrambler produce 24 bits de salida cada ciclo, por lo que para producir 24 bits necesitamos solo 1 ciclo de ejecución, en contraposición con los 24 ciclos necesarios con la versión no empaquetada.

3.2.8.3 Mejoras realizadas

Las mejoras realizadas para este módulo ya han sido comentadas puesto que se plantearon en etapas iniciales del desarrollo.

En cuanto a las optimizaciones realizadas, cabe destacar el uso de las directivas de compilación chess_prepare_for_pipelining y la de indicación de ausencia de dependencias de datos sobre los buffers, restrict.

3.2.8.4 Rendimiento parcial del módulo y pruebas

La siguiente tabla muestra los datos obtenidos tras la simulación del programa en el simulador de CoolFlux. Se puede apreciar el abrumador incremento de eficiencia en la función de Unscrambling al empaquetar los datos.

Tabla 16: Rendimiento Unscrambling.

Función	Ciclos por llamada	Palabras en PMEM	Palabras en XMEM	Palabras en YMEM
generatePRBS	120008 (43776 bits generados)	16	3	1824
UnScrambling	71 (768 bits reconstruidos)	6	2*[32, 1824] + 4	1824
generatePRBSNon24packed	359335 (20000 bits generados)	13	3	43776
UnScramblingNon24packed	1543 (768 bits unscrambled)	6	2*[768, 43776] +4	43776

3.3 Pruebas realizadas

Las únicas pruebas que se han realizado sobre el funcionamiento acoplado de varios módulos se realizaron a requerimiento de uno de los grupos de desarrollo que estaban trabajando en DAB dentro de NXP. Dicho grupo estaba realizando una aplicación que realizara la acción del canal de demodulación en el receptor de CoolFlux, y lo hacían trabajando en MATLAB.

El canal de demodulación es una estructura que engloba los módulos de FFT, Demodulador Diferencial y Frequency De-Interleaving. Las pruebas solicitadas consistían en el seguimiento de la señal a través del canal de demodulación y su comparación con los resultados obtenidos por dicho equipo de trabajo sobre MATLAB. Estas pruebas eran de interés con el fin de comprobar el nivel de precisión de CoolFlux en el tratamiento de señales en el dominio del tiempo y de la frecuencia. Para su realización fue necesario diseñar funciones que iniciaran el procesamiento de los datos en momentos distintos del canal de demodulación, es decir, invocando a módulos distintos. Las pruebas consistieron tanto en el contraste de los resultados en CoolFlux a la salida de cada módulo como su comparación con los datos proporcionados por el grupo de desarrollo colaborador.

Los resultados obtenidos demostraron, en primer lugar, la correcta sincronización de los módulos que forman el canal de demodulación, y en segundo las buenas prestaciones de CoolFlux en lo que a términos de precisión en el tratamiento de la señal se refiere.

Por tanto, cabe concluir que el resultado fue satisfactorio y que estas pruebas finalizaron con éxito.

Además de lo anteriormente mencionado, un segundo tipo de pruebas fue realizado. Esta vez no se trataba de testear la funcionalidad y correcto desempeño de uno

de los módulos realizados o del sistema integrado. El objetivo de las siguientes acciones de testeo era comprobar la correcta decodificación del FIC realizada por un programa de aproximación que tenían en su poder en la compañía. Para esta prueba hubo que estudiar en profundidad la composición del FIC y realizar una trama a base de bits que fuera descriptiva de un sistema real. Finalmente, la prueba no se pudo realizar por motivos ajenos a nuestra voluntad, pero sí que se llevó a cabo la simulación de la información contenida en el FIC referente a un canal real.

Capítulo 4. Conclusiones

Tal y como comentamos en la introducción de este documento, los objetivos marcados al inicio de la actividad eran diversos. En lo referente a la implementación de la recepción de DAB, el objetivo era obtener una aproximación real de lo que tendría que ser el receptor y poder cotejar su consumo en ciclos y memoria, con el fin de saber si sería necesario escalar varios DSPs para tratar esta recepción de señal.

