UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

PROYECTO FIN DE CARRERA

ACELERADOR HARDWARE PARA EL PROCESADO DE EVENTOS EN TIEMPO REAL (AHPETIR)

Autor: Omar Espino Santana

Tutores: Pedro Pérez Carballo

Pedro Hernández Fernández

Rosalva Carrascosa González

Titulación: Ingeniero de Telecomunicación

Fecha: Mayo 2012

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

PROYECTO FIN DE CARRERA

ACELERADOR HARDWARE PARA EL PROCESADO DE EVENTOS EN TIEMPO REAL (AHPETIR)

HOJA DE FIRMAS

Alumno:

Fdo.: Omar Espino Santana

Tutor: **Tutor:** **Tutora:**

Fdo.: Pedro Pérez Carballo Fdo.: Pedro Hernández Fernández Fdo.: Rosalva Carrascosa González

Titulación: Ingeniero de Telecomunicación

Fecha: Mayo 2012

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

PROYECTO FIN DE CARRERA

ACELERADOR HARDWARE PARA EL PROCESADO DE EVENTOS EN TIEMPO REAL (AHPETIR)

HOJA DE EVALUACIÓN

CALIFICACIÓN:

Presidente:

Fdo.:

Vocal:	**Secretario:**
Fdo.:	Fdo.:

Titulación: Ingeniero de Telecomunicación

Fecha: ……

Agradecimientos

A mis padres, por toda la confianza que siempre han depositado en mí y apoyarme en todas y cada una de las decisiones que he tenido que tomar.

A mi hermano, por todos esos buenos ratos que he compartido con él. Por todas las bromas que nos hemos gastado y nos seguiremos gastando.

A mi tío Tino, por preocuparse más que yo de cada examen que he tenido. Por alegrarse con cada uno de mis éxitos y estar siempre atento.

A Paloma, por ser mi compañera de estudios, prácticas, trabajos, y casi cualquier tarea que hemos tenido que hacer a lo largo de la carrera. Por ser también mi amiga fuera de la escuela y prestarme su ayuda siempre que la he necesitado.

A Cristina y a Ester, por todos esos momentos que hemos pasado, ya sea dentro de la universidad estudiando para algún examen, como fuera olvidándonos todo lo posible de las mil y una tareas que nos queda por hacer.

A Cassandra, Dani y Raquel, por ser grandes compañeros y grandes amigos. A ellos, y a otros tantos colegas, que han hecho que estos años hayan sido más amenos y divertidos, y más llevaderos en muchos casos.

A Adrián y a Rita, por haber sido grandes compañeros de equipo y ayudarme siempre que lo necesité.

A Pedro P. Carballo, por ser mi tutor, tanto en este proyecto, como en otros tantos que han surgido. Por aconsejarme siempre que lo he requerido y prestarme siempre su ayuda.

A Pedro Hernández, por estar siempre dispuesto a responder mis dudas, y ayudarme en tantas tareas durante este proyecto.

A los que he nombardo, y a los que me he dejado, muchas gracias.

Índice de contenido

Índice de figuras

Índice de tablas

Resumen

Actualmente existe un conjunto de sistemas con requerimientos temporales específicos conocidos como Sistemas en Tiempo Real. Estos requieren que su ejecución esté acotada en el tiempo, por lo que son objeto de optimización contínua con el fin de mejorar sus prestaciones.

El objetivo principal de este proyecto es la aceleración de una aplicación de procesado de eventos en tiempo real, haciendo uso para ello de flujos de diseño de circuitos integrados. La arquitectura propuesta la compondrá un System-on-Chip debido a que abordar el diseño del mismo mediante un flujo estándar de diseño es ineficiente para sistemas tan complejos.

Con este objetivo se realiza un análisis de los procesadores de eventos, destacando las necesidades de estos, en especial en el ámbito de las comunicaciones y de la arquitectura de los mismos.

En base a las necesidades de la aplicación de referencia se presenta la arquitectura del sistema propuesto, así como el flujo de desarrollo del mismo. Al tratarse de un sistema empotrado, se definen dos líneas de desarrollo, por un lado identificando el hardware del sistema y por otro lado el software de la aplicación empotrada a ser ejecutada en el microprocesador del SoC.

En este proyecto se lleva a cabo el flujo de manera completa, implementando finalmente el diseño sobre dispositivo físico, realizando depuración sobre este y midiendo el rendimiento del mismo. También se analiza dónde se encuentra el cuello de botella.

La tecnología de implementación será la de dispositivos FPGA debido a las ventajas que estas ofrecen en las etapas de prototipado del diseño. Además, en la actualidad, ciertas familias de FPGA ofrecen recursos físicos que ayudan al desarrollo de sistemas empotrados, al incluir microprocesadores RISC embebidos en el mismo chip donde se encuentra el resto de la lógica programable.

Este proyecto se desarrolla dentro de las actividades del proyecto RT-CORE realizado conjuntamente por el Instituto Universitario de Microelectrónica Aplicada (IUMA) y la empresa Edosoft Factory S. L.

Abstract

Currently there are a set of systems with specific time requirements known as Real-Time Systems. They require its execution to be bounded in time, so they are target of continuous optimization to improve its performance.

This project is focused on the acceleration of a real-time event processor application, using integrated circuits design flows. The proposed architecture consists in a System-on-Chip since using a standard design flow is not efficient enough for complex designs.

With this aim, an analysis of event processors is performed, highlighting its needs, specifically in the communications field and its architectures.

Based on the reference application needs, the architecture of the propounded system is presented, as its development flow. Since it is an embedded system, the work has to be divided into two main development lines: the hardware partition and the software of the embedded application of the SoC microprocessor.

This project ends with the implementation of the design in the physical device, measuring its performance and analyzing where its bottleneck is.

The implementation technology will be FPGA devices due to the innumerable advantages they offer at prototyping stages. Futhermore, nowadays, some FPGA families include RISC microprocessors embedded in the same chip where the programmable resources are, making them appropriate for embedded system designs.

This project is born in Institute of Applied Microelectronics (IUMA) and Edosoft Factory S.L. project: RT-CORE.

Acrónimos

ABS	*Anti-lock Braking System*
API	*Application Programming Interface*
APU	*Auxiliary Processor Unit*
ARM	*Advanced RISC Machine*
ARP	*Address Resolution Protocol*
ASIC	*Application-Specific Integrated Circuit*
ASMBL	*Advanced Silicon Modular Block*
ATX	*Advanced Technology Extended*
AXI	*Advanced Extensible Interface*
BRAM	*Block RAM*
BSB	*Base System Builder*
BSD	*Berkeley Software Distribution*
CAD	*Computer-Aided Design*
CAS	*Column Address Stroble*
CE	Coprocesador de Eventos
CLB	*Configurable Logic Block*
CMT	*Clock Management Tiles*
CPU	*Central Processing Unit*
CtoS	*C-to-Silicon Compiler*
DCI	*Digitally Controlled Impedance*
DCM	*Digital Clock Manager*
DCR	*Device Control Register*
DDR	*Double Data Rate*
DHCP	*Dynamic Host Configuration Protocol*
DIMM	*Dual In-line Memory Module*
DMA	*Direct Memory Access*
DNS	*Domain Name System*
DSP	*Digital Signal Processor*
DVI	*Digital Visual Interface*
EEPROM	*Electrically Erasable Programmable Read-Only Memory*
EDIF	*Electronic Data Interchange Format*
EDK	*Embedded Development Kit*
ELF	*Executable and Linkable Format*
EMAC	*Ethernet MAC*
EOF	*End Of Frame*
EOP	*End Of Payload*
EPR	Elementos de Procesamiento Reconfigurables
ESL	*Electronic System Level*
FIFO	*First In, First Out*
FPGA	*Field Programmable Gate Array*
FPU	*Floating-Point Unit*
FSM	*Finite-State Machine*
GMII	*Gigabit Media Independent Interface*
GPGPU	*General Purpose computing on Graphics Processing Unit*
GPIO	*General Purpose Input/Output*
GPP	*General Purpose Processor*

GPU	*Graphics Processing Unit*
HAL	*Hardware Abstraction Layer*
HDL	*Hardware Description Language*
HID	*Human Interface Device*
HTTP	*HyperText Transfer Protocol*
HTTPS	*HyperText Transfer Protocol Secure*
HW	*Hardware*
ICMP	*Internet Control Message Protocol*
ICON	*Integrated Controller*
IEEE	*Institute of Electrical and Electronics Engineers*
IGMP	*Internet Group Management Protocol*
IIC	*Inter-Integrated Circuit*
ILA	*Integrated Logic Analyzer*
INTC	*Interrupt Controller*
IOB	*In/Out Block*
IP	*Intellectual Property*
IP	*Internet Protocol*
IRQ	*Interrupt Request*
ISDB	*Integrated Signal Data Base*
IUMA	Instituto Universitario de Microelectrónica Aplicada
JTAG	*Join Test Action Group*
LCD	*Liquid Crystal Display*
LED	*Light-Emitting Diode*
LL	*LocalLink*
LUT	*LookUp Table*
lwIP	*lightweight IP*
MAC	*Media Access Control*
MCI	*Memory Controller Interface*
MHS	*Microprocessor Hardware Specification*
MII	*Media Independent Interface*
MMCM	*Mixed-Mode Clock Manager*
MPD	*Microprocessor Peripheral Definition*
MSS	*Microprocessor Software Specification*
NGD	*Native Generic Database*
NoC	*Network-on-Chip*
ODT	*On-Die Termination*
PCB	*Printed Circuit Board*
PCI	*Peripheral Component Interconnect*
PFC	Proyecto Fin de Carrera
PHY	*Physical Layer*
PID	*Process Identifier*
PLB	*Processor Local Bus*
PLL	*Phase-Locked Loop*
POSIX	*Portable Operating System Interface*
PS2	*Personal System/2*
RAM	*Random Access Memory*
RGMII	*Reduced Gigabit Media Independent Interface*
RISC	*Reduced Instruction Set Computing*
ROM	*Read-Only Memory*
RTL	*Register Transfer Level*
SATA	*Serial Advanced Technology Attachment*
SGMII	*Serial Gigabit Media Independent Interface*

SIMD	*Single Instruction, Multiple Data*
SMBus	*System Management Bus*
SoC	*System-on-Chip*
SOF	*Start Of Frame*
SOP	*Start Of Payload*
SPI	*Serial Peripheral Interface*
SPLB	*Slave Processor Local Bus*
SRAM	*Static Random Access Memory*
STR	Sistemas en Tiempo Real
SW	*Software*
TCL	*Tool Command Language*
TCP	*Transmission Control Protocol*
TEMAC	*Tri-mode Ethernet MAC*
TIC	Tecnología de la Información y la Comunicación
TLM	*Transaction-Level Modeling*
TRCE	*Timing Reporter and Circuit Evaluator*
UART	*Universal Asynchronous Receiver/Transmitter*
UCF	*User Constraint File*
UDP	*User Datagram Protocol*
USB	*Universal Serial Bus*
VCD	*Value Change Dump*
VHDL	*Very High Speed Integrated Circuit Hardware Description Language*
VLAN	*Virtual Local Area Network*
WIF	*Waveform Interchange Format*
XMD	*Xilinx Microprocessor Debugger*
XPM	*Cross-Phase Modulation*
XPS	*Xilinx Platform Studio*
XST	*Xilinx Synthesis Technology*

Capítulo 1: Introducción

1 Antecedentes

Existen multitud de aplicaciones que han de ser ejecutadas bajo restricciones temporales. Los sistemas controlados por estas aplicaciones se conocen como Sistemas en Tiempo Real (STR). El elemento condicionante suele ser el tiempo máximo de ejecución de un algoritmo que permita tiempos de respuesta del sistema lo más bajos posibles y siempre por debajo de un límite impuesto por el entorno de la aplicación.

La aceleración *hardware* es una posible solución a las restricciones anteriormente indicadas. Este enfoque implica la realización de una partición del código de la aplicación *software* a ejecutar, separándola en módulos, y diseñar aceleradores *hardware* para procesar aquellos módulos críticos que representen un mayor coste computacional [1].

Las FPGAs (*Field Programmable Gate Array*) representan un recurso adecuado a utilizar para realizar los bloques descritos. Por su facilidad a la hora de reprogramar y su bajo coste para un

bajo número de unidades, serán el principal medio a usar. Este Proyecto Fin de Carrera tiene su mayor aplicación en aquellas situaciones en las que sea necesario reducir al máximo la latencia de ejecución para así aumentar la cantidad de eventos a procesar en un tiempo dado [2].

1.1 Sistemas en Tiempo Real (STR)

Un sistema en tiempo real (STR) es aquel sistema digital que interactúa activamente con su entorno con una dinámica conocida entre sus entradas, salidas y restricciones temporales, para darle un correcto funcionamiento de acuerdo con los conceptos de predictibilidad, estabilidad, controlabilidad y alcanzabilidad [3].

El correcto funcionamiento de estos sistemas depende no sólo del resultado lógico que se genere, sino que también depende del tiempo utilizado en producir dicho resultado. De esta forma, atendiendo al entorno en el que se desarrolle la aplicación, existirá un tiempo límite que el sistema deberá cumplir para que sea exitoso, considerándose todos aquellos resultados obtenidos después del límite como fallos del sistema aún cuando estos sean los valores esperados. Se puede hablar de dos tipos de STR: *Hard* RT y *Soft* RT. Para el primero de los casos la restricción es completa, con tiempos de respuesta estrictos. Para el segundo, los tiempos de procesamiento pueden estar comprendidos en un determino margen pero no son críticos cuando la salida sigue una determinada cadencia, aunque con cierta latencia inicial.

Un ejemplo típico de sistema en tiempo real tipo *Hard* RT es el antibloqueo de las ruedas de un automóvil (ABS). En este caso el tiempo límite en el que debe operar es aquel en el que las ruedas deban liberarse antes de que se bloqueen. Si el sistema libera las ruedas una vez estas ya se han bloqueado, habrá fallado[4]. Un ejemplo de un sistema *Soft* RT puede ser un sistema de transmisión de vídeo.

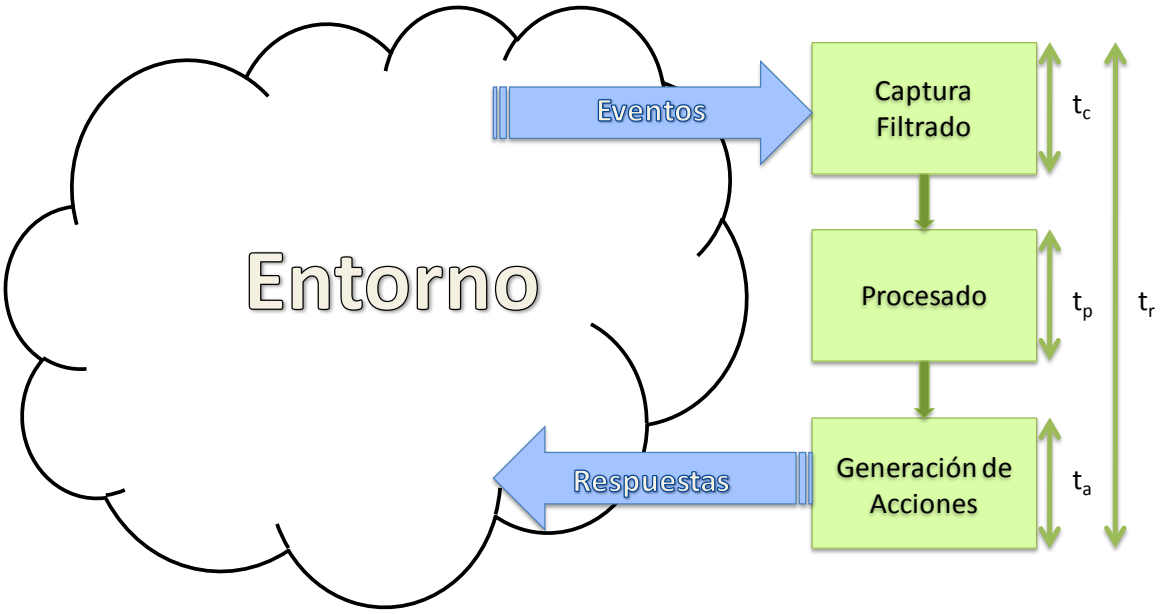


Figura 1. Esquema de un procesador de eventos.

Los sistemas de procesado de eventos son ejemplos típicos de STR. En ellos existe una tasa de llegada de eventos los cuales deben ser procesados en un tiempo límite. En general existirán unas condiciones, que en conjunto con los eventos, serán los que decidan qué acción deberá realizar. El procesado que llevará a cabo el sistema será comprobar si cada evento de entrada cumple o no, algunas de las condiciones impuestas con anterioridad y decidir, en base a ello, la acción a desempeñar. Un ejemplo son los sistemas automatizados de compra y venta en el mercado de valores. Estos sistemas deben decidir si realizar una acción en función de los cambios en los precios de los productos y de unas estrategias previamente establecidas [5].

El funcionamiento de los sistemas de procesado de eventos se puede dividir en las siguientes tareas: captura y filtrado de eventos, procesado de los eventos y generación de las acciones. En la captura y filtrado, se requerirá que el sistema obtenga los eventos que se producen en su entorno y filtre aquellos parámetros que sean de utilidad para su posterior procesado, descartando aquella información sobrante que pueda sobrecargar innecesariamente el resto de tareas. En el procesado se realizarán las comparaciones que se hayan definido con el fin de comprobar si se cumple una o varias condiciones previamente establecidas. Por último el generador de acciones será el encargado de enviar información sobre qué acciones se deben llevar a cabo en función de las condiciones cumplidas según el procesador.

Con el fin de no perder ningún evento, será necesario que la suma de los tiempos dedicados a cada una de las tareas no supere el tiempo límite, t_r, que vendrá determinado por la cantidad de eventos por segundo que llegan al sistema.

$$t_{captura} + t_{procesado} + t_{generación\ acción} < t_r$$

Debido precisamente a esta última condición es por la cual los sistemas de procesado de eventos se encuentran dentro de los sistemas en tiempo real.

1.2 Soluciones tecnológicas

Los sistemas de procesado de eventos con los que se trabaja en este Proyecto Fin de Carrera pueden ser ejecutados sobre diferentes soluciones arquitecturales, teniendo cada una de ellas unas características determinadas.

Una primera posibilidad es la ejecución de la aplicación sobre un procesador de propósito general (GPP). Este tipo de dispositivos están preparados para poder realizar cualquier tarea, mediante un conjunto de instrucciones que ejecutadas en un orden específico permiten una gran variedad de posibilidades. Su principal ventaja es la flexibilidad. Sin embargo tiene la desventaja de ser el método poco eficiente.

En el lado opuesto se encuentran los circuitos integrados para aplicaciones específicas (ASIC). Al contrario que los GPP, se tratan de soluciones diseñadas de forma optimizada para una aplicación concreta, restringiendo su programabilidad pero permitiendo mejor eficiencia. Otra característica fundamental de este método es la posibilidad de concurrencia real entre tareas, al incluir en el circuito integrado diferentes módulos que realizan distintas operaciones en el mismo instante[6].

En cuanto a la programación, para las aplicaciones a ejecutar sobre un GPP se pueden usar lenguajes de alto nivel como Java, Ada, Basic, C++, etc. El código generado será compilado para transformarlo a lenguaje máquina, es decir, un conjunto de instrucciones entendibles por el procesador.

En los ASIC, el flujo de desarrollo tiene otra filosofía. Para el diseño de estos sistemas se usan normalmente lenguajes de descripción hardware (HDL) tales como Verilog o VHDL. Los bloques descritos en estos lenguajes son posteriormente sintetizados haciendo uso de las librerías tecnológicas de los fabricantes. En la actualidad se ha incrementado el nivel de abstracción hasta usar lenguajes de descripción de sistemas como puede ser C/C++ o SystemC [7], que tras una síntesis de alto nivel, transforma la descripción algorítmica en una microarquitectura a nivel RTL que luego es optimizada e implementada siguiendo flujos tradicionales de síntesis [8, 9].

Mientras que en el dominio *software*, es decir, aplicaciones para ser ejecutadas sobre un GPP, se puede compilar, modificar y recompilar el código, permitiendo así optimizar el diseño tras varias iteraciones, en el diseño de un ASIC, el coste de la fabricación, tanto temporal como de capital, impide que esta sea una alternativa de diseño. Es necesario que el ASIC funcione correctamente desde el principio. Sin embargo esto requiere de un proceso de verificación muy costoso, por lo que el uso de FPGAs, dispositivos semiconductores que contienen bloques de lógica donde su funcionalidad e interconexión son programables, facilita las tareas iniciales de prototipado, llegando incluso a utilizarse directamente para pequeñas tiradas tal como se indicó anteriormente. Las FPGAs además poseen recursos de memoria internos, en algunos casos núcleos procesadores y otros bloques funcionales que facilitan la implementación de un sistema electrónico en chip (SOC) [2].

Se puede concluir que las principales ventajas en el uso de FPGAs frente a otras soluciones de diseño electrónico son su bajo coste durante las fases de desarrollo del sistema, lo que permite la producción de un reducido número de unidades (frente al coste de fabricar un ASIC) y su sencilla reprogramabilidad. Esta última propiedad permite una optimización post-diseño [10]. Es aquí donde esta tecnología tiene una ventaja competitiva para este tipo de problemas de procesado de eventos.

Los kits de diseño que proporcionan los fabricantes incluyen placas sobre las que vienen interconectadas la FPGA con diversos recursos. Muchos de estos incluyen bloques de interfaz como pueden ser USB, Ethernet, PS2, etc. Además, también incluyen sus propias fuentes de reloj, bloques de memoria e incluso CPUs.

Otras arquitecturas posibles son el uso de GPUs (*Graphics Processing Unit*) gracias a su potencia de cómputo. Aunque en el proceso de renderizado llevado a cabo por las GPUs existen claramente dos etapas diferenciadas (procesado de vértices y procesado de píxeles) con necesidades computacionales distintas, en los últimos modelos se ha optado por arquitecturas donde los elementos de procesado son comunes para ambas tareas. Esto último, sumado a los entornos de desarrollo proporcionados por los fabricantes, hacen que la GPU sea cada vez más flexible, permitiendo su uso para la ejecución de aplicaciones no vinculadas al renderizado (GPGPU – *General Purpose Computing on GPU*). Aunque su programación no es tan flexible como la de las CPUs de propósito general, su gran cantidad de elementos de cómputo, junto a su arquitectura optimizada para la concurrencia mediante SIMD (*Single Instruction Multiple Data*)

puede ser muy útil en el procesamiento de eventos, cuando quiere compararse una gran cantidad de ellos en una misma condición.

Por otro lado se encuentran los DSPs (*Digital Signal Processor*), que son procesadores optimizados para el procesado de señales, siendo normalmente especializados en operaciones del tipo MAC (Multiplicado y Acumulación). Este tipo de procesadores son más eficientes que las CPUs o las GPUs, pero mucho menos flexible en su programación, siendo muchas veces necesario programarlos en lenguaje ensamblador. En la figura 2 se comparan las diferentes soluciones indicadas teniendo en cuenta su eficiencia frente a su flexibilidad. En este Proyecto Fin de Carrera, como se verá más adelante, se trata de aumentar la eficiencia del sistema, moviendo funciones del dominio *software*, por tanto ejecutadas en un GPP, más flexibles, a su implementación en el dominio *hardware*, más eficientes pero menos flexibles. Con objeto de mantener una cierta flexibilidad en el desarrollo de estas funciones, se emplean técnicas de diseño basadas en la síntesis de alto nivel [11].

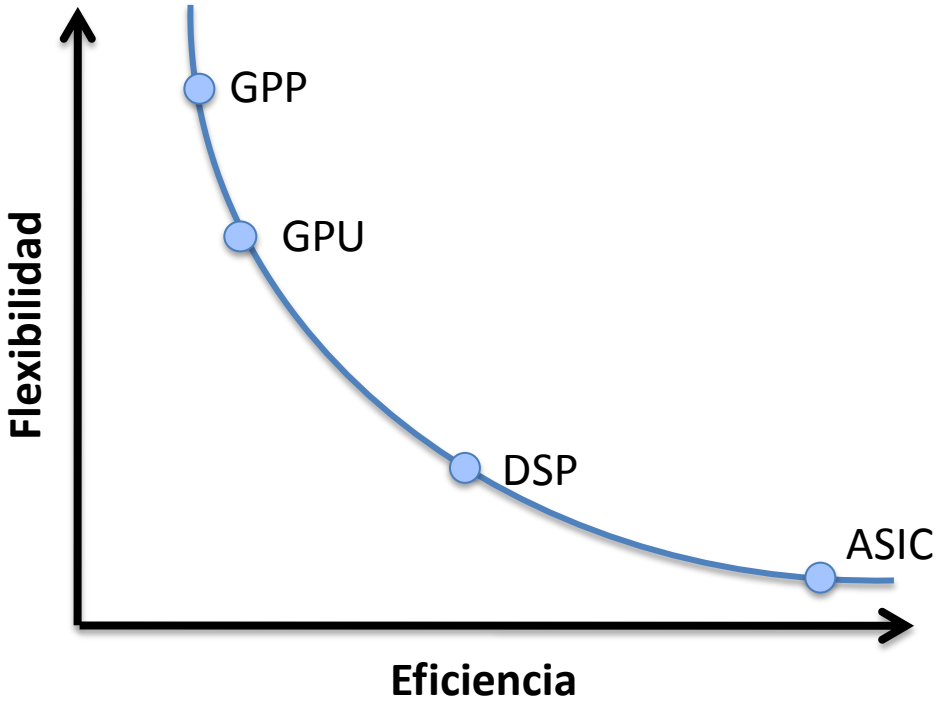


Figura 2. Comparativa de flexibilidad y eficiencia de las arquitecturas.

1.3 *System-on-Chip* (SoC)

Los *System-on-Chip* son el resultado de la tendencia actual de integrar todos los componentes de un sistema electrónico en un único chip o circuito integrado. Esta tendencia es también visible en los métodos de diseño, basados cada vez más en la reutilización de los bloques IP (*Intellectual Property*) tanto *hardware* como *software*, formando plataformas fácilmente verificables. En la actualidad, y con el fin de aumentar aún más la integración de sistemas, se comienzan a diseñar sistemas electrónicos en los que cada bloque implementa una interfaz de

red, y donde la interconexión de estos se realiza a través de redes, que simulan las actuales redes de ordenadores. Estos sistemas se conocen como *Network-on-Chip* (NoC).

Los SoC se utilizan en el control de buena parte de aplicaciones, como en los dispositivos electrónicos de consumo (videoconsolas, reproductores de audio/vídeo…), en la automoción (control de airbag, climatizador…), en la industria (control de motores, robótica…), en las comunicaciones (teléfonos móviles, modem...), etc. En el diseño SoC se dan cita tres elementos conceptuales:

- La arquitectura *hardware* construida en torno a un sistema basado en microprocesador y múltiples bloques IP independientes conectados a este.
- La arquitectura *software* que incluye los sistemas operativos, los lenguajes de programación, compiladores, herramientas de modelado, simulación y evaluación.
- La integración de la arquitectura *hardware* y *software* proporciona la arquitectura global del SoC [12].

Los diseños SoC más exigentes incluyen al menos un procesador programable y a menudo una combinación de un procesador de control RISC y un DSP de procesado de señales digitales. También incluyen estructuras de comunicaciones sobre chips: bus o buses de procesadores y de periféricos y, en ocasiones, un bus de sistema de alta velocidad. Para los procesadores SoC es muy importante que el chip tenga unidades de memoria jerarquizadas y enlaces con memorias externas.

El diseño y uso de SoC implica, además del *hardware*, diseño e ingeniería a nivel de sistemas, compromisos *hardware-software* y particiones, así como arquitectura, diseño e implementación de *software* [13].

En la figura 3 se puede ver el diagrama de bloques de un dispositivo SoC en el que se identifican el microprocesador (ARM) así como un conjunto de bloques que añaden funcionalidad al sistema, todos integrados en el mismo dispositivo físico [14].

1.4 Metodologías de particionado HW/SW

La metodología clásica de diseño *hardware* hace del diseño de sistemas empotrados una tarea demasiado complicada, por lo que con la evolución en la integración de estos sistemas, las metodologías de diseño han evolucionado en concordancia. En la figura 4, adaptada de [15], se representa un diagrama de flujo del diseño de sistemas empotrados. La primera tarea en este flujo consiste en analizar el diseño a realizar, y realizar la partición HW/SW. Esta partición consiste en decidir qué partes del diseño serán ejecutadas por el microprocesador del sistema empotrado y qué partes serán ejecutadas mediantes bloques *hardware* específicos. La decisión debe estar basada en el coste de cómputo de cada función, así como de las necesidades de conectividad y dependencia entre ellos.

A partir de dicha partición, se realizarán, de forma independiente los diseños de la aplicación *software* empotrada y por otro la descripción *hardware* del resto de bloques. Cada uno de ellos seguirá el flujo clásico de compilación (en el caso *software*) y de síntesis (en el caso *hardware*), para, al final, realizar una integración de ambos.

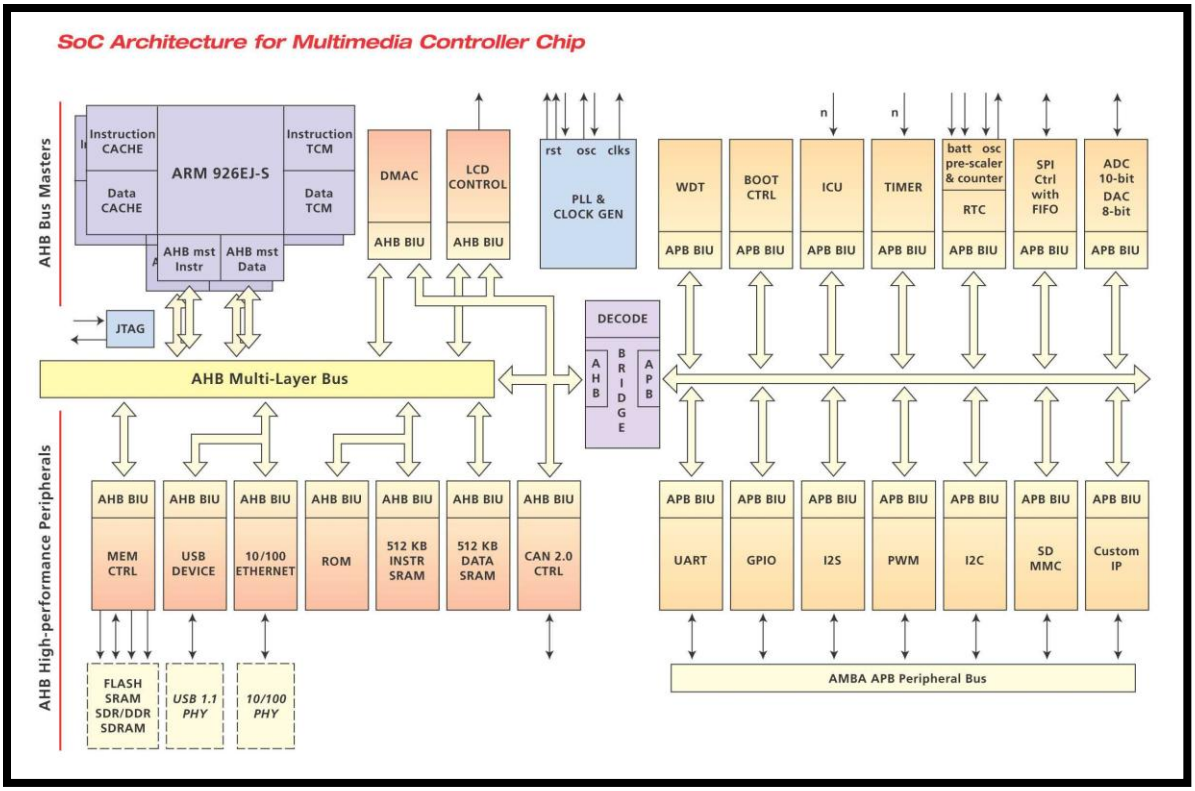


Figura 3. Arquitectura de un *System-on-Chip*.

Esta metodología busca mejorar los tiempos de diseño, al ser posible reutilizar una gran parte de bloques *hardware* a conectar al microprocesador y que son comunes a un gran número de sistemas empotrados, como son controladores de memoria, de red, o de periféricos.

2 Objetivos

El principal objetivo de este Proyecto Fin de Carrera es el establecimiento de la metodología de diseño necesaria para la aceleración *hardware* de una aplicación de procesado de eventos usando para ello una FPGA con el fin de alcanzar la mayor tasa de eventos procesados por segundo.

Para ello se establece el flujo de diseño *hardware*/*software* de un sistema empotrado en una FPGA, en el que se integra un microprocesador y un procesador específico *hardware* optimizado para la ejecución de la aplicación original.

El primer paso del flujo consiste en realizar el análisis de la aplicación, con el fin de establecer las especificaciones del diseño *hardware*, y las del diseño de la aplicación empotrada. A partir de dichos resultados, se establece la partición necesaria del sistema, al migrar a *hardware* aquellas funciones que requieren de recursos de cómputo elevados, mientras se mantienen en *software* los módulos que requieran de flexibilidad, sin necesidades adicionales de cómputo.

Teniendo en cuenta que se trata de un proyecto complejo, esta partición se da como punto de entrada para realizar el modelado del sistema por parte del equipo de desarrollo.

El resto de la metodología incluye la síntesis de alto nivel, la síntesis lógica, verificación post-síntesis, implementación y prototipado tomando como tecnología objetivo la plataforma basada en FPGAs.

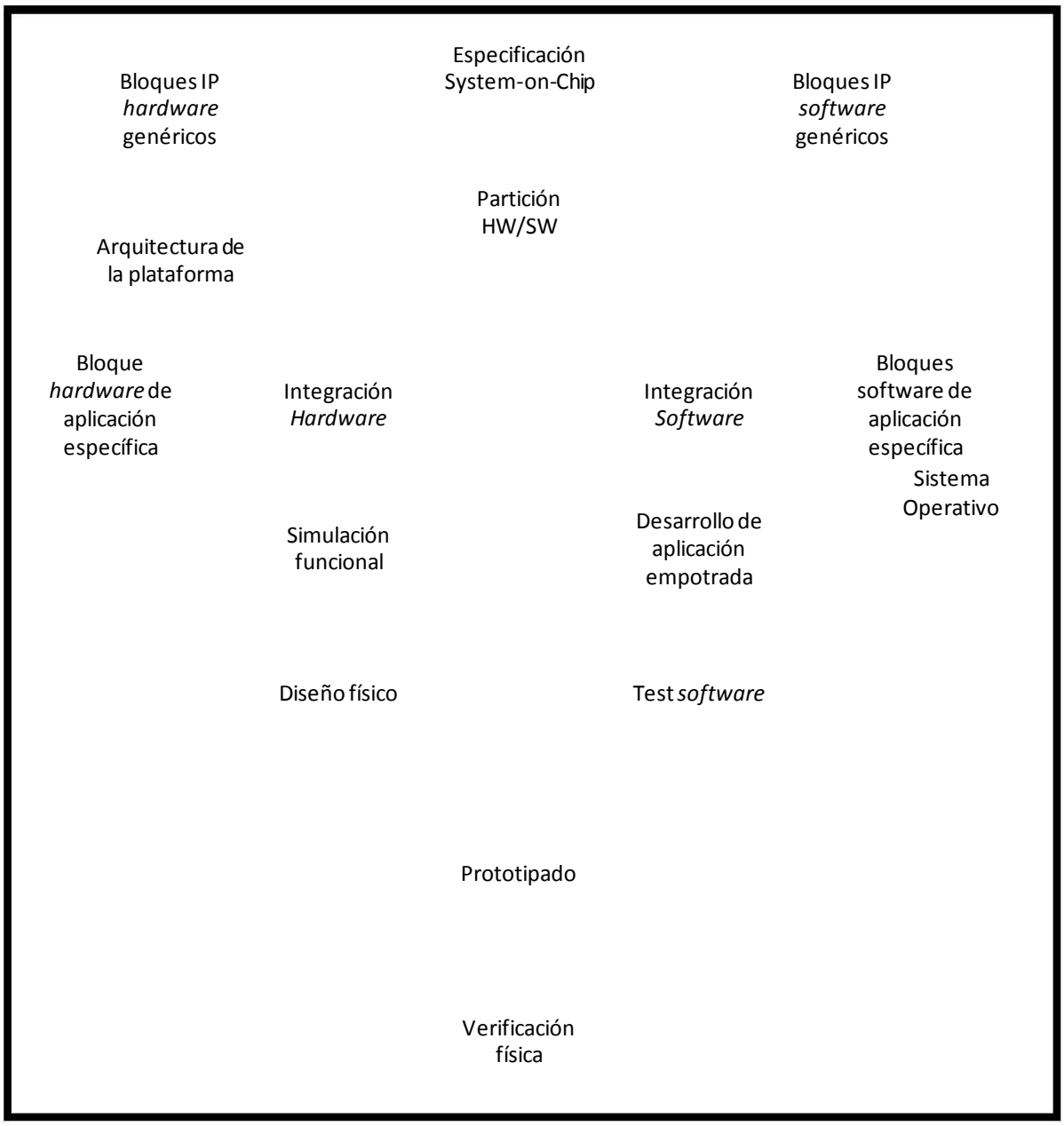


Figura 4. Diagrama de flujo del diseño de sistemas empotrados.

