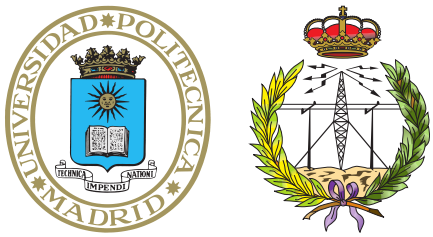
Departamento de Ingeniería Telemática y Electrónica

Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación



Contribuciones metodológicas para el desarrollo de aplicaciones de radio software en arquitecturas multiprocesador heterogéneas

Tesis Doctoral

Autor: **Pedro José Lobo Perea**
Ingeniero de Telecomunicación por la Universidad Politécnica de Madrid

Directores:

César Sanz Álvaro
Doctor Ingeniero de Telecomunicación por la Universidad Politécnica de Madrid

Eduardo Juárez Martínez
Docteur Ès Sciences Techniques por la *École Polytechnique Fédéral de Lausanne*

Diciembre de 2021

TÍTULO:	Contribuciones metodológicas para el desarrollo de aplicaciones de radio software en arquitecturas multiprocesador heterogéneas
AUTOR:	D. Pedro José Lobo Perea
DIRECTORES:	D. César Sanz Álvaro D. Eduardo Juárez Martínez

El Tribunal nombrado con fecha de de por el Mgfco. y Excmo. Sr. Rector de la Universidad Politécnica de Madrid, compuesto por los doctores:

Presidente:	D.
Vocal:	D.
Vocal:	D.
Vocal:	D.
Secretario:	D.
Suplente:	D.
Suplente:	D.

realizado el acto de lectura y defensa de la tesis en la Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación de la Universidad Politécnica de Madrid, acuerda otorgar la calificación de

---

Madrid, a de de

EL PRESIDENTE EL SECRETARIO

LOS VOCALES

AGRADECIMIENTOS

Llegar a este punto no hubiera sido posible sin la contribución, la ayuda y el apoyo de muchas personas, a los que quiero expresar desde aquí mi más profundo agradecimiento.

En primer lugar a mis directores de tesis, César y Eduardo, por toda la ayuda que he recibido y por la paciencia infinita que habéis tenido durante todos estos años. Si esto ha llegado a buen término ha sido sin duda gracias a vosotros.

Al resto de compañeros del Grupo de Diseño Electrónico y Microelectrónico (GDEM), tanto presentes (Matías, Ángel, Fernando, Miguel, Jaime, Juan) como pasados (Miguel Ángel, M.C., Rubén), por crear un entorno en el que es un placer trabajar y por estar siempre dispuestos a echar un capote cuando ha hecho falta.

A los estudiantes que han realizado conmigo su proyecto de fin de estudios, y especialmente a Gonzalo, Daniel y Miguel Ángel (Miky), por vuestra contribución a que esto haya sido posible.

A los compañeros del departamento (SEC, primero, y DTE, después), por el apoyo, los momentos de trabajo y los de no trabajo, que afortunadamente empezamos a recuperar poco a poco.

A Gloria, Miguel y Diego, por los muchos ratos que os he robado.

Y a todos aquellos, demasiado numerosos para contarlos, que en algún momento me han dado una palmadita, una colleja o una voz, en función de la situación, el momento y el estilo personal, para animarme a terminar ya de una vez.

Y para finalizar quiero dedicar este momento a la memoria de mis padres. No habéis llegado a verlo, pero sé cuánto lo habríais disfrutado.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE LISTADOS

RESUMEN

La radio *software* (*Software Radio*) o radio definida por *software* (*Software Defined Radio*, *SDR*) consiste en la implementación de sistemas de radiocomunicación empleando procesadores programables en lugar de *hardware* específico. El primer uso del término data de hace casi cuatro décadas, pero fue en la segunda mitad de la década de los 90 cuando el concepto tomó impulso gracias a que la tecnología disponible en ese momento comenzó a hacer ya posible su uso práctico, y en la actualidad puede afirmarse que todos los dispositivos de radiocomunicaciones emplean esta técnica en mayor o menor medida.

A pesar de sus evidentes ventajas en cuanto a flexibilidad y rapidez de desarrollo frente a los sistemas basados en *hardware* específico, el principal problema de la radio *software* ha estado siempre en sus elevados requerimientos en cuanto a capacidad de cómputo. Aunque el avance continuo de la tecnología permite construir sistemas cada vez más capaces, las prestaciones que se demandan del equipamiento de comunicaciones crecen de manera simultánea, de modo que los sistemas de radio *software* siempre requieren la máxima capacidad de cómputo que la tecnología sea capaz de ofrecer. Esto pasa, desde hace ya casi dos décadas, por el uso de sistemas multiprocesador, debido a la dificultad de mantener el ritmo de mejora en el rendimiento de un único procesador a base de avances en la tecnología de fabricación. A menudo, además, estos sistemas multiprocesador son heterogéneos; es decir, emplean procesadores de varios tipos, con distintas características y capacidades.

El uso de sistemas multiprocesador heterogéneos plantea un nuevo reto: a pesar de que los primeros sistemas con multiprogramación datan de los años 60, el problema de cómo programar de forma eficiente este tipo de sistemas dista mucho de estar resuelto de forma general. Esto es además especialmente evidente en determinados nichos de aplicación, y en el ámbito concreto de la radio *software* las técnicas de programación que han recibido mayor atención y recursos de investigación a nivel general en estas dos últimas décadas no proporcionan resultados satisfactorios o lo hacen a costa de requerir un esfuerzo considerable.

Este es el contexto en el que se desarrolla esta tesis doctoral, cuyo objetivo es desarrollar una metodología de programación para aplicaciones de radio *software* sobre sistemas multiprocesador heterogéneos. Para conseguir este objetivo se han seguido una serie de pasos que se resumen a continuación.

En primer lugar se ha seleccionado una aplicación de referencia lo suficientemente compleja para permitir evaluar los resultados obtenidos. La aplicación seleccionada ha sido un receptor de televisión digital terrestre DVB–T.

A continuación se ha empleado Matlab/Simulink, una herramienta comercial que es prácticamente un estándar *de facto* en el ámbito del procesado de señal, para crear un modelo funcional del receptor DVB–T, y se ha usado después este modelo para implementar el receptor en una plataforma *hardware* específica para radio *software* que combina un DSP y una FPGA. Este proceso ha servido para evaluar las posibilidades que ofrecen varias herramientas de diseño de alto nivel basadas en Simulink.

En paralelo con este trabajo se ha realizado una búsqueda de herramientas que sirvieran como base para atacar el problema que se pretendía resolver en este trabajo de tesis, la programación de aplicaciones de radio *software* en sistemas multiprocesador heterogéneos. Se ha decidido basar el trabajo en GNU Radio, un kit de *software* libre para desarrollo para aplicaciones de radio *software* distribuído con licencia GNU GPL. Aunque en principio es una herramienta pensada para desarrolo en PC, sus características hacen posible adaptarla a plataformas empotradas especializadas, y al distribuirse con licencia GPL la disponibilidad del código fuente para su estudio y modificación es total. El trabajo de modelado realizado con Matlab/Simulink ha servido de base para realizar una segunda implementación del receptor DVB–T basada en GNU Radio.

Tal como se distribuye, GNU Radio puede emplear varios procesadores pero únicamente en sistemas SMP (multiproceso simétrico); por tanto, el siguiente paso ha sido diseñar una extensión de GNU Radio que posibilita su uso en plataformas específicas para radio *software* sacando partido de todos los recursos (procesadores o aceleradores especializados) que puedan ofrecer. Esta extensión se ha diseñado de modo que sea lo más general posible y pueda adaptarse a cualquier plataforma para la que existan unas herramientas mínimas: un compilador de C++ y un mecanismo de comunicación entre procesadores accesible mediante una API POSIX estándar. La metodología propuesta es aplicable en plataformas que tengan aceleradores especializados en funciones específicas o aceleradores programables que puedan implementar una o varias funciones, sin limitaciones a priori en cuanto al número o tipo de aceleradores.

La extensión a GNU Radio se ha implementado en dos plataformas *hardware* distintas. En primer lugar se ha utilizado una plataforma relativamente sencilla basada en un procesador OMAP 3530 de Texas Instruments, que combina un procesador de propósito general y un DSP. Esta primera implementación ha servido como prueba de concepto para validar el diseño de la extensión.

En segundo lugar se ha empleado una plataforma mucho más potente basada en un procesador de la familia KeyStone II del mismo fabricante, que cuenta con cuatro procesadores de propósito general y ocho DSPs. Tras adaptar la extensión de GNU Radio a esta nueva plataforma se ha portado a la misma el receptor DVB–T, trasladando después dos de las tareas con mayor carga computacional (la decodificación convolucional con el algoritmo de Viterbi y el cálculo de la FFT) a los DSPs. Se ha conseguido de este modo una reducción de un 63 % en el tiempo de ejecución del receptor respecto a la versión que emplea únicamente los procesadores de propósito general. Estos resultados confirman que la versión extendida de GNU Radio es capaz de distribuir las tareas de manera eficiente entre los recursos disponibles en la plataforma objetivo.

Finalmente se ha sintetizado todo el trabajo realizado en una propuesta de metodología para el desarrollo de aplicaciones de radio *software* en arquitecturas multiprocesador heterogéneas basada en la extensión de GNU Radio, y se ha creado una implementación de referencia que puede emplearse como base para su uso en nuevas plataformas *hardware*. Esta implementación de referencia está disponible en el servidor Gitlab del Centro de Investigación en Tecnologías Software y Sistemas Multimedia para la Sostenibilidad (CITSEM) de la Universidad Politécnica de Madrid.

ABSTRACT

Software Radio or Software Defined Radio (SDR) is the implementation of radio communication systems using programmable processors instead of dedicated hardware. The first use of the term dates back almost four decades, but it was in the second half of the 1990s that the concept gained momentum as the technology available at that time began to make its practical use possible, and today it can be said that all radio devices employ this technique to a greater or lesser extent.

Despite its obvious advantages in terms of flexibility and speed of development over systems based on dedicated hardware, the main problem with software radio has always been its high computational requirements. Although the continuous advance of technology makes it possible to build ever more capable systems, the performance demanded from communications equipment is also constantly growing, so that radio software systems always require the maximum computational capacity that the technology is able to offer. Since almost two decades now, this has led to the use of multiprocessor systems, due to the difficulty of keeping pace with improvements in the performance of a single processor based on advances in manufacturing technology. In addition, these multiprocessor systems are often heterogeneous, i.e. they use processors of various types, with different characteristics and capabilities.

The use of heterogeneous multiprocessor systems poses a new challenge: although the first systems with multiprogramming date back to the 1960s, the problem of how to efficiently program such systems is far from being generally solved. This is particularly true in certain application niches; for the specific field of radio software, the programming techniques that have received the most attention and research resources at a general level in the last two decades do not provide satisfactory results or do so at the cost of considerable effort.

This is the context in which this Ph.D. thesis is developed, whose objective is to develop a programming methodology for radio software applications on heterogeneous multiprocessor systems. The steps that have been followed in order to achieve this objective are summarised below.

First, a sufficiently complex reference application has been selected to allow the evaluation of the results obtained. The selected application has been a DVB-T digital terrestrial television receiver.

Matlab/Simulink, a commercial tool that is practically a de facto standard in the field of signal processing, has been used to create a functional model of the DVB-T receiver. This model has been used to implement the receiver on a radio-specific

hardware platform combining a DSP and an FPGA. This process has been used to evaluate the possibilities offered by several high-level design tools based on Simulink.

At the same time, a search has been carried out for tools that could be used as a basis to attack the problem that this thesis work was intended to solve, the programming of radio software applications in heterogeneous multiprocessor systems. The choice was GNU Radio, a free software development kit for radio software applications distributed under the GNU GPL licence. Although it is a tool designed for PC development, its characteristics make it possible to adapt it to specialised embedded platforms, and as it is distributed under the GPL licence, the source code is fully available for study and modification. The modelling work carried out with Matlab/-Simulink has served as the basis for a second implementation of the DVB–T receiver based on GNU Radio.

As distributed, GNU Radio can use multiple processors but only on SMP (symmetric multiprocessing) systems; therefore, the next step has been to design an extension to GNU Radio that makes it possible to use it on specific software radio platforms, taking advantage of all the resources (processors or specialised accelerators) that they can offer. This extension has been designed to be as general as possible and can be adapted to any platform for which minimum tools exist: a C++ compiler and an interprocessor communication mechanism accessible through a standard POSIX API. The proposed methodology is applicable to platforms that have accelerators specialised in specific functions or programmable accelerators that can implement one or several functions, without a priori limitations on the number or type of accelerators.

The extension to GNU Radio has been implemented on two different hardware platforms. First, a relatively simple platform based on a Texas Instruments OMAP 3530 processor, which combines a general purpose processor and a DSP, has been used. This first implementation has served as a proof of concept to validate the extension design.

Secondly, a much more powerful platform based on a processor of the KeyStone II family from the same manufacturer, which has four general-purpose processors and eight DSPs, has been used. After adapting the GNU Radio extension to this new platform, the DVB-T receiver has been ported to it, and two of the most computationally intensive tasks (the convolutional decoding with the Viterbi algorithm and the FFT calculation) have been transferred to the DSPs. This has resulted in a 63 % reduction in receiver execution time compared to the version using only the general-purpose processors. These results confirm that the extended version of GNU Radio is able to distribute the tasks efficiently among the resources available on the target platform.

Finally, all the work has been synthesised into a proposed methodology for the development of radio applications on heterogeneous multiprocessor architectures based

on the GNU Radio extension, and a reference implementation has been created that can serve as a basis for use on new hardware platforms. This reference implementation is available on the Gitlab server of the Centro de Investigación en Tecnologías Software y Sistemas Multimedia para la Sostenibilidad (CITSEM) of the Universidad Politécnica de Madrid.

1

INTRODUCCIÓN, OBJETIVOS Y METODOLOGÍA

En este capítulo se introduce la temática y la motivación de la tesis doctoral, se definen los objetivos de la investigación, se explica la metodología seguida durante el trabajo de investigación y se define la estructura del resto de la memoria.

1.1 INTRODUCCIÓN

1.1.1 *La radio software*

El concepto de radio *software* (*Software Radio*) o radio definida por *software* (*Software Defined Radio*, *SDR*) consiste en la implementación de sistemas de radiocomunicación empleando procesadores programables en lugar de *hardware* específico. La figura 1 muestra un diagrama genérico de una radio *software*.

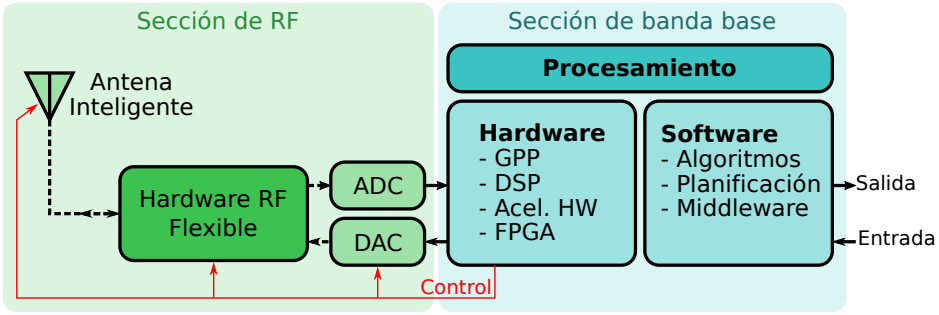


Figura 1: Diagrama genérico de un sistema de radio *software*

La radio *software* lleva siendo objeto de investigación desde hace más de tres décadas [1]. Las ventajas de esta idea respecto a los sistemas realizados con *hardware* específico parecen claras a priori:

- Un mismo equipo *hardware* puede implementar varios protocolos de comunicaciones inalámbricas[1], a veces incluso de forma simultánea.
- Es posible actualizar el equipamiento en servicio ya existente para implementar nuevos protocolos de comunicaciones inalámbricas.

1 En ciertos ámbitos cada procotolo inalámbrico recibe el nombre de *forma de onda*.

- Una implementación basada en *software* suele resultar más rápida y económica que una basada en *hardware* específico, lo que permite reducir tanto ciclos de desarrollo como costes.

A pesar de estas ventajas, la radio *software* tiene un claro inconveniente, que es la enorme capacidad de cómputo que requiere. El demostrador Speakeasy fase I [2], presentado en 1995 y considerado como la primera implementación completa de una radio *software*, necesitaba emplear decenas de procesadores de señal TMS320C40, la punta de la tecnología en aquel momento, y ocupaba un armario de dos metros de altura, todo ello para implementar sistemas que proporcionaban un canal de voz y datos con tasas binarias de algunas decenas de kbps.

Los progresos en la tecnología de fabricación de circuitos electrónicos y en las arquitecturas *hardware* para procesado de señal han aumentado enormemente la capacidad de cómputo disponible desde entonces, pero también lo han hecho las prestaciones demandadas de los sistemas de radiocomunicaciones. Por citar únicamente algunos ejemplos relevantes:

- La telefonía móvil ha pasado de ofrecer tasas binarias en el orden de los 400 kbps usando canales de 200 kHz con una única portadora en los sistemas 2G a mediados de la década de los 90, a tasas del orden de 1 Gbps empleando canales de hasta 100 MHz con 4096 portadoras[2] en los sistemas 5G que se están desplegando actualmente.

- El estándar IEEE 802.11g, publicado en 2003 y conocido actualmente como Wi-Fi 3, permitía obtener una tasa binaria máxima de 54 Mbps empleando 64 portadoras en canales de hasta 20 MHz. El estándar IEEE 802.11ax (WiFi 6), publicado recientemente, permite alcanzar los 10 Gbps empleando 512 portadoras individuales en un ancho de banda de 160 MHz.

- El estándar de televisión digital DVB–T, publicado en 1997, permite transmitir hasta 32 Mbps en un canal de 8 MHz empleando hasta 8192 portadoras. Su sucesor DVB–T2, publicado en 2008, soporta una tasa binaria máxima de unos 50 Mbps en el mismo canal, aumentando el número máximo de portadoras hasta 32768.

2 Todos los sistemas modernos emplean alguna variación de OFDM. En este y los siguientes ejemplos, el número de portadoras indicado corresponde al tamaño de la FFT que es necesario emplear. El número de portadoras útiles suele ser del orden del 75 u 80 % del tamaño de la FFT.

- Las modulaciones empleadas en las portadoras individuales han evolucionado desde 4 o 6 bits por símbolo (16QAM, 64QAM) en los sistemas de principios de siglo a 10 bits por símbolo (1024QAM) en los sistemas actuales[3].

Los algoritmos de corrección de errores empleados en los sistemas mencionados han ido incrementando en general sus requisitos computacionales de forma proporcional a las tasas binarias empleadas, aunque la investigación en este campo ha permitido encontrar algoritmos más eficientes (LDPC, códigos polares combinados con CRC) que los empleados en los sistemas más antiguos mencionados (códigos convolucionales y Turbo) [3, 4].

En resumen, el avance tecnológico ha hecho viable construir sistemas de radio *software* prácticos, pero estos demandan siempre la máxima capacidad computacional que la tecnología pueda proporcionar en cada momento. El lector interesado puede encontrar una extensa introducción a la radio *software* y su estado actual en [5].

1.1.2 *El problema de la arquitectura*

El avance tecnológico al que se ha hecho referencia llegó a un punto de inflexión a mediados de la década de los 2000. Hasta ese momento el incremento en el rendimiento de los procesadores estaba sustentado en gran medida en la conocida ley de Moore [6, 7], que lleva cumpliéndose desde hace más de cincuenta años: el número de componentes (transistores) que pueden introducirse en un circuito integrado se duplica cada dos años aproximadamente. La reducción en tamaño de los transistores implicaba un aumento en la velocidad de funcionamiento, y de ahí buena parte del aumento del rendimiento.

Pero el aumento de la densidad de integración de los transistores y de su frecuencia de funcionamiento implican también un aumento de la potencia disipada por unidad de superficie, y a mediados de los 2000 se llegó a lo que se ha llamado el muro de potencia (*power wall*): la potencia máxima disipable en un circuito integrado por unidad de superficie, que está en el entorno de los 100 W/cm^2.

El efecto de este muro de potencia puede verse claramente en la figura 2, que muestra la evolución de varias características de procesadores de propósito general en las últimas cinco décadas[4]. Mientras que la densidad de integración ha crecido a ritmo

3 Todos los sistemas mencionados pueden emplear varios esquemas de modulación para las portadoras individuales; los esquemas indicados son los más complejos (es decir, con mayor número de bits por símbolo) soportados en cada sistema.

4 Aunque la figura 2 está construida con datos de microprocesadores de propósito general o para sistemas de cálculo científico de alto rendimiento (HPC, *High Performance Computing*), la situación es extrapolable a cualquier otro tipo de sistema basado en procesadores programables.

prácticamente constante durante todo este tiempo (triángulos anaranjados), la frecuencia de funcionamiento (cuadrados verdes) y la potencia consumida (triángulos invertidos marrones) se han estancado desde 2003 aproximadamente. Esta limitación ha reducido notablemente el ritmo de incremento en el rendimiento de un único procesador (círculos azules), que ya solo crece por las mejoras a nivel de arquitectura.

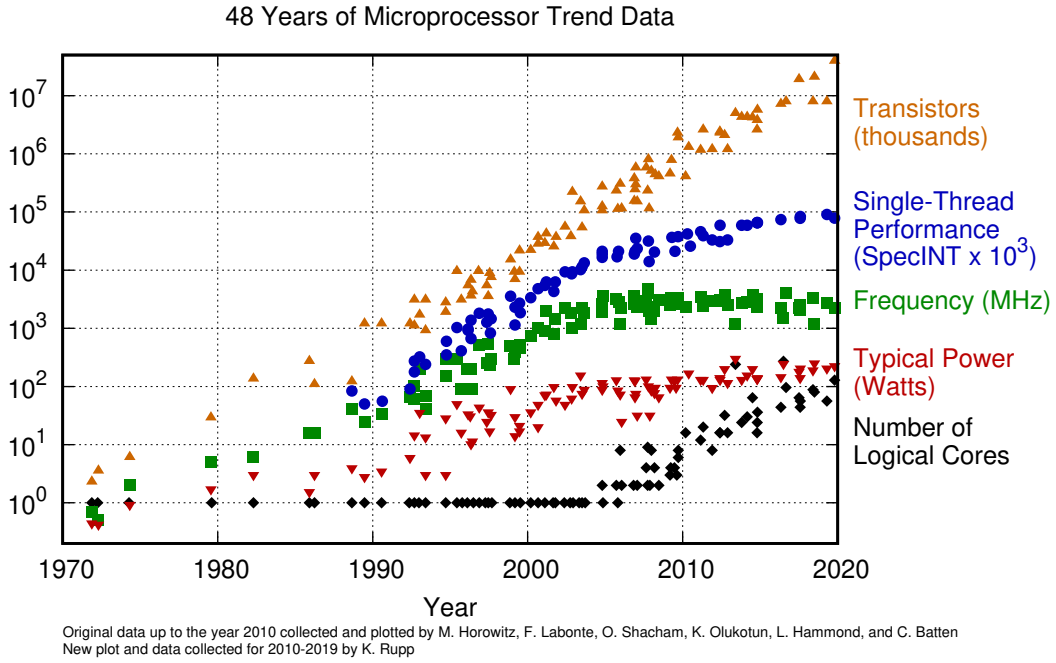


Figura 2: Evolución de los microprocesadores en las últimas cinco décadas [8]

La solución se ha buscado, tal como se muestra en la figura mediante los rombos negros, en los sistemas multiprocesador. Si bien ya no es posible incrementar la frecuencia de funcionamiento, la evolución tecnológica ha hecho posible integrar varios procesadores (que pasan a denominarse núcleos o *cores*) en cada chip, y en la actualidad es muy habitual encontrar en casi cualquier tipo de sistema procesadores con dos, cuatro u ocho núcleos, o incluso varias decenas en sistemas de alto rendimiento.

Otra variante de esta forma de seguir buscando la mejora en el rendimiento que pone especial énfasis en conseguir además la máxima eficiencia energética ha sido el desarrollo de procesadores específicos para una aplicación concreta, conocidos como ASIP (*Application-Specific Instruction-set Processor*). Además del uso de varios núcleos, estos procesadores emplean dos estrategias:

- Añadir a los núcleos, o a algunos de ellos, unidades funcionales específicas para determinadas operaciones. Estos núcleos pueden emplear un juego de

instrucciones (ISA, *Instruction Set Architecture*) propio o, más frecuentemente, se puede hacer sobre un núcleo cuya ISA sea extensible, añadiendo instrucciones que empleen la unidad funcional específica [9, 10].

- Añadir coprocesadores *hardware* especializados en determinadas tareas que resulten especialmente pesadas para los núcleos programables.

En el ámbito específico de la radio *software* los ASIPs pueden tener instrucciones que realicen operaciones aritméticas sobre números complejos o sobre cuerpos de Galois[5] (GF, *Galois Field*), o que permitan intercalar o mezclar bits o bytes en una palabra. El uso de coprocesadores *hardware* especializados contradice los fundamentos de la radio *software*, pero no es extraño encontrarlos como solución de compromiso para determinadas operaciones[6] en sistemas con requisitos de consumo energético estrictos, ya que el *hardware* dedicado siempre es más eficiente en términos energéticos que el programable.

Por otro lado, de forma más o menos simultánea a la generalización de los procesadores con varios núcleos empezó a explorarse el uso de las unidades de procesado de gráficos (GPU, *Graphics Processing Unit*) para computación de propósito general. Los algoritmos empleados en el procesado de gráficos se basan con mucha frecuencia en la repetición de operaciones relativamente simples en múltiples áreas de una imagen de forma independiente; por ello, la arquitectura de las GPUs ha estado basada desde hace mucho tiempo en un número elevado de procesadores sencillos que aplican en paralelo una determinada transformación a las distintas partes de la imagen. De nuevo, la evolución tecnológica ha permitido que estos procesadores crezcan cada vez más en complejidad y en número, y a principios de la década de los 2000 empezó a explorarse la utilización de GPUs para computación de propósito general (GPGPU, *General-Purpose computing on GPU*) [11, 12]. Poco después este tipo de uso de las GPUs comenzó a recibir pleno soporte por parte de sus fabricantes, y en 2007 NVIDIA publicó la primera versión del API CUDA (acrónimo de *Compute Unified Device Architecture*).

En la actualidad el uso de GPGPU está enormemente extendido y pueden proporcionar un rendimiento (tanto bruto como por vatio) notablemente superior al de los procesadores de propósito general, pero solo si la aplicación es adecuada a sus características. Esto implica fundamentalmente dos cosas: que la carga computacional esté asociada mayoritariamente a un único algoritmo, y que dicho algoritmo posea un alto grado de paralelismo a nivel de datos, es decir, que se pueda aplicar de forma

5 Los cuerpos de Galois son estructuras algebraicas de utilidad en áreas como la codificación contra errores o la criptografía.

6 Típicamente el cálculo de la FFT o la decodificación de códigos de protección contra errores.

simultánea e independiente a un número elevado de grupos de datos. Como se verá más adelante, las aplicaciones de radio *software* no suelen cumplir estas características y por tanto no son en general especialmente adecuadas para su implementación mediante GPGPU.

De una u otra manera, desde hace ya años las plataformas que se emplean para la implementación de sistemas de radio *software* están siempre basadas en sistemas multiprocesador, a menudo heterogéneos (es decir, que emplean núcleos procesadores de varios tipos y con diferentes capacidades).

1.1.3 *El problema de la programación*

Como se ha descrito en el apartado anterior, el estancamiento en el aumento de rendimiento debido a factores tecnológicos se palía, al menos en parte, con el incremento en el número de núcleos. Sin embargo esta solución introduce un nuevo problema: la programación de estos sistemas multiprocesador. A pesar de que los primeros sistemas con multiprogramación aparecieron a principios de la década de 1960 [13] y de que este ha sido un campo objeto de intensa investigación desde entonces, el problema de cómo programar sistemas multiprocesador de forma eficiente y razonablemente sencilla dista mucho de estar resuelto de forma general [14], y más aún en el caso de sistemas heterogéneos.

Han existido, y existen aún, decenas de aproximaciones y soluciones a este problema, y la característica común que todas comparten es que ninguna puede atender todas las posibles áreas de aplicación. En [15] se describen algunas propuestas relativamente recientes orientadas al ámbito del procesado de señal e imagen; la mayoría de ellas proporcionan herramientas para aprovechar únicamente el paralelismo a nivel de datos, obviando el paralelismo a nivel de tarea inherente a las aplicaciones de radio *software*, no dan soporte al uso de procesadores especializados, son difícilmente trasladables a sistemas empotrados, son soluciones propias (productos comerciales) que no pueden emplearse en plataformas no soportadas por el fabricante, o una combinación de varios de estos factores. A la hora de implementar una aplicación de radio *software* en una plataforma propia es difícil que alguna de las soluciones descritas resulten útiles sin emplear un esfuerzo considerable.

Por otro lado, existe una tendencia significativa hacia el empleo de APIs estandarizadas para la programación de sistemas heterogéneos, especialmente los que emplean GPGPU y *hardware* programable (FPGAs, *Field Programmable Gate Array*). El uso de estas APIs es útil para tratar de homogeneizar la programación en distintas plataformas y aplicaciones, lo que siempre ayuda a reducir el esfuerzo requerido y permite rentabilizar las habilidades aprendidas [16].

Las APIs estándar más extendidas son OpenMP [17] y OpenCL [18]. La primera se estandarizó en 1997 y nació originalmente para facilitar la programación de sistemas multiprocesador simétricos, aunque en sucesivas revisiones se le ha añadido soporte para el uso de aceleradores *hardware*. La segunda es bastante posterior (su primera versión data de 2009) y tiene como objetivo ser el Java ("*write once, run everywhere*") de los sistemas multiprocesador heterogéneos, soportando en principio todo tipo de configuraciones. A pesar de las indudables ventajas que comportan estas APIs estándar, la realidad es que los resultados que se obtienen dependen fuertemente de la calidad de la implementación que exista para cada plataforma *hardware* y a veces distan de ser óptimos, muy especialmente en sistemas empotrados. En estos últimos, además, es frecuente que el soporte al paralelismo a nivel de tarea, incorporado en revisiones relativamente recientes de ambos estándares, sea deficiente o directamente inexistente [19, 20].

1.1.4 *GNU Radio: una herramienta de código abierto para radio software*

Todo lo expuesto hasta ahora indicaba en el momento de iniciar este trabajo de tesis que existía un problema sin resolver en el ámbito de la programación de sistemas multiprocesador heterogéneos para aplicaciones de radio *software*, especialmente en sistemas empotrados, y por ello se decidió enfocar la investigación en este problema.

Al comenzar el trabajo parecía razonable, antes de comenzar de cero, buscar alguna herramienta o metodología de programación que pudiera emplearse como punto de partida. El abanico de posibilidades no era muy grande, ya que la mayoría de las herramientas existentes presentaban problemas de acceso (bien por ser desarrollos comerciales o simplemente por no estar fácilmente disponibles) o parecían requerir un esfuerzo considerable antes de empezar a obtener resultados.

La solución se encontró en un *kit* de desarrollo para aplicaciones de radio *software* llamado *GNU Radio* [21]. Como su nombre indica, este *software* se distribuye bajo la Licencia Pública General de GNU (*GNU General Public License*) [22], y es por tanto *software* libre y de código abierto, por lo que es plenamente accesible para su estudio y modificación. Al estar orientado precisamente a aplicaciones de radio *software* emplea un modelo de programación basado en flujo de datos que aprovecha de forma natural el paralelismo a nivel de tarea inherente a estos sistemas, y aunque en principio está pensado para el uso en ordenadores típicos de escritorio sus características hacían pensar que podía emplearse sin grandes dificultades en sistemas empotrados.

Por tanto, se decidió emplear este *software* como base de este trabajo de tesis, desarrollando sobre el mismo una metodología de programación de sistemas multiprocesador heterogéneos.

1.2 OBJETIVOS Y METODOLOGÍA

El objetivo principal de este trabajo de tesis es:

Elaborar una metodología de desarrollo para aplicaciones de radio software *sobre plataformas multiprocesador heterogéneas aplicable a sistemas empotrados*.

La metodología empleada en el trabajo de tesis ha sido la siguiente:

- Se ha seleccionado una aplicación de referencia lo suficientemente compleja para permitir evaluar los resultados obtenidos. La aplicación seleccionada ha sido un receptor de televisión digital terrestre DVB–T.

- Se ha empleado una herramienta comercial (Matlab/Simulink) para crear un modelo funcional del receptor DVB–T. Se ha utilizado después este modelo para implementar el receptor en una plataforma específica para radio *software* basada en la combinación de un DSP y una FPGA. La implementación ha servido para evaluar las posibilidades que ofrecen varias herramientas de diseño de alto nivel basadas en Simulink.

- En paralelo se ha realizado una segunda implementación en PC utilizando GNU Radio. Esta implementación ha servido para conocer en profundidad las posibilidades de esta herramienta de *software* libre y llegar a la conclusión de que es posible utilizarla como base para elaborar una metodología de desarrollo de sistemas de radio *software* sobre arquitecturas multiprocesador heterogéneas.

- A continuación se ha diseñado un mecanismo que permite extender GNU Radio para su uso en este tipo de arquitecturas y validar el diseño con una implementación en una arquitectura completa. Para esta implementación se ha seleccionado una tarjeta de desarrollo basada en el procesador OMAP 3530 de Texas Instruments, que combina un procesador de propósito general y un DSP.

- Por último se ha repetido el trabajo de adaptación para la familia de procesadores KeyStone II del mismo fabricante y se ha portado el receptor DVB–T a una plataforma de desarrollo basado en un procesador de esta familia. Esto ha permitido generalizar la metodología empleada en el trabajo y validar el resultado obtenido con ella.

La figura 3 resume de forma gráfica los pasos que se han descrito.

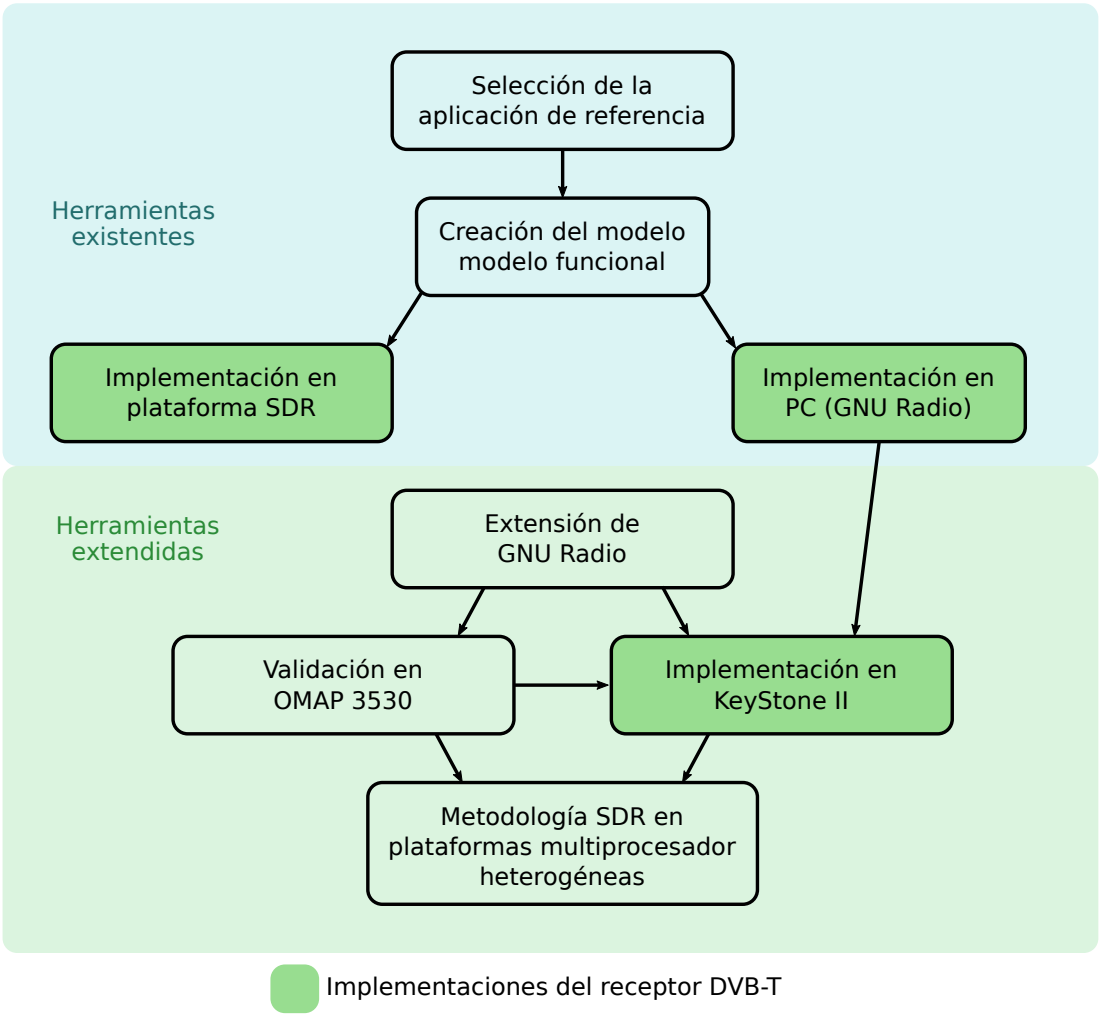


Figura 3: Metodología empleada en el trabajo de tesis

1.3 ORGANIZACIÓN DE LA MEMORIA

La memoria se ha estructurado en siete capítulos. El primer capítulo es esta introducción en la que se justifica la motivación de este trabajo de tesis y se describen sus objetivos y la metodología empleada.

En el capítulo 2 se proporciona información relevante sobre el contexto en el que se ha desarrollado el trabajo de tesis: se describe el estándar de televisión digital terrestre DVB–T, empleado como aplicación de referencia para la evaluación de los resultados obtenidos; se amplía información sobre el concepto de radio *software*, sus

características y sus aplicaciones; y se describe el estado del arte en cuanto a arquitecturas *hardware* para aplicaciones de radio *software* y metodologías de programación empleadas en estas arquitecturas.

En el capítulo 3 se describe el trabajo de modelado e implementación del receptor DVB–T con Matlab/Simulink, se evalúan los resultados obtenidos con las herramientas automáticas de implementación y se describe la implementación del receptor DVB–T con GNU Radio.

En el capítulo 4 se describe el funcionamiento interno de GNU Radio y su extensión para adaptarlo a sistemas multiprocesador heterogéneos que emplean diversos procesadores como aceleradores para algunas funciones. La extensión se describe de forma genérica, de modo que sea válida para cualquier plataforma.

En el capítulo 5 se describe cómo se ha implementado la extensión propuesta en dos plataformas basadas, respectivamente, en procesadores OMAP 3530, que incluye un núcleo de propósito general ARM Cortex A8 y un núcleo DSP C64x+, y KeyStone II, que incluye cuatro núcleos de propósito general ARM Cortex A17 y ocho núcleos DSP C66. La implementación en OMAP 3530 sirve como prueba de concepto para validar el mecanismo de extensión propuesto. La implementación en KeyStone II incluye, además de la extensión de GNU Radio, el receptor DVB–T y su modificación para que haga uso de los núcleos DSP.

En el capítulo 6 se resume la metodología de programación de sistemas multiprocesador heterogéneos que se propone como resultado principal de este trabajo de tesis, incluyendo las directrices para poder aplicarla a otras plataformas de desarrollo.

Finalmente, en el capítulo 7 se resumen las aportaciones realizadas en la tesis, incluyendo las publicaciones y comunicaciones a congresos que se han llevado a cabo durante su desarrollo, y se proponen algunas ideas de continuación de la línea de trabajo de la tesis.

La memoria concluye con la relación de referencias bibliográficas que se han consultado durante el desarrollo de la tesis.

2

CONTEXTO Y ESTADO DEL ARTE

2.1 INTRODUCCIÓN

En este capítulo se describen brevemente las tecnologías y la aplicación que han sido utilizadas en el desarrollo de esta tesis y que permiten contextualizar la investigación llevada a cabo. A la vez que se realiza una descripción de las mismas, con objeto de que la memoria de tesis resulte autocontenida, se establece también cuál es el estado del arte.

En el apartado 2.2 se describe el estándar de televisión digital terrestre DVB–T, cuyo receptor ha sido utilizado en esta tesis como aplicación de referencia para validar la metodología de diseño sobre arquitecturas multiprocesador heterogéneas. Se ha elegido esta aplicación por su elevada complejidad y por la experiencia previa del grupo de investigación en el que se ha desarrollado la tesis en los sistemas de codificación/decodificación de televisión digital. Sin embargo, con el trabajo de esta tesis se ha pretendido avanzar más en la integración del sistema incluyendo, en lo posible, el receptor de radio.

Para la integración del receptor de radio, resulta interesante aplicar el concepto de *radio software* ya presentado en el capítulo anterior y que se describe con más detalle en el apartado 2.3. La materialización de sistemas de radiocomunicación sobre procesadores programables, empleando técnicas de tratamiento digital de la señal, se viene utilizando desde hace ya algunos años, si bien, el uso de arquitecturas multiprocesador heterogéneas como tecnología de soporte para estas implementaciones no es tan frecuente y, además, su diseño plantea unos retos metodológicos no triviales, a los que esta tesis aporta contribuciones.

En el apartado 2.4 se describen diversas arquitecturas comerciales o propuestas en la literatura científica, que pueden servir como soporte de las aplicaciones de *radio software*. Se describen arquitecturas convencionales, procesadores especializados, procesadores reconfigurables y los entornos de desarrollo asociados. Finalmente se describen arquitecturas multiprocesador heterogéneas en las que la coexistencia de uno o varios núcleos de procesadores de propósito general y uno o varios núcleos de procesadores digitales de señal, se perfilan como la alternativa más flexible para este tipo de aplicaciones en las que los estándares evolucionan de forma muy rápida. Sin

embargo, este tipo de arquitecturas presentan el inconveniente, ya mencionado en el párrafo anterior, del coste del desarrollo sobre las mismas.

El capítulo concluye con el apartado 2.5, en el que se presentan algunas metodologías de diseño sobre arquitecturas multiprocesador heterogéneas, área en la que este trabajo de tesis realiza contribuciones.

2.2 EL ESTÁNDAR DE TELEVISIÓN DIGITAL TERRESTRE DVB–T

En el inicio de este trabajo se decidió elegir un estándar de comunicaciones inalámbricas concreto para poder comparar los resultados que se fueran obteniendo en un marco común. Se decidió emplear DVB–T y su extensión para recepción en movilidad DVB–H como aplicación de referencia.

Desde entonces DVB–H, como se verá más adelante, ha perdido buena parte del interés que despertó inicialmente, y DVB–T está en proceso de ser sustituido por su sucesor DVB–T2. No obstante, la elección realizada sigue siendo válida para el fin que se perseguía, por lo que todo el trabajo realizado se ha mantenido orientado al estándar DVB–T. A fin de facilitar la descripción de las diversas implementaciones que se han ido realizando, a continuación se presenta una introducción a las características más relevantes de dicho estándar.

2.2.1 *Introducción a la familia de estándares DVB*

DVB (*Digital Video Broadcasting*) es una familia de estándares de origen europeo para televisión digital mantenidos por el Proyecto DVB [23], un consorcio formado por más de 180 empresas y organismos públicos, y publicados por ETSI (*European Telecommunications Standards Institute*).

La familia de estándares DVB está compuesta por más de cien normas, pero sin duda las que pueden considerarse principales son las que definen la transmisión de televisión digital por satélite (DVB–S), por cable (DVB–C) y terrestre (DVB–T). Los estándares DVB–S y DVB–C fueron publicados en 1994, seguidos tres años más tarde por DVB–T en 1997. Desde entonces DVB–T ha sido desplegado en más de 70 países en todo el mundo. En España, por ejemplo, las primeras licencias de emisión con este sistema se otorgaron en 1999 y en 2002 comenzó la emisión regular en *simulcast* (es decir, retransmitiendo el mismo contenido) de las cadenas de TV analógica con cobertura nacional.

Simultáneamente al desarrollo de DVB, la "Grand Alliance" estadounidense (un consorcio similar al Proyecto DVB) desarrolló el sistema ATSC (*Advanced Television Systems Committee*), y Japón hizo lo propio con el sistema ISDB (*Integrated Services*

Digital Broadcasting). Estos tres sistemas (DVB, ATSC e ISDB) son los empleados en prácticamente todo el mundo para la transmisión de televisión digital. Todos ellos utilizan MPEG–2 como sistema de codificación de fuente para audio y vídeo [24] y como trama de sistema [25]. Algunos años más tarde China desarrolló su propio sistema, llamado DTMB (*Digital Terrestrial Multimedia Broadcast*).

Durante la década de los 2000 el constante incremento de las capacidades de los terminales de telefonía móvil hizo pensar en la difusión de servicios audiovisuales a este tipo de dispositivos. Las redes de telefonía móvil 3G que comenzaban a desplegarse en esa época eran poco adecuadas para este tipo de servicios, lo que propició la aparición de nuevos sistemas de transmisión. Dentro de la familia DVB se publicaron los estándares DVB–H en 2004, orientado a la difusión terrestre, y DVB–SH en 2007, para la difusión vía satélite.

El estándar DVB–H, que es fundamentalmente una extensión de DVB–T para mejorar la recepción en condiciones de movilidad y disminuir el consumo de energía en el receptor, fue empleado en varias pruebas de campo, especialmente durante el campeonato mundial de fútbol que se celebró en Alemania en 2006. En ese mismo año varios operadores comenzaron a comercializar servicios basados en dicho sistema, tanto en Europa como en otros países. Sin embargo, al menos en Europa, el sistema ha sido un fracaso comercial y la mayoría de los operadores han cancelado sus servicios entre los años 2010 y 2012. Esto, unido a la evolución que entre tanto han experimentado los sistemas de telefonía móvil, con 4G completamente implantada y servicios 5G ya disponibles en grandes ciudades, hace muy improbable que el estándar DVB–H vuelva a tener alguna oportunidad de adopción significativa.

Mientras tanto el estándar DVB–T ya estaba plenamente establecido tanto en Europa como en múltiples países de todo el mundo. Entre los años 2006 y 2012 fueron cesando las emisiones de televisión analógica en toda Europa, un proceso que se conoció como el "apagón analógico" y que en España se produjo en abril de 2010.

Más o menos en ese período comenzaron también las emisiones comerciales de televisión en alta definición. En 2004 comienza a operar el canal HD1, emitido a través de los satélites Astra usando DVB–S. En 2008 comienza la emisión regular de TV en alta definición terrestre, usando DVB–T, en Italia y Francia.

Las emisiones en alta definición multiplican por cuatro o cinco la tasa binaria requerida por cada servicio. Este impacto fue mitigado en parte cambiando el estándar de compresión de vídeo de MPEG–2 a MPEG–4 o MPEG–4/AVC en las emisiones en HD, pero simultáneamente se trabajó en la evolución del estándar DVB–T para aumentar la capacidad de cada canal. En 2009 se publicó el estándar DVB–T2, y en ese mismo año comenzaron las emisiones en alta definición en Reino Unido empleando este nuevo estándar. Actualmente más de 30 países han comenzado el despliegue de

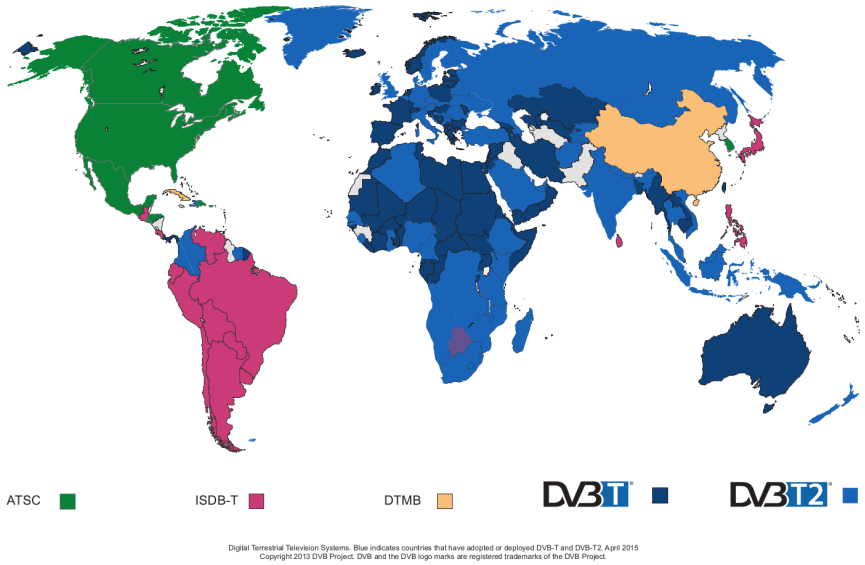


Figura 4: Mapa de adopción de estándares de televisión digital terrestre [26]

servicios de televisión digital terrestre basados en DVB–T2, y otros 40 lo han adoptado. En total, 150 países en todo el mundo han desplegado o adoptado DVB–T o T2 como sistema de televisión digital terrestre (figura 4).