En los siguientes apartados exploraremos los resultados obtenidos en la implementación y concluiremos con los motivos por los cuales pensamos que los objetivos del proyecto han sido cubiertos.

4.1 Rendimiento

A continuación pasamos a mostrar las cifras de rendimiento definitivas del proyecto realizado. Dichos resultados se calcularon teniendo en consideración el peor de los casos de recepción de DAB, el modo 1, en el cual por cada símbolo OFDM hay más información a tratar, y tomando como referencia el número de datos que habría que tratar por segundo.

Tabla 17: Rendimiento total por módulos.

Función	Ciclos por segundo	Palabras en PMEM	Palabras en XMEM	Palabras en YMEM
Automatic Gain Control	$13 * 10^6$	67	4107	0
FFT	$23 * 10^6$	??	??	??
Differential demodulation	$1,95 * 10^6$	13	4100	2048
Frequency De-Interleaving	$4,2 * 10^6$		4096	3841
Quantization	$6,3 * 10^6$	178	Depende del tamaño del Subcanal	43776
Time De-Interleaving	Depende del tamaño del Subcanal a tratar		5135	256
De-puncturing	$1,8 * 10^6$ (UEP caso peor)	729	26624 (UEP caso peor)	18444 (UEP caso peor)
Decodificador de Viterbi	$1,2 * 10^6$ (UEP caso peor)			
Energy Dispersal Unscrambler	32000 (UEP caso peor)	22	368	368

Si sumamos los requerimientos de ciclos por segundo para el caso peor que se han calculado, el resultado es de 51 * 10^6 , a lo cual habría que añadir el coste en ciclos del Front-End y de los decodificadores del FIC y MSC, además de la estimación para el consumo en ciclos de procesador del Time-Deinterleaving. En total no sumaría más de 100 millones de ciclos de procesador necesarios por segundo de recepción.

En la gráfica de la ilustración 33, observamos el peso relativo que tiene cada módulo realizado dentro del consumo de ciclos total calculado para el sistema desarrollado. La porción correspondiente al time-de-interleaving se ha igualado a la máxima porción calculada, la correspondiente a la FFT, aunque en realidad es muy variable pues depende del subcanal. La conclusión que se saca es que la primera parte del sistema de recepción, el AGC y la FFT, será sin duda la más costosa en tiempo de ejecución.

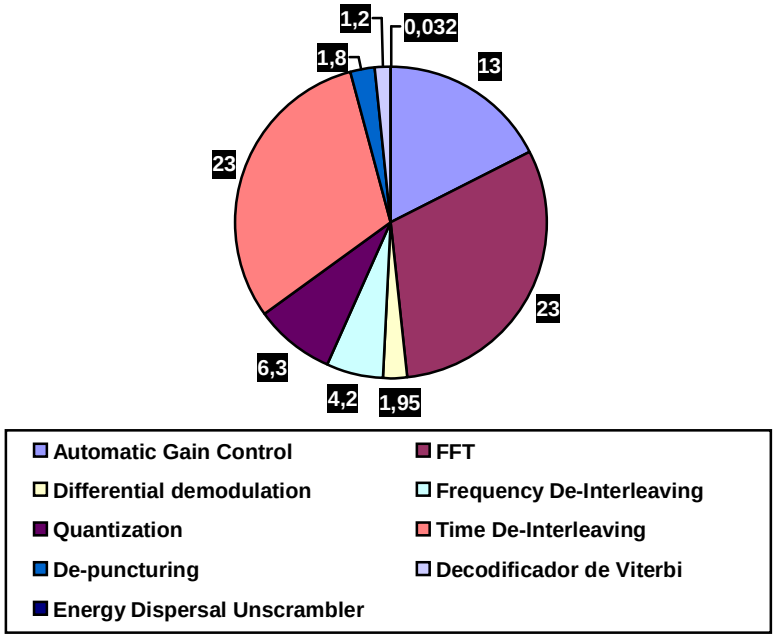


Ilustración 33: Gráfica de ciclos por segundo

4.2 DSPs necesarios

En el apartado anterior se ha revelado la cifra de ciclos de procesador estimada como requerimiento del sistema DAB en un procesador CoolFlux, esto es, próxima a 100 millones de ciclos. Dado que CoolFlux tiene una frecuencia de reloj de 300 MHz, la cual se corresponde con 300 millones de ciclos de procesador por segundo, podemos concluir que DAB podría ser abordable por un solo DSP y no sería necesario escalar varios para el tratamiento del mismo.