3 Peticionario

Actúa como peticionario de este Proyecto Fin de Carrera la División de Sistemas Industriales y CAD (SICAD) del Instituto Universitario de Microelectrónica Aplicada (IUMA) de la Universidad de Las Palmas de Gran Canaria, en el marco de las líneas de investigación promovidas por la citada división. En particular, este proyecto ha sido realizado bajo petición de la empresa Edosoft Factory S.L.

Por otro lado, la realización de un Proyecto Fin de Carrera es requisito indispensable para la obtención del título de Ingeniero de Telecomunicación por la Escuela de Ingeniería de Telecomunicación y Electrónica de la Universidad de Las Palmas de Gran Canaria.

4 Estructura del documento

El presente documento se divide en siete capítulos en los cuales se describe el trabajo realizado, estructurado tal y como a continuación se indica:

- **Capítulo 1: Introducción.** Se detallan antecedentes así como los objetivos del proyecto, su peticionario y la estructura del documento.
- **Capítulo 2: Dominio de aplicación.** Se describen en este capítulo aspectos arquitecturales y conceptuales específicos del presente Proyecto Fin de Carrera, con el fin de ubicarlo en un espacio de conocimientos.
- **Capítulo 3: Metodología de diseño.** Descripción de las tecnologías, herramientas y lenguajes usados en el proyecto, así como el flujo de diseño utilizado especificando secuencialidad de tareas y dependencias de las mismas.
- **Capítulo 4: Diseño de la plataforma.** Se describen en este capítulo los diferentes bloques *hardware* que componen la plataforma diseñada, así como los pasos seguidos en su diseño. A su vez se describe la composición de la plataforma *software* y la metodología de diseño de la plataforma *software*.
- **Capítulo 5: Síntesis del coprocesador de eventos.** Este capítulo recoge los pasos llevados a cabo para, partiendo de un modelo de descripción *hardware* en alto nivel, alcanzar una descripción *netlist* del mismo adaptado al dispositivo de implementación.
- **Capítulo 6: Integración y prototipado.** Se presentan en este capítulo los pasos seguidos para la integración del diseño sintetizado del coprocesador de eventos en la plataforma previamente diseñada y su depurado físico tras la implementación del mismo, así como resultados de rendimiento.
- **Capítulo 7: Conclusiones y líneas futuras.** Se exponen las conclusiones generales del Proyecto Fin de Carrera y se presentan algunas líneas de trabajo que pueden derivarse de su realización.

Capítulo 2:
Dominio de aplicación

1 Introducción

En este capítulo se revisarán las características principales de los procesadores de eventos en general, y de las particularidades de la aplicación de referencia usada en este Proyecto Fin de Carrera. En concreto se da una visión general de las aplicaciones de procesado de eventos y su arquitectura, continuando con la presentación de los principios de *Algorithmic Trading*. Se presentará la aplicación de referencia y sus requerimientos, explicando las posibles alternativas en cuanto a la solución arquitectural.

2 Procesadores de eventos

Se entiende por procesador de eventos un sistema capaz de encontrar patrones en secuencias de entrada de datos. La naturaleza de los eventos y de los patrones que reconoce el procesador varía en función de la aplicación para la que se use, no obstante existen propiedades comunes que se expondrán a lo largo de este capítulo.

2.1 Aplicaciones de procesado de eventos

Existen multitud de entornos diferentes que requieren de un procesador de eventos que analice secuencias de entrada y reconozca patrones en ellas. Este procesado puede realizarse en tiempo real, es decir, que el sistema tome como entrada eventos en el mismo instante en que estos son generados; o pueden realizarse tras ser almacenados en una base de datos. La condición de que el sistema realice el procesado en tiempo real o con eventos almacenados dependerá de las necesidades específicas de la aplicación.

Un ejemplo de aplicación sin necesidad de procesado en tiempo real, es el análisis de rentabilidad de un producto para una empresa de ventas *online*. Los eventos de entrada estarán compuestos por las compras del producto, las opiniones de los consumidores, las devoluciones de los clientes, etc. En general, el procesado de dicho tipo de eventos debe realizarse con posterioridad a su generación, y además, no se requiere que las acciones a llevar a cabo, se realicen en tiempo real [16].

Un ejemplo en el que sí se requiere una respuesta en tiempo real es en el de la seguridad de redes. Una red de ordenadores puede ser el objetivo de ataques informáticos por parte de usuarios maliciosos que quieran acceder a la información almacenada en alguno de los nodos de dicha red. Con el fin de protegerse de dichos ataques, los administradores de redes introducen medidas de seguridad pasiva como son los cortafuegos, evitando así conexiones no deseadas. En la figura 5 se muestra un ejemplo de una red sencilla, protegida por un cortafuegos.

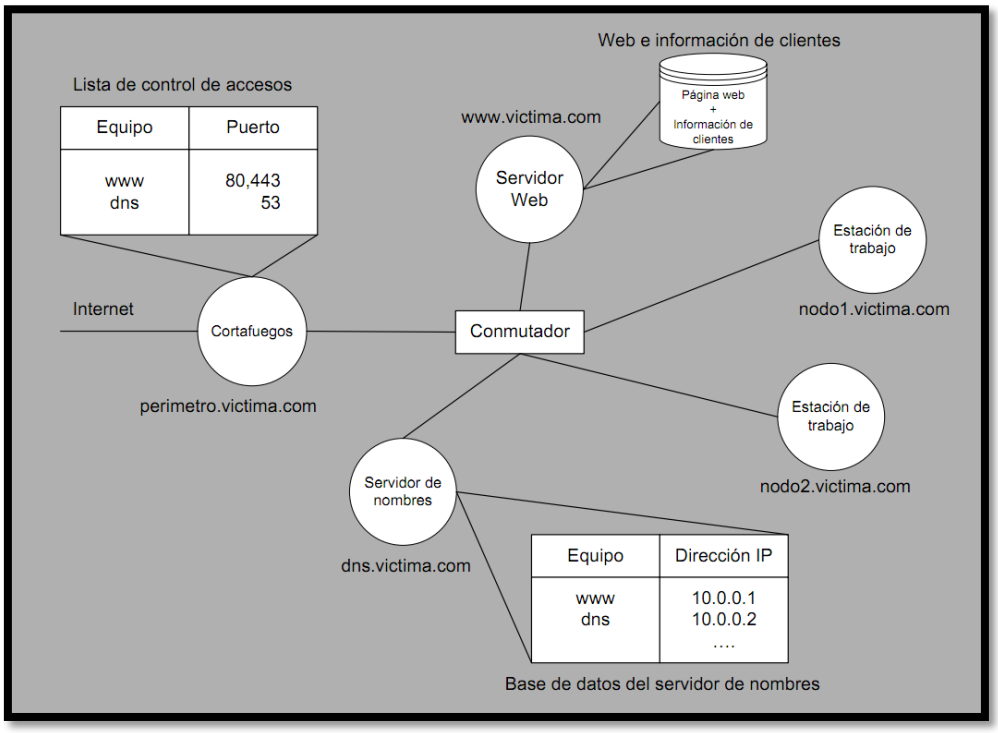


Figura 5. Estructura de una red informática genérica.

En la imagen se observa que existe, en la conexión entre la red e Internet, un cortafuegos que solo permite conexiones HTTP, HTTPS y DNS, con el fin de evitar conexiones a otro tipo de equipos.

El atacante, en este caso, intentará acceder a la red a través de los dos únicos nodos a los que tiene acceso, para lo cual, estudiará las vulnerabilidades del equipo y del sistema operativo que se encuentre tanto en el servidor DNS como en el servidor Web.

Para evitar estos ataques, un segundo sistema de seguridad consiste en analizar las peticiones Web y DNS, tratando de buscar patrones que se alejen del uso típico del servicio. En este apartado entra el juego el procesador de eventos, donde los eventos son las peticiones Web de entrada, y las acciones pueden ser la interrupción del servicio a una dirección IP determinada, cuando se comprueba un uso malintencionado. Este tipo de respuestas deben generarse en tiempo real, para evitar comprometer los datos almacenados en la red [17].

Otro ejemplo de uso de procesadores de evento en tiempo real, y que se corresponde a la aplicación de referencia usada en este Proyecto Fin de Carrera, es el de *Trading automático*. Este término hace alusión a sistemas electrónicos que generan órdenes de compra y/o de venta, automáticamente, en función de los parámetros de algún mercado de valores, sin intervención directa de alguna persona [18].

Este tipo de aplicaciones se encuentra en auge en la actualidad, proporcionando hasta el 80% de las transacciones en algunos mercados de valores [19, 20]. En este tipo de sistemas, se requiere de procesadores de eventos capaces de procesar una gran cantidad de eventos de mercado con una mínima latencia, y proporcionar una orden de compra o de venta (si se cumpliese un cierto patrón predefinido) en el mínimo tiempo posible.

2.2 Arquitectura de los procesadores de eventos

Si se define un procesador de eventos, en su versión más simple, es decir, un sistema que para un determinado conjunto de eventos de entrada, ejecuta una acción en caso de que presente unas condiciones determinadas previamente, la arquitectura del procesador puede dividirse en tres grandes bloques: interfaz de entrada/salida, núcleo de procesamiento y ejecutor de resultados (figura 6).

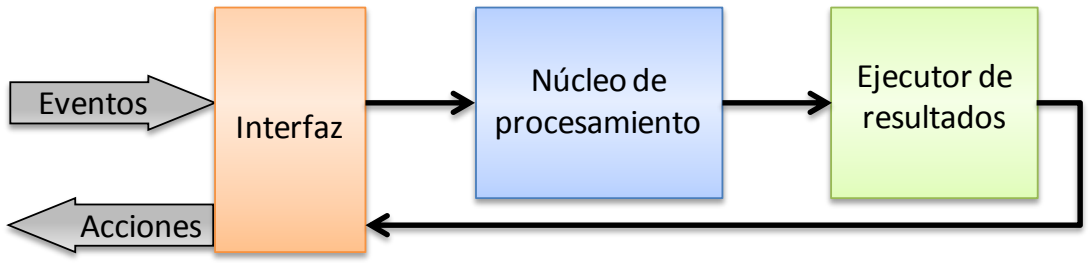


Figura 6. Arquitectura simple de un procesador de eventos.

La función del módulo de interfaz será la de recoger los eventos del medio de transmisión del que provengan, y empaquetarlos de forma óptima al núcleo de procesamiento. Este último, eje central del procesador de eventos, será el encargado de realizar las comprobaciones de cumplimiento de las condiciones establecidas previamente. Por último, el ejecutor de resultados es un bloque esclavo del núcleo de procesamiento, que realizará una acción predefinida en caso de que el núcleo de procesamiento encuentre una coincidencia de su patrón de búsqueda. En caso de que el procesador de eventos esté comprobando más de una condición, y que cada una lleve asociada una acción diferente, existirá un código de operación que será comunicado por el núcleo al ejecutor de resultados.

La arquitectura planteada, como ya se adelantó, es la arquitectura más simple de un procesador de eventos, que trata con eventos de entrada de forma independiente, y que por su sencillez, es la que alcanza un rendimiento mayor. Sin embargo, en otras aplicaciones es necesario contemplar dependencias entre tramas, ya sea en base a fuentes de eventos diferentes, a códigos internos a cada evento, o simplemente a muestras temporales. Ejemplos de estas necesidades son:

- Condición de actuación asociada a la diferencia entre dos fuentes de eventos, como que dos sensores de temperatura indiquen valores diferentes, pero que cada uno genere su propia trama.
- Que secuencias de eventos diferentes estén asociadas a productos diferentes de una bolsa de valores. Por ejemplo, si la condición de compra es una relación de precios entre un producto y otro.
- Cuando la condición de actuación se corresponde a una tendencia que, por lo tanto, requiere de una memoria de valores históricos de tramas de eventos previos.

Este tipo de necesidades implica la necesidad de modificar la arquitectura del procesador de eventos para añadir el soporte a un almacenaje de información que contenga el estado actual del sistema, en el que se encuentre la información requerida sobre tramas pasadas y su evolución, si fuese necesario (figura 7).

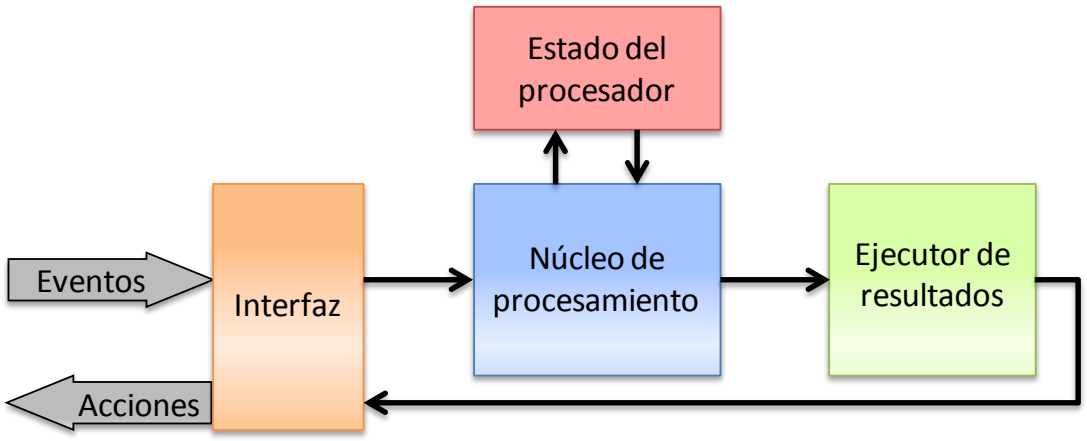


Figura 7. Arquitectura de procesador de eventos con memoria.

En ambas arquitecturas, el diseño del procesador de eventos lleva intrínsecamente definidas las condiciones de actuación del procesador de eventos. Sin embargo, es algo común la necesidad de que dichas condiciones, así como las acciones asociadas a ellas, puedan modificarse en tiempo de ejecución. Para ello, hay que modificar tanto el núcleo de procesamiento como el ejecutor de resultados, para que tanto las condiciones como sus acciones, puedan leerse de las variables de estado, en lugar de estar predefinidas en el diseño. Además, se requiere que la interfaz sea capaz de discernir entre eventos y reglas de procesado para enviar unas al núcleo de procesamiento, y las otras a las variables de configuración (figura 8).

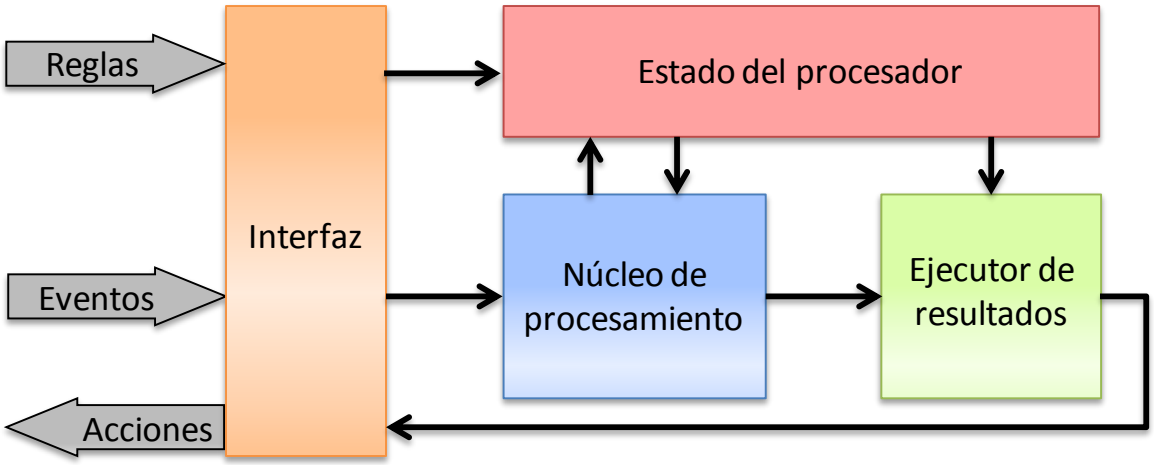


Figura 8. Arquitectura de un procesador de eventos con estrategias modificables en ejecución.

Otra posible modificación a las arquitecturas propuestas, consiste en permitir que las acciones del procesador de eventos modifiquen el estado del procesador. Puede resultar de utilidad, definir condiciones de procesamiento, que al cumplirse, creen nuevas condiciones en el procesador, para abarcar así unas estrategias de procesado mucho más complejas y que puedan satisfacer más necesidades. Para conseguirlo, se requiere que el bloque ejecutor de resultados pueda modificar el estado del procesador, generando nuevas condiciones y sus respectivas acciones (figura 9).

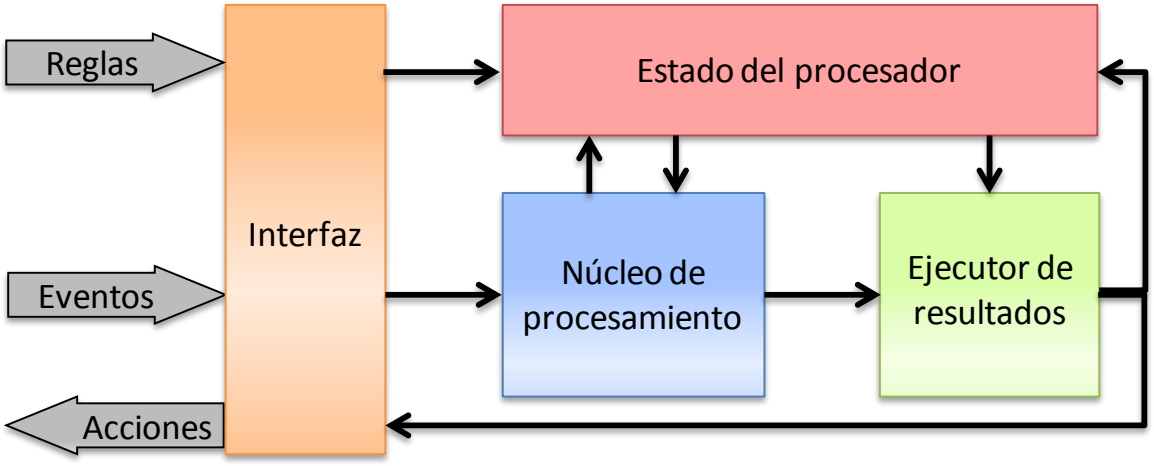


Figura 9. Arquitectura de un procesador de eventos con autoconfiguración.

Cada procesador de eventos puede tener unas u otras necesidades que modifiquen en mayor o menor medida las arquitecturas descritas anteriormente, aunque siempre manteniendo la misma filosofía de bloques de interfaz, núcleo y ejecutor.

3 Algorithmic Trading

El *Algorithmic Trading* es la práctica de operar en los mercados financieros siguiendo algoritmos predefinidos. De esta forma, las operaciones son llevadas a cabo por sistemas electrónicos en lugar de por agentes de bolsa.

Con el continuo avance de las tecnologías de la información y las comunicaciones (TIC), no es de extrañar que un mundo como es el del mercado financiero evolucione con ellas. Es por ello que, cada vez más, en aquellos mercados con una alta liquidez, se utilicen nuevos métodos tecnológicos que aseguren una rentabilidad mayor a los inversores.

En los mercados financieros, cada orden realizada por un agente de bolsa, provoca cambios en los títulos que se intercambian, ya no solo en la misma bolsa, sino incluso en otras bolsas del mundo. Esta dependencia entre títulos, y los continuos cambios en sus precios, es lo que aprovechan los inversores para conseguir rentabilidad en sus inversiones a través de la plusvalía.

Los inversores generan rentabilidad de su capital invertido a través de dos factores:

- **Dividendos**: Se trata de la repartición de los beneficios que realizan las sociedades mercantiles al realizar el cierre de un año, tras haber subsanado deudas de ejercicios anteriores, entre sus accionistas.
- **Plusvalía**: Se trata de la diferencia entre el precio de compra de un título y su precio de venta con posterioridad. Si dicha diferencia es positiva, el inversor ganará un valor que se conoce como plusvalía, y si es negativo, perderá su valor que se conoce como minusvalía.

Las técnicas de *Algorithmic Trading* suelen ir asociadas a alcanzar una rentabilidad del capital invertido mediante la plusvalía generada entre la compra y venta de valores. En muchos casos, y gracias a los sistemas automatizados que se presentan, pueden generarse muchas veces operaciones con valores de plusvalía muy pequeños, pero que repetidas muchas veces en cortos periodos de tiempo, generan al final del día grandes beneficios para el inversor.

De esta forma, los sistemas de *Algorithmic Trading* permiten realizar un gran volumen de operaciones, que serían inviables si se realizasen por agentes de bolsa directamente. Este incremento en el número de operaciones provoca también, que en muchos mercados de capitales actuales, la mayor parte de las transacciones sean en realizad realizados por este tipo de técnicas, en lugar de por agentes de bolsa.

Evidentemente, debido a la gran dependencia que existe en la economía mundial a estos mercados, existen diferentes puntos de vista respecto a estas nuevas técnicas. Por un lado hay quienes defienden su uso pues es una evolución lógica del sistema financiero gracias a los avances

tecnológicos. Sin embargo, existe quien cree que provoca inestabilidad en los mercados financieros.

En las técnicas de este tipo de transacciones existen tres importantes líneas de actuación: definir las mejores estrategias de adquisición y venta de títulos, la creación de sistemas electrónicos capaces de generar las órdenes de mercado lo más rápido posible, adelantándose así a otros posibles interesados en realizar otras acciones que puedan afectar al mercado en el que se trabaje y la de posicionar dichos sistemas, lo más cercano posible (en lo que a dispositivos de red se refiere) del nodo del mercado en cuestión dentro de la red electrónica de comercio.

4 Aplicación de referencia

En este Proyecto Fin de Carrera, se usará como modelo de referencia, una aplicación de *Algorithmic Trading* dedicada a la generación de órdenes de mercado en base a estrategias definidas previamente por el inversor.

En particular, la aplicación de referencia utilizada es propiedad de Edosoft Factory S. L., y por tanto no puede revelarse información respecto a su estructura interna [21].

En el desarrollo del presente Proyecto Fin de Carrera, se parte de la aplicación ya desarrollada, que servirá como modelo de referencia para la verificación de la funcionalidad, para la realización de las medidas de rendimiento y para la mejora del diseño planteado.

Se parte de un diseño en alto nivel descrito en SystemC, resultado del estudio de la aplicación original y su posterior partición. Este diseño es fruto del trabajo dentro del proyecto RT_CORE del Instituto Universitario de Microelectrónica Aplicada y su diseño no es descrito en esta memoria por encontrarse protegido por una cláusula de confidencialidad. En lugar de ello, se presentan en este documento los métodos utilizados para su implementación *hardware*, así como las diferentes peculiaridades en su flujo de desarrollo.

Se pueden enumerar a continuación, algunas características que hacen del procesador de eventos de Edosoft un diseño complejo y de gran versatilidad con el que poder manejar todo tipo de estrategias de *Algorithmic Trading*.

- Permite crear estrategias, compuestas por órdenes, cada una con una acción a realizar y un conjunto dinámico de condiciones de disparo.
- Permite definir acciones de activación de nuevas órdenes y estrategias previamente almacenadas en el sistema.
- Pueden definirse condiciones temporales de comienzo, finalización y duración de estrategias.
- Pueden definirse comparaciones entre un gran número de atributos de cada título, comparando atributos diferentes de productos diferentes. Además, las comparaciones pueden ser complejas, no solo de igualdad o de mayor o menor, sino también de valores calculados a partir de atributos, del estilo $K_1 \cdot Atributo + K_2$.

La descripción en alto nivel realizada en SystemC está formada por 34 módulos de procesamiento y 29 módulos controladores de memoria interna. Las memorias internas usarán recursos internos de la FPGA con el fin de almacenar estructuras de datos necesarias para la implementación de la aplicación original. Los bloques de procesamiento pueden dividirse, según su funcionalidad, entre los cuatro bloques conceptuales presentados anteriormente: interfaz, núcleo de procesamiento, ejecutor de resultados y estado del procesador. Para realizar los análisis de la implementación se utilizará esta nomenclatura de cuatro bloques en lugar de la verdadera arquitectura del diseño, con el fin de evitar publicar información protegida por la cláusula de confidencialidad.

4.1 Requerimientos del sistema

A partir del estudio de la aplicación *software* de RT-CORE se deducen un conjunto de requerimientos que deben estar presentes en la arquitectura final del sistema. En los siguientes puntos se listan de forma cualitativa las principales necesidades identificadas y la propuesta de soluciones que se plantea incluir en el prototipo final.

- **Interfaz de entrada de eventos**: Los eventos se reciben a través de una interfaz TCP/IP, encapsulados ya sea en formatos estándar o propietario. Por otra parte se plantea que la tasa de tráfico a tratar es del orden de un millón de eventos por segundo. Ello plantea la necesidad de disponer de una interfaz Gigabit Ethernet que sea capaz de gestionar hasta 1 Gbps de tráfico de red. Para gestionar esta interfaz, el sistema debe incluir, además de los componentes *hardware* necesarios, la correspondiente implementación de la pila TCP/IP (generalmente implementados en *software* por flexibilidad). Por ello se requiere de la correspondiente infraestructura *hardware*/*software*.
- **Reconfigurabilidad**: Existen dos formas principales de añadir reconfigurabilidad al sistema: programabilidad mediante la utilización de uno o varios microprocesadores empotrados que ejecutan una aplicación *software* y reconfigurabilidad *hardware* utilizando dispositivos programables tipo FPGA. El dispositivo FPGA deberá proporcionar mecanismos que soporten su reconfiguración total o parcial a través de puertos dedicados.
- **Almacenamiento local de los datos de entrada:** Normalmente los datos que provienen de la interfaz Ethernet se envía a colas FIFO para su posterior procesamiento.
- **Preparación/ordenación/procesamiento de eventos y reglas**: El procesamiento de los eventos de entrada requiere una fase de preparación y almacenamiento local en un formato que facilite su acceso por las unidades funcionales. Generalmente requiere de estructuras especializadas por lo que será necesario crear unidades específicas para su tratamiento. Los requisitos impuestos por estas unidades depende de la estrategia a utilizar, pero tienen en común la necesidad de almacenamiento local y de unidades lógicas (comparadores, etc.) rápidas. En el caso de usar estructuras complejas de memoria será necesario disponer de unidades que calculen las direcciones usando registros base y unidades aritméticas.

- **Jerarquía de memoria**: La necesidad de procesamiento de datos generalmente se organiza de forma jerarquizada, facilitando la localidad de los datos. Es necesario organizar la memoria en diferentes niveles ya sea en el propio dispositivo (BRAM), o disponer de un controlador de memoria externa con soporte de utilización de memorias de tipos SRAM o DDR2/3.
- **Otros recursos:** Como se deduce de los requisitos anteriores, la posibilidad de disponer de un procesador empotrado facilita la gestión de la pila TCP/IP y de otros parámetros de configuración del sistema final, aportando la flexibilidad *software* deseable

5 Soluciones arquitecturales

Una vez definidos los requerimientos de la aplicación, se deben estudiar las diferentes posibilidades de implementación de la arquitectura del RT_CORE. Existen diferentes alternativas de implementación, considerando aspectos de flexibilidad, capacidad de incorporación de reglas al sistema o capacidad de procesamiento.

5.1 Arquitectura basada en núcleos de tipo *soft-processor* integrados

Esta solución se basa en la creación de un sistema compuesto de N núcleos procesadores (dependiendo de la capacidad de la FPGA) que incremente el paralelismo de procesamiento del sistema. Se trata de procesadores de tipo *soft-core* que conforman el sistema multiprocesador. La idea es usar dichos núcleos para realizar el procesado de eventos de forma flexible, aprovechando la potencia de cómputo del núcleo. Esta solución multiprocesador es la extensión masiva de la utilización de una CPU multinúcleo. La ventaja que ofrece esta solución es la facilidad de reprogramación y la adaptación del núcleo procesador a las características de la aplicación, eliminando unidades funcionales no utilizadas, y adaptando tamaños de *cache* y otros recursos. Ejemplos de este tipo de procesadores son MicroBlaze (Xilinx), o Altera NIOSII. Otras soluciones disponibles como núcleos procesadores *hardIP* es el caso de procesadores PowerPC405 o PowerPC440, con arquitectura superescalar de doble vía. La figura 10 representa el diagrama de bloques de una arquitectura basada en núcleos de tipo *soft-processor* integrados.

Las prestaciones de este tipo de soluciones están condicionadas por la arquitectura de interconexión de los procesadores y la arquitectura del subsistema de memoria diseñado.

5.2 Arquitectura de implementación *hardware*

Una segunda alternativa, ubicada en el otro extremo del espacio de soluciones, es disponer de una implementación *hardware* del sistema. Evidentemente, se pierde flexibilidad a costa de incrementar la velocidad de procesamiento. En este sentido, la implementación del procesamiento de las reglas se ejecuta en elementos de procesamiento reconfigurables (EPR) implementados directamente sobre la FPGA. Esta solución presenta una ganancia de un orden de magnitud sobre la solución basada en PC. A diferencia del caso anterior, el esfuerzo de diseño se

centra en la síntesis de los procesadores de evento, su interconexión y su integración con los módulos de entrada y salida de eventos. En la figura 11 se representa la arquitectura general de esta solución. Como se puede observar, el sistema consta de una interfaz de red Gigabit Ethernet, conectada al microprocesador, que envía las tramas de eventos a los bloques de procesamiento (que incluyen la interfaz, el núcleo de procesamiento, el ejecutor de resultados y las memorias que almacenen el estado del procesador). Existe una interfaz con memoria de alta velocidad (DDR2/3) usada por la CPU.

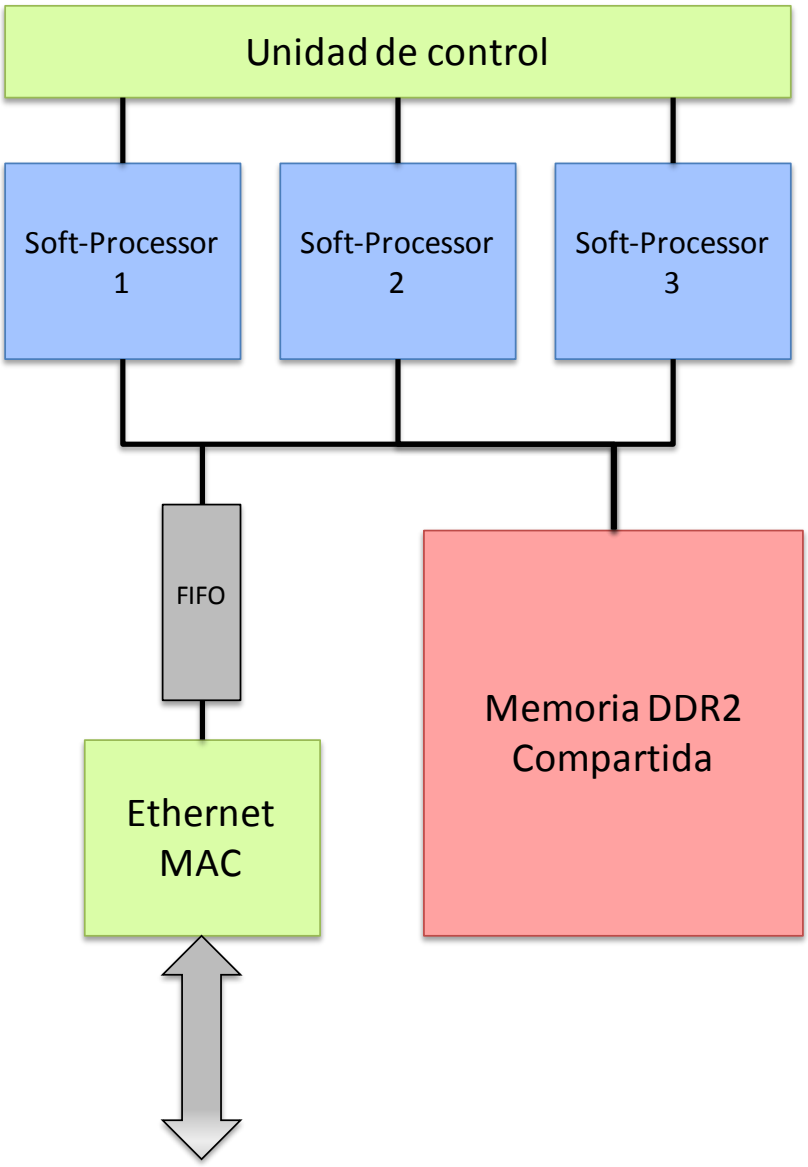


Figura 10. Arquitectura basada en *soft-processor*.

La principal característica de esta arquitectura es que las estrategias de comparación son las que definen el esquema lógico de los procesadores, consiguiendo una mayor tasa de procesado, a costa de la flexibilidad, pues no está pensada para modificar las estrategias en tiempo de ejecución. Aun así, existen técnicas de reconfiguración parcial de la FPGA, que permiten reprogramar parte de la lógica del dispositivo, mientras el resto permanece en funcionamiento.

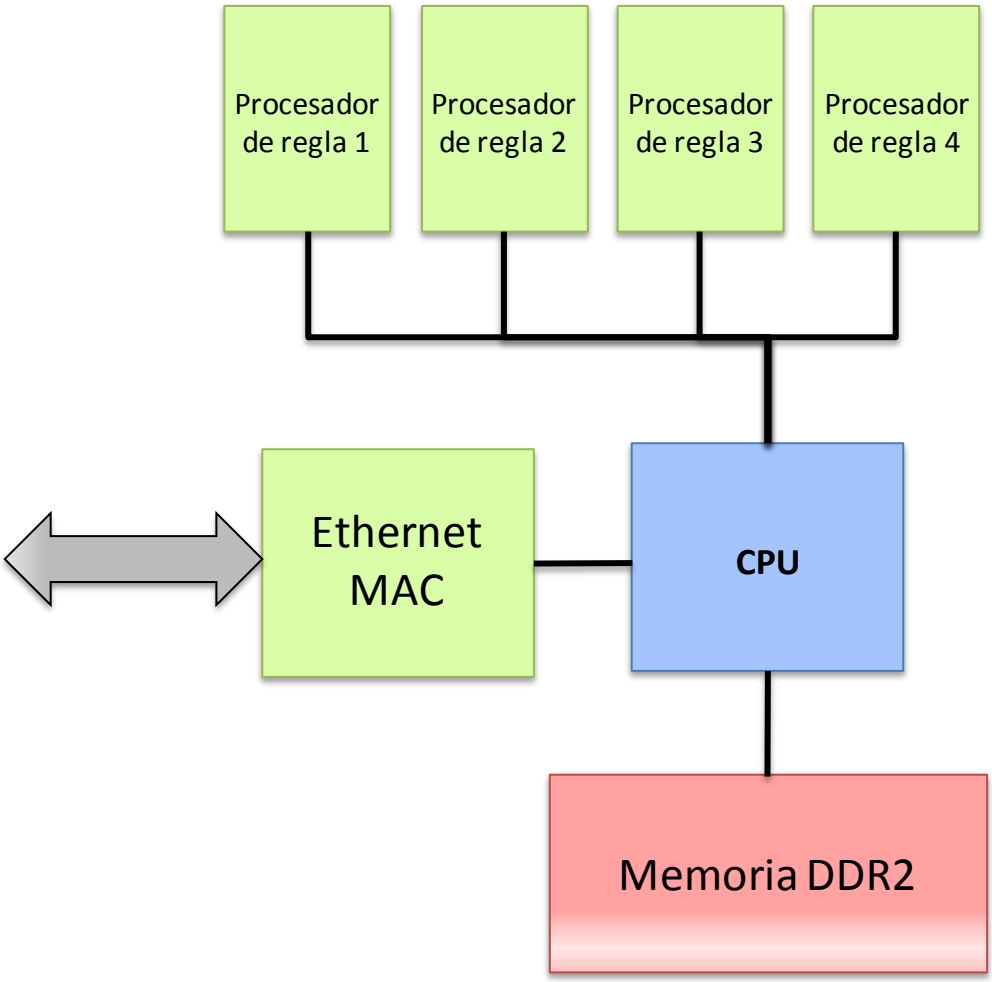


Figura 11. Arquitectura de implementación *hardware*.

Dependiendo de los recursos disponibles en la FPGA, es posible implementar un número elevado de estrategias, aunque finito. Este uno de los principales inconvenientes de esta arquitectura.

5.3 Arquitectura de implementación híbrida

Esta solución plantea un punto intermedio en el espacio de soluciones, entre las dos descritas anteriormente. La principal característica de esta implementación es que la estrategia de comparación está mapeada en memoria interna de la FPGA (BRAM).