2.2.2 *Características de DVB–T*

Muchas de las características de DVB–T están heredadas de los estándares que le precedieron, DVB–S y DVB–C. Algunos de los requisitos más significativos que se definieron para el sistema previamente a su diseño fueron los siguientes [27]:

- El sistema debía ser tan similar como sea posible a los sistemas de transmisión via satélite y cable.
- El sistema debía proporcionar un área de cobertura óptima en recepción estacionaria con una antena de tejado. Era deseable la recepción estacionaria con receptores portables, pero la recepción móvil no era un objetivo de diseño.
- Debía ser posible la transmisión en redes terrenas de frecuencia única (SFN, *Single Frequency Network*).
- El estándar debía asegurar que con la tecnología disponible en el año 1997 fuera posible producir un receptor doméstico con un precio razonable.

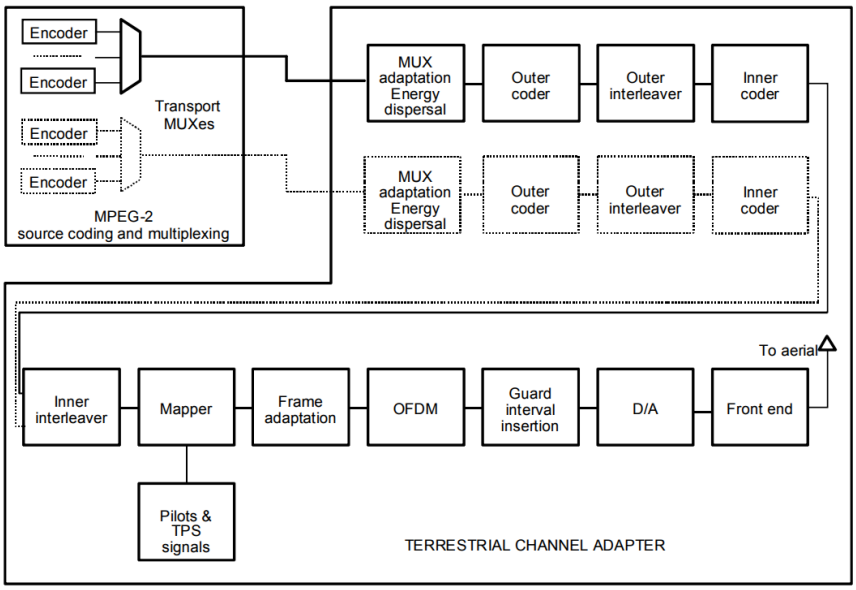


Figura 5: Diagrama de bloques funcional de un transmisor DVB–T [28]

- El sistema debía ofrecer la posibilidad de emplear *transmisión jerárquica*.

Las características técnicas de DVB–T están determinadas en gran medida por estos requisitos. Por ejemplo, el requisito de posibilitar las SFN determinó el uso de OFDM (*Orthogonal Frequency Division Multiplex*) como técnica de modulación, y el esquema de protección contra errores mediante códigos encadenados que se describirá más adelante es el mismo que se diseñó para DVB–S.

La figura 5 muestra un esquema funcional de un transmisor DVB–T. La entrada es un flujo de transporte (*Transport Stream*, TS) MPEG–2 que contiene todos los servicios que se desean transmitir, o dos TS si se emplea transmisión jerárquica. Esta técnica permite contrarrestar la brusca degradación de la calidad de recepción al disminuir la calidad de la señal recibida que caracteriza a los sistemas de transmisión digital, frente a la degradación gradual que experimentan los sistemas analógicos en condiciones similares.

La transmisión jerárquica permite transmitir simultáneamente dos canales, uno de alta prioridad que puede decodificarse aunque la señal recibida sea débil a costa de una capacidad relativamente reducida, y otro de baja prioridad con mayor capacidad pero que precisa de una mejor calidad de recepción. Por cada canal se transmite un TS distinto que puede contener, por ejemplo, dos versiones con distinta calidad del

mismo programa. La vía de procesado del canal de baja prioridad aparece punteada en la figura 5.

A continuación se ofrece una breve descripción de cada paso del procesado que efectúa el transmisor antes de emitir la señal de radio.

2.2.2.1 *Adaptación del múltiplex y dispersión de energía*

El TS está compuesto por paquetes de longitud fija (188 bytes) que comienzan con un byte de sincronización fijo, 47_{HEX}. Este módulo genera una secuencia pseudoaleatoria de bits que se combinan mediante una operación lógica XOR con los bits del TS, de modo que se aleatoriza la secuencia de entrada evitando la aparición de patrones repetitivos. Estos patrones tendrían más adelante el efecto de que la energía de la señal transmitida no se distribuyera de manera uniforme por el canal de transmisión, lo que podría tener consecuencias indeseadas [27, p. 191] que no corresponde detallar aquí.

Los paquetes de transporte se agrupan de ocho en ocho. Los bytes de sincronización de cada paquete se dejan inalterados (no se combinan con la secuencia pseudoaleatoria) y el primer byte de cada grupo de ocho se invierte bit a bit, lo que permite recuperar en el receptor la sincronización a nivel de TS.

2.2.2.2 *Codificación*

El flujo de transporte aleatorizado se protege frente a errores en la transmisión mediante la aplicación de dos códigos encadenados separados por una operación de entrelazado a nivel de byte. Esta forma de combinar dos códigos mejora enormemente la capacidad de corrección frente a la que se obtendría aplicando cualquiera de ellos por separado [29]. El entrelazado aumenta además la resistencia del código frente a errores producidos en ráfaga.

2.2.2.2.1 CODIFICACIÓN EXTERNA Cada paquete de transporte se codifica mediante un código Reed Solomon acortado RS(204,188,8)[1] derivado del código RS(255,239, 8). Este código añade 16 bytes a cada paquete de transporte, que pasa así de 188 a 204 bytes.

Esta codificación permite corregir cualquier error que afecte a un máximo de 8 bytes cualesquiera de un paquete de transporte.

1 Un código RS(n,k,t) toma como entrada palabras de k símbolos y genera palabras codificadas de n símbolos, y puede corregir errores en t símbolos en la palabra codificada [30]. Si el código opera sobre $GF(2^8)$, como es el caso, cada símbolo está compuesto por 8 bits (1 byte).

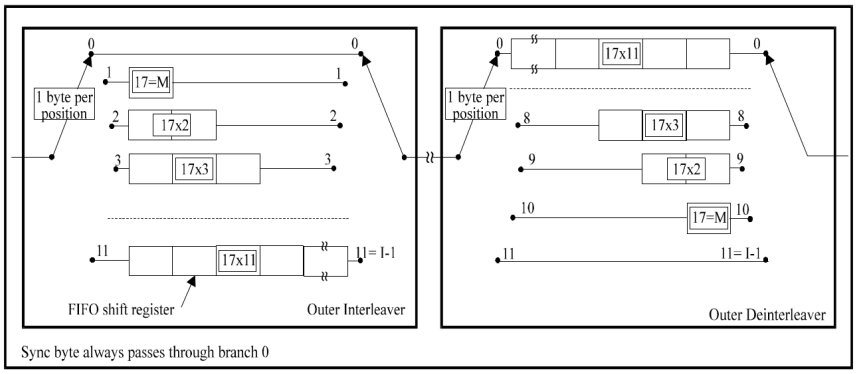


Figura 6: Entrelazado y desentrelazado externo en un transmisor DVB–T [28]

2.2.2.2.2ENTRELAZADO EXTERNO El flujo de paquetes de transporte codificados se introduce en un entrelazador convolucional de Forney [31] con I = 12 y M = 17: los bytes de entrada se van distribuyendo cíclicamente por doce ramas, cada una de las cuales tiene una FIFO con una longitud que va creciendo en múltiplos de diecisiete bytes. La figura 6 muestra tanto el proceso de entrelazado como el de desentrelazado.

Este proceso de entrelazado hace que los bytes que a la salida del entrelazador son contiguos pertenezcan en realidad a doce paquetes distintos o, visto de otro modo, hace que los 204 bytes de un paquete de transporte codificado se distribuyan a lo largo de 2244 bytes ($I \times (I-1) \times M$) a la salida del entrelazador. Esto aumenta notablemente la resistencia del código ante errores en ráfaga.

La estructura del entrelazador hace que a su salida se mantenga el patrón de sincronización de los paquetes de transporte: un byte de sincronización (47_{HEX}) cada 204 bytes con uno de cada ocho bytes de sincronización invertido.

2.2.2.2.3CODIFICACIÓN INTERNA A la salida del entrelazador se aplica un código convolucional[2] de tasa 1/2 con 64 estados ($K = 7$) cuyos polinomios generadores son $G_1 = 171_{OCT}$ y $G_2 = 133_{OCT}$ (figura 7).

A este código se le puede aplicar opcionalmente uno de entre cuatro patrones de punteado, lo que permite obtener códigos de tasa 1/2 (sin punteado), 2/3, 3/4, 5/6 y 7/8. Esto permite seleccionar el mejor compromiso para cada red concreta entre robustez y eficiencia del código.

2 Un código convolucional es un código lineal en el que la matriz generadora tiene una estructura tal que la operación de codificación puede verse también como una operación de filtrado o convolución [30]. La secuencia codificada puede así obtenerse a partir de las salidas intercaladas de uno o más filtros digitales que reciben como entrada la secuencia a codificar.

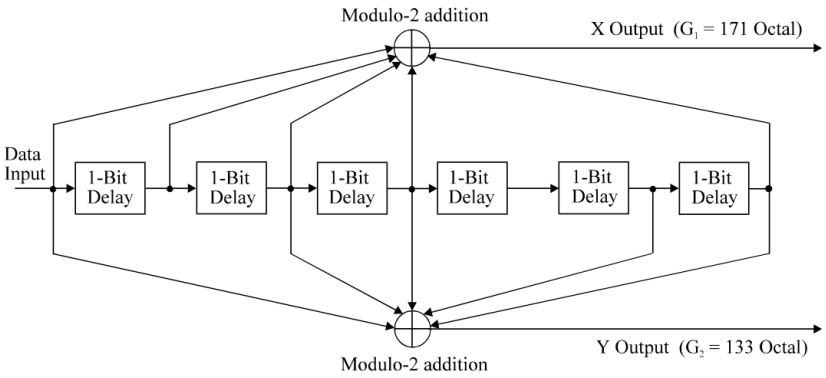


Figura 7: Codificador convolucional de DVB–T [28]

2.2.2.3 *Entrelazado interno*

A la salida de la cadena de aleatorización y codificación que se acaba de describir se obtiene un flujo de bits, o dos si se emplea transmisión jerárquica ya que en este caso cada uno de los canales tiene su propia cadena de aleatorización y codificación (véase la figura 5). El entrelazado interno combina estos dos flujos de bits (si se da el caso) en uno solo y los reordena de modo que se maximice la probabilidad de que un posible error de demodulación en una portadora OFDM individual se traduzca en un error en un único bit [27, p. 247].

La salida del entrelazador interno está compuesta por grupos de dos, cuatro o seis bits, en función de la modulación que se escoja para las portadoras de datos individuales. Existen por tanto cinco configuraciones distintas del entrelazador interno: modulación no jerárquica modulando las portadoras individuales con QPSK, 16QAM o 64QAM, y transmisión jerárquica modulando las portadoras individuales con 16QAM o 64QAM.

2.2.2.4 *Mapeado de símbolos*

A partir de la salida del entrelazador interno y de los parámetros de configuración del transmisor se generan los símbolos que se van a transmitir en cada una de las portadoras individuales que componen el símbolo OFDM. El estándar define dos posibles modos de transmisión: 2K, en el que se emplean 1705 portadoras individuales, y 8K, en el que se emplean 6817 portadoras. Estas portadoras pueden ser de tres tipos:

1. Portadoras de datos.

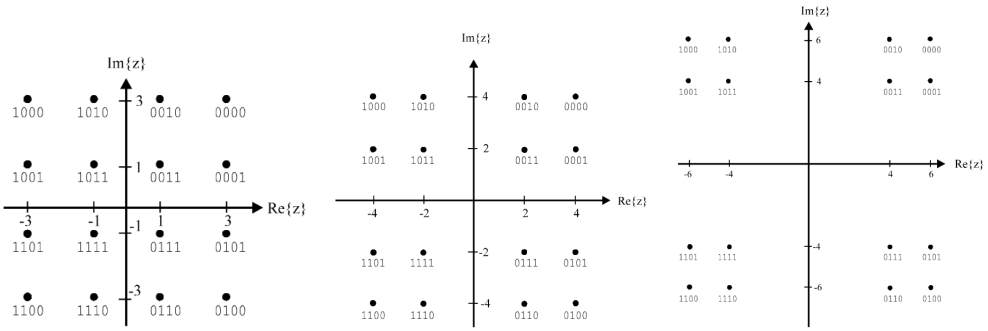


Figura 8: Constelaciones 16QAM en DVB–T con transmisión jerárquica para $\alpha = 1$, $\alpha = 2$ y $\alpha = 4$ [28]

2. Pilotos continuos y dispersos.

3. Pilotos TPS (*Transmission Parameter Signalling*).

2.2.2.4.1PORTADORAS DE DATOS Las portadoras de datos transmiten los símbolos generados a partir de los bits obtenidos a la salida del entrelazador interno, que a su vez corresponden (tras los procesos descritos de aleatorización y codificación) a la información del flujo o flujos de transporte MPEG–2 que constituyen la carga útil de la señal.

Como ya se dijo anteriormente las portadoras de datos se pueden modular con QPSK, 16QAM o 64QAM, transmitiendo dos, cuatro o seis bits por símbolo, respectivamente. Si se emplea transmisión jerárquica sólo se pueden emplear las dos modulaciones superiores, pero además es posible seleccionar entre tres constelaciones posibles para cada modulación mediante un parámetro denominado *ratio de constelación* (α) que puede tomar los valores uno, dos o cuatro. Esto arroja un total de siete posibles constelaciones, ya que las constelaciones 16QAM y 64QAM son iguales para transmisión jerárquica con $\alpha = 1$ y para transmisión no jerárquica. La figura 8 muestra las constelaciones 16QAM para los tres posibles valores de α.

El número de portadoras de datos en cada símbolo OFDM es 1 512 en modo 2K y 6 048 en modo 8K.

2.2.2.4.2PILOTOS CONTINUOS Y DISPERSOS Además de las portadoras de datos, cada símbolo OFDM incluye una serie de portadoras que se modulan con BPSK y transmiten un valor conocido que se obtiene de una secuencia pseudoaleatoria. Estas portadoras reciben el nombre de pilotos y sirven para facilitar la sincronización en el

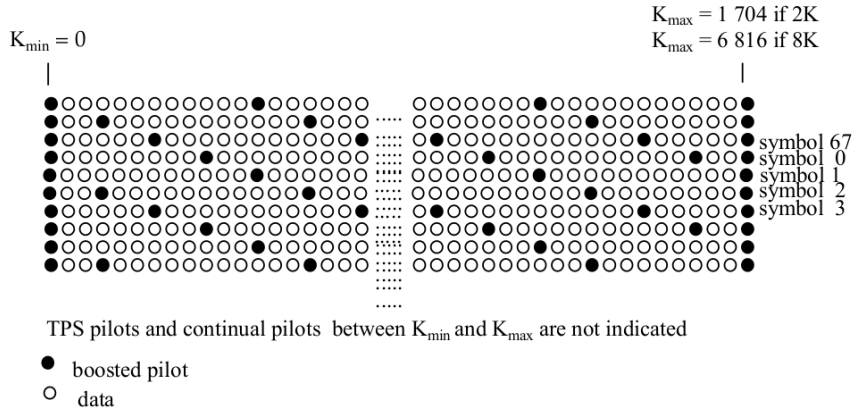


Figura 9: Distribución de los pilotos dispersos en los símbolos OFDM en DVB–T [28]

receptor. Se transmiten con una potencia ligeramente superior a la de las portadoras de datos. Hay dos tipos de pilotos: continuos y dispersos.

Pilotos continuos Los pilotos continuos reciben este nombre porque están presentes siempre en las mismas portadoras en todos los símbolos OFDM. En modo 2K se transmiten 45 pilotos continuos, y en modo 8K se transmiten 177.

Pilotos dispersos Los pilotos dispersos varían su posición según un patrón cíclico que se repite cada cuatro símbolos OFDM según se puede ver en la figura 9.

El objetivo de los pilotos dispersos es doble: por un lado facilitan aún más la sincronización del receptor, y por otro permiten que el receptor pueda evaluar de forma continua en el tiempo el estado del canal de transmisión (*estimación de canal*), lo que le permitirá corregir muchas de las alteraciones que éste haya introducido en la señal recibida. En cada símbolo OFDM hay cierto número de pilotos dispersos que coinciden con la posición de un piloto continuo, pero la distribución de unos y otros está calculada de modo que en cada símbolo OFDM hay 176 pilotos continuos o dispersos en modo 2K y 704 en modo 8K.

Pilotos TPS La señal transmitida se organiza en tramas OFDM, cada una de las cuales está compuesta por 68 símbolos OFDM consecutivos. En cada uno de los símbolos OFDM de una trama se transmite un bit de lo que el estándar DVB–T denomina el "bloque TPS" (de *Transmission Parameter Signalling*), que es un bloque de 68 bits que contiene un identificador del transmisor de 16 bits e información sobre los parámetros que está empleando el transmisor: modo, constelación empleada en

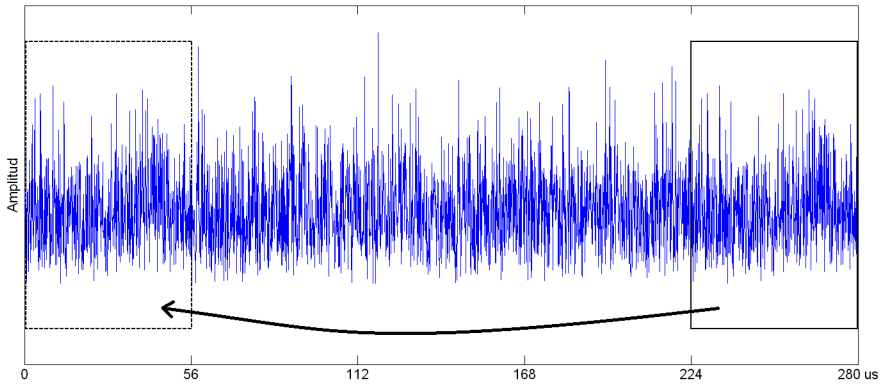


Figura 10: Ejemplo de símbolo OFDM DVB–T con una duración útil de 224 μs y longitud del prefijo cíclico 1/4

las portadoras de datos, tasa del código convolucional, etc. La información contenida en el bloque TPS está además protegida mediante un código BCH(67,53,t=2).

En cada símbolo OFDM se transmite un bit del bloque TPS, que se emplea para modular en DBPSK 17 portadoras en modo 2K y 68 portadoras en modo 8K. Es decir, cada bit está replicado en 17 ó 68 portadoras.

2.2.2.5 *Modulación OFDM*

Los símbolos que se transmiten en cada una de las portadoras individuales constituyen los coeficientes de la transformada discreta de Fourier del símbolo OFDM. La modulación OFDM consiste en calcular la transformada inversa, con lo que se obtienen las muestras en el dominio del tiempo del símbolo.

Tras este cálculo se añade el *prefijo cíclico*: una parte del final del símbolo OFDM se replica al principio del mismo (figura 10). La adición del prefijo cíclico tiene dos efectos fundamentales: reduce en gran medida las consecuencias de la interferencia entre símbolos (ISI, *Inter Symbol Interference*) provocados por los ecos de la señal debidos a la propagación multitrayecto característica de los entornos urbanos, y permite la construcción de redes de frecuencia única (SFN) que, como se recordará, era uno de los requisitos de diseño de DVB–T.

2.2.2.6 *Conversión D/A y transmisión*

Por último, los sucesivos símbolos OFDM generados se concatenan, se realiza la conversión D/A, se amplifica la señal al nivel de potencia requerido y se radia. La

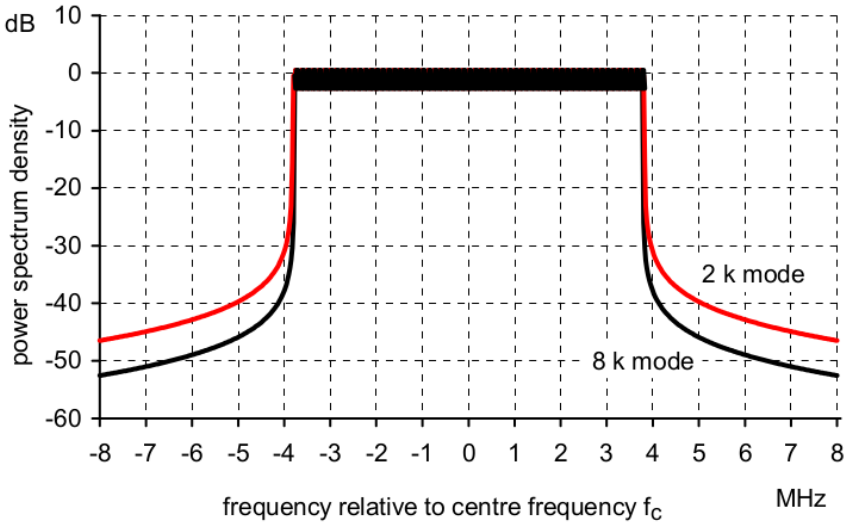


Figura 11: Densidad espectral de potencia de la señal DVB–T para un canal de 8 MHz [28]

figura 11 muestra la densidad espectral de potencia de la señal radiada para un ancho de banda de 8 MHz.

2.3 RADIO SOFTWARE

El término *radio software* fue utilizado por primera vez en la literatura científica y técnica por Joseph Mitola III a principios de los años 90 [32] para designar sistemas de radiocomunicaciones que en lugar de emplear *hardware* específico para procesar la señal (mezclar, filtrar, codificar, etc.) implementan las funciones necesarias en *hardware* programable empleando técnicas de procesado digital de señales, lo que permite que un determinado *hardware* pueda ser reconfigurado para realizar funciones distintas en cada momento.

No hay una definición exacta de qué es un sistema de radio *software* y ni siquiera existe un consenso generalizado acerca del grado de reconfigurabilidad que permite catalogar un determinado sistema como radio *software*. Una definición práctica podría ser que un sistema de radio *software* es aquel que está definido en buena parte mediante *software* y cuyo funcionamiento a nivel físico puede ser modificado de forma significativa mediante cambios en dicho *software* [33].

La implementación de un sistema de radio *software* ideal requeriría, o bien digitalizar la señal en la misma antena, o bien diseñar un frontal de radiofrecuencia con

una configurabilidad absoluta. Ambos enfoques son inviables en la práctica tanto por limitaciones tecnológicas como económicas. Los sistemas prácticos de radio *software* siguen el modelo que se mostró en la figura 1 en el capítulo de introducción, repetida aquí para mayor comodidad.

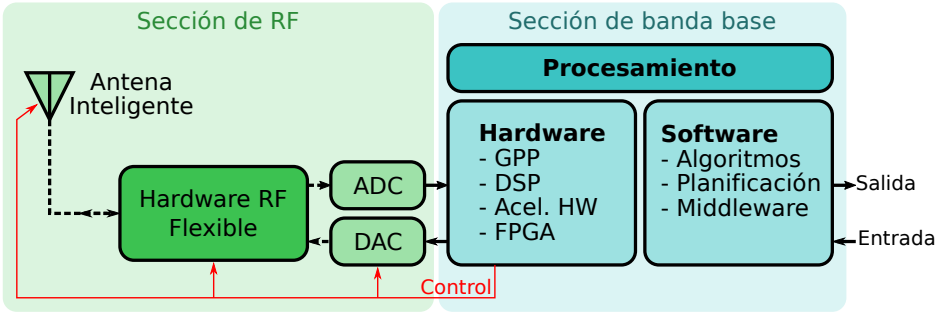


Figura 1: Diagrama genérico de un sistema de radio *software* (repetición de la figura mostrada en la página 1)

2.3.1 *Características de la radio software*

En general los sistemas de radio *software* cuentan a priori con una serie de características propias que los distinguen de los sistemas basados en *hardware* específico. La característica definitoria y fundamental es, como ya se ha dicho, la reprogramabilidad: un sistema de radio *software* puede cambiar en principio cualquiera de los parámetros que definen la comunicación (ancho de banda, modulación, codificación, protocolos...), o todos ellos (que forman en conjunto lo que en la terminología de este campo suele llamarse una *forma de onda*), sin más que reprogramar el equipo.

De esto se derivan otras características, algunas de ellas ventajas y otras inconvenientes, aunque a menudo una característica puede categorizarse de uno u otro modo dependiendo de la aplicación concreta para la que se esté diseñando el sistema de radio. A continuación se enumeran las que se consideran más relevantes.

1. Interoperabilidad. La posibilidad de variar la forma de onda hace que un determinado equipo pueda en principio interoperar con cualquier otro.

2. Eficiencia y adaptabilidad en el uso de recursos. Es posible optimizar el uso de un determinado recurso que puede cambiar de un instante a otro (energía en un momento, ancho de banda en otro, velocidad de transmisión en otro...). Llevado al extremo esta capacidad lleva a lo que se conoce como *radio cognitiva*, que se describe más adelante.

3. Complejidad y facilidad de desarrollo. La posibilidad de reutilizar el *hardware* hace que desarrollar una nueva forma de onda requiera únicamente un desarrollo *software*, lo que en principio debe acortar el ciclo de diseño y reducir los costes de desarrollo. En la práctica esto no siempre es así, ya que el entorno *software* puede llegar a adquirir una notable complejidad.

   El paradigma de esto último es el programa JTRS (*Joint Tactical Radio System*) [35, 36] del departamento de defensa del gobierno estadounidense, iniciado en 1997, cuyo objetivo era desarrollar la nueva generación de sistemas de comunicaciones militares basándose en radio *software*. Uno de los principales desarrollos de este programa fue la SCA (*Software Communications Architecture*), una infraestructura estándar orientada a objetos que debía permitir la interconexión de elementos *hardware* y *software* de distintos fabricantes para formar una radio. Sin embargo todo el sistema se reveló tan complejo que tanto los plazos como los costes de los proyectos integrados en el programa fueron creciendo continuamente hasta que finalmente fue cancelado a finales de 2011. El lector interesado puede consultar en [37] un análisis sobre algunas causas de los problemas encontrados a lo largo del programa.

4. Adaptabilidad a condiciones cambiantes. Es posible construir sistemas de comunicaciones que puedan adaptarse a cambios bruscos en las condiciones de funcionamento, como fallos *hardware*, cambios en el canal de comunicaciones o cambios en señales interferentes [34]. Esta puede ser una característica muy deseable en determinadas aplicaciones, como por ejemplo los sectores aeroespacial (satélites y sondas) y de defensa.

5. Coste y consumo. En aplicaciones de elevado volumen de producción y bajo coste puede seguir siendo más rentable desarrollar un Application Specific Integrated Circuit (ASIC). En cuanto a consumo, los procesadores programables son menos eficientes que un ASIC con una funcionalidad similar y lo mismo ocurre con *hardware* de radio frecuencia diseñado específicamente para una banda determinada frente a uno genérico con un gran ancho de banda. El uso de MEMS[38, 39] para esta última aplicación está contribuyendo a reducir las diferencias en este sentido.

2.3.2 *Aplicaciones de la radio software*

Aunque, como ya se ha visto, la radio *software* tiene características que afectan o son aplicables a cualquier tipo de sistema en el que se emplee, lo realmente interesante

es que abre posibilidades que no serían realizables con tecnologías tradicionales. A continuación se enumeran algunas de ellas.

1. Codificación, modulación y ancho de banda adaptativos. La idea de poder modificar de forma dinámica estos parámetros en función de las condiciones de cada momento y lugar es ya muy antigua, pero la radio *software* simplifica enormemente su aplicación. La inmensa mayoría de los estándares modernos lo emplean, incluyendo (los más recientes) la adaptación del ancho de banda utilizado, típicamente seleccionando el número de canales de ancho de banda fijo a emplear.

2. Radio jerárquica. Un usuario móvil tiene acceso a distintas redes de comunicaciones en distintos lugares y momentos. Por lo general cada tipo de red tiene una capacidad de transmisión y una cobertura que están inversamente relacionadas. Por ejemplo, en casa, en el trabajo o en determinados lugares públicos el usuario puede tener acceso a una red WiFi; dentro de grandes núcleos urbanos puede acceder a una red móvil 5G, mientras que en áreas con menos densidad de población podría haber disponible únicamente cobertura 4G, 3G o incluso 2G; y como último recurso podría tener acceso a una red global por satélite cuando no hay otra opción. La radio *software* puede permitir el acceso a todas estas redes desde un único terminal, gestionando incluso el cambio de una red a otra de forma transparente para el usuario cuando éste se mueve.

3. Radio cognitiva. Este concepto también fue introducido por Joseph Mitola III [40]. Se trata no sólo de que la radio sea reconfigurable sino de que además sea "consciente" de su entorno, escogiendo todos los parámetros de comunicación (banda utilizada, modulación, codificación...) óptimos en cada momento en base a un ciclo cognitivo (figura 12). La radio cognitiva es un campo de investigación muy activo actualmente, aunque por el momento sus ambiciosas metas están derivando en la práctica hacia una radio que se pueda adaptar automáticamente al espectro disponible en cada momento.

4. Radio verde (*green radio*). En los últimos años se está investigando activamente en cómo aplicar técnicas de radio *software* y radio cognitiva para reducir el consumo global de las redes de comunicaciones inalámbricas [41].

2.4 ARQUITECTURAS PARA RADIO SOFTWARE

Una de las características de los sistemas de radio *software* es que demandan una elevada capacidad de cálculo. Por ejemplo, en [42] se estima que una implementación

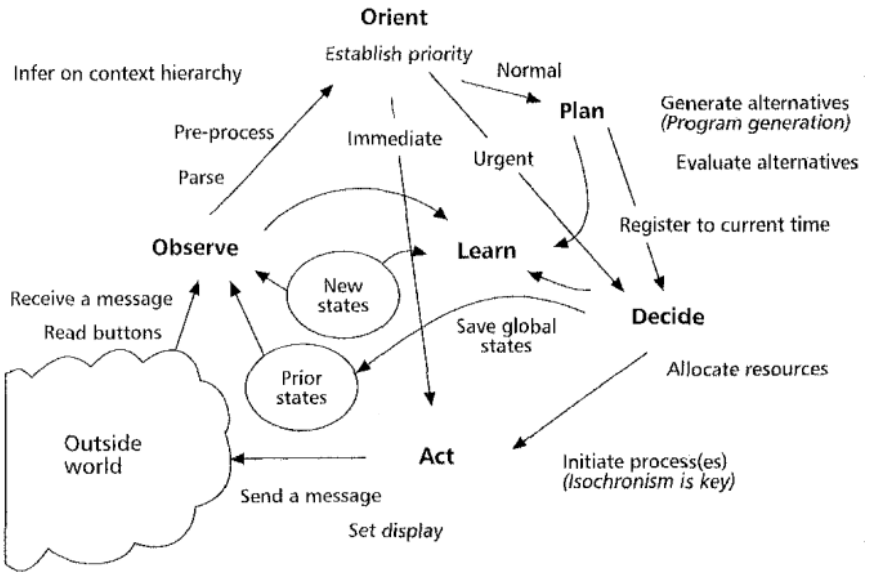


Figura 12: Ciclo cognitivo tal como se describe en [40]

de LTE[3] requiere una capacidad de proceso de entre 100 y 300 GOPS [4]. A lo largo de la última década se han dedicado grandes esfuerzos a investigar en arquitecturas de procesadores que permitan satisfacer estos requerimientos con un consumo de energía aceptable, lo que especialmente en el caso de terminales móviles es un fuerte condicionante. Es un hecho aceptado que ejecutar todas las operaciones necesarias en un único procesador, de ser posible, necesitaría una velocidad de funcionamiento tan elevada que el consumo de energía del procesador lo haría inviable en la práctica, por lo que se puede afirmar que todas las propuestas recurren a arquitecturas con varios procesadores, ya sean todos del mismo tipo (homogéneas) o no (heterogéneas).

Según se sugiere en [43], las arquitecturas propuestas pueden dividirse en dos grandes grupos: las basadas en *hardware* reconfigurable y las basadas en procesadores. Entre estas últimas cabe distinguir entre arquitecturas basadas en DSP y las de tipo *many core*, con decenas o incluso cientos de procesadores.

A continuación se muestran algunas de las arquitecturas más relevantes de acuerdo a la clasificación mencionada. No se pretende hacer una enumeración exhaustiva sino únicamente mostrar ejemplos relevantes de cada uno de los tipos nombrados.

3 LTE (*Long-Term Evolution*) es el estándar de comunicaciones inalámbricas en el que se basan los sistemas de telefonía móvil de cuarta generación (4G).

4 1 GOPS es equivalente a mil millones de operaciones por segundo (*Giga-Operations Per Second*).

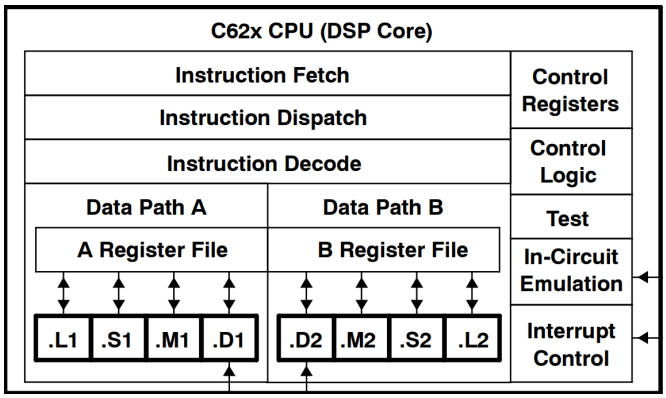


Figura 13: Diagrama de bloques de la arquitectura C62x [44]

Algunas de estas arquitecturas han derivado en desarrollos comerciales, mientras que otras no han llegado a salir del ámbito académico.

2.4.1 *Arquitecturas basadas en procesadores de tipo DSP*

En esta sección se muestran arquitecturas basadas en un número no demasiado elevado (típicamente hasta ocho) de procesadores optimizados para el procesado de señal. En muchas ocasiones estos procesadores se complementan con algún acelerador especializado, por ejemplo para la decodificación de códigos convolucionales.

2.4.1.1 *C6000*

Esta arquitectura de Texas Instruments se incluye a modo de ejemplo de DSP que puede considerarse como tradicional, ya que se presentó en 1997 con dos versiones, una de punto fijo (C62x [44]) y otra de punto flotante (C67x). La base común es una arquitectura VLIW con ocho unidades funcionales (dos multiplicadores y seis ALUs) separadas en dos rutas de datos (*datapaths*) independientes, como se puede ver en la figura 13. Esta base se ha ido actualizando en sucesivas iteraciones (C64x, C64x+ y la actual C66x [45]), que fundamentalmente han añadido y después mejorado capacidades SIMD. La arquitectura C66x incluye también las capacidades de punto flotante que antes sólo estaban disponibles en la C67x y sus derivadas.

Esta arquitectura ha sido empleada en multitud de SoC que combinan núcleos DSP, núcleos ARM de propósito general y diversos aceleradores *hardware* especializados. Ejemplos de este tipo de sistemas son algunos de los procesadores OMAP, concebidos como procesadores de aplicaciones para terminales móviles, los DaVinci, para proce-

sado de imágenes y vídeo, y los KeyStone y KeyStone II, de los que existen sistemas tanto de propósito general como especializados en comunicaciones o en aplicaciones multimedia. Parte de este trabajo de tesis se ha desarrollado utilizando procesadores de estas familias, que se irán describiendo con mayor detalle según vaya siendo necesario.

2.4.1.2 *SODA*

SODA [46] es un desarrollo de la Universidad de Michigan en Ann Arbor en colaboración con ARM. Está basado en cuatro DSPs que cuentan con una unidad escalar y una unidad SIMD muy ancha (32 vías), más un núcleo ARM de propósito general (figura 14).

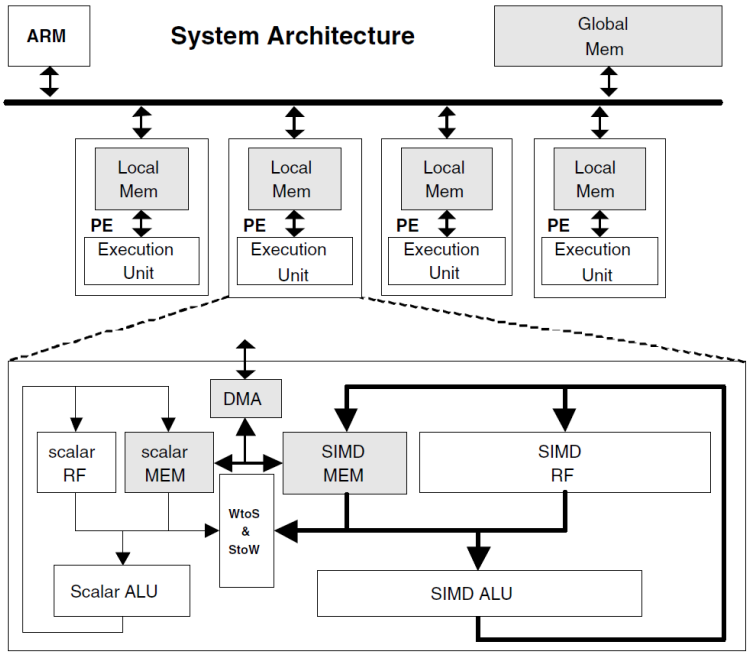
Además de para operaciones típicas, la unidad SIMD proporciona soporte para operaciones de movimiento de datos (escalar a vectorial y viceversa) e incluye también una red de permutación (*shuffle network*) que permite acelerar algoritmos como el cálculo de la FFT y la decodificación de Viterbi y Turbo. En esta arquitectura se han realizado implementaciones de IEEE 802.11a y W–CDMA.

Existen dos versiones posteriores de esta arquitectura. Por un lado, ARM desarrolló un prototipo de SoC denominado Ardbeg [47] en el que se reducen los DSPs de cuatro a dos, aunque la unidad SIMD se mejora haciéndola más flexible y añadiendo paralelismo a nivel de instrucción, y se añade un coprocesador para la decodificación Turbo.

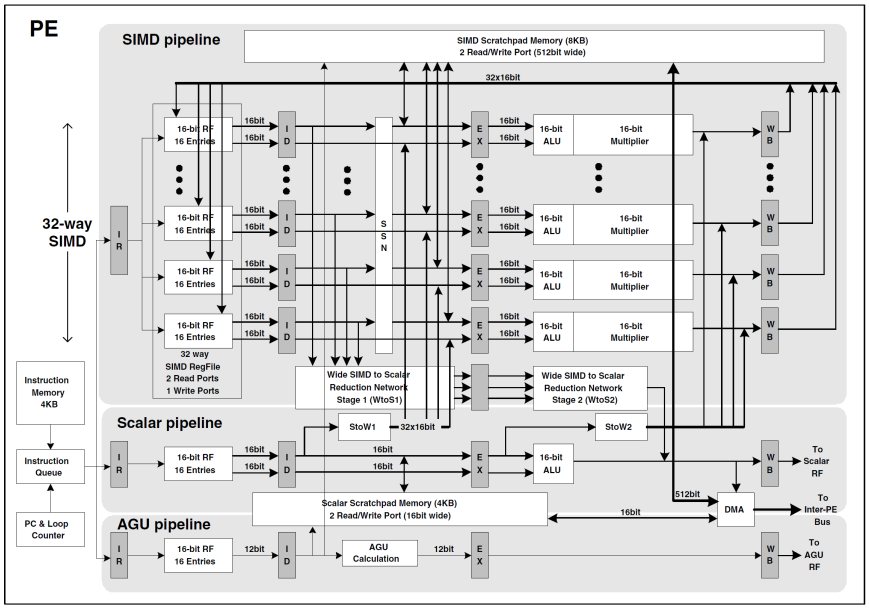
Por otro lado, se desarrolló una segunda versión de SODA denominada AnySP [48] que mantiene los cuatro núcleos DSP y amplía y mejora la unidad SIMD. La nueva versión consiste en ocho unidades de ocho vías que se pueden combinar de varias formas para formar una unidad de hasta 64 vías. En esta versión se ha implementado una radio 4G y un decodificador de vídeo H.264 funcionando simultáneamente. La figura 15 muestra la estructura de ambas evoluciones de SODA.

2.4.1.3 *LeoCore*

El procesador LeoCore [49], desarrollado por la empresa sueca CoreSonic AB, es un ASIP (*Application Specific Instruction Processor*) altamente especializado con una arquitectura que sus autores denominan SIMT (*Single Instruction Multiple Tasks*). El juego de instrucciones incluye, además de las habituales en cualquier arquitectura RISC, las llamadas instrucciones SIMT, que son ejecutadas por procesadores especializados en paralelo con otras instrucciones. Las instrucciones SIMT realizan cálculos sobre vectores complejos de longitud arbitraria, como por ejemplo la mariposa básica del algoritmo de cálculo de la FFT. La figura 16 muestra un esquema de la arquitec-

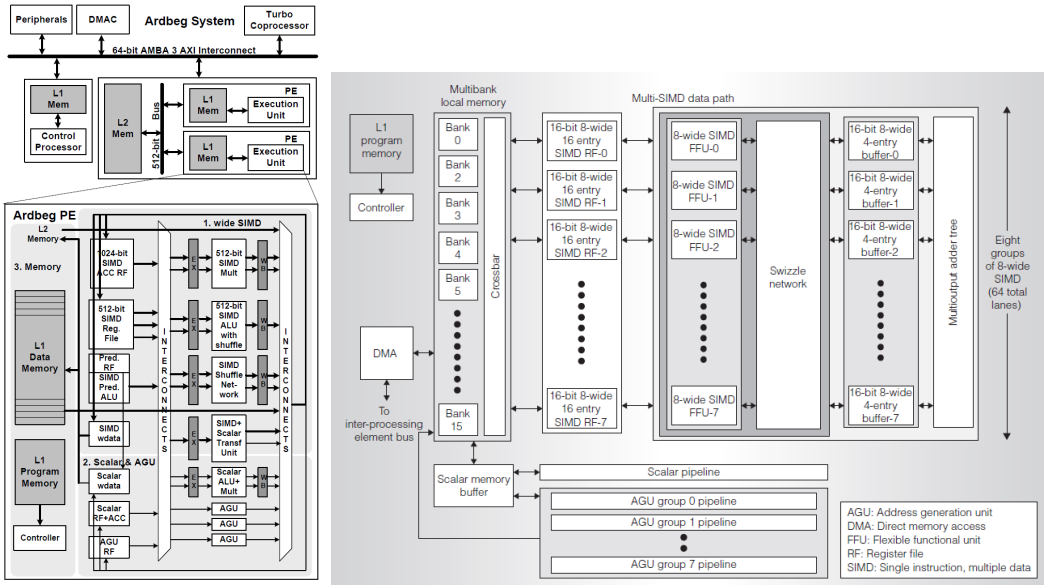


(a) Diagrama de bloques



(b) Detalle del elemento de proceso (PE)

Figura 14: Procesador SODA [46]



(a) Ardbeg [47] (b) AnySP [48]

Figura 15: Estructura de los procesadores Ardbeg y AnySP

tura del procesador y un ejemplo de cómo las instrucciones SIMT se ejecutan fuera del núcleo RISC.

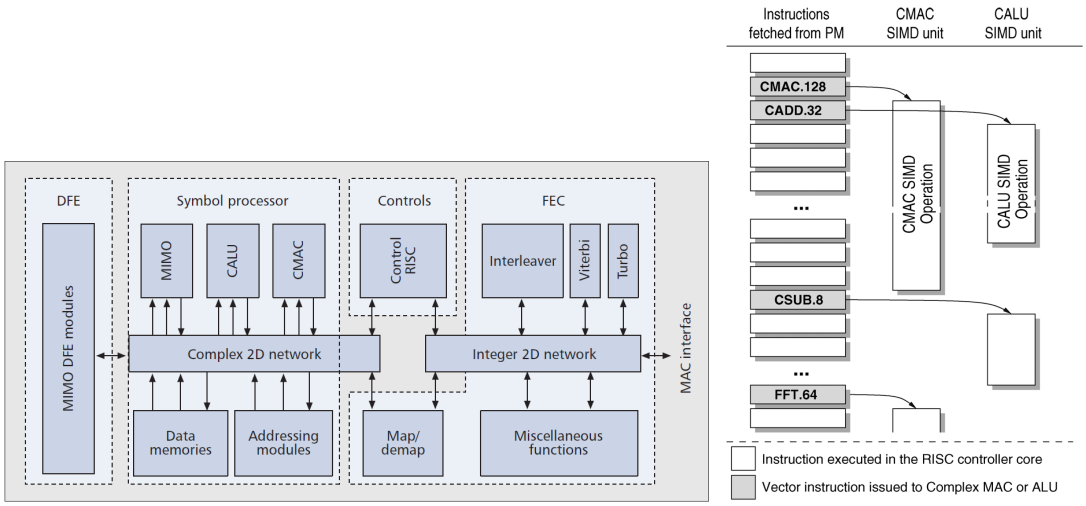
Los autores han publicado datos de rendimiento para una implementación de DVB–T/H [50] con un consumo de tan sólo 70 mW.

CoreSonic fue adquirido en 2012 por MediaTek Inc., por lo que es de suponer que esta arquitectura o derivados de ella se empleen en la gama actual de procesadores de este último fabricante.

2.4.1.4 *Sandblaster*

La arquitectura Sandblaster [51], desarrollada por Sandbridge Technologies, consiste en un DSP cuya característica más destacable es una unidad vectorial SIMD que contiene unidades funcionales especializadas para acelerar operaciones necesarias en los algoritmos utilizados habitualmente en los sistemas modernos de comunicaciones: aritmética en cuerpos de Galois, operaciones para el cálculo de la FFT, para la decodificación de Viterbi y de códigos Turbo, etc.

La figura 17 muestra la estructura del SoC SB3500, que como se puede ver combina un procesador de propósito general (ARM) con tres núcleos DSP Sandblaster. En este



(a) Diagrama de bloques [49] (b) Ejemplo [50]

Figura 16: Procesador LeoCore y ejemplo de ejecución de instrucciones en paralelo

sistema se ha implementado el procesado en banda base de un terminal LTE categoría 2 [52].

2.4.1.5 *DXP*

La arquitectura DXP (*Deep eXecution Processor*) [53] es un desarrollo de la empresa Icera. Está basada en una unidad de control y una unidad vectorial SIMD de cuatro vías con juegos de instrucciones separados, al estilo LIW. La unidad vectorial tiene como características fundamentales unas unidades de ejecución con segmentación profunda (es decir, con un elevado número de etapas) y un mapa de configuración programable que contiene datos de configuración y constantes para las unidades de ejecución.

Icera fue adquirida por Nvidia en 2011. La arquitectura DXP se usa en el procesador para LTE i500 [54].

2.4.2 *Arquitecturas Many-Core*

En esta sección se describen arquitecturas basadas en la utilización de un número elevado de procesadores, decenas o incluso cientos.