4.3 Estudio de viabilidad

Según todo lo anteriormente expuesto, creemos que los objetivos planteados al iniciar el proyecto se han cumplido con creces.

En primer lugar, se ha comprobado la viabilidad de realizar un receptor DAB para correr sobre CoolFlux, por lo que se ha dotado a la compañía de una valiosa piedra de toque a partir de la cual comenzar un diseño definitivo.

En segunda instancia, y en la misma línea, se ha realizado una estimación del precio en ciclos de procesador de un sistema como este, la cual ha resultado ser satisfactoria en los términos inicialmente planteados, puesto que no será necesario escalar varios DSPs para tratar el problema. El estudio de viabilidad ha sido muy positivo.

En tercer y último lugar, durante el desarrollo del proyecto se han encontrado características mejorables en la arquitectura del procesador para la recepción de DAB y que pueden ser beneficiosas en otras aplicaciones del procesador

4.4 Aspectos futuros

En el desarrollo de una aplicación para la recepción de señal digital transmitida mediante un estándar de comunicación tan complejo y variado como es DAB, son muchos los aspectos a tener en cuenta. Por este y por otros motivos, hay aspectos de la recepción de la radio digital que no han sido cubiertos en este proyecto. Por otro lado, como ya se ha comentado, no era el objetivo del mismo la implementación de un receptor completa para dejar listo el producto para el mercado: se trató de un estudio de viabilidad, y por tanto hubo que sacrificar algunas cosas para priorizar otros aspectos. Detallamos estas cuestiones a las que hubo que renunciar en los siguientes apartados.

4.4.1 Fuera de objetivo

Dentro de la implementación básica del receptor DAB, una las cuestiones que no han sido abordadas en este proyecto han sido el muestreo y adquisición de las señales de radio, el llamado Front End. La compañía NXP Semiconductors tiene amplia experiencia en el desarrollo de receptores de señal digital, y por tanto el know-how del que disponen sugería concentrar los esfuerzos en otras materias.

La siguiente cuestión que no forma parte del alcance de este proyecto es el tratamiento de la información una vez que ya se ha reconstruido la secuencia de bits transmitida. En este sentido, después del módulo de dispersión de energía, el Energy Dispersal Unscrambler, se ha procesar la información de distinta manera en función del origen: si se trata del FIC, habrá que decodificar la información que en él se contiene y rellenar con ella las estructuras de datos necesarias para el tratamiento del MSC. En el caso del MSC, la información puede tratarse de señal de audio o datos exclusivamente, o datos y audio de manera simultánea. Esta información ha de ser tratada de manera específica.

En un principio se planteo la opción de trabajar en ellas, siempre supeditado a la evolución de la parte más importante del proyecto, la decodificación de la señal. Sin embargo, la decisión de suprimir estas cuestiones del alcance de nuestro proyecto se tomó al comprobar que no habría tiempo para abordarlas con seguridad.

Además de estas cuestiones, algunas características de DAB no fueron implementadas por tratarse de servicios muy específicos que en ningún caso tendrían una incidencia relevante en la estructura global del receptor, antes bien habría que hacer pequeñas adaptaciones al mismo para incluir estas funcionalidades. Ejemplo de esto es la posibilidad de codificación de la información para que solo los usuarios autorizados puedan recibirla adecuadamente (mediante métodos criptográficos).