Los accesos a memoria, ralentizan el procesado frente a la solución *hardware*, pero permite una mayor flexibilidad, al permitir modificar las estrategias mediante la alteración de los datos almacenados en memoria.

Además de esto, pueden reutilizarse los núcleos de procesamiento para varias estrategias, permitiendo así una mayor capacidad.

Para permitir la modificación del contenido de los bloques de memoria internos existen dos posibilidades:

- Modificarlas a través del puerto JTAG de la FPGA, opción dependiente de la tecnología.
- Integrar un bloque dedicado a la recepción de tramas de estrategia que modifiquen el contenido de las memorias.

En función de la ocupación del procesador de eventos y de la capacidad del dispositivo programable, puede replicarse el núcleo de procesamiento para aumentar el rendimiento del sistema final. En dicho caso es necesario implementar una unidad de despacho, que identifique eventos asociados a una u otra estrategia. Es importante recalcar que en este tipo de sistemas, no es posible realizar procesado de eventos fuera de orden, al existir condiciones de disparo que hacen alusión a tendencias de atributos de títulos. En la figura 12 se muestra un ejemplo de arquitectura híbrida.

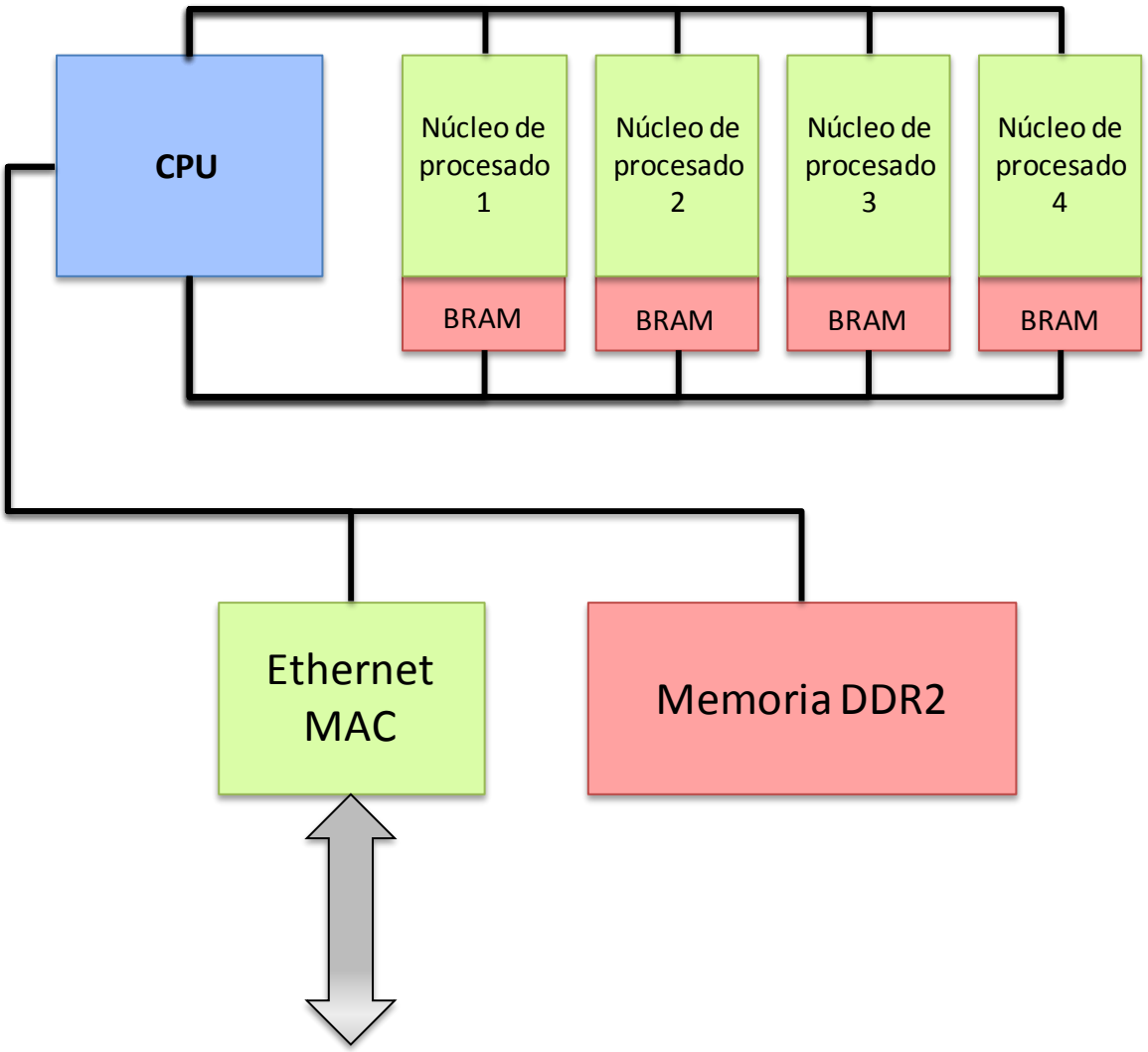


Figura 12. Arquitectura híbrida.

5.4 Comparativa entre arquitecturas

Las tres soluciones planteadas proporcionan tres puntos diferentes del espacio de soluciones, es decir, tres compromisos diferentes entre prestaciones y flexibilidad/capacidad del sistema. En la figura 13 se presenta una gráfica que describe la relación entre estas arquitecturas, presentando los resultados sobre los ejes ya comentados.

También cabe destacar que, en función de las necesidades del sistema, puede descartarse algunas de las soluciones. Un ejemplo es la necesidad de modificar las estrategias en tiempo de ejecución, lo cual no es viable con una solución *hardware*, siendo necesaria una solución híbrida o basada en *soft-processor*.

Para el presente Proyecto Fin de Carrera se ha optado por una solución híbrida, ya que era una especificación inicial la necesidad de modificar las estrategias sin interrumpir, en la medida de lo posible, el procesado de eventos continuo.

Además de las características anteriores, se ha implementado un bloque dedicado a recibir tramas con estrategias empaquetadas, y almacenar en memoria interna los parámetros de estas. Esta decisión viene dada por la necesaria conservación de la funcionalidad de la aplicación original, que estaba preparada para este tipo de reconfigurabilidad. La estructura de las tramas de entrada y de salida viene dada también por mantener la compatibilidad con el modelo original a acelerar.

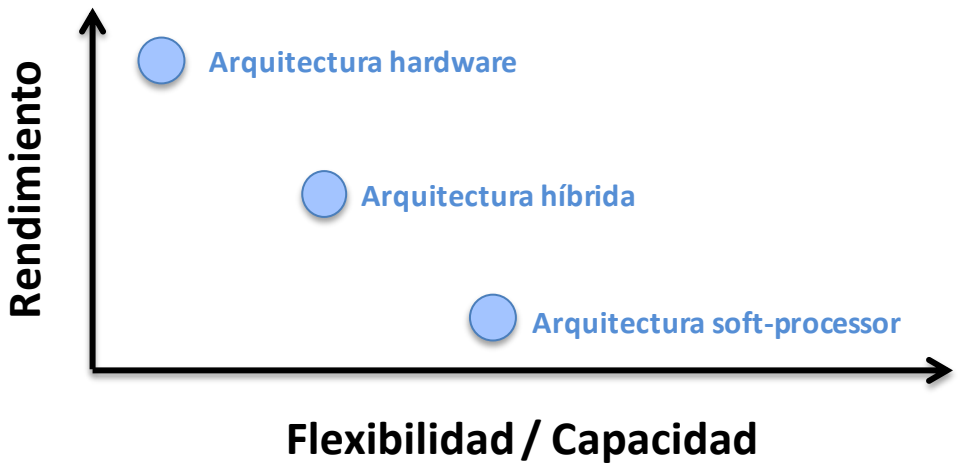


Figura 13. Comparativa entre arquitecturas.

6 Conclusiones

En el presente capítulo se han explicado las diferentes arquitecturas conceptuales de un procesador de eventos, así como sus posibles aplicaciones y características a tener en cuenta en su diseño.

Se ha presentado brevemente, en relación al modelo de referencia de procesador de eventos utilizado en este proyecto, el concepto de *Algorithmic Trading*, así como su importancia en los mercados actuales.

Además se ha presentado brevemente el modelo de referencia utilizado para realizar su implementación hardware con el fin de mejorar sus prestaciones.

Por último se han descrito las diferentes soluciones arquitecturales que se presentan a la hora de realizar una adaptación *hardware* de la aplicación original. Además se adelanta una comparativa entre ellas, así como la solución adoptada en este proyecto.

Capítulo 3:
Metodología de diseño

1 Introducción

La metodología se define como el conjunto de métodos que se siguen en una investigación científica, entendiendo un método como el modo de decir o hacer con orden [22].

En el presente capítulo se definen las distintas etapas en las que consiste el diseño del sistema propuesto en este PFC. Además se introducirán aquellos elementos que caracterizan dichas etapas y ayudan a comprender posteriormente su desarrollo.

En el marco en el que se desarrolla este PFC se parte de una aplicación *software* con el fin de obtener un sistema electrónico capaz de realizar la misma funcionalidad, incrementando sus prestaciones, y bajo condiciones de funcionamiento similares. Esto implica unos requisitos de comunicaciones elevadas por lo que es necesario definir todo el proyecto como un único bloque *hardware*. Es por ello que se toma la decisión, de definir un sistema compuesto por diferentes bloques con funciones específicas, entre el que estará el bloque procesador de eventos.

Debido a las necesidades expuestas anteriormente, se presentan dos líneas de actuación en el proceso de diseño:

- Un diseño de un circuito integrado partiendo de una descripción del diseño en un lenguaje a nivel de sistemas, tal como SystemC.
- El diseño de una plataforma compuesta por un microprocesador empotrado y un conjunto de periféricos al que se le conectará el coprocesador de eventos y que tendrá por objeto dotar al bloque *hardware* de las necesidades de comunicación requeridas.

2 Metodología de diseño

En todo diseño, ya sea *hardware*, *software* o de cualquier índole, se pueden enumerar tres fases claves en la metodología de diseño, a saber: especificación, diseño y verificación.

En la primera fase se analizarán las especificaciones del sistema a realizar, ya sean estas funcionales, temporales o de cualquier otra índole que se haya dado. Una vez definidas y acotadas, se procederá al diseño del sistema en el que se decidirá la tecnología, las etapas de desarrollo, la partición del diseño (si la hubiese), etc. Ello produce una primera versión funcional del sistema que se utilizará para su verificación.

Con una primera versión terminada, se verificará que cumple con las especificaciones definidas en primera instancia, para, adoptar las medidas correctoras necesarias en fases tempranas del ciclo de diseño, repitiendo el proceso hasta obtener una versión que cumpla con las especificaciones funcionales.

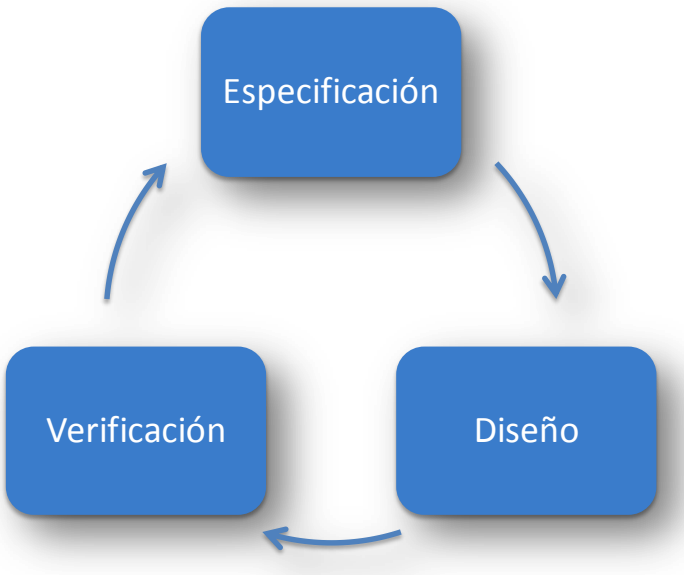


Figura 14. Flujo simple de diseño.

En el diseño que se plantea, los datos de entrada al sistema se deberán recoger en formato de tramas TCP/IP. El tratamiento de la comunicación por la red, implica, no solo la extracción de la carga útil de datos de los paquetes, sino tratar los paquetes de control tanto del protocolo TCP como de los inferiores (IP, ARP, Ethernet, etc). Para dar solución a estos requerimientos, se ha optado por el diseño de un *System-on-chip (*SoC*)* compuesto por un microprocesador empotrado y un conjunto de periféricos interconectados mediante diferentes soluciones (Buses, DMA, etc). De esta forma, el coprocesador de eventos pasará a ser un bloque perteneciente a una plataforma, siendo el microprocesador el encargado de preparar las tramas de datos y hacerlas disponibles para este.

Los datos de entrada del coprocesador de eventos se presentarán como un paquete de datos organizados al byte, con un formato previamente establecido y que entenderán tanto la aplicación *software* que es ejecutada en el microprocesador empotrado como el bloque *hardware* que los procesará [23].

Atendiendo a lo comentado anteriormente, se presenta dos líneas de trabajo:

- Diseño de la plataforma, en el que se incluirá la selección de aquellos bloques *hardware* necesarios, así como del diseño de la aplicación *software* a ejecutar en el microprocesador.
- Diseño del coprocesador de eventos, partiendo de la especificación de la aplicación *software* disponible.

Como tecnología sobre la que implementar el diseño se ha optado por la FPGA Virtex 5 disponible en una placa de desarrollo ML510 de Xilinx. Este dispositivo posee dos núcleos microprocesadores PowerPC440 empotrados en la FPGA. De igual forma, la placa posee dos puertos Gigabit Ethernet que permite disponer de ancho de banda suficiente para la implementación del prototipo básico.

En la figura 15 se muestra el flujo utilizado para el diseño del sistema completo, en el que se incluye la síntesis de alto nivel del bloque coprocesador, bloque *hardware* que hará de periférico en una plataforma completa con microprocesador incluido.

El flujo de diseño propuesto está centrado en la implementación sobre la FPGA. Este flujo es más simple que el del diseño de un ASIC. En general, las etapas iniciales del flujo serán comunes, aunque optimizadas para la tecnología destino, pues las descripciones a nivel de sistema y RTL serán comunes. Sin embargo, la síntesis lógica, que traduce la descripción RTL, mapeándola a ecuaciones booleanas y posteriormente a células lógicas de una determinada biblioteca, será dependiente de la tecnología a utilizar, siendo el espacio de soluciones más reducido en el caso de las FPGAS. Además de esto, la fase de implementación está mucho más acotada en un dispositivo como una FPGA donde los recursos están pre-asignados a una posición definida, siendo necesaria únicamente su programación.

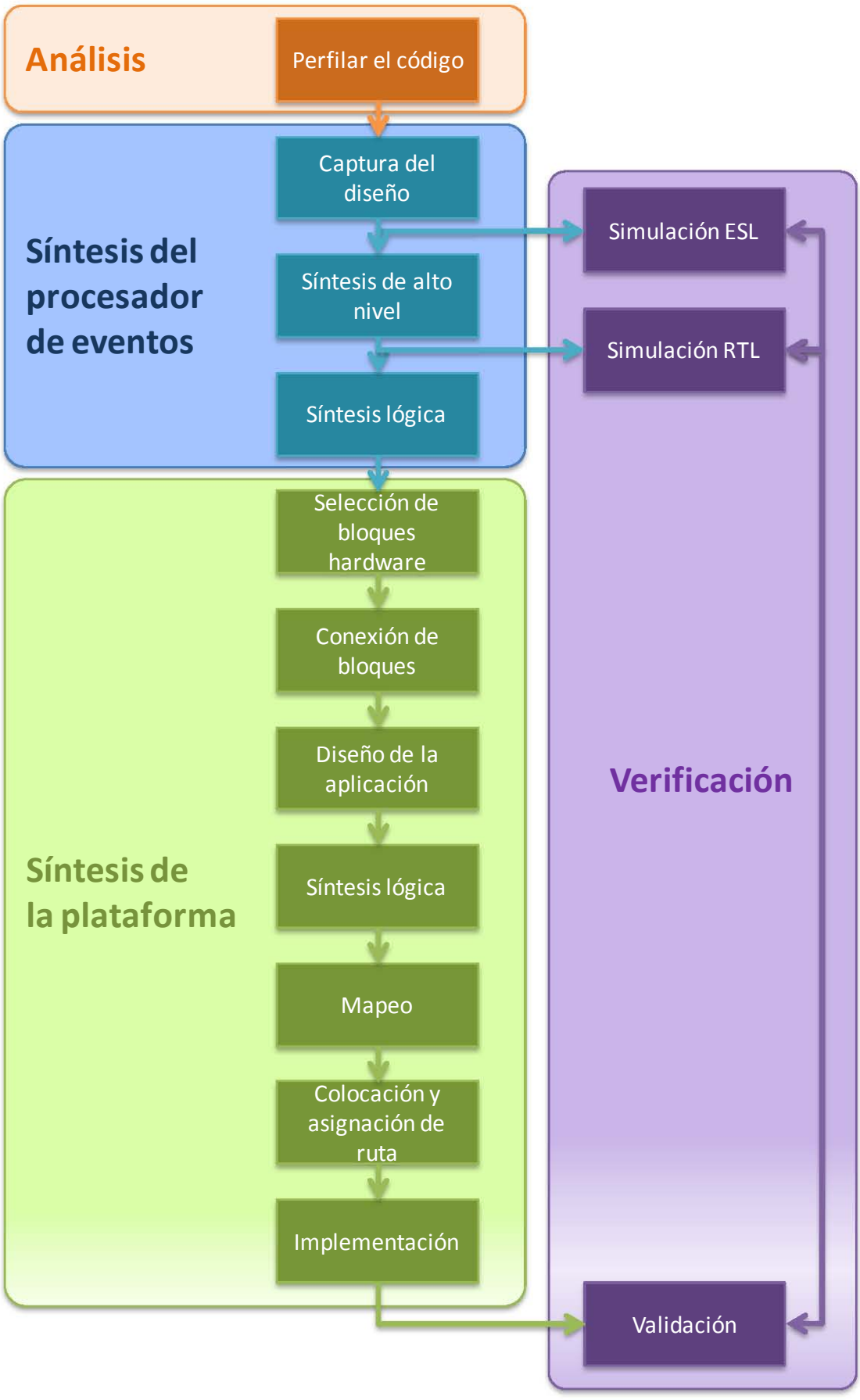


Figura 15. Diagrama del flujo de diseño.

2.1 Análisis

En la fase de análisis del diseño se realiza la obtención de un perfil de ejecución del programa original. Con ello se pretende obtener información acerca de la complejidad de la aplicación, su modularidad, así como de características de coste computacional y temporal de las distintas funciones que la componen. Esta información, junto a un estudio conceptual de la dependencia de datos, facilita durante la fase de diseño la toma de decisiones relacionados con la partición, las necesidades de comunicación y su conectividad, produciendo como resultado la jerarquía del diseño del coprocesador.

2.2 Diseño del coprocesador de eventos

A raíz del flujo presentado, es lógico pensar que es necesario priorizar el diseño del coprocesador de eventos (en adelante CE), ya que para poder realizar su verificación se requiere de un bloque *hardware* que integrar a la plataforma. Sin embargo, y con el fin de paralelizar el desarrollo, se decidió diseñar la plataforma y validarla de forma concurrente con la captura del diseño y simulación ESL.

Para ello es necesario sustituir el bloque a desarrollar por otro que permita comprobar el correcto funcionamiento de todo el sistema. Para poder realizar esto de forma eficaz, es importante conocer la finalidad de la plataforma propuesta, esto es, ser capaz de hacer llegar los datos provenientes de una interfaz Ethernet, al bloque *hardware*. Es por ello que, para validar el correcto funcionamiento de la plataforma, basta con asegurar que es capaz de comunicarse por la red en ambos sentidos (recibir eventos y enviar respuestas) y comunicarse, también en ambos sentidos y a través del protocolo decidido, con el bloque que actualmente ocupa el lugar del EP. Para comprobar ambas direcciones, es necesario que este reciba y envíe datos, por lo que, por sencillez se ha optado por el diseño de un bloque hardware de echo que recibirá una trama de datos, y la volverá a enviar de vuelta en el mismo orden simulando así, la funcionalidad del CE.

En el diseño del CE, la primera fase consiste en la captura del diseño, es decir, traducir la aplicación a una descripción *hardware* de alto nivel en SystemC. En esta primera fase se definen las particiones internas del bloque, los protocolos de comunicación y sincronismo, así como las memorias que son accedidas por distintos módulos. Es también en esta etapa donde debe definirse la interfaz del módulo mediante el protocolo con el que se comunicará con el resto del sistema.

Cuando la descripción del CE haya sido verificada mediante, simulación a nivel ESL, comparando su funcionalidad con la de la aplicación original, se debe realizar la primera fase de síntesis, que traducirá el diseño a una descripción a nivel RTL en VHDL o Verilog. Existen parámetros objeto de optimización a la hora de sintetizar el diseño, entre los cuales se destacan la latencia , el área o la potencia. En muchos casos habrá que establecer una estrategia final ya que no es posible optimizar todos los parámetros de forma simultánea. Es decisión del diseñador, en función de sus requisitos, guiar la síntesis con el fin de obtener la solución deseada. Estas decisiones de optimización se incluirán en todos los niveles de síntesis, tanto de alto nivel y lógica, como en la síntesis física, durante la cual se mapean y colocan las células lógicas en los distintos recursos del dispositivo sobre el que se implementa.

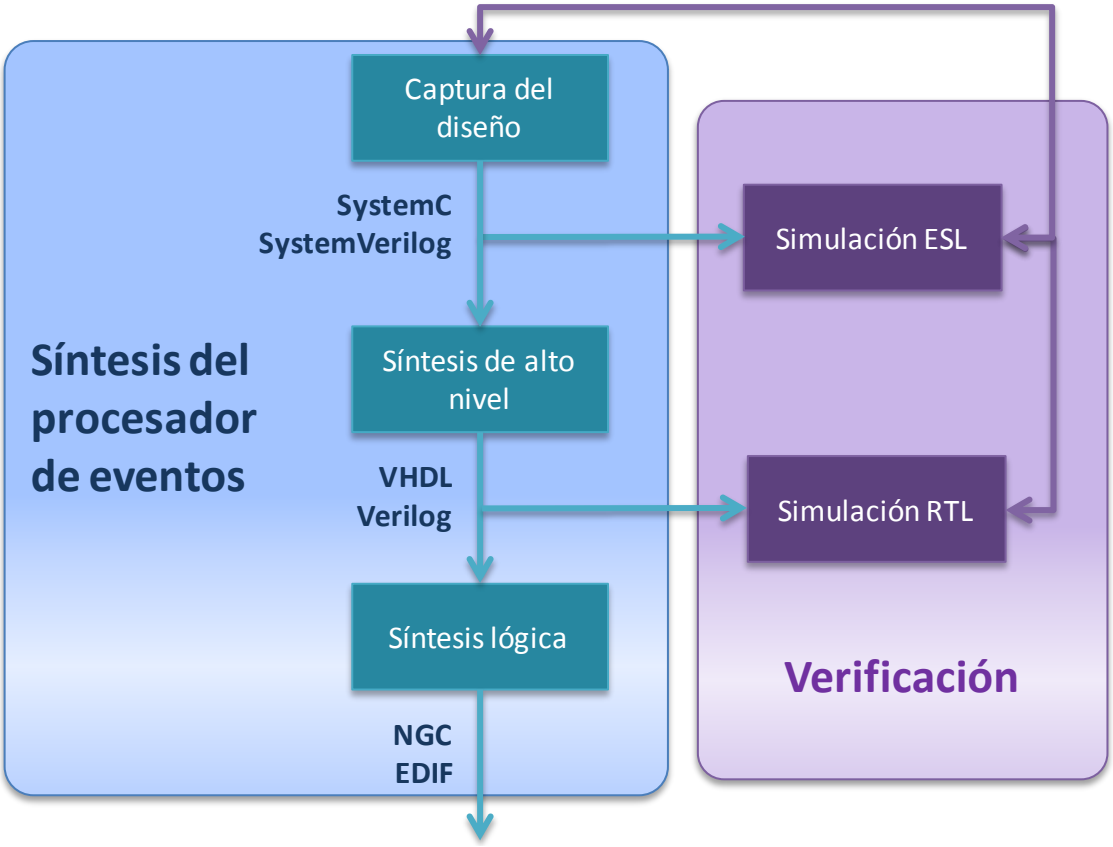


Figura 16. Síntesis del procesador de eventos.

En este momento deberá comprobarse que la solución obtenida tras la síntesis inicial mantiene la funcionalidad especificada inicialmente. En caso de disconformidades se modificará la descripción de alto nivel, de forma que la solución obtenida por la herramienta coincida con su diseño de entrada.

Una vez obtenida una descripción a nivel RTL que cumpla con las especificaciones, se deberá realizar una síntesis lógica que la traduzca a un diseño compuesto por la interconexión de células básicas de la librería de la tecnología de implementación con la que se trabajará. Esta fase tiene por objetivo la obtención de un *netlist* (fichero de instancias de células básicas y sus conexiones), generalmente NGC o EDIF. Los parámetros objeto son por una parte la determinación del ciclo de diseño y por otra la optimización del área/recursos utilizados.

2.3 Diseño de la plataforma

El diseño de la plataforma *hardware* comienza con la creación del sistema, fase en la que se instanciarán los diferentes bloques necesarios, configurando aquellos atributos parametrizables. En este paso se importará también el dispositivo periférico diseñado por el usuario (el coprocesador de eventos o el bloque de *echo*, según sea el caso) y se conectará al resto del sistema [23].

Una vez llegados a este punto, se deberá seguir el flujo necesario para la implementación sobre una FPGA, es decir, sintetizar el diseño (la mayoría de los bloques instanciados se disponen

en formato HDL), mapearlo sobre los elementos disponibles en la tecnología sobre la que se implementa el diseño, para luego decidir la colocación de cada uno de estos elementos y su ruta de interconexión.

Una vez que la plataforma *hardware* está disponible, se diseñará el software del sistema, eligiendo el sistema operativo sobre el que se ejecutará la aplicación, así como aquellas librerías a incluir para dar soporte a las comunicaciones necesarias. A ello se le añade la aplicación de usuario, en este caso la aplicación que recibe los eventos por la interfaz de entrada al sistema, que en el caso que nos ocupa se trata de tramas TCP/IP, los trata si fuera necesario, y los envía al periférico diseñado anteriormente. A continuación, la aplicación software, así como las librerías asociadas, deberán compilarse y enlazarse para generar un fichero ejecutable de tipo ELF que se ejecutará en el microprocesador de la plataforma.

Con el fin de validar el correcto funcionamiento del sistema se deberá descargar el fichero de configuración de la FPGA generado tras la implementación, así como el ejecutable para el microprocesador. Hecho esto, se enviarán eventos a través de la interfaz Gigabit Ethernet y se esperarán las tramas de salida del mismo.

2.4 Verificación

La fase de verificación se realizará a distintos niveles de abstracción, así como a distintas vistas del diseño. Así, se realiza la verificación funcional del sistema tanto en su descripción ESL como a nivel RTL, verificación física tras su implementación, así como verificación de las restricciones temporales tras un análisis del *slack* de todas las conexiones. Finalmente se realiza el prototipado del sistema, verificando su funcionalidad en tiempo real.

3 Tecnologías

Se presentan a continuación las tecnologías de implementación con las que se trabajará en este PFC.

3.1 Familia de FPGA Xilinx Virtex-5

La familia Virtex-5 de Xilinx usa la segunda generación de la arquitectura basada en columnas ASMBL™ (*Advanced Silicon Modular Block*), estando dividida en cinco sub-familias con el fin de cubrir todas las posibles necesidades de los diseños a implementar. Además de las células lógicas programables, las FPGAs de la familia Virtex-5 contienen una gran cantidad de bloques de tipo Hard-IP (bloques *hardware* no programables embebidos en la FPGA) como pueden ser, BlockRAMs y FIFOs de 36Kbit, DSPs 24 x 18, tecnología SelectIO™ con impedancia controlada digitalmente (DCI), bloques de interfaz ChipSync™ source-synchronous, funcionalidad de monitorización del sistema, generadores de reloj interno mediante DCMs (Digital Clock Manager) y PLLs (Phase-Locked-Loop) y opciones avanzadas de configuración, incluyendo reconfiguración parcial. Otras funcionalidades de la FPGA están soportadas por la presencia de bloques

transceptores de alta velocidad y bajo consumo, bloques integrados de interfaz PCI Express®, MACs (*Media Access Controllers*) Ethernet tri-modo (10/100/1000 Mbps) y microprocesadores PowerPC® 440 de altas prestaciones. Está fabricada con tecnología de 65 nm [24].

3.1.1 Resumen de características de la familia Virtex-5

A continuación se listan las principales aspectos de la familia de FPGAs Virtex-5 de Xilinx. Se referencia al lector a la hoja de características para una descripción más completa [24].

- Cinco subfamilias:
  - Virtex-5 LX: Para aplicaciones de lógica general.
  - Virtex-5 LXT: Para aplicaciones de lógica general que requieran conectividad serie avanzada.
  - Virtex-5 SXT: Para aplicaciones de procesado de señales que requieran conectividad serie avanzada.
  - Virtex-5 TXT: Para sistemas que requieran conectividad serie avanzada de doble densidad.
  - Virtex-5 FXT: Para sistemas empotrados con conectividad serie avanzada.
- Compatibilidad entre plataformas:
  - Las familias LXT, SXT y FXT usan encapsulados compatibles, pudiéndose reutilizar diseños de PCBs modificando únicamente el voltaje.
- Estructura de lógica de altas prestaciones:
  - Tecnología de LUTs de 6 entradas.
  - Opción de LUTs de 5 entradas duales.
  - Rutas reducidas mejoradas.
  - Opción de RAM distribuida de 64 bits.
  - LUTs con registros de desplazamiento de 32 bits / Opción dual de LUTs con registros de desplazamiento de 16 bits
- Gestión de reloj CMT:
  - Bloques DCM para retardo de buffer nulo, síntesis de frecuencia y desfase de reloj.
  - Bloques PLL para filtrado de jitter, retardo de buffer nulo, síntesis de frecuencia y división de reloj enganchado en fase.
- Bloques FIFO y RAM integrada de 36 Kbit:
  - Bloques de RAM de dos puertos.
  - Lógica de FIFO programable.
  - Programable:
    - Anchos de x36 para memorias de dos puertos.
    - Anchos de x72 para memorias de un puerto.
  - Circuitería interna opcional de corrección de errores.
  - Opcionalmente se puede programar cada bloque como dos bloques independientes de 18 Kbit.
- Tecnología SelectIO™:
  - Operaciones de entrada/salida con voltaje de 1,2 V a 3,3V.
  - Interfaz con tecnología source-synchronous ChipSync™.

    - Impedancia controlada digitalmente (DCI)
    - Bancos de E/S de granuralidad fina.
    - Soporte para interfaz de memoria de alta velocidad.
- Slices DSP48E:
    - Multiplicadores 24 x 18 en complemento a dos.
    - Sumador, restador y acumulador opcional.
    - Segmentación opcional.
    - Funcionalidad de lógica a nivel de bits opcional.
    - Conectividad en cascada dedicada.
- Opciones de configuración flexibles:
    - Interfaces SPI y *Parallel FLASH*.
    - Soporte *multi-bitstream* con lógica de reconfiguración dedicada.
    - Detección automática de ancho de bus.
- Capacidad de monitorizar el sistema en todos los dispositivos:
    - Monitorización de temperatura *On-chip*/*Off-chip*.
    - Monitorización de tensión de alimentación *On-chip*/*Off-chip*.
    - Acceso por JTAG a todos los valores monitorizados.
- Bloques *Endpoint* integrados para diseños de PCI Express®.
    - Subfamilias LXT, SXT, TXT y FXT.
    - Acorde con la especificación base v1.1 de PCI Express®.
    - Soporte en cada bloque de pista x1, x4 o x8
    - Funcionalidad compartida conjuntamente con los transceptores RocketIO™.
- MACs Ethernet tri-modo 10/100/1000 Mbps.
    - Subfamilias LXT, SXT, TXT y FXT.
    - Los transceptores RocketIO™ pueden ser usados como PHY o conectados a PHY externos usando las diferentes soluciones MII (*Media Independent Iterface*).
- Transceptores RocketIO™ GTP con tasas de 100 Mbps a 3,75 Gbps.
    - Subfamilias LXT y SXT.
- Transceptores RocketIO™ GTX con tasas de 150 Mbps a 6,5 Gbps.
    - Subfamilias TXT y FXT.
- Microprocesadores PowerPC 440:
    - Solo la subfamilia FXT.
    - Arquitectura RISC.
    - Pipeline de 7 etapas.
    - *Caches* de instrucciones y de datos de 32 KB.
    - Estructura de interfaz de procesador optimizada.
- Tecnología de 65 nm de cobre.
- Voltaje de núcleo de 1,0 V.
- Encapsulados *Flip-Chip* estándar y opciones libres de plomo.

3.1.2 Arquitectura

A continuación se describe la arquitectura interna de algunos recursos existentes en las FPGAs de la *familia* Virtex-5 y que son de interés para el presente trabajo.

3.1.2.1 Configurable Logic Blocks (CLBs)

Los CLBs son los recursos lógicos principales que permiten la implementación lógica de circuitos tanto combinacionales como secuenciales. Cada CLB está conectado a una matriz de interconexión local, que a su vez lo conecta con la matriz general de interconexión. Un elemento CLB (figura 17) está formado por una pareja de *slices*, las cuales no tienen conexión entre ellas, y cada uno de los *slices* están organizados por columnas. Cada columna tiene su propia cadena de conexión en cascada. Para cada CLB, el *slice* en la parte inferior se etiqueta como SLICE(0), y los de la parte superior como SLICE(1) [25].

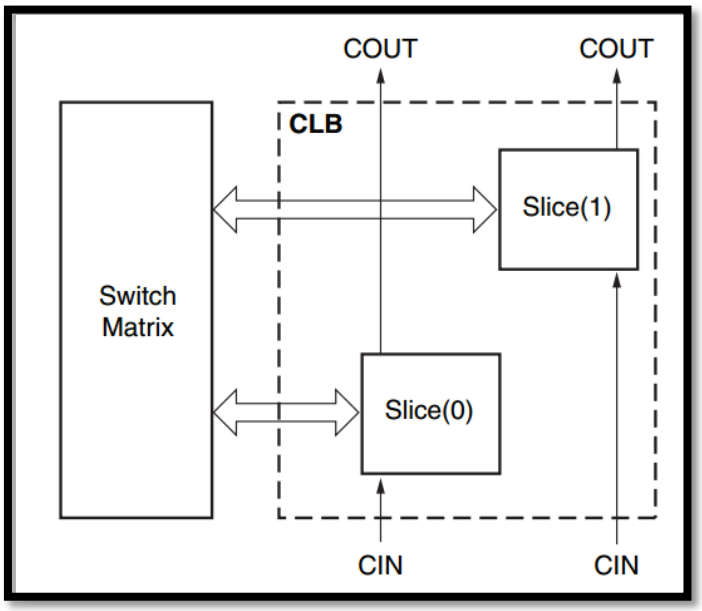


Figura 17. Arquitectura de un CLB.

Con el fin de identificar los slices de toda la FPGA, estos se etiquetan como una pareja de coordenadas, precedidas por una X o una Y si se refiere a la posición horizontal o vertical. Esto se puede observar en la figura 18.

Cada slice tiene cuatro generadores de funciones lógicas implementadas como tablas o LUTs, cuatro elementos de almacenamientos (flip-flops o registros de un bit), multiplexores con múltiple funcionalidad y lógica de acarreo. Estos elementos son usados por todos los *slices* para proveer de funciones ROM, aritméticas y lógicas. Estos *slices* son llamados SLICEL (figura 20). Existen otros *slices* que, además, soportan funcionalidades adicionales: almacenar datos usando RAM distribuida y desplazamiento de datos con registros de 32 bits de ancho. Estos se conocen

como SLICEM (Figura 19). A continuación se muestra la estructura interna de cada uno de estos *slices* [25].

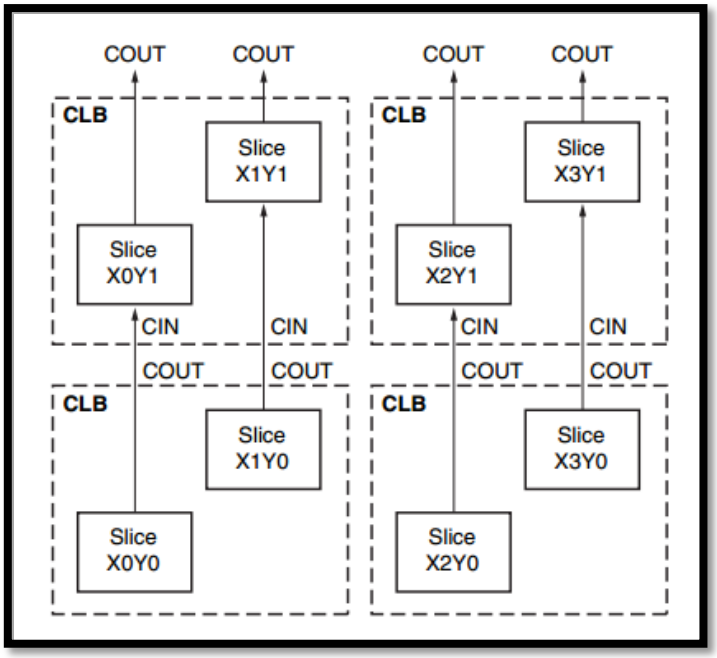


Figura 18. Etiquetado de *slices*.