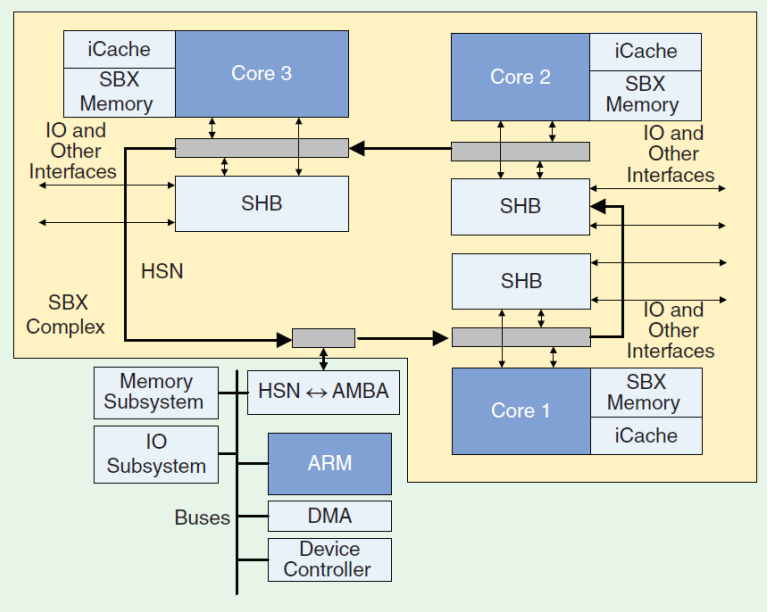


Figura 17: Esquema del SoC SB3500 [42]

2.4.2.1 *TILEPro64*

La arquitectura del TILEPro64 [55], desarrollado por Tilera Corp., está basada en 64 DSPs VLIW de 3 vías, como se muestra en la Figura 18. Su principal característica es que los procesadores pueden agruparse de forma flexible para formar uno o varios grupos SMP (*Symmetric Multi-Processing*) que ejecutan un sistema operativo estándar (Linux), de modo que la forma de programarlos puede ser muy similar a la de un ordenador común.

Según Tilera el procesador está orientado a aplicaciones de red (especialmente diversas formas de monitorización de tráfico), multimedia (por ejemplo transcodificación de vídeo) y de infraestructura de comunicaciones inalámbricas (como por ejemplo estaciones base WiMAX o LTE). En evoluciones posteriores de esta arquitectura, como la familia TILE–Gx, se ha sustituido el procesador de base por un núcleo RISC de 64 bits de propósito general, más orientados a aplicaciones de centro de datos. La compra de Tilera por EZChip en 2014 ha contribuido aún más a este cambio de orientación.

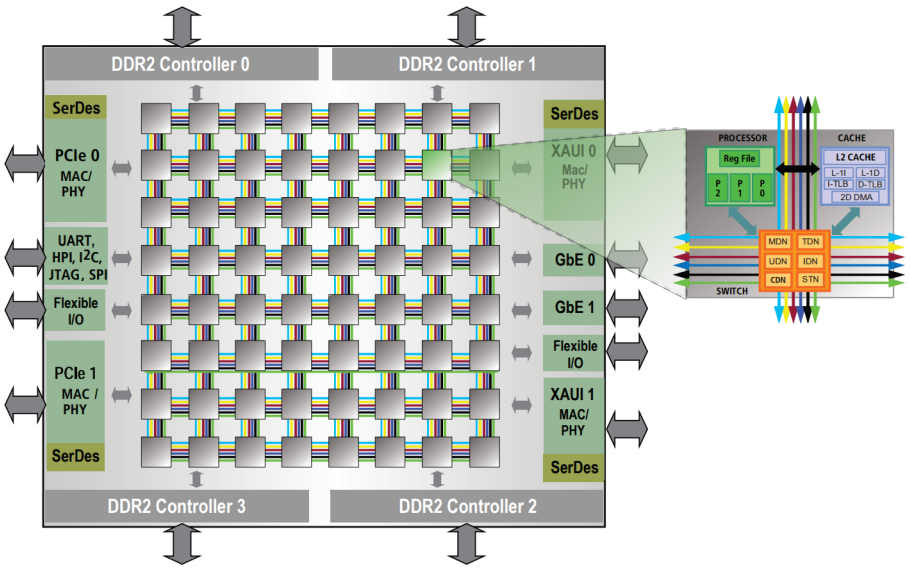


Figura 18: Esquema del procesador TILEPro64 [55]

2.4.2.2 *AsAP y Kilocore*

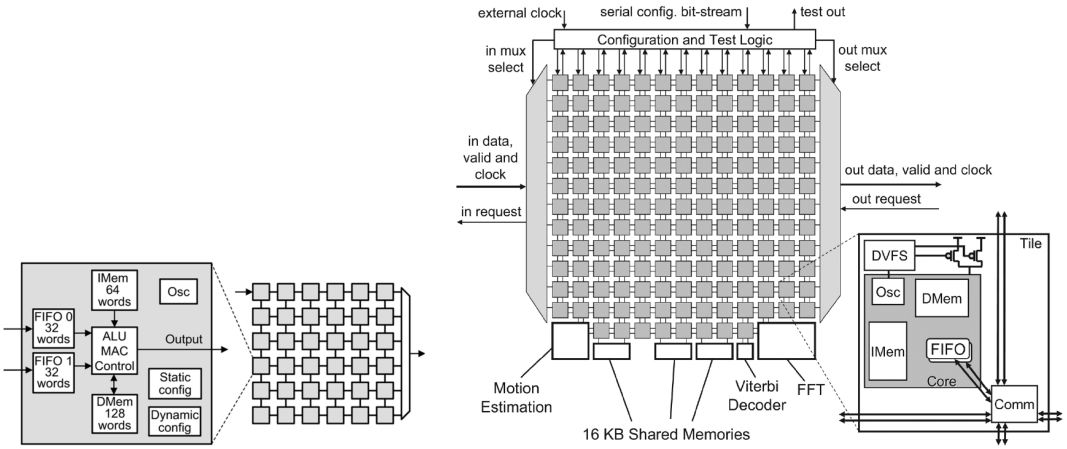
AsAP (*Asynchronous Array of simple Processors*) es un desarrollo de la Universidad de California en Davis. Está compuesta por un número elevado de procesadores (36 en la primera versión [56], 167 en una segunda versión llamada AsAP2 [57]) muy simples, con una memorias de instrucciones y datos muy reducidas: 64 y 128 palabras, respectivamente.

Los procesadores están dispuestos en una matriz bidimensional, tal como se puede ver en la figura 19. Cada procesador cuenta con dos memorias FIFO de 32 palabras en las que cualquiera de sus cuatro vecinos puede escribir.

La primera versión de la arquitectura ofrecía unas prestaciones insuficientes ya que según los resultados publicados sólo podía procesar el 30 % de la carga necesaria para un transmisor IEEE 802.11a [58].

La segunda iteración del diseño, AsAP2, además de aumentar el número de procesadores incluye tres bloques de memoria compartida de 16 kB y tres aceleradores para decodificación de Viterbi, cálculo de la FFT y estimación de movimiento. En esta versión se implementó un receptor IEEE 802.11a [59] con un consumo de 130 mW.

En el año 2016 se presentó una tercera versión de la arquitectura cuyo nombre, Kilocore, resaltaba el hecho de alcanzar los mil procesadores [60]. Los procesadores individuales mantenían una arquitectura similar a la de las versiones anteriores, con una capacidad de memoria ligeramente superior a la de las versiones anteriores (128



(a) AsAP [56] (b) AsAP2 [57]

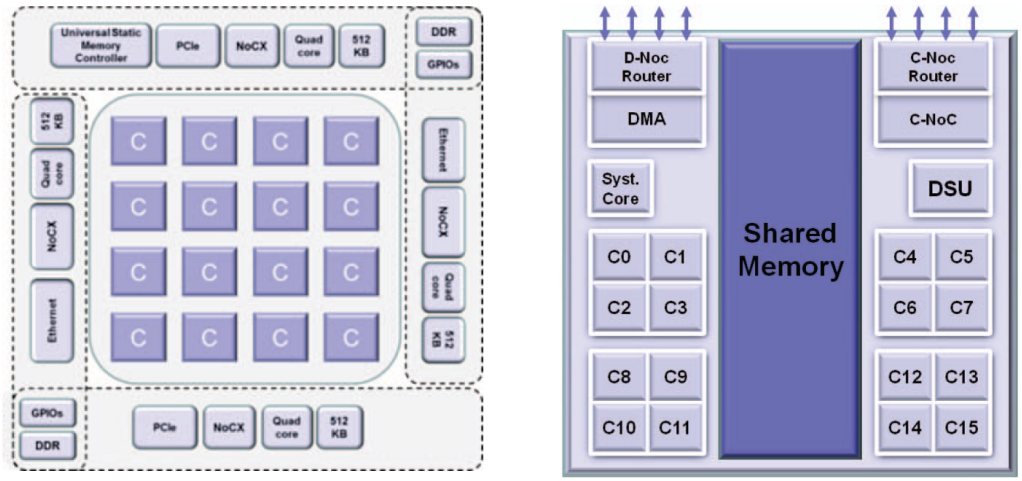
Figura 19: Estructura de los procesadores AsAP y AsAP2

palabras de memoria de instrucciones y 512 *bytes* de memoria de datos), pero aún así muy reducidas. Al igual que en AsAP2 se incluyen bloques de memoria adicional, en esta ocasión 12 bloques de 64 kB, pero desaparecen los aceleradores especializados.

2.4.2.3 *Kalray MPPA*

MPPA, además del acrónimo (*Massively Parallel Processor Array*) utilizado en ocasiones para denominar de forma general las arquitecturas basadas en matrices de procesadores como las que se están viendo en esta sección, es el nombre con el que la compañía francesa Kalray comercializa su propia arquitectura. La primera versión, llamada MPPA-256 y lanzada en 2013 [61], estaba formada por 16 *clusters* de 16 procesadores VLIW de 32 bits, más otros 32 procesadores dedicados a tareas de gestión del sistema, uno en cada *cluster* y un procesador de cuatro núcleos SMP en cada uno de los cuatro subsistemas de E/S. La figura 20 muestra la estructura general y la del *cluster*.

Existen dos versiones posteriores de la arquitectura MPPA. La segunda versión, presentada en 2015 y revisada en 2017, introduce relativamente pocos cambios en la arquitectura, siendo quizá el más significativo la adición de un coprocesador criptográfico en cada núcleo VLIW. La tercera versión, presentada en 2020, mantiene el diseño básico pero reduce el número de *clusters* a 5, con 80 procesadores de cómputo en total, y la arquitectura de éstos sigue siendo VLIW pero pasan a ser de 64 bits, y además se añade a todos ellos un coprocesador para operaciones con tensores pa-



(a) Diagrama de bloques (b) Detalle del *cluster*

Figura 20: Arquitectura del procesador MPPA-256 [61]

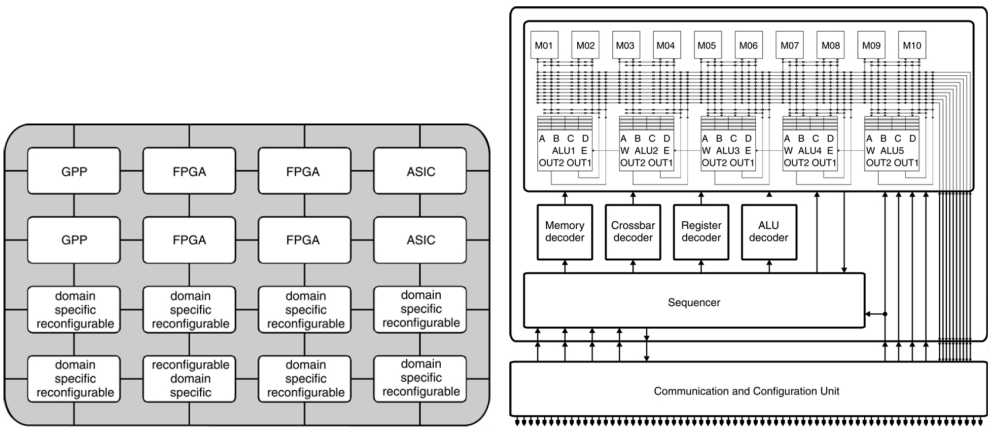
ra mejorar el rendimiento en aplicaciones de aprendizaje automático e inteligencia artificial en general.

2.4.3 *Procesadores reconfigurables*

Los procesadores reconfigurables se sitúan en un punto medio entre los procesadores al uso, en los que la arquitectura es fija, y los dispositivos de lógica programable (FPGA). Es relativamente habitual referirse a ellos con el término CGRA (*Coarse Grain Reconfigurable Architecture*).

Su característica principal es que en general incluyen una serie de unidades funcionales, típicamente ALUs de cierta complejidad, cuyo número puede variar entre unas pocas y varias decenas, y una unidad de configuración y control que controla tanto las funciones concretas implementadas por las unidades funcionales como el flujo de datos entre ellas. A menudo incluyen también una pequeña cantidad de memoria local.

Algunas CGRA están diseñadas para integrarse dentro de un procesador como una unidad funcional más, formando lo que puede verse como un procesador en el que el usuario puede definir la funcionalidad de algunas instrucciones, que además puede ser relativamente compleja. Otras, por el contrario, están pensadas para formar parte de lo que suele llamarse un SoC configurable (CSoC), que puede estar formado por elementos que van desde procesadores de propósito general hasta circuitos de



(a) Estructura del SoC (b) Procesador reconfigurable

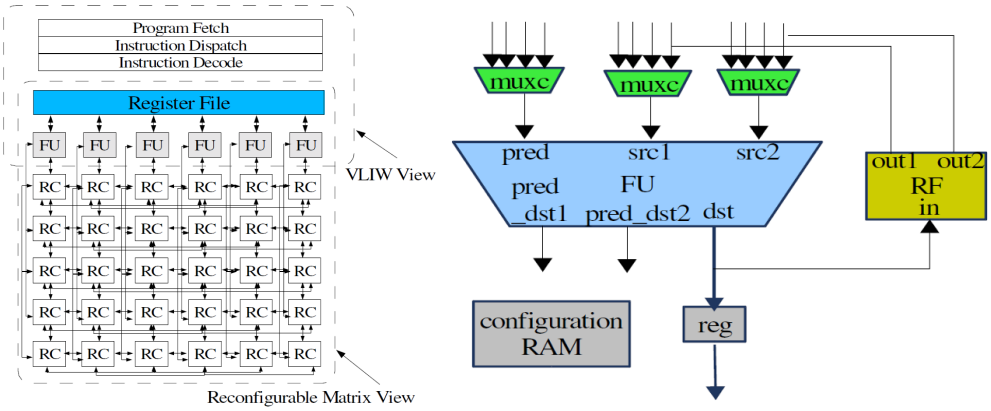
Figura 21: Arquitectura del procesador Montium [63]

propósito específico (ASIC). A continuación se describen dos ejemplos de CGRA que se han considerado representativos de ambas posibilidades.

2.4.3.1 *Montium*

Uno de los tipos de arquitectura que se han propuesto para el diseño efectivo de SoC para procesado de señal son las llamadas arquitecturas en mosaico (*tile architecture*) [62]. Estas arquitecturas están compuestas por elementos de procesado (teselas o *tiles*) que se conectan entre sí mediante algún tipo de NoC. En las propuestas más generales las teselas pueden ser de lo más variado, desde procesadores de propósito general a ASIC, pasando por DSPs, CGRAs o FPGAs. La Figura 21.a muestra un esquema de este tipo de arquitectura.

Montium [63], desarrollado en la Universidad de Twente, es un ejemplo de CGRA diseñado para integrarse en un SoC de esta clase. Su estructura, que puede verse en la Figura 21.b, consiste básicamente en cinco unidades de procesamiento formadas por una ALU y dos pequeñas memorias locales de 1 kB. Un secuenciador que incluye una pequeña memoria con capacidad para 256 instrucciones controla tanto las operaciones que realizan las ALUs como el flujo de datos entre todos los elementos, ALUs y memorias. El conjunto de ALUs, memorias y secuenciador forman el procesador de la tesela (*TP, Tile Processor*). Finalmente, un elemento denominado CCU (*Communication and Configuration Unit*) controla tanto la configuración de la tesela como la comunicación con el resto de elementos del SoC.



(a) Diagrama de bloques (b) Detalle de la unidad reconfigurable (RC)

Figura 22: Estructura del procesador ADRES [68]

La programación de Montium se realiza combinando una mezcla de métodos formales y heurísticos, tal como se describe en [64]. El flujo de diseño para el SoC completo está basado en redes de procesos de Kahn [65], una metodología común en aplicaciones de procesado de señal [66]. Los nodos de esta red de procesos representan los distintos algoritmos empleados en la aplicación. Un GPP controla el mapeo entre los nodos de la red de procesos y las teselas que componen el SoC en base a una biblioteca de implementaciones de cada algoritmo para cada tipo de procesador. A menudo cada algoritmo está implementado únicamente en un tipo de procesador. Cada una de estas implementaciones individuales se realiza de forma independiente; en el caso de Montium, se define una metodología para la programación pero el proceso es fundamentalmente manual, con ayuda únicamente de herramientas básicas (ensamblador y herramienta de gestión de configuración).

En [67] se describen los resultados de implementar diversos algoritmos de los estándares HiperLAN/2 y DAB (FFT, decodificadores Viterbi y Turbo) en Montium.

2.4.3.2 *ADRES*

ADRES [68] es un desarrollo de IMEC, la Universidad Católica de Lovaina y la Universidad Libre de Bruselas. Su objetivo es reducir las ineficiencias que según sus autores se derivan, tanto a nivel *hardware* como *software*, de la independencia entre elementos reconfigurables y procesadores.

La estructura general del procesador ADRES se puede ver en la Figura 22.a. La idea básica es integrar una matriz de unidades reconfigurables (RC) con un procesador VLIW. Sólo uno de los dos elementos (procesador o matriz reconfigurable) está

activo en cada momento. Esto facilita que tanto los registros como las unidades funcionales del procesador VLIW sean accesibles también desde la matriz reconfigurable y puedan utilizarse para la comunicación e intercambio de datos entre ambas partes.

La Figura 22.b muestra un detalle de la estructura de la RC. La memoria de configuración puede almacenar varias configuraciones diferentes que pueden cambiarse de un ciclo a otro. La configuración se puede cargar también de la jerarquía de memoria del procesador VLIW utilizando sus unidades funcionales. Cada RC tiene también un pequeño conjunto de registros locales.

La programación se realiza en C mediante un compilador desarrollado para esta arquitectura llamado DRESC [69], desarrollado a partir de IMPACT [70]. Este compilador identifica bucles acelerables mediante la matriz reconfigurable y genera o bien la información de configuración de la matriz reconfigurable para estas secciones o bien código para el procesador VLIW en las secciones no acelerables.

ADRES se ha usado en diversos productos comercializados por IMEC. Por ejemplo en [71] se describe una implementación con tecnología de 90 nm que funcionando a 400 MHz es capaz de realizar el procesado en banda base de un sistema MIMO-OFDM con una capacidad de transmisión superior a los 100 Mbps, y en [72] se describe un SoC que incorpora dos núcleos ADRES, un procesador ARM, aceleradores específicos para, por ejemplo, decodificación de Viterbi, y otros elementos.

2.5 METODOLOGÍA

Como se ha podido ver en la sección anterior las opciones *hardware* que se proponen para los sistemas de radio *software* son múltiples y variadas, y van desde sistemas que no difieren mucho de los procesadores que podemos encontrar en PCs de sobremesa hasta arquitecturas muy orientadas a la aplicación. Todas ellas, sin embargo, comparten la característica de que están basadas en varios núcleos, procesadores o unidades de procesamiento que deben trabajar en paralelo para obtener el máximo rendimiento posible o, en ocasiones, simplemente para que funcionen.

El diseño de aplicaciones y la programación de este tipo de sistemas es a menudo un problema tan complejo como el diseño de la arquitectura en sí, a veces incluso más. El objetivo principal es extraer el mayor grado posible de paralelismo en las tareas a implementar, para obtener así el mayor rendimiento posible de los recursos de computación de los que disponga la arquitectura. Tradicionalmente se reconocen tres niveles de paralelismo:

1. A nivel de instrucción: ILP (*Instruction Level Parallelism*).

2. A nivel de datos: DLP (*Data Level Parallelism*).

3. A nivel de tarea: TLP (*Task Level Parallelism*).

Existe ILP cuando es posible ejecutar simultáneamente varias instrucciones de un programa. Se han desarrollado multitud de técnicas a nivel de arquitectura para poder extraer el máximo grado de ILP de un programa cualquiera: segmentación, ejecución fuera de orden, ejecución superescalar, VLIW... La segmentación es una técnica básica que se encuentra en prácticamente cualquier procesador, por lo que no se hará más mención de ella. De las restantes, en los procesadores orientados al procesado de señal es relativamente frecuente recurrir a arquitecturas VLIW, como se ha podido ver en la sección anterior. El resto de técnicas mencionadas suelen usarse en procesadores de propósito general pero no en los orientados al procesado de señal.

El DLP consiste en aplicar una determinada operación (por ejemplo la suma o el producto) o un conjunto de operaciones a varios conjuntos de datos simultáneamente. Es frecuente que los algoritmos empleados en los sistemas de radio *software* sean adecuados para este tipo de paralelismo, por lo que, como también se ha visto en la sección anterior, es frecuente que las arquitecturas orientadas a esta aplicación incluyan unidades vectoriales SIMD más o menos complejas.

Por último, el TLP consiste en realizar dos o más tareas distintas simultáneamente. Las aplicaciones dominadas por flujo de datos, como es la radio *software*, contienen un elevado grado de este tipo de paralelismo, lo que favorece y facilita el uso de arquitecturas multiprocesador en este ámbito.

La búsqueda de ILP y DLP se realiza habitualmente mediante compiladores que optimicen el código generado para la arquitectura de destino, aunque no es infrecuente la programación manual a bajo nivel de porciones críticas de los algoritmos. El TLP, por otro lado, se obtiene bien con herramientas de más alto nivel, bien de forma manual o mediante metodologías híbridas.

En esta sección se exponen algunas de las metodologías de desarrollo del *software* para este tipo de sistemas, un campo en el que la investigación es muy activa y en el que se sitúan las contribuciones más significativas de este trabajo de tesis.

2.5.1 *Diseño sin herramientas automáticas*

En el campo de los procesadores de propósito general es inconcebible emplear un procesador que no vaya acompañado al menos de un conjunto básico de herramientas: compilador (habitualmente de C o C++), depurador, etc. Esto es así debido a que este tipo de procesadores han sido investigados, desarrollados y utilizados desde hace decenios, y hay ya disponible un trabajo ingente sobre herramientas de desarrollo para ellos que hace relativamente asequible crear estas herramientas para una nueva arquitectura. Sin olvidar, además, que en realidad los conjuntos de instrucciones que

emplean estos procesadores están ya firmemente establecidos tanto en la industria como en el ámbito académico, y la inmensa mayoría de las nuevas arquitecturas que se desarrollan usan uno de estos conjuntos de instrucciones (habitualmente ARM o, más recientemente, RISC-V, aunque se siguen empleando otros como Power o MIPS) o se basan fuertemente en ellos. Aunque en menor medida, esto es también aplicable a las arquitecturas de DSPs tradicionales ya establecidas, como pueden ser C64/C66, StarCore, etc. Muchas de estas familias de procesadores están además soportadas por herramientas de desarrollo de más alto nivel, que son capaces de generar automáticamente código en C o C++ a partir de especificaciones de sistemas basadas en modelos, en lenguajes de especificación formal, etc.

Sin embargo, el esfuerzo necesario para desarrollar herramientas para una arquitectura que sea significativamente diferente de las que se acaban de mencionar es mucho mayor, con frecuencia comparable al necesario para desarrollar la propia arquitectura, y tanto más cuanto menos convencional sea ésta. Esto hace que no sea raro, especialmente cuando la arquitectura *hardware* sobre la que se está implementando el sistema no ha alcanzado aún un cierto grado de madurez, que el desarrollo se deba realizar total o parcialmente sin ayuda de herramientas que automaticen el proceso.

En estos casos es habitual partir de un modelo de alto nivel del sistema, bien sea programándolo en algún lenguaje como C o C++, o bien con alguna herramienta de simulación como Matlab/Simulink. A partir de este modelo de alto nivel, que sirve además para realizar una validación funcional del sistema, se identifican los algoritmos que hay que implementar y se recodifican manualmente para adaptarlos a la arquitectura de destino. Poco a poco se construye una biblioteca de algoritmos que puede constituir la base de un sistema de desarrollo, como se describirá en los siguientes apartados.

Ejemplos de arquitecturas descritas en la literatura científico-técnica en las que las aplicaciones se programaron manualmente a bajo nivel son AsAP o Montium. En este último caso los autores publicaron una reflexión a posteriori sobre el proceso de diseño [73] en la que se hace mención explícita a este tipo de problemas.

2.5.2 *Diseño basado en APIs estándar de paralelización*

En el ámbito de la programación de sistemas multiprocesador está ampliamente aceptada la idea de que la mejor manera de extender y facilitar el uso de modelos de programación paralela pasa por el uso de estándares abiertos y comúnmente aceptados tanto en la industria como en la academia [16]. Desde hace ya años se ha extendido el uso de varios de estos estándares, que aunque son especialmente rele-

vantes en el ámbito de la computación de alto rendimiento (HPC, *High Performance Computing*) también se aplican con frecuencia en la programación de sistemas empotrados multiprocesador, tanto homogéneos como heterogéneos. A continuación se describen brevemente dos de estos estándares, OpenMP y OpenCL, que han sido y son utilizados con cierta frecuencia en el ámbito de la radio *software*.

2.5.2.1 *OpenMP*

La primera versión de OpenMP [17] data de 1997 y su objetivo era fundamentalmente facilitar la paralelización en sistemas SMP de operaciones repetitivas y regulares (bucles) que aparecen frecuentemente en la computación científica. Desde entonces ha sido extendido en las sucesivas versiones que han ido apareciendo, y en la actualidad soporta también la paralelización a nivel de tarea y el uso de aceleradores *hardware* externos.

Desde el punto de vista del programador, OpenMP consiste en un conjunto de directivas que se añaden al código original (C/C++ o Fortran) para indicar al compilador que aplique determinadas transformaciones a los bloques de código a los que afectan, desde aplicar instrucciones SIMD a un bucle o partirlo en varios hilos para aprovechar todos los núcleos de un sistema SMP a descargar una sección de código a un acelerador externo. Distintos compiladores pueden aplicar distintas transformaciones a un mismo código dependiendo de sus capacidades y de las características del sistema en el que debe correr el código, pero un objetivo explícito de OpenMP desde sus comienzos es que el código siga siendo funcionalmente equivalente independientemente de que las directivas sean procesadas y aplicadas o no.

2.5.2.2 *OpenCL*

El objetivo de la primera versión de OpenCL [18], aparecida en 2009, era proporcionar una interfaz de programación estándar para el uso de aceleradores *hardware* masivamente paralelos, fundamentalmente las unidades de procesamiento de gráficos (GPUs, *Graphics Processing Unit*) aplicadas a computación de propósito general, una tarea para la que en aquel momento cada fabricante proporcionaba su propia API, como CUDA de Nvidia, que aún hoy sigue siendo bastante utilizada pese a ser privativa de un único fabricante.

Consta de dos elementos: un dialecto de C o C++ que se emplea para escribir las secciones de código, llamadas *kernels*, que deben ejecutarse en los dispositivos aceleradores, y un API de bajo nivel que permite definir y controlar cómo deben ejecutarse estas secciones en los dispositivos disponibles.

Es una API mucho más compleja que OpenMP ya que pretende ser el Java ("*write once, run everywhere*") de la programación de sistemas multiprocesador heterogéneos. Existen implementaciones de OpenCL de todos los grandes fabricantes de CPUs y GPUs (Intel, AMD, Nvidia, ARM...), pero también de fabricantes de DSPs (Texas Instruments), de FPGAs (Intel, Xilinx) y de procesadores menos convencionales como Kalray.

2.5.2.3 *Uso de OpenMP y OpenCL en sistemas de radio software*

Como se dijo antes, tanto OpenMP como OpenCL se han usado con frecuencia para la programación de sistemas de radio *software* tanto en PC como en sistemas empotrados. Algunos ejemplos pueden ser [74], donde se usa OpenMP para implementar un generador de símbolos OFDM en un DSP con ocho núcleos, o [75], que presenta la implementación de un decodificador LDPC en la misma plataforma. El mismo algoritmo se implementa en [76] usando OpenCL en una plataforma similar.

Estos ejemplos son representativos de una característica general de los usos de OpenMP y OpenCL sobre plataformas empotradas que pueden encontrarse en la literatura científico-técnica: se emplean para paralelizar un único algoritmo, o incluso solo una parte de un algoritmo, pero no una aplicación de radio *software* completa. Esto es debido a que, en general, las implementaciones de estos estándares no alcanzan un buen rendimiento cuando se emplean para acelerar varios algoritmos de manera que se ejecuten simultáneamente, algo habitualmente necesario en una aplicación de radio *software* completa. OpenCL, por ejemplo, ofrece la posibilidad de particionar un dispositivo para ejecutar varios *kernels* simultáneamente, pero no es algo que se use con mucha frecuencia y puede incluso dar lugar a problemas importantes de rendimiento, tal como se describe por ejemplo en [19]. Si se desea profundizar más en esta problemática puede consultarse el trabajo descrito en [20], donde se discute acerca de algunos patrones de uso de OpenCL y su rendimiento al ejecutar varios *kernels* de forma concurrente.

Una de las ventajas potenciales de emplear APIs estándar como OpenMP u OpenCL es la portabilidad, es decir, la posibilidad de reutilizar el código paralelizado en varias plataformas, que especialmente en el caso de OpenCL pueden tener características muy distintas. Sin embargo, la realidad suele ser que si bien es posible que un determinado código funcione correctamente en varias plataformas sin más que recompilarlo, el rendimiento puede estar muy lejos de ser óptimo a no ser que se dedique cierto esfuerzo en adaptar el código a las características específicas de cada plataforma [77].

La conclusión que se puede obtener de los ejemplos mostrados es que el uso de APIs estándar como OpenMP y OpenCL para implementar determinados algoritmos

en arquitecturas multiprocesador, tanto homogéneas como heterogéneas, presenta sin duda ventajas claras en determinados aspectos, pero existen también ciertos inconvenientes que dificultan su uso para algunos nichos de aplicación, especialmente en el caso de plataformas empotradas.

2.5.3 *Diseño con herramientas orientadas al dominio de aplicación*

Como ya se ha dicho antes, cuando se diseña un sistema de radio *software* es habitual realizar en una fase temprana del proceso un modelo de alto nivel que sirva, entre otras cosas, para validar funcionalmente el diseño y para explorar distintas alternativas en la elección de algoritmos para realizar cada una de las funciones del sistema.

Aunque es posible realizar este modelo programándolo directamente en algún lenguaje de alto nivel como C o C++, lo más habitual es emplear alguna herramienta orientada a la aplicación que facilite tanto la creación del propio modelo como la extracción de resultados a partir de él. Típicamente el modelo se crea mediante una herramienta gráfica que permite construir un diagrama de bloques del sistema, que puede incluir uno, varios o todos los elementos que lo componen: fuente de señal, equipo de transmisión, canal de transmisión, equipo de recepción y destino de la señal. Una vez creado el modelo y validado el diseño del sistema, el siguiente paso en el proceso de desarrollo puede variar mucho según la herramienta empleada y la arquitectura de destino.

Es muy frecuente que la herramienta de modelado sea capaz de generar código automáticamente, típicamente en C o C++, a partir del diagrama de bloques del sistema. Este código puede ser general o estar orientado a alguna plataforma (procesador, sistema operativo) específica. En este último caso puede ocurrir que el código generado pueda emplearse como implementación definitiva del sistema, pero esto sólo suele ocurrir en el caso de arquitecturas ya maduras y cuya estructura no se aleja de lo que puede considerarse habitual en un sistema empotrado.

Una variante relativamente frecuente de este caso es que la herramienta incluya una biblioteca de bloques que cuentan ya con una implementación optimizada para la arquitectura de destino. Si el sistema se puede construir con los bloques incluidos en la biblioteca es posible también llegar a la implementación final desde la propia herramienta de alto nivel.

Sin embargo, como ya se mencionó en el apartado anterior, lo más habitual cuando la arquitectura de destino no es una estándar es que en el mejor de los casos se pueda utilizar el código generado automáticamente por la herramienta como punto de partida para llegar a una implementación optimizada. El grado de reutilización de este

código depende tanto de las cualidades del mismo como de las herramientas de bajo nivel (compiladores, etc.) disponibles para la arquitectura de destino. Hay herramientas que generan un código razonablemente legible y modificable manualmente por un desarrollador, mientras que el que generan otras es críptico y difícilmente aprovechable. Por otro lado, el compilador para la arquitectura de destino puede aceptar un código relativamente estándar o bien, por ejemplo, aceptar únicamente un subconjunto de las estructuras de control o los tipos de datos del lenguaje de programación, en cuyo caso es probable que el código generado automáticamente no sea reutilizable o requiera grandes modificaciones.

En el límite, es posible que la implementación en la arquitectura de destino deba ser realizada partiendo por completo de cero. Aún en este caso extremo, el modelo de alto nivel puede utilizarse para comparar los resultados que se obtienen en la reimplementación con los obtenidos en la validación del sistema.

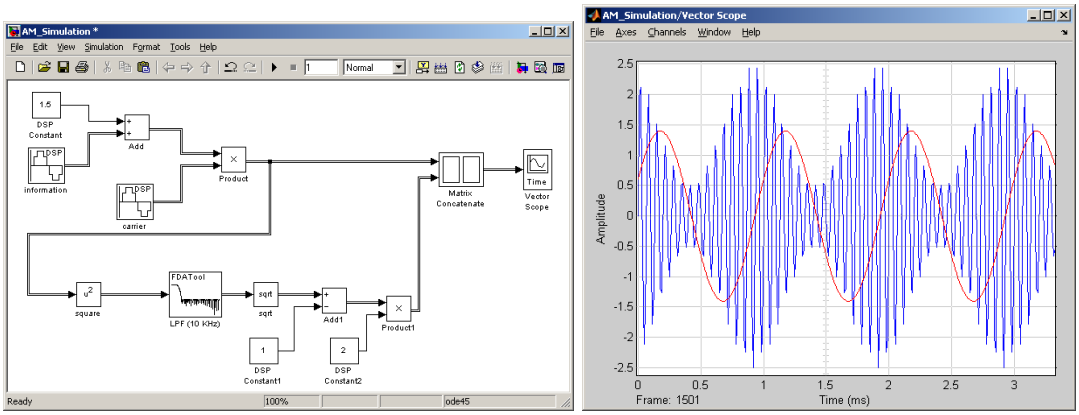
La variedad de herramientas de alto nivel disponibles es muy grande. Existen herramientas comerciales, algunas de ellas firmemente establecidas en el mercado, y herramientas de código abierto. Entre las primeras se pueden destacar Matlab/Simulink de Mathworks, SystemVue de KeySight Technologies o LabView de National Instruments. Entre las de código abierto están Scilab/Xcos, SciPy, GNU Radio y Preesm, entre otras.

A continuación se describen dos de las herramientas mencionadas que se han considerado especialmente relevantes, haciendo especial hincapié en cómo pueden emplearse en los posibles flujos de diseño que se han descrito.

2.5.3.1 *Matlab / Simulink*

A finales de la década de 1970 Cleve Moler, profesor de matemáticas en la Universidad de Nuevo México, desarrolló un lenguaje interpretado para que sus alumnos pudieran crear aplicaciones de cálculo numérico sobre matrices sin necesidad de aprender Fortran. Pocos años después se asoció con Jack Little y Steve Bangert, que reescribieron el intérprete en C, añadieron capacidades gráficas y crearon un sistema extensible de módulos, y los tres crearon en 1984 la compañía MathWorks. Al propio Matlab se añadió poco después Simulink, un entorno gráfico de programación que permite modelar sistemas dinámicos multidominio creando diagramas de bloques.

Desde entonces el conjunto Matlab/Simulink se ha convertido prácticamente en un estándar de facto para aplicaciones de cálculo y modelado, tanto en la industria como en el ámbito académico, en multitud de campos de las ciencias aplicadas y la ingeniería: economía, biología computacional, procesado de señales, comunicaciones, sistemas de control, modelado físico de sistemas, etc. El alto grado de aceptación que alcanzó poco tiempo después de su aparición, junto con su extensibilidad, ha



(a) Modelo Simulink (b) Señales modulada y demodulada

Figura 23: Modelo de modulador y demodulador AM realizado con Simulink [78]

propiciado que otras compañías creen bibliotecas de funciones para Matlab (*toolboxes*) y de bloques para Simulink (*blocksets*), y actualmente existe un enorme ecosistema de módulos disponibles para multitud de aplicaciones.

Parte del éxito de Matlab y Simulink se debe a que tanto MathWorks como otras compañías han desarrollado módulos que permiten sintetizar código en C a partir de modelos de Simulink, creando aplicaciones que corren tanto en un ordenador personal como en diversos tipos de sistemas empotrados basados en microcontroladores y DSPs. Existen también varias herramientas que permiten generar, con ciertas restricciones, una implementación *hardware* a partir de un modelo y trasladarla a una FPGA.

En el ámbito que nos ocupa, Matlab y Simulink permiten modelar con facilidad sistemas orientados a flujo de datos, como son los de radio *software*, y cuentan además con extensas bibliotecas que implementan los algoritmos habituales en estas aplicaciones, por lo que es una herramienta muy utilizada para esta tarea. La figura 23 muestra un modelo simple de un modulador y demodulador AM realizado con Simulink, junto con las señales que se obtienen como resultado de la ejecución del modelo.

Como ejemplo del uso que se le puede dar a esta herramienta, en [42] se describe el flujo de diseño que se muestra en la figura 24, empleado en IMEC para la plataforma BEAR, basada en el procesador ADRES descrito en la sección 2.4.3.2. Se parte de una descripción del sistema en Matlab que es refinada de forma manual. A continuación se genera un modelo en C del sistema mediante una herramienta comercial, y de nuevo el código que ha generado esta herramienta se modifica y optimiza manualmente

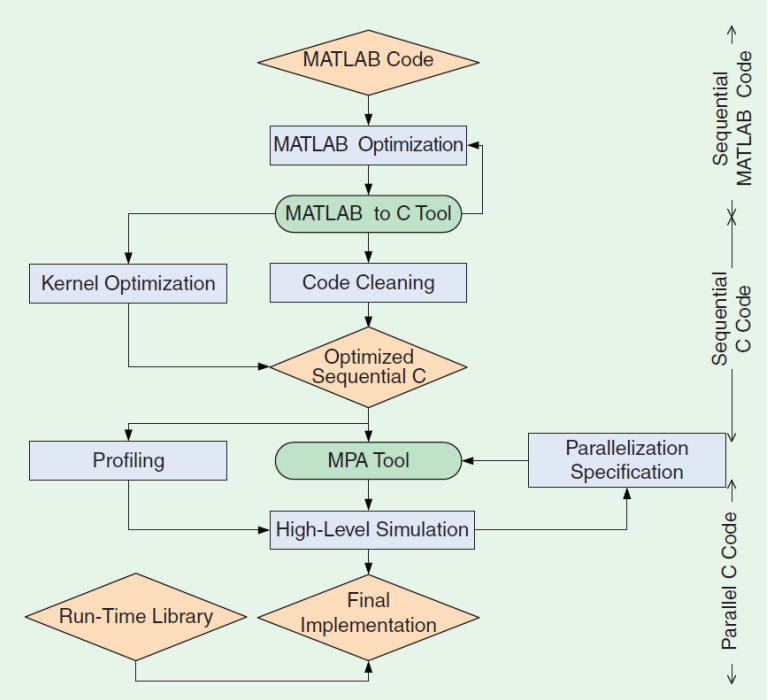


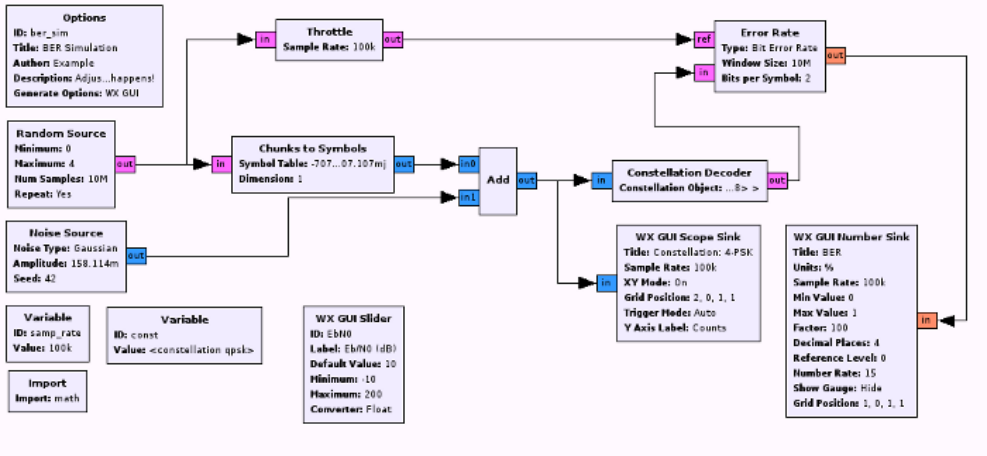
Figura 24: Flujo de diseño para BEAR, una plataforma basada en ADRES [42]

para adaptarlo a otra herramienta propia llamada MPA [79] que distribuye las tareas especificadas en el código entre los procesadores disponibles en la plataforma. Como se puede ver, este es un claro ejemplo de flujo híbrido que combina herramientas automáticas con procesos heurísticos manuales.

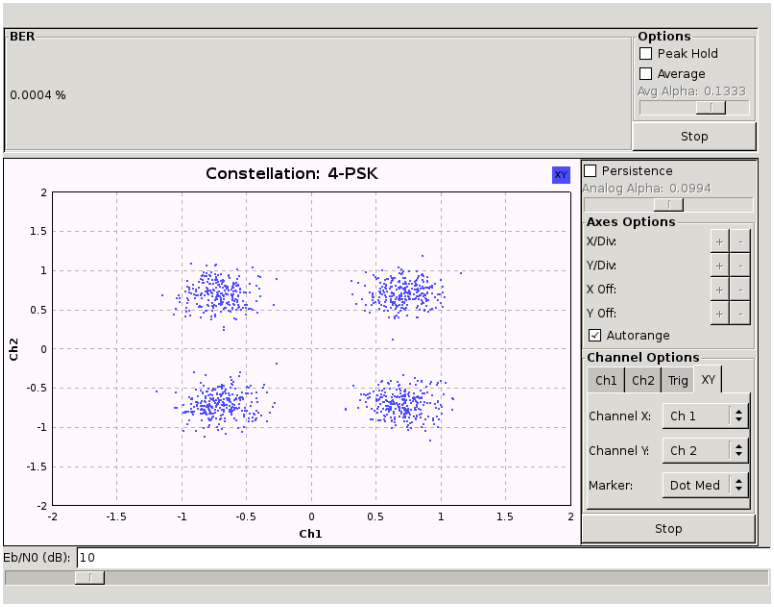
En el capítulo 3 se expondrán algunos resultados que se han obtenido dentro de este trabajo de tesis empleando esta metodología.

2.5.3.2 *GNU Radio*

GNU Radio [21, 80] es un kit de desarrollo libre y de código abierto para aplicaciones de radio *software*, creado inicialmente por Eric Blossom a principios de la década de 2000 a partir de Pspectra, un *software* desarrollado en el proyecto SpectrumWare [81] del Instituto de Tecnología de Massachussets (M.I.T.). En sus inicios fue usado principalmente por radioaficionados, pero al poco tiempo atrajo la atención del mundo académico y actualmente es muy empleado en investigación, especialmente en áreas como desarrollo de procotolos para comunicaciones inalámbricas, radio cognitiva, etc.



(a) Modelo GRC



(b) Resultado de la simulación

Figura 25: Modelo simulable realizado con GRC [82]

Básicamente, GNU Radio proporciona una biblioteca de bloques de procesado de señal, que implementan los algoritmos habituales en sistemas de radio *software* (filtrado, codificación, modulación, sincronización, etc.), y un planificador que permite conectar estos bloques entre sí para crear aplicaciones. El desarrollo se realiza en C++ y Python, aunque también hay disponible un editor gráfico llamado GRC (*GNU Radio Companion*) que permite construir las aplicaciones de modo similar a como se hace en Simulink, tal como se puede ver en la figura 25.

GNU Radio nació como un *software* para PC desarrollado en Linux, pero ha sido portado a otros sistemas, como por ejemplo el procesador CELL [83, 84]. Más recientemente, en [85] se describe el uso de GNU Radio en los SoC Zynq de Xilinx, que cuenta con dos procesadores ARM más lógica programable, con la intención de utilizar esta lógica programable para implementar coprocesadores *hardware*.

Como se describe en el capítulo 4, parte del trabajo desarrollado en esta tesis ha consistido precisamente en portar GNU Radio a determinados sistemas multiprocesador heterogéneos. Esto permite utilizar un flujo de diseño continuo, basado en este caso en GNU Radio, desde las primeras fases de modelado de alto nivel y exploración algorítmica hasta la implementación final en una arquitectura *hardware* adecuada para este tipo de aplicaciones, algo que como se ha podido comprobar a lo largo de esta sección no es frecuente en arquitecturas novedosas o poco convencionales.

2.6 RESUMEN

En este capítulo se ha contextualizado el trabajo de investigación desarrollado en esta tesis. Se ha presentado la aplicación de referencia empleada, el receptor DVB–T, analizando con cierto detalle la algoritmia por su interés en la posterior fase de implementación que se expone en el capítulo 3. A continuación se han descrito las características de la radio *software* por tratarse de una alternativa de diseño que resulta apropiada para la materialización de este tipo de aplicaciones. La complejidad del problema elegido conduce a la necesidad de emplear arquitecturas avanzadas, algunas de las cuales se describen en la sección 2.4, que presentan el inconveniente del coste de desarrollo sobre las mismas. De entre las diferentes alternativas, -las basadas en DSP, las arquitecturas multiprocesador y las basadas en procesadores reconfigurables-, se ha optado por arquitecturas multiprocesador heterogéneas. En la sección 2.5 se describen las metodologías y las dificultades de diseño sobre este tipo de plataformas a las que esta tesis realiza aportaciones.

3 DISEÑO DE ALTO NIVEL

Como ya se indicó en el capítulo 1, se ha escogido un receptor DVB–T como aplicación de referencia para validar la metodología de diseño que se propone en esta tesis y compararla con otras. En este capítulo se describe el proceso de diseño e implementación del receptor DVB–T mediante el uso de las dos herramientas de alto nivel descritas en el capítulo 2. Estas implementaciones tienen como objetivo evaluar dichas herramientas, identificar las ventajas e inconvenientes de cada una y analizar las posibilidades que ofrecen y los resultados que se pueden obtener de ellas en cada una de las fases del desarrollo de un sistema de radio *software*.

Por un lado, se ha implementado el receptor sobre una plataforma de desarrollo específica para aplicaciones de radio *software*, utilizando herramientas basadas en Matlab / Simulink. En primer lugar se ha creado un modelo funcional del receptor que ha servido para estudiar y seleccionar los algoritmos de procesado de señal adecuados. A continuación se han evaluado varias herramientas que en principio debían permitir llegar a una implementación de forma más o menos automática a partir de dicho modelo. Estas herramientas permiten además realizar un codiseño *hardware/software*. Por último, se ha realizado una implementación completa del receptor con la metodología que se ha considerado más adecuada tras la fase de evaluación. Este trabajo y las conclusiones que derivan de él se describen en el apartado 3.1.

Por otro lado, se ha desarrollado una versión del receptor sobre PC utilizando GNU Radio. Esta implementación, que ha servido después como base para el trabajo descrito en el capitulo 5, se describe en el apartado 3.2.

3.1 DISEÑO BASADO EN MATLAB / SIMULINK

Como ya se dijo anteriormente, Matlab es un estándar *de facto* en múltiples campos de la ciencia y la ingeniería, incluyendo las telecomunicaciones, y existen además varias herramientas que permiten en principio obtener implementaciones basadas tanto en FPGA como en procesadores empotrados a partir de modelos de alto nivel. Por todo ello fue la elección natural como referencia para explorar las posibilidades que ofrecen este tipo de herramientas.

La versión de Matlab que se ha utilizado ha sido la R2009b. Las herramientas de diseño automático que se han evaluado son:

- *Real-Time Workshop*. Este producto de Mathworks genera, a partir de un modelo Simulink, una aplicación en lenguaje C para PC o para algunos procesadores para sistemas empotrados, como por ejemplo DSPs de la familia C6000 de Texas Instruments.