4.4.2 Dentro de objetivo

Además de las cuestiones anteriormente abordadas, hay partes de la implementación objetivo que no se han llegado a depurar, testear y validar. Este es el caso del módulo de Decodificación de Viterbi. Para esta decodificación se incluyó hardware específico en el DSP, y las herramientas del entorno de desarrollo aún no estaban actualizadas para incluir esta funcionalidad.

Por otro lado, a falta de tramas de señal de DAB reales, no se ha podido depurar, testear y validar la funcionalidad del sistema de recepción integrado. Dado que cada módulo ha sido testeado y validado independientemente, esta cuestión no es crítica para el futuro desarrollo en la compañía, dado que existe un alto grado de certeza de que el sistema integrado funciona correctamente.

4.5 Dificultades encontradas

En la realización de este proyecto de Sistemas Informáticos han surgido no pocas adversidades que ha sido necesario solventar. En este apartado explicamos brevemente algunas de las más relevantes.

La primera dificultad viene derivada del escaso conocimiento de los alumnos de la materia sobre la cual se ha trabajado. El sistema DAB ha supuesto una fuente importante de problemas durante las primeras etapas del proyecto, así como un importante reto, debido fundamentalmente a la escasa documentación técnica de alto nivel disponible sobre la materia en cuestión.

También ha representado una dificultad añadida la escasa experiencia de los alumnos en lo referente a las cuestiones teóricas y prácticas que engloba el trabajo con un DSP, y en particular el nulo conocimiento del DSP objetivo, CoolFlux, en el momento de iniciar el proyecto. Como consecuencia se hizo necesario dedicar algún tiempo a adquirir conocimientos sobre el funcionamiento de dicho procesador y sus características principales, así como una breve aproximación a otros procesadores existentes en el mercado.

Bibliografía

[1] Wolfgang Hoeg, Thomas Lauterbach; *Digital Audio Broadcasting: Principles and Applications***;** Wiley

[2] ETSI EN 300 401 v1.4.1; *Radio Broadcasting Systems; Digital Audio Broadcasting (DAB) to mobile, portable and fixed receivers;* European Broadcasting Union

[3] S. M. Kuo, W. S. Gan; *Digital Signal Processors: Architectures, Implementations, and Applications;* Prentice Hall 2004

[4] P. Lapsley, J. Bier, A. Shoham, E.A. Lee; *DSP Processor Fundamentals. Architectures and Features;* IEEE Press 1997

[5] Federico J. Barrero García; *Procesadores digitales de señal de altas prestaciones de Texas Instruments;* McGraw-Hill 2005

[6] Hans Roeven, Jeroen Coninx, Marleen Ade; *CoolFlux DSP: The embedded ultra low power C-programmable DSP core;* Intl. Signal Processing Conf. (GSPx), Santa Clara, 2004 © 2004 Global Technology Conferences, Inc.

[7] P. Dytrych, M. Adé, J. Coninx, J. David, P. Vandebroek; *The design of a very low power MP3 decoder accelerator;* Philips PDSL DSP valley, 2002

[8] *CoolFlux BSP Assembly Programmer's Manual,* documentación interna NXP Semiconductors

[9] *CoolFlux BSP Interface Manual,* documentación interna NXP Semiconductors

[10] *CoolFlux BSP: C Programmer's Manual,* documentación interna NXP Semiconductors

[11] *Software Way of Working CoolFlux DSP Software Development,* documento interno NXP Semiconductors

[12] www.coolfluxdsp.com

[13] www.retarget.com, Target Compiler Technologies

[14] Patterson, Hennessey. *Computer organization & design*, Morgan Kaufmann Publishers, 1998

[15] www.**worlddab**.org; The World DAB Forum

[16] http://www.rtve.es/dab/tapi141.pdf

[17] Johan van Ginderdeuren; *Collaboration Makes New Commercial Ultra Low-Power Solution Accessible To Designers;* http://www.synopsys.com/news/pubs/compiler/art2_coolflux-aug04.html

[18] Peter Clarke; *Philips points Target compilation at Coolflux DSP*; Eetimes: http://www.eetimes.com/article/showArticle.jhtml?articleId=22104087