3.1.2.2 Block RAMs (BRAMs)

Además de la memoria distribuida disponible en los *slices* de tipo SLICEM y de las interfaces de memoria SelectIO™, los dispositivos de la familia Virtex-5 tienen un gran número de bloques de memoria RAM de 36 Kb. Cada bloque de 36 Kb puede configurarse como dos bloques independientes de 18 Kb. Las BRAMs están colocadas en columnas dentro de la FPGA, y su número depende de la subfamilia y el tamaño del dispositivo. Los bloques de 36 Kb pueden conectarse en cascada para formar bloques de más capacidad con una mínima penalización temporal [25].

Los módulos de memoria de doble puerto de 36 Kb consisten en un área de almacenamiento de 36 Kb, además de dos puertos completamente independientes etiquetados como A y B. La estructura es totalmente simétrica, y ambos puertos son intercambiables (figura 21).

Los datos pueden ser escritos en cada puerto de forma independiente o en ambos y pueden ser leídos de igual forma. Cada operación de escritura es síncrona, cada puerto tiene su puerto de direccionamiento, su puerto de datos de entrada y su puerto de datos de salida, su reloj y sus señales de control.

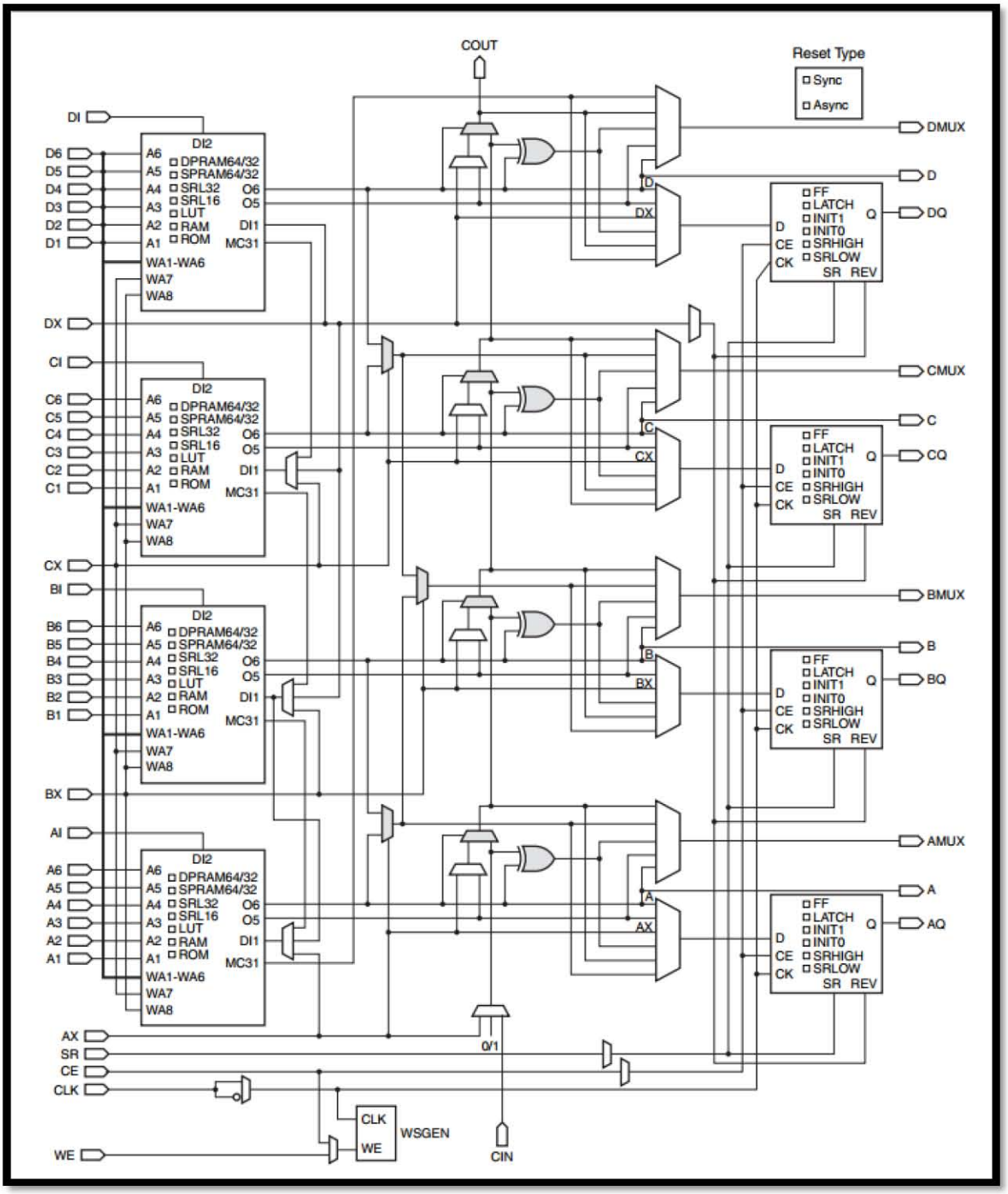


Figura 19. Estructura interna de un SLICEM.

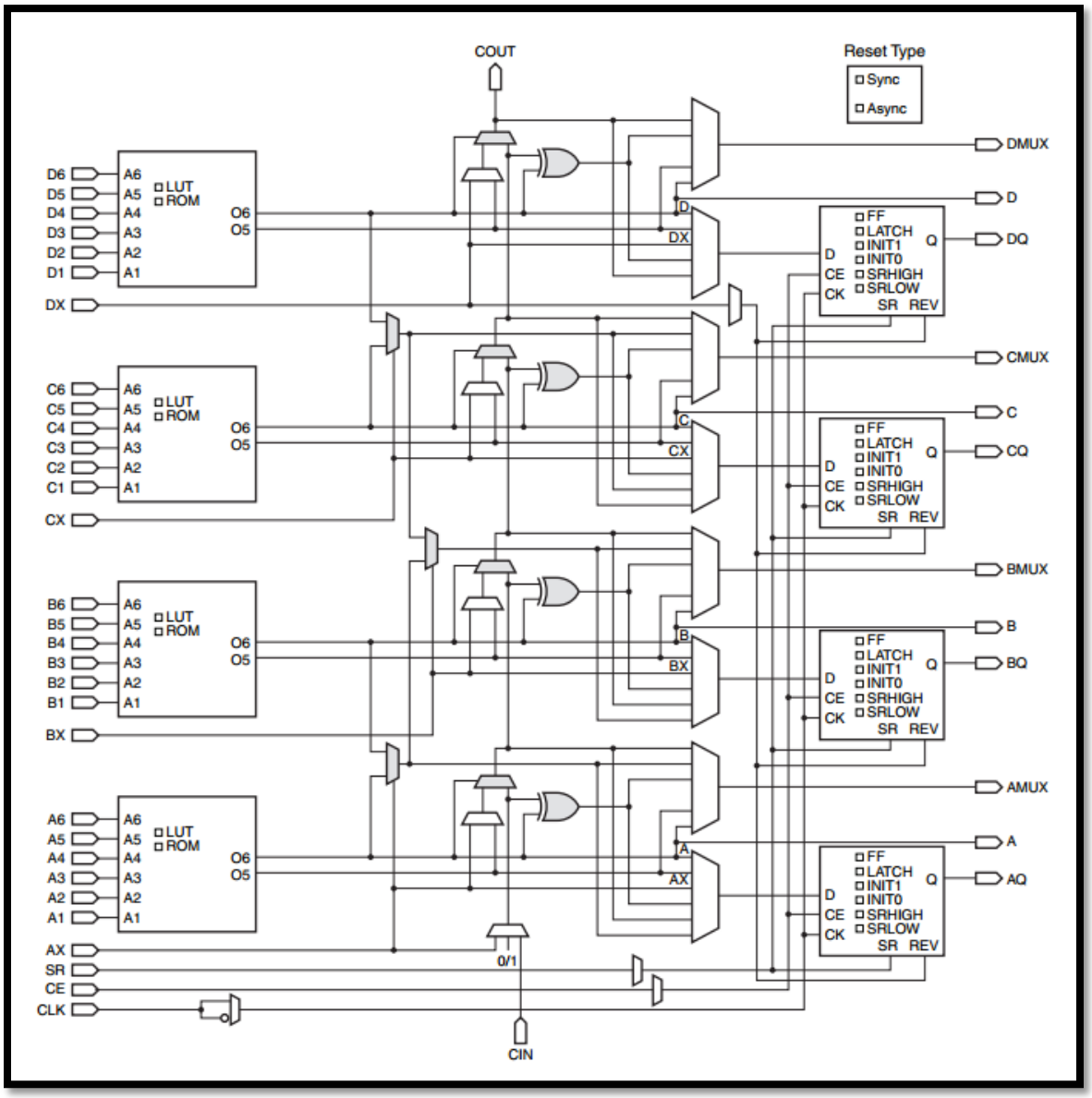


Figura 20. Estructura interna de un SLICEL.

En estos módulos de memoria no existe ningún tipo de comprobación que evite el intento de escritura en la misma dirección por parte de ambos puertos, por lo que es responsabilidad del diseñador que hace uso de ellos evitar que esta situación. En caso de que ocurra esta escritura simultánea, no existirá daño físico, pero sí se pueden corromper los datos.

Tabla 1. Descripción de los puertos de una BRAM.

Nombre del puerto	Descripción
DI[A|B]	Bus de datos de entrada
DIP[A|B]	Bus de datos de entrada de paridad, que pueden ser usados para ampliar el bus de datos de entrada.
ADDR[A|B]	Bus de direcciones
WE[A|B]	Señal de control de escritura a nivel de byte

Nombre del puerto	Descripción
EN[A|B]	Señal de control de activación de la memoria. Si está desactivado no se escriben datos y se lee siempre el último dato leído.
SSR[A|B]	*Set*/*Reset* síncrono
CLK[A|B]	Señal de reloj
DO[A|B]	Bus de datos de salida
DOP[A|B]	Bus de datos de salida de paridad, que pueden ser usados para ampliar el bus de datos de salida.
REGCE[A|B]	Señal de control del registro de salida.
CASCADEINLAT[A|B]	Pin de entrada de cascada para el modo 64K x 1 cuando la opción de registrar la salida está desactivada.
CASCADEOUTLAT[A|B]	Pin de salida de cascada para el modo 64K x 1 cuando la opción de registrar la salida está desactivada.
CASCADEINREG[A|B]	Entrada de cascada para el modo 64K x 1 cuando la opción de registrar la entrada está activada.
CASCADEOUTREG[A|B]	Salida de cascada para el modo 64K x 1 cuando la opción de registrar la salida está activada.

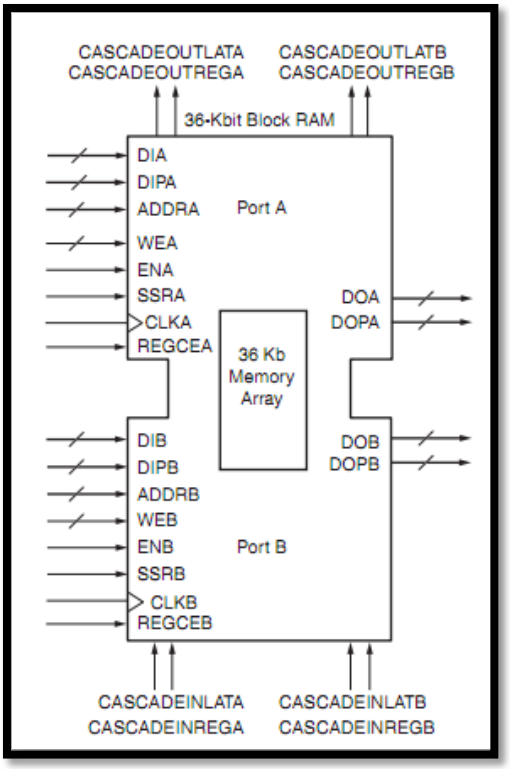


Figura 21. Puertos de una BRAM de doble puerto.

Las lecturas en modo *latch* tienen un retardo de un único flanco de reloj. La dirección es registrada en el puerto de lectura, y el dato almacenado en memoria se carga en los *latches* de salida tras el tiempo de acceso de la RAM. Cuando la opción de registrar la salida está activa, la lectura tiene un ciclo de reloj adicional de retardo (el dato es válido un flanco de reloj más tarde, pero el tiempo de acceso se reduce del tiempo de una RAM al de un registro).

Las escrituras se realizan en un único ciclo. La dirección se registra en el puerto de escritura y el dato se almacena en memoria. Existen tres modos de escritura, para definir qué ocurre si se lee y se escribe en la misma posición de memoria en el mismo ciclo: *WRITE_FIRST*, *READ_FIRST* y *NO_CHANGE*.

- *WRITE_FIRST*

Es el modo por defecto. En él, si se lee y escribe en el mismo ciclo en la misma dirección de memoria, el dato es tanto almacenado en memoria como en el puerto de salida. En la figura 22 se representa el comportamiento temporal de la memoria en el que se muestra cómo el mismo dato a escribir en memoria, se escribe en el puerto de salida en el siguiente flanco de reloj.

A efectos prácticos, este modo equivale a que los datos se escribiesen durante la primera mitad de un ciclo de reloj, y se leyesen durante la segunda mitad, hecho que da nombre al modo de escritura.

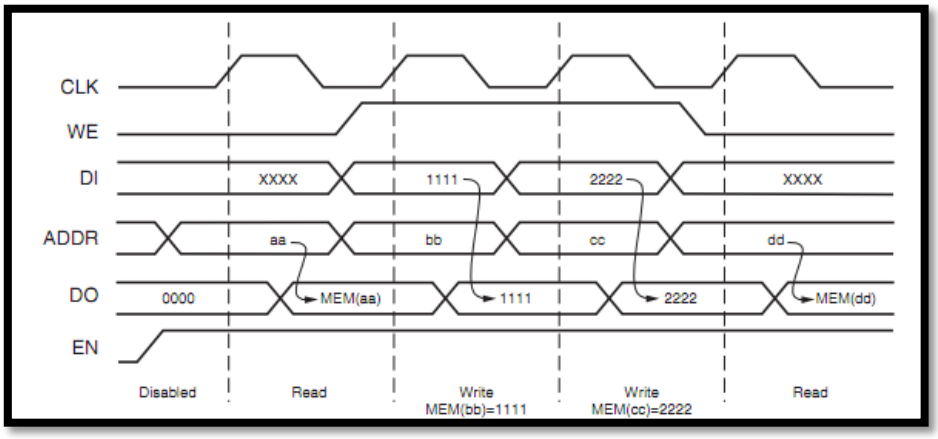


Figura 22. Modo de escritura *WRITE_FIRST*.

Como puede observarse en el tercer ciclo de reloj, etiquetado como "*Write MEM(bb)=1111*", en el puerto de salida de datos se lee el mismo valor que se almacena en ese mismo ciclo desde el puerto de entrada de datos, es decir un valor *1111*. Lo mismo ocurre en el siguiente flanco de reloj con el valor *2222*.

- *READ_FIRST*

Al contrario que ocurría con el modo *WRITE_FIRST*, este modo equivale a que la lectura se realice durante la primera mitad del ciclo de reloj y la escritura en la segunda mitad. Esto produce que, en el instante en que se escribe en memoria, se leerá el valor anterior almacenado en esa misma posición, en lugar del dato que actualmente se estuviese escribiendo.

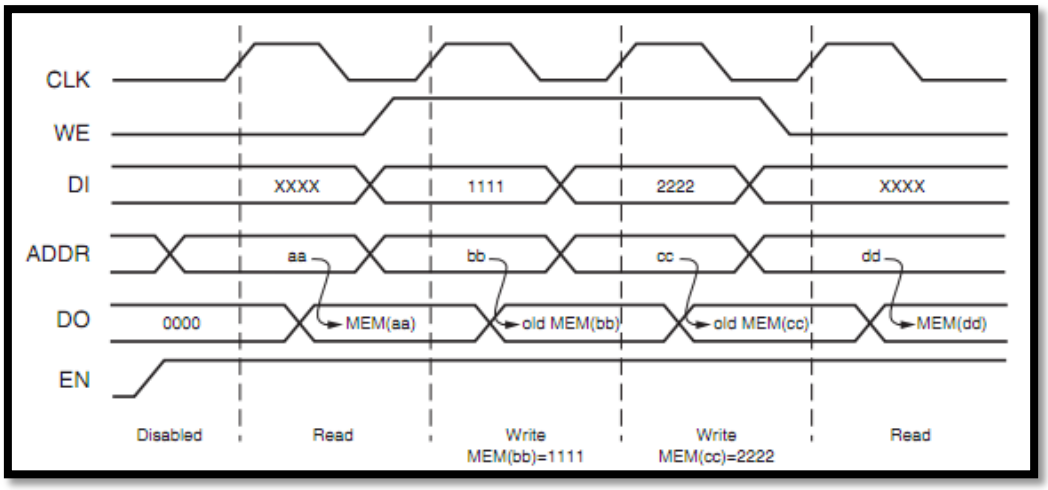


Figura 23. Modo de escritura *READ_FIRST*.

En este caso ocurre que, para unos mismos valores en los puertos de entrada, los valores que se leen en los puertos de salida en los mismos instantes que para el modo anterior no corresponden a los que actualmente se están escribiendo, sino a los valores que existían en esa dirección de memoria hasta el instante anterior (representados como "*old MEM(bb)*" y "*old MEM(cc)*").

- *NO_CHANGE*

Este último modo de escritura pretende evitar que puedan realizarse escrituras y lecturas simultáneamente. Con esto se consigue que, durante los ciclos en los que se están realizando escrituras en memoria, los valores en el puerto de salida de datos permanece estático, es decir, el último valor leído.

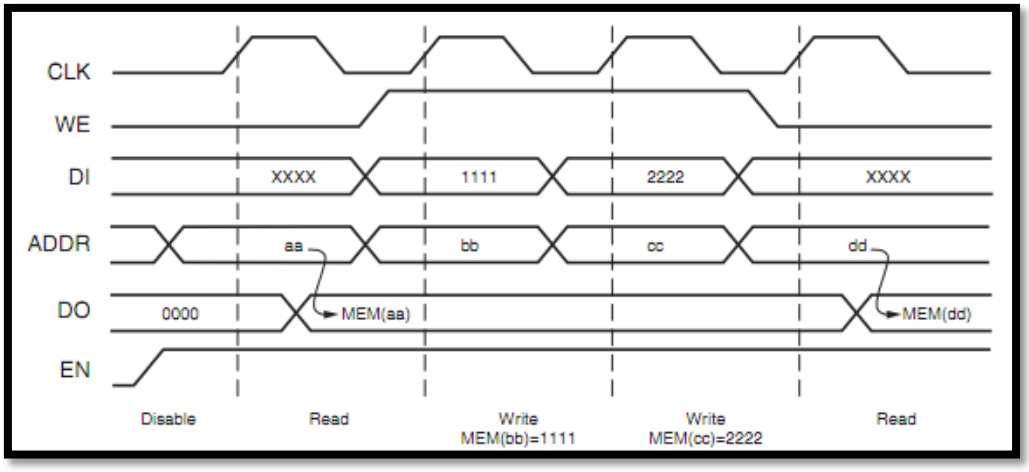


Figura 24. Modo de escritura *NO_CHANGE*.

Puede observarse como, durante los dos ciclos de escritura, en las direcciones *bb* y *cc*, el dato leído en el puerto de salida de datos corresponde a la lectura del ciclo anterior, es decir, "*MEM(aa)*".

Con el fin de eliminar el tiempo de acceso a la memoria de la ruta de datos desde la BRAM hasta la lógica adicional de cómputo, estos bloques de memoria permiten registrar la salida, a costa de una latencia de un ciclo de reloj más. Este tipo de accesos son muy útiles para diseños con segmentación, que requieren de frecuencias altas de funcionamiento, pero tienen una latencia de ejecución mayor en número de ciclos.

Otra de las características de estas memorias es la lógica añadida para implementar FIFOs con ellas sin necesidad de usar CLBs para ello, ya que está integrada de forma *hardware*.

3.1.2.3 DSP48E

Existen, entre los recursos disponibles en las FPGAs de la familia Virtex-5, unos *slices* especiales dedicados a realizar operaciones aritméticas lentas y de gran coste de recursos si se realizaran con los CLBs. Entre sus funciones están la multiplicación, la multiplicación y acumulación, la multiplicación y suma, la suma de tres operadores, *barrel shifter*, entre otros [26].

Se permite, además, realizar varias operaciones lentas con un único slice, mediante multiplexado en tiempo.

Estos DSP de segunda generación soportan entradas de 24 x 18 bits para la multiplicación, aunque, en caso de ser necesarias entradas mayores, se pueden conectar en cascada para realizar las mismas operaciones sobre datos más anchos.

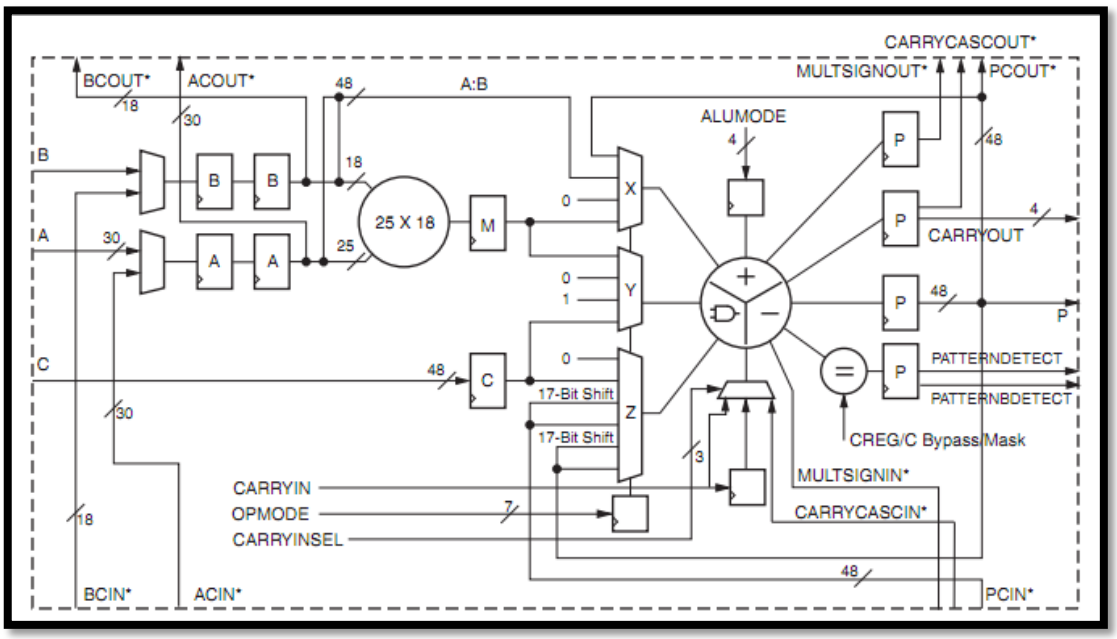


Figura 25. Estructura de un *slice* DSP48E[1].

[1] Las señales marcadas con (*) están dedicadas a la conexión de la columna de DSP y no son accesibles a través del resto de la lógica programable.

3.2 Kits de desarrollo ML507 y ML510

Xilinx proporciona kits de desarrollo que incluyen placas de prototipado, herramientas de diseño, etc. de cara a la realización del prototipado del sistema. Estas placas pueden incluir una o varias FPGAs, aparte de distinta funcionalidad adicional de cara a su utilización en diferentes dominios de aplicaciones.

Para este proyecto se han utilizado dos placas de prototipado, ambas soportando a la familia de FPGAs Virtex 5 FX. En concreto so se han utilizado las placas ML507 y la ML510.

La placa ML507 [27] incluye la FPGA XCV5FX70T. El diagrama de bloques del dispositivo se muestra a continuación.

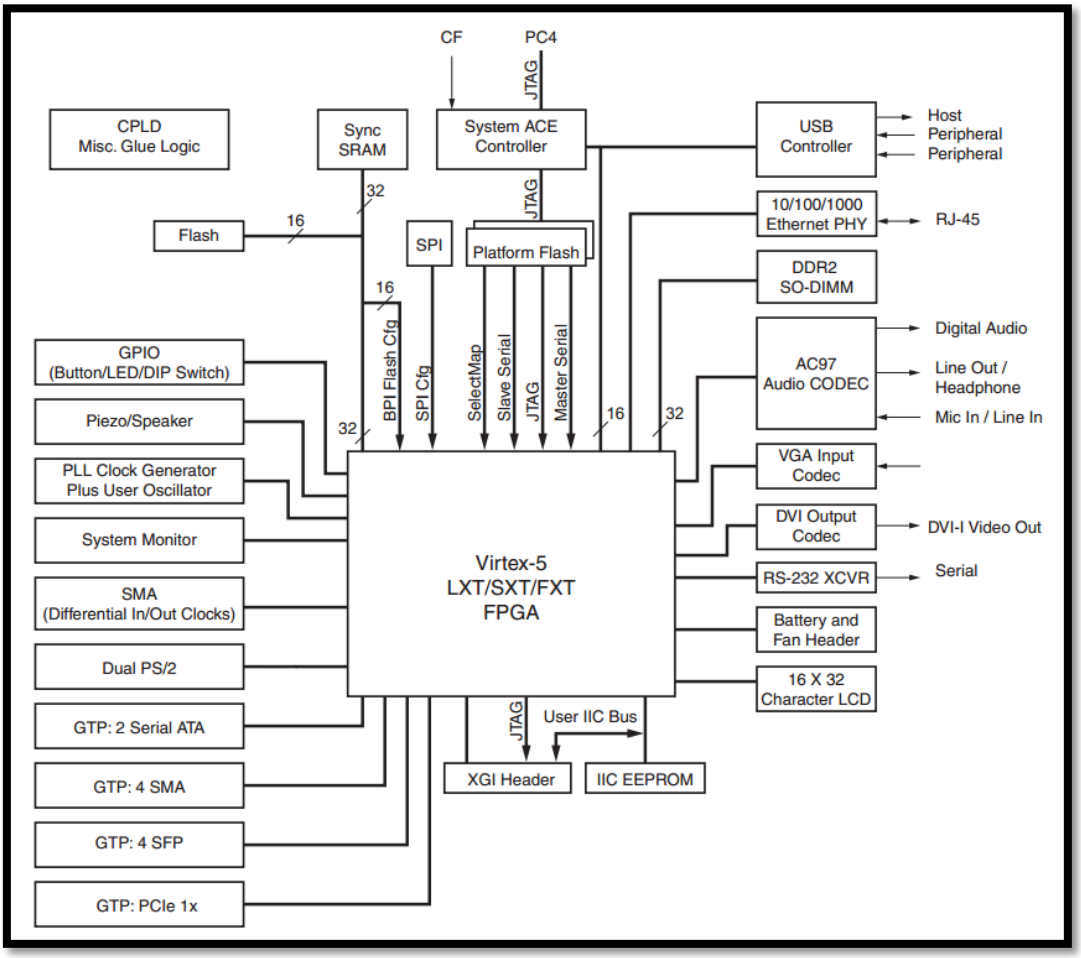


Figura 26. Diagrama de bloques de la placa ML507.

Entre los recursos que se representan, de relevancia para el presente proyecto son el controlador a nivel físico de Ethernet, la memoria RAM DDR2, el controlador SystemACE así como el controlador RS-232 para comunicación serie.

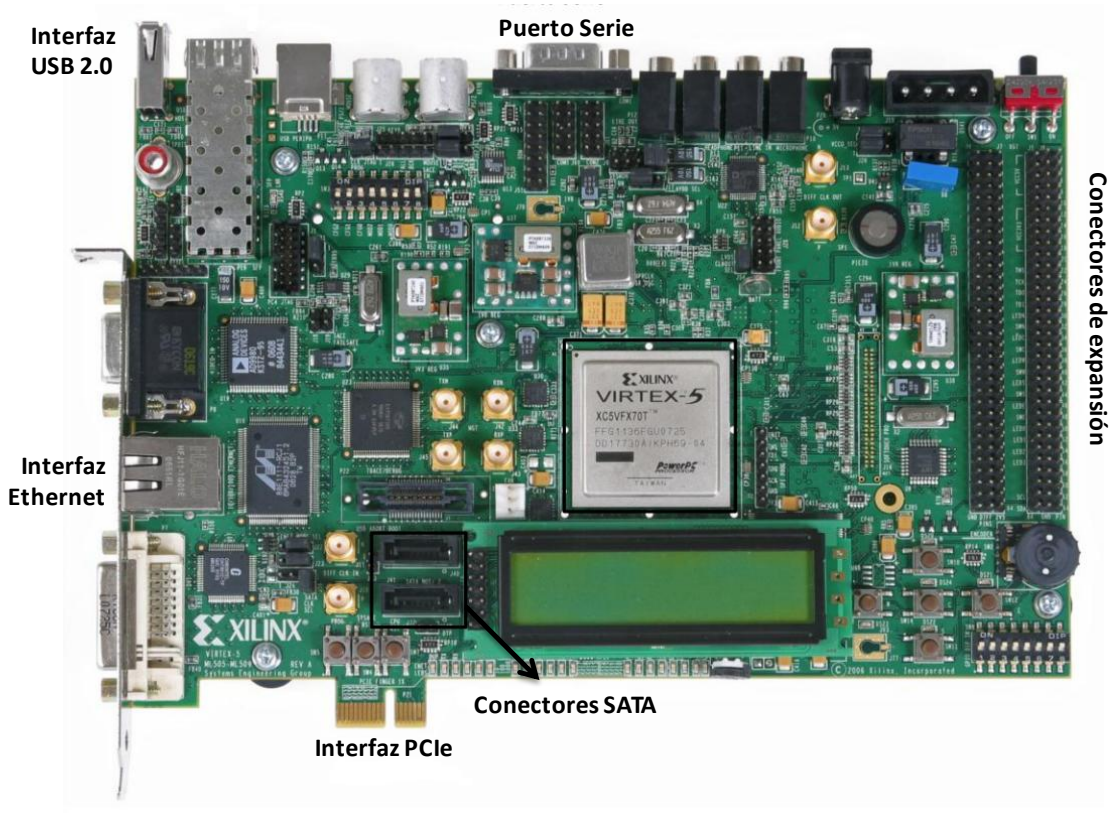


Figura 27. Placa de prototipado ML507 (superior).

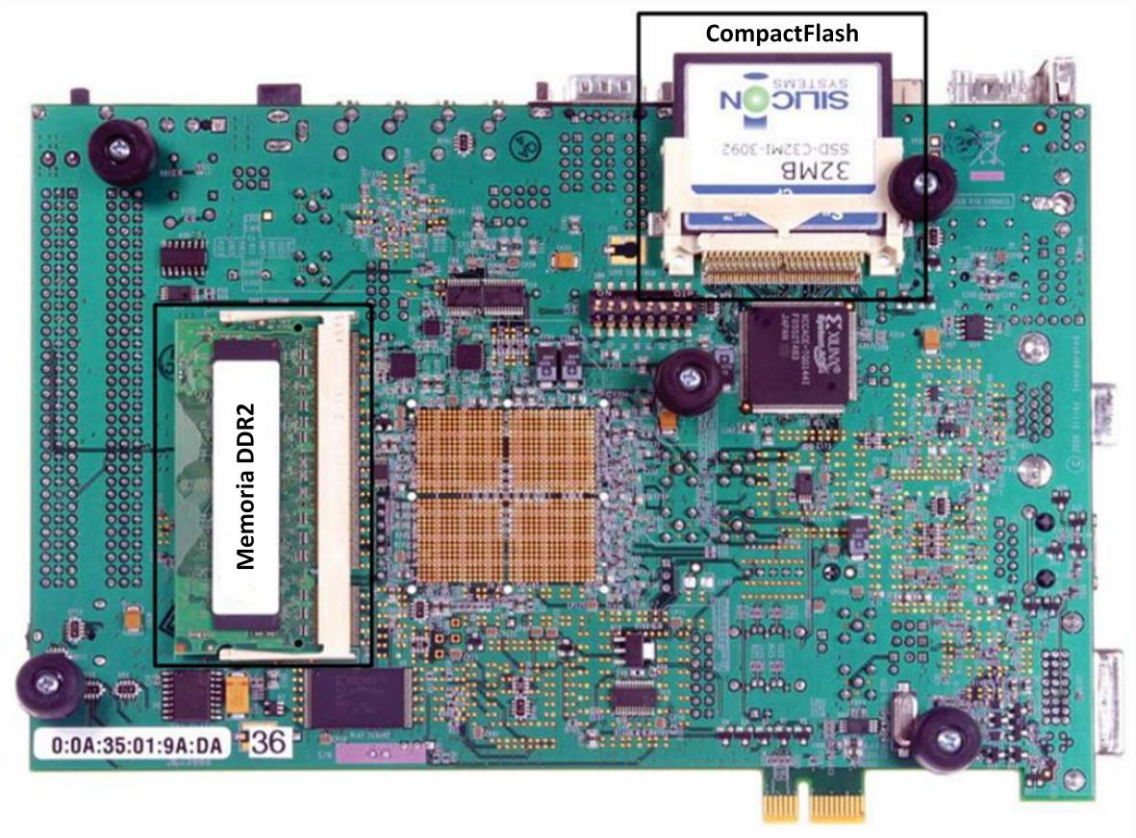


Figura 28. Placa de prototipado ML507 (inferior).

En la figura 27 y en la figura 28 muestra dicha placa, tanto de la cara superior como inferior. En la primera de ellas puede identificarse el dispositivo programable FPGA, la interfaz serie, la interfaz Ethernet, el puerto de expansión, las interfaces SATA y PCI Express®. En la inferior se encuentran identificadas la memoria DDR2 y la memoria flash con una tarjeta en formato CompactFlash.

La segunda placa utilizada es la ML510 que contiene una FPGA XCV5FX130T [28]. Esta placa tiene un formato tipo placa base ATX, incluyendo varios slots PCI y PCI-X. Esta placa cubre las necesidades del diseño en cuanto a interfaces planteadas como pueden ser la de Gigabit Ethernet, puerto serie, y acceso a SystemACE, etc.

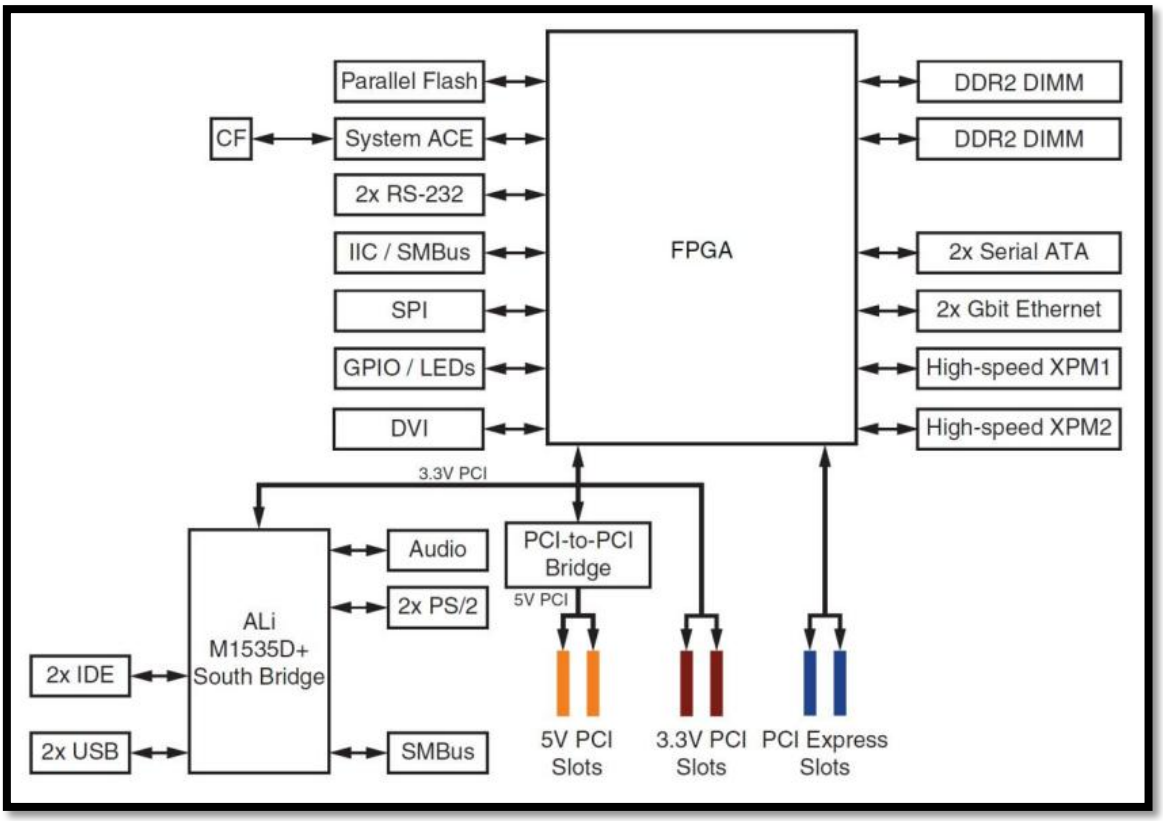


Figura 29. Diagrama de bloques de la placa ML510.