- *HDL Coder*. Este producto, también de Mathworks, genera modelos *hardware* sintetizables en VHDL o Verilog a partir de funciones Matlab y modelos Simulink. El código generado puede emplearse en síntesis para FPGA o ASIC.

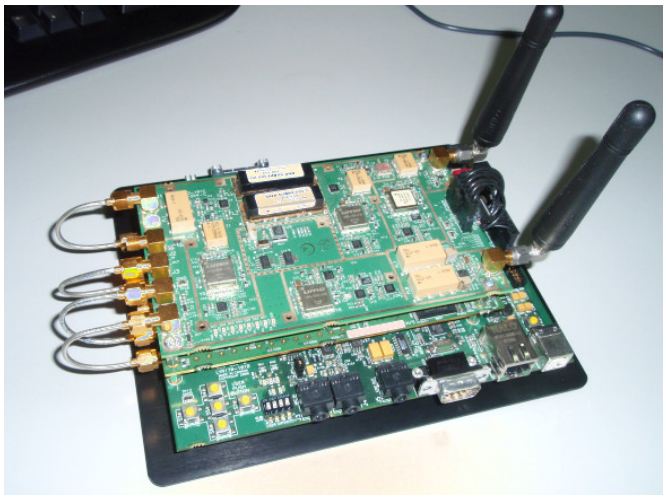
- *System Generator for DSP*. Es un producto de Xilinx que guarda cierta similitud con HDL Coder, aunque orientado a la síntesis para FPGAs de este fabricante. Su biblioteca de bloques incluye funciones básicas habituales en sistemas digitales como funciones aritméticas y lógicas, registros, contadores, etc., recursos concretos disponibles en las distintas FPGAs, y un amplio catálogo de IPs de alto nivel como funciones trigonométricas, FFT, filtros FIR e IIR, etc.

Para esta evaluación se ha empleado como plataforma de desarrollo un sistema denominado *SFF SDR Development Platform*, desarrollado y comercializado por la compañía canadiense Lyrtech Inc.[1] Este sistema, que puede verse en la figura 26a, está compuesto por tres módulos: uno de radiofrecuencia, otro de conversión A/D y D/A, y un tercero de procesado de señal. Este último cuenta con un procesador DaVinci TMS320DM6446 de Texas Instruments, que incluye un procesador de propósito general ARM9 y un DSP C64x+, y con una FPGA Virtex 4 SX35 de Xilinx. La figura 26b muestra un diagrama de bloques del sistema en el que se pueden ver las características de cada uno de los tres módulos [86].

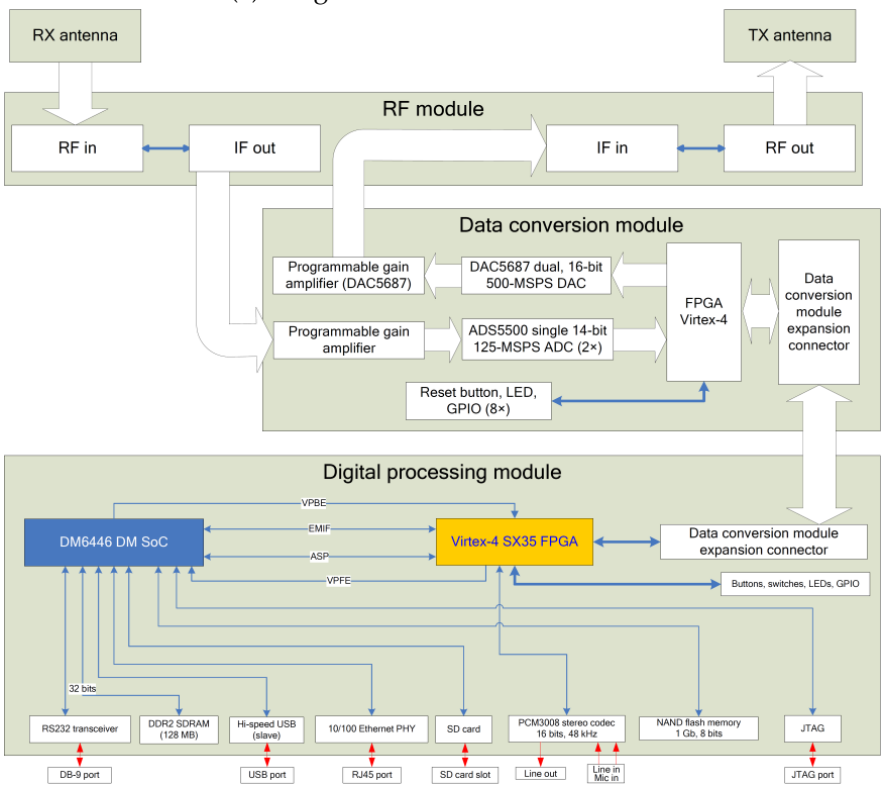
Este sistema ofrecía dos posibilidades para el desarrollo *hardware* y *software*. Por un lado existía la posibilidad de emplear flujos de diseño más o menos tradicionales basados en las herramientas de Xilinx (*ISE*) y Texas Instruments (*Code Composer Studio*), respectivamente. Por otro lado se ofrecía un kit de desarrollo basado en modelos que recibía el nombre de MBDK (*Model-based Design Kit*). Este MBDK estaba basado en Real-Time Workshop para la programación del procesador DM6446 y *System Generator for DSP* para la configuración de la FPGA.

Hay que señalar que el procesador incluido en el módulo de procesado de señal no es probablemente el más adecuado para aplicaciones de comunicaciones. La familia DaVinci estaba especialmente diseñada para el procesamiento de vídeo y no incluía algunos aceleradores *hardware* para comunicaciones que sí tenían otros DSPs del mismo fabricante. Sin embargo, en el momento de seleccionar el sistema de desarrollo

1 Lyrtech fue disuelta a finales del año 2011. Su portafolio de productos fue adquirido por la también canadiense Nutaq.



(a) Imagen del sistema *SFF SDR DP*



(b) Diagrama de bloques [86]

Figura 26: Sistema de desarrollo para radio *software SFF SDR Development Platform*

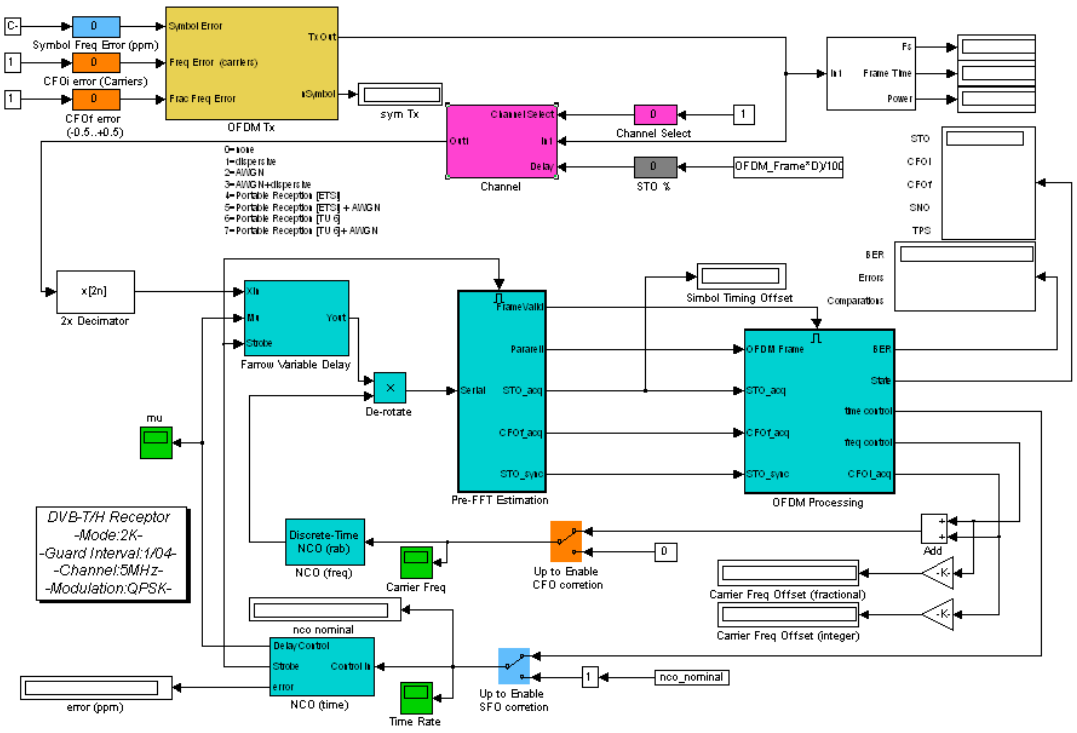


Figura 27: Modelo en Simulink del receptor DVB–T

no había ninguna otra alternativa que ofreciera características similares en un rango de precio comparable, y se consideraron especialmente interesantes las opciones que ofrecía en cuanto a herramientas de desarrollo que permitían evaluar diferentes alternativas metodológicas, en línea con los objetivos de esta tesis.

3.1.1 *Modelo Simulink del receptor*

La figura 27 muestra el nivel más alto del modelo Simulink del receptor DVB–T que se ha desarrollado en este trabajo. Consta de tres partes diferenciadas:

- Simulación del transmisor (color marrón).
- Simulación del canal de transmisión (color rosa).
- Receptor (color azul).

Las dos primeras partes obviamente no forman parte del receptor pero sirven para probarlo, validar su funcionamiento y caracterizar sus prestaciones. Cumplen por

tanto una función meramente instrumental, por lo que no se van a describir en detalle.

A la entrada del receptor se tiene la señal recibida muestreada en cuadratura y trasladada a banda base. A grandes rasgos, las tareas que debe realizar el receptor con esta señal son:

1. Determinar el instante de comienzo de cada símbolo OFDM y eliminar el prefijo cíclico.
2. Trasladar el símbolo OFDM al dominio de la frecuencia.
3. Extraer la información de las portadoras de datos.
4. Decodificar la información extraída y revertir la aleatorización.

Tras estos pasos se obtiene a la salida del receptor el flujo de transporte MPEG–2 que constituye la carga útil de la señal DVB–T, y que debe ser procesada posteriormente por un decodificador de audio y vídeo.

Los pasos 2 y 4 son casi inmediatos en esta etapa del desarrollo, ya que existen *blocksets* de Simulink (concretamente el *Communications Blockset*) que proporcionan las funcionalidades necesarias ya implementadas. Los pasos 1 y 3, por el contrario, deben implementarse partiendo prácticamente de cero, y requieren corregir previamente una serie de efectos indeseados que se introducen tanto en los equipos de transmisión y recepción como en el canal de comunicación. Estos efectos no deseados son:

- STO (*Symbol Timing Offset*). Es el error que se produce en la estimación del comienzo de cada símbolo OFDM en el receptor.
- SFO (*Sampling Frequency Offset*). Es la error entre la frecuencia de operación del conversor D/A del transmisor y el conversor A/D del receptor. Suele expresarse como la diferencia relativa entre el período de muestreo en el receptor (T') y en el transmisor (T): $\zeta = (T' - T)/T$.
- CFO (*Carrier Frequency Offset*). Es la diferencia entre la frecuencia central real empleada por el transmisor y el receptor, debida a las desviaciones respecto a la frecuencia nominal de los osciladores que se usan en las etapas de radiofrecuencia del transmisor y el receptor para subir (bajar) la señal a (de) la banda de transmisión. En sistemas OFDM suele medirse en múltiplos de la separación entre portadoras y es frecuente considerar por separado el CFO entero (*CFOi*) y fraccionario (*CFOf*)[2].

2 Por ejemplo, si el CFO es 2,37 veces la distancia entre portadoras el CFOi sería 2 y el CFOf sería 0,37.

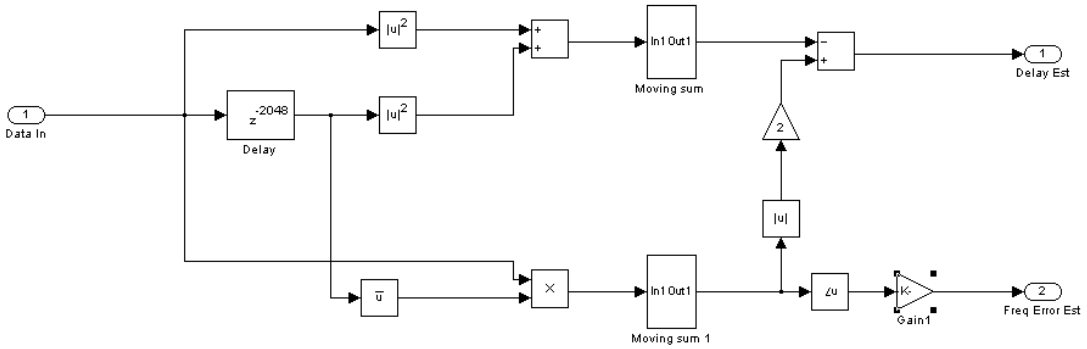


Figura 28: Modelo Simulink del estimador del comienzo de símbolo y del CFOf según el algoritmo descrito en [88]

- Efecto del canal. Además de introducir ruido, el canal de comunicación es *selectivo*: la atenuación y el retardo o desviación de fase que sufre la señal no son homogéneos en toda la banda utilizada.

Buena parte del trabajo de desarrollo del receptor consiste en la elección e implementación de algoritmos que permitan estimar estos errores. Una vez estimados, es relativamente sencillo corregir sus efectos, al menos en parte:

- El STO y el SFO se pueden corregir mediante un filtro de retardo fraccionario variable, que se ha implementado según se describe en [87, p. 520]. En el diagrama de la figura 27 es el bloque etiquetado como "*Farrow Variable Delay*". El filtro se gobierna mediante un oscilador controlado numéricamente (NCO, *Numerically Controled Oscillator*) etiquetado como "*NCO (time)*".
- El CFO es muy sencillo de corregir desplazando la señal en frecuencia. Al trabajar con muestreo en cuadratura esto implica únicamente multiplicar la señal de entrada por una señal exponencial compleja de la frecuencia adecuada, tal como se hace en el diagrama de la figura 27 a la salida del filtro de retardo fraccionario variable mencionado en el punto anterior.
- La selectividad del canal puede mitigarse en buena medida mediante ecualización, que es sencilla de realizar en el dominio de la frecuencia. La estimación de canal es también relativamente simple, dado el gran número de pilotos que contiene cada símbolo OFDM.

En el modelo de la figura 27 los bloques "*Pre-FFT Estimation*" y "*OFDM Processing*" son los que realizan la mayor parte del procesado. En el primero se realiza el proce-

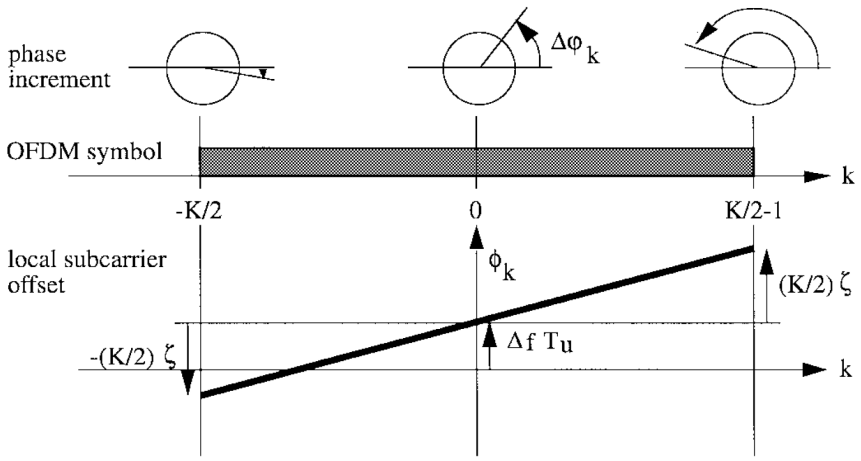


Figura 29: Efecto del CFO (Δf) y el SFO (ζ) en la fase de las portadoras individuales del símbolo OFDM recibido [89]

sado en el dominio del tiempo, que incluye la estimación del instante de comienzo del símbolo OFDM, del STO y del CFOf. Para ello se utiliza el algoritmo descrito en [88], cuyo detalle de implementación se muestra en la figura 28. Este fragmento del receptor se ha utilizado para evaluar el funcionamiento de las herramientas de síntesis automática, tal como se describe más adelante.

El bloque "*OFDM Processing*" realiza el procesado de la señal en el dominio de la frecuencia, que incluye la estimación y corrección del SFO, la estimación de canal, la ecualización y la decodificación de la información hasta la obtención del flujo de transporte MPEG–2. Para la estimación del SFO se utiliza el método descrito en [89]: según se puede ver en la Figura 29, el SFO produce una rotación en los símbolos transmitidos en cada portadora que es proporcional al índice de la portadora y a la magnitud del SFO. Este puede estimarse, por tanto, a partir de la rotación que experimentan los pilotos, ya que su fase es conocida.

3.1.2 *Evaluación de herramientas automáticas de implementación*

Para evaluar el funcionamiento de las herramientas de generación automática de código que se han mencionado anteriormente se seleccionó una parte del receptor DVB–T que fuera a la vez lo suficientemente pequeña para poder ser modificada según los requerimientos de cada herramienta, y lo suficientemente importante en términos de requisitos de cálculo para que los resultados obtenidos fuesen significativos. El subsistema seleccionado fue el estimador de tiempo de inicio del símbolo

	Ciclos de reloj
Real-Time Workshop	975 829 506
Código manual	123 554 545

Tabla 1: Comparación entre ciclos de reloj consumidos por el código generado por Real-Time Workshop y el codificado manualmente [90]

y del CFO mostrado en la figura 28. A continuación se presenta un resumen de los resultados más significativos. El trabajo completo puede consultarse en [90].

3.1.2.1 *Real-Time Workshop*

Se comparó el rendimiento del código generado para el DSP del procesador DM6446 por Real-Time Workshop a partir del modelo ya presentado en la figura 28 con una implementación del mismo algoritmo codificada a mano en lenguaje C por un programador con aproximadamente un año de experiencia en desarrollo con DSPs de la familia C6000.

La tabla 1 muestra el número de ciclos de reloj empleado por ambas implementaciones del algoritmo en el simulador del procesador DM6446 integrado en el *Code Composer Studio*, el entorno de desarrollo de Texas Instruments. La columna "Sólo CPU" corresponde a una versión del simulador en la que no se tienen en cuenta las pausas en la ejecución debidas a los accesos a memoria o a operaciones de entrada/salida, mientras que la columna "Incl. cache" corresponde a una versión del simulador que sí tiene en cuenta la jerarquía de memoria caché y los retardos producidos por los accesos a memoria. Para cada versión del código (generado automáticamente o codificado manualmente) se muestra el número total de ciclos de reloj empleados para procesar 10 símbolos OFDM en modo 2K con un prefijo cíclico de 1/4 de símbolo y el número de ciclos de reloj empleados por una única operación del algoritmo, el cálculo del módulo de un valor complejo. Para este último caso únicamente se registraron los resultados de la simulación que incluía el efecto de los accesos a memoria.

Como puede verse, el código manual resultó ser mucho más eficiente que el generado por Real-Time Workshop con un tiempo de ejecución un 87 % inferior, casi un orden de magnitud por debajo. Estos resultados son coherentes con estudios similares publicados en la literatura cientifico-técnica, como por ejemplo en [91]. Por otro lado, además de esta diferencia de rendimiento hubo ciertos problemas para conseguir un entorno de desarrollo plenamente operativo debido a las diferencias en las versiones de alguna de las herramientas que eran requeridas por otras. Este tipo de

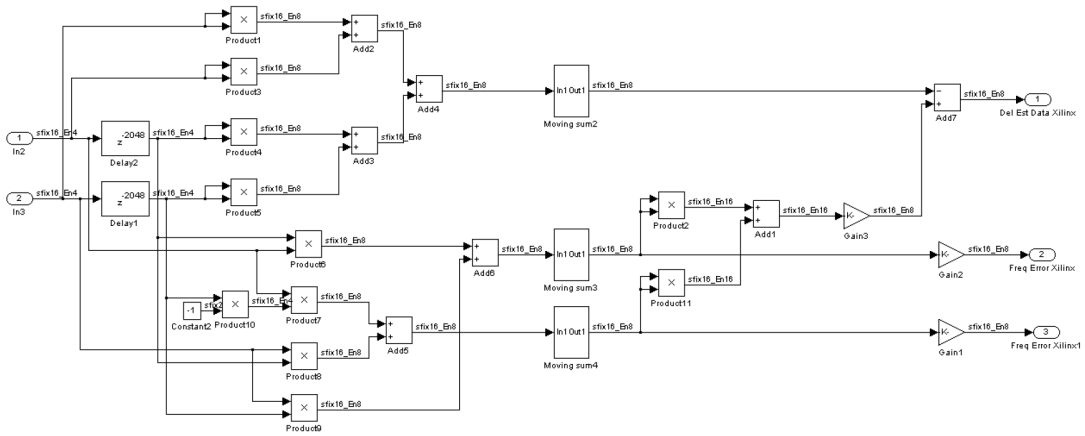


Figura 30: Estimador del comienzo de símbolo y del CFOf modificado para la síntesis con HDL Coder [90]

inconsistencias no son extrañas cuando se combinan herramientas de varios fabricantes, y en ocasiones pueden dificultar seriamente el proceso de desarrollo en este tipo de flujos de diseño.

Por otra parte, el tiempo de desarrollo de la versión codificada a mano fue de aproximadamente una semana, incluyendo el tiempo necesario para crear un banco de pruebas que permitiera verificar el correcto funcionamiento del código. En el caso del Real-Time Workshop el desarrollo fue mucho más rápido, unos dos días.

3.1.2.2 *HDL Coder*

En este caso el primer escollo que se encontró al usar esta herramienta es que permite usar únicamente un subconjunto tanto de los tipos de datos como de la biblioteca de bloques de Simulink. Notablemente, no se puede usar el tipo de datos complejo, que se empleaba de forma extensiva en todo el modelo del receptor. Tampoco se podían usar, por ejemplo, los bloques de cálculo del módulo al cuadrado o de la fase que se emplean en el modelo de la figura 28. Por tanto, fue necesario modificar de forma significativa dicho modelo para adaptarlo a los requisitos de HDL Coder.

El resultado de esta adaptación es el modelo que se muestra en la figura 30. En él, se han separado explícitamente los datos complejos en parte real e imaginaria, y se ha desarrollado el cálculo del módulo en sus operaciones básicas constituyentes. También se puede observar que el modelo no es totalmente equivalente al mostrado en la figura 28 ya que se ha omitido una operación de cálculo de fase. Esta opera-

```
Device utilization summary:
---------------------------

Selected Device : 4vsx35ff668-10

 Number of Slices: 51237 out of 15360 333% (*)
 Number of Slice Flip Flops: 90211 out of 30720 293% (*)
 Number of 4 input LUTs: 224 out of 30720 0%
 Number of IOs: 82
 Number of bonded IOBs: 0 out of 448 0%
 Number of DSP48s: 12 out of 192 6%

WARNING:Xst:1336 - (*) More than 100% of Device resources are used
```

Figura 31: Extracto del informe de síntesis de ISE [90]

ción requiere el cálculo de la arcotangente, que no es fácilmente sustituible por otras operaciones elementales. Dado que el objetivo en este momento no era tanto obtener una implementación funcionalmente completa como evaluar las capacidades de HDL Coder, se decidió eliminarla.

Tras obtener el modelo de la figura 30, que ya era sintetizable, se comprobó que el resultado de la síntesis no era satisfactorio ya que los recursos utilizados excedían los disponibles en la FPGA de la tarjeta, como se puede ver en el extracto del informe de síntesis que se muestra en la figura 31.

Tras investigar la causa de esta elevada utilización de recursos, que no parecía en modo alguno justificada por la complejidad del modelo, se descubrió que el problema consistía en que se estaban utilizando los *flip-flops* de los bloques de lógica configurable de la FPGA en lugar de los bloques de RAM para implementar la memoria necesaria para los bloques de retardo que contiene el modelo.

La conclusión a la que se llegó tras la realización de estas pruebas es que la utilidad práctica de HDL Coder para sistemas de radio *software* es limitada, debido a que por un lado las limitaciones en cuanto a bloques de Simulink sintetizables hacen necesaria una adaptación profunda de los modelos, que llega a ser prácticamente una reescritura, y por otro lado porque el resultado de la síntesis de algunos bloques puede no ya no ser óptimo, sino ser de tan baja calidad que impida obtener una implementación físicamente realizable.

3.1.2.3 *System Generator for DSP*

Al contrario que las anteriores, que son más o menos genéricas, esta herramienta es específica para las FPGAs de un determinado fabricante, en este caso Xilinx. Por

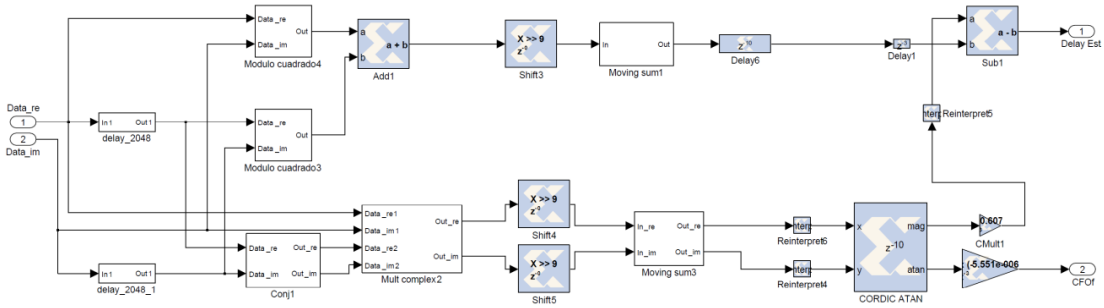


Figura 32: Estimador del comienzo de símbolo y del CFOf modificado para la síntesis con *System Generator for DSP* [92]

un lado, es mucho más utilizable que HDL Coder ya que proporciona acceso a la práctica totalidad del catálogo de IPs de Xilinx, que incluye algoritmos típicos de procesado de señal como filtros, FFT, etc. Pero por otro lado, la herramienta no oculta los detalles de bajo nivel del diseño *hardware*. Cualquier algoritmo que no se encuentre ya implementado en el mencionado catálogo de IPs debe ser construido a partir de bloques de la biblioteca de *System Generator for DSP*, lo que supone en la práctica emplear una metodología muy similar a la típica de las herramientas de diseño de circuitos digitales basadas en captura de esquemas.

La figura 32 muestra la reimplementación con *System Generator for DSP* del estimador del comienzo de símbolo. Todos los bloques que tienen de fondo el logotipo de Xilinx pertenecen a la biblioteca de *System Generator for DSP*, y los que tienen fondo blanco son bloques compuestos que implementan operaciones como el producto complejo o una línea de retardo que están a su vez construidos con bloques de dicha biblioteca.

En este caso se dio el mismo problema con los bloques de retardo que se describió en el apartado anterior. Sin embargo, en esta ocasión sí pudo solventarse satisfactoriamente rediseñando estos bloques, utilizando explícitamente bloques de memoria RAM de la biblioteca de *System Generator for DSP*. Aunque no parece un procedimiento muy deseable teniendo en cuenta que se supone que se desea mantener un nivel de abstracción relativamente alto, al menos con esta herramienta es posible resolver cualquier situación de este tipo.

3.1.3 *Implementación en la* SFF SDR Development Platform

Vistos los resultados obtenidos tras las pruebas descritas en las secciones anteriores, se decidió realizar el desarrollo de la parte *software* programando el DSP directamente

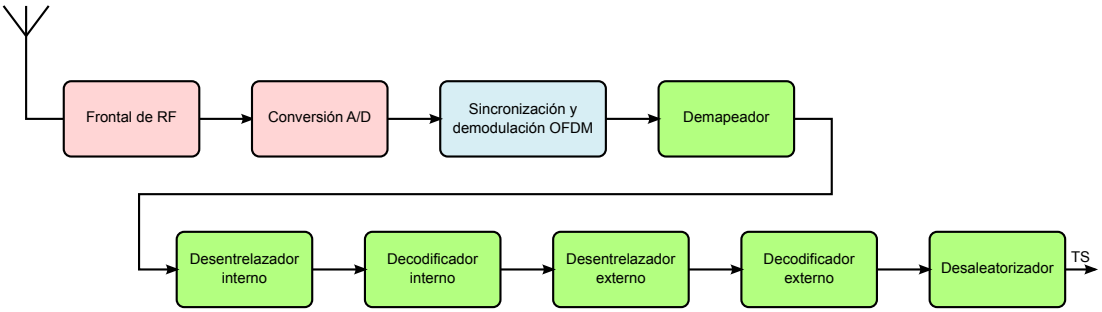


Figura 33: Diagrama de bloques del receptor DVB–T [92]

en lenguaje C, y emplear el *System Generator for DSP* de Xilinx para desarrollar la parte *hardware*.

La partición *hardware/software* se decidió en base a una estimación de la capacidad de procesamiento disponible en el DSP. La figura 33 muestra este reparto sobre el diagrama de bloques del receptor. Los bloques con fondo rojizo corresponden a los módulos de radiofrecuencia y de conversión de datos del sistema *SFF SDR DP*, el bloque con fondo azul se implementó en la FPGA del módulo de procesado de datos, y los bloques con fondo verde se implementaron en el núcleo DSP del procesador DM6446.

Los resultados de este trabajo están descritos con detalle en [92]. A continuación se resumen las conclusiones más significativas.

En cuanto al desarrollo de la parte *hardware* implementada en la FPGA, la herramienta *System Generator for DSP* permite obtener un buen resultado, entendiendo como tal una implementación eficiente tanto en prestaciones como en consumo de recursos. Sin embargo, en cuanto a metodología de diseño, se pudo comprobar que los beneficios que proporciona esta herramienta son limitados, ya que es necesaria una reimplementación completa que en no pocos casos requiere un rediseño a bajo nivel de determinados bloques.

En cuanto al desarrollo del *software* para el DSP, se realizó utilizando la metodología recomendada por el fabricante empleando su *Reference Framework 5* (RF5) [93], basado en el sistema operativo DSP/BIOS [94]. Cada uno de los bloques funcionales con fondo verde en la figura 5 se codificó en una tarea separada, y las tareas se comunican entre sí mediante colas de mensajes SCOM [95].

Si bien no hay muchas conclusiones que obtener de esta parte del desarrollo en cuanto a metodología, ya que la que se utilizó es la que puede considerarse tradicional, sí resultó útil como punto de comparación con el desarrollo sobre GNU Radio que se describe más adelante.

3.1.4 *Conclusiones*

Tras el proceso descrito, la principal conclusión que se ha obtenido acerca de las herramientas evaluadas aplicadas al ámbito de la radio *software* es que, si bien facilitan en gran medida el diseño de alto nivel del sistema (exploración algorítmica, validación funcional del sistema, etc.), su utilidad a la hora de obtener una implementación eficiente del mismo en una plataforma especializada es limitada, al menos con las versiones de las herramientas disponibles en el momento de la evaluación. En el desarrollo de *software* se ha comprobado que Real-Time Workshop permite obtener rápidamente una implementación funcionalmente correcta, pero con un rendimiento muy pobre, lo que la descarta para aplicaciones que, como el receptor DVB–T, requieren una gran capacidad de proceso.

En cuanto a las herramientas para diseño *hardware*, se ha comprobado que HDL Coder presenta unas limitaciones tan severas que descartan su uso en cualquier aplicación realmente compleja. La alternativa es utilizar la herramienta del fabricante de *hardware*, que permite obtener un resultado bueno a costa de hacer visibles muchos detalles de bajo nivel. Esto perjudica inevitablemente la productividad, que se supone que es el punto fuerte de este tipo de herramientas.

La implementación del receptor DVB–T en la plataforma *SFF SDR Development Platform* se publicó en una revista indexada [96] y constituye una de las aportaciones de este trabajo de tesis (véase la sección 7.2.1.1).

3.2 DISEÑO BASADO EN GNU RADIO

La implementación del receptor DVB–T sobre GNU Radio, que se describe en profundidad en [97], se basó en gran medida en el trabajo descrito en la sección 3.1.1. Tanto la estructura del receptor como los algoritmos empleados son prácticamente idénticos, por lo que no se va a repetir su descripción.

Hay que destacar que se ha constatado que el proceso de exploración algorítmica podría haberse realizado utilizando GNU Radio en vez de Simulink sin inconvenientes significativos, y en algunos casos incluso con mayor facilidad, aunque sin duda la facilidad de uso tiene una fuerte componente subjetiva dependiente de la familiaridad que se tenga con uno u otro entorno.

Durante la implementación del receptor DVB–T se han podido distinguir dos casos de uso bien diferenciados. En primer lugar, existe la posibilidad de que un determinado algoritmo pueda implementarse conectando entre sí bloques ya disponibles en la biblioteca de GNU Radio. Un ejemplo de este caso es el estimador de comienzo de símbolo que se presentó en la figura 28, y que se utilizó como referencia en la

```
#!/usr/bin/env python

import math
from gnuradio import gr

class ofdm_sync_vdb(gr.hier_block2):
  def __init__(self, fft_length, cp_length):
    gr.hier_block2.__init__(self, "ofdm_sync_vdb",
    gr.io_signature (1, 1, gr.sizeof_gr_complex), # Input signature
    gr.io_signature2(2, 2, gr.sizeof_float,       # Output signature
    gr.sizeof_float))

    # Block instantiation
    # mid / upper branch
    self.delay = gr.delay(gr.sizeof_gr_complex, fft_length)
    self.magsqrd1 = gr.complex_to_mag_squared()
    self.magsqrd2 = gr.complex_to_mag_squared()
    self.adder = gr.add_ff()
    self.avgfilt1 = gr.fir_filter_fff(1, [0.5 for i in range(cp_length)])
    self.diff = gr.sub_ff()
    # lower branch
    self.conjg = gr.conjugate_cc();
    self.mult = gr.multiply_cc();
    self.avgfilt2 = gr.fir_filter_ccf(1, [1.0 for i in range(cp_length)])
    self.c2mag = gr.complex_to_mag()
    self.angle = gr.complex_to_arg()

    # Inner connections
    self.connect(self, self.magsqrd1, (self.adder,0))
    self.connect(self, self.delay, self.magsqrd2, (self.adder,1))
    self.connect(self.adder, self.avgfilt1, (self.diff,1))
    self.connect(self.delay, self.conjg, (self.mult,0))
    self.connect(self, (self.mult,1))
    self.connect(self.mult, self.avgfilt2, self.c2mag, (self.diff,0))
    self.connect(self.avgfilt2, self.angle)

    # Outputs
    self.connect(self.diff, (self,0))
    self.connect(self.angle, (self,1))
```

Listado 1: Código Python correspondiente al diagrama de bloques de la figura 34

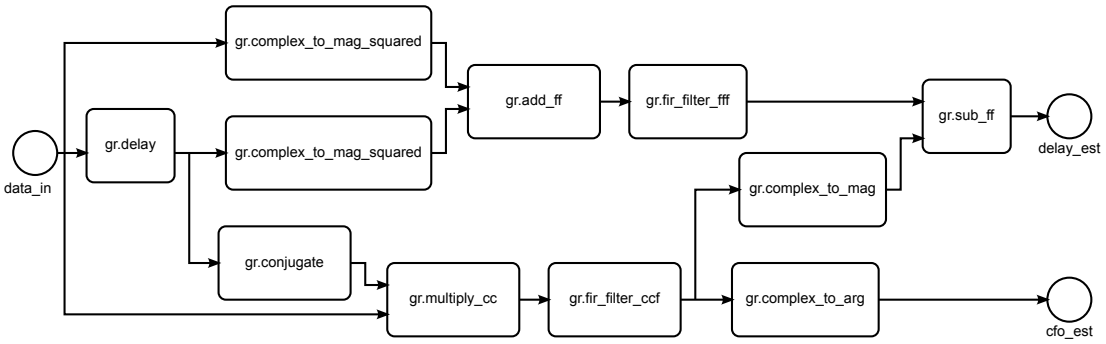


Figura 34: Diagrama de bloques del estimador del comienzo de símbolo y del CFOf implementado en GNU Radio [97]

evaluación de las herramientas de diseño automático descrita en la sección 3.1.2. La figura 34 muestra el diagrama de bloques de dicho algoritmo tal como se implementó en lenguaje Python. El código correspondiente se muestra en el listado 1. Como puede comprobarse, la construcción de bloques jerárquicos (es decir, construidos mediante composición de otros bloques) resulta muy sencilla.

Por otro lado, existen algoritmos que o bien no pueden construirse de este modo o bien resulta más sencillo codificarlos directamente. Aunque esto también puede hacerse en Python, lo más común por motivos de rendimiento es hacerlo en C++. En este caso se ha comprobado que, si se dispone ya de alguna implementación en C o C++, resulta bastante sencillo crear un nuevo bloque que actúe como *carcasa* o *wrapper*, como suele denominarse, para reutilizar el código disponible. De hecho, varios bloques de la biblioteca de GNU Radio están construidos de este modo, como por ejemplo los de cálculo de la FFT, que utilizan en realidad una librería muy extendida denominada FFTW [98]. En el desarrollo del receptor esta reutilización de código se produjo en varias ocasiones, tanto con código disponible públicamente con licencia libre, como el empleado en el decodificador Reed-Solomon [99], como para reutilizar código desarrollado para la implementación descrita en la sección 3.1.3.

La tabla 2 muestra el rendimiento obtenido por el receptor en un PC con procesador Intel Core 2 Q8200 (4 núcleos a 2,33 GHz). La prueba consistió en medir el tiempo empleado en la recepción de un flujo de transporte MPEG–2 con un tamaño total de 1 790 512 bytes (9 524 paquetes de transporte) para cada una de las modulaciones de las portadoras de datos posibles. En todos los casos el intervalo de guarda es 1/2 y no se aplica punteado tras la codificación interna. Para cada caso se muestra la capacidad nominal a nivel de flujo de transporte, el tiempo empleado en la recepción y la capacidad efectiva del receptor.

Modo	Modulación	C. nominal (Mbps)	Tiempo (s)	Capacidad (kbps)
2K	QPSK	4,98	45,85	312
	16QAM	9,95	44,29	323
	64QAM	14,93	44,38	323

Tabla 2: Rendimiento obtenido por el receptor implementado con GNU Radio [97]

Como puede verse el rendimiento obtenido, que está limitado por el decodificador interno (es decir, por el decodificador de Viterbi [100]), dista mucho de alcanzar el necesario para llegar al funcionamiento en tiempo real, aunque es comparable al obtenido por desarrollos similares. Por ejemplo, en [101] se cita un rendimiento inferior en un orden de magnitud al necesario para llegar al funcionamiento en tiempo real en una implementación parcial del receptor, aunque los resultados no son directamente comparables, y en [102], en un trabajo del mismo grupo, se cita un rendimiento únicamente para el decodificador de Viterbi casi 8 veces inferior al necesario para llegar a tiempo real, aunque no se dan detalles concretos sobre los parámetros de DVB–T empleados. En cualquier caso no se ha dedicado esfuerzo a optimizar el rendimiento debido a que el interés principal de este trabajo está en la metodología.

3.2.1 *Conclusiones*

Tras el trabajo descrito en esta sección se ha constatado que GNU Radio proporciona un entorno válido para el desarrollo de aplicaciones de radio *software* complejas. Por otro lado, existen precedentes del uso de esta herramienta en arquitecturas multiprocesador heterogéneas, ya que en [83, 84] se describe cómo emplear GNU Radio para programar el procesador Cell [103], que contenía un núcleo PowerPC de propósito general y ocho procesadores SIMD. Todo ello permite pensar que es un buen candidato para servir como base de una metodología de desarrollo que permita abarcar desde el primer desarrollo de alto nivel hasta la implementación final en un sistema empotrado.

3.3 RESUMEN

En este capítulo se han evaluado dos herramientas de alto nivel para el diseño de la aplicación tomada como referencia en esta tesis.

La primera de ellas (Matlab / Simulink) se ha revelado muy cómoda para el diseño de alto nivel y la exploración del espacio de diseño, pero poco eficiente de cara a la implementación final, tanto en la generación de *software* ejecutable en el procesador de la arquitectura, como en la generación de código HDL para la síntesis de hardware en FPGA.

La segunda herramienta evaluada (GNU Radio) también resulta práctica para el diseño de alto nivel y se ha comprobado que el código resultante tiene un buen rendimiento. El inconveniente de GNU Radio es que emplearlo fuera del entorno PC para el que ha sido concebido, implica en primer lugar portarlo a la arquitectura objetivo (DSP, GPP o arquitectura multiprocesador), aunque existe un trabajo previo que sugiere que es viable hacerlo. Es por ello que, dentro del trabajo desarrollado en esta tesis, se ha realizado el portado de GNU Radio a diferentes plataformas con objeto de poder definir una metodología general que facilite su portado a cualquier plataforma. Este trabajo y las conclusiones metodológicas se describen en los capítulos 4, 5 y 6.

4

PORTADO Y EXTENSIÓN DE GNU RADIO

En el capítulo 3 se expusieron las ventajas de utilizar GNU Radio como herramienta de alto nivel para el diseño de sistemas de radio *software* sobre arquitecturas multiprocesador, así como sus dos inconvenientes fundamentales: el primero, que se trata de una herramienta para PC, y el segundo, que su planificador no está diseñado para arquitecturas multiprocesador heterogéneas. En este capítulo se presenta la metodología que, con carácter general[1], puede emplearse para portar GNU Radio a otras plataformas, así como una propuesta de modificación de la arquitectura *software* de la herramienta, extendiéndola para que pueda ser empleada en arquitecturas multiprocesador heterogéneas haciendo uso de todos los procesadores disponibles.

En primer lugar se explica en la sección 4.1 el funcionamiento interno de GNU Radio, para a continuación exponer en la sección 4.2 la arquitectura *software* propuesta para distribuir tareas entre varios procesadores. Esta sección consta de cuatro partes: en primer lugar se discute la adaptabilidad de GNU Radio a sistemas empotrados, después se introduce el mecanismo de envío de trabajos a otros procesadores en un caso sencillo, a continuación se expone el mecanismo completo de funcionamiento en el caso más general, y por último se presentan detalladamente los algoritmos que definen el funcionamiento de cada aspecto de la arquitectura *software* propuesta.

4.1 FUNCIONAMIENTO DE GNU RADIO

Como se describió en el capítulo 2, GNU Radio consiste básicamente en dos elementos:

- Una infraestructura *software* para la construcción de aplicaciones de procesado digital de señal, que incluye:
  - Un mecanismo de definición de bloques de procesado de señal.
  - Un mecanismo para conectar estos bloques entre sí mediante *buffers* de memoria, formando un grafo de procesado de señal.
  - Un planificador que gestiona la ejecución de cada uno de los bloques que componen el grafo en función del estado de sus *buffers* de entrada y salida.

1 Los detalles particulares para las arquitecturas *hardware* evaluadas se presentan en el capítulo 5.

- Una biblioteca de bloques de procesado de señal.

Una aplicación GNU Radio consiste en construir un grafo de procesado de señal empleando bloques de la biblioteca, y después lanzar su ejecución. Existen dos tipos de bloques especiales denominados *fuentes* (*sources*) y *sumideros* (*sinks*) que actúan, como su nombre indica, como fuentes o sumideros de señal. Los bloques fuente pueden generar la señal internamente u obtenerla de un fichero, otra aplicación o algún dispositivo *hardware* como una tarjeta de sonido o un periférico que haga de frontal de radiofrecuencia; en concreto en las pruebas experimentales desarrolladas en esta tesis se han empleado frontales de la familia USRP (*Universal Software Radio Peripheral*) [104]. De igual modo, un bloque sumidero puede enviar la señal que recibe a un fichero, una aplicación o un periférico, mostrarla por pantalla o simplemente descartarla.

El núcleo de GNU Radio está programado en C++, y este es el lenguaje que se utiliza también para codificar los bloques de procesado de señal. Puede emplearse también para construir grafos y aplicaciones, pero lo más habitual es utilizar para esto el lenguaje Python[2], ya que todos los elementos de GNU Radio se exportan a este lenguaje mediante una herramienta denominada *SWIG* [105]. Existe también una herramienta denominada GRC (*GNU Radio Companion*) que permite construir las aplicaciones mediante una interfaz gráfica y genera automáticamente el código Python correspondiente.

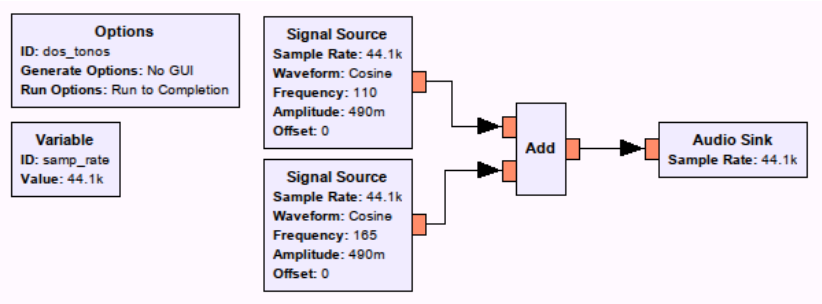


Figura 35: Ejemplo de aplicación GNU Radio realizada con GNU Radio Companion: suma de dos señales sinusoidales

La figura 35 muestra un ejemplo de aplicación GNU Radio sencilla realizada con GNU Radio Companion, que genera dos señales sinusoidales, las suma y envía la señal resultante a la tarjeta de sonido del ordenador. El listado 2 muestra el código

2 También se pueden codificar bloques de procesado de señal en Python, pero por motivos de rendimiento no es la opción más habitual.

```
#!/usr/bin/env python
##################################################
# Gnuradio Python Flow Graph
# Title: Dos Tonos
# Generated: Thu Nov 12 12:37:28 2015
##################################################

from gnuradio import analog
from gnuradio import audio
from gnuradio import blocks
from gnuradio import gr

class dos_tonos(gr.top_block):

    def __init__(self):
        gr.top_block.__init__(self, "Dos Tonos")

        ##################################################
        # Variables
        ##################################################
        self.samp_rate = samp_rate = 44100

        ##################################################
        # Blocks
        ##################################################
        self.src_0 = analog.sig_source_f(samp_rate, analog.GR_COS_WAVE, 110, 0.49, 0)
        self.src_1 = analog.sig_source_f(samp_rate, analog.GR_COS_WAVE, 110*3/2, 0.49, 0)
        self.sum_0 = blocks.add_vff(1)
        self.sink_0 = audio.sink(samp_rate, "", False)

        ##################################################
        # Connections
        ##################################################
        self.connect((self.src_0, 0), (self.sum_0, 0))
        self.connect((self.src_1, 0), (self.sum_0, 1))
        self.connect((self.sum_0, 0), (self.sink_0, 0))

if __name__ == '__main__':
    tb = dos_tonos()
    tb.start()
    tb.wait()
```

Listado 2: Código Python generado a partir del grafo de la figura 35

```
int
add_ff::work(int noutput_items,
gr_vector_const_void_star &input_items,
gr_vector_void_star &output_items)
{
  // These variables are not really needed but they improve readability
  const float *ino = (const float *) input_items[0];
  const float *in1 = (const float *) input_items[1];
  float       *out = (float *) output_items[0];

  // This is where the real work is done
  for (int i = 0; i < noutput_items; ++i) {
    out[i] = ino[i] + in1[i];
  }

  // Return the number of output items produced
  return noutput_items;
}
```

Listado 3: Ejemplo de código C++ para un bloque que suma dos señales

Python resultante, generado por el propio GNU Radio Companion a partir del esquema gráfico. En el código Python puede verse cómo crear una aplicación consiste en crear un bloque jerárquico de un tipo específico que se emplea para definir un grafo, derivándolo de una clase denominada *top_block*. En la inicialización de la clase derivada, denominada "dos_tonos", se instancian los cuatro bloques que componen el grafo y posteriormente se conectan. Como se observa al final del listado 2, el código de la aplicación consiste únicamente en instanciar un bloque del tipo que se acaba de definir y lanzar su ejecución.

Codificar un bloque de procesado de señal resulta bastante sencillo. GNU Radio define una jerarquía de clases C++, que mediante SWIG es utilizable también desde Python. En el caso más simple, crear un nuevo bloque de procesado de señal consiste únicamente en definir una nueva clase derivada de una clase base adecuada y redefinir un método denominado *work* en la clase derivada. Este método recibe tres parámetros: la lista de *buffers* de entrada, la lista de *buffers* de salida, y el número de datos de salida que debe producir. Cada vez que se dan las condiciones necesarias para que el bloque entre en ejecución el planificador llama a este método.