La placa presenta las siguientes características:

- Placa base en formato ATX, con alimentación compatible ATX.
- Dos DIMMS DDR2, cada uno de 512 MB y 72 bits de ancho.
- Tarjeta CompactFlash de 512 MB y controlador compatible SystemACE.
- Chip de memoria flash Intel P30 StrataFlash de 256 MB.
- Dos conectores RJ-45 con sus controladores a nivel físico PHY compatibles con 10/100/1000 Ethernet.
- Interfaz PCI y PCI Express.

- Dos UARTs con conectores RS-232.
- Interfaces HID: DVI, LEDs, pantalla LCD, interruptores, botones, PS/2 y conectores de audio (entrada y salida).
- Dos puertos USB.
- Dos puertos SATA.
- Conector especial (XPM) de alta velocidad para los transceptores RocketIO GTX, SPI4.2, GPIO y alimentación.
- Puertos de depurado y programación JTAG.
- Transceptores de alta velocidad a través de interfaces RocketIO GTX.
- Interfaz IIC/SMBus.
- EEPROM SPI.

A continuación se muestra una vista en planta de la placa de prototipado, y de su frontal de conexiones (trasera de la caja donde se instala).

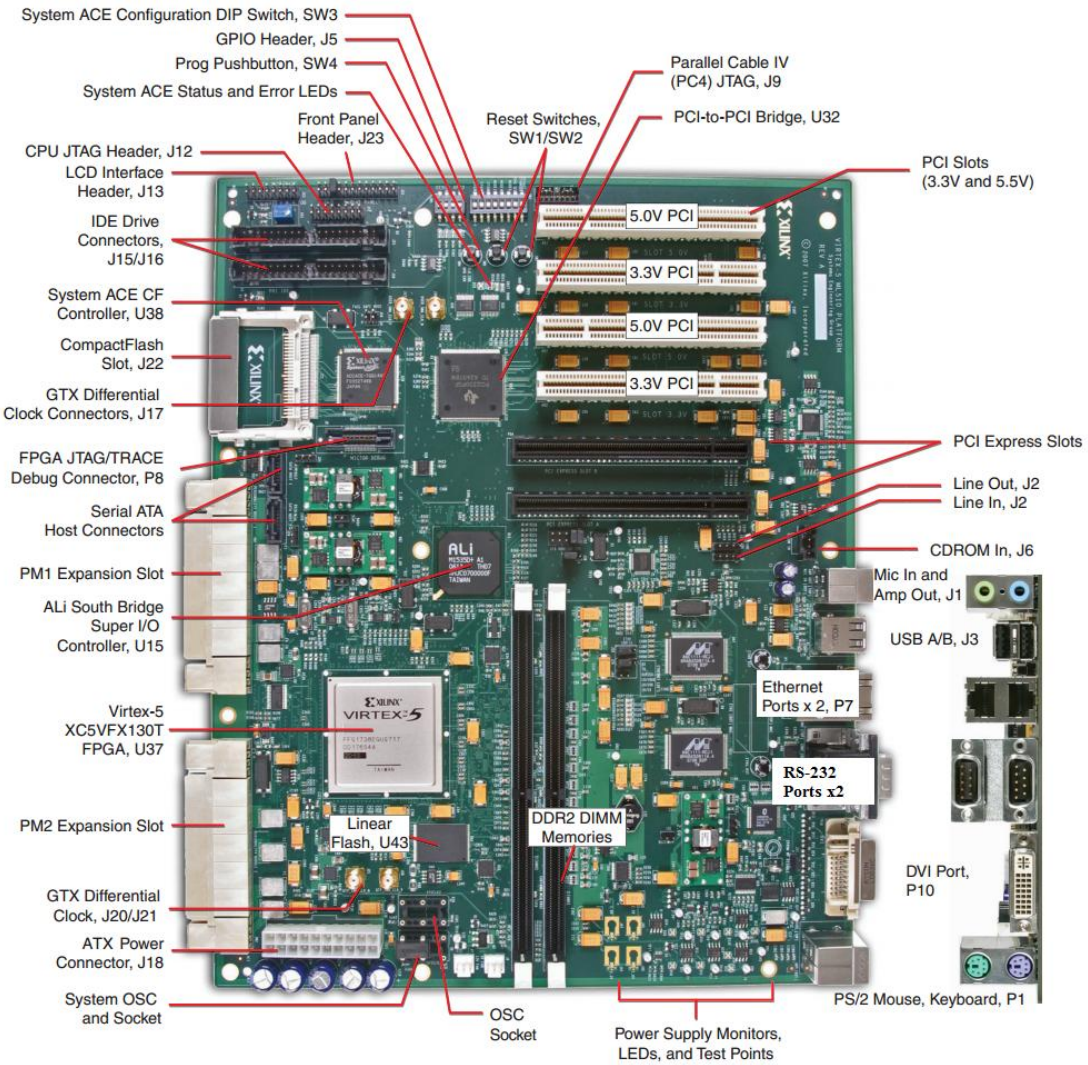


Figura 30. Placa de prototipado ML510.

3.3 FPGAs XC5VFX70T y XC5VFX130T

Los dos dispositivos FPGA utilizados en el proyecto son de la serie Virtex-5 FX. En concreto han sido el XC5VFX70T y el XC5VFX130T [24]. En la tabla 2 se resumen las principales características de cada uno de ellos, en cuanto a número de *slices*, bloques DSPs, memorias de bloque, núcleos procesadores *hard-core* y otros recursos de interés. Unos de los factores a tener en cuenta de cara a la elección del dispositivo es que el factor de utilización de los recursos del sistema depende de la naturaleza del diseño. Por ejemplo, no es posible maximizar el número de LUTs y de registros a la vez.

Tabla 2. Recursos de las FPGAs usadas.

Dispositivo	CLBs		DSP48E	BRAMs		CMTs	PowerPC	EMACs
	Fil x Col	Slices		Bloques	Max Kb			
XC5VFX70T	160 x 38	11200	128	148	5328	6	1	4
XC5VFX130T	200 x 56	20480	320	298	10728	6	2	6

4 Lenguajes

Debido a la creciente complejidad de los diseños de sistemas electrónicos, se hace necesaria la aparición de nuevos métodos para realizar la captura del diseño. En la actualidad, hay una tendencia cada vez mayor a la descripción del sistema electrónico a nivel de sistemas (ESL) [29], por lo que no han tardado en aparecer lenguajes de descripción hardware con un nivel de abstracción mayor tales como son SystemC y SystemVerilog.

4.1 SystemC

Es el lenguaje usado en el presente PFC para realizar la descripción *hardware* del periférico diseñado. Se detallan a continuación algunos conceptos de interés para la correcta comprensión de los siguientes capítulos así como las innovaciones que aporta este lenguaje en las metodologías de diseño *hardware*.

Para profundizar en las características de SystemC se recomienda la lectura de [30], el manual de referencia de la última versión de la librería, así como [7] con guías de diseño y ejemplos.

4.1.1 Definición

SystemC es una librería de clases de C++ desarrollada conjuntamente con una nueva metodología de diseño y verificación funcional para describir en alto nivel *hardware* a nivel de sistemas.

El uso de SystemC en conjunto con las herramientas estándar de desarrollo para C++ tiene como objeto poder modelar a nivel de sistema el diseño electrónico, y poder validarlo con los menores costes temporales posibles, además de conseguir un modelo que cumpla las especificaciones con el fin de que la siguientes fases de diseño puedan usarlo como referencia de funcionalidad.

La razón de elegir C++ como lenguaje de alto nivel para el desarrollo de la librería SystemC es que se trata de un lenguaje de alto nivel muy extendido, además de que sus compiladores generan códigos muy eficientes.

SystemC proporciona aquellas características que no están soportadas por C++, necesarias para el desarrollo de sistemas electrónicos:

- **Noción temporal**: C++ no permite conocer los instantes lógicos en los que se producen los eventos.
- **Concurrencia**: C++, y en general cualquier lenguaje orientado al diseño de *software* no está preparado para describir sistemas concurrentes, algo que es intrínseco en el diseño de *hardware*.
- **Tipos de datos**: Los tipos de datos que soportan los lenguajes de programación no contemplan factores necesarios en el *hardware* como son el valor de alta impedancia Z en una señal.

La librería de SystemC, a través de la definición de nuevas clases de objetos en C++, da solución a estas necesidades sin la necesidad de redefinir sintácticamente el lenguaje. Esto sumado a que se permite insertar código en C y C++ en su diseño, permite realizar una coverificación HW/SW.

4.1.2 Elementos principales

A continuación se detallan algunos elementos de relevancia en el modelado de sistemas con SystemC.

4.1.2.1 Módulos

Es la clase usada por SystemC para modelar la estructura del diseño, dando soporte a conceptos tales como jerarquía, interconexión, etc.. Un módulo encapsula un componente del diseño, que puede contener a su vez un conjunto de módulos interconectados a través de canales.

```
SC_MODULE (adder_reg) {
  …
};
```

4.1.2.2 Puertos

En cada módulo podrán definirse puertos de entrada, de salida y bidireccionales, con el fin de interconectar módulos entre sí, o si fuese el módulo jerárquicamente superior, para representar

las entradas y salidas del diseño. Además del sentido del puerto, cada uno podrá tener un tipo de datos, que podrá ser cualquier tipo de datos soportado por C/C++, tipos definidos por el usuario, o tipos de la librería SystemC.

```
sc_in<bool> clk;
sc_in<sc_int<8> > a;
sc_in<sc_int<8> > b;
sc_out<sc_int<9> > c;
```

4.1.2.3 Señales/canales

Al igual que los puertos sirven para comunicar, pero en lugar de módulos la comunicación se realiza entre procesos. Las señales son internas a cada módulo, y no son visibles desde otros módulos externos al contenedor de la señal en cuestión. Como los puertos, los datos pueden ser de tipos de C/C++, de usuario o de SystemC.

```
sc_signal<sc_int<9> > temp;
```

4.1.2.4 Procesos

En ellos se describe las funcionalidad de un módulo, que consistirá en un código escrito en C/C++ haciendo uso tanto de funciones del lenguaje raíz, como de métodos de las clases de la librería SystemC. Existen tres tipos de procesos: SC_METHOD, SC_THREAD, SC_CTHREAD.

SC_THREAD

Son procesos que solo se ejecutan una vez al comienzo de la ejecución del sistema. Sin embargo, permiten suspender el proceso con el fin de ser reanudado en el mismo estado en el que fue suspendido cada vez que se produce un evento en su lista de sensibilidad. Cada vez que la ejecución del proceso encuentre una secuencia *wait();* entrará en suspensión. En general, los procesos de este tipo suelen definirse como un bucle infinito, de forma que cuando termine vuelva a ejecutarse, y se define su latencia como el número de eventos necesario para una ejecución del bucle. Puede usarse para definir la ejecución de un camino de datos segmentados, por ejemplo.

A continuación se muestra un ejemplo de declaración, definición, registro y lista de sensibilidad de un proceso de tipo SC_THREAD. Sin embargo, se recomiendo que la definición del mismo se encuentre separado en un fichero *.cpp, y el resto en un fichero de cabecera *.h [31].

```
SC_MODULE (adder_reg) {
…
void reg();
…
void adder_reg::reg() {
 while (true) {
 …
 wait();
 …
   }
}
…
```

```
SC_THREAD(reg);
sensitive << clk.pos();
```

SC_CTHREAD

Se trata de de una modificación sobre los tipo SC_THREAD. Su principal diferencia es que en el registro se le añade un evento como sensibilidad, de forma que en la definición pueden realizarse nuevas formas de suspensión. Un ejemplo es la realización de una suspensión que se reanude un número de eventos más tarde, en lugar de en el siguiente evento. Otra es la posibilidad de suspender el proceso hasta que se cumpla una condición (y se genere el evento).

Estas formas de suspensión pueden también conseguirse con el proceso SC_THREAD como se muestra a continuación, pero requiere que el kernel reanude el proceso para volver a ponerlo en reposo en cada evento, haciendo la ejecución más lenta.

A continuación se muestran los modos de suspensión comentados, a la izquierda en su implementación para un SC_THREAD y su equivalente en un SC_CTHREAD a la derecha.

`for (i=0; i!=N; i++) wait();` `do wait() while(!expr);`	`wait(N);` `wait_until(expr.delay);`

SC_METHOD

Los procesos de tipo SC_METHOD se ejecutan cada vez que se produce un evento en su lista de sensibilidad y no pueden ser suspendidos. En general se usan para simular un comportamiento combinacional ya que se realiza en tiempo cero y no se suspende para volver a reanudarse. Al igual que en el tipo SC_THREAD se debe indicar en la lista de sensibilidad los eventos que provocarán la ejecución del método.

```
SC_MODULE (adder_reg) {
…
void add();
…
void adder_reg::add() {
  …
}
…
SC_METHOD(add);
sensitive << a << b;
```

A continuación se detalla un ejemplo de diseño de un sumador con la salida registrada en el que se hace uso de dos procesos en paralelo. Uno de ellos se encargará del proceso combinacional de realizar la suma y otro de registrar la señal [32].

```
#include "systemc.h"

SC_MODULE(adder_reg) {
  sc_in<sc_int<8> > a;
  sc_in<sc_int<8> > b;
```

```
  sc_out<sc_int<9> > c;
  sc_in<bool> clk;

  sc_signal<sc_int<9> > temp;

  void add() { temp = a + b; }
  void reg() { while (1) {c = temp; wait();} }

  SC_CTOR (adder_reg) {
    SC_METHOD(add);
    sensitive << a << b;

    SC_THREAD(reg);
    sensitive << clk.pos();
  }
};
```

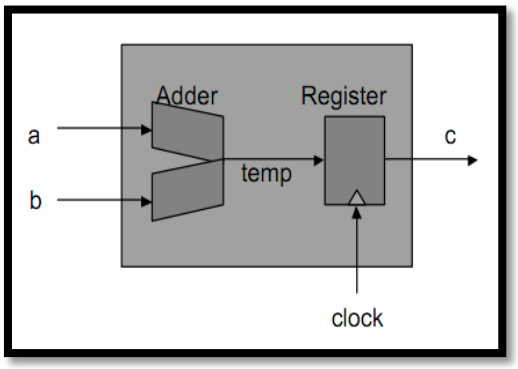


Figura 31. Diseño de ejemplo de uso de procesos en SystemC.

Como puede observarse en el ejemplo anterior, por un lado existe un proceso de tipo SC_METHOD que se ejecuta en tiempo cero cada vez que ase produzca un evento en alguno de los puertos de entrada (a y/o b), realizando la suma de ambas y almacenándola en *temp*. Por otro lado, en cada flanco de subida del reloj, el valor almacenado en *temp* se escribe en el puerto de salida *c*. Esta descripción cumple funcionalmente con el diseño propuesto en la figura 31, incluyendo que en un ciclo hayan varias modificaciones de las señales de entrada, almacenándose en la salida la última antes del flanco de subida del reloj (el proceso SC_METHOD se ejecutará, tantas veces como cambios haya en los puertos de entrada, pero solo se escribirá en el puerto de salida el valor que tuviese en el momento en el que se reanuda el proceso SC_THREAD).

4.1.2.5 Relojes

Los relojes son señales con una notación temporal, que permite dotar al sistema de relaciones temporales en su simulación. Se pueden definir varios relojes, con distintos tiempos de ciclo, ciclos de trabajo, así como fase relativa arbitraria.

```
sc_clock clk("clk", 10, SC_NS, 0.5, 0.0, SC_PS);
```

En el ejemplo anterior se ha declarado un reloj llamado *clk*, con un periodo de 10 nanosegundos, un ciclo de trabajo del 50% y un desfase de 0 picosegundos.

Las variables temporales en SystemC se almacenan como un entero de 64 bits y las unidades en las que se pueden representar son: *SC_FS*, *SC_PS*, *SC_NS*, *SC_US*, *SC_MS*, *SC_SEC*.

4.1.3 Características principales

A continuación se detallan algunas características fundamentales de SystemC que hacen de él una buena alternativa en el diseño *hardware* de alto nivel.

- **Simulación basada en eventos.** Al contrario que en una simulación por ciclos de reloj, la simulación basada en eventos presenta unos tiempos de simulación mucho más bajos, ya que no precisa de la ejecución de todos los procesos paralelos del sistema en cada ciclo de reloj, sino que pueden haber varios suspendidos en espera de que se cumpla uno de sus eventos.
- **Múltiples niveles de abstracción.** SystemC permite realizar modelos *untimed* con varios niveles de abstracción, desde funcional hasta RTL. Esto permite refinar iterativamente el código hasta llegar a un modelo RTL que cumpla la funcionalidad y usarlo como fuente para pasar a un diseño en VHDL o Verilog y de ahí seguir el flujo estándar de diseño electrónico.
- **Trazado de formas de onda.** Con SystemC es también posible el trazado de formas de ondas en los formatos VCD, WIF e ISDB.

4.1.4 Estructura de capas

La figura 32 muestra la arquitectura del lenguaje SystemC. Los bloques centrales son el núcleo de SystemC. Como puede observarse, está construido sobre el lenguaje C++ [7].

Las capas superiores, son librerías y estándares de diseño que el usuario puede decidir si usarlos o no. Gracias a esta estructura, se pueden seguir añadiendo librerías sobre el núcleo sin tener que modificar este.

El núcleo del lenguaje está formado por el simulador orientado por eventos, junto con los elementos comentados anteriormente (puertos, módulos, procesos, etc.). Los tipos de datos son necesarios para el modelado *hardware* y ciertos tipos de desarrollo *software*. Los canales primarios son los canales incorporados que tienen un amplio uso tal como las señales y las FIFOs.

4.1.5 Metodología de diseño

Como ya se indicó anteriormente, SystemC representa, además de una librería de clases, una metodología de diseño que tiene como objetivo acelerar las fases de diseño a nivel funcional así como su verificación.

En un flujo tradicional de diseño electrónico es una práctica habitual realizar un diseño funcional en C/C++ con el fin de crear un modelo de alto nivel que cumpla las especificaciones tras varias iteraciones de diseño y optimización, para luego realizar una traducción a un lenguaje de descripción *hardware* a nivel RTL (VHDL o Verilog). Este tipo de metodología presente inconvenientes conocidos como pueden ser:

- Traducir el código C/C++ a VHDL/Verilog puede producir errores difíciles de depurar. Aún con un modelo en C/C++ que cumpla con las especificaciones, es difícil realizar una traducción sistemática.
- Caducidad del modelo C/C++. Una vez el diseño se realiza a nivel RTL, todas las optimizaciones se realizarán sobre este, lo que llevará a que el modelo en alto nivel quede desfasado.
- Imposibilidad de reutilizar los test de diseño. La naturaleza de los tests de una aplicación en C/C++ son muy diferentes a un *testbench* de un sistema *hardware*, lo que obligará a rediseñar los sistemas de test para la versión RTL del diseño.

La metodología que presenta SystemC pretende dar solución a los problemas anteriormente citados y se representa en el diagrama de flujo de la figura 33.

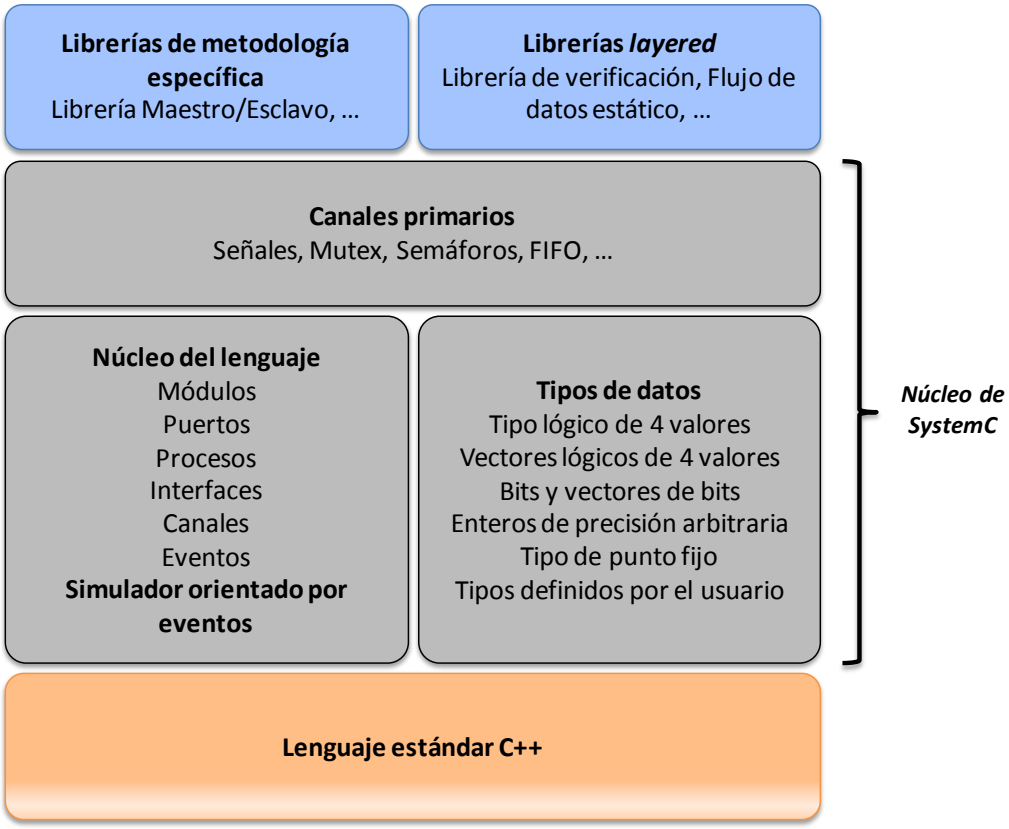


Figura 32. Arquitectura de capas de SystemC.

Las principales ventajas de este flujo modificado pueden resumirse en los siguientes puntos:

- La metodología de refinamiento permite escribir una primera versión del sistema en C/C++ e ir incluyendo características de SystemC al diseño en varias iteraciones.
- El realizar el sistema en SystemC para luego sintetizarlo hace que no sea necesario conocer otros lenguajes de descripción *hardware*.
- El tiempo de desarrollo se reduce, tanto en diseño como, lo que es más importante, en verificación.

- Los *testbenchs* pueden ser reutilizados desde la primera versión hasta la versión más refinada y cercana a RTL.

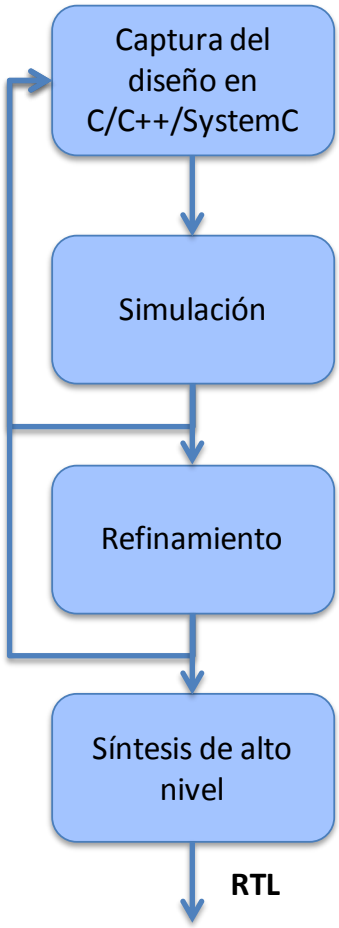


Figura 33. Metodología de SystemC.

Una parte importante de la metodología implica la verificación de que la solución obtenida sea equivalente a la especificada. En este caso deberíamos volver a realizar la verificación del sistema, adaptando el *testbench* inicial a los requerimientos de latencia presente en el modelo preciso a nivel de ciclos, obtenidos durante la síntesis de alto nivel. Otra forma de realizar la verificación es utilizar métodos de comparación formal entre las diferentes representaciones.

4.2 Verilog

Verilog es un lenguaje de descripción hardware (HDL) usado para diseñar circuitos electrónicos. En el presente PFC se ha utilizado Verilog como lenguaje a nivel RTL ya que la herramienta de síntesis de alto nivel realizará la transformación del diseño desde SystemC a Verilog. Es por ello que todo el diseño a nivel RTL está descrito en este lenguaje [33].

A continuación se muestran dos ejemplos de descripción RTL en Verilog. En la figura 34 se muestra la descripción RTL de un flip-flop tipo D, sensible al flanco de subida del reloj.

Asimismo, en el siguiente ejemplo se muestra una descripción de comportamiento mediante múltiples procesos concurrentes, el primero de ellos combinacional y el segundo secuencial.

```
module flipflop (d, clk, q, q_bar);
input d, clk;
output q, q_bar;
reg q, q_bar;

always @ (posedge clk)
begin
  q <= #1 d;
  q_bar <= #1 !d;
end
endmodule
```

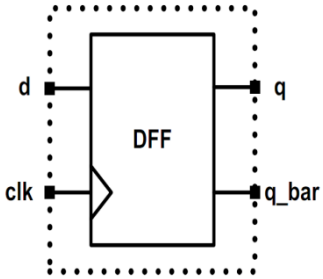


Figura 34. Flip-flip tipo D.

```
module (clk, A, B, C, D);
input   clk, A, B;
output  C, D;
reg     C;

always@(C)
begin
  D <= C + 10;
end

always@(posedge clk)
begin
  C <= A + B;
end
endmodule
```

4.2.1 Señales y puertos

En Verilog existen dos tipos básicos de señales:

- ***Nets* (*wire*):** representan una conexión directa entre dos puntos del circuito, es decir, es el equivalente a un cable. No almacena información más allá de la conexión.
- ***Registers*:** almacena información además de conectar puntos. No necesariamente una señal de tipo *reg* se sintetizará en un registro, sino que mantiene el estado de esa señal.

4.2.2 Puertos

Por otra parte, los puertos definen la interfaz de los módulos. Existen puertos de entrada y de salida y su tipo puede definirse. Las señales de entrada, *input*, no se declaran y serán siempre de tipo *wire*. Las salidas sin embargo, *output*, pueden estar registradas, por lo que habrá que especificar si es te tipo *wire* o *reg*.

4.2.3 Procesos

Un proceso consiste en un conjunto de sentencias que, ejecutándose de forma secuencial, describe el comportamiento de un circuito electrónico. Todos los procesos son ejecutados de forma concurrente entre ellos. Todas las asignaciones realizadas dentro de un proceso se hagan sobre señales de tipo *reg*. Existen dos tipos básicos de procesos en Verilog:

- ***Initial*:** son procesos que se ejecutan una vez al arranque del sistema y una única vez. No son sintetizables, por lo que solo tienen utilidad en el diseño de *testbenchs*.
- ***Always*:** permiten modelar procesos que se ejecutan en un bucle infinito. La condición de arranque del proceso definirse por un evento o por un tiempo determinado.

```
initial
begin
  clk = 0;
  reset = 0;
  enable = 0;
  data = 0;
end
```

```
always @(a or b or sel)
begin
  if (sel == 1)
      y = a;
  else
      y = b;
end
```

5 Herramientas

A continuación se detallarán las herramientas que se han utilizado durante el desarrollo de este PFC en sus distintas etapas.

5.1 Callgrind

Callgrind es una herramienta del entorno Valgrind para obtener el perfil de ejecución de un programa y así determinar su carga computacional, su requisito de recursos de memoria entre otros factores. Para ello crea el grafo de llamadas a partir del historial de llamadas a funciones durante su ejecución. La información que proporciona es, principalmente, el número de instrucciones ejecutadas en cada ejecución de una función así como las relaciones de llamador-llamado entre las distintas funciones, así como su número.

Callgrind propaga los costes de una función para sumarla al coste de la función que llama a otra, de tal forma que si una función *foo* llama a otra *bar*, el coste de *foo* será el de la ejecución de sus instrucciones, más el coste de bar. De esta forma, la función main deberá tener un coste cercano al 100%.

Al contrario que ocurre con otras herramientas para perfilar código como por ejemplo gprof, Callgrind no requiere recompilar la aplicación con una opción específica, sino que basta con lanzar la ejecución desde la propia herramienta. Para ello basta con lanzar el Callgrind indicándole el programa a ejecutar y sus argumentos [34].

```
valgrind --tool=callgrind [callgrind options] your-program
[program options]
```

La ejecución terminará con la escritura de un fichero de salida con el nombre de *callgrind.out.<pid>*. Esta información puede representarse como una tabla con una fila por cada función ejecutada y una columna de coste *inclusive* (coste de la función y de las funciones a las que llama), coste *self* (coste de las instrucciones de la función en cuestión, sin tener en cuenta las llamadas que hace) así como número de ejecuciones de la función.

```
callgrind_annotate [options] callgrind.out.<pid>
```

5.2 KCachegrind

KCachegrind es una herramienta de representación de los datos obtenidos por Callgrind. Esta aplicación es una solución gráfica para hacer más entendible al programador la información del perfil de la aplicación de estudio. Una vez lanzada, toma como entrada el fichero callgrind.out.<pid> generado por Callgrind y realiza una descripción gráfica de costes y jerarquía de llamadas de las funciones ejecutadas [35].

A continuación se muestra un ejemplo de uso para un código ejemplo que calcula el producto vectorial de dos vectores de mil elementos cada uno, inicializados mediante una función de librería *random*.

```
#include <stdlib.h>

int multiplicar (int op1, int op2) {
 return (op1 * op2);
}

int producto_vectorial(int *vector1, int *vector2, int longv) {
  int producto = 0;
  int i;
  for (i = 0; i < longv; i++) {
    producto += multiplicar(vector1[i], vector2[i]);
  }
  return producto;
}

void inicializar_vector(int *vector, int longv) {
  int i;
  for (i = 0; i < longv; i++) {
    vector[i] = rand() % 11;
  }
}

int main () {
 int vector1[1000];
 int vector2[100];
 int producto;
 inicializar_vector(vector1, 1000);
 inicializar_vector(vector2, 1000);
 producto = producto_vectorial(vector1, vector2, 1000);
}
```

En la figura 36 se muestra el grafo de llamadas obtenido a través de callgrind y visualizado con KCachegrind. En él puede distinguirse la jerarquía de llamandas a funciones, partiendo del proceso del kernel del sistema operativo que realiza la llamada a la función main. A partir de ahí se realizan llamadas a inicializar_vector y a producto_vectorial, las cuales cada una a su vez llaman a otras subrutinas. Además, en cada bloque representante de una función se muestra el coste computacional en porcentaje respecto al total de la aplicación.

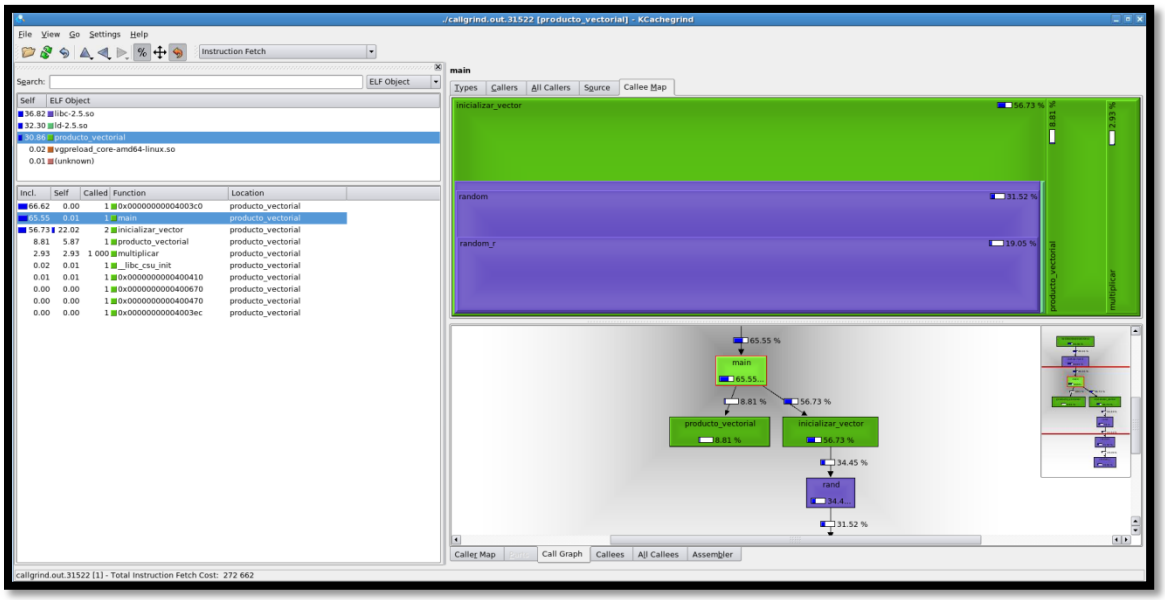


Figura 35. Interfaz de usuario de KCachegrind.

5.3 C-to-Silicon Compiler

Cadence CtoS (C-to-Silicon Compiler) es una herramienta de síntesis de alto nivel. Este tipo de herramientas, tiene como objeto reducir el tiempo de diseño en sistemas complejos, y en la mayoría de los casos, mejorar la calidad de los resultados, en comparación con las traducciones hechas a mano por los diseñadores.

CtoS genera una descripción RTL del diseño en el estándar de IEEE Verilog, de forma que el diseño puede seguir la ruta de síntesis mediante herramientas de síntesis lógica disponibles, ya sea la propia de Cadence (Encounter RTL Compiler) o cualquier otra de las que existen en el mercado (Xilinx Synthesis Technology, Synplify Pro/Premier, etc.). Además del código RTL, CtoS genera descripciones de comportamiento, tanto en SystemC como en Verilog, para su simulación funcional [36].

Existen dos modos de ejecución de la síntesis en CtoS, ya sea en modo interactivo mediante el uso de una interfaz gráfica y en modo *batch* mediante el uso de *scripts* en TCL. Normalmente, la primera vez que se sintetiza un diseño se hará en modo interactivo para facilitar la toma de decisiones de forma interactiva por parte del diseñador con la ayuda de las herramientas gráficas que posee el entorno. Cada decisión tomada en ella, generará una o varias órdenes de consola, que pueden consultarse para escribir el *script* de síntesis correspondiente.

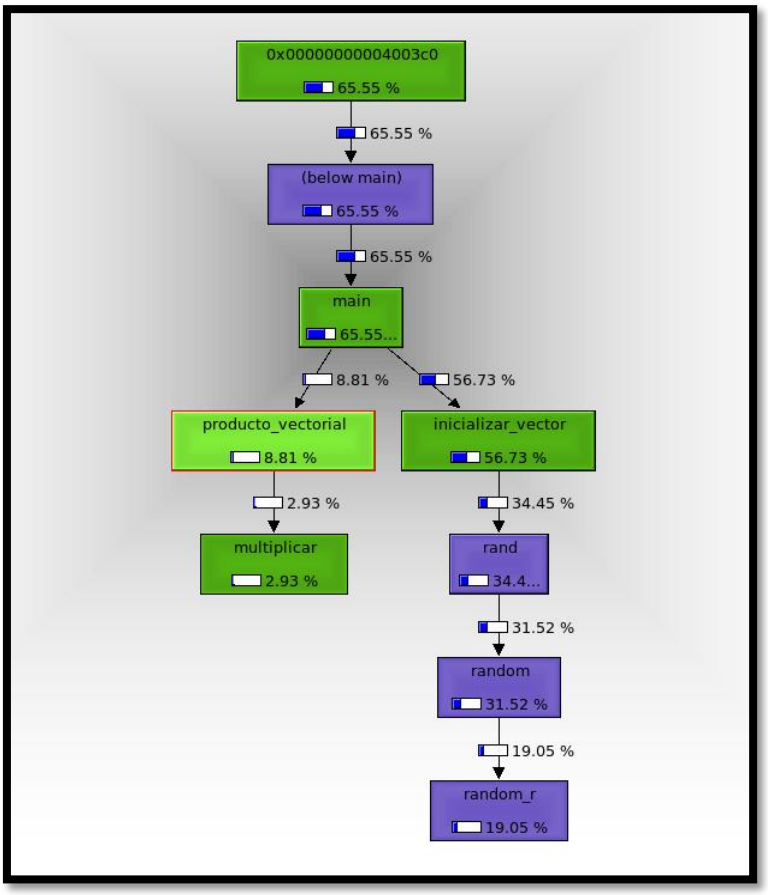


Figura 36. Grafo de llamadas en KCachegrind.

5.3.1 Flujo de diseño

En la figura 37 se representa el flujo de diseño en CtoS que incluye las etapas que se enumeran y describen a continuación.

1. Comienzo

El primer paso a realizar consiste en crear un nuevo diseño o abrir uno existente. Existen diferentes alternativas, ya sea el diseño basado en C/C++, SystemC o TLM.

2. Configuración del diseño

Aquí habrá que indicar si el diseño será SystemC, TLM o C/C++. Se definirá aquí la fuente de reloj, con sus propiedades de nombre, frecuencia, ciclo de trabajo, etc. Se incluirán los ficheros fuente, así como aquel que corresponde al *top* del diseño.

3. Especificación de la micro-arquitectura

En este apartado se realizan las transformaciones arquitecturales para hacer el diseño sintetizable. Los elementos a transformar son: funciones y bucles.

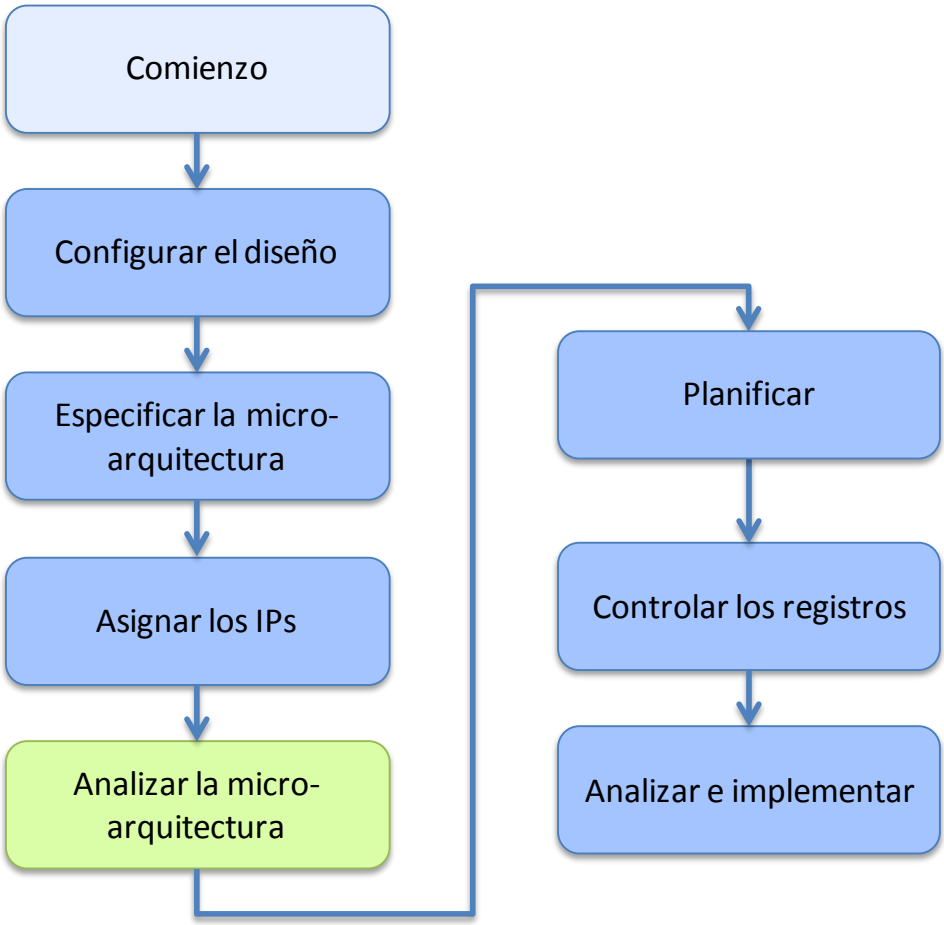


Figura 37. Flujo de diseño de CtoS.

Algunas funciones no son sintetizables por tener dos caminos diferentes con distinta latencia, lo cual impide al que hace la llamada conocer la espera que debe realizar. Otra causa puede ser que acceda a vectores donde se pueda escribir. Esto se soluciona haciendo las funciones "en línea" (inline), es decir, sustituir la llamada por el código que la describe.

Por otro lado, los bucles combinacionales (bucles que se realizan idealmente en tiempo cero) no son sintetizables, por lo que puede decidirse entre varias formas de solucionarlo:

- Desenrollar el bucle para ser realizado enteramente en tiempo cero. Esto puede ser inviable para bucles de muchas iteraciones, pues haría del bucle la ruta crítica con una latencia muy alta.
- Romper el bucle para que cada iteración se realice en un ciclo de reloj.
- Realizar una segmentación el bucle. Se divide la ejecución del bucle en un número determinado de etapas, de tal forma que cada iteración del bucle se encuentra en una etapa del mismo simultáneamente, paralelizando así varias iteraciones.

Decidir qué solución tomar es responsabilidad del diseñador, y deberá tener en cuenta restricciones tanto de área como de frecuencia (figura 38).

4. Asignar los IPs

En este paso se asignarán las variables de tipo vector/array del diseño a memorias internas de la FPGA, o a descripciones HDL de memorias.

5. Analizar la micro-arquitectura (opcional)

Llegados a este punto, el diseño está completamente transformado, a falta de que el CtoS realice las optimizaciones oportunas, por lo que puede realizarse un análisis temporal, de área y de potencia (menos exacto que el que se realiza tras el planificador) con el fin de modificar algunas de las decisiones tomadas anteriormente.

6. Planificar

En este punto, el planificador de CtoS asignará recursos a las operaciones del diseño fuente. En caso de que el planificador no pueda resolver esta tarea, se presentan algunas acciones que faciliten la obtención de una solución:

- Controlar la dependencia con vectores de datos. Esto hará que el planificador conozca las dependencias y pueda añadir estados para resolverlas.
- Controlar los estados. En caso de que el planificador no pueda resolver conflictos secuenciales, el diseñador puede añadir manualmente estados para guiar a CtoS a encontrar una solución a dichos conflictos.
- Controlar los recursos. El diseñador podrá *a priori*, crear unos recursos iniciales sobre los que mapear las operaciones del diseño.

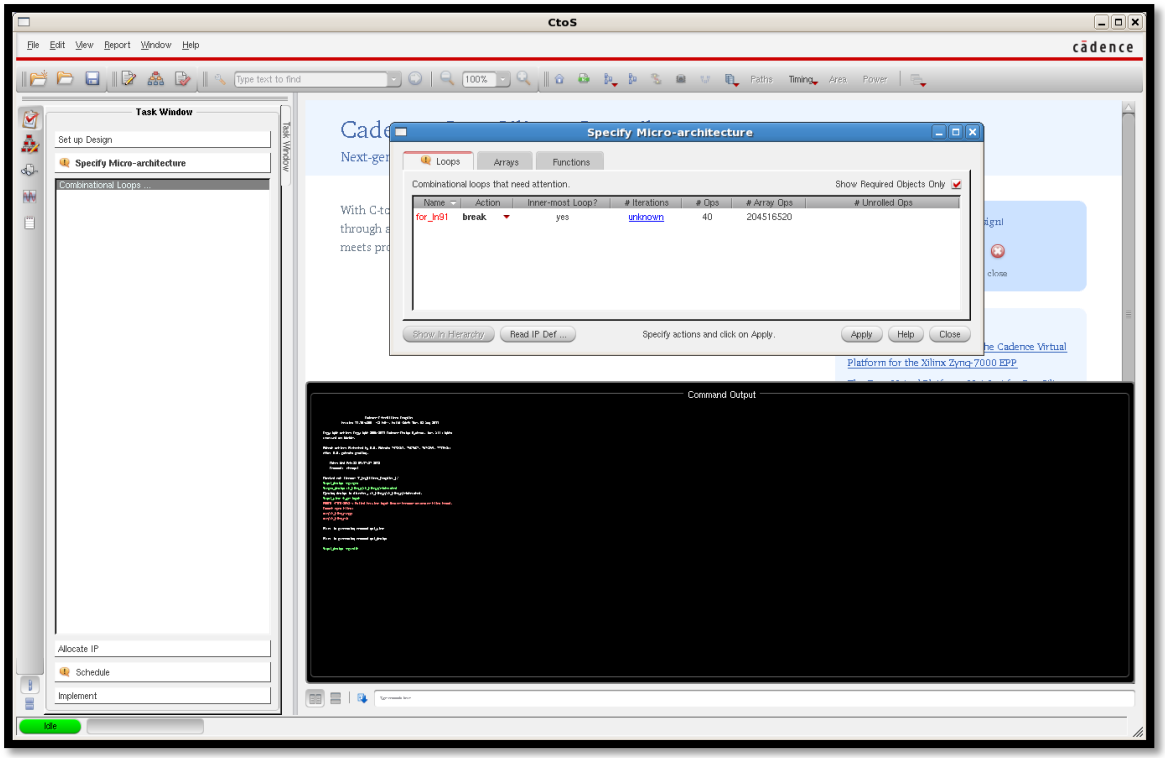


Figura 38. Etapa de especificación de micro-arquitectura en CtoS.

7. Asignación y control de registros

En esta etapa puede decidirse si incluir una lógica de *reset* adicional para aquellos registros que por defecto no lo tengan. También puede decidirse si registrar o no todas las salidas de los recursos de lógica combinacional asociados a operaciones del diseño. Aquí se debe decidir entre minimizar registros (decisión normalmente asociada a diseño ASIC donde el coste en área de los registros es muy alto) o minimizar el número de multiplexores (asociado normalmente al diseño basado en FPGAs).

8. Análisis e implementación

En este paso se podrán realizar informes de distinto tipo, tales como consumo de recursos, o de latencia en ciclos de reloj de cada bloque.

También se podrán generar aquí los ficheros de salida, como puede ser la descripción RTL, un *script* que realice la mismas acciones que las realizadas a través de la interfaz gráfica, o un fichero contenedor en SystemC de la descripción RTL, con el fin de poder realizar una simulación de esta, usando el *testbench* en SystemC con el que se verificó el modelo en alto nivel.

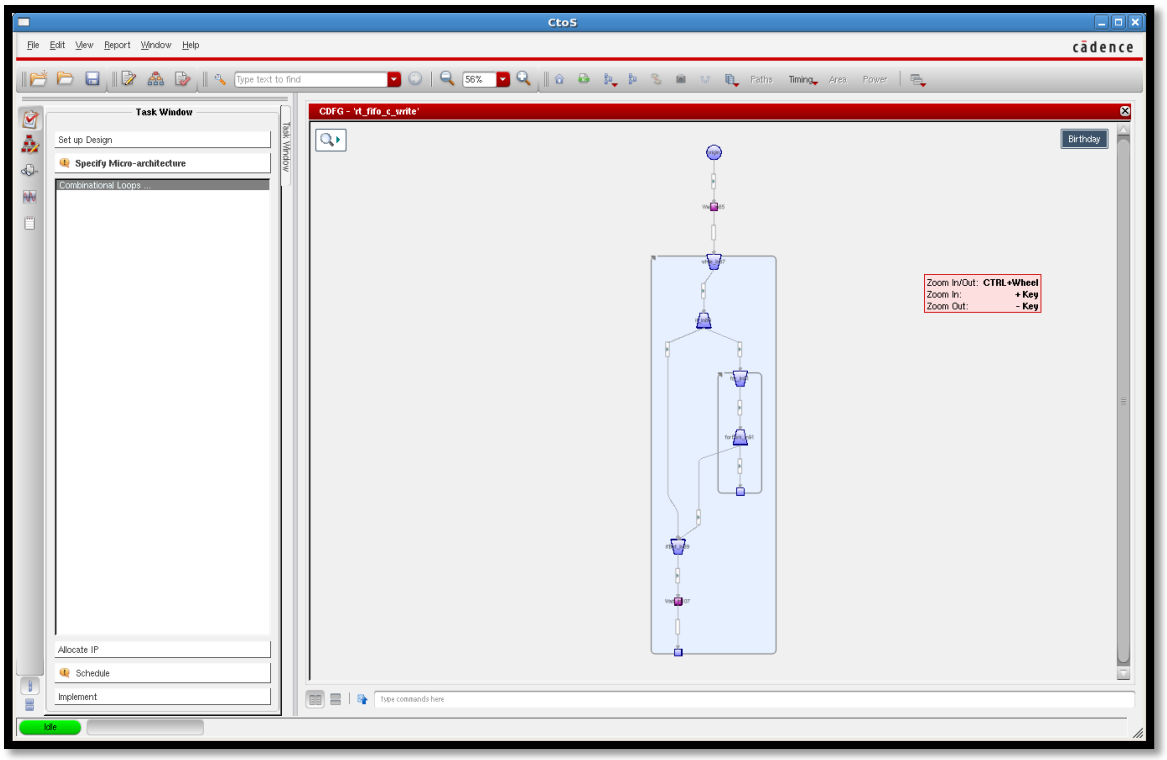


Figura 39. Grafo de control y datos en CtoS.

5.3.2 Características no soportadas de SystemC

Como se comentó en el apartado del lenguaje SystemC, este está pensando no solo para la realización de síntesis de alto nivel, sino también para descripciones puramente de comportamiento, y es por ello que muchas de sus funcionalidades no son sintetizables por las herramientas de síntesis de alto nivel. A continuación se enumeran algunas.

Tabla 3. Características no soportadas de SystemC en CtoS.

Tipo de características	Características no soportadas	
Clases del lenguaje	• get_child_objects • sc_process_handle • sc_event_and_list • sc_event_or_list • sc_event • sc_time register_port	• default_event • sc_fifo • sc_buffer • sc_mutex • sc_semaphore • sc_event_queuesc_clock
Tipos de datos	• get_child_objects • sc_generic_base	• sc_numrep • float
Utilidad de clases	• sc_trace • sc_report • sc_report_handler • sc_exception	• sc_copyright • sc_version • sc_release
Anclaje de puertos	Solo podrán hacerse estáticamente en el constructor del módulo.	
Punteros	Solo se podrán usar para apuntar a variables de forma que estáticamente cada puntero pueda saberse a que variable apunta.	

5.4 Synplify Premier

Synopsys Synplify Premier es una herramienta que la herramienta usada en este proyecto para la realización de la síntesis lógica. Está especialmente centrada en el diseño sobre FPGA [37].

Synplify Premier soporta las siguientes características necesarias en la fase de síntesis de diseño hardware:

- **Diseño jerárquico**. Debido a la creciente complejidad de los diseños y a su tamaño, cada vez más es típica la división en subdiseños con el fin de trabajar en paralelo. Synplify Premier permite la realización de proyectos divididos en subproyectos independientes. Permite tanto diseño *top-down* como *bottom-up*.
- **Modo rápido**. Es un modo de síntesis que se realiza con una aceleración de hasta 3x, permitiendo realizar un análisis inicial del sistema.
- **Multiprocesado usando particiones**. Pueden definirse lo que Synplify llama puntos de compilación (*Compile Points*) que son en realidad particiones del diseño con el fin de que pueda lanzarse en paralelo la síntesis de cada una de las particiones, con el consecuente ahorro de tiempo de síntesis.
- **Diseño incremental**. Haciendo uso de los puntos de compilación, se pueden heredar soluciones de cada partición sintetizadas con anterioridad, para que únicamente sea necesario resintetizar la partición que haya sido modificada. Así, durante la fase de optimización, se puede ahorrar una gran cantidad de tiempo de procesamiento si toda la optimización se realiza en un número reducido de particiones. Además, estas divisiones pueden portarse a las siguientes herramientas de posicionado y ruteado.

5.4.1 Vistas

En Synplify puede representarse el diseño en dos vistas, una antes de realizar la síntesis lógica, que se conoce como vista RTL, en la que se representa a modo de bloques la descripción de los ficheros fuentes HDL (figura 40). La segunda vista es la tecnológica en la que cada bloque se representa como el conjunto de recursos sobre el que ha sido mapeado, ya sean estos registros, memorias de bloque RAM o LUTs (figura 41).

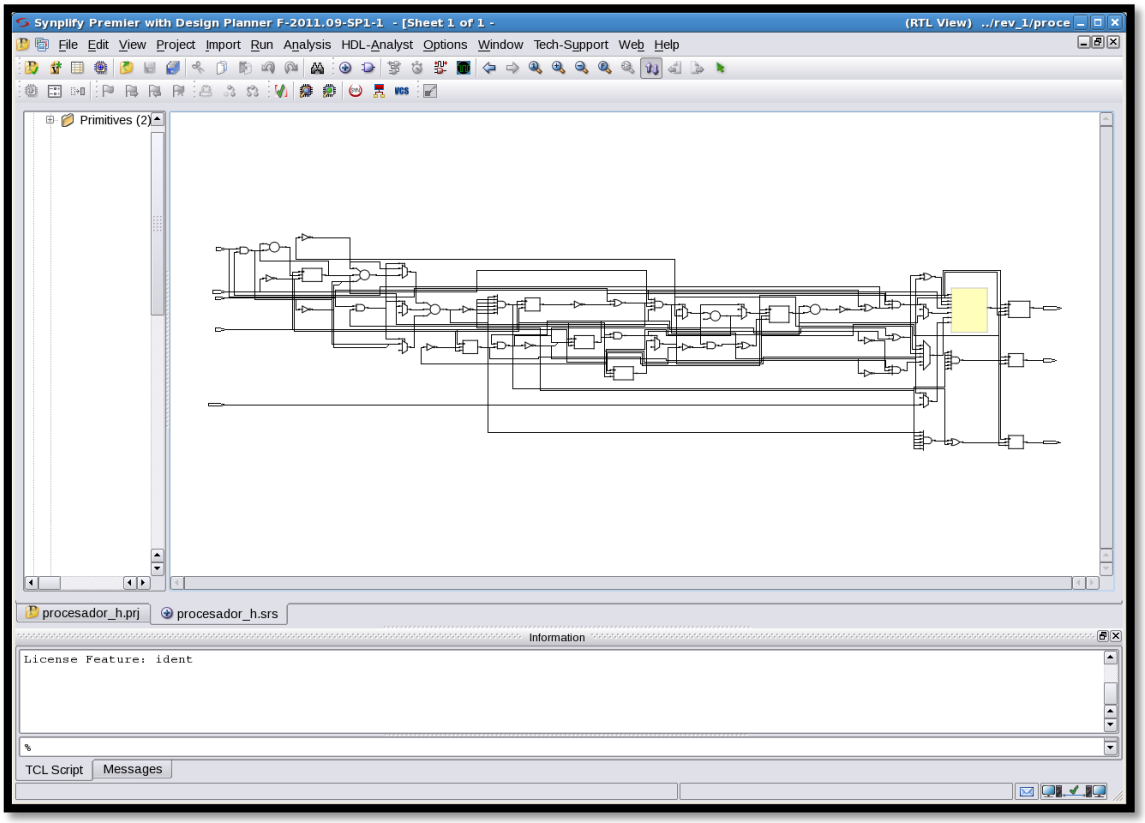


Figura 40. Vista RTL de Synplify Premier.

5.5 Incisive Enterprise Simulator

Incisive es el entorno de simulación que Cadence propone para la verificación a distintos niveles de abstracción de los diseños *hardware*.

Con las nuevas tendencias de diseño a cada vez más alto nivel, las herramientas de simulación deben adecuarse a nuevos requerimientos. *Incisive* ha progresado junto con los flujos de desarrollo para dar solución a todas las necesidades de verificación que puedan surgir en el diseño de sistemas electrónicos [38]. Sus principales características se resumen en los siguientes puntos:

- Permite la verificación a todos los niveles de abstracción usados, desde a nivel de transacciones (TLM), hasta código HDL.
- Soporta todos los lenguajes estandarizados por el IEEE para descripción Hardware: SystemC, SystemVerilog, Verilog, VHDL, etc.

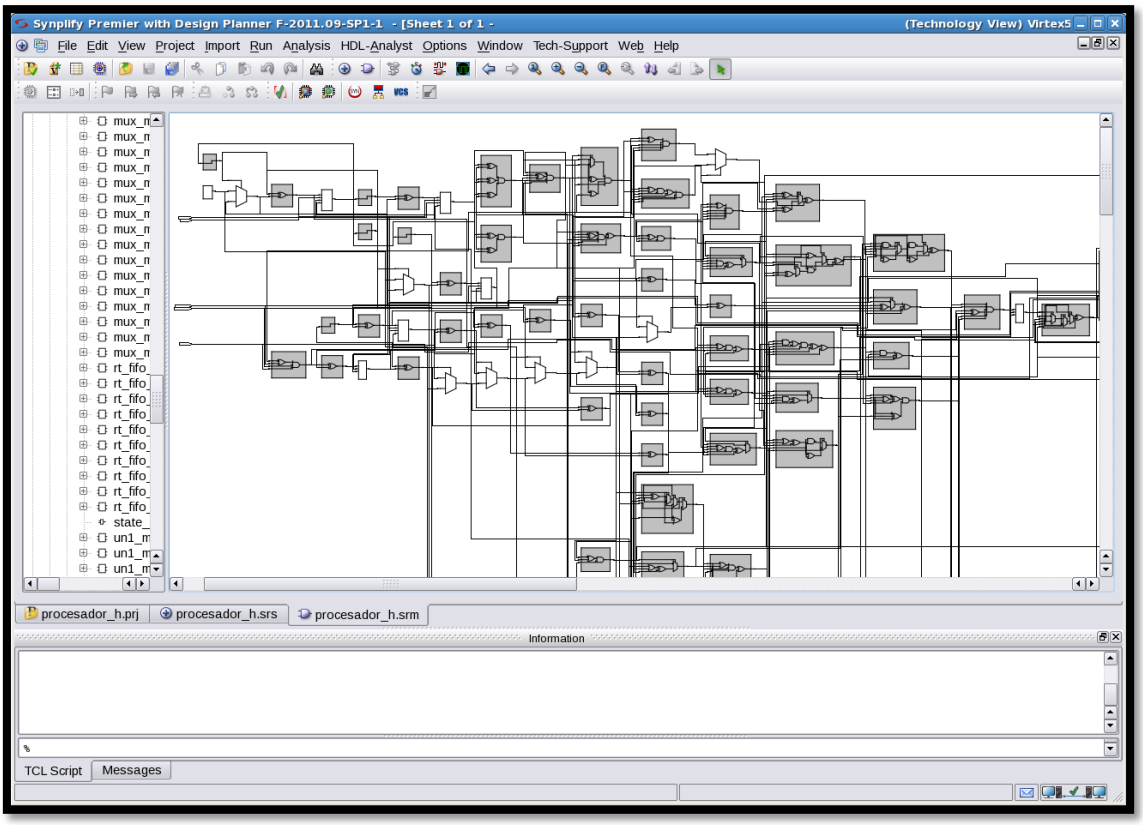


Figura 41. Vista tecnológica de Synplify Premier.

- Permite simulación cruzada con sistemas escritos por combinación de los lenguajes soportados.
- Soporta ejecución en línea de comandos o con forma de onda.

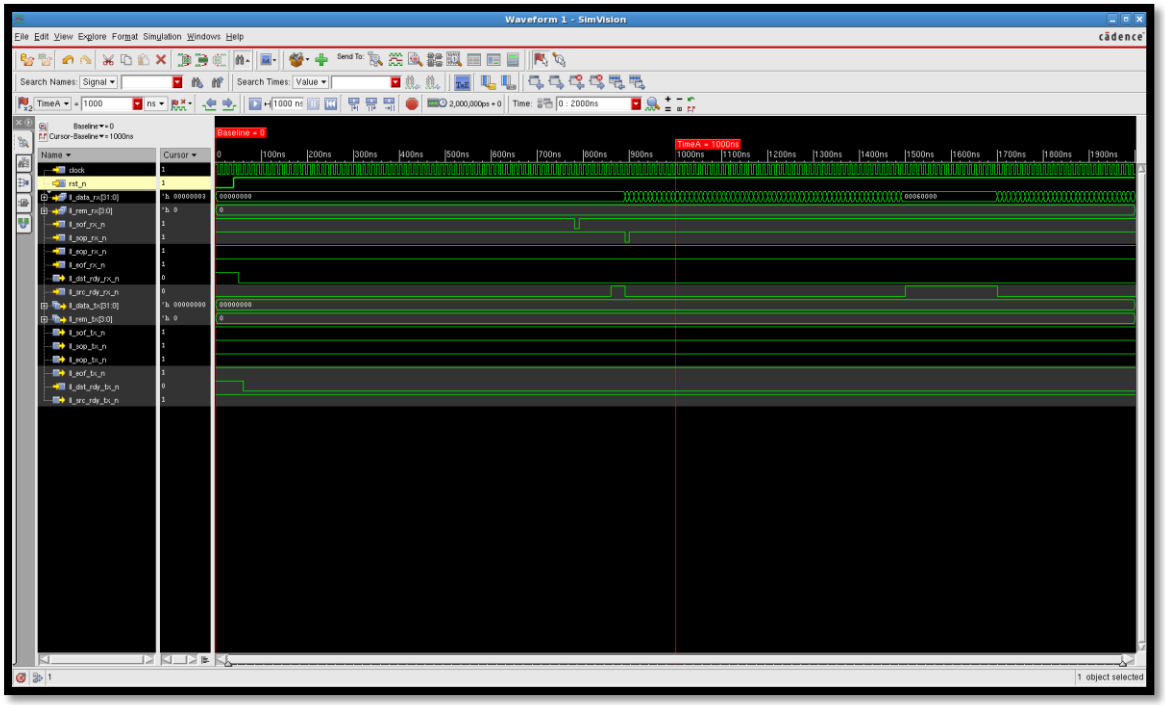


Figura 42. Incisive Enterprise Simulator en modo forma de onda.

5.6 Xilinx Platform Studio (XPS)

Xilinx Platform Studio es la herramienta que Xilinx proporciona para la realización de plataformas *hardware* en las que se dispone de uno o de varios microprocesadores tanto Microblaze como PowerPC.

En conjunto con otras herramientas de Xilinx, XPS permite crear a nivel de plataforma un diseño formado por el microprocesador y un conjunto de periféricos, ya sean importados de la propia librería de IPs que proporciona Xilinx como diseñados por el usuario, que podrán ser importados en formatos HDL o netlist..

El objetivo de XPS es dar facilidad al diseñador a realizar instancias e interconectar distintos bloques de gran complejidad de forma sencilla y gráfica, pudiendo tener una visión global del sistema completo sin perderse en tener que entender a nivel de bit cada una de las interfaces de los distintos IPs.

La aplicación integra otras herramientas de Xilinx para poder implementar el diseño. De esta forma, una vez realizado el diseño, puede realizarse todos los pasos hasta la programación de la FPGA sin tener que exportar el diseño a otras herramientas. Para ello, hace uso de otras herramientas[39], tales como:

Tabla 4. Herramientas integradas en XPS.

Herramienta	Descripción
Xilinx CORE Generator System	Genera la descripción HDL de bloques IP parametrizables por el diseñador.
Platform Generator	Genera la descripción HDL de la plataforma completa a partir de los distintos ficheros sinstetizados o no de cada bloque IP y del fichero de especificaciones hardware (*MHS file*).
PlanAhead	El PlanAhead, como se comentará más adelante, es una herramienta que se divide en tres etapas (Map, Place&Route y BitGen). Aunque XPS no realiza llamadas al PlanAhead, sí que lo hace a las tres etapas independientemente.
Xilinx Synthesis Technology (XST)	Es la herramienta de síntesis de Xilinx. Realiza la síntesis lógica que traduce la descripción HDL a una descripción estructural.
Software Development Kit (SDK)	Es un entorno de desarrollo software basado en Eclipse que Xilinx ha desarrollado para la realización y depurado de la aplicación software que se ejecutará en los microprocesadores de la plataforma.
Library Generator	Crea las librerías necesarias en función de las especificaciones software del proyecto (*MSS file*). Estas librerías son las que contendrán aquellas funciones que se hayan requerido en la elaboración de la aplicación del microprocesador.
GNU Compiler	Es el compilador que traduce la aplicación a lenguaje máquina del microprocesador usado en la plataforma. Se incluyen las versiones para Microblaze y PowerPC.

Herramienta	Descripción
Xilinx Microprocessor Debugger	Se trata de un *Shell* con el que configurar el microprocesador de la placa. Con él se descarga el fichero ejecutable así como se dan las instrucciones para el arranque de la aplicación y la interrupción de detención.
System ACE File Generator	Genera el fichero de configuración Xilinx System ACE, que congrega el fichero *bitstream* de configuración *hardware* de la FPGA y el fichero *software* ejecutable del microprocesador.

En la figura 44 se muestra la interfaz de usuario de XPS, en la que se identifica el microprocesador conectado a distintos bloques IP a través de las interfaces de conexión disponibles, bus PLB o DMA.

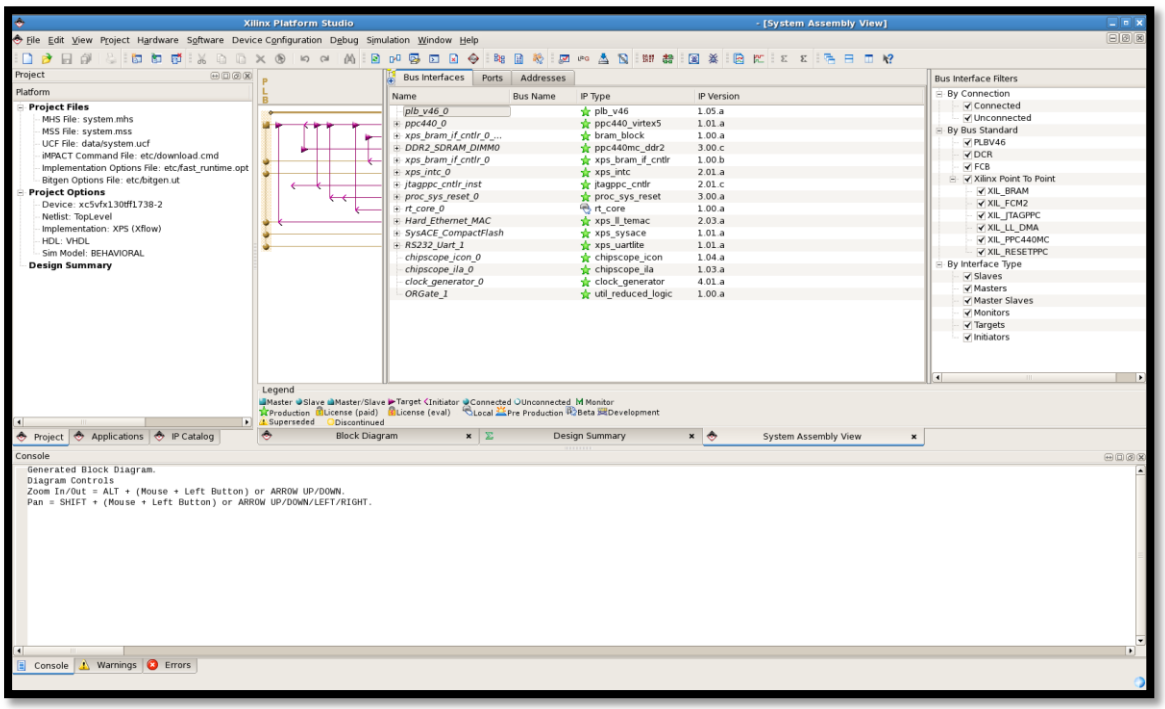


Figura 43. Interfaces de bus en XPS.

5.7 PlanAhead

Xilinx PlanAhead es un entorno avanzado que facilita la realización de las tareas de colocación y asignación de interconexiones de los recursos de la FPGA. Dispone de diferentes tipos de estrategias predefinidas (optimización global, temporal, optimización por bloques, etc), permite la creación de estrategias definidas por el usuario a partir de la asignación de diferentes opciones des los programas de *back-end* integrados.

PlanAhead permite la importación de diseños en formato HDL, en cuyo caso requiere una fase previa de síntesis que realiza mediante la invocación de la herramienta XST indicada anteriormente, o directamente en formato *netlist* EDIF o NGC [40].

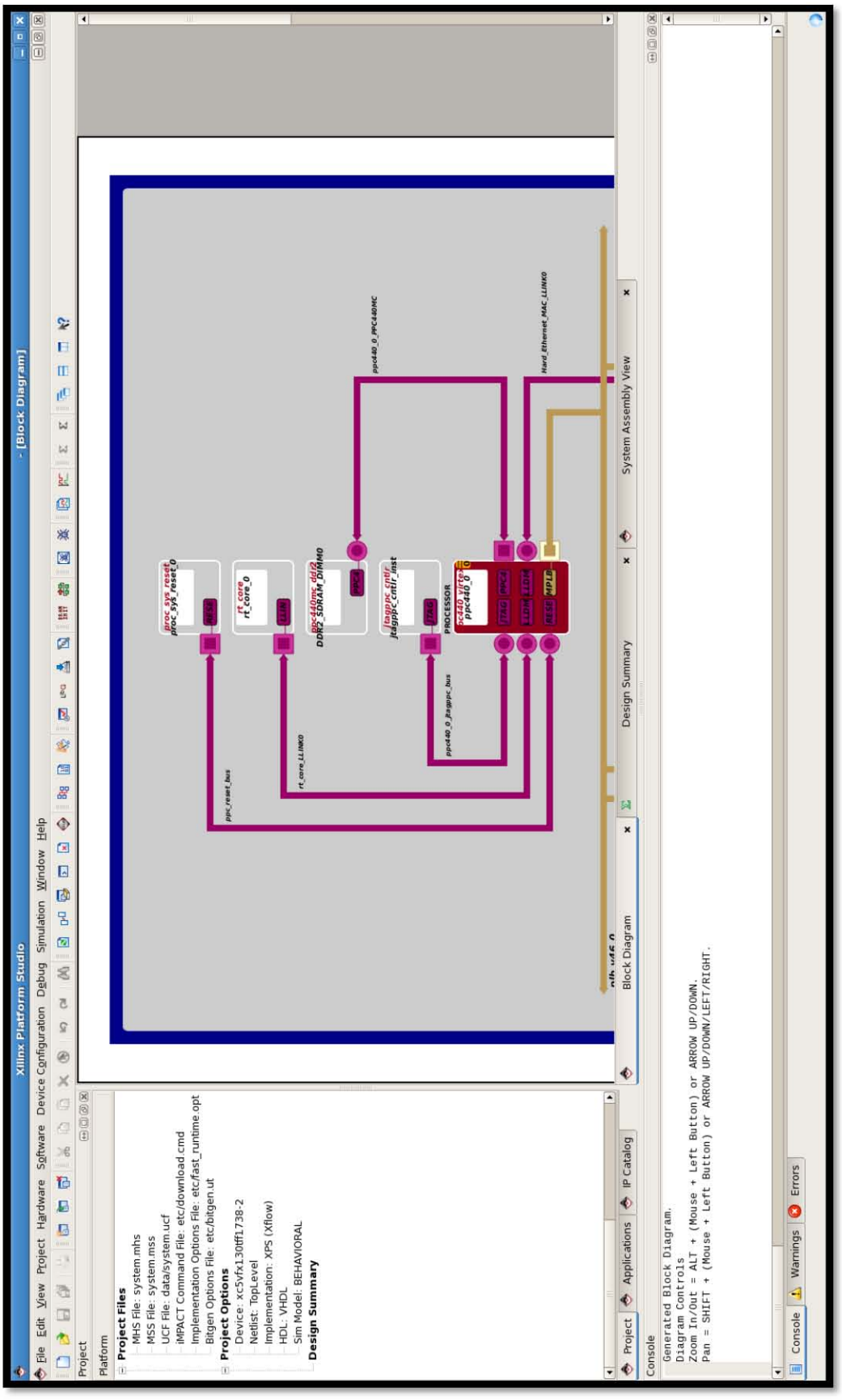


Figura 44. Diagrama de bloques en XPS.

Además de la posibilidad de definir estrategias para los algoritmos de colocación o de ruteado, también pueden incluirse restricciones de área en la colocación. Se permite al diseñador definir áreas de la FPGA donde ubicar cada uno de los módulos del diseño, lo cual puede llevar a soluciones más eficientes.

Una vez concluida la fase de implementación, realiza un análisis temporal que comprobará si se cumplen todas las restricciones temporales, tanto impuestas por el usuario como las impuestas por la herramienta a través de la definición de reloj.

Permite además lanzar los algoritmos en paralelo en diferentes máquinas con el fin de paralelizar el proceso.

Una vez se obtiene un diseño completamente ruteado, permite llamar al generador del fichero de configuración de la FPGA (*bitstream*).