El listado 3 muestra un ejemplo de cómo se podría codificar el método *work* de un bloque que suma dos señales. Como se puede ver, el código consiste básicamente en un bucle que recorre los *buffers* de entrada sumando cada par de muestras y dejando el resultado en el *buffer* de salida.

Existe también la posibilidad de crear un bloque jerárquico, formado mediante la interconexión entre sí de varios bloques, lo que permite emplear la metodología de diseño jerárquico habitual en sistemas complejos.

4.1.1 *Funcionamiento del planificador*

Existen dos versiones del planificador en GNU Radio que difieren en cómo gestionan los hilos (*threads*) en la aplicación. La versión más simple, que fue la primera disponible, emplea un único hilo para todos los bloques, de modo que éstos van ejecutándose secuencialmente. Este planificador recibe el nombre de *STS* (*Single Thread Scheduler*).

A mediados de 2008 se introdujo otro planificador que crea un hilo por cada bloque de procesado de señal, llamdo *TPB* (*Thread Per Block*). Este es el planificador que actualmente se emplea por defecto, ya que permite explotar el paralelismo a nivel de tarea que implícitamente existe en el grafo de procesado de señal y sacar partido a los procesadores multinúcleo que son habituales en los PCs desde hace ya varios años.

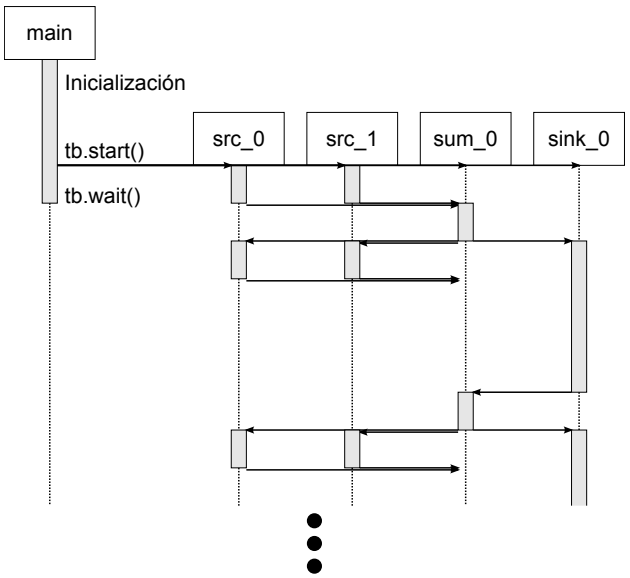


Figura 36: Diagrama de ejecución de la aplicación presentada en la figura 35

La figura 36 muestra un ejemplo simplificado de cómo se podría ejecutar la aplicación de la figura 35 utilizando el planificador TPB. Cada columna representa un hilo (*thread*). El tiempo transcurre hacia abajo y los estados "en ejecución" y "suspendido"

de cada hilo se representan mediante un trazo grueso con fondo gris y una línea discontinua, respectivamente.

La aplicación comienza con un único hilo, etiquetado como *main* en la figura, en el que se ejecuta la inicialización contenida en el método *__init__* en el código del listado 2. En este paso se crean los bloques y, al conectarlos entre sí, se crean también los *buffers* de memoria que se utilizan para el paso de datos de unos a otros, uno por cada pareja de salida y entrada. A continuación, la llamada al método *start()* del bloque que contiene el grafo crea los cuatro hilos (uno por bloque) etiquetados en la figura como *src_0*, *src_1*, *sum_0* y *sink_0*.

En cada uno de estos hilos el planificador entra entonces en un bucle en el que se ejecuta el método de procesado de señal del bloque cada vez que hay datos suficientes en los *buffers* de entrada y espacio suficiente en los de salida. Si alguna de estas condiciones no se produce el hilo queda suspendido. Cada vez que un bloque termina de procesar un conjunto de datos (cuya longitud establece dinámicamente el planificador en función del tamaño de los *buffers* y otros factores) notifica este hecho a todos sus bloques vecinos, lo que se refleja en la figura 36 mediante las flechas horizontales de un hilo a otro.

Para una descripción más exhaustiva del planificador puede consultarse [106].

4.2 ADAPTACIÓN A SISTEMAS MULTIPROCESADOR HETEROGÉNEOS

4.2.1 *Consideraciones sobre el uso de GNU Radio en sistemas empotrados*

Hasta el momento se han descrito las características de GNU Radio tal como ha sido concebido para su uso en un ordenador personal. El ecosistema completo, incluyendo la blblioteca de bloques, la interfaz con Python, el GRC, etc., implica una serie de requisitos previos en cuanto a sistema operativo, memoria disponible y, especialmente, componentes *software* de soporte de los que no se ha hablado hasta el momento, pero que deben ser tenidos en cuenta a la hora de considerar el uso de GNU Radio en sistemas empotrados.

En principio puede parecer que los requisitos *software* de GNU Radio limitan su uso a sistemas de tipo PC. La lista de dependencias que aparece en el manual de GNU Radio es larga (Boost, Cppunit, FFTW, Python, SWIG, NumPy, GSL...), y muchas de ellas pueden considerarse poco adecuadas para sistemas empotrados. Sin embargo, en realidad muchos de estos requisitos son opcionales y se usan para alguna finalidad que, aunque útil, no es esencial, o se usan sólo para determinados bloques de procesado de señal. La parte realmente esencial de GNU Radio, que comprende el planificador, el sistema de gestión de *buffers* y algunos otros elementos, requieren

únicamente disponer de un compilador de C++ y de un sistema operativo que proporcione algunas partes concretas del API POSIX [107], especialmente la gestión de hilos mediante el API *pthreads*[3] y la sincronización mediante semáforos[4].

En una fase preliminar de este trabajo de tesis se exploró la posibilidad de portar GNU Radio a DSPs de la familia C6000 de Texas Instruments, tal como se describe en detalle en [108]. Se encontraron problemas de cierta relevancia debido a las herramientas de desarrollo (compilador y herramientas de depuración) disponibles en ese momento, de los cuales el más importante fue que el compilador de C++ de Texas Instruments no soportaba plenamente algunas características importantes del lenguaje. Debido a estas causas, esta vía de trabajo fue abandonada en favor de la alternativa que se describe más adelante. Muchos de estos problemas han sido resueltos posteriormente en las versiones más recientes de las herramientas, pero ya no se ha vuelto a retomar este trabajo.

Por otro lado, como se vio en el capítulo 2, sección 2.4, algunas arquitecturas para radio *software* están basadas en la combinación de procesadores de propósito general con otro tipo de procesadores especializados que funcionan como aceleradores. En estos casos no es extraño que el o los GPP utilicen Linux como sistema operativo, como ocurre en el procesador CELL descrito en la sección 2.5.3.2.

Un ejemplo claro de este tipo de configuraciones se encuentra en parte de la línea de productos de Texas Instruments, que contiene SoC que combinan procesadores de propósito general ARM con DSPs. En la sección 3.1 se describió el sistema de desarrollo SFF SDR DP, que contiene un SoC DaVinci DM6446 basado en un procesador ARM9 y un DSP C64x+, y en la familia OMAP también hay múltiples ejemplos de configuraciones similares.

En este contexto, y tras la experiencia obtenida con la implementación del receptor DVB-T/H en GNU Radio descrita en la sección 3.2, se formula la hipótesis de que GNU Radio pueda ser extendido para permitir su uso en arquitecturas de este tipo, de modo que la aplicación GNU Radio se ejecute en el procesador o procesadores de propósito general, y se puedan lanzar trabajos desde estos procesadores hacia los DSPs o aceleradores. Esta hipótesis se valida en esta tesis con la propuesta de modificación de la arquitectura *software* de GNU Radio.

3 El API *pthreads* estaba definido anteriormente en el estándar POSIX.1c. Aunque actualmente todas las partes de POSIX.1 se han fusionado en un único estándar, es habitual seguir utilizando las denominaciones antiguas.

4 Los semáforos formaban parte anteriormente de la extensión de tiempo real definida en POSIX.1b.

4.2.2 *Extensión de GNU Radio para el uso de un acelerador externo*

En este apartado se describe la extensión de GNU Radio para el escenario más simple, en el que se dispone de una arquitectura con un solo procesador de propósito general –sobre el que se ejecuta un sistema operativo con las características necesarias para permitir el portado de GNU Radio– y un solo DSP que actúa como acelerador. La suposición de que existe un único GPP es aplicable también sin pérdida de generalidad a las arquitecturas que cuentan con varios GPPs en configuración SMP, en cuyo caso se entenderá que al hablar del GPP se está hablando de todo el grupo de procesadores SMP.

Un objetivo de esta propuesta de extensión es que sea lo más general posible, facilitando al máximo su uso en diversas arquitecturas *hardware*. Para ello se propone el uso de una única primitiva de comunicación entre procesos, el paso de mensajes, que es de esperar que esté disponible en cualquier arquitectura multiprocesador.

Antes de describir la propuesta es conveniente definir el significado concreto con el que van a emplearse algunos términos a partir de este punto:

BLOQUE Bloque de procesado de señal GNU Radio que se ejecuta en el GPP.

FUNCIÓN Una implementación concreta de un algoritmo de procesado de señal, que puede ser o no parametrizable. Por ejemplo, pueden considerarse funciones la suma de dos señales, un filtro, el cálculo de la DFT o el estimador de comienzo de símbolo OFDM de la figura 28.

ACELERADOR Cualquier procesador en el que se implementan funciones que no sea el GPP en el que se ejecuta la aplicación GNU Radio.

TRABAJO Operación concreta de aplicación de una función a un determinado conjunto de datos que el GPP envía a un acelerador.

TRABAJADOR Hilo o proceso en un acelerador en el que se puede ejecutar una función, es decir, procesar un trabajo. Cada acelerador puede tener uno o varios trabajadores, pero su número es siempre fijo.

BLOQUE HUECO Bloque que en realidad no realiza el procesado de señal, sino que envía trabajos a un acelerador.

Supóngase en principio que se desea implementar un único bloque de procesado de señal de modo que el procesado no se realice en el GPP, sino en el DSP o acelerador. Teniendo en cuenta el funcionamiento de GNU Radio descrito en la sección 4.1, se ha propuesto un mecanismo sencillo y general para enviar trabajos a otro procesador

que consta de dos partes: una para la aplicación GNU Radio que se ejecuta en el GPP, y otra para el DSP o acelerador. La primera consiste en crear un bloque "hueco" que en cada invocación de su método *work*[5] realice los siguientes pasos:

1. Enviar un mensaje al acelerador para que procese los datos.
2. Esperar la recepción de un mensaje del acelerador que indique que el procesamiento ha concluido.
3. Devolver el resultado del mismo modo que cualquier otro bloque.

La segunda consiste en crear un programa para el acelerador que tenga el siguiente funcionamiento:

1. Esperar a recibir un mensaje de un GPP que indique que hay que procesar un conjunto de datos.
2. Realizar el procesado de los datos.
3. Enviar un mensaje al GPP indicando la finalización de la operación.
4. Volver al punto 1.

En este punto son pertinentes tres consideraciones acerca del mecanismo que se acaba de exponer. En primer lugar, el bloque de procesado de señal que se ejecuta en el GPP queda detenido durante el tiempo que dura el procesamiento en el acelerador. Esto no representa ningún problema siempre que se utilice el planificador TPB ya que los hilos de los demás bloques no se ven afectados y pueden ejecutarse normalmente. Como esta es la situación por defecto, se supondrá en adelante que esto es así.

En segundo lugar, no se está teniendo en cuenta cómo se transfieren los datos del GPP al acelerador y viceversa. El mecanismo de transferencia puede variar enormemente de una arquitectura a otra: memoria compartida, transferencia por DMA, conexión punto a punto, NoC... Sin embargo, sea cual sea, el procedimiento que se acaba de exponer sigue siendo esencialmente válido. La adaptación de este mecanismo genérico a arquitecturas concretas se tratará en el capítulo 5.

En tercer y último lugar, nótese que en el mecanismo propuesto GNU Radio se ejecuta únicamente en el GPP, y no se impone ningún requisito especial al acelerador más que ser capaz de recibir y enviar los mensajes correspondientes. Sin embargo, nada impide utilizar también GNU Radio para programar el acelerador, si es que esto

5 Aunque en realidad el método que se reimplementa al definir un bloque de procesado de señal no es siempre *work*, se supondrá que es así ya que simplifica la explicación y no se pierde generalidad.

es posible. Es decir, el mecanismo diseñado es utilizable con cualquier acelerador que vaya desde un procesador no convencional que sea necesario programar en lenguaje de ensamble o cualquier otro mecanismo similar de bajo nivel, hasta un procesador en el que se pueda ejecutar una aplicación GNU Radio.

4.2.3 *Generalización de la extensión de GNU Radio para un número arbitrario de funciones y aceleradores*

En la descripción anterior se ha supuesto que hay un único GPP, un único acelerador y un único bloque de procesado de señal que envía trabajos al acelerador. Sin embargo, en la situación más general habrá varios bloques acelerados, que pueden ser varias instancias del mismo tipo de bloque, varios bloques distintos o cualquier combinación de ambos casos, y varios aceleradores, que pueden ser iguales o no, y cada uno de ellos puede ser capaz de realizar un único algoritmo de procesado de señal o varios distintos. A continuación se propone una arquitectura de gestión de trabajos que da soporte a todas las posibles alternativas que se acaban de enumerar.

Con objeto de mantener la máxima generalidad posible, se supondrá que existe una cola de mensajes en el GPP y una cola en cada acelerador. Cualquier acelerador puede enviar un mensaje a la cola del GPP, y éste a su vez puede enviar mensajes a la cola de cualquier acelerador. Todas las colas son unidireccionales. Esta es la funcionalidad mínima que se puede requerir a un mecanismo de paso de mensajes; si en una arquitectura concreta no fuera posible que varios aceleradores pudieran enviar mensajes a una única cola en el GPP, se verá que la arquitectura propuesta es aplicable también a una situación en la que haya una cola en el GPP por cada acelerador.

Al intentar aplicar el mecanismo definido en el apartado 4.2.2 en la situación general (varias instancias de varios bloques huecos, varios aceleradores que implementan varias funciones) aparecen los siguientes requisitos:

1. Cuando un bloque hueco tiene un trabajo que enviar necesita un mecanismo que le permita seleccionar un acelerador que implemente la función adecuada y que tenga trabajadores libres.

2. Los bloques huecos quedan suspendidos cuando envían un trabajo a un acelerador. Es necesario un mecanismo que los reactive cuando el resultado del trabajo se recibe en la cola de mensajes del GPP.

Para resolver estas necesidades se define un nuevo elemento en GNU Radio, el *gestor de trabajos*, que realiza ambas funciones. El gestor de trabajos proporciona dos

recursos: una API para la gestión de trabajos (envío, espera de resultados y cualquier operación auxiliar necesaria), y un hilo de atención a los mensajes recibidos en la cola del GPP.

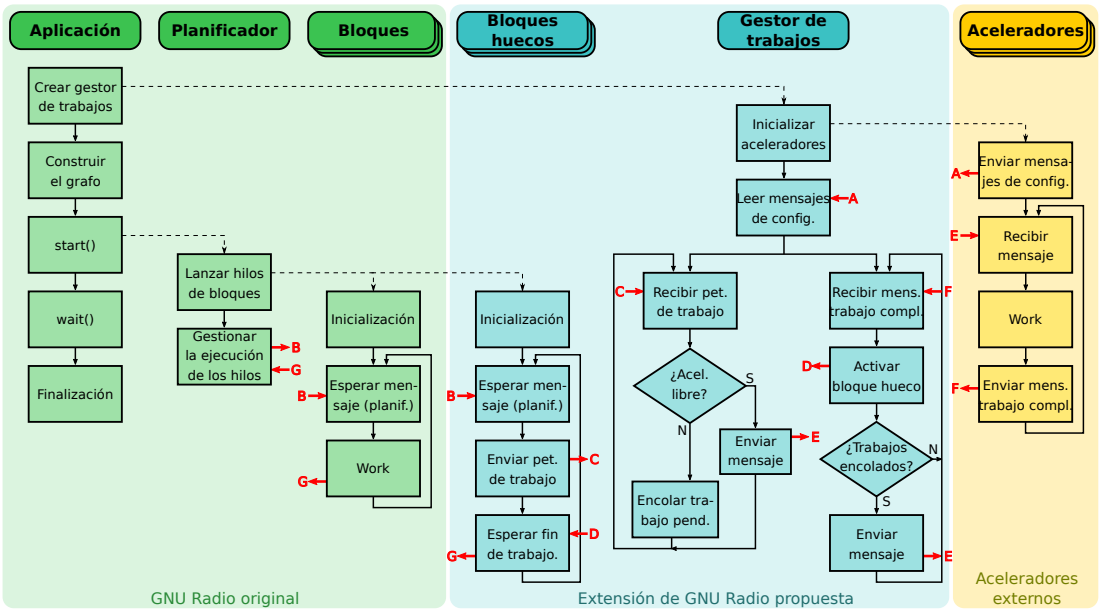


Figura 37: Diagrama de flujo del mecanismo propuesto para el uso de aceleradores en GNU Radio

El diagrama de flujo mostrado en la figura 37 describe la interacción entre los elementos que se han introducido (gestor de trabajos, bloques huecos y aceleradores) y entre éstos y los elementos de GNU Radio previamente existentes. El área verde contiene dichos elementos preexistentes, el área azul muestra los elementos que incorpora la extensión propuesta en este trabajo de tesis y por último el área amarilla contiene los aceleradores. Las líneas continuas muestran el flujo de ejecución, mientras que las líneas discontinuas indican la instanciación o el comienzo de la ejecución de algún elemento. Las flechas rojas representan los mensajes enviados o recibidos por los distintos elementos, y cada tipo de mensaje se identifica mediante una letra mayúscula de la A a la G.

El funcionamiento es el siguiente:

1. Al iniciar la aplicación se instancia el gestor de trabajos, que a su vez inicializa los aceleradores si es necesario. Cada acelerador envía un mensaje (tipo "A") al gestor de trabajos que contiene información sobre sus capacidades, en especial la lista de funciones que es capaz de procesar. El gestor de trabajos usa estos

mensajes para construir un mapa de aceleradores y funciones que más adelante servirá también para almacenar información operativa sobre el estado (por ejemplo, libre u ocupado) de los primeros.

2. Cuando el planificador de GNU Radio determina que un bloque hueco debe procesar los datos presentes en su *buffer* de entrada lo activa (mensaje tipo "B"), provocando la ejecución del método *work* del bloque. En lugar de enviar un mensaje directamente al acelerador como se describió en la sección 4.2.2, el bloque hueco usa la API del gestor de trabajos para enviarle una petición de trabajo[6] (mensaje "C"). A continuación el bloque hueco entra en estado suspendido hasta que reciba una notificación del gestor de trabajos indicando que el trabajo ha sido procesado (mensaje "D").

3. Cuando el gestor de trabajos recibe una petición de trabajo (mensaje "C"), busca un acelerador libre que pueda procesar la función requerida en el mapa que construyó en la fase de inicialización y le envía un mensaje (tipo "E"). Si no hay ningún acelerador libre que pueda procesar la petición, ésta se añade a una cola de peticiones pendientes.

4. Cuando un acelerador termina de procesar un trabajo, envía un mensaje (tipo "F") al gestor de trabajos. Éste busca qué bloque solicitó el trabajo y le notifica la finalización del mismo (mensaje "D"). Si la cola de trabajos pendientes no está vacía, el gestor de trabajos comprueba en este punto si el acelerador que ha quedado libre puede gestionar alguno de los trabajos pendientes, y si es así lo desencola y lo envía al acelerador.

5. Cuando un bloque hueco suspendido recibe la notificación de que el trabajo que solicitó ha sido completado (mensaje "D") ya puede notificar al planificador de GNU Radio que ha terminado de procesar los datos (mensaje "G").

El funcionamiento aquí descrito puede escalarse desde el caso más simple, con un único acelerador que procese una única función como en la sección 4.2.2, al caso más complejo que pueda imaginarse, en el que haya N aceleradores y M funciones y cada acelerador pueda procesar un subconjunto distinto de funciones. Es probable que este último caso represente un escenario artificialmente complejo que no llegue a darse nunca en la práctica, pero es importante resaltar que el mecanismo propuesto no impone ninguna limitación en cuanto a tamaño a la arquitectura que se desee implementar.

6 Pueden ser una o varias, según el caso, como se verá en el capítulo 5.

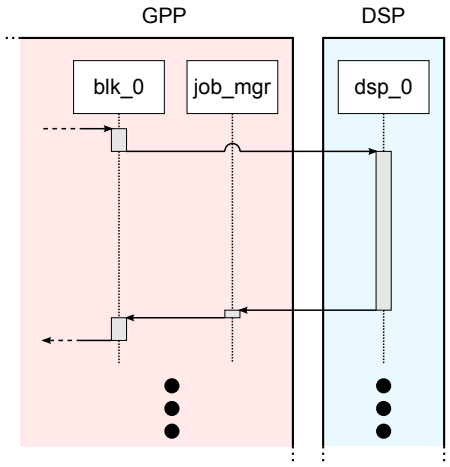


Figura 38: Diagrama de hilos de ejecución del mecanismo propuesto para el uso de aceleradores en GNU Radio

La figura 38 muestra, empleando la misma notación que en el ejemplo de la figura 36, el diagrama de hilos de ejecución del mecanismo de gestión de trabajos. El hilo etiquetado como "blk_0" corresponde a un bloque hueco, y el hilo "job_mgr" es el que el gestor de trabajos utiliza para atender los mensajes de los aceleradores. Ambos se ejecutan en el GPP, con fondo rojizo. El hilo "dsp_0" es un trabajador que se ejecuta en el acelerador, con fondo azulado.

Cuando el planificador de GNU Radio lanza la ejecución del método *work* del bloque "blk_0" (mensaje "B" en la figura 37), lo que se hace en éste es emplear la API del gestor de trabajos para preparar un trabajo y enviarlo al acelerador (mensajes "C" y "E" en la figura 37). El hilo de "blk_0" queda entonces suspendido esperando la recepción del resultado. En el acelerador, el trabajador "dsp_0" recibe el mensaje ("E"), ejecuta el trabajo y devuelve el resultado enviando un mensaje a la cola del GPP ("F" en la figura 37), tras lo cual vuelve a quedar suspendido esperando un nuevo trabajo. El hilo "job_mgr" se activa para recibir el mensaje, activa de nuevo al hilo "blk_0" (mensaje "D" en la figura 37) y vuelve a quedar suspendido. Por último, el hilo "blk_0" finaliza la ejecución de *work* y el planificador envía las notificaciones pertinentes a otros bloques del grafo, como ya se vio en el apartado 4.1.

4.2.4 *Descripción del gestor de trabajos*

En la sección 4.2.3 se ha introducido el gestor de trabajos y se han descrito someramente las tareas que realiza. En esta sección se describen en detalle estas tareas,

cuáles son los pasos de que consta cada una de ellas y qué estructuras de datos son necesarias para realizarlas. En ocasiones se presentan fragmentos de código C++ para describir algunas partes del gestor; dichos fragmentos deben entenderse como meramente ilustrativos y en ellos se ha prescindido de gran parte de la formalidad del lenguaje para evitar que lo que se quiere ilustrar quede oculto dentro de un código que si bien es técnicamente necesario no aporta nada a la descripción. En otros casos se recurrirá a una notación de tipo pseudo-código.

A partir del diagrama de flujo de la figura 37 se pueden deducir las tres funciones del gestor de trabajos, que son:

- Inicializar los aceleradores y construir las estructuras de datos necesarias.
- Facilitar a los bloques huecos el envío de trabajos (mensajes) a los aceleradores y la espera de los resultados.
- Recibir los mensajes de los aceleradores, reactivando los bloques huecos que han quedado bloqueados a la espera de resultados de los trabajos enviados.

La inicialización se realiza, como es lógico, una única vez, al principio de la ejecución de la aplicación GNU Radio. El envío de trabajos y espera de resultados se hace desde el hilo propio de cada bloque, a través de una sencilla API que ofrece el gestor de trabajos. La recepción de mensajes se hace, como ya se ha dicho antes, en un hilo específico del gestor de trabajos. A continuación se describen detalladamente cada una de estas tres funciones.

4.2.4.1 *Inicialización*

Los pasos de inicialización se resumen en el algoritmo 1. El primer paso es crear la cola de mensajes por la que recibirá los mensajes de los aceleradores, tanto los mensajes iniciales como los resultados de los trabajos. Tras este paso viene la inicialización de los aceleradores, que es totalmente dependiente de la plataforma y de los aceleradores que se empleen. En algunos casos no será necesaria ninguna acción por parte del GPP, mientras que en otros será necesario un proceso más o menos complejo que puede incluir, por ejemplo, inicialización (*reset*) *hardware*, carga de *software* en la memoria de los aceleradores, configuración de los mecanismos de comunicación y arranque de los aceleradores. En el capítulo 5 se describe este paso para dos plataformas concretas.

Una vez inicializados los aceleradores el gestor de trabajos debe construir una estructura de datos, que llamaremos *mapa de recursos*, que le permita realizar de la forma lo más eficiente posible tres operaciones: localizar un trabajador libre para

```
crear la cola de mensajes del GPP;
para cada acelerador hacer
    añadir a lista de aceleradores;
    crear y/o configurar la cola de mensajes del acelerador;
    inicializar el acelerador;
fin
mientras no leídos todos hacer
    leer mensaje de inicialización;
    para cada trabajador hacer
        añadir a lista de trabajadores;
        para cada función hacer
            si función nueva entonces
                añadir a lista de funciones;
            fin
            añadir función a lista de funciones del trabajador;
            añadir trabajador a lista de trabajadores para la función;
        fin
    fin
fin
lanzar el hilo de atención de mensajes de los aceleradores;
```

Algoritmo 1: Inicialización del gestor de trabajos

una determinada función cuando se solicite un trabajo, reactivar el bloque hueco adecuado cuando se reciba un resultado y, por último, encontrar en la lista de trabajos pendientes el trabajo más adecuado para un trabajador que haya quedado libre.

El gestor de trabajos obtiene la información necesaria para construir este mapa de recursos de los propios aceleradores, ya que tras su inicialización cada acelerador envía un mensaje inicial a la cola del GPP el que informa al gestor de trabajos del número de trabajadores que posee y de la lista de funciones que cada uno de ellos implementa.

Al recibir cada uno de estos mensajes, el gestor de trabajos va construyendo el mapa de recursos, que está formada por tres listas: de funciones, de aceleradores y de trabajadores. Estas listas están enlazadas de forma cruzada, de modo que desde la entrada de una función puede accederse a todos los trabajadores que la implementan, y desde un trabajador puede accederse a todas las funciones que implementa. Esto hace que la construcción del mapa sea algo compleja pero permite que las búsquedas de información en el mapa sean muy rápidas, como se verá más adelante.

La figura 39 muestra un ejemplo de mapa de recursos que contiene información de dos aceleradores y cuatro funciones. El acelerador "ac_1" tiene dos trabajadores: "tr_1", que implementa las funciones "fn_1" y "fn_2", y "tr_2", que implementa las funciones "fn_1" y "fn_3". El acelerador "ac_2" solo tiene un trabajador, "tr_3", que implementa las funciones "fn_2" y "fn_4".

Figura 39: Ejemplo de mapa de recursos del gestor de trabajos

La inicialización finaliza creando el hilo en el que se ejecuta el bucle de atención de mensajes de los aceleradores, que solo pueden ser ya resultados de trabajos.

4.2.4.2 *Envío de trabajos y espera de resultados: la API del gestor de trabajos*

El gestor de trabajos ofrece una API pública que permite, entre otras cosas, enviar un trabajo a los aceleradores y esperar la recepción del resultado de uno o varios trabajos. El listado 4 muestra los elementos fundamentales de esta API.

```
class job_manager
{
  [...]
  bool submit_job(job_descriptor_t* jd);
  bool wait_job(job_descriptor_t* jd);
  int wait_jobs(job_descriptor_t* jd[], wait_mode_t mode);
  [...]
};
```

Listado 4: API pública del gestor de trabajos

El elemento básico que maneja esta API es el trabajo, representado mediante un *descriptor de trabajo*. Estos descriptores están definidos en la clase *job_descriptor_t*, que almacena tanto los datos del trabajo (función y argumentos de entrada y salida) como la información referente a su gestión: estado (en ejecución, terminado...), trabajador asignado, etc. El descriptor de trabajo contiene también la información de sincronización necesaria para bloquear el hilo de un bloque hueco mientras se espera a la finalización de un trabajo y para reactivarlo cuando esto ocurre.

Para enviar un trabajo el bloque hueco construye el descriptor correspondiente y hace una llamada al método *submit_job* del gestor de trabajos. Este método busca en primer lugar un trabajador libre, recorriendo la lista de trabajadores que implementan la función indicada por el descriptor. Si se encuentra, envía el mensaje de solicitud de trabajo al acelerador correspondiente, actualiza el estado del trabajo y del trabajador; si no hay ningún trabajador libre, encola la solicitud de trabajo en una cola de trabajos pendientes. Por último, el método retorna el control al llamante.

Es importante el hecho de que en ningún caso se para la ejecución del hilo, tanto si se envía la solicitud de trabajo a un acelerador como si se encola en la cola de trabajos pendientes. Esto permite que el bloque hueco utilice cualquier estrategia de aceleración posible: puede limitarse a enviar un trabajo y esperar el resultado; puede dividir el procesamiento en varios trabajos, lanzarlos y esperar los resultados; y combinar cualquiera de las dos estrategias anteriores con parte de procesamiento en el GPP.

El bloqueo de la ejecución del bloque hueco se produce al esperar la finalizacion de uno o varios trabajos, para lo cual se emplean los métodos *wait_job* y *wait_jobs*, respectivamente. En realidad, *wait_job* proporciona únicamente una interfaz simplificada para *wait_jobs*, por lo que únicamente se describirá este último.

El método *wait_jobs* recibe como argumento la lista de descriptores de los trabajos que debe esperar y un parámetro (*mode*) que indica si la ejecución del hilo del bloque hueco debe reanudarse en cuanto termine alguno de los trabajos o únicamente cuando hayan terminado todos; esta posibilidad proporciona al implementador del bloque hueco una gran flexibilidad a la hora de elegir la estrategia de operación más adecuada en cada caso. La operación es sencilla, y consiste en un bucle que comienza recorriendo la lista de descriptores comprobando el estado de cada trabajo. La información de estado de cada descriptor incluye, además del estado del trabajo (como se verá más adelante, el hilo de recepción de mensajes marca el trabajo como terminado cuando se recibe un mensaje de fin de trabajo), una marca que indica si el trabajo ya estaba terminado en una iteración anterior del bucle.

Una vez comprobada toda la lista de descriptores se evalúa, en función del modo de espera elegido, si el método debe retornar. Esto es así cuando el modo de espera

es de algún trabajo y al menos uno ha cambiado de estado desde la última iteración del bucle, o cuando el modo de espera es de todos los trabajos y todos ellos han finalizado. En caso afirmativo el trabajo ha terminado y se retorna.

Si no es así, lo que puede ocurrir bien en la primera iteración del bucle cuando ningún trabajo ha finalizado, o bien en sucesivas iteraciones cuando el modo de espera es a todos los trabajos y aún queda alguno por finalizar, la ejecución del hilo del bloque hueco se suspende hasta que el hilo de recepción de mensajes de los aceleradores lo reactive. Cuando esto ocurre, es decir, cuando el hilo de recepción de mensajes de los aceleradores recibe un mensaje correspondiente a alguno de los trabajos por finalizar, se comienza una nueva iteración del bucle.

El mecanismo de bloqueo de la ejecución del hilo del bloque hueco y de su reactivación desde el hilo de atención de mensajes requiere dos cosas: que este último pueda identificar el descriptor (es decir, el trabajo) que corresponde a cada mensaje de terminación que recibe, y que pueda identificar el hilo que está bloqueado esperando a cada trabajo. La información que requiere la primera tarea depende de las características concretas de cada acelerador, aunque siempre puede resolverse asignando un identificador único a cada trabajo que se almacene en el descriptor y se incluya también en los mensajes de solicitud y de fin de trabajo. La identificación del hilo se resuelve, asumiendo que el gestor de trabajos se ejecuta en un sistema POSIX, incluyendo una variable de condición (*condition variable*) en el descriptor de trabajo; esta variable de condición permite al hilo de recepción de mensajes enviar una señal al bloque hueco cuyo hilo ha quedado bloqueado esperando precisamente a una señal sobre dicha variable. En el capítulo 5 se proporciona información concreta sobre cómo se ha resuelto esto en las implementaciones que se han realizado.

4.2.4.3 *Recepción de mensajes de los aceleradores*

Tal como se vió en la descripción de la inicialización del gestor de trabajos, el último paso de dicha fase es la creación de un hilo que se encarga de procesar los mensajes que los aceleradores envían a la cola de mensajes del GPP, que corresponden siempre al resultado de una petición de trabajo enviada a un acelerador. En esta sección se describe en detalle el funcionamiento de dicho hilo.

En el hilo de atención a los mensajes de los aceleradores se ejecuta un bucle infinito cuyo funcionamiento se resume en el algoritmo 2. Cuando se recibe un mensaje se identifican a partir de la información del mismo el trabajo (descriptor) al que corresponde y el trabajador que lo ha procesado, que se marcan como terminado y libre, respectivamente. A continuación se notifica al bloque hueco que el trabajo ha finalizado usando la variable de condición contenida en el descriptor de trabajo.

```
repetir
    leer mensaje;
    marcar trabajador libre y trabajo completado;
    notificar trabajo completado;
    mientras quedan trabajos pendientes hacer
        mirar siguiente trabajo pendiente;
        si funcion trabajo y trabajador coinciden entonces
            desencolar trabajo;
            enviar trabajo;
            break;
        fin
    fin
hasta que;
```

Algoritmo 2: Bucle de atención de mensajes de los aceleradores

A continuación se recorre la cola de trabajos pendientes hasta encontrar uno cuya función esté implementada por el trabajador que acaba de quedar libre. Si se encuentra, el trabajo se saca de la cola y se envía al acelerador tal como se describió en la sección anterior. Por último, el hilo vuelve a bloquearse hasta que se reciba el siguiente mensaje.

4.3 RESUMEN

En este capítulo se ha descrito en primer lugar el funcionamiento interno de GNU Radio, con especial énfasis en el planificador que gobierna la ejecución de los bloques de procesado de señal y el uso de hilos (*threads*) para explotar el paralelismo a nivel de tarea inherente a las aplicaciones de radio *software*.

A continuación se ha discutido la adecuación de GNU Radio para su uso en sistemas empotrados, llegando a la conclusión de que tanto sus partes esenciales (fundamentalmente, el planificador y el sistema de gestión de *buffers* de memoria) como los componentes básicos de la biblioteca de bloques de procesado de señal requieren únicamente de un compilador de C++ y un sistema operativo que soporte las componentes de gestión de hilos (*pthreads*) y de sincronización mediante semáforos del API POSIX. Esto permite, por un lado, pensar que GNU Radio puede emplearse con relativamente poco esfuerzo en, al menos, los procesadores de propósito general que suelen formar parte de las arquitecturas multiprocesador heterogéneas empleadas

con frecuencia en sistemas de radio *software*; y, por otro lado, formular la hipótesis de que es posible extender GNU Radio para emplear los procesadores especializados y/o los aceleradores *hardware* que dichas arquitecturas puedan contener.

Por último, se ha propuesto una extensión de la arquitectura *software* de GNU Radio para lograr dicho objetivo. La extensión propuesta está basada en la idea de *bloques huecos* que de cara a GNU Radio se comporten como cualquier otro bloque de procesado de señal de su biblioteca, y cuya función es enviar solicitudes de procesamiento (*trabajos*) a los coprocesadores o aceleradores disponibles en la plataforma. Se introduce además un nuevo elemento, el *gestor de trabajos*, que gestiona las peticiones de trabajos, las comunicaciones con los aceleradores, la recepción de resultados de los trabajos, y la suspensión y reactivación de los bloques huecos cuando estos quedan a la espera de la recepción de dichos resultados. Esta extensión de GNU Radio es una de las contribuciones principales de este trabajo de tesis (véase la sección 7.1.2).

5 IMPLEMENTACIONES

En este capítulo se describen las implementaciones de la extensión de GNU Radio descrita en el capítulo 4 en dos plataformas que combinan núcleos de propósito general ARM y DSPs de la familia C6000, de la que se habló brevemente en el capítulo 2. La sección 5.1 describe la primera implementación, realizada en una plataforma basada en un procesador OMAP 3530 de Texas Instruments, que ha permitido realizar una validación inicial de la extensión en un sistema relativamente sencillo con solo un núcleo GPP y un núcleo DSP empleado como acelerador. La sección 5.2 describe una segunda implementación en un procesador mucho más potente de la familia KeyStone II, también de Texas Instruments, con cuatro núcleos GPP y ocho núcleos DSP. Es en esta segunda plataforma en la que se ha empleado un receptor DVB–T como caso de uso para validar la utilidad de la metodología propuesta.

5.1 IMPLEMENTACIÓN EN PROCESADOR OMAP 3530

A la hora de seleccionar una plataforma en la que implementar la extensión de GNU Radio descrita en el capítulo 4, el objetivo inicial era encontrar una que combinase un procesador de propósito general capaz de ejecutar Linux y algún tipo de DSP que pudiera utilizarse como acelerador y que además pudiera conectarse con un frontal de RF que permitiera realizar un prototipo funcional completo de una aplicación de radio *software*. El sistema SFF SDR DP del que se habló en el capítulo 3 era un candidato interesante ya que su sección de procesado en banda base está basada en un SoC DM6446 de la familia DaVinci de Texas Instruments, que incluye un procesador ARM9 como GPP y un DSP C64x+ que podría emplearse como acelerador. Sin embargo, aunque ciertamente es posible utilizar el sistema operativo Linux en el núcleo ARM del DM6446 [109], el fabricante de la plataforma SFF SDR DP no ofrecía soporte para ello y eso hacía que fuera una elección muy arriesgada.

No obstante, existía otra familia de SoC de Texas instruments muy similar a la DaVinci en cuanto a arquitectura y programación, la OMAP (*Open Multimedia Applications Platform*). El primer OMAP (el OMAP 1510, introducido en el año 2001) era un procesador de aplicaciones para dispositivos móviles cuya principal característica era que incorporaba un procesador ARM9 de propósito general y un núcleo DSP C55x, una variante de bajo consumo de la familia C6000. Desde entonces han aparecido de-

cenas de procesadores OMAP con distintas configuraciones, muchas de las cuales se han distribuido únicamente a fabricantes de dispositivos móviles (teléfonos, tablets, etc.). Algunas de ellas, sin embargo, se han comercializado a través de canales abiertos y se han utilizado en ocasiones para construir sistemas de desarrollo económicos. Uno de los ejemplos más notables es la *Beagleboard*, un sistema de desarrollo basado en el procesador OMAP 3530 cuyo diseño *hardware* está disponible con licencia Creative Commons y que en el momento de su aparición en el año 2008 se podía adquirir por unos 150 euros.

Por todo ello se decidió realizar el trabajo de implementación en una tarjeta Beagleboard o similar, ya que el uso de una plataforma más sencilla y con soporte para Linux haría más facil el trabajo y el resultado sería trasladable con pocas modificaciones a otros SoC relacionados. La tarjeta de desarrollo finalmente seleccionada fue la *IGEP v2* [110], que puede verse en la figura 40. Esta tarjeta, fabricada por la empresa barcelonesa ISEE, es muy similar a la Beagleboard mencionada anteriormente y utiliza el mismo SoC OMAP 3530. La principal diferencia es que tiene más memoria RAM (512 MB en lugar de 256), lo que facilita el trabajo de desarrollo con ella, pero a efectos del trabajo que se describe aquí ambas tarjetas son totalmente equivalentes.

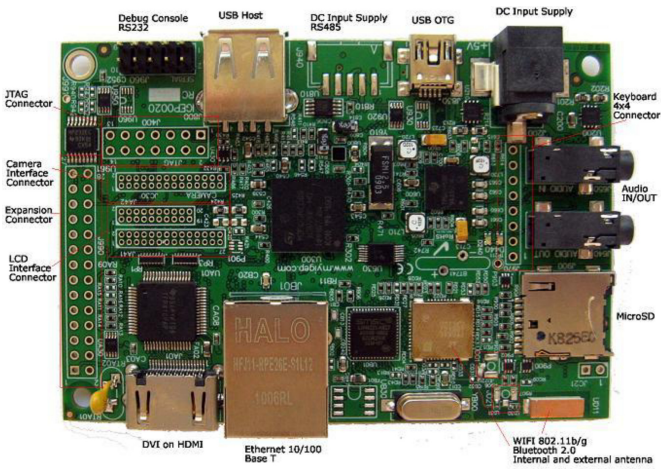


Figura 40: Tarjeta IGEP v2 [110]

La figura 41 muestra el diagrama de bloques funcional del OMAP 3530. En la esquina superior izquierda aparecen los dos elementos más relevantes para el trabajo que aquí se describe, los bloques *MPU Subsystem*, que contiene un procesador de propósito general ARM Cortex A8 a 720 MHz, e *IVA 2.2 Subsystem*, que contiene un DSP C64x+ a 520 MHz[1] que se emplea como acelerador.

1 Ambas frecuencias de reloj son valores máximos.

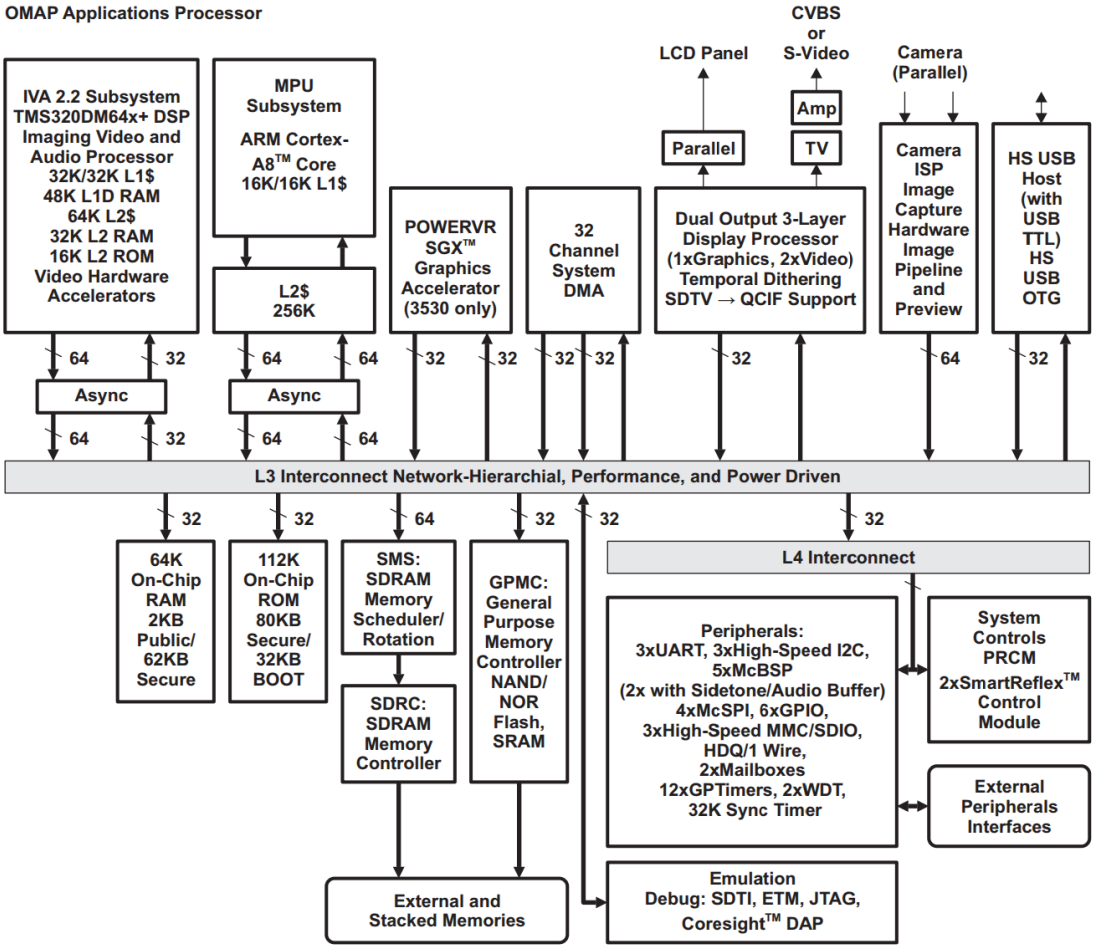


Figura 41: Diagrama de bloques del SoC OMAP 3530 [111]

La distribución de Linux empleada en la tarjeta IGEP v2 es Ångström [112], una distribución orientada a sistemas empotrados basada en OpenEmbedded [113], un sistema para automatizar la compilación cruzada de paquetes de *software*. El portado de GNU Radio a esta plataforma puede resultar laborioso debido a la gestión de los prerrequisitos de GNU Radio, por un lado porque en la distribución Ångström los paquetes[2] están diseñados con una granularidad mucho mayor que en las distribu-

2 Un *paquete* (*package*, en inglés) es un fichero que permite instalar una aplicación, librería, componente del sistema operativo o, en general, una colección de ficheros en un sistema. El paquete contiene, además de los ficheros necesarios, información de las posibles interdependencias entre unos paquetes y otros, de modo que se puedan instalar o desinstalar aplicaciones, librerías, etc., garantizando que el conjunto

ciones habituales y eso puede obligar a comprobar e instalar manualmente decenas de paquetes, y por otro porque algunos de los prerrequisitos pueden no estar disponibles en forma de paquete para la arquitectura concreta que se esté utilizando y haya que instalarlos manualmente. Aún así es un proceso que no reviste especial dificultad por lo que no se va a describir aquí. El lector interesado puede consultar los detalles en [114].

El DSP, por su parte, utiliza un sistema operativo de tiempo real (RTOS, *Real-Time Operating System*) propio de Texas Instruments llamado *DSP/BIOS* [94]. Este sistema operativo tiene una organización modular, como es habitual en los RTOS para sistemas empotrados, e incluye entre los servicios que proporciona mecanismos de comunicación entre procesos (que en el argot empleado en DSP/BIOS reciben el nombre de *tareas*). Uno de esos módulos, denominado *MSGQ*, implementa colas de mensajes. La comunicación entre ambos procesadores, ARM y DSP, es posible mediante un *software* proporcionado por Texas Instruments con el nombre *DSPLink*, que se compone de dos partes:

- Una parte DSP/BIOS que incluye, entre otras cosas, un módulo que extiende los mecanismos de comunicación entre procesos ya existentes para que, manteniendo la misma API, operen entre procesadores distintos, incluyendo procesadores de distintos tipos como GPP y DSP.
- Una parte Linux compuesta a su vez por:
  - Un *driver* que permite al procesador ARM controlar el DSP (arrancarlo, pararlo, etc.) y comunicarse con él.
  - Una librería que, empleando las facilidades provistas por el *driver*, ofrece varias APIs (llamadas *componentes*) de comunicaciones entre procesadores ARM y DSP.

En resumen, la combinación del módulo MSGQ de DSP/BIOS y el componente MSGQ de DSPLink permite enviar y recibir mensajes entre ambos tipos de procesadores. La coincidencia de nombres entre módulo y componente no es casual, ya que el componente replica en el lado ARM la API del módulo DSP/BIOS.

El intercambio de datos entre GPP y DSP, sin embargo, no se realiza únicamente mediante los mensajes, ya que el tamaño de estos es limitado. Como puede verse en la figura 41, tanto el ARM como el DSP pueden acceder a zonas de memoria comunes,

de paquetes instalados en el sistema permanece siempre coherente. Existen varios *sistemas de gestión de paquetes*, cada uno de los cuales define el formato y la información que contienen los paquetes y proporciona herramientas (aplicaciones) para la gestión (creación, instalación, eliminación, etc.) de los mismos. La distribución Ångström emplea el gestor de paquetes *Opkg*.

bien sea la memoria interna de 64 kB incluida en el OMAP 3530 (bloque *On-Chip RAM*) o bien la memoria principal que se conecta al controlador de memoria SDRAM (bloque *SDRC: SDRAM Memory Controller*), de modo que el intercambio de datos se realiza mediante *buffers* de memoria compartida y los mensajes que se intercambian entre procesadores contienen las direcciones de inicio de los *buffers* y sus tamaños.