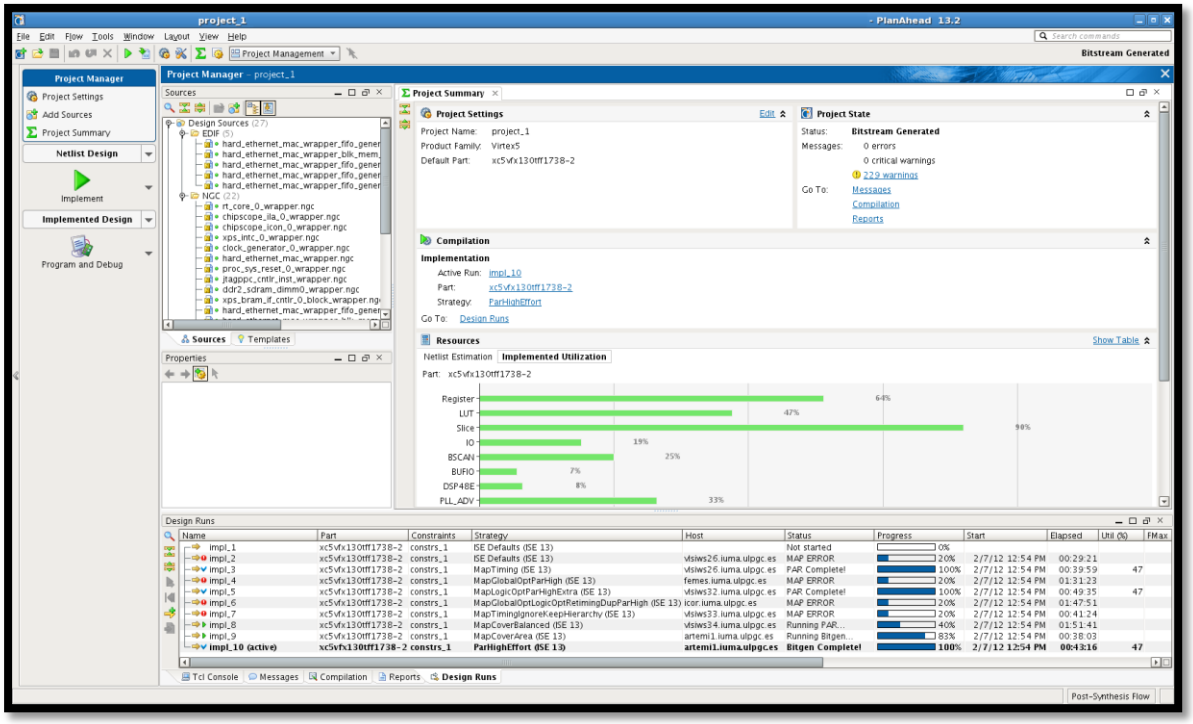


Figura 45. Estrategias en paralelo en PlanAhead.

5.8 iMPACT

Es la aplicación usada para configurar la FPGA. Haciendo uso del cable de configuración USB y del fichero bitstream, facilitando la realización de su programación.

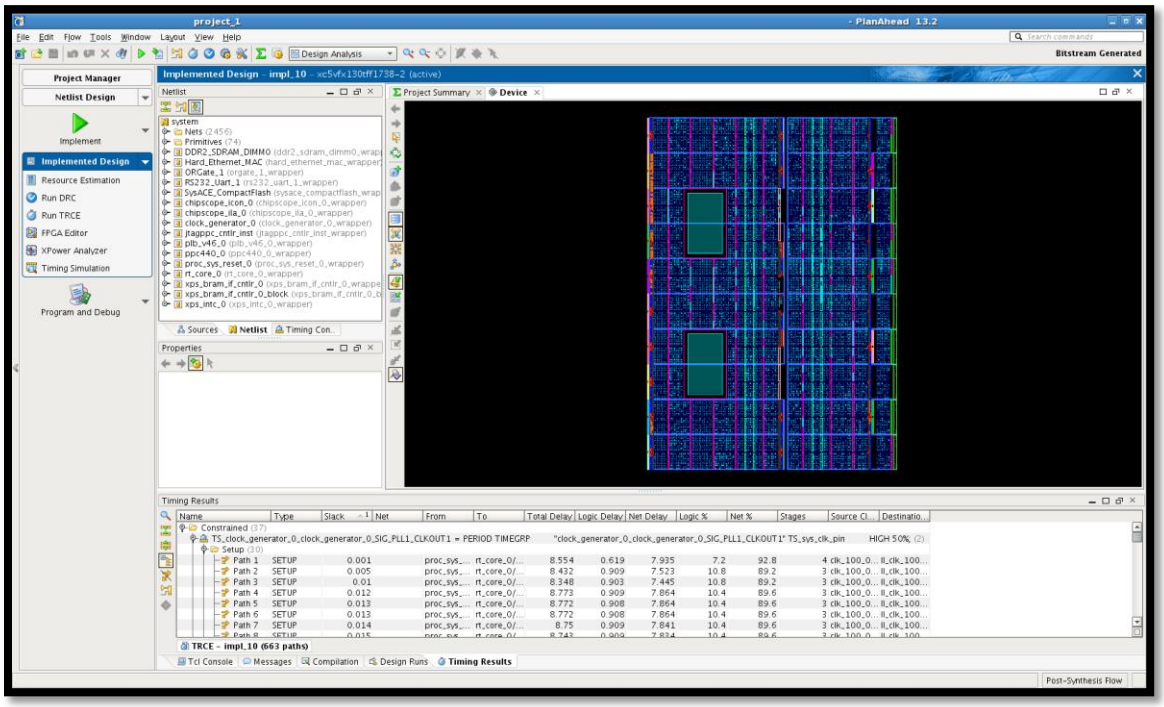


Figura 46. Diseño implementado en PlanAhead.

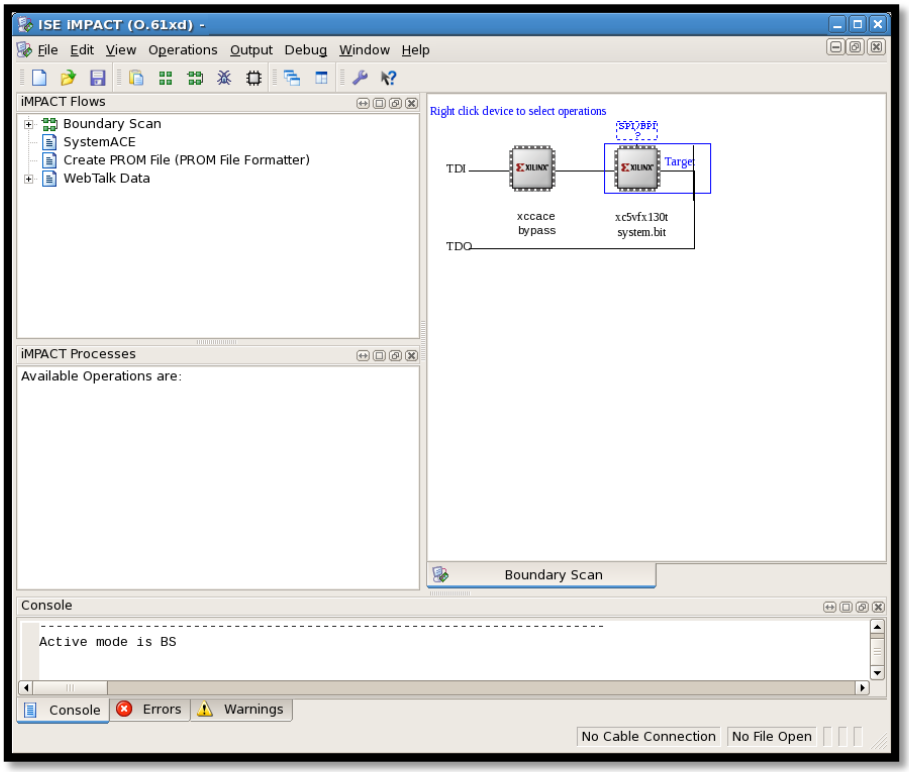


Figura 47. Programación de la FPGA con iMPACT.

5.9 ChipScope Analyzer

ChipScope Analyzer es una herramienta que facilita el depurado del diseño *hardware* mediante la utilización de mecanismos de tipo ILA (In-Circuit Logic Analyzer). Permite controlar los bloques de analizador lógicos internos en la FPGA, obteniendo la forma de onda de los mismos. Durante la fase de diseño de la plataforma, existen bloques IP contenidos en la librería de depurado de Xilinx, que tienen por objetivo almacenar muestras de los puntos de prueba a los que se conecten. Durante su instanciación se deberá configurar el ancho de cada punto de prueba, así como la longitud en número de muestras que podrá almacenar cuando comience la captura (las muestras se almacenarán en memorias de bloque interna, por lo que deberá prestarse interés a dicha cantidad) [41].

Una vez la FPGA ha sido configurada, con bloques de análisis lógico internos conectados a aquellas señales que se quieran capturar, es cuando entra en juego el *software* ChipScope, el cual permitirá configurar las condiciones de disparo, es decir, las condiciones bajo las cuales comenzarán a capturarse datos.

La conexión entre el software de análisis lógico y los bloques *hardware* de captura de datos se realiza a través del puerto de configuración JTAG de la placa de prototipado, que tiene conexión directa con los puertos de configuración de la FPGA.

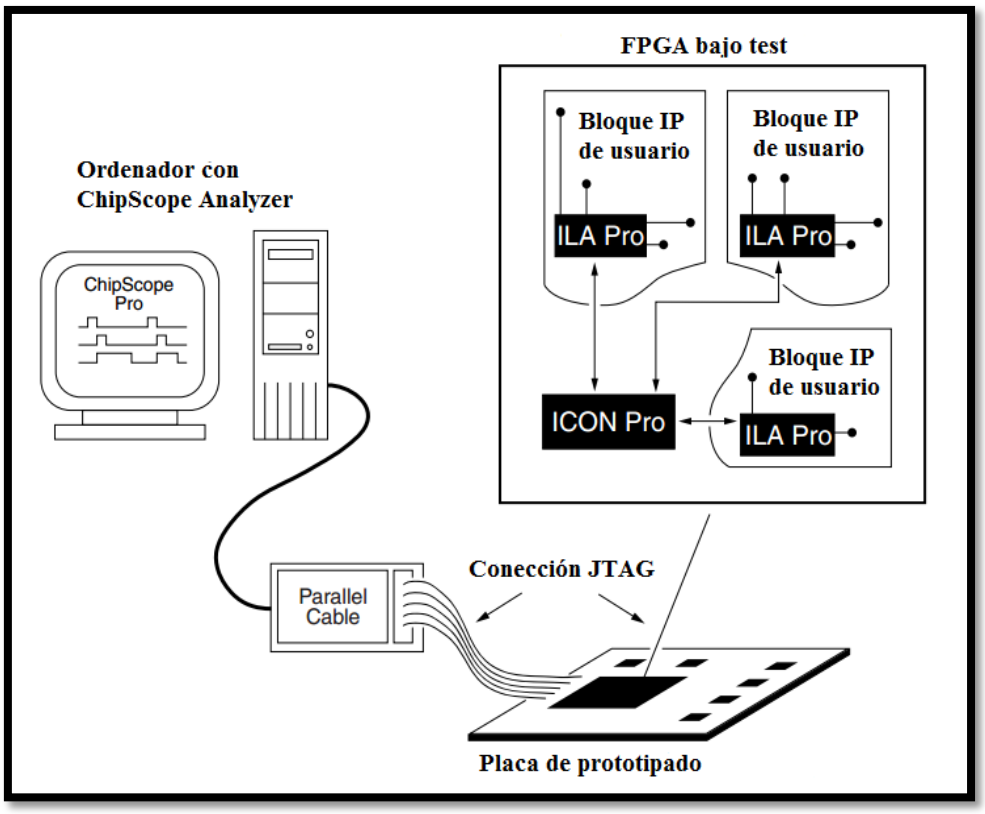


Figura 48. Conexión para analizador lógico ChipScope.

Se pueden configurar varias ventanas de captura, con la consecuente disminución de número de muestras que se puedan capturar por ventana. Cada señal conectada a un puerto de disparo, puede usarse tanto para definir condiciones como datos de captura.

Se pueden formar condiciones complejas en forma de composiciones AND y/o OR.

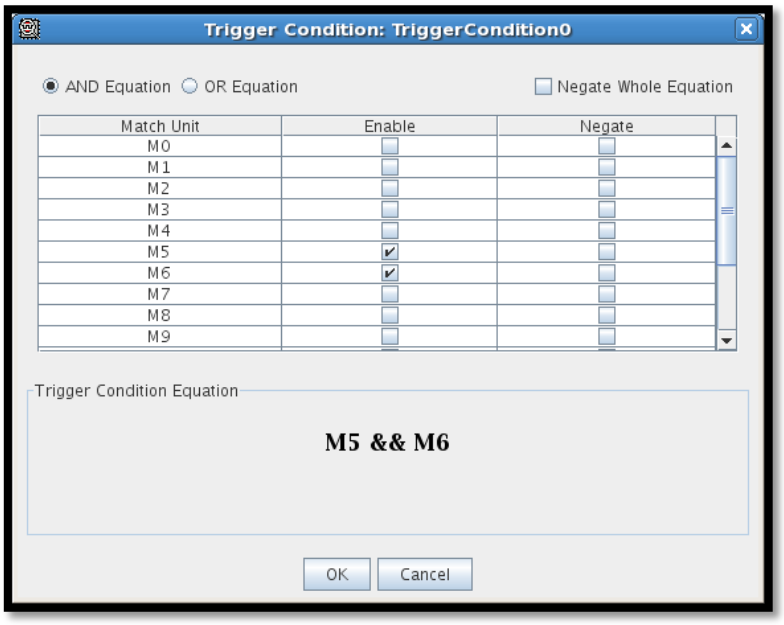


Figura 49. Condiciones de disparo en ChipScope.

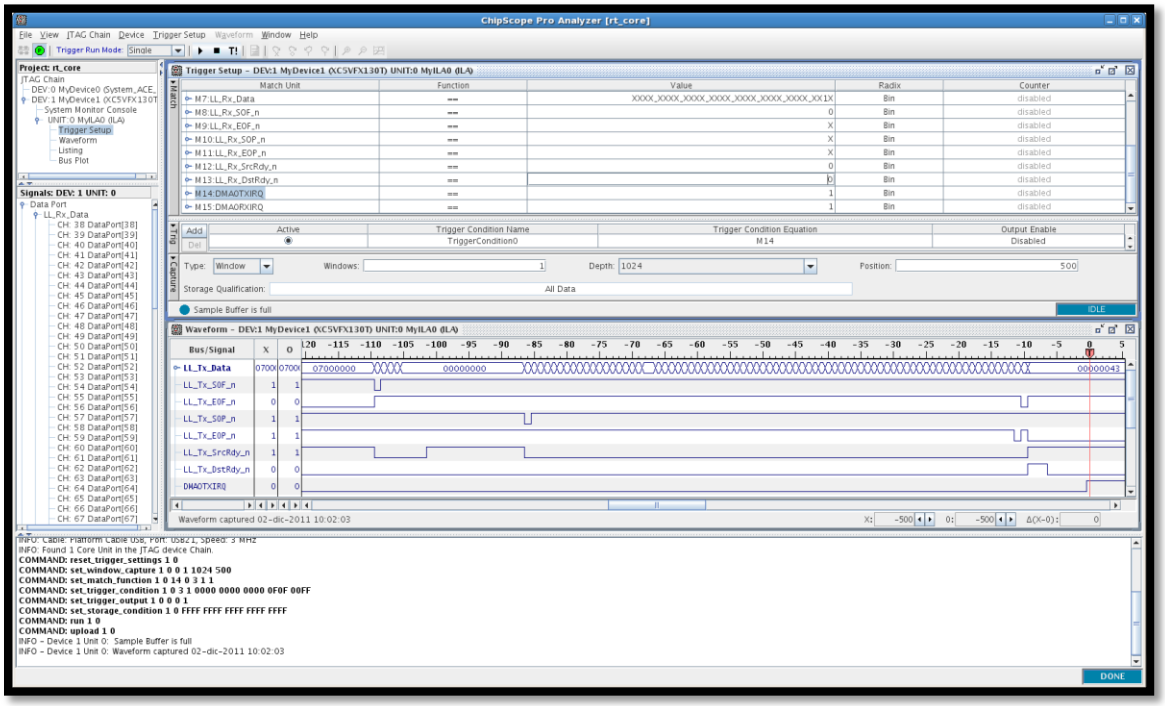


Figura 50. Captura de datos con ChipScope.

5.10 FPGA Editor

FPGA Editor es una herramienta de edición de diseños físicos ya implementados para las FPGAs de Xilinx. Esta aplicación toma como entrada un diseño completamente implementado, es decir, tras haber realizado la colocación y ruteado de los recursos en uso. Con ella pueden modificarse conexiones, así como mover células ya mapeadas hasta otros recursos que se encuentren libres, con el fin de reducir ciertas rutas o simplemente modificar parte de la funcionalidad del diseño, sin tener que realizar todo el flujo desde HDL o incluso desde SystemC si fuese el caso.

Una utilidad muy importante que presenta FPGA Editor, es la posibilidad de modificar las conexiones entre las señales de disparo de los bloque de analizador lógico, para así cambiar los puntos de prueba a diferentes señales del diseño sin volver a realizar el flujo de síntesis desde el comienzo. Además, puesto que la conexión de los analizadores lógico se hace a nivel de la plataforma, solo es posible su conexión en Xilinx Platform Studio a puertos de ese nivel de jerarquía, es decir, a los puertos de entrada y de salida de cada bloque IP, pero no a señales internas a cada bloque. Para conectarlos a señales internas debe hacerse uso del FPGA Editor una vez el diseño haya sido completamente implementado [42].

Cada señal debe cambiarse bit a bit, por lo que se presenta la posibilidad de hacer uso de *scripts* para realizar buses enteros de un número indefinido de bits indefinidamente sin tener que hacer a mano cada conexión.

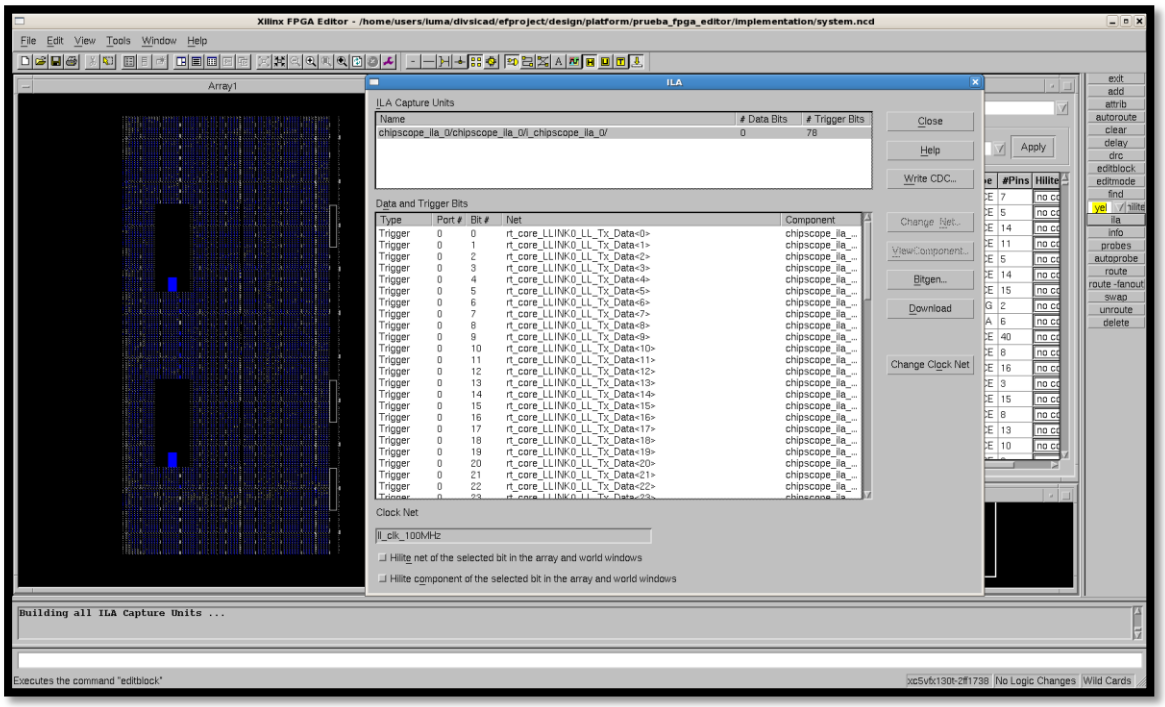


Figura 51. Modificación de conexión del analizador lógico con FPGA Editor.

6 Flujo de diseño propuesto

A continuación, y a partir de las herramientas, lenguajes y tecnologías presentadas en los apartados anteriores, se concreta un flujo de diseño especificando la herramienta a usar en cada etapa del mismo. Este flujo se presenta en la figura 52.

El primer paso consiste en realizar el perfilado del código de referencia con el que se trabajará en el PFC. Para realizar esta tarea se hace uso conjuntamente de las herramientas Callgrind para realizar el perfil, y de KCachegrind para representarlo de forma que sea más fácil de estudiar.

Una vez realizado el perfil, en función de este se realizará la partición de la aplicación, analizando los kernels de procesamiento intensivo que es necesario implementar como un bloque *hardware* IP, y que otra parte será ejecutada de forma *software* sobre el microprocesador empotrado de la FPGA, e incluso que otra parte se mantiene en ejecución en el procesador de propósito general.

En este punto se divide el flujo en dos líneas de actuación tratando de reducir la dependencia directa entre ellas, que son el diseño de la plataforma y el del bloque IP.

El primer paso del diseño de la plataforma es añadir los bloques *hardware* necesarios usando para ello la herramienta de Xilinx XPS. Con ella se instanciarán aquellos recursos necesarios y se configurarán los parámetros que corresponda. Puesto que en este punto no se dispone aún de una versión del procesador de eventos *hardware*, se describirá uno que simplemente reciba y envíe tramas a través de la interfaz *hardware*/*software* que se haya decidido usar. Una vez diseñada la plataforma *hardware*, el siguiente paso es definir la plataforma *software*, compuesta por el sistema operativo, así como librerías necesarias. Con toda la plataforma definida, se procederá a diseñar la aplicación *software* que se ejecutará en el procesador empotrado. Todas estas etapas se realizan con la herramienta Xilinx Platform Studio.

A partir de aquí habrá que seguir el flujo de implementación de FPGA, que puede realizarse enteramente a través de las herramientas integradas en XPS, o exportar el diseño a PlanAhead en función de las necesidades del diseño, y programar la FPGA con iMPACT.

Ya programada la FPGA, se hará uso de ChipScope para verificar el correcto funcionamiento sobre el sistema físico.

Para el flujo de diseño del bloque IP se comienza con la captura del diseño en SystemC, verificando su correcto funcionamiento comparando siempre los resultados con la aplicación original. Una vez la descripción ESL sea funcionalmente correcta, se realizará la síntesis de alto nivel mediante la herramienta de Cadence CtoS, para luego verificar que el diseño obtenido a nivel RTL es funcionalmente idéntico a su predecesor. Para esta comparación se hará uso del Incisive Simulator de Cadence. Tras esta comprobación se realizará la síntesis lógica para la obtener del netlist del bloque IP diseñado.

Llegados a este punto solo queda integrar el bloque IP a la plataforma diseñada y verificada, para volver a realizar la implementación, programación y verificación. Se hará uso además de la

herramienta FPGA Editor, con el fin de poder configurar los puntos de prueba del analizador lógico y acceder a señales internas del CE.

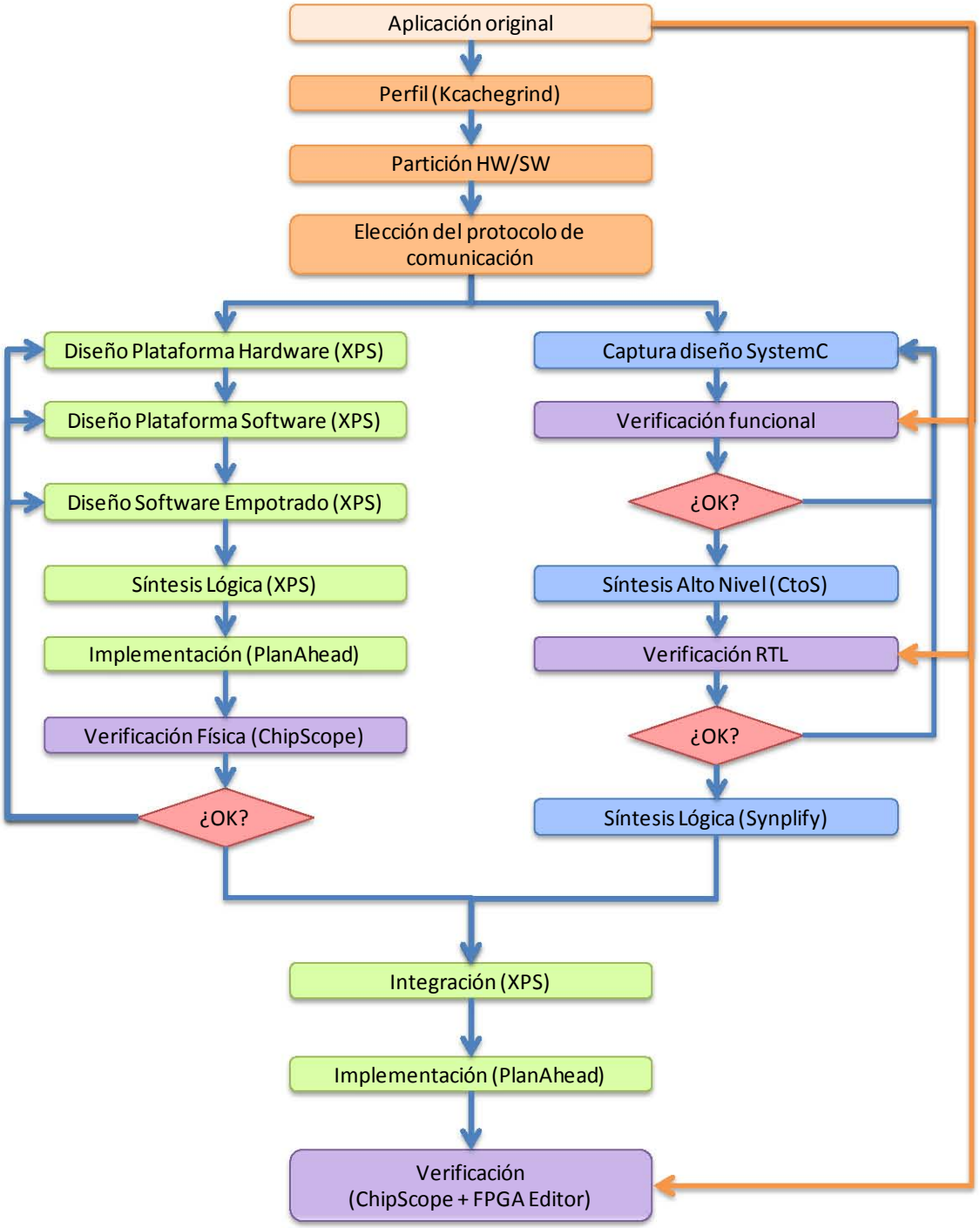


Figura 52. Flujo de diseño propuesto.

7 Conclusiones

En el presente capítulo se han presentado la tecnología, metodología y técnicas fundamentales para la correcta realización de este proyecto.

Por una parte se ha presentado la tecnología de implementación, que para el presente Proyecto Fin de Carrera corresponde a la familia de FPGAs Virtex-5 de Xilinx. Se han destacado de estas, aquellas propiedades que las hacen de interés para el desarrollo de este proyecto así como una pequeña comparativa entre la primera opción de placa de prototipado, y la que se terminó por usar para la implementación.

Además se han explicado los dos lenguajes de descripción hardware usados en su desarrollo, uno de nivel de sistemas como es SystemC, y otro de nivel RTL como es Verilog.

Por último se han descrito las herramientas a las que se hará alusión durante la descripción del desarrollo del proyecto, explicando sus objetivos y a qué etapa del flujo pertenecen, para terminar presentando el flujo de diseño propuesto.

Capítulo 4:
Diseño de la plataforma

1 Introducción

En este capítulo se describen los pasos seguidos en el diseño de la plataforma, tanto en lo que a bloques *hardware* se refiere, como a la plataforma *software* y la aplicación a ejecutar en el procesador empotrado.

Se pretende conseguir un sistema empotrado, capaz de recibir las tramas de datos de entrada al CE, empaquetarlas, y hacerlas disponibles al bloque IP. Para poder recibir estos eventos de entrada, el sistema debe ser capaz de comunicarse mediante protocolos TCP/IP.

Otra necesidad del sistema es la capacidad de ser autoprogramable, esto es, que una vez se apague y se encienda el sistema, la FPGA vuelva a configurarse, y el procesador empotrado comience a ejecutar una vez más la aplicación diseñada.

Estos y otros requisitos son los que imponen los bloques IP que deben añadirse a la plataforma, así como los valores de sus atributos.

2 Requisitos

A continuación se recogen los requisitos que debe cumplir la plataforma, y que darán lugar luego a la inclusión de ciertos bloques en la misma.

- Comunicación TCP/IP a través del puerto Ethernet de la placa de desarrollo.
- Capacidad de monitorización del sistema.
- Capacidad de autoreconfiguración tras un *reset*.
- Posibilidad de depuración mediante analizador lógico interno (In-Circuit Logic Analyzer – ILA).

La solución propuesta está basada en un sistema en chip, para poder así gestionar el stack TCP/IP a través de un microprocesador empotrado. De esta forma, el diseño se basa en un SoC programable tanto en hardware (tecnología FPGA) como en software (uso de un microprocesador empotrado). La familia de dispositivos Virtex 5 FXT de Xilinx dispone de uno o varios núcleos microprocesadores empotrados de tipo PowerPC 440 del fabricante IBM. La función de este microprocesador en la plataforma será la de recibir los datos desde la interfaz de red Gigabit Ethernet, realizar el preprocesado de los eventos de entrada, enviarlos al Coprocesador de Eventos (CE), recibir las notificaciones desde el CE y enviarlas hacia la interfaz de red. Además, dará soporte a otras funciones auxiliares que se ejecutan en el procesador para informar el estado del mismo.

En los apartados siguientes se explican los bloques que componen la plataforma, así como su función y configuración.

3 Bloques IP

Xilinx incluye en su entorno de diseño un conjunto de bloques IP que el diseñador puede usar para crear su plataforma. De estos bloques IP puede estar disponible tanto una descripción RTL, como una *blackbox* instanciada en un *wrapper* en HDL, o simplemente pueden ser recursos que se encuentran físicamente en la FPGA.

3.1 Procesador empotrado

Se trata de un recurso que se encuentra empotrado en la FPGA. En particular, en el modelo V5FX130T existen dos instancias de este bloque. El bloque contiene, además del PowerPC 440, otros módulos de acceso a los buses de datos y de control, que se han incluido con el fin de no consumir lógica adicional programable del resto del dispositivo como ocurría en los dispositivos Virtex 4 [43].

Para mejorar los accesos a memoria por parte del procesador y de otros IPs, la conexión con memoria se realiza a través de una matriz de interconexión (*crossbar switch*), que acepta peticiones de transferencia de las cachés de datos e instrucciones del procesador, así como de

cuatro bloques DMA y dos interfaces esclavas PLB, que se encuentran dentro del bloque de procesador. Estas transferencias pueden tener como destino la interfaz del controlador de memoria o una interfaz maestra PLB.

Además de las características anteriormente citadas, existen otras interfaces en el bloque de procesador. La primera es una interfaz para una unidad de procesamiento auxiliar (APU), como puede ser por ejemplo una unidad de coma flotante (FPU), ya que los procesadores PowerPC 440 carecen de ella en su ruta de datos. Otra unidad presente es la interfaz de registro de control del dispositivo (DCR), un mecanismo usado por el PowerPC para establecer y controlar el estado de otros periféricos del sistema. De esta forma, el microprocesador puede comunicarse con los periféricos para tareas de control sin saturar el bus principal del sistema (PLB). Por último, existen además otras interfaces de control, como las señales de reloj, de reset, así como de gestión de potencia.

La instancia de un bloque de procesador se realiza mediante un *wrapper* que no consume lógica, y que tiene como única función recoger los valores de parámetros del bloque en cuestión.

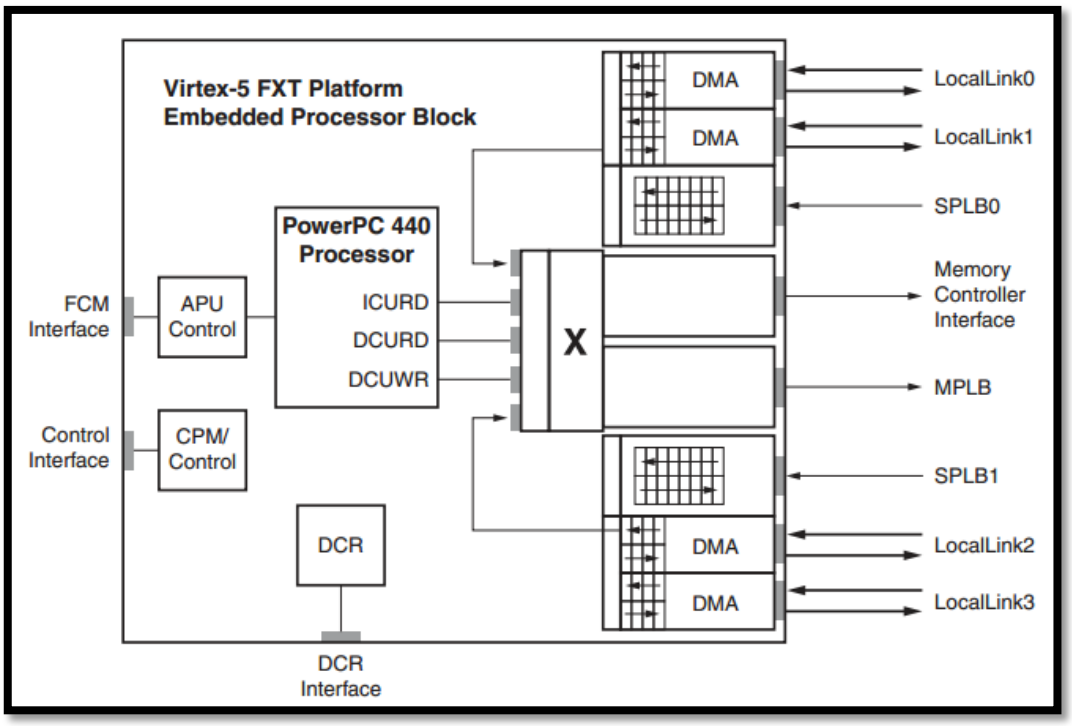


Figura 53. Diagrama del bloque de procesador.

Como puede observarse en la figura 53, el *crossbar* posee cinco interfaces esclavas PLB:

- tres entradas que provienen del PowerPC 440
  - acceso a instrucciones
  - lectura de datos
  - escritura de datos)
- dos entradas provenientes desde los bloques DMA e interfaces PLB esclavas (SPLB) superior e inferior.

Como puede observarse, cada dos bloques DMA y una interfaz esclava PLB externa comparten un único recurso de acceso a la memoria. Los bloques DMA tienen interfaces LocalLink (LocalLink0-3) para la conexión con los periféricos externos.

Para la interconexión entre el procesador y cualquier periférico existirán dos alternativas:

- conectar el periférico al bus PLB a través de una interfaz esclava (se proporciona por parte de la herramienta Xilinx Platform Studio un asistente para la creación de un *wrapper* con la conectividad necesaria)
- conectar el periférico directamente a memoria a través de un bloque DMA y que éste genere una interrupción cuando envíe o transmita una trama completa. En esta segunda opción el periférico deberá tener una interfaz de tipo LocalLink con la que comunicarse con el bloque DMA (para este caso no existe ningún asistente).

En el otro extremo del *crossbar*, es decir, los elementos a los que acceden las interfaces esclavas PLB nombradas anteriormente, se encuentran dos interfaces maestras. La primera es la interfaz con el controlador de memoria principal (Memory Controller Interface – MCI). La segunda es una interfaz maestra PLB que permite al PowerPC acceder al bus para comunicarse con los periféricos.

Uno de los puntos clave del *crossbar* es que los arbitrajes para el acceso a la memoria principal y a la interfaz maestra PLB son independientes, pudiéndose realizar transacciones paralelas en ambos puertos.

3.2 Controlador de memoria DDR2

Este bloque controlador de memoria permite la utilización directa desde la FPGA de la memoria externa disponible en la plataforma y que servirá de memoria principal del PowerPC 440. El bloque procesador se conecta al controlador a través de la interfaz correspondiente del controlador de memoria (MCI) [44].

Las características más importantes de este bloque son las siguientes:

- Soporta una frecuencia de trabajo de hasta 333 MHz.
- Soporta interfaces de datos de 16, 32 y 64 bits de ancho de palabra.
- Soporta memorias de tipo DIMM de *single-rank*.
- Soporta las siguientes características para memorias DDR2 SDRAM:
    - Latencias CAS (3, 4, 5)
    - Latencias AL (0, 1, 2, 3, 4)
    - Tecnología ODT
    - Longitud de ráfagas (4, 8)
- Soporta gestión de bancos (hasta cuatro).
- Realiza la inicialización de la memoria del dispositivo al arranque.
- Realiza ciclos de *auto-refresh*.