El uso de memoria compartida en esta plataforma requiere resolver un problema: Linux es un sistema operativo que emplea memoria virtual, por lo que las direcciones de memoria que manejan los procesos no corresponden a direcciones físicas. Además, debido a la paginación, lo que para un proceso aparenta ser un *buffer* único estará repartido en general en varias páginas de memoria física no contiguas. DSP/BIOS, por el contrario, no emplea memoria virtual. La solución al problema es emplear un *driver* para Linux denominado *CMEM*, proporcionado por Texas Instruments. La misión de CMEM es proporcionar acceso a las aplicaciones Linux a regiones de memoria físicamente contigua[3], junto con un mecanismo de traducción para obtener las direcciones físicas de esas regiones de memoria.

La memoria gestionada por CMEM no puede estar gestionada simultáneamente por el gestor de memoria virtual de Linux, por lo que para emplear parte de la memoria externa (512 MB, en el caso de la tarjeta IGEP v2) para el intercambio de datos entre ARM y DSP es necesario indicar a Linux en el arranque que la cantidad de memoria que tiene disponible es menor que la total. Las pruebas que se describen en la sección 5.1.2 se han realizado asignando 480 MB de memoria externa a Linux, lo que deja 32 MB para ser gestionados por CMEM.

5.1.1 *Ejemplo de implementación de un bloque*

A continuación se presenta un ejemplo de implementación en el DSP de un bloque que aplica un filtro de media móvil. Se ha elegido esta función porque es sencilla, porque la biblioteca de bloques de GNU Radio incluye un bloque llamado *gr::blocks::moving_average* que realiza esta misma función, y porque en el DSP se puede implementar mediante la función *DSP_fir_gen* de la biblioteca *DSPLIB* que proporciona Texas Instruments, ilustrando así la posibilidad de emplear los recursos proporcionados por el fabricante de la plataforma cuando estos existan.

El ejemplo se presenta mostrando varios fragmentos de código (listados 5 a 8) correspondientes a distintas partes de la implementación del bloque. Debe tenerse en cuenta que estos fragmentos están muy simplificados; se han eliminado elementos del código tales como comprobación de errores, fragmentos de inicialización o determinados ajustes en las longitudes de los *buffers* que, si bien son necesarios para el

3 De ahí el nombre del *driver* CMEM: *Contiguous MEMory*.

```
1  #define MAX_ARGS_DIRECT  8
2
3  typedef enum {
4    GCT_S32,
5    GCT_U32,
6    GCT_PTR     = 0x80000000 // If set, pointer to...
7  } gd_tag_t;
8
9  typedef union gd_arg_union
10 {
11   int32_t   s32;
12   uint32_t  u32;
13   void *    ptr;
14 } gd_arg_union_t;
15
16 typedef struct gd_job_direct_args
17 {
18   uint32_t       nargs;
19   gd_tag_t       tag[MAX_ARGS_DIRECT];
20   gd_arg_union_t arg[MAX_ARGS_DIRECT];
21 } gd_job_direct_args_t;
22
23 typedef struct gd_job_desc
24 {
25   gd_job_status_t       status;
26   gd_proc_id_t          proc_id;
27   gd_job_direct_args_t  input;
28   gd_job_direct_args_t  output;
29 } gd_job_desc_t;
```

Listado 5: Definición simplificada del descriptor de trabajo

correcto funcionamiento del bloque, alargan notablemente el código y dificultan la tarea de exposición con detalles que en este punto no son relevantes.

El listado 5 muestra la definición del descriptor de trabajo haciendo especial hincapié en el sistema de paso de parámetros de entrada y salida entre ARM y DSP, que se realiza mediante sendos *arrays* de longitud fija cuyos elementos son uniones (tal como se definen en C y C++) que permiten alojar datos de cualquier tipo. Un *array* paralelo de etiquetas (la estructura *gd_tag_t*) permite identificar el tipo del dato en cada posición. Los mensajes enviados entre ARM y DSP en ambos sentidos consisten fundamentalmente en este descriptor de trabajo, junto con alguna información adicional requerida por el API MSGQ.

El listado 6 muestra la definición del bloque hueco *gdsp_moving_avg_s* (que, como puede verse, es una clase derivada de la clase base *gr_sync_block* de GNU Radio) junto con su constructor y su destructor. En la definición (líneas 1 a 21) pueden verse los siguientes elementos:

- Constantes como el tamaño de los *buffers* de memoria (línea 3) y el nombre identificativo del tipo de bloque (línea 4). Las líneas 23 y 24 contienen la instanciación de estas constantes.
- La referencia (puntero) al gestor de trabajos (línea 6). En la descripción del constructor se verá cómo se obtiene dicha referencia.
- Una variable para almacenar la longitud de la media (línea 8), que se debe pasar como parámetro en la instanciación del bloque.
- Los punteros a los *buffers* que se van a emplear, tanto virtuales (línea 10) como físicos (línea 11).
- La sobrecarga del método *work*, como se describió en la sección 4.1.

El constructor (líneas 26 a 47) realiza las siguientes tareas:

- Obtiene la referencia al gestor de trabajos (línea 32). El gestor de trabajos es un objeto único en la aplicación, y para asegurar esta unicidad y facilitar la obtención de referencias al objeto cuando es necesario se emplea un patrón de diseño *software* denominado *Singleton*[4]. Para la implementación de este patrón de diseño se ha recurrido a la librería *Boost* [116].
- Obtiene un identificador numérico de la función a partir de su nombre. Este identificador numérico se emplea internamente para localizar trabajadores, tal como se describió en la sección 4.2.4.
- Reserva memoria para los *buffers* que se emplearán (líneas 35 a 37) y obtiene sus direcciones físicas de inicio (líneas 39 a 41), todo ello mediante el API CMEM.
- Por último, calcula los coeficientes del filtro FIR que se utiliza en el DSP para implementar la media móvil (líneas 43 a 47).

El destructor (líneas 49 a 54) tiene como única misión liberar los *buffers* de memoria cuando el bloque es destruido.

4 Un patrón de diseño *software* (del inglés *design pattern*) es una solución general y reutilizable a un problema de diseño que aparece con frecuencia. Por ejemplo, *Singleton* es un patrón creacional que garantiza la existencia de una única instancia de una clase y proporciona un mecanismo de acceso global a dicha instancia. El lector interesado puede obtener más información en [115].

```
1  class gdsp_moving_avg_s : public gr_sync_block
2  {
3    static const int CMEM_BUF_LEN;
4    static const char d_proc_name[];
5
6    boost::shared_ptr<job_manager> d_mgr;
7
8    size_t d_vlen;
9
10   short *virt_ptr1, *virt_ptr2, *virt_ptr3;
11   unsigned int phys_ptr1, phys_ptr2, phys_ptr3;
12
13   gdsp_moving_avg_s (size_t vlen);
14
15 public:
16   ~gdsp_moving_avg_s();
17
18   int work (int noutput_items,
19             gr_vector_const_void_star &input_items,
20             gr_vector_void_star &output_items);
21 };
22
23 const int gdsp_moving_avg_s::CMEM_BUF_LEN = 40960;
24 const char gdsp_moving_avg_s::d_proc_name[] = "moving_avg";
25
26 gdsp_moving_avg_s::gdsp_moving_avg_s(size_t vlen)
27   : gr_sync_block ("gdsp_moving_avg_s",
28                    gr_make_io_signature (1, 1, sizeof(short)),
29                    gr_make_io_signature (1, 1, sizeof(short))),
30     d_vlen (vlen)
31 {
32   d_mgr = job_manager::singleton();
33   d_fn_id = d_mgr->lookup_proc(d_proc_name);
34
35   virt_ptr1 = (short *) CMEM_alloc(CMEM_BUF_LEN, 0);
36   virt_ptr2 = (short *) CMEM_alloc(CMEM_BUF_LEN, 0);
37   virt_ptr3 = (short *) CMEM_alloc(CMEM_BUF_LEN, 0);
38
39   phys_ptr1 = CMEM_getPhys((void *)virt_ptr1);
40   phys_ptr2 = CMEM_getPhys((void *)virt_ptr2);
41   phys_ptr3 = CMEM_getPhys((void *)virt_ptr3);
42
43   short coeff = (short) (floor(1.0 / d_vlen) * 65536);
44   for (unsigned int i = 0; i < d_vlen; ++i) {
45     virt_ptr2[i] = coeff;
46   }
47 }
48
49 gdsp_moving_avg_s::~gdsp_moving_avg_s()
50 {
51   CMEM_free((void *) virt_ptr1, 0);
52   CMEM_free((void *) virt_ptr2, 0);
53   CMEM_free((void *) virt_ptr3, 0);
54 }
```

Listado 6: Definición, constructor y destructor del bloque *gdsp_moving_avg_s*

El listado 7 muestra la definición del método *work*, que es el invocado por el planificador de GNU Radio cuando el bloque debe procesar un lote de datos. El método recibe como parámetros de entrada el número de elementos de salida que debe generar y dos *arrays* de punteros a los *buffers* de entrada y salida, respectivamente. El bloque de media móvil solo tiene una entrada y una salida, por lo que en este caso ambos *arrays* tienen longitud uno. El funcionamiento de este método es el siguiente:

- En primer lugar se almacenan los punteros a los *buffers* de entrada y salida en variables locales del tipo adecuado en cada caso (líneas 5 y 6).
- Se calcula el número de datos que caben en los *buffers* de comunicaciones con el DSP (línea 8).
- Se calcula cuántos trabajos habrá que enviar al DSP (línea 9) y cuántos datos quedan pendientes para un último trabajo (línea 10).
- Se construye el descriptor de trabajo que se va a emplear para todos los trabajos (líneas 12 a 20). La función tiene cuatro parámetros de entrada (número de datos en el *buffer* de entrada, número de coeficientes del filtro y direcciones de los *buffers* de coeficientes y de datos de entrada) y uno de salida (la dirección del *buffer* de salida). Los coeficientes del filtro son constantes y se calcularon en el constructor.
- Bucle principal (líneas 22 a 29). En cada iteración se copian un lote de datos de entrada en el *buffer* de entrada al DSP, se envía el trabajo, se espera a su finalización y se copian los datos del *buffer* de salida del DSP al *buffer* de salida de GNU Radio.
- Se envía un último trabajo con los datos de entrada restantes (líneas 31 a 38).
- Se libera el descriptor de trabajos (línea 40) y se informa al planificador de GNU Radio de cuántos datos de salida se han generado (línea 41).

Por último, el listado 8 muestra el código que implementa la función en el DSP. No se muestra el bucle de recepción de mensajes, etc., descrito en la sección 4.2.4, que es común y compartido para todas las funciones que se implementen en el DSP. Como puede verse, lo único que hace el código es extraer los parámetros de entrada del mensaje recibido y llamar a la función *DSP_gen_fir*, descrita anteriormente.

```
1  int gdsp_moving_avg_s::work(int noutput_items,
2                              gr_vector_const_void_star &input_items,
3                              gr_vector_void_star &output_items)
4  {
5    const short *data_in = (const short *) input_items[0];
6    short *data_out = (short *) output_items[0];
7
8    int n_buffer_samples = (CMEM_BUF_LEN / sizeof(short));
9    int n_jobs =  noutput_items / n_buffer_samples;
10   int n_remaining_samples = noutput_items %n_buffer_samples;
11
12   job_descriptor_t *jd = d_mgr->alloc_job_desc();
13   jd->proc_id = d_fn_id;
14   jd->input.nargs = 4;
15   jd->output.nargs = 1;
16   jd->input.arg[0].s32 = n_buffer_samples;
17   jd->input.arg[1].s32 = d_vlen;
18   jd->input.arg[2].u32 = phys_ptr2;  // Coeff. buffer
19   jd->input.arg[3].u32 = phys_ptr1;  // Input buffer
20   jd->output.arg[0].u32 = phys_ptr3; // Output buffer
21
22   for(int i = 0; i < n_jobs; ++i) {
23     memcpy(virt_ptr1, &data_in[i * n_buffer_samples],
24            n_buffer_samples * sizeof(short));
25     d_mgr->submit_job(jd);
26     d_mgr->wait_job(jd);
27     memcpy(&data_out[i * n_buffer_samples], virt_ptr3,
28            n_buffer_samples * sizeof(short));
29   }
30
31   if (n_remaining_samples > 0) {
32     memcpy(virt_ptr1, &data_in[n_jobs * n_remaining_samples],
33            n_remaining_samples * sizeof(short));
34     d_mgr->submit_job(jd);
35     d_mgr->wait_job(jd);
36     memcpy(&data_out[n_jobs * n_remaining_samples], virt_ptr3,
37            n_remaining_samples * sizeof(short));
38   }
39
40   d_mgr->free_job_desc(jd);
41   return noutput_items;
42 }
```

Listado 7: Método *work* del bloque *gdsp_moving_avg_s*

```
1  void fn_moving_avg(job_descriptor_t *job)
2  {
3    int nh, len, nr;
4
5    len = job->input.arg[0].s32;
6    nh = job->input.arg[1].s32;
7    nr = len - nh + 1;
8    DSP_fir_gen( (short *) job->input.arg[3].u32,
9                 (short *) job->input.arg[2].u32,
10                (short *) job->output.arg[0].u32,
11                nh, nr);
12 }
```

Listado 8: Código DSP para el bloque *gdsp_moving_avg_s*

5.1.2 *Resultados obtenidos*

El correcto funcionamiento de esta implementación se ha verificado con varias versiones de una aplicación sencilla, que únicamente genera una señal y le aplica el filtro de media móvil que se ha descrito en la sección anterior. En la figura 42 pueden verse las cinco versiones de la aplicación que se han empleado:

(a) Versión GPP. El filtrado se realiza en el ARM.

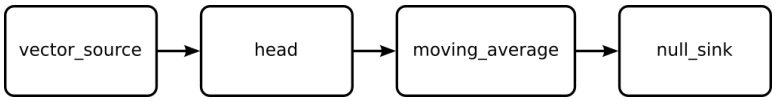
(b) Versión GPP doble. El filtrado se realiza en el ARM, pero se generan y filtran dos señales en vez de una.

(c) Versión DSP. El filtrado se realiza en el DSP.

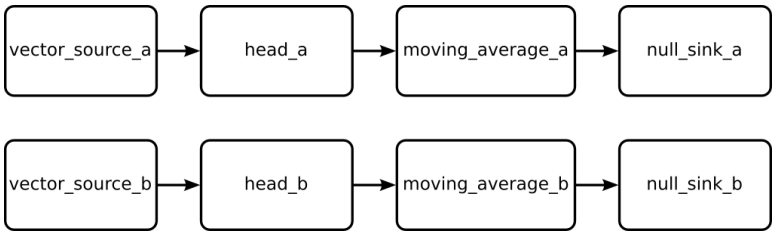
(d) Versión GPP+DSP. Se genera una única señal pero se duplica el filtrado, que se realiza tanto en el ARM como en el DSP.

(e) Versión GPP+DSP separada. Se generan dos señales independientes; una se filtra en el ARM y la otra en el DSP.

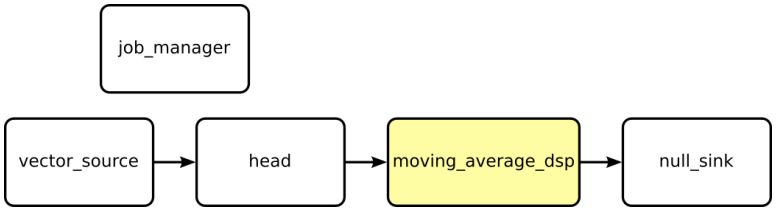
Los bloques que se ejecutan en el DSP (*moving_average_dsp*) se han resaltado con fondo amarillo.



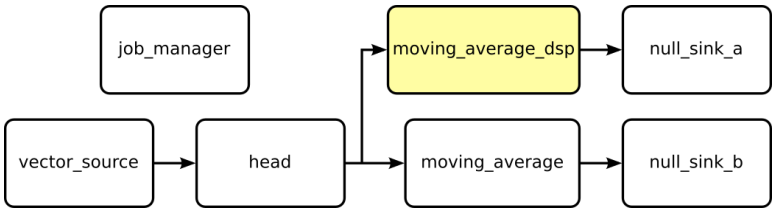
(a) Versión GPP



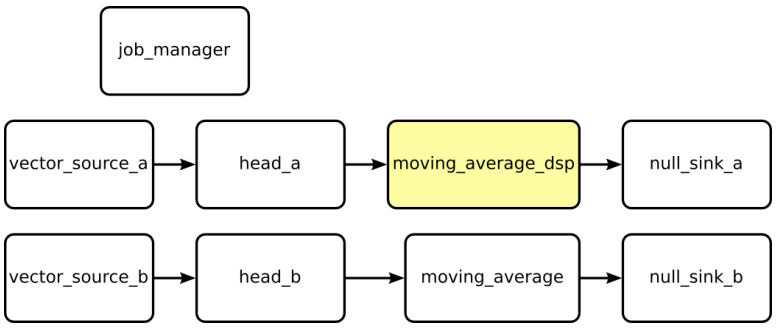
(b) Versión GPP doble



(c) Versión DSP



(d) Versión GPP+DSP



(e) Versión GPP+DSP separada

Figura 42: Grafos de las aplicaciones empleadas en la plataforma OMAP 3530 [114]

El comportamiento de las distintas versiones puede deducirse de la figura 43, que muestra el tiempo de ejecución de cada una de ellas.

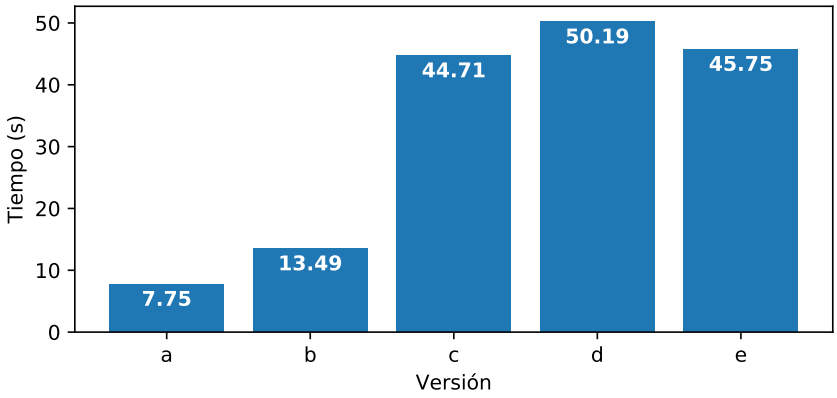


Figura 43: Tiempo de ejecución en OMAP 3530 de los grafos mostrados en la figura 42

La versión b tarda, como era de esperar, aproximadamente el doble que la versión a. La versión c muestra que el DSP es considerablemente más lento que el ARM, lo que no es demasiado sorprendente si se piensa que su velocidad de reloj es notablemente inferior (de hecho, en esta prueba es la mitad).

El tiempo de la versión d puede ser desconcertante a primera vista, ya que es prácticamente la suma de los tiempos de las versiones a y c, lo cual parece indicar que el ARM y el DSP no trabajan simultáneamente. Sin embargo, lo que ocurre es un efecto del funcionamiento del planificador debido a que los dos bloques de media móvil comparten la señal de entrada. Esto queda claro en la versión e, cuyo tiempo de ejecución es prácticamente el mismo que el de la versión c; en este caso ambos bloques de filtrado sí se ejecutan simultáneamente en los dos procesadores y el tiempo total corresponde al del bloque más lento.

El objetivo que se perseguía con esta implementación no era una mejora de rendimiento en una aplicación concreta, sino una confirmación de que la hipótesis de partida era correcta y de que esta metodología de trabajo podía dar buenos resultados. Es por ello que los resultados obtenidos, aunque muestran que el DSP es significativamente más lento que el ARM en esta plataforma, pueden considerarse satisfactorios. Es seguro que se podría haber obtenido un mejor rendimiento del DSP ya que no se ha hecho ningún esfuerzo de optimización, especialmente en la transferencia de datos entre ARM y DSP, pero ello no hubiese aportado ningún resultado adicional para este trabajo de tesis.

5.2 IMPLEMENTACIÓN EN PROCESADOR KEYSTONE II

Mientras aún se estaba trabajando en la implementación en el procesador OMAP 3530 descrita en la sección anterior, Texas Instruments presentó la familia de SoC KeyStone II. Al igual que su predecesora KeyStone, esta familia está construida de forma modular alrededor de un sofisticado sistema de comunicaciones interno de alto rendimiento denominado *TeraNet* que conecta un número variable de procesadores y periféricos. La familia KeyStone ya incluía SoCs con entre 1 y 8 núcleos DSP C66x; la KeyStone II introdujo SoCs que combinan núcleos C66x con procesadores ARM. Otros productos anteriores del fabricante ya incluían estos dos tipos de procesador: algunos miembros de las familias DaVinci, como el DM6446 mencionado en las secciones 3.1, 4.2 y 5.1, y OMAP, como el propio OMAP 3530; sin embargo, en todos ellos había únicamente un núcleo de cada tipo. La familia KeyStone II, por el contrario, tiene SoCs que incorporan hasta 8 núcleos C66x y 4 núcleos ARM. Al ser además una familia orientada a sistemas de alto rendimiento, es obvio que las posibilidades que ofrece son mucho mayores que las del OMAP 3530 empleado en la primera implementación.

La familia KeyStone II (Texas Instruments emplea el término *arquitectura*) se divide a su vez en *plataformas*, que difieren entre sí fundamentalmente en el tipo (ARM o C66x) y número de núcleos que pueden incluir y en la existencia de coprocesadores *hardware* especializados en determinadas tareas; actualmente existen las plataformas K2H, K2K, K2E, K2L y K2G. El SoC seleccionado para la implementación de la extensión propuesta de GNU Radio en la arquitectura KeyStone II fue el 66AK2H14 [117]. Es el SoC más potente de la plataforma K2H, que tiende a favorecer los recursos de cómputo general, con 4 núcleos ARM Cortex A15 a 1,4 GHz y 8 núcleos C66x a 1,2 GHz, además de una cantidad considerable de periféricos tanto internos (como por ejemplo controladores de DMA) como para subsistemas externos (controladores de memoria externa DDR3 y Flash, controladores de comunicaciones Ethernet, USB, PCIe, I²C, etc.). Hubiera sido aún más apropiado el SoC TCI6638K2K, de la plataforma K2K, que es muy similar al 66AK2H14 pero incluye además algunos coprocesadores *hardware* específicos para funciones de comunicaciones como FFT o decodificadores Viterbi y Turbo; lamentablemente, la distribución de este SoC ha sido limitada y en el momento de la adquisición de la plataforma de desarrollo no estaba disponible.

La arquitectura del SoC 66AK2H14, cuyo diagrama de bloques se muestra en la figura 44, guarda cierta similitud estructural con la del OMAP 3530. Los núcleos se integran en una estructura que Texas Instruments denomina *CorePac*, que incluye además las memorias *cache* de nivel 1 y 2 y algunos periféricos como temporizadores,

controladores de interrupciones o módulos para depuración. Todos los núcleos ARM se integran en un único *CorePac* [118] y comparten la memoria *cache* de nivel 2 (4 MB), mientras que cada núcleo C66x tiene su propio *CorePac* [119]. Todos los *CorePacs* tienen acceso a los controladores de memoria externa y a un bloque de memoria interna compartida (*Multicore Shared Memory Controller*, MSMC [120]) que en esta plataforma tiene un tamaño de 6 MB.

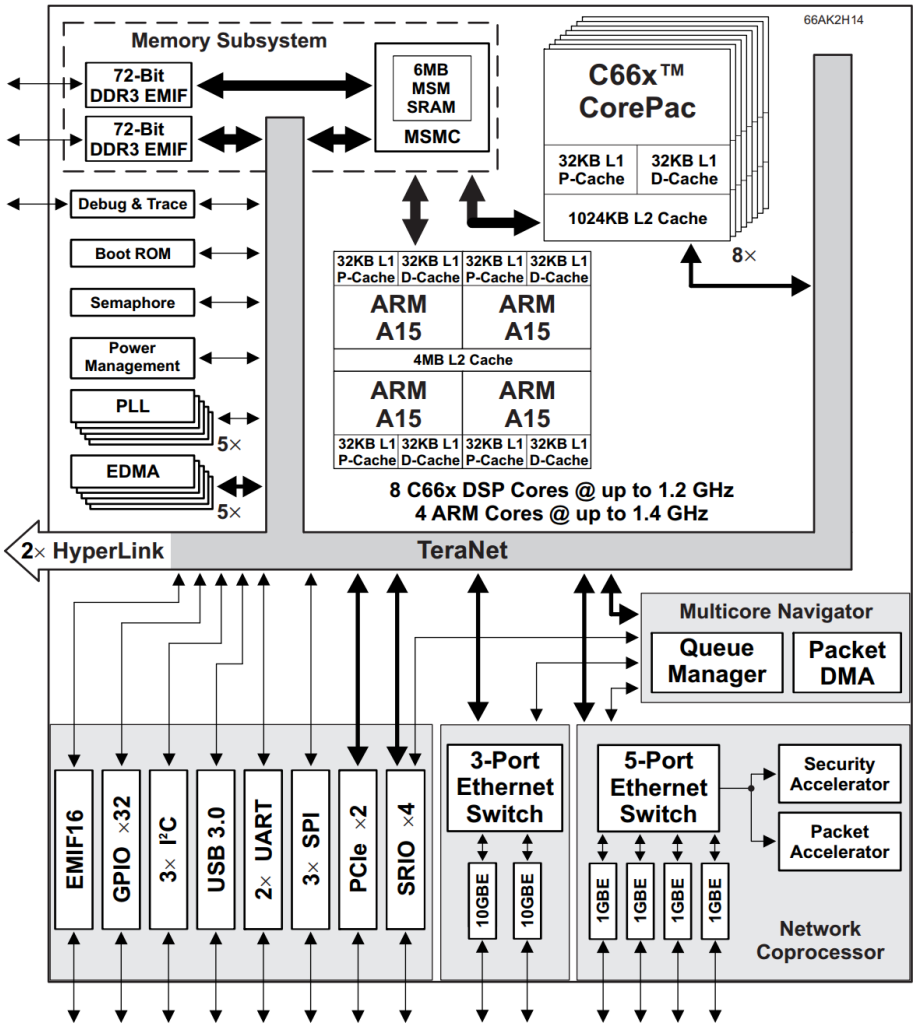


Figura 44: Diagrama de bloques del SoC 66AK2H14 [117]

La tarjeta de desarrollo utilizada ha sido la EVMK2H (figura 45), que cuenta como características más reseñables para este trabajo con 4 GB de memoria RAM, dos interfaces Ethernet a 1 Gbps y permite conectar un emulador JTAG para depuración[5].

Figura 45: Tarjeta de desarrollo EVMK2H

A nivel de *software* existen de nuevo similitudes con OMAP 3530. Los núcleos ARM corren también un sistema operativo Linux, aunque no se emplea la distribución Ångström sino una específica creada por Texas Instruments denominada Arago. Esta distribución está creada con Yocto [121], un sistema de generación de distribuciones Linux personalizadas para sistemas empotrados que es, en cierto modo, una evolución o extensión de OpenEmbedded. Es un sistema de cierta complejidad, y aunque su uso puede no ser estrictamente necesario ya que Texas Instruments proporciona una imagen binaria del sistema operativo, sí es interesante manejarlo adecuadamente para poder personalizar la imagen del sistema o añadir nuevos paquetes de *software*. Por ello se dedican unos párrafos a continuación para describir la distribución Arago y cómo utilizar Yocto para personalizarla.

5 La tarjeta incluye un emulador XDS-200, pero para este trabajo se ha empleado un emulador Blackhawk USB560

Yocto consiste fundamentalmente en herramientas para la generación de *software* para sistemas empotrados (compilación cruzada, generación de paquetes[6], generación de imágenes binarias de sistemas de ficheros, etc.) y metadatos que indican qué componentes *software* (paquetes, entre otras cosas) se necesitan para un sistema concreto y qué dependencias existen entre estos componentes.

Aparte del compilador cruzado, la principal herramienta que emplea Yocto se llama Bitbake y es el sistema de control de la generación del *software*: compilación, enlazado, gestión de dependencias, etc. Es en cierto modo similar a otros sistemas de gestión de la compilación como Make, CMake, SCons o Apache Ant, aunque está específicamente orientada a la generación de *software* para sistemas empotrados. Bitbake llama *recetas* (*recipes*) a los metadatos que le proporcionan la información necesaria para, por ejemplo, generar un paquete: de dónde descargar el código fuente, qué modificaciones (parches) específicos para su uso en sistemas empotrados hay que aplicar, qué dependencias tiene con otros paquetes, etc.

Las recetas que tienen alguna relación entre ellas se agrupan en *capas* (*layers*) que proporcionan una determinada funcionalidad. Por ejemplo, la capa *openembedded-core* agrupa recetas que se consideran básicas para generar el *software* de cualquier sistema empotrado basado en Linux; la capa *meta-ti* contiene las recetas que proporcionan los BSP (*Board Support Package*) para sistemas basados en procesadores de Texas Instruments; y la capa *meta-sdr* contiene recetas de programas y librerías relacionadas con radio *software*. Las capas pueden tener a su vez dependencias entre ellas que conforman una relación jerárquica. De hecho, la distribución Arago consiste básicamente en una capa, *meta-arago-distro*, que incluye (depende de) las capas *openembedded-core* y *meta-ti*, entre otras.

Las instrucciones de instalación del SDK (*Software Development Kit*) de Texas Instruments para la plataforma K2H [122] indican cómo obtener e instalar Yocto y las capas que componen Arago. Una vez instalado el SDK se puede obtener un listado de las recetas disponibles, junto con la capa a la que pertenece cada una, con el comando:

```
bitbake-layers show-recipes
```

Para generar un paquete únicamente hay que ejecutar el comando bitbake con la receta deseada como parámetro. Por ejemplo, el siguiente comando genera el paquete ntp:

```
bitbake ntp
```

La mayoría de las recetas, como en el ejemplo anterior, generan únicamente un paquete (junto con los de sus dependencias, si las hubiera), pero existen también

6 El gestor de paquetes empleado en las distribuciones creadas con Yocto es *Opkg*, el mismo que empleaba la distribución Ångström mencionada en la sección 5.1.

recetas más complejas que agrupan varios paquetes o que realizan otras tareas, como por ejemplo generar una imagen binaria del sistema de ficheros completo una vez que se ha obetnido, configurado, compilado e instalado todo el *software* necesario para ello.

Se ha mencionado en esta descripción una capa, *meta-sdr*, que contiene recetas relacionadas con radio *software*, incluyendo el propio GNU Radio. Sin embargo, en el trabajo presentado en esta memoria no se ha utilizado esta capa, debido a que las recetas no permiten la flexibilidad en la configuración de los paquetes que se requiere para el desarrollo del propio *software* al que se refieren las recetas. Por poner solo un ejemplo, para este trabajo se han utilizado varias instalaciones de GNU Radio con diferentes opciones de compilación, desde una sin ninguna optimización y con toda la información posible para depuración, empleada para el desarrollo, hasta otra con opciones de optimización máxima y sin información de depuración, empleada para las pruebas de rendimiento. Las recetas incluidas en las capas no permiten habitualmente este tipo de configuraciones tan específicas para tareas de desarrollo.

Los núcleos C66x, por su parte, utilizan de nuevo el RTOS propio de Texas Instruments, aunque su nombre, que hasta la versión 5, la utilizada en el trabajo desarrollado en la plataforma OMAP 3530, era DSP/BIOS, ha pasado a ser *SYS/BIOS* en la versión 6, que ha sido la empleada en la plataforma KeyStone II.

Este cambio de nombre de DSP/BIOS a SYS/BIOS, y posteriormente a *TI-RTOS* (aunque este último nombre se sigue usando en combinación con SYS/BIOS) forma parte del intento de Texas Instruments por homogeneizar las APIs de desarrollo *software* en sus múltiples plataformas *hardware*. Hay que decir, sin embargo, que esta homogeneización, que ha afectado especialmente a las APIs de comunicación entre procesadores, ha sido en general bastante desordenada, hasta el punto de que durante la realización de este trabajo de tesis en la plataforma KeyStone II ha habido cambios de nombre en SDKs y componentes *software*, componentes *software* que se han movido de un SDK a otro, e incluso APIs que han sido abandonadas mientras la documentación disponible seguía indicando su uso. Documentación que, por otro lado, Texas Instruments parecía ir escribiendo sobre la marcha, a menudo muchos meses después de haber liberado el *software* que debía describir, y que estaba disponible no como los manuales habituales en formato PDF, sino en forma de Wiki en la que secciones enteras contenían simplemente la anotación "*TBD*" (presumiblemente *To Be Defined/Described/Documented* o similar). Esto ha supuesto en determinados momentos un retraso importante en el desarrollo del trabajo.

En lo que concierne a la tarea de portado que se está describiendo, el cambio más notorio es que el API de comunicación entre procesadores se agrupa en un compo-

nente denominado *IPC* (*Inter-Processor Communications*). Hay dos APIs que sustituyen al API MSGQ descrita en la sección 5.1:

- *MessageQ* [123] para el plano de control.
- *MsgCom* [124] para el plano de datos.

Es decir, según la documentación del momento de Texas Instruments MessageQ es un API de intercambio de mensajes cortos, de control, mientras que MsgCom es un API para intercambiar mensajes que contienen grandes cantidades de datos, y que aprovecha todos los recursos *hardware* de la interconexión TeraNet y del módulo Multicore Navigator (véase la figura 44) para maximizar la tasa de transferencia con un uso reducido de CPU.

El acceso a memoria compartida, por otro lado, se gestiona mediante CMEM, al igual que en la plataforma OMAP 3530, y su API recibe únicamente algunos cambios menores.

5.2.1 *Adaptación del gestor de trabajos*

La adaptación del gestor de trabajos a la plataforma KeyStone II debía resultar en principio sencilla. El grueso del trabajo debía ser la actualización del código de intercambio de mensajes entre GPPs y aceleradores (núcleos C66) sustituyendo el API MSGQ por MsgCom, que a priori era la opción más adecuada dados los requerimientos de transferencia de datos entre núcleos ARM y C66. Esta tarea no parecía compleja, ya que la funcionalidad ofrecida por ambas APIs es similar. En la mayoría de los casos existe una equivalencia directa entre funciones de MSGQ y MsgCom y solo había que sustituir unas por otras, como por ejemplo cambiar las llamadas a MSGQ_put() por Msgcom_putMessage(). La principal diferencia entre ambas APIs está en que MsgCom requiere una inicialización más compleja, ya que permite elegir entre varios mecanismos de transporte de mensajes en función de las características concretas de la comunicación, como por ejemplo si la comunicación se realiza entre procesadores de igual o distinto tipo, o de si el volumen de datos a transferir requiere el uso de recursos específicos de la interconexión TeraNet.

En este punto se hicieron evidentes las carencias en cuanto a documentación que se han mencionado antes, ya que las descripciones de algunas de las funciones del API o de sus parámetros eran excesivamente escuetas y en varios de los posibles casos de uso faltaban incluso las secciones de documentación en las que se describían. La principal guía de uso eran algunos programas de ejemplo suministrados con el SDK, pero eran pocos y solo cubrían algunos de los casos de uso más sencillos.

Con todo, el mayor inconveniente surgió al realizar una consulta acerca del uso de MsgCom en un foro de soporte de Texas Instruments [125]; la respuesta fue que el API MsgCom estaba en proceso de ser retirada y se recomendaba sustituirla por el API MessageQ. Este hecho no se indicaba ni en la documentación específica de ninguna de las dos APIs ni en la descripción general en la que se describía el uso previsto para cada una de ellas. Ante esta información se optó por reescribir nuevamente el código de intercambio de mensajes empleando el API MessageQ.

El API MessageQ es también muy similar a MSGQ, y existe una correspondencia casi directa entre una y otra, aún en mayor grado que en el caso de MsgCom. Sin embargo, la documentación indica claramente que MessageQ está diseñada para el intercambio de mensajes cortos de control (aunque no se indica el tamaño máximo de los mensajes). A diferencia de MsgCom, MessageQ no utiliza los recursos disponibles en la interconexión TeraNet para agilizar el movimiento de datos, por lo que se emplea el mismo esquema de gestión explícita de *buffers* en memoria compartida y traducción de punteros mediante el *driver* CMEM que se describió en la sección 5.1.

Una vez aclarada la situación de las APIs de comunicaciones, la adaptación del gestor de trabajos a la arquitectura KeyStone II se realizó, como inicialmente se preveía, sin complicaciones reseñables, en un plazo de unas pocas semanas. Tras finalizar la adaptación se realizó el portado de la aplicación elegida como caso de uso, un receptor DVB–T, y a continuación se seleccionaron las partes del mismo con mayor carga computacional para distribuirlas entre todos los procesadores disponibles. Este trabajo de portado y paralelización de la aplicación se describe con detalle más adelante, en la sección 5.2.2. Antes de entrar en ello, no obstante, se expondrán dos características que se han añadido al gestor de trabajos:

- Se ha implementado una API de tipo *malloc* para la gestión de *buffers* de memoria compartida.

- Se ha añadido la posibilidad de seleccionar un trabajador concreto para el procesamiento de un trabajo.

Aunque ambas son características generales, es decir, no están ligadas a la implementación de una aplicación concreta, su conveniencia se hizo patente durante la implementacion del caso de uso elegido, el receptor DVB–T.

A continuación se describen tanto la motivación como la implementación de estas dos mejoras, para terminar esta sección sobre la adaptación del gestor de trabajos con una breve descripción del bucle de atención a mensajes que constituye la base del código que se ejecuta en los núcleos C66.

5.2.1.1 *API para la gestión de* buffers *de memoria compartida*

Como ya se ha explicado anteriormente, la gestión de la memoria compartida entre los núcleos ARM y C66 se realiza mediante el API que proporciona CMEM, que prácticamente no cambia entre una plataforma y otra. El gestor de trabajos no interviene en este proceso, y son los bloques huecos los que directamente reservan y liberan los *buffers* compartidos como mejor convenga. Esta aproximación permite emplear el esquema de gestión de memoria que mejor se adapte a cada función; por ejemplo, en una función se podrían reservar los *buffers* compartidos en la instanciación del bloque hueco y se liberarían únicamente cuando este se destruya, mientras que en otra puede resultar más adecuado reservar los *buffers* compartidos cada vez que se envíe una petición de trabajo y liberarlos inmediatamente tras recibir y procesar la respuesta.

En la plataforma OMAP 3530, tal como se describió en la sección 5.1, se implementaron únicamente algunas funciones sencillas para validar la extensión propuesta a GNU Radio. En ellas los bloques huecos empleaban directamente el API CMEM para gestionar los *buffers* compartidos, y no había ningún inconveniente en ello puesto que todas las aplicaciones usaban únicamente una función acelerada, y de hecho solo instanciaban un bloque hueco.

En la plataforma KeyStone II, por el contrario, se ha implementado una aplicación considerablemente más compleja que emplea varias funciones aceleradas, lo que presenta dos problemas. Por un lado, el API CMEM requiere una inicialización única que habría que coordinar si varios bloques huecos la utilizan. Por otro, el API CMEM está diseñado para un uso muy planificado de la memoria, con pocas operaciones de reserva y liberación, y ofrece poca flexibilidad para gestionar un escenario en el que varios hilos concurrentes pueden reservar y liberar *buffers* sin coordinación entre ellos.

Para resolver este problema se ha diseñado la siguiente solución: la aplicación reserva un único *buffer* de memoria compartida con el tamaño suficiente para cubrir las necesidades de todos los bloques huecos que vaya a usar (en la práctica, el tamaño de la memoria compartida MSMC, véase la figura 44), y se emplea un gestor de memoria dinámica para repartir este *buffer* compartido entre los bloques huecos que lo necesiten. El gestor de memoria dinámica empleado es el conocido *dlmalloc* [126], publicado por Douglas S. Lea, profesor de la Universidad Estatal de Nueva York en Oswego (*State University of New York at Oswego*), en 1987, y cuya última revisión data de 2012. Este gestor proporciona las dos funciones estándar *malloc()* y *free()*, y se ha diseñado una pequeña clase C++ llamada *cmem_mgr* a modo de envoltorio (*wrapper*) que facilita su uso. La clase tiene un método estático, *make*, que crea una instancia del gestor de memoria que, a su vez, utiliza el API CMEM para reservar la zona de

memoria compartida. Es el gestor de trabajos quien como parte de su inicialización se encarga de ello, como en el siguiente ejemplo que reserva una zona de 4 MB:

```
d_cmem_mgr = cmem_mgr::make(4 * 1024 * 1024);
```

A partir de ese momento los bloques huecos pueden emplear los métodos estáticos *malloc* y *free* de la clase *cmem_mgr* para reservar y liberar *buffers* de memoria compartida, que estarán dentro de la zona reservada por el gestor de memoria:

```
static void* cmem_mgr::malloc(size_t size);
static void cmem_mgr::free(void* buffer);
```

De ese modo la utilización del API CMEM queda restringida a la implementación del gestor de memoria, y no hay problema alguno en reservar y liberar de modo concurrente cuantos *buffers* compartidos sean necesarios (mientras no se agote la zona compartida, obviamente).

5.2.1.2 *Selección de un trabajador concreto para el procesamiento de un trabajo*

Esta mejora es menos relevante que la anterior, pero aún así puede facilitar la implementación de determinadas funciones. Se trata simplemente de permitir que al enviar una petición de trabajo al gestor de trabajos se pueda especificar un trabajador concreto (que en la implementación en la plataforma KeyStone II es equivalente a decir un núcleo C66 concreto) que debe ejecutar el trabajo. Esto puede ser beneficioso en funciones que guarden información de estado entre invocaciones sucesivas, como podría ser por ejemplo el decodificador de Viterbi no paralelo que se describe un poco más adelante, en la sección 5.2.2.1.

En función de la arquitectura *hardware* de la plataforma en la que se esté implementando la aplicación, el garantizar que es siempre el mismo acelerador el que ejecuta los trabajos que corresponden a un bloque hueco puede evitar tener que transferir la información de estado primero a y luego desde el acelerador entre trabajos sucesivos. A cambio, el seleccionar un acelerador limita las opciones disponibles para el gestor de trabajos y puede llevar a un uso poco eficiente de los aceleradores, de modo que hay que sopesar cuidadosamente las ventajas e inconvenientes que pueda tener esta opción antes de decidirse a usarla.

5.2.1.3 *Adaptación del bucle de atención a mensajes*

La adaptación del bucle de atención a mensajes que se ejecuta en los núcleos C66 no tiene en realidad ninguna particularidad más allá de lo que ya se ha explicado en el inicio de esta sección, pero se incluye en este punto para ofrecer una imagen completa del funcionamiento de la extensión de GNU Radio.

El listado 9 muestra la construcción de la tabla de funciones implementadas, que en este ejemplo son cinco. Las líneas 1 a 8 definen la tabla de nombres de las funciones, que se utiliza en la comunicación inicial entre el gestor de trabajos y los aceleradores para construir una tabla de identificadores numéricos que son los que se emplean en el resto de las comunicaciones. Las líneas 10 a 14 contienen la declaración de los prototipos de las funciones, y finalmente las líneas 16 a 18 construyen la tabla de funciones que se empleará más adelante para determinar qué función es la que hay que ejecutar en cada petición de trabajo recibida.

```
1  static proc_def_t function_table =
2  {
3    "add_ii",
4    "viterbi_decoder_bb",
5    "fft_vcc",
6    "null",
7    "copy"
8  };
9
10 extern void fn_add_ii(job_msg_t * args);
11 extern void fn_viterbi_decoder_bb(job_msg_t * args);
12 extern void fn_fft_vcc(job_msg_t * args);
13 extern void fn_null(job_msg_t * args);
14 extern void fn_copy(job_msg_t * args);
15
16 typedef void (*ptrFunc)(job_msg_t *);
17 static ptrFunc fn_table[] = {fn_add_ii, fn_viterbi_decoder_bb, fn_fft_vcc,
18                              fn_null, fn_copy};
```

Listado 9: Definición de la tabla de funciones

El listado 10 muestra de forma simplificada el código que se ejecuta en los núcleos C66 tras el arranque[7]. Las líneas 1 y 2 crean y abren, respectivamente, las dos colas de mensajes que se emplean para la comunicación entre los GPP y los aceleradores. Las líneas 4 a 9 construyen y envían el mensaje inicial de configuración que el acelerador envía al GPP, tal como se describió en la figura 37 en la sección 4.2.3.

```
1   h_rx_queue = MessageQ_create(rx_queue_name, NULL);
2   MessageQ_open(tx_queue_name, &id_tx_queue);
3
4   initial_msg_t* initial_msg =
5       (initial_msg_t *) MessageQ_alloc(0, sizeof(initial_msg_t));
6   MessageQ_setMsgId(initial_msg, DNUM);
7   initial_msg->core_id = DNUM;
```

7 En el código mostrado se omiten algunas partes que dificultan la legibilidad y no resultan relevantes para la funcionalidad que se desea mostrar, como por ejemplo código para la comprobación y el tratamiento de errores.

```
 8    initial_msg->fn_id_tbl = function_table;
 9    MessageQ_put(id_tx_queue, (MessageQ_Msg) initial_msg);
10
11    do {
12      job_msg_t * job_msg;
13
14      MessageQ_get(h_rx_queue, (MessageQ_Msg *) &job_msg, MessageQ_FOREVER);
15
16      (*(fn_table[job_msg->job.proc_id]))(job_msg);
17
18      MessageQ_put(id_tx_queue, (MessageQ_Msg) job_msg);
19    } while (1);
```

Listado 10: Bucle de atención a mensajes

El bucle de atención a mensajes propiamente dicho está en las líneas 11 a 19 y es prácticamente autoexplicativo: se recibe el mensaje de petición de trabajo (línea 14), se llama a la función que procesa la petición (línea 16) empleando el identificador de función que contiene el mensaje como índice en la tabla de funciones, y finalmente se devuelve el resultado (línea 18). El bucle se ejecuta indefinidamente.

5.2.2 *Portado y paralelización del receptor DVB–T*

Una vez descrito el trabajo de adaptación de la extensión de GNU Radio a la plataforma KeyStone II, en esta sección se describe la implementación de una aplicación compleja, un receptor DVB–T, que emplea la extensión de GNU Radio para aprovechar todos los procesadores disponibles en la plataforma.

El punto de partida es el receptor DVB–T que se incorporó en 2015 al módulo *gr-dtv* de GNU Radio. El receptor consta de una serie de bloques implementados en C++ cuyo portado a la plataforma KeyStone II es inmediato, sin más que compilarlos en ella, pero la aplicación del receptor en sí, que se muestra en la figura 46, está realizada con *GRC*, la herramienta gráfica incluida en GNU Radio (véase la sección 4.1) para la construcción de aplicaciones. GRC permite crear gráficamente un grafo de procesado de señal conectando entre sí los bloques disponibles, y genera a partir de la descripción gráfica una aplicación codificada en Python. Tal como se discutió en la sección 4.2 el subsistema Python no se ha portado a la plataforma KeyStone II por lo que ha sido necesario reescribir la aplicación en C++, una tarea que resulta muy sencilla.

Para realizar todas las pruebas de funcionamiento del receptor en sus diferentes versiones se ha utilizado un *software* denominado *gbDVB* [127] que permite simular toda la cadena de transmisión de un sistema DVB–T. Con este *software* se ha generado un fichero que simula la salida del frontal de RF del receptor en banda base,

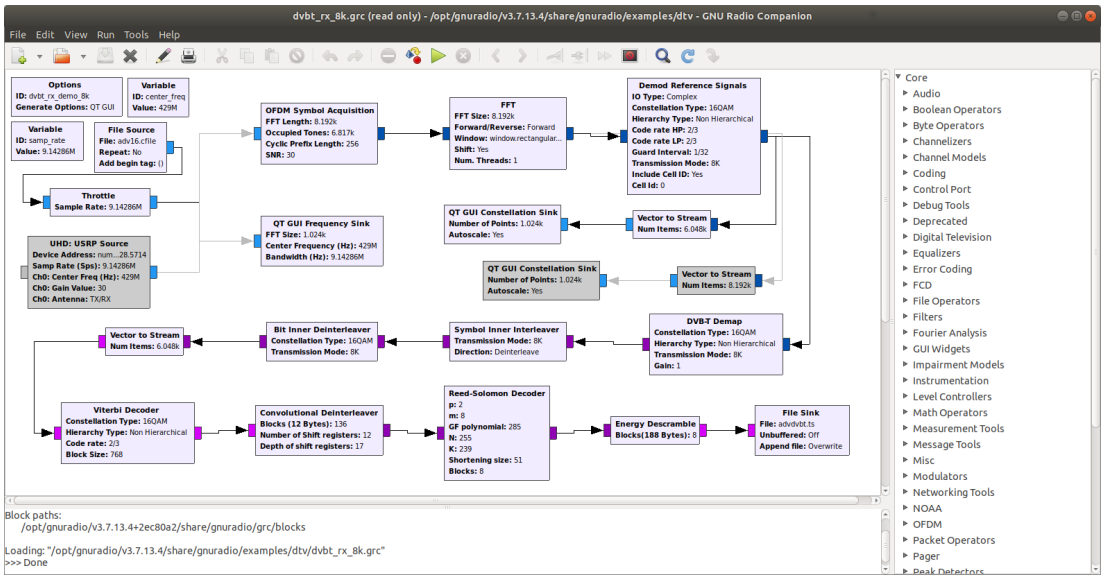


Figura 46: Receptor DVB–T distribuido con GNU Radio

Tabla 3: Características de la señal DVB–T empleada en todas las pruebas

Modo	2K
Ancho de banda	5 MHz
Intervalo de guarda	1/4 de símbolo
Tasa de codificación	1/2
Bits por símbolo	2

muestreada en cuadratura. Las características de la señal simulada se muestran en la tabla 3.

El fichero de pruebas contiene 11 411 símbolos OFDM con 2 560 muestras por símbolo (2 048 más el intervalo de guarda). La tasa de muestreo es igual al ancho de banda multiplicado por 8/7, 5,714285 Mmuestras/s, de modo que la duración de la señal simulada es 5,112128 segundos. El formato de almacenamiento de las muestras es en punto flotante (IEEE 754) de 32 bits, por lo que cada muestra ocupa 8 *bytes* (4 para la componente I y otros 4 para la componente Q) y el fichero completo tiene 233 697 280 *bytes*, casi 223 MB.

Tabla 4: Resultados del perfilado del receptor DVB–T

Bloque	Uso de CPU (%)
Decodificador Viterbi	52.6
FFT	16.1
Procesado de señales de referencia	12.1
Demapeo de símbolo	8.7
Adquisición de símbolo	6.3
Otros	4.1

Tras comprobar el correcto funcionamiento del receptor con la señal de pruebas se ha realizado un perfilado del mismo para obtener el porcentaje de uso de CPU [8] de cada bloque. Los resultados de este perfilado se muestran en la tabla 4.

Claramente, el bloque que más CPU consume es el decodificador convolucional o de Viterbi, que emplea más de la mitad del tiempo total de CPU del receptor, seguido ya a cierta distancia por el bloque que realiza el cálculo de la transformada discreta de Fourier. Por tanto el primer paso para validar el funcionamiento de la implementación de la extensión de GNU Radio es trasladar la ejecución de este bloque a los núcleos C66. Este portado se ha realizado en dos fases; en primer lugar se ha implementado una versión del decodificador que se ejecuta en un único núcleo C66, y a la vista de los resultados obtenidos se ha realizado una segunda versión que emplea todos los núcleos C66 disponibles. Ambas versiones, junto con los resultados obtenidos, se describen en detalle en las dos siguientes secciones, 5.2.2.1 y 5.2.2.2. Por último se ha llevado también la ejecución de la FFT a los núcleos C66, tal como se describe en la sección 5.2.2.3.

5.2.2.1 *Portado del decodificador Viterbi a un núcleo C66*

El primer paso para trasladar la decodificación de Viterbi a un núcleo C66 fue realizar una búsqueda de implementaciones disponibles en C o C++ que pudieran adaptarse con relativa facilidad. El hecho de que el código convolucional empleado en DVB–T sea muy común[9] facilitó esta tarea, y en el momento de realizar esta búsqueda existían dos implementaciones libremente disponibles, ambas en C: una

8 Recuérdese que en este punto el receptor DVB–T se ejecuta íntegramente en los núcleos ARM del KeyStone II.

9 Como ya se describió en la sección 2.2, es un código convolucional de tasa $1/2$ con 64 estados ($K = 7$) cuyos polinomios generadores son $G_1 = 171_{OCT}$ y $G_2 = 133_{OCT}$. Este código es conocido por haber sido empleado, entre otros usos, en multitud de misiones de la agencia espacial estadounidense, como por ejemplo las sondas *Voyager*.

debida a Phil Karn [99] y otra a Chip Fleming [128]. La primera es, de hecho, la que usan como base las dos implementaciones que incluye el propio GNU Radio, una en el módulo *gr-fec* y otra en el *gr-dtv*. Sin embargo presenta el inconveniente de incorporar una serie de optimizaciones para un código convolucional concreto y para poder utilizar las instrucciones SIMD de determinados procesadores, como por ejemplo las instrucciones MMX, SSE y sucesivas de la arquitectura x86. Aunque esto hace que sea una implementación relativamente eficiente, también hace que el código sea más complejo y dificulta en cierta medida su adaptación a otros usos.

Por ello se decidió utilizar la implementación de Fleming para este cometido, ya que aunque tampoco es completamente genérica emplea un código más general, con un estilo incluso didáctico[10] que facilita su adaptación y posterior modificación.

El código original está diseñado para ejecutarse en un ordenador personal y la función que realiza la decodificación recibe como parámetro un puntero a un *buffer* que contiene toda la secuencia de datos codificados. El uso en GNU Radio, por el contrario, implica ejecutar multitud de veces la función de decodificación, que va recibiendo una secuencia de datos codificados que en principio es ilimitada y está dividida en trozos cuyo tamaño es variable, ya que depende de la longitud de los *buffers* de entrada y salida que haya seleccionado el planificador de GNU Radio y de la secuenciación en la activación de los bloques que forman la aplicación. Por tanto es necesario preservar la información de estado del decodificador entre llamadas consecutivas a la función de decodificación.

La información de estado se ha agrupado en una estructura C llamada *vit_decod* y se ha separado el código del decodificador en dos funciones: *vit_instantiate*, que inicializa la información de estado de una nueva instancia de decodificador, y *vit_decode_data*, que realiza la decodificación de un lote de datos. El modo de uso del decodificador se ejemplifica con el siguiente pseudo-código:

```
struct vit_decod vd;

vit_instantiate(&vd);
while (data_exists) {
  n_output_data = vit_decode_data(&vd, input_buffer, output_buffer,
                                  n_input_data);
}
```

10 De hecho, el código de Fleming se distribuye dentro de un tutorial sobre codificación convolucional.

El bloque hueco GNU Radio envía dos tipos de mensajes al acelerador. El envío del primero, para la instanciación de un decodificador, se produce en el constructor del bloque hueco, tal como se muestra a continuación[11] en el listado 11:

```
1  viterbi_decoder_bb_impl::viterbi_decoder_bb_impl()
2    : gr::sync_decimator("viterbi_decoder_bb",
3                         gr::io_signature::make(1, 1, sizeof(unsigned char)),
4                         gr::io_signature::make(1, 1, sizeof(unsigned char)),
5                         2),
6      d_proc_name("viterbi_decoder_bb")
7  {
8    d_jm = job_manager::singleton();
9    d_fn_id = d_jm->lookup_proc(d_proc_name);
10
11   d_dec = (vit_decod*) cmem_mgr::malloc(sizeof(vit_decod));
12   job_descriptor_t* jd;
13
14   jd = d_jm->alloc_job_descriptor();
15   jd->d_sync = &d_sync;
16   jd->proc_id = d_fn_id;
17   jd->output.nargs = 0;
18   jd->input.nargs = 2;
19   jd->input.tag[0] = GCT_U32;
20   jd->input.arg[0].u32 = 7;
21   jd->input.tag[1] = GCT_PTR;
22   jd->input.arg[1].ptr = (void *) CMEM_getPhys(d_dec);
23
24   d_jm->submit_job(jd);
25   d_jm->wait_job(jd);
26
27   d_jm->free_job_descriptor(jd);
28 }
29
30 viterbi_decoder_bb_impl::~viterbi_decoder_bb_impl()
31 {
32   if (d_dec) {
33     cmem_mgr::free(d_dec);
34   }
35 }
```

Listado 11: Constructor del bloque *viterbi_decoder_bb*

El código del constructor está en las líneas 1 a 28. La línea 8 obtiene la referencia al gestor de trabajos necesaria para utilizar su funcionalidad, como se hace por ejemplo

11 En el código mostrado se omiten algunas partes que dificultan la legibilidad y no resultan relevantes para la funcionalidad que se desea mostrar, como por ejemplo código para el tratamiento de errores tras llamar a algunas funciones.

en la siguiente línea para traducir el nombre de la función a invocar en el acelerador a un identificador. La línea 11 reserva memoria compartida para que el acelerador aloje en ella la información de estado del decodificador. En las líneas 12 a 22 se obtiene un descriptor de trabajo del gestor de trabajos y se construye el mensaje de instanciación que se envía al acelerador. La línea 24 envía el mensaje al acelerador, y la siguiente línea (25) bloquea el bloque hueco hasta la recepción de la respuesta, tras lo que solo queda liberar el descriptor de trabajo previamente reservado. Obsérvese el uso de la función *CMEM_getPhys()* del API de CMEM en la línea 22 para la traducción de direcciones virtuales a físicas, tal como se describió en la sección 5.1.

La memoria compartida reservada para la información de estado del decodificador se libera, si fuera necesario, mediante el destructor del bloque hueco (líneas 30 a 35).

El mensaje de decodificación de datos se envía cada vez que el planificador de GNU Radio invoca al método *work* del bloque hueco, cuyo código se muestra a continuación en el listado 12:

```
1  int
2  viterbi_decoder_bb_impl::work(int noutput_items,
3                                gr_vector_const_void_star &input_items,
4                                gr_vector_void_star &output_items)
5  {
6    const unsigned char *in = (const unsigned char *) input_items[0];
7    unsigned char *out = (unsigned char *) output_items[0];
8
9    job_descriptor_t * jd;
10
11   size_t nbytes = noutput_items * sizeof(unsigned char);
12   unsigned char* cmem_i = (unsigned char*) cmem_mgr::malloc(nbytes * 2);
13   unsigned char* cmem_o = (unsigned char*) cmem_mgr::malloc(nbytes);
14
15   memcpy(cmem_i, in, nbytes * 2);
16
17   jd = d_jm->alloc_job_descriptor();
18   jd->d_sync = &d_sync;
19   jd->proc_id = d_fn_id;
20   jd->input.nargs = 5;
21   // arg[0]: vit_decod *
22   // arg[1]: const unsigned char *channel_output_vector
23   // arg[2]: unsigned char *decoder_output_matrixb
24   // arg[3]: int nitems
25   // arg[4]: int initialize
26   jd->output.nargs = 1;
27   jd->input.tag[0] = (tag_t) (GCT_PTR);
28   jd->input.arg[0].ptr = (void *) CMEM_getPhys(d_dec);
29   jd->input.tag[1] = (tag_t) (GCT_PTR);
30   jd->input.arg[1].ptr = (void *) CMEM_getPhys(cmem_i);
```

```
31    jd->input.tag[2] = (tag_t) (GCT_PTR);
32    jd->input.arg[2].ptr = (void *) CMEM_getPhys(cmem_o);
33    jd->input.tag[3] = GCT_U32;
34    jd->input.arg[3].u32 = noutput_items;
35    jd->input.tag[4] = GCT_U32;
36    jd->input.arg[4].u32 = o;
37
38    d_jm->submit_job(jd);
39
40    d_jm->wait_job(jd);
41
42    int np = jd->output.arg[o].u32;
43    memcpy(out, cmem_o, np);
44
45    d_jm->free_job_descriptor(jd);
46    cmem_mgr::free(cmem_i);
47    cmem_mgr::free(cmem_o);
48
49    // Tell runtime system how many output items we produced and consumed.
50    if (np == noutput_items) {
51      // This is the most frequent case. It happens almost always, in fact.
52      return np;
53    }
54    // The number of produced items is less than the scheduler asked. It
55    // should only happen at the first call, when the traceback buffer is
56    // still empty.
57    // Tell the scheduler we consumed all the available input data, ...
58    consume_each(noutput_items * decimation());
59    // ...how many output data we produced...
60    produce(o, np);
61    // ...and finally tell that we have already called produce() on all
62    // outputs (yes, we've got only one)
63    return WORK_CALLED_PRODUCE;
64  }
```

Listado 12: Método *work* del bloque *viterbi_decoder_bb*

Las líneas 12 y 13 reservan sendos *buffers* de memoria compartida para los datos de entrada y salida del decodificador. El *buffer* compartido de entrada se llena en la línea 15, y entre las líneas 17 y 36 se construye el mensaje para el acelerador. En las líneas 38 y 40 se envía el mensaje de petición de trabajo a un acelerador y se espera la recepción de la respuesta que indica que el trabajo se ha completado. Las líneas 42 y 43 obtienen el número de datos decodificados y copian dichos datos del *buffer* compartido de salida al del bloque hueco, para después liberar los recursos utilizados (líneas 45 a 47). El resto del códgo (líneas 49 a 63) devuelven el control al planificador de GNU Radio, informándole de la cantidad de datos que el bloque ha generado.

La figura 47 muestra el resultado obtenido al ejecutar el receptor DVB–T para una secuencia. La fila inferior corresponde a la ejecución del receptor empleando únicamente los núcleos ARM, mientras que la superior muestra el resultado cuando el decodificador Viterbi se ejecuta en uno de los núcleos C66. En ambos casos la barra inferior, de color azul, muestra el tiempo real transcurrido y la barra superior, de color salmón, muestra el tiempo de CPU empleado por los núcleos ARM.

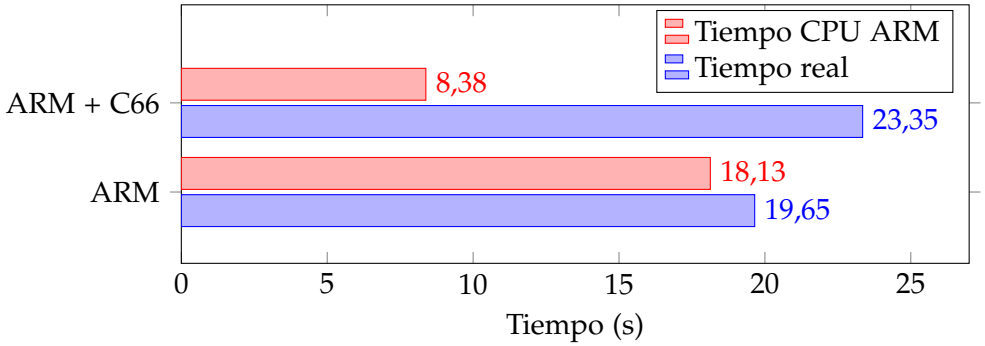


Figura 47: Tiempos de ejecución del receptor DVB–T ejecutando el decodificador Viterbi en un núcleo C66

Como puede verse, el tiempo real de ejecución del receptor aumenta en casi un 20 %, de 19,7 a 23,4 segundos, al trasladar el decodificador Viterbi a un núcleo C66. Esto es debido fundamentalmente a que la ejecución del decodificador Viterbi, que como se vio en la tabla 4 domina el tiempo de ejecución del receptor DVB–T, es más lenta en un núcleo C66 que en un núcleo ARM. Las causas de este resultado pueden ser múltiples, ya que ambos tipos de núcleo tienen diferente arquitectura y velocidad de reloj (más lenta en los C66), emplean diferentes compiladores y, no menos importante, ejecutan diferentes implementaciones del algoritmo de decodificación. Es obvio que a priori un acelerador que resulta ser más lento que el propio GPP no parece útil, pero debe recordarse que el objetivo real de este trabajo no es obtener una implementación óptima del receptor DVB–T sino validar la metodología de desarrollo que se propone.

A este respecto, también puede verse que el tiempo total de procesador ocupado en los núcleos ARM se reduce notablemente al descargar el decodificador Viterbi a un núcleo C66, con una disminución de casi un 54 %, de 18,1 a 8,4 segundos. Eso significa que aunque el tiempo total de ejecución del receptor aumente, al emplear un núcleo C66 los núcleos ARM están inactivos buena parte del tiempo, durante el cual podrían dedicarse a la ejecución de otras tareas.

Es seguro que la ejecución del decodificador Viterbi en el núcleo C66 puede mejorarse, probablemente de forma significativa. Aún así no se ha dedicado a ello ningún esfuerzo, ya que esta tarea de optimización puede requerir un trabajo significativo y, como se verá, no es necesario para el objetivo de validar la metodología de desarrollo propuesta.

Para este fin es mucho más útil demostrar que esta metodología permite emplear de forma eficiente todos los recursos de los que dispone la plataforma en la que se está trabajando. Dado que el resultado obtenido hasta el momento muestra que el decodificador de Viterbi sigue siendo el elemento dominante en la ejecución del receptor DVB–T, el siguiente paso ha sido paralelizar la implementación del decodificador para que pueda distribuirse por todos los núcleos C66 disponibles en la plataforma.

5.2.2.2 *Paralelización del decodificador Viterbi*

El algoritmo de Viterbi es a priori poco adecuado para una implementación distribuida en varios procesadores, ya que la mayor carga computacional está en el cálculo de las métricas de cada secuencia posible de estados, denominados habitualmente *caminos* (*paths*), y la posterior selección de los caminos supervivientes, eliminando los menos probables. Esta operación se suele conocer por las siglas *ACS* (del inglés *Add-Compare-Select*), y en cada paso hay que repetirla 2^{K-1} veces para evaluar otros tantos caminos[12]. Las 2^{K-1} operaciones ACS se pueden hacer en paralelo, pero tras cada iteración hay que recorrer hacia atrás los caminos supervivientes y en este proceso se actualiza la información de estado del decodificador. Esto es lo que hace inviable paralelizar el algoritmo de este modo, ya que tras cada lote de operaciones ACS habría que sincronizar los cambios en la información de estado entre todos los procesadores empleados.

Por ello se emplea otra estrategia para la paralelización: se divide la secuencia de datos a decodificar en tantos fragmentos como procesadores haya, y cada procesador decodifica uno de estos fragmentos de forma independiente a los demás. Lógicamente en los puntos de división se pierde la continuidad en la información de estado, pero esto puede solucionarse solapando los fragmentos tal como se describe en [129]: cuando el decodificador comienza su tarea en un punto arbitrario de la secuencia de datos recibidos, una vez ha procesado suficientes datos (5 veces K, la longitud de la información de estado en el codificador o *constraint length*) la salida del decodificador ya es igual a la que daría si hubiera comenzado la decodificación en el inicio de la secuencia. Por tanto, si el sistema dispone de C aceleradores:

12 Recuérdese que en el código empleado en DVB–T $K = 7$ y por tanto hay 64 estados posibles.

- La secuencia de N datos de entrada se divide en C lotes de longitud $\frac{N-5K}{C} + 5K$. El comienzo de cada lote se solapa con los últimos $5K$ datos del lote anterior.
- Se envía cada uno de los lotes a un acelerador.
- Se concatenan las salidas de los aceleradores. Cada uno de ellos genera $N - 5K$ bits de salida.

La figura 48 muestra un ejemplo de división de la secuencia de datos en un sistema con 3 aceleradores. La relación de tamaños entre los datos codificados depende de la tasa de código r; en el código empleado en DVB–T la tasa de código es $r = 2$, por lo que los datos codificados ocupan el doble que los decodificados.

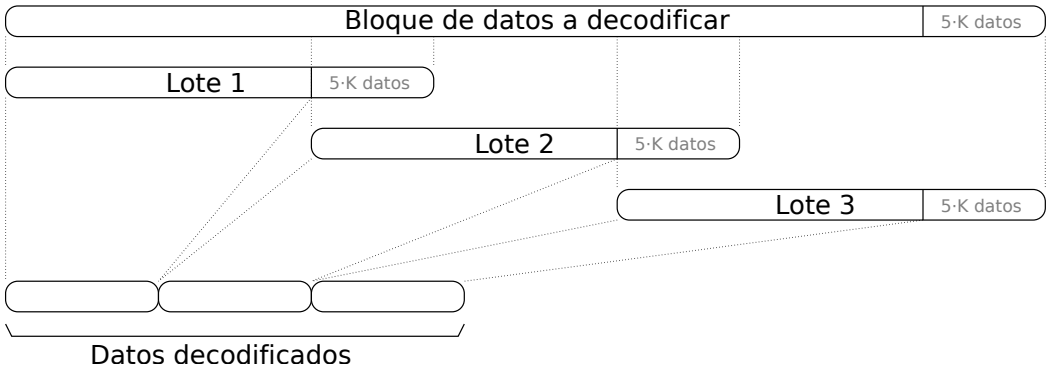


Figura 48: Paralelización del algoritmo de Viterbi según [129]

A continuación se muestra en el listado 13 la implementación del método *work* del bloque hueco de la versión paralelizada del decodificador de Viberbi. Como es habitual se han omitido algunos fragmentos de código, como por ejemplo comprobación y gestión de errores, que son necesarios en la implementación pero complicarían innecesariamente esta exposición.

```
1  int
2  viterbi_decoder_sl_bb_impl::work(int noutput_items,
3                                   gr_vector_const_void_star &input_items,
4                                   gr_vector_void_star &output_items)
5  {
6    const unsigned char *in = (const unsigned char *) input_items[0];
7    unsigned char *out = (unsigned char *) output_items[0];
8
9    unsigned char* cmem_i = (unsigned char*) cmem_mgr::malloc(noutput_items * 2);
10   unsigned char* cmem_o = (unsigned char*) cmem_mgr::malloc(noutput_items);
11
12   memcpy(cmem_i, in, noutput_items * 2);
```

```
13
14    // Slice and remainder size.
15    int njobs = d_jm->ncores();
16    int slsize, remsize;
17    slsize = (noutput_items - (5 * l_K)) / njobs;
18    remsize = noutput_items % njobs; // Will be added to last slice
19
20    job_descriptor_t** jdarr = new job_descriptor_t *[njobs];
21    bool* donearr = new bool[njobs];
22    unsigned char* start = cmem_i;
23    unsigned char* start_o = cmem_o;
24
25    for (int jobnr = 0; jobnr < njobs; ++jobnr) {
26      job_descriptor_t *jd = d_jm->alloc_job_descriptor();
27      jdarr[jobnr] = jd;
28      jd->d_sync = &d_sync;
29      jd->proc_id = d_fn_id;
30      jd->input.nargs = 5;
31      // arg[0]: vit_decod *
32      // arg[1]: const unsigned char *channel_output_vector
33      // arg[2]: unsigned char *decoder_output_matrixb
34      // arg[3]: int nitems
35      // arg[4]: int initialize
36      jd->output.nargs = 1;
37      jd->input.tag[0] = (tag_t) (GCT_PTR);
38      jd->input.arg[0].ptr = (void *) CMEM_getPhys(d_dec[jobnr]);
39      jd->input.tag[1] = (tag_t) (GCT_PTR);
40      jd->input.arg[1].ptr = (void *) CMEM_getPhys(start);
41      jd->input.tag[2] = (tag_t) (GCT_PTR);
42      jd->input.arg[2].ptr = (void *) CMEM_getPhys(start_o);
43      jd->input.tag[3] = GCT_U32;
44      jd->input.arg[3].u32 = slsize + (5 * l_K);
45      if (jobnr == njobs - 1) {
46        // The last job nedds to have its size adjusted
47        jd->input.arg[3].u32 += remsize;
48      }
49      jd->input.tag[4] = GCT_U32;
50      jd->input.arg[4].u32 = 1;
51
52      d_jm->submit_job(jd);
53
54      start += 2 * (slsize);
55      start_o += slsize;
56    } // end for(jobnr)
57
58    d_jm->wait_jobs(njobs, jdarr, donearr, WAIT_ALL);
59    int np = 0;
```

```
60    for (int jobnr = 0; jobnr < njobs; ++jobnr) {
61      np += jdarr[jobnr]->output.arg[0].u32;
62      d_jm->free_job_descriptor(jdarr[jobnr]);
63    }
64    memcpy(out, cmem_o, np);
65
66    delete[] jdarr;
67    delete[] donearr;
68    cmem_mgr::free(cmem_i);
69    cmem_mgr::free(cmem_o);
70
71    // Tell runtime system how many output items we produced.
72    return np;
73  }
```

Listado 13: Método *work* del bloque *viterbi_decoder_sl_bb*

El código es muy similar al empleado para la versión no paralelizada del decodificador. Las líneas 9 a 12 reservan los *buffers* de memoria compartida de entrada y salida y llenan el de entrada con los datos codificados. Las diferencias comienzan en las líneas 15 a 23, en las que se calcula el tamaño del lote (*slice*) que se va a enviar a cada acelerador y se preparan dos *arrays* para manejar los trabajos que se van a enviar. En las líneas 25 a 56 se construyen y envían los mensajes que para cada acelerador, actualizando en cada iteración los punteros que manejan los lotes de entrada y salida. La línea 58 suspende la ejecución del bloque hueco hasta que el gestor de trabajos notifique la finalización de todos los trabajos enviados. Finalmente se copian los datos al *buffer* de salida de GNU Radio y se liberan los recursos utilizados (líneas 59 a 69) y se devuelve el control al planificador de GNU Radio.

La figura 49 muestra el resultado obtenido al procesar la misma secuencia de prueba que se empleó en la implementación no paralela. Cada columna muestra el tiempo de ejecución del receptor con el número de aceleradores C66 indicado, donde 0 significa que la decodificación Viterbi se realiza en los núcleos ARM. De nuevo se muestra para cada caso el tiempo real de proceso a la izquierda, en color azul, y el tiempo de CPU empleado por los núcleos ARM a la derecha, en color salmón.

Los dos primeros grupos de columnas reproducen los resultados de la figura 47, como cabe esperar. Al pasar de uno a dos núcleos C66 el tiempo real empleado se reduce al 52 %, de 23,35 a 12,19 segundos, muy cerca de la mejora que idealmente se podría esperar. Al añadir un tercer C66, sin embargo, se produce de nuevo una mejora pero esta vez muy reducida, y a partir de ahí ya no hay mejora adicional al seguir añadiendo núcleos C66. Este comportamiento indica que con uno y dos C66 el tiempo de procesamiento está dominado por la ejecución del decodificador de Viterbi en los DSPs, pero a partir de la introducción de un tercer C66 el cuello de botella pasa

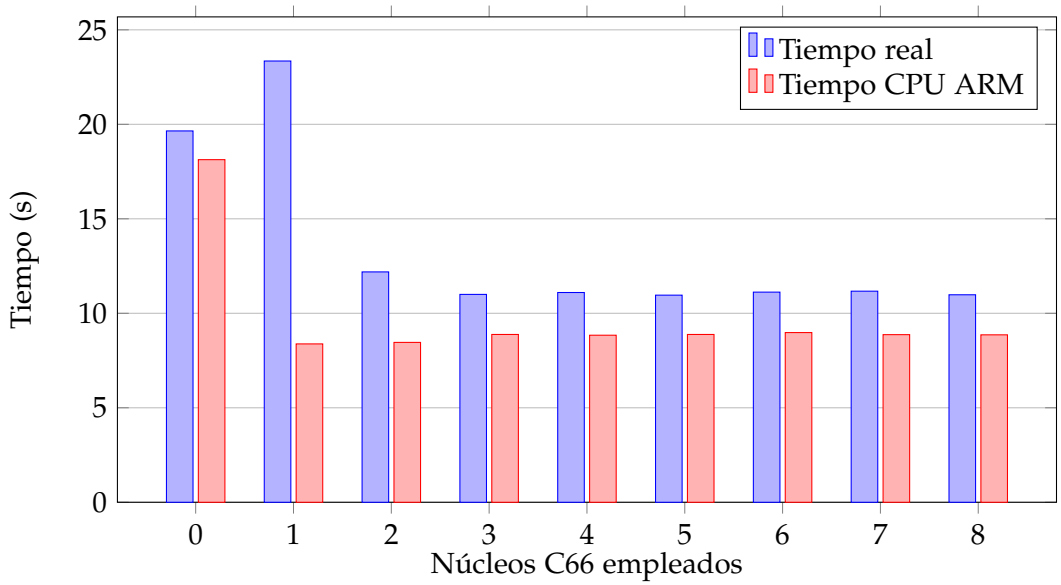


Figura 49: Tiempos de ejecución del receptor DVB–T ejecutando el decodificador Viterbi en varios núcleos C66

a estar en los núcleos ARM, de modo que reducir aún más el tiempo de ejecución del decodificador de Viterbi no tiene reflejo en el tiempo total de proceso.

El tiempo de CPU consumido en los ARM, por otro lado, tiene su mínimo al usar un solo DSP, se incrementa ligerísimamente al introducir un segundo C66 y sube un poco más con el tercero, manteniéndose prácticamente constante a partir de ahí. Este comportamiento es consistente con el hecho de que al aumentar el número de aceleradores crece la sobrecarga derivada de comunicaciones y gestión de *buffers*, aunque el efecto es tan pequeño que queda casi enmascarado en las pequeñas variaciones entre una ejecución y otra del receptor debidas a factores externos al mismo (fundamentalmente, otros procesos en ejecución en el sistema).

La conclusión obtenida de esta segunda implementación del decodificador de Viterbi es que hasta el momento el mecanismo de distribución de trabajos propuesto parece funcionar correctamente, pero hay aún margen para llevar más funciones de los núcleos ARM a los aceleradores C66. Esto permitiría además verificar la extensión propuesta con varias tareas distintas. Consultando de nuevo los datos de perfilado del receptor en la tabla 4 se puede ver que el siguiente bloque en uso de CPU en el receptor DVB–T es la FFT, con algo más de un 16 %. Aunque lejos del decodificador de Viterbi, es suficiente para que la presumible mejora en rendimiento sea significativa y, además, permite demostrar una de las características del mecanismo propuesto:

permite utilizar implementaciones optimizadas proporcionadas por el fabricante de los aceleradores cuando estas exsistan.

5.2.2.3 *FFT*

Tal como se ha explicado en el apartado anterior, el interés de descargar la ejecución de la FFT en los aceleradores C66 en el receptor DVB–T es, por un lado, demostrar el funcionamiento correcto y eficiente de la extensión de GNU Radio propuesta cuando hay más de una función acelerada, y por otro demostrar el uso de recursos proporcionados por el fabricante de los aceleradores cuando estén disponibles.

En el caso de la FFT, Texas Instruments proporciona en su librería C66X-FFTLIB [130] varias implementaciones de esta función con diversas características y grados de optimización en función de las características deseadas (número de dimensiones, tamaño, etc.). La FFT necesaria para el receptor DVB–T es relativamente sencilla, ya que su tamaño es siempre una potencia de dos[13].

El listado 14 muestra el código del método *work* del bloque hueco que realiza la FFT, con las simplificaciones que se vienen aplicando. Como puede verse, lo único que hace es copiar los datos de entrada al *buffer* compartido, construir y enviar la petición de trabajo y copiar el resultado al *buffer* de GNU Radio al recibir la respuesta. La única peculiaridad es que en este último paso existe la posibilidad de intercambiar las dos mitades del resultado (líneas 37 a 39), quedando ordenadas las frecuencias normalizadas en el rango $[-\pi, \pi)$ en vez de $[0, 2\pi)$; el receptor DVB–T utiliza esta característica ya que al manejar la señal en banda base la frecuencia central del canal de transmisión queda trasladada a 0 Hz.

```
1  int
2  fft_vcc_impl::work(int noutput_items,
3                     gr_vector_const_void_star &input_items,
4                     gr_vector_void_star &output_items)
5  {
6    const gr_complex *in = (const gr_complex *) input_items[0];
7    gr_complex *out = (gr_complex *) output_items[0];
8
9    unsigned int input_data_size = input_signature()->sizeof_stream_item (0);
10   unsigned int output_data_size = output_signature()->sizeof_stream_item (0);
11
12   for (int count = 0; count < noutput_items; ++count) {
13     job_descriptor_t* jd;
14
15     memcpy(d_x_buffer, in, input_data_size);
```

13 El tamaño de la FFT en el receptor es lo que da el nombre a cada uno de los modos de operación de DVB–T (2K, 8K).

```
16      in  += d_fft_size;
17
18      jd = d_jm->alloc_job_descriptor();
19      jd->d_sync = &d_sync;
20      jd->proc_id = d_fn_id;
21      jd->output.nargs = 0;
22      jd->input.nargs = 4;
23      jd->input.tag[0] = GCT_U32;
24      jd->input.arg[0].u32 = d_fft_size;
25      jd->input.tag[1] = GCT_PTR;
26      jd->input.arg[1].ptr = (void *) CMEM_getPhys(d_x_buffer);
27      jd->input.tag[2] = GCT_PTR;
28      jd->input.arg[2].ptr = (void *) CMEM_getPhys(d_w_buffer);
29      jd->input.tag[3] = GCT_PTR;
30      jd->input.arg[3].ptr = (void *) CMEM_getPhys(d_y_buffer);
31
32      d_jm->submit_job(jd);
33
34      d_jm->wait_job(jd);
35
36      if (d_forward && d_shift) {  // apply a fft shift on the data
37        unsigned int len = (unsigned int)(ceil(d_fft_size/2.0));
38        memcpy(&out[0], &d_y_buffer[len], sizeof(gr_complex)*(d_fft_size - len));
39        memcpy(&out[d_fft_size - len], &d_y_buffer[0], sizeof(gr_complex)*len);
40      } else {
41        memcpy(out, d_y_buffer, output_data_size);
42      }
43      out += d_fft_size;
44
45      d_jm->free_job_descriptor(d_jdarr[jobnr]);
46    }
47
48    return noutput_items;
49  }
```

Listado 14: Método *work* del bloque *fft_vcc*

El listado 15 muestra (simplificado, como es habitual) el código que se ejecuta en el núcleo C66 para obtener la FFT. Las líneas 8 a 11 extraen los parámetros del mensaje de petición de trabajo (tamaño de la FFT y punteros a tres *buffers*: entrada, factores multiplicativos (que se precalculan en la inicialización) y salida), y las líneas 12 a 27 determinan si el tamaño de la FFT es una potencia par o impar de 2, ya que la función DSPF_sp_fftSPxSP_opt de la librería C66X-FFTLIB, que es la que se utiliza para el cálculo de la FFT, requiere este dato. Finalmente en la línea 29 se llama a dicha función.

```
1  void fn_fft_vcc(job_msg_t * msg)
2  {
3    // arg[0]: N
4    // arg[1]: gr_complex *in
5    // arg[2]: gr_complex *tw
6    // arg[3]: gr_complex *out
7
8    unsigned int N = msg->job.input.arg[0].u32;
9    float *ptr_x = (float*) msg->job.input.arg[1].ptr;
10   float *ptr_w = (float*) msg->job.input.arg[2].ptr;
11   float *ptr_y = (float*) msg->job.input.arg[3].ptr;
12   int min_radix;
13   {
14     int i;
15     int j = 0;
16
17     for (i = 0; i <= 31; i++)
18       if ((N & (1 << i)) == 0)
19         j++;
20       else
21         break;
22
23     if (j %2 == 0)
24       min_radix = 4;
25     else
26       min_radix = 2;
27   }
28
29   DSPF_sp_fftSPxSP_opt(N, ptr_x, ptr_w, ptr_y, NULL, min_radix, 0, N);
30
31   msg->job.output.nargs = 0;
32 }
```

Listado 15: Código C66 para el bloque *fft_vcc*

La figura 50 muestra el resultado obtenido al procesar la secuencia de prueba. Al igual que en la figura 49, cada columna muestra el tiempo de ejecución del receptor con el número de aceleradores C66 indicado. El tiempo real de proceso se muestra en cada caso a la izquierda, con color azul, y el tiempo de CPU empleado por los núcleos ARM a la derecha, en color salmón.

Si se comparan estos resultados con los del apartado anterior (figura 49), puede verse cómo el tiempo total de ejecución usando uno o dos núcleos C66 es peor ahora, como cabe esperar: mientras el tiempo de ejecución del receptor está dominado por los núcleos C66, trasladar más trabajo hacia ellos solo puede empeorar la situación. Al añadir el tercer núcleo C66, por el contrario, el tiempo de ejecución se reduce

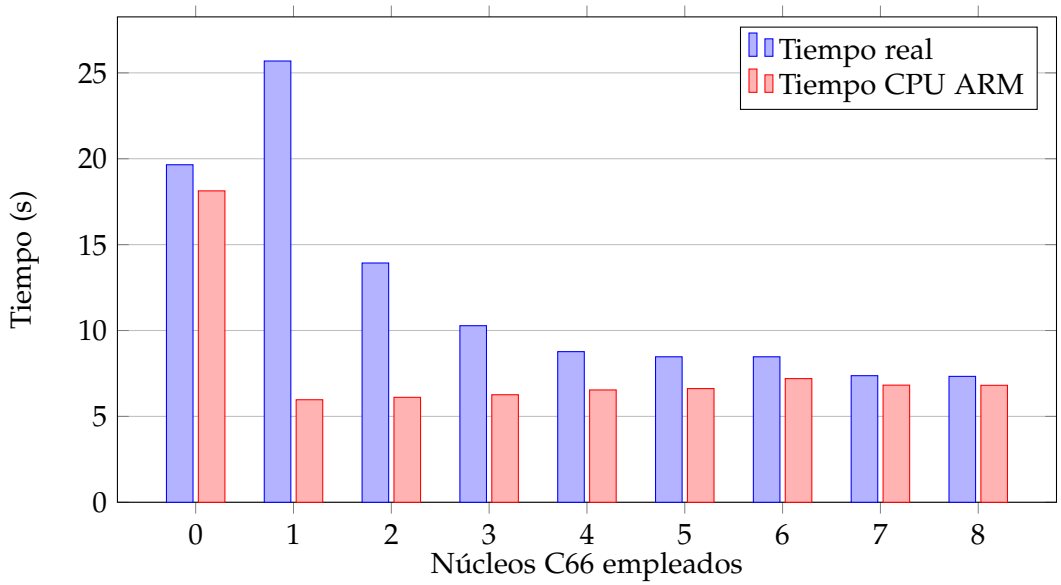


Figura 50: Tiempos de ejecución del receptor DVB–T ejecutando el decodificador Viterbi y la FFT en varios núcleos C66

al descargar los núcleos ARM del cálculo de la FFT, ya que en este caso el tiempo de ejecución pasa a esar dominado por estos últimos. Se observa también cómo el tiempo de ejecución sigue disminuyendo, aunque en mucha menor medida, al añadir aún más núcleos C66.

También es reseñable el comportamiento del tiempo de CPU empleado en los núcleos ARM, en el que se percibe un ligero ascenso con el número de núcleos C66 empleados debido al incremento en el número de mensajes intercambiados. Este efecto también es perceptible en la figura 49, pero en este caso se hace más patente debido a la disminución del tiempo total de CPU al descargar el cálculo de la FFT de los núcleos ARM.

5.2.3 *Resultados*

La figura 51 resume los resultados que se han presentado en las secciones anteriores. Las barras con fondo rayado corresponden a la versión del receptor DVB–T en la que únicamente el decodificador de Viterbi se ejecuta en los núcleos C66 (figura 49), y las barras con fondo sólido a la versión en la que lo hacen tanto el decodificador de Viterbi como el cálculo de la FFT (figura 50).

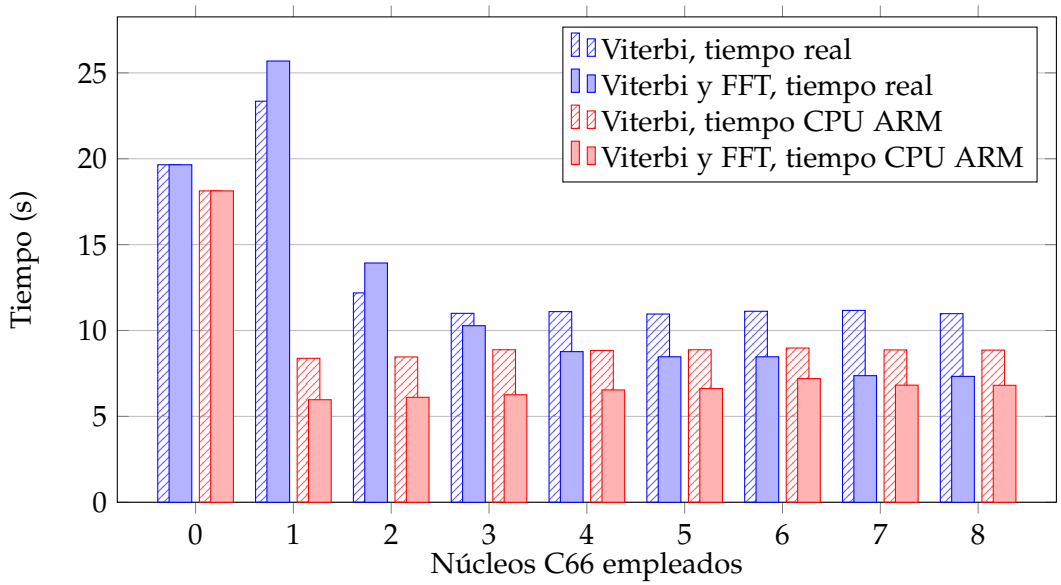


Figura 51: Comparación de los tiempos de ejecución del receptor DVB–T con el cálculo de la FFT en los núcleos ARM o C66

La superposición de ambos conjuntos de datos permite ver con mayor claridad la diferencia en el comportamiento de las dos versiones del receptor DVB–T. Lógicamente, cuando el receptor se ejecuta únicamente en los núcleos ARM (o núcleos C66) no hay diferencias. Con 1 o 2 núcleos C66 el tiempo total de ejecución está limitado por ellos en ambas versiones, y por ello cargarlos aún más con el cálculo de la FFT empeora la situación.

A partir de tres núcleos C66 el comportamiento es muy distinto en cada versión. Cuando solo el decodificador de Viterbi se ejecuta en los núcleos C66 el tiempo de ejecución pasa a estar dominado por los núcleos ARM, y por ello añadir aún más núcleos C66 no representa ninguna mejora. Por el contrario, cuando también se descarga el cálculo de la FFT en los núcleos C66 el tiempo total de ejecución sigue dominado por ellos y por tanto se sigue observando una mejora cuantos más se utilizan. Al emplear los ocho núcleos C66 disponibles se consigue una reducción de casi un 63 % en el tiempo total de ejecución del receptor, que pasa de 19,65 segundos a 7,33.

Pero es importante resaltar que el objetivo de los experimentos que se han descrito no es obtener el menor tiempo de ejecución posible en una aplicación concreta, sino verificar que la metodología de programación de sistemas multiprocesador heterogéneos que se propone en este trabajo de tesis resulta efectiva para la implementación de aplicaciones complejas. Los resultados obtenidos corroboran esto.

5.3 RESUMEN

En este capítulo se han presentado los resultados obtenidos en dos implementaciones de la extensión de GNU Radio que se describió en el capítulo 4. La primera se ha realizado en una plataforma basada en el procesador OMAP 3530 de Texas Instruments y ha servido como prueba de concepto en un sistema relativamente sencillo. La segunda se ha realizado en un procesador de la familia KeyStone II de Texas Instruments, mucho más potente y complejo, que incluye cuatro núcleos ARM y ocho núcleos DSP C66. La adaptación de la primera implementación a una nueva plataforma, aunque esta sea similar en muchos aspectos a la primera, sirve como ejemplo del portado de la extensión propuesta a nuevas arquitecturas *hardware*. Esto se trata con mayor generalidad en el siguiente capítulo.

Para evaluar la implementación en la plataforma KeyStone II se ha empleado una aplicación compleja, un receptor DVB–T. Tras un perfilado de la aplicación se ha trasladado la ejecución de dos bloques del receptor, el decodificador de Viterbi y el bloque de cálculo de la FFT, a los núcleos C66 del procesador. Los resultados obtenidos confirman la validez de la metodología de programación propuesta.

6

SÍNTESIS DE LA METODOLOGÍA PROPUESTA

En este capítulo se resume la metodología de desarrollo de sistemas sobre plataformas multiprocesador heterogéneas que se propone en este trabajo de tesis, que se ha validado implementándola sobre dos plataformas distintas y la implementación de un receptor DVB–T sobre una de las plataformas, tal como se describe en el capítulo 5.

La metodología se basa en el uso de GNU Radio, un *kit* de desarrollo para aplicaciones de radio *software* con licencia GPL. GNU Radio permite construir aplicaciones empleando un modelo de computación de flujo de datos; las aplicaciones toman la forma de un grafo cuyos nodos son bloques de procesado de señal. El planificador de GNU radio gobierna la ejecución de los bloques de procesado de señal en función de la disponibilidad de datos en los *buffers* que los conectan. El trabajo desarrollado en esta tesis extiende GNU Radio, tal como se describe en las secciones 4.2.2 y 4.2.3, de modo que se pueden sustituir uno o varios bloques de procesado de señal por lo que se ha llamado *bloques huecos*, que descargan el procesamiento real en uno o varios aceleradores externos. Los bloques huecos no se comunican directamente con los aceleradores; lo hacen a través de un elemento añadido a GNU Radio, el *gestor de trabajos*, que se encarga de gestionar la ejecución de lotes de procesamiento, denominados *trabajos*, en los aceleradores disponibles.

El diseño del gestor de trabajos, descrito en la sección 4.2.4, permite una flexibilidad absoluta en la organización de la aplicación. Es capaz de gestionar un número arbitrario de bloques, aceleradores y funciones (algoritmos implementados en los aceleradores) sin imponer ninguna restricción en la correspondencia entre aceleradores y funciones. Esta característica es especialmente reseñable porque es la que dota de generalidad a la metodología propuesta, haciendo que sea aplicable en plataformas muy dispares en cuanto al número y tipo de aceleradores disponibles. A modo de ejemplo, la metodología propuesta es aplicable en plataformas que tengan:

- Un único acelerador especializado en una única función, como por ejemplo el cálculo de la FFT.

- Varios aceleradores especializados cada uno de ellos en una única función.

- Un único acelerador programable que pueda implementar varias funciones. Un ejemplo de este tipo de plataforma sería el procesador OMAP 3530 empleado en la implementación descrita en la sección 5.1.
- Varios aceleradores programables homogéneos, como en el procesador KeyStone II empleado en la implementación descrita en la sección 5.2.
- Varios aceleradores programables pero no homogéneos, de modo que el conjunto de funciones que cada uno pueda gestionar sea distinto. Un ejemplo podría ser un procesador similar al KeyStone II en el que alguno de los núcleos C66 tuviera un acelerador *hardware* específico para una determinada función.

6.1 APLICACIÓN DE LA METODOLOGÍA PROPUESTA A UNA NUEVA PLATAFORMA

Los pasos a seguir para aplicar la metodología propuesta en una nueva plataforma son los siguientes:

- Portado de GNU Radio y de las aplicaciones de interés al o a los GPPs de la plataforma de destino.
- Selección de los mecanismos de comunicación entre procesadores de entre los disponibles en la plataforma de destino.
- Adaptación del gestor de trabajos a la plataforma de destino.
- Perfilado de las aplicaciones en la plataforma de destino.
- Traslado de los bloques seleccionados a los aceleradores.

Para el trabajo de portado puede emplearse como punto de partida la implementación en procesadores KeyStone II descrita en la sección 5.2. El código completo de dicha implementación está disponible públicamente en el servidor Gitlab del Centro de Investigación en Tecnologías Software y Sistemas Multimedia para la Sostenibilidad (CITSEM) de la Universidad Politécnica de Madrid [131].

A continuación se describe con detalle cada uno de los pasos enumerados anteriormente.

6.1.1 *Portado de GNU Radio a la plataforma de destino*

Este paso debe ser en general sencillo siempre que la plataforma de destino cumpla con los requisitos mínimos que se establecieron en la sección 4.2.1, es decir, que

disponga de un compilador de C++ y soporte las APIs POSIX para gestión de hilos (*threads*) y semáforos.

Prácticamente todas las plataformas empotradas actuales están construidas en torno a procesadores ARM que usan Linux como sistema operativo, por lo que este proceso será en general muy similar a lo que se ha descrito en el capítulo 5.

6.1.2 *Selección de los mecanismos de comunicación entre procesadores*

Este paso es, sin duda, el más dependiente de las características concretas de la plataforma de destino. Habitualmente la comunicación entre procesadores y aceleradores emplea:

- Un mecanismo *hardware* de paso de mensajes cortos, a veces con una cola FIFO que permita acumular unos pocos mensajes.
- Una zona de memoria compartida.
- Un mecanismo de notificación de eventos o interrupciones.

Con frecuencia se usa una combinación de estos elementos; por ejemplo, es habitual que el paso de mensajes cortos y la notificación de eventos estén combinados en un único mecanismo *hardware*. En plataformas complejas como la KeyStone II pueden existir otros elementos como controladores de DMA o incluso procesadores específicos dedicados exclusivamente a la gestión de las comunicaciones de los procesadores dedicados a la gestión y al cálculo.

Funcionalmente, cualquier mecanismo de comunicación es válido para implementar la metodología propuesta, aunque la elección o la presencia de un tipo de mecanismo u otro tendrá desde luego implicaciones en el rendimiento del sistema. En función de factores como la complejidad de la plataforma o la familiaridad de los desarrolladores con ella puede adoptarse incluso una estrategia progresiva, seleccionando primero el mecanismo más sencillo disponible e introduciendo después otras herramientas que permitan mejorar el rendimiento.

6.1.3 *Adaptación del gestor de trabajos a la plataforma de destino*

Una vez seleccionado un mecanismo de comunicaciones es necesario modificar el gestor de trabajos para que haga uso de él. En la sección 5.2.1 se describe este proceso para la plataforma KeyStone II en la que se ha validado la metodología propuesta.

El trabajo que requiere este paso tiene también una gran dependencia de las características concretas de la plataforma de destino. En caso de emplear una plataforma

comercial, lo habitual será que el fabricante de la misma proporcione una o varias APIs para comunicación entre procesadores que facilite en gran medida esta tarea; si se trabaja sobre una plataforma *hardware* diseñada específicamente para la aplicación, este paso puede requerir un esfuerzo considerable.

En el servidor Gitlab del CITSEM puede encontrarse un diseño de referencia [132] (`https://gitlab.citsem.upm.es/pedro.lobo/gr-gap`) que sirve como plantilla para implementar el gestor de trabajos en una nueva plataforma, así como los gestores de trabajos de las dos implementaciones realizadas en este trabajo de tesis.

6.1.4 *Perfilado de las aplicaciones en la plataforma de destino*

Este paso permite seleccionar qué bloques de la aplicación de interés son los mejores candidatos para descargar en los aceleradores disponibles. Es muy habitual que ya se disponga de información a priori, ya que el coste computacional de los algoritmos de procesado de señal es siempre un factor relevante y tanto la reducción de dicho coste computacional como la optimización tanto a nivel *hardware* como *software* de las implementaciones de los algoritmos reciben grandes esfuerzos de investigación y son objeto de numerosas publicaciones en la literatura científica.

Nótese que se está suponiendo que ya se dispone de una implementación en GNU Radio de la aplicación de interés; esta tarea queda fuera del ámbito de la metodología propuesta.

En la sección 5.2.2 se describe la realización de este paso para la aplicación seleccionada para la validación de la metodología propuesta en este trabajo de tesis.

6.1.5 *Traslado de bloques a los aceleradores*

El último paso es trasladar a los aceleradores los bloques de procesado de señal seleccionados. De nuevo, el trabajo a realizar depende fuertemente de las características de la plataforma de destino. Es posible emplear distintas estrategias de implementación en función de las características del algoritmo que se desea portar y del número y tipo de aceleradores disponibles: portar el algoritmo completo o solo una parte, utilizar un único acelerador o paralelizar el trabajo empleando varios aceleradores, almacenar información de estado (si fuera necesaria) en los aceleradores o en el bloque hueco, etc.

En las secciones 5.2.2.1 a 5.2.2.3 se describe en detalle este proceso para la plataforma KeyStone II en la que se ha validado la metodología propuesta. El proceso de portado del decodificador de Viterbi (secciones 5.2.2.1 y 5.2.2.2) muestra un ejemplo

de cómo es posible adoptar diferentes estrategias de implementación en función de las características del algoritmo y de la plataforma.

En el diseño de referencia disponible en el servidor Gitlab del CITSEM [132] puede encontrarse también una plantilla de implementación de un bloque hueco, además del código de las dos implementaciones realizadas en este trabajo de tesis.

6.2 RESUMEN

En este capítulo se ha sintetizado la guía para la aplicación de la metodología propuesta a otras plataformas, indicando las dependencias existentes con estas.

El lector interesado dispone en el servidor Gitlab del CITSEM del código fuente de las implementaciones realizadas en la tesis [131, 133], así como de plantillas para la implementación del gestor de trabajos y de bloques huecos en nuevas plataformas [132].

7 RESULTADOS, APORTACIONES Y TRABAJO FUTURO

En este capítulo se resumen los resultados obtenidos en este trabajo de tesis, se listan las aportaciones de diversos tipos que se han generado durante su desarrollo (publicaciones, contribuciones a proyectos de investigación y dirección de trabajos de estudiantes) y se presentan posibles vías de continuación del trabajo realizado.

7.1 RESULTADOS

7.1.1 *Metodología de desarrollo para plataformas basadas en sistemas multiprocesador heterogéneos*

El principal resultado obtenido en este trabajo de tesis corresponde con el objetivo definido en el capítulo de introducción, sección 1.2: Elaborar una metodología de desarrollo para aplicaciones de radio *software* sobre plataformas multiprocesador heterogéneas aplicable a sistemas empotrados.

La metodología de desarrollo propuesta, resumida en el capítulo 6, cumple con este objetivo. Además, como beneficio adicional, contribuye a completar un flujo de diseño consistente para todo el proceso de desarrollo de sistemas de radiocomunicaciones, ya que hace posible emplear un mismo *software*, GNU Radio, tanto para tareas de alto nivel como el prototipado, simulación y evaluación de un sistema de comunicaciones completo, como para una implementación eficiente de los elementos de cualquier sistema de comunicaciones en una plataforma *hardware* especializada.

7.1.2 *Extensión de GNU Radio para sistemas multiprocesador heterogéneos*

La extensión de GNU Radio que se describe en el capítulo 4 permite ampliar el uso de este *software* más allá de los ordenadores personales para los que fue originalmente concebido, aumentando así el rango de escenarios en los que puede ser útil. Este resultado es comparable por su alcance a otras propuestas como RFNoC [134], un sistema que permite utilizar bloques de procesado de señal implementados en una FPGA desde GNU Radio.

7.2 APORTACIONES

7.2.1 *Publicaciones*

7.2.1.1 *Publicaciones en revistas indexadas en JCR*

- P.J. Lobo, E. Juárez, F. Pescador, C. Sanz. "Efficient Open Source Software Radio on Heterogeneous Multicore Embedded Platforms". *IEEE Consumer Electronics Magazine*, vol. 10, no. 2, pp. 27-36. Marzo 2021. DOI: 10.1109/MCE.2020.3010179 [135].

  En este artículo se presenta el resultado principal de esta tesis, la metodología de desarrollo para aplicaciones de radio *software* sobre plataformas multiprocesador heterogéneas, junto con los resultados obtenidos en la implementación del receptor DVB–T en la plataforma KeyStone II que han servido para validar la metodología propuesta.

- P.J. Lobo, E. Juárez, F. Pescador, G. Maturana, M.C. Rodríguez. "A DVB–H Receiver and Gateway Implementation on a FPGA- and DSP-based Platform". *IEEE Transactions on Consumer Electronics*, vol. 57, no. 2, pp. 372-378. Mayo 2011. DOI: 10.1109/TCE.2011.5955169 [96].

  En este artículo se presenta la implementación del receptor DVB–T/H en la plataforma *SFF SDR Development Platform* que se ha descrito en la sección 3.1.3.

7.2.1.2 *Publicaciones en congresos internacionales con revisión por pares*

- F. Pescador, P.J. Lobo, E. Seisdedos, E. Juárez, M.J. Garrido. "Real Time DVB–H Gateway Based on DSP". *IEEE International Conference on Consumer Electronics (ICCE 2011)*, pp. 749-750. Enero 2011 [136].

7.2.2 *Dirección de trabajos fin de estudios*

- David Mayorga Muñoz. "Implementación parcial de un receptor DVB–H basado en GNU Radio". Fecha de lectura: 2 de octubre de 2008. Calificación: 10 (SB).

- Francisco Montalvo Cárdenas. "Modelo en SystemC de un receptor de DVB–H basado en Radio Software". Fecha de lectura: 25 de noviembre de 2008. Calificación: 9,5 (SB).

- Diana Cañadas Martín. "Estudio comparativo de sistemas de comunicaciones inalámbricas para terminales móviles multiestándar". Fecha de lectura: 8 de julio de 2009. Calificación: 10 (SB).
- Miguel Ángel Torres Garrido. "Metodología para portado de código de GNU Radio a procesadores digitales de señal". Fecha de lectura: 31 de mayo de 2010. Calificación: 10 (MH).
- Enrique Velasco Segovia, Jesús Rodríguez García-Castro. "Evaluación de herramientas de síntesis automática para diseño basado en modelos". Fecha de lectura: 17 de julio de 2010. Calificación: 9,5 (SB).
- Daniel Sánchez Villalba. "Receptor DVB–H basado en GNU Radio". Fecha de lectura: 24 de septiembre de 2010. Calificación: 10 (MH).
- Antonio Vélez Felguera. "Portado de GNU Radio a los procesadores OMAP 3530 y DaVinci DM6446". Fecha de lectura: 28 de septiembre de 2010. Calificación: 9 (SB).
- Gonzalo Maturana Moreno. "Diseño de un receptor para sistemas DVB–T y DVB–H basado en tecnologías FPGA y DSP". Fecha de lectura: 28 de septiembre de 2011. Calificación: 10 (MH).
- Carlos Sánchez Martín. "Adaptación de GNU Radio a arquitecturas multiprocesador de punto fijo". Fecha de lectura: 12 de julio de 2012. Calificación: 10 (SB).
- Lisbel García Castellanos. "Portado del algoritmo de Viterbi a una arquitectura multiprocesador". Fecha de lectura: 18 de julio de 2013. Calificación: 8,5 (NT).
- Ernesto Alonso Gutiérrez. "Adaptación de GNU Radio a procesadores KeyStone II 66AK2xxx". Fecha de lectura: 14 de octubre de 2015. Calificación: 10 (SB).
- Miguel Ángel Pérez Segarra. "Modelado en Simulink de la sincronización en un receptor DVB–H". Fecha de lectura: 26 de septiembre de 2017. Calificación: 10 (MH).

7.3 TRABAJO FUTURO

Como es inevitable, el trabajo presentado en esta tesis no se encuentra en un estado final cerrado y existen muchas vías para continuarlo o extenderlo que podrían ser de interés. A continuación se enumeran algunas de las que se consideran más relevantes.

- Aplicar la metodología propuesta sobre plataformas *hardware* más recientes. La plataforma KeyStone II fue introducida por Texas Instruments en 2012 y aunque ha recibido algunas actualizaciones (el último SoC de la familia apareció en 2017) no parece que esté siendo activamente mantenida, especialmente en las versiones orientadas a alto rendimiento. Actualmente existe un movimiento muy activo en torno a arquitecturas basadas en la ISA abierta RISC-V [137], que podría ser interesante explorar. Un ejemplo representativo puede ser BlackParrot [138], una arquitectura multinúcleo abierta disponible bajo una licencia de tipo BSD[1] que contempla en su diseño el uso de distintos tipos de aceleradores *hardware*.

- Introducir soporte en GNU Radio para *buffers* de conexión entre bloques que no requieran copia. Tal como se ha visto en el capítulo 5, el intercambio de datos entre los GPPs y los aceleradores en las implementaciones realizadas requiere la copia de datos entre los *buffers* de memoria gestionados por GNU Radio y los accesibles por los aceleradores. La integración de estos últimos en el sistema de gestión de *buffers* de GNU Radio eliminaría la necesidad de estas copias, mejorando el rendimiento de las aplicaciones. Recientemente se están produciendo contribuciones a GNU Radio en esta área [139] que podrían facilitar esta tarea.

- Evaluar la adecuación de la metodología propuesta para soportar algoritmos de aprendizaje automático (*ML*, *Machine Learning*). Este tipo de algoritmos se han introducido a lo largo de los últimos años en prácticamente todas las áreas de aplicación del ámbito de las comunicaciones y el procesado de señal, y dentro del campo de la radio *software* se pueden encontrar propuestas de uso de SVMs (*Support Vector Machine*) o CNNs (*Convolutional Neural Network*) para determinar características de señales desconocidas, identificar categorías de usuarios o reconocer señales interferentes, por poner algunos ejemplos [140, 141].

1 Las licencias BSD (*Berkeley Software Distribution*) son una familia de licencias para distribución de *software* abierto y libre. La principal diferencia con las licencias GPL es que requieren únicamente el reconocimiento de uso y la atribución de la autoría, pero no la redistribución del código derivado.

BIBLIOGRAFÍA

[1] J. Mitola III. «The software radio architecture». En: *IEEE Communications Magazine* 33.5 (mayo de 1995), págs. 26-38. ISSN: 0163-6804. DOI: 10.1109/35.393001.

[2] R.I. Lackey y D.W. Upmal. «Speakeasy: the military software radio». En: *IEEE Communications Magazine* 33.5 (mayo de 1995), págs. 56-61. ISSN: 1558-1896. DOI: 10.1109/35.392998.

[3] John Dielissen et al. «Multistandard FEC Decoders for Wireless Devices». En: *IEEE Transactions on Circuits and Systems II: Express Briefs* 55.3 (mar. de 2008), págs. 284-288. ISSN: 1558-3791. DOI: 10.1109/TCSII.2008.918964.

[4] Zhenzhi Wu, Chen Gong y Dake Liu. «Computational Complexity Analysis of FEC Decoding on SDR Platforms». En: *Journal of Signal Processing Systems* 89.2 (nov. de 2017), págs. 209-224. ISSN: 1939-8115. DOI: 10.1007/s11265-016-1184-8.

[5] Rami Akeela y Behnam Dezfouli. «Software-defined Radios: Architecture, state-of-the-art, and challenges». En: *Computer Communications* 128 (sep. de 2018), págs. 106-125. DOI: 10.1016/j.comcom.2018.07.012.

[6] Gordon E. Moore. «Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.» En: *Proceedings of the IEEE* 86.1 (ene. de 1998), págs. 82-85. ISSN: 1558-2256. DOI: 10.1109/JPROC.1998.658762.

[7] Mark T. Bohr y Ian A. Young. «CMOS Scaling Trends and Beyond». En: *IEEE Micro* 37.6 (nov. de 2017), págs. 20-29. ISSN: 1937-4143. DOI: 10.1109/MM.2017.4241347.

[8] Karl Rupp. *Microprocessor Trend Data*. Ver. 3. 16 de jul. de 2020. URL: https://github.com/karlrupp/microprocessor-trend-data (visitado 24-02-2021).

[9] Philip Brisk. «Architecture and design automation for application-specific processors». En: *2011 9th IEEE International Conference on ASIC*. Oct. de 2011, págs. 1094-1097. DOI: 10.1109/ASICON.2011.6157399.

[10] C. A. R. A. Melo y E. Barros. «Oolong: A Baseband processor extension to the RISC-V ISA». En: *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. Jul. de 2016, págs. 241-242. DOI: 10.1109/ASAP.2016.7760808.

[11] M. Macedonia. «The GPU enters computing's mainstream». En: *Computer* 36.10 (oct. de 2003), págs. 106-108. ISSN: 1558-0814. DOI: 10.1109/MC.2003.1236476.

[12] Cyril Zeller et al. «Programming Graphics Hardware». En: *Eurographics 2004 - Tutorials*. Eurographics Association, 2004. DOI: 10.2312/egt.20041034.

[13] Saul Rosen. «Electronic Computers: A Historical Survey». En: *ACM Comput. Surv.* 1.1 (mar. de 1969), págs. 7-36. ISSN: 0360-0300. DOI: 10.1145/356540.356543.

[14] David Patterson. «The trouble with multi-core». En: *IEEE Spectrum* 47.7 (jul. de 2010), págs. 28-32, 53. ISSN: 1939-9340. DOI: 10.1109/MSPEC.2010.5491011.

[15] Hahn Kim y R. Bond. «Multicore software technologies». En: *Signal Processing Magazine, IEEE* 26.6 (nov. de 2009), págs. 80-89. ISSN: 1053-5888. DOI: 10.1109/MSP.2009.934141.

[16] J. Díaz, C. Muñoz-Caro y A. Niño. «A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era». En: *IEEE Transactions on Parallel and Distributed Systems* 23.8 (2012), págs. 1369-1386.

[17] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 5.0*. Nov. de 2018.

[18] Khronos OpenCL Working Group. *The OpenCL Specification, Version 2.2*. Mayo de 2017.

[19] Michael Piscopo. «Study on Implementing OpenCL in Common GNURadio Blocks». En: *Proceedings of the GNU Radio Conference*. Vol. 2. 1. 2017, pág. 67. URL: https://pubs.gnuradio.org/index.php/grcon/article/view/15.

[20] Pekka Jääskeläinen et al. «Exploiting Task Parallelism with OpenCL: A Case Study». En: *Journal of Signal Processing Systems* 91.1 (oct. de 2018), págs. 33-46. DOI: 10.1007/s11265-018-1416-1.

[21] *GNU Radio*. URL: https://www.gnuradio.org/ (visitado 25-11-2021).

[22] *Licencias GNU*. URL: https://www.gnu.org/licenses/ (visitado 12-10-2021).

[23] *The DVB Project*. https://www.dvb.org/. (Visitado 25-11-2021).

[24] *ISO/IEC 13818-2:2013 – Information technology – Generic coding of moving pictures and associated audio information – Part 2: Video*. 2013.

[25] *ISO/IEC 13818-1:2015 – Information technology – Generic coding of moving pictures and associated audio information – Part 1: Systems*. 2015.

[26] *DVB–T and T2 world adoption map*. https://www.dvb.org/resources/public/images/site/dvb-t_map.pdf. URL: https://web.archive.org/web/20160707081834/https://dvb.org/resources/public/images/site/dvb-t_map.pdf (visitado 25-11-2021).

[27] Ulrich Reimers. *DVB – The Family of International Standards for Digital Video Broadcasting*. 2nd. Signals and Communication Technology. Springer-Verlag Berlin Heidelberg, 2004. DOI: 10.1007/978-3-662-11577-0.

[28] *ETSI EN 300 744 v1.6.1: Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for digital terrestrial television*. Ene. de 2009.

[29] Jeff Foerster y John Liebetreu. *FEC Performance of Concatenated Reed-Solomon and Convolutional Coding with Interleaving*. Inf. téc. IEEE 802.16 Broadband Wireless Access Working Group, 2000.

[30] Todd K. Moon. *Error correction coding: mathematical methods and algorithms*. Wiley-Interscience, 2005. ISBN: 0-471-64800-0.

[31] G. D. Forney. «Burst-Correcting Codes for the Classic Bursty Channel». En: *Communication Technology, IEEE Transactions on* 19.5 (oct. de 1971), págs. 772-781. ISSN: 0018-9332. DOI: 10.1109/TCOM.1971.1090719.

[32] J. Mitola III. «Software radios-survey, critical evaluation and future directions». En: *National Telesystems Conference, 1992. NTC-92.* Mayo de 1992, págs. 13/15-13/23. DOI: 10.1109/NTC.1992.267870.

[33] Jeffrey Reed. *Software Radio: A Modern Approach to Radio Engineering*. Upper Saddle River, NJ, USA: Prentice Hall Press, 2002. ISBN: 0-13-081158-0.

[34] Eugene Grayver. *Implementing Software Defined Radio*. Springer-Verlag New York, 2013. ISBN: 978-1-4419-9331-1. DOI: 10.1007/978-1-4419-9332-8.

[35] K.V. Davis. «JTRS-an open, distributed-object computing software radio architecture». En: *Digital Avionics Systems Conference, 1999. Proceedings. 18th*. Vol. 2. 1999, 9.A.6-1-9.A.6-8. DOI: 10.1109/DASC.1999.863665.

[36] J. Place, D. Kerr y D. Schaefer. «Joint Tactical Radio System». En: *MILCOM 2000. 21st Century Military Communications Conference Proceedings*. Vol. 1. 2000, págs. 209-213. DOI: 10.1109/MILCOM.2000.904941.

[37] Vincent J. Kovarik y Raghavan Muralidharan. «Model-Based Systems Engineering: Lessons Learned from the Joint Tactical Radio System». En: *Journal of Signal Processing Systems* 89.1 (oct. de 2017), págs. 97-106. ISSN: 1939-8115. DOI: 10.1007/s11265-016-1218-2.

[38] S. Lucyszyn. «Review of radio frequency microelectromechanical systems technology». En: *Science, Measurement and Technology, IEE Proceedings* - 151.2 (mar. de 2004), págs. 93-103. ISSN: 1350-2344. DOI: 10.1049/ip-smt:20040405.

[39] C.T.-C. Nguyen. «MEMS-based RF channel selection for true software-defined cognitive radio and low-power sensor communications». En: *Communications Magazine, IEEE* 51.4 (abr. de 2013), págs. 110-119. ISSN: 0163-6804. DOI: 10.1109/MCOM.2013.6495769.

[40] J. Mitola III y G.Q. Maguire Jr. «Cognitive radio: making software radios more personal». En: *Personal Communications, IEEE* 6.4 (ago. de 1999), págs. 13-18. ISSN: 1070-9916. DOI: 10.1109/98.788210.

[41] Congzheng Han et al. «Green radio: radio techniques to enable energy-efficient wireless networks». En: *Communications Magazine, IEEE* 49.6 (jun. de 2011), págs. 46-54. ISSN: 0163-6804. DOI: 10.1109/MCOM.2011.5783984.

[42] M. Palkovic et al. «Future Software-Defined Radio Platforms and Mapping Flows». En: *Signal Processing Magazine, IEEE* 27.2 (mar. de 2010), págs. 22-33. ISSN: 1053-5888. DOI: 10.1109/MSP.2009.935386.

[43] Omer Anjum et al. «State of the art baseband DSP platforms for Software Defined Radio: A survey». En: *EURASIP Journal on Wireless Communications and Networking* 2011.1 (2011), pág. 5. ISSN: 1687-1499. DOI: 10.1186/1687-1499-2011-5.

[44] *TMS320C6201 Fixed-point Digital Signal Processor*. SPRS051H. Texas Instruments, Inc. Ene. de 1997. URL: http://www.ti.com/lit/ds/symlink/tms320c6201.pdf (visitado 25-11-2021).

[45] *TMS320C66x DSPCPU and Instruction Set Reference Guide*. SPRUGH7. Texas Instruments, Inc. Nov. de 2010.

[46] Yuan Lin et al. «SODA: A Low-power Architecture For Software Radio». En: *33rd International Symposium on Computer Architecture (ISCA'06)*. Ieee, 2006, págs. 89-101. ISBN: 0-7695-2608-X. DOI: 10.1109/ISCA.2006.37.

[47] Mark Woh et al. «From SODA to scotch: The evolution of a wireless baseband processor». En: *2008 41st IEEE/ACM International Symposium on Microarchitecture*. Ieee, nov. de 2008, págs. 152-163. ISBN: 978-1-4244-2836-6. DOI: 10.1109/MICRO.2008.4771787.

[48] Mark Woh et al. «AnySP: Anytime Anywhere Anyway Signal Processing». En: *IEEE Micro* 30.1 (ene. de 2010), págs. 81-91. ISSN: 0272-1732. DOI: 10.1109/MM.2010.8.

[49] D. Liu et al. «Bridging dream and reality: Programmable baseband processors for software-defined radio». En: *Communications Magazine, IEEE* 47.9 (sep. de 2009), págs. 134-140. ISSN: 0163-6804. DOI: 10.1109/MCOM.2009.5277467.

[50] A. Nilsson, E. Tell y D. Liu. «An 11 mm^2, 70 mW Fully Programmable Baseband Processor for Mobile WiMAX and DVB-T/H in 0.12 μm CMOS». En: *Solid-State Circuits, IEEE Journal of* 44.1 (ene. de 2009), págs. 90-97. ISSN: 0018-9200. DOI: 10.1109/JSSC.2008.2007167.

[51] Mayan Moudgill et al. «The Sandblaster 2.0 Architecture And SB3500 Implementation». En: *Proceedings of the Software Defined Radio Technical Forum (SDR Forum)*. 2008.

[52] Zhenyu Tu et al. «On the performance of 3GPP LTE baseband using SB3500». En: *System-on-Chip, 2009. SOC 2009. International Symposium on*. Oct. de 2009, págs. 138-142. DOI: 10.1109/SOCC.2009.5335659.

[53] Simon Knowles. «The SoC Future is Soft». En: *IEE Cambridge Processor Seminar*. Dic. de 2005.

[54] *The modem innovation inside NVIDIA i500 and Tegra 4i*. NVIDIA Whitepaper. 2013. URL: http://www.nvidia.com/docs/IO/116757/NVIDIA_i500_whitepaper_FINALv3.pdf (visitado 25-11-2021).

[55] Tilera Corp. *TILEPro64 Processor Product Brief*. 2011.

[56] Zhiyi Yu et al. «AsAP: An Asynchronous Array of Simple Processors». En: *IEEE Journal of Solid-State Circuits* 43.3 (mar. de 2008), págs. 695-705. ISSN: 0018-9200. DOI: 10.1109/JSSC.2007.916616.

[57] Dean N. Truong et al. «A 167-Processor Computational Platform in 65 nm CMOS». En: *IEEE Journal of Solid-State Circuits* 44.4 (abr. de 2009), págs. 1130-1144. ISSN: 0018-9200. DOI: 10.1109/JSSC.2009.2013772.

[58] Zhiyi Yu et al. «An asynchronous array of simple processors for dsp applications». En: *Solid-State Circuits Conference, 2006. ISSCC 2006. Digest of Technical Papers. IEEE International*. Feb. de 2006, págs. 1696-1705. DOI: 10.1109/ISSCC.2006.1696225.

[59] Anh Thien Tran, Dean Nguyen Truong y B. Baas. «A Reconfigurable Source-Synchronous On-Chip Network for GALS Many-Core Platforms». En: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 29.6 (jun. de 2010), págs. 897-910. ISSN: 0278-0070. DOI: 10.1109/TCAD.2010.2048594.

[60] B. Bohnenstiehl et al. «KiloCore: A 32-nm 1000-Processor Computational Array». En: *IEEE Journal of Solid-State Circuits* 52.4 (abr. de 2017), págs. 891-902. ISSN: 1558-173X. DOI: 10.1109/JSSC.2016.2638459.

[61] B. D. de Dinechin et al. «A clustered manycore processor architecture for embedded and accelerated applications». En: *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. Sep. de 2013, págs. 1-6. DOI: 10.1109/HPEC.2013.6670342.

[62] Russell Tessier y Wayne Burleson. «Reconfigurable Computing for Digital Signal Processing: A Survey». English. En: *Journal of VLSI signal processing systems for signal, image and video technology* 28.1-2 (2001), págs. 7-27. ISSN: 0922-5773. DOI: 10.1023/A:1008155020711.

[63] Paul Heysters, Gerard Smit y Egbert Molenkamp. «A Flexible and Energy-Efficient Coarse-Grained Reconfigurable Architecture for Mobile Systems». English. En: *The Journal of Supercomputing* 26.3 (2003), págs. 283-308. ISSN: 0920-8542. DOI: 10.1023/A:1025699015398.

[64] G. J. M. Smit y G. K. Rauwerda. «Reconfigurable Architectures for Adaptable Mobile Systems». En: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'05), Las Vegas, Nevada, USA*. Ed. por T. P. Plaks et al. Las Vegas, Nevada, USA: CSREA Press, jun. de 2005, págs. 17-25. ISBN: 1-932415-74-2.

[65] Gilles Kahn. «The semantics of a simple language for parallel programming». En: *In Information Processing'74: Proceedings of the IFIP Congress*. Vol. 74. 1974, págs. 471-475.

[66] E.A. Lee y T.M. Parks. «Dataflow process networks». En: *Proceedings of the IEEE* 83.5 (mayo de 1995), págs. 773-801. ISSN: 0018-9219. DOI: 10.1109/5.381846.

[67] Gerard K. Rauwerda, Paul M. Heysters y Gerard J. M. Smit. «Towards Software Defined Radios Using Coarse-Grained Reconfigurable Hardware». En: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16.1 (ene. de 2008), págs. 3-13. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2007.912075.

[68] B. Mei et al. «ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix». En: *Field-Programmable Logic and Applications 13th International Conference, FPL 2003 Lisbon, Portugal, September 1-3, 2003 Proceedings*. Lisbon: Springer Berlin / Heidelberg, 2003, págs. 61-70. ISBN: 0-7695-2085-5. DOI: 10.1007/b12007.

[69] B. Mei et al. «DRESC: a retargetable compiler for coarse-grained reconfigurable architectures». En: *IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings.* Ieee, 2002, págs. 166-173. ISBN: 0-7803-7574-2. DOI: 10.1109/FPT.2002.1188678.

[70] Pohua P. Chang et al. «IMPACT: An Architectural Framework for Multiple-instruction-issue Processors». En: *Proceedings of the 18th Annual International Symposium on Computer Architecture*. ISCA '91. Toronto, Ontario, Canada: ACM, 1991, págs. 266-275. ISBN: 0-89791-394-9. DOI: 10.1145/115952.115979.

[71] Bruno Bougard et al. «A Coarse-Grained Array based Baseband Processor for 100Mbps+ Software Defined Radio». En: *2008 Design, Automation and Test in Europe*. Ieee, mar. de 2008, págs. 716-721. ISBN: 978-3-9810801-3-1. DOI: 10.1109/DATE.2008.4484763.

[72] V. Derudder et al. «A 200Mbps+ 2.14nJ/b digital baseband multi processor system-on-chip for SDRs». En: *VLSI Circuits, 2009 Symposium on*. Jun. de 2009, págs. 292-293.

[73] G.J.M. Smit et al. «Lessons learned from designing the MONTIUM - a coarse-grained reconfigurable processing tile». En: *2004 International Symposium on System-on-Chip, 2004. Proceedings.* IEEE, 2004, págs. 29-32. ISBN: 0-7803-8558-6. DOI: 10.1109/ISSOC.2004.1411138.

[74] P. B. Nikishkin et al. «Sub-band OFDM Implementation on multicore DSP». En: *2016 24th Telecommunications Forum (TELFOR)*. Nov. de 2016, págs. 1-4. DOI: 10.1109/TELFOR.2016.7818818.

[75] M. Sever y E. Çavus. «Parallelizing LDPC Decoding Using OpenMP on Multicore Digital Signal Processors». En: *45th International Conference on Parallel Processing Workshops (ICPPW)*. Ago. de 2016, págs. 46-51. DOI: 10.1109/ICPPW.2016.22.

[76] A. Kharin et al. «LDPC decoder implementation on DSP+ARM platform with OpenCL». En: *2018 7th Mediterranean Conference on Embedded Computing (MECO)*. Jun. de 2018, págs. 1-4. DOI: 10.1109/MECO.2018.8406076.

[77] S.J. Pennycook et al. «An investigation of the performance portability of OpenCL». En: *Journal of Parallel and Distributed Computing* 73.11 (2013). Novel architectures for high-performance computing, págs. 1439-1450. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2012.07.005.

[78] Jacob Fainguelernt. *From MATLAB and Simulink to Real-Time Using TI DSP's*. OpenStax, 26 de oct. de 2012. URL: http://cnx.org/content/col10713/1.1/ (visitado 17-11-2015).

[79] J.-Y. Mignolet et al. «MPA: Parallelizing an Application onto a Multicore Platform Made Easy». En: *Micro, IEEE* 29.3 (mayo de 2009), págs. 31-39. ISSN: 0272-1732. DOI: 10.1109/MM.2009.46.

[80] Thomas W. Rondeau. «On the GNU Radio Ecosystem». En: *Opportunistic Spectrum Sharing and White Space Access*. John Wiley & Sons, Ltd, 2015. Cap. 2, págs. 25-48. ISBN: 9781119057246. DOI: 10.1002/9781119057246.ch2.

[81] DavidL. Tennenhouse y VanuG. Bose. «The SpectrumWare approach to wireless signal processing». English. En: *Wireless Networks* 2.1 (1996), págs. 1-12. ISSN: 1022-0038. DOI: 10.1007/BF01201458.

[82] Martin Braun. *Simulations with GNU Radio*. GNU Radio. 4 de oct. de 2012. URL: https://web.archive.org/web/20121004220439/https://gnuradio.org/redmine/projects/gnuradio/wiki/Simulations.

[83] Nick McCarthy et al. «High-Performance SDR: GNU Radio and the IBM Cell Broadband Engine». En: *Virginia Tech Wireless Personal Communications Symposium*. 2008.

[84] Eric A. Blossom. «gcell – An SPE Scheduler and Asynchronous RPC Mechanism for the Cell Broadband Engine». En: *SDR Forum Technical Conference*. Oct. de 2008.

[85] R.G. Machado y A.M. Wyglinski. «Software-Defined Radio: Bridging the Analog-Digital Divide». En: *Proceedings of the IEEE* 103.3 (mar. de 2015), págs. 409-423. ISSN: 0018-9219. DOI: 10.1109/JPROC.2015.2399173.

[86] *Small Form Factor SDR Evaluation Module / Development Platform User's Guide*. Lyrtech, Inc. Mar. de 2009.

[87] Heinrich Meyr, Marc Moeneclaey y Stefan Fechtel. *Digital Communication Receivers: Synchronization, Channel Estimation, and Signal Processing*. New York, NY, USA: John Wiley & Sons, Inc., 1997. ISBN: 0-471-50275-8.

[88] J.-J. Van De Beek, P B Orjesson y M Sandell. «ML estimation of time and frequency offset in OFDM systems». En: *IEEE Transactions on Signal Processing* 45.7 (1997), págs. 1800-1805. DOI: 10.1109/78.599949.

[89] Michael Speth et al. «Optimum Receiver Design for Wireless Broad-Band Systems Using OFDM — Part I». En: *IEEE Transactions on Communications* 47.11 (1999), págs. 1668-1677. DOI: 10.1109/26.803501.

[90] Enrique Velasco Segovia y Jesús Rodríguez García-Castro. «Evaluación de herramientas de síntesis automática para diseño basado en modelos». Proyecto Fin de Carrera. E. U. I. T. Telecomunicación, Universidad Politécnica de Madrid, jul. de 2010.

[91] A.J. Roscoe, S.M. Blair y Graeme M. Burt. «Benchmarking and optimisation of Simulink code using Real-Time Workshop and Embedded Coder for inverter and microgrid control applications». En: *Universities Power Engineering Conference (UPEC), 2009 Proceedings of the 44th International*. Sep. de 2009, págs. 1-5.

[92] Gonzalo Maturana Moreno. «Diseño de un receptor para sistemas DVB–T y DVB–H basado en tecnologías FPGA y DSP». Proyecto Fin de Carrera. E. T. S. I. Telecomunicación, Universidad Politécnica de Madrid, jul. de 2011.

[93] Todd Mullanix et al. *Reference Frameworks for eXpressDSP Software: RF5, An Extensive, High-Density System*. SPRA795A. Texas Instruments, Inc. Abr. de 2003. URL: http://www.ti.com/lit/an/spra795a/spra795a.pdf.

[94] *TMS320 DSP/BIOS v5.42 User's Guide*. SPRU423I. Texas Instruments, Inc. Texas Instruments, ago. de 2012. URL: http://www.ti.com/lit/ug/spru423i/spru423i.pdf.

[95] Alan Campbell et al. *Reference Frameworks for eXpressDSP Software: API Reference*. SPRA147C. Texas Instruments, Inc. Abr. de 2003. URL: http://www.ti.com/lit/an/spra147c/spra147c.pdf.

[96] P.J. Lobo et al. «A DVB–H receiver and gateway implementation on a FPGA– and DSP–based platform». En: *Consumer Electronics, IEEE Transactions on* 57.2 (mayo de 2011), págs. 372-378. ISSN: 0098-3063. DOI: 10.1109/TCE.2011.5955169.

[97] Daniel Sánchez Villalba. «Receptor DVB-H basado en GNU Radio». Proyecto Fin de Carrera. E. U. I. T. Telecomunicación, Universidad Politécnica de Madrid, sep. de 2010.

[98] *FFTW: Fastest Fourier Transform in the West*. URL: http://www.fftw.org/ (visitado 25-11-2021).

[99] Phil Karn. *DSP and FEC library version 3.0.1*. Ago. de 2007. URL: http://www.ka9q.net/code/fec/ (visitado 25-11-2021).

[100] A.J. Viterbi. «Error bounds for convolutional codes and an asymptotically optimum decoding algorithm». En: *Information Theory, IEEE Transactions on* 13.2 (abr. de 1967), págs. 260-269. ISSN: 0018-9448. DOI: 10.1109/TIT.1967.1054010.

[101] Luca Rose. «R-DVB: Software Defined Radio implementation of DVB-T signal detection functions for digital terrestrial television». Tesis de mtría. Scuola di Dottorato in Ingegneria "Leonardo da Vinci", Universitá di Pisa, 2009. URL: https://etd.adm.unipi.it/theses/available/etd-04032009-134746/unrestricted/R_dvb.pdf.

[102] V. Pellegrini et al. «On the Computation/Memory Trade-Off in Software Defined Radios». En: *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*. Dic. de 2010, págs. 1-5. DOI: 10.1109/GLOCOM.2010.5683494.

[103] J.A. Kahle et al. «Introduction to the Cell multiprocessor». En: *IBM Journal of Research and Development* 49.4.5 (jul. de 2005), págs. 589-604. ISSN: 0018-8646. DOI: 10.1147/rd.494.0589.

[104] *Listado de productos de Ettus Research (USRP, etc.)* http://www.ettus.com/product. URL: https://www.ettus.com/products/ (visitado 25-11-2021).

[105] *SWIG: Simplified Wrapper and Interface Generator*. URL: http://www.swig.org/ (visitado 25-11-2021).

[106] Tom Rondeau. *Explaining the GNU Radio scheduler*. Sep. de 2013. URL: http://www.trondeau.com/blog/2013/9/15/explaining-the-gnu-radio-scheduler.html (visitado 12-11-2018).

[107] *IEEE Std 1003.1-2008 - Standard for Information Technology - Portable Operating System Interface (POSIX®)*. IEEE Computer Society, 2008. DOI: 10.1109/IEEESTD.2008.4694976.

[108] Miguel Ángel Torres Garrido. «Metodología para portado de código de GNU Radio a procesadores digitales de señal». Proyecto Fin de Carrera. E. U. I. T. Telecomunicación, Universidad Politécnica de Madrid, mayo de 2010.

[109] Jan Krämer y Michael Schwall. *Embedded Linux on the Lyrtech SFF-SDR*. 2011. URL: https://www.cel.kit.edu/english/lyrtech.php (visitado 25-11-2021).

[110] *IGEP v2 Hardware Reference Manual*. ISEE. Feb. de 2013. URL: https://www.isee.biz/support/downloads/item/igepv2-hardware-reference-manual-2.

[111] *OMAP3530 and OMAP3525 Applications Processors*. SPRS507H. Texas Instruments, Inc. Feb. de 2008. URL: %5Curl%7Bhttp://www.ti.com/product/OMAP3530/technicaldocuments%7D.

[112] *The Ångström Linux Distribution*. http://www.angstrom-distribution.org/. URL: https://web.archive.org/web/20190510202610/http://angstrom-distribution.org/ (visitado 25-11-2021).

[113] *OpenEmbedded build framework*. URL: http://www.openembedded.org (visitado 25-11-2021).

[114] Carlos Sánchez Martín. «Adaptación de GNU Radio a arquitecturas multiprocesador de punto fijo». Proyecto Fin de Carrera. E. U. I. T. Telecomunicación, Universidad Politécnica de Madrid, jul. de 2012.

[115] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, oct. de 1994. ISBN: 0201633612.

[116] *Boost C++ Libraries*. URL: http://www.boost.org/ (visitado 25-11-2021).

[117] *Multicore DSP+ARM KeyStone II System-on-Chip (SoC)*. SPRS866E. Texas Instruments, Inc. Nov. de 2012. URL: http://www.ti.com/product/66AK2H14/technicaldocuments.

[118] *ARM CorePac User Guide*. SPRUHJ4. Texas Instruments, Inc. Oct. de 2012.

[119] *TMS320C66x DSP CorePac User Guide*. SPRUGW0C. Texas Instruments, Inc. Jul. de 2013.

[120] *Multicore Shared Memory Controller (MSMC) User Guide*. SPRUHJ6. Texas Instruments, Inc. Nov. de 2012.

[121] *The Yocto Project*. URL: http://www.yoctoproject.org/ (visitado 25-11-2021).

[122] *Processor SDK for 66AK2HX Processors*. URL: https://www.ti.com/tool/PROCESSOR-SDK-K2H (visitado 25-11-2021).

[123] Texas Instruments Inc. *Processor SDK IPC User's Guide*. 20 de abr. de 2020. URL: http://software-dl.ti.com/processor-sdk-rtos/esd/docs/latest/rtos/index_Foundational_Components.html#ipc.

[124] Texas Instruments Inc. *MsgCom API*. Archived on June 2017. Texas Instruments. 25 de jun. de 2013. URL: https://web.archive.org/web/20170614021630/http://processors.wiki.ti.com/index.php/Msgcom (visitado 14-06-2017).

[125] *Problem with ARM-DSP communication (66K2H14,Msgcom)*. Texas Instruments E2E Support Forum. 14 de abr. de 2015. URL: https://e2e.ti.com/support/processors/f/processors-forum/416055/problem-with-arm-dsp-communication-66k2h14-msgcom.

[126] Douglas S. Lea. *Doug Lea's Malloc*. ftp://g.oswego.edu/pub/misc. 29 de ago. de 2012. URL: https://web.archive.org/web/20190520205604/ftp://g.oswego.edu/pub/misc (visitado 20-05-2019).

[127] Giuseppe Baruffa. *gbDVB DVB–T simulation tools v3.4*. DSP Laboratory, Universitá degli Studi di Perugia, 6 de jun. de 2009. URL: http://dsplab.diei.unipg.it/software/gbdvb (visitado 20-04-2021).

[128] Chip Fleming. *A Tutorial on Convolutional Coding with Viterbi Decoding*. Spectrum Applications. 5 de jul. de 2002. URL: http://linas.org/mirrors/home.netcom.com/2002.09.18/~chip.f/viterbi/tutorial.html (visitado 28-04-2020).

[129] G. Fettweis y H. Meyr. «Feedforward Architectures for Parallel Viterbi Decoding». En: *Journal of VLSI signal processing systems for signal, image and video technology* 3 (jun. de 1991), págs. 105-119. ISSN: 0922-5773. DOI: 10.1007/BF00927838.

[130] *C66X-FFTLIB: FFT Library for C66x Floating Point Devices*. Ver. 2.0.0.2. Texas Instruments, Inc., 12 de mar. de 2014. URL: https://www.ti.com/tool/FFTLIB (visitado 28-04-2020).

[131] Pedro J. Lobo. *GNU Radio extension for KeyStone II processors*. 16 de sep. de 2021. URL: https://gitlab.citsem.upm.es/pedro.lobo/gr-ks2.

[132] Pedro J. Lobo. *GNU Radio extension, reference design for a Generic Accelerator Platform*. 16 de sep. de 2021. URL: https://gitlab.citsem.upm.es/pedro.lobo/gr-gap.

[133] Pedro J. Lobo. *GNU Radio extension for the OMAP 3530 processor*. 16 de sep. de 2021. URL: https://gitlab.citsem.upm.es/pedro.lobo/gr-c6x.

[134] Martin Braun, Jonathan Pendlum y Matt Ettus. «RFNoC: RF Network-on-Chip». En: vol. 1. 1. 2016. URL: https://pubs.gnuradio.org/index.php/grcon/article/view/3.

[135] Pedro J. Lobo et al. «Efficient Open Source Software Radio on Heterogeneous Multicore Embedded Platforms». En: *IEEE Consumer Electronics Magazine* 10.2 (mar. de 2021), págs. 27-36. ISSN: 2162-2256. DOI: 10.1109/MCE.2020.3010179.

[136] F. Pescador et al. «Real time DVB-H gateway based on DSP». En: *2011 IEEE International Conference on Consumer Electronics (ICCE'11)*. Ene. de 2011, págs. 749-750. DOI: 10.1109/ICCE.2011.5722843.

[137] Andrew Waterman y Krste Asanović, eds. *The RISC-V Instruction Set Manual*. 2019. URL: https://riscv.org/technical/specifications/ (visitado 25-11-2021).

[138] Daniel Petrisko et al. «BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs». En: *IEEE Micro* 40.4 (jul. de 2020), págs. 93-102. ISSN: 1937-4143. DOI: 10.1109/MM.2020.2996145.

[139] David Sorber. *Improving GNU Radio Accelerator Device Dataflow*. FOSDEM 2021. 7 de feb. de 2021. URL: https://archive.fosdem.org/2021/schedule/event/fsr_improving_gnu_radio_accelerator_device_dataflow/ (visitado 26-11-2021).

[140] Shamnaz Riyaz et al. «Deep Learning Convolutional Neural Networks for Radio Identification». En: *IEEE Communications Magazine* 56.9 (sep. de 2018), págs. 146-152. DOI: 10.1109/mcom.2018.1800153.

[141] Muhammad Sajjad Khan et al. «Support Vector Machine-Based Classification of Malicious Users in Cognitive Radio Networks». En: *Wireless Communications and Mobile Computing* 2020 (jul. de 2020), págs. 1-11. DOI: 10.1155/2020/8846948.

LISTA DE ACRÓNIMOS

A/D Analog to Digital.
ALU Arithmetic Logic Unit.
API Application Programming Interface.
ASIC Application Specific Integrated Circuit.
ASIP Application Specific Instruction-set Processor.
ATSC Advanced Television Systems Committee.

BPSK Binary Phase Shift Keying.

CFO Carrier Frequency Offset.
CGRA Coarse Grain Reconfigurable Architecture.
CITSEM Centro de Investigación en Tecnologías Software y Sistemas Multimedia para la Sostenibilidad.
CNN Convolutional Neural Network.
CPU Central Processing Unit.
CSoC Configurable System-on-Chip.

D/A Digital to Analog.
DAB Digital Audio Broadcasting.
DLP Data Level Parallelism.
DSP Digital Signal Processor.
DTMB Digital Terrestrial Multimedia Broadcast.
DVB Digital Video Broadcasting.

ETSI European Telecommunications Standards Institute.

FFT Fast Fourier Transform.
FIFO First In First Out.
FIR Finite Impulse Response.
FPGA Field Programmable Gate Array.

GNU GNU's Not UNIX.
GPGPU General Purpose computing on Graphical Processing Units.
GPL GNU General Public License.
GPP General Purpose Processor.
GPU Graphics Processing Unit.

HDL Hardware Description Language.
HPC High Performance Computing.

IIR Infinite Impulse Response.
ILP Instruction Level Parallelism.
ISDB Integrated Services Digital Broadcasting.
ISO International Organization for Standardization.

JTRS Joint Tactical Radio System.

LDPC Low Density Parity Code.
LTE Long-Term Evolution.
LUT Look-Up Table.

MBD Model Based Design.
MBDK Model Based Design Kit.
MEMS Microelectromechanical Systems.
ML Machine Learning.
MPEG Moving Picture Experts Group.
MPPA Massively Parallel Processor Array.
MSMC Multicore Shared Memory Controller.

NCO Numerically Controlled Oscilator.
NoC Network-on-Chip.

OFDM Orthogonal Frequency Division Multiplexing.
OMAP Open Multimedia Applications Platform.

PC Personal Computer.
POSIX Portable Operating System Interface.
PSK Phase Shift Keying.

QAM Quadrature Amplitude Modulation.
QPSK Quad Phase Shift Keying.

RTOS Real-Time Operating System.

SCA Software Communications Architecture.
SDK Software Development Kit.
SDR Software Defined Radio.
SFN Single-Frequency Network.
SFO Sampling Frequency Offset.

SIMD Single Instruction Multiple Data.
SMP Symmetric Multiprocessing.
SoC System-on-Chip.
STO Symbol Timing Offset.
SVM Support Vector Machine.

TLP Task Level Parallelism.
TPS Transmission Parameter Signalling.
TS Transport Stream.

UPM Universidad Politécnica de Madrid.
USRP Universal Software Radio Peripheral.

VHDL VHSIC Hardware Description Language.
VHSIC Very High Speed Itegrated Circuit.
VLIW Very Long Instruction Word.

WiMAX Wordwide Ineroperability for Microwave Access.