Con el fin de optimizar el rendimiento del controlador, el EDK de Xilinx generará unas restricciones de usuario para cada tipo de encapsulado y dispositivo. En caso de hacer uso de

placas de prototipado de Xilinx, la herramienta automáticamente añadirá estas restricciones al fichero de restricciones de usuario (UCF).

A continuación se muestra el diagrama del bloque de procesador, con el controlador de memoria conectado a la interfaz MCI.

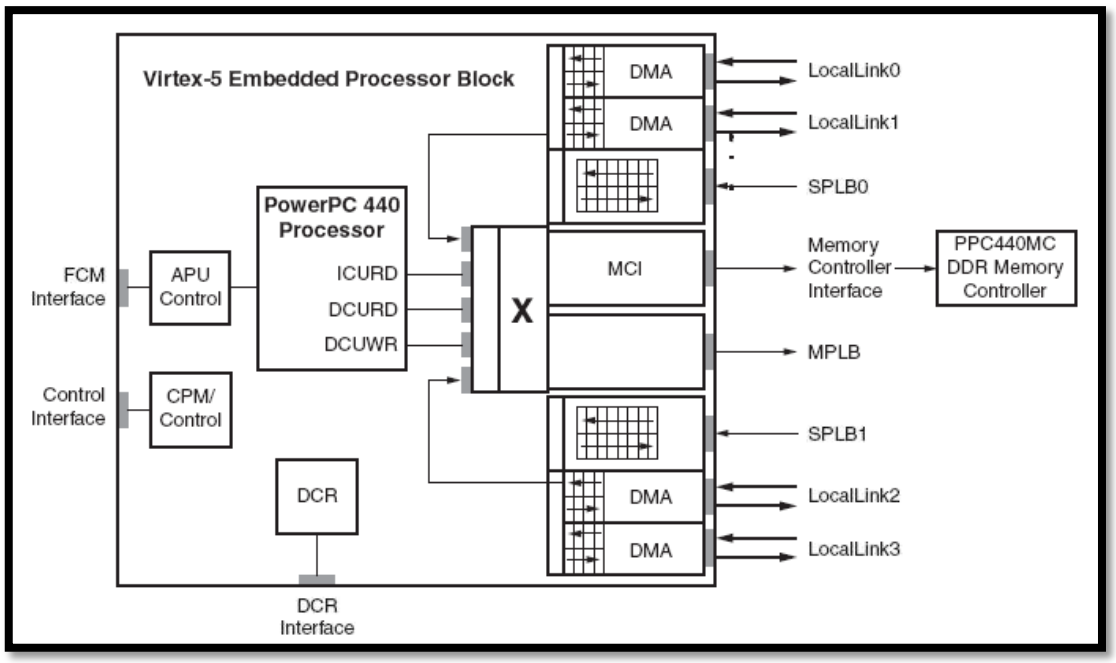


Figura 54. Conexión MCI entre el bloque procesador y el controlador DDR2.

Con el fin de maximizar el rendimiento y minimizar las latencias de acceso a memoria, se ha optado por usar el ancho máximo de bus de datos posible en la conexión con la memoria, es decir de 64 bits. Este controlador no está empotrado de forma física en la FPGA, sino que es proporcionado por Xilinx como una descripción en un lenguaje HDL (específicamente en Verilog), por lo que consumirá lógica programable de la FPGA. A continuación se exponen los costes de recursos para la configuración especificada, así como los valores de los atributos para dicha configuración.

Tabla 5. Coste de recursos del controlador DDR2.

Speed Grade	Frecuencia Máxima (MHz)	C_DDR_DWIDTH	Slices	Block RAM	Slice Flip-Flops	LUTs de 6 entradas
-2	300	64-bit	900	4	2025	1985

3.3 Controlador de interrupciones

El PowerPC 440 tiene un único puerto de interrupción. Por lo tanto, se hace necesaria la inclusión de un controlador que tenga como entrada varias interrupciones de dispositivos periféricos, y genere una salida que sirva de señal de interrupción para el bloque de procesador. A los registros para el control, la activación y la admisión o no de interrupciones se acceden a través de una interfaz esclava PLB [45].

Las características claves del controlador de interrupciones se pueden resumir en las siguientes:

- Se conecta como un esclavo de 32 bits a buses PLB de 32, 64 o 128 bits de ancho.
- Posee hasta 32 entradas de interrupciones configurables.
- Salida simple de un bit de interrupción.
- Posible conexión en cascada para permitir mayor número de entradas de interrupción.
- La prioridad entre interrupciones está determinada por su posición en el vector. EL bit menos significativo de la entrada de interrupción es el de mayor prioridad.
- Registro de activación de interrupción para permitir desactivar interrupciones de forma individual.
- Registro de activación maestro para desactivar la salida de interrupción.
- Cada entrada de interrupción es configurable por flanco o por nivel. Para las configuradas por flanco puede indicarse si es en flanco de subida o de bajada, y para las de nivel, si este es alto o bajo.
- Sincronización automática de flanco cuando una entrada se configura de esta manera.
- La salida de interrupción es configurable por flanco (de subida o de bajada) y por nivel (alto o bajo).

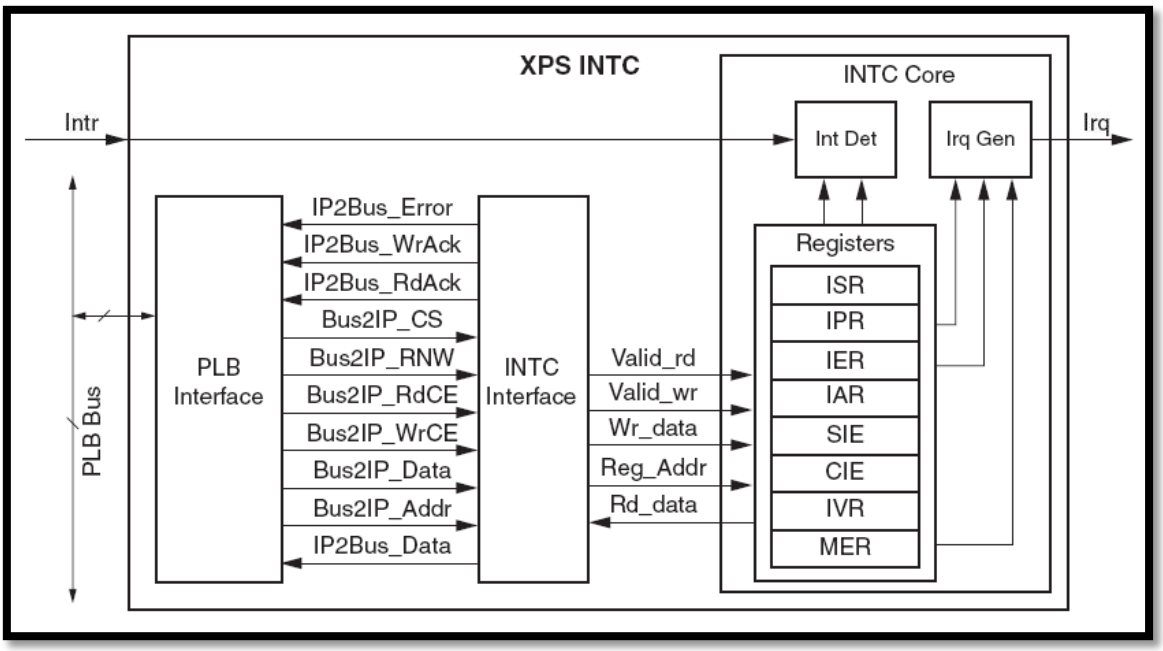


Figura 55. Diagrama del controlador de interrupciones.

Como puede observarse en la figura 55, el controlador de interrupciones se puede dividir en tres módulos:

- El módulo de interfaz es el encargado de comunicarse con el bus PLB ***(PLB Interface)***.
- Módulo de traducción de señales PLB/Controlador de interrupciones (***INTC Interface)***.

- Núcleo del controlador ***(INTC Core)***. Está formado por:
  - Detector de interrupciones **(*Int Det)***
  - Generador de la interrupción para el PowerPC 440 ***(Irq Gen)***
  - Banco de registros interno ***(Registers)***.

Al igual que ocurría con el controlador de memoria DDR2, Xilinx proporciona este bloque IP en forma de descripción HDL (en este caso, en VHDL). Es por ello que, en función del dispositivo donde se implemente y los parámetros de la instancia, el consumo de recursos varía. A continuación se muestra en una tabla el coste en recursos programables para el caso que nos ocupa.

Tabla 6. Coste de recursos del controlador de interrupciones.

Número de interrupciones	Slices	Slice Flip-Flops	LUTs	Frecuencia Máxima (MHz)
8	169	145	138	167

3.4 Controlador JTAGPPC

El controlador JTAGPPC consiste en un *wrapper* para la primitiva JTAGPPC, que permite conectar el JTAG del PowerPC a la cadena JTAG de la propia FPGA [46].

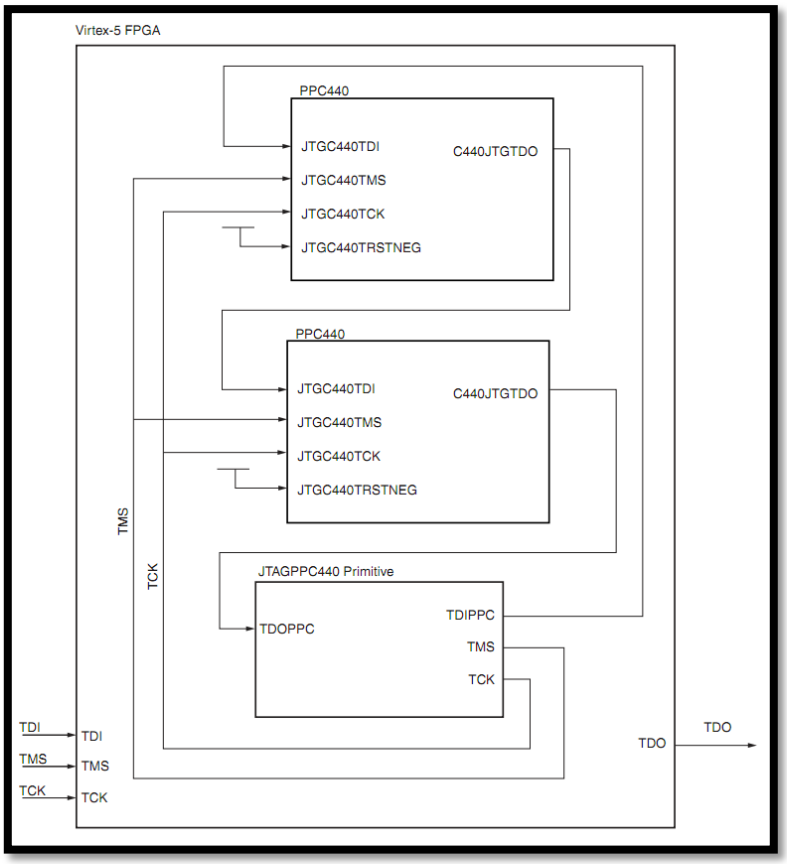


Figura 56. Conexión en cadena de JTAG para PowerPC440.

El bloque de procesador tiene puertos dedicados que ayudan a la depuración del *software* del sistema. Estos puertos, pueden conectarse directamente a pines de la FPGA y depurar así el *software* a través de un puerto de la placa de prototipado dedicado exclusivamente para ello. También es posible conectar el microprocesador al resto de la cadena de depurado JTAG de la FPGA, de forma que se puede depurar desde el mismo puerto con el que se configura el dispositivo programable.

Es posible conectar hasta dos PowerPC 440 (ver figura 56) para aquellos dispositivos que lo soporten, realizando automáticamente la conexión de los puertos JTAG a la cadena global del dispositivo.

Existen otras soluciones de conexión de los puertos de depuración de los dos PowerPC 440, además de hacer conexión directa de cada uno de ellos hacia pines de la FPGA, pueden encadenarse entre ellos, sin conectarse al resto de la FPGA, o multiplexarlos para compartir un único grupo de pines de la FPGA para ambos microprocesadores. Este mismo controlador es usado en las familias de FPGAs con PowerPC 405 empotrados, como son algunas Virtex-4.

3.5 Controlador Ethernet

Se trata del controlador de acceso al medio (MAC) de Ethernet. El bloque IP TEMAC –Tri-mode Ethernet MAC– presenta soporte para tres velocidades de transmisión: 10/100/1000 Mbps (Ethernet, Fast Ethernet y Gigabit Ethernet) [47].

El bloque tiene dos interfaces de comunicación, una de control y otra de datos. La interfaz de control se conecta como esclava al bus PLB del sistema, mientras que la interfaz de datos se conectará a uno de los bloque DMA del bloque de procesador usando una interfaz LocalLink.

Las características principales de este bloque se enumeran a continuación:

- Dispone de FIFOs independientes de transmisión y recepción de longitud configurable: 2K, 4K, 8K, 16K o 32K.
- Filtrado de tramas corruptas o erróneas.
- Soporte para un gran número de interfaces de capa física (PHY).
- Acceso a los registros del chip PHY a través de la interfaz MII.
- Operaciones *Full-Duplex*.
- Soporte opcional para tramas *jumbo*.
- Opción para realizar *checksum* parcial en hardware.
- Soporte para tramas VLAN.
- Opción para etiquetar, decapar y traducción de VLAN.
- Soporte para pausa de tramas para control de flujo.
- Opción de filtrado extendido para tramas *multicast*.
- Opción de obtención de estadísticas.
- Una o dos interfaces Ethernet *Full Duplex* con interfaz de control compartida e interfaces independientes de datos e interrupciones.

En las FPGAs Virtex-5 existen recursos físicos correspondientes a EMACs (Ethernet Media Access Controller). El bloque IP que se presenta se trata de un *wrapper* que consume recursos programables extras para facilitar el uso de los EMACs, añadiendo FIFOs y registros de control.

En la figura 57 se muestra un diagrama de bloques del *wrapper*, en el que se señalan los elementos *hardware* englobados, así como los recursos añadidos como las FIFOs y ciertos registros de control, además de funcionalidad para VLAN. En la parte superior se encuentra la lógica dedicada a integrar la interfaz con el bus PLB, dando acceso al microprocesador a leer y escribir en los registros internos del TEMAC, así como a los registros de configuración DCR y de estadísticas. En la parte inferior se representan dos canales simétricos, una para cada instancia EMAC interna (un bloque TEMAC puede integrar dos interfaces de red), seguidas de un filtro de Cheksum o buffer VLAN, en función de que opción haya sido activada. Por último, en el límite derecho se encuentra el wrapper que instancia los recursos específicos como son los EMACs integrados en la FPGA, así como bloques de entrada salida (IOBs).

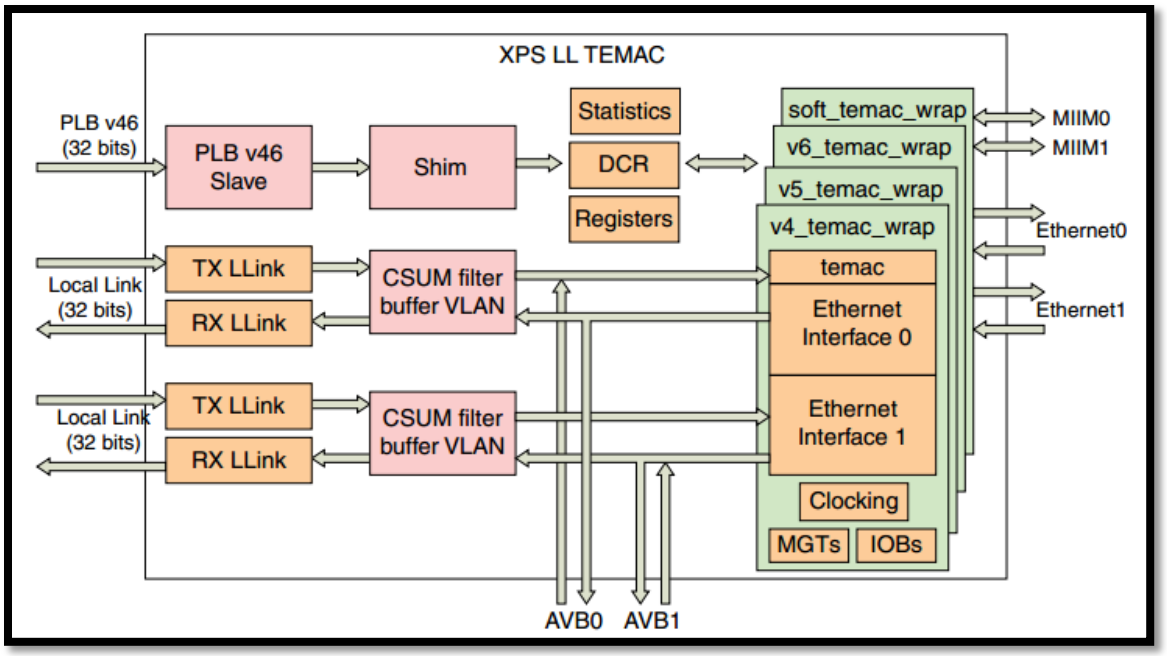


Figura 57. Diagrama del controlador Ethernet.

Las interfaces que soporta son MII (Media Independent Interface) para 10/100 Mbps (Ethernet y Fast Ethernet), 1000 Base-X para 1000 Mbps y las interfaces GMII (Gigabit Media Independent Interface), RGMII (Reduced Gigabit Media Independent Interface) versiones 1.3 y 2.0 y SGMII (Serial Gigabit Media Independent Interface) para las tres velocidades 10/100/1000 Mbps.

Además del soporte para las interfaces PHY comentadas, puede incluirse lógica adicional (consumiendo recursos programables de la FPGA) para una comprobación del *checksum* en *hardware*, liberando de carga al microprocesador. También tiene la opción de incluir lógica adicional para dar soporte a tramas de VLAN, realizando la traducción y etiquetado. Sin embargo, no puede incluirse a la vez que el soporte para *checksum*.

Al igual que ocurría con los bloques anteriormente descritos, el consumo de lógica adicional varía notablemente en función de los parámetros seleccionados para los bloques IP. La tabla 7 resume el consumo de recursos orientativo para los parámetros escogidos.

Tabla 7. Configuración del controlador Ethernet y coste de recursos utilizados.

Parámetro	Descripción	Valor
C_TEMAC1_ENABLED	Indica si se hará uso o no del segundo EMAC que puede albergar el *wrapper*	**0**
C_TEMAC_FIFO	Tamaño de las FIFOs, uno por cada sentido de la transmisión	4096
C_TEMAC_CSUM	Define si se realiza el *checksum* en hardware o software	0
C_TEMAC_VLAN	Define si soporta o no VLANs	0
C_TEMAC_TYPE	Define el tipo de TEMAC a usar: 0 = Virtex 5 Hard 1 = Virtex 4 Hard 2 = Soft 3 = Virtex 6 Hard	0
Slices	Número de *slices* usados	880
Flip-Flops	Número de Flip/Flops usados	1573
BRAMs	Número de memorias de bloque de tipo BRAM usados	18
LUTs	Número de LUTs usadas	1570
BUFGs	Número de Buffers de tipo G usados	6

Para la solución planteada en el presente PFC se ha decidido usar una única interfaz Ethernet, por lo que el parámetro C_TEMAC1_ENABLED se mantendrá a *0*. En cuanto al tamaño de las FIFOS, se establece 2KB como un tamaño suficiente para tramas con una longitud máxima de 1518 bytes, y 4KB como aquel que proporciona un mejor rendimiento. Si se fuesen a usar tramas *jumbo* sería necesario un tamaño de FIFO de 16KB o 32KB.

Puesto que no se usarán VLAN para la recepción de eventos, también se ha deshabilitado esta opción en el controlador. Por último, el parámetro "*C_TEMAC_TYPE*" indica si el controlador usa un EMAC hardware integrado en la FPGA, o si por el contrario, en caso de no disponer de dicho recursos (por ejemplo, si la FPGA fuese una Spartan-6) se hace uso de un *soft-IP* que usa CLBs para implementarlo. En el caso de la Virtex-5, se usan los recursos integrados en el dispositivo.

3.6 Controlador de interfaz System ACE

El controlador System ACE representa la interfaz entre el bus PLB y el microprocesador de interfaz (MPU) del periférico System ACE Compact Flash. El objetivo para incluir este bloque es proporcionar la funcionalidad de configurar la FPGA a partir de un fichero almacenado en una tarjeta externa de tipo Compact Flash. El circuito integrado encargado de acceder a la tarjeta se encuentra disponible en la placa de prototipado, siendo el controlador descrito el encargado de

servir de interfaz entre dicho circuito externo y el bus integrado en la FPGA. El circuito externo tiene acceso a los pines JTAG de la FPGA con el fin de cargar la configuración de la misma. A través de la conexión PLB se carga la aplicación *software* del PowerPC al arrancar el sistema [48].

En la figura 58 se presenta el diagrama de bloques del controlador.

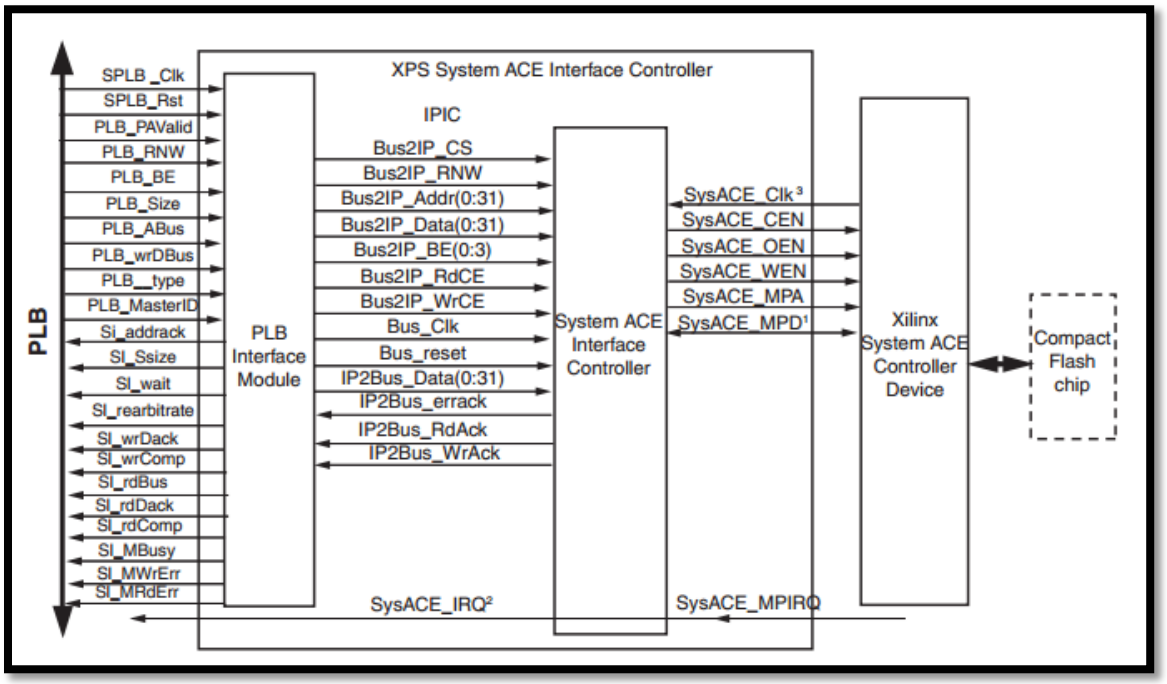


Figura 58. Diagrama del controlador de interfaz System ACE.

Como puede observarse, el controlador está formado por dos módulos principales. Por una parte, el *PLB Interface Module* facilita la conexión al bus PLB disponible en la plataforma. El segundo módulo, *System ACE Interface Controller*, por su parte facilita la conexión con el dispositivo System ACE externo.

El coste aproximado de recursos programables para este bloque es de 310 Flip-Flops y 112 LUTs.

3.7 Transceptor serie UART

Para poder monitorizar el sistema, se ha optado por usar una comunicación serie, ya que se carece de un controlador de pantalla. Así, cualquier mensaje del sistema será enviado a través del puerto serie a un equipo de monitorización [49].

Este periférico, como los demás, tiene una interfaz esclava PLB, y dispone de diferentes atributos configurables, como son por ejemplo, la velocidad de transmisión, el uso de paridad, etc. Las características principales del bloque son:

- Soporta interfaz de bus de 8 bits.
- Un canal de transmisión y otro de recepción (*Full-Duplex*).
- FIFO de transmisión y FIFO de recepción, ambas de 16 caracteres.

- Diferente número de bits por carácter configurable (5-8).
- Bit de paridad configurable (par o impar).
- Configurable velocidad de transmisión.

El bloque IP realiza la conversión paralelo-serie de los datos recibidos por el bus PLB, y serie-paralelo de los datos recibidos a través del periférico serie.

El dispositivo puede ser monitorizado a través de los registros internos. Además, genera una interrupción cuando existen datos en la FIFO de recepción, o cuando la FIFO de transmisión se vacía. El diagrama de bloques se presenta en la figura 59.

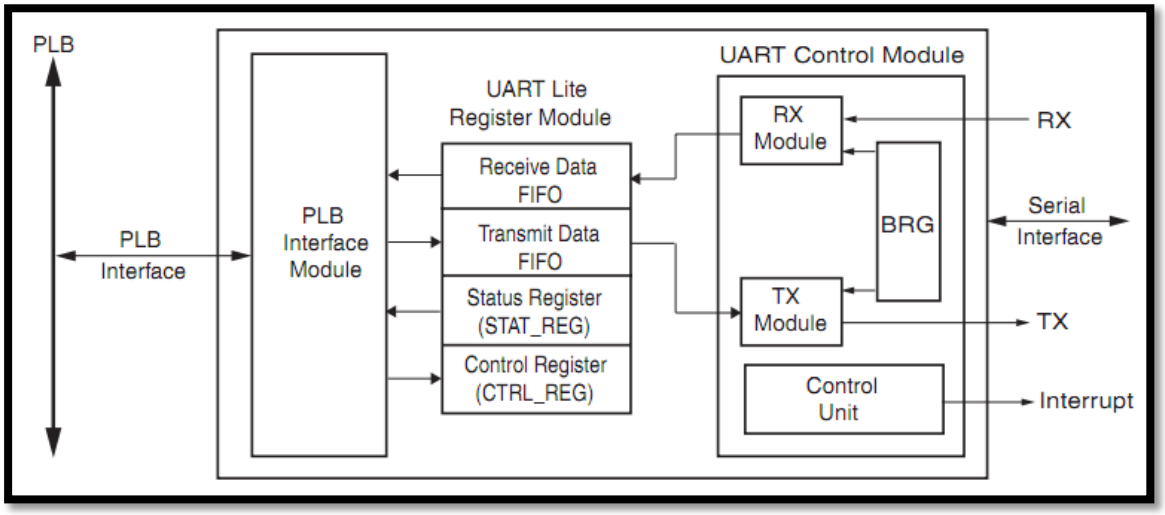


Figura 59. Diagrama de bloques de la UART.

Existen tres módulos internos. El primero es el de la interfaz del PLB, que implementa el protocolo del bus PLB, y almacena o lee los datos de las FIFOs o de los registros internos. El segundo consiste en el conjunto de registros internos y de las FIFOs, de transmisión y recepción. Por último, el tercer bloque es el que realmente hace la labor de transmitir y recibir los datos en formato serie.

3.8 Generador de reloj

En un sistema de esta complejidad es necesario un conjunto de fuentes de reloj con diferentes características de frecuencia y fase. El generador de reloj es un bloque que tiene por objeto generar, a partir de una fuente de reloj común, todas las señales de reloj para el resto de bloques [50].

Este bloque se implementa mediante recursos especiales disponibles en la FPGA. En particular usa los DCMs (Digital Clock Manager), PLLs (*Phase-Locked Loop*) y MMCMs (*Mixed-Mode Clock Manager*) de la FPGA.

Durante la implementación, la herramienta de diseño de la plataforma (Xilinx Platform Studio) mostrará un error en caso de que no sea posible generar la señal de reloj propuesta. En la figura 60 se muestra el diagrama de bloques del generador de reloj.

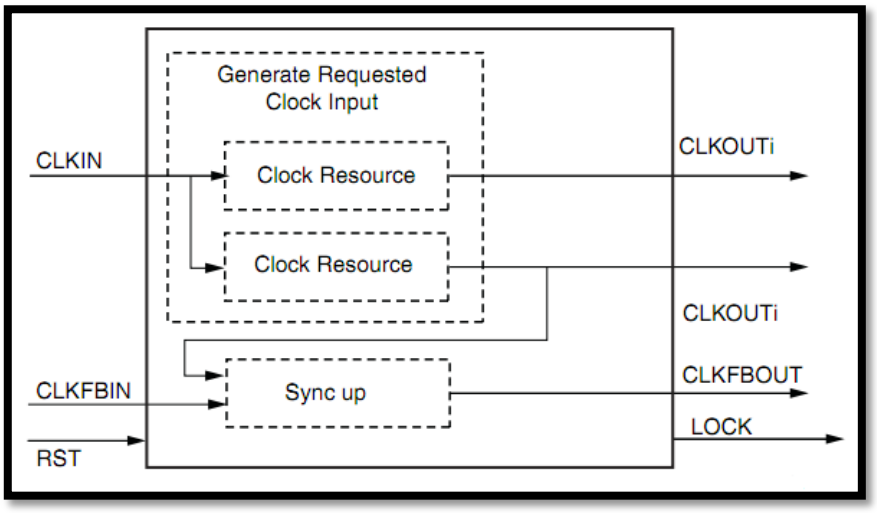


Figura 60. Diagrama de bloques del generador de reloj.

3.9 Analizador lógico integrado

Se trata de un bloque, implementado a partir de lógica configurable, que sirve para monitorizar las señales internas del sistema, como lo haría un analizador lógico. Al ser un bloque síncrono, se le aplicarán las mismas restricciones temporales que al resto del diseño [51].

Algunas de sus características más destacables son:

- Provee de comunicación entre el *software* de **ChipScope Analyzer** y los bloques de captura mediante los bloques de control (ICON).
- Se pueden definir distintos puertos de disparo y su ancho, así como el ancho del bus de datos y la longitud de captura.
- Se pueden combinar los puertos de disparo para formar condiciones de disparo complejas.

Las señales del sistema se conectan a las entradas del bloque ILA (Integrated Logic Analyzer), y estas se capturan en tiempo real, a la frecuencia del diseño que las contenga. Antes de que el sistema sea implementado, se seleccionarán el número de señales a capturar así como el número de muestras a capturar. La comunicación con el bloque ILA se realiza a través del puerto JTAG de la FPGA y a través del bloque de control ICON, tal como se muestra en la figura 61.

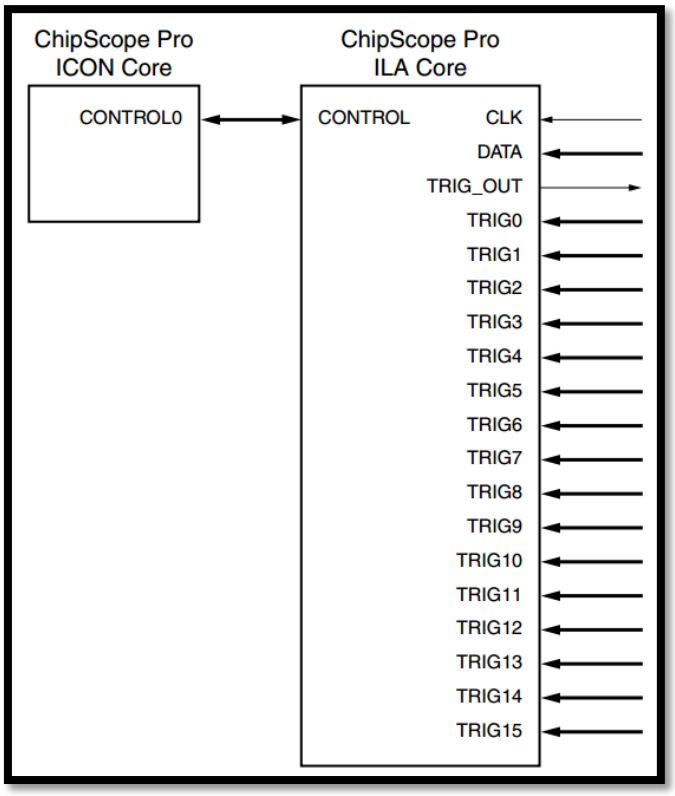


Figura 61. Diagrama de bloques ILA-ICON.

Además de la captura de la señal de entrada de datos, todas las señales usadas como condicionantes del disparo también son capturadas, pudiendo así analizar un gran número de señales simultáneamente. El ancho de cada una de ellas es, a su vez, configurable para adaptarse a las condiciones del diseño objeto del análisis.

La cantidad de recursos consumidos por este bloque se verá afectada por los parámetros de configuración utilizados. Por un lado, las muestras capturadas se almacenan inicialmente en memorias de bloque BRAM internas de la FPGA a la velocidad con la que el diseño las genera, para luego ser enviadas al software de análisis a través del puerto JTAG de la FPGA (de esta forma, no se usan otros pines de la FPGA). Por ello, un incremento en el número de muestras capturadas por disparo incrementa el costo de recursos de BRAMs del bloque en cuestión. Es responsabilidad del diseñador, definir, en función del tipo de señal que pretenda capturar, el número de muestras que necesita para la verificación del sistema. Si, por ejemplo, se pretende capturar transacciones de datos de 300 palabras, no será necesaria una longitud de captura mayor a 512.

Por otro lado, el consumo de lógica programable (LUTs y Flip-Flops) dependerá del tipo de test de condiciones realizado, que puede ser básico (solo se aceptan patrones del tipo igual o distinto) o extendido, que añaden también condiciones del tipo mayor que, menor que, mayor o igual que, y menor o igual que. Estas últimas son útiles para el caso de usar como señal de disparo un bus de datos de valores numéricos.

3.10 Controlador de bloques del analizador lógico

Realiza la tarea de comunicación entre los bloques del analizador lógico integrado (ILA) y el bloque JTAG Boundary Scan de la FPGA, para poder controlar dichos bloques a través del puerto de programación del dispositivo [52].

En total, un controlador (ICON) puede conectarse a un máximo de 15 bloques ILA. En función del número de bloques a los que se conecta, su consumo de lógica programable varía. Para el diseño propuesto en el presente PFC se ha usado un único ILA, por lo que con un único puerto de control del ICON es suficiente. Sus costes de recurso aproximados son:

Tabla 8. Coste de recursos del bloque de control ICON.

LUTs	Flip-Flops	DSP Slices	BlockRAMs	Frecuencia Máxima
90	108	0	0	398 MHz

4 Interfaces de comunicación

En el diseño de un sistema electrónico, tan importante son los bloques que lo componen, como el protocolo usado para la comunicación entre ellos. En este apartado se describirán los protocolos de comunicación disponibles en la plataforma *hardware* diseñada.

A la hora de insertar un bloque IP específico, Xilinx proporciona dos tipos de protocolos de alta velocidad para su interconexión a la plataforma. Por una parte, permite la conexión al bus PLB mediante el uso de un asistente para el diseño de un *wrapper* de una interfaz PLB. Por otra, es posible utilizar un bloque DMA del bloque de procesador mediante el protocolo LocalLink.

En el presente PFC se ha optado por conectar el coprocesador de eventos diseñado a través de un bloque DMA. Esto proporciona una conexión de alta velocidad y ancho de banda entre el PowerPC y el CE. Esta solución permite al PowerPC seguir ejecutando instrucciones mientras el bloque DMA realiza la transmisión, mejorando así el rendimiento del sistema. Por otra parte, se libera al bus PLB para realizar otro tipo de transacciones, tal como se verá más adelante.

4.1 Processor Local Bus (PLB)

Se trata de una arquitectura de bus para microprocesadores diseñada por IBM para los diseños de tipo *System-on-Chip*. Para el diseño de la plataforma propuesta en este PFC, el microprocesador usado es un IBM PowerPC 440, preparado para diseños con bus PLB, por lo que la mayoría de periféricos están interconectados mediante este tipo de solución [53].

Como se ha descrito anteriormente, la mayoría de los IPs dispone de interfaces esclavas PLB, de forma que el esqueleto central del diseño lo forman las conexiones del bus y su árbitro. Su interconexión se realiza de forma muy simple en el entorno XPS de Xilinx.

4.2 LocalLink (LL)

LocalLink es una especificación de protocolo de comunicación propuesto por Xilinx para transacciones de datos en forma de tramas. En particular, es de interés para este PFC debido a que la conexión entre los bloques DMA del bloque de procesador y CE al que se conecta se hace con esta especificación. Se trata de un protocolo síncrono y punto a punto [54].

Una interfaz LocalLink se compone de un conjunto de señales de control, un puerto de datos, además de las señales de reloj y de *reset*. La comunicación es *simplex* (en un solo sentido), siendo necesarios dos canales LocalLink, simétricos, para conseguir una comunicación *full-duplex*.

Las señales de control son en ambos sentidos para cada uno de los canales, ya que, como se comentará posteriormente, el destino de la transmisión avisará a la fuente si está, o no, preparado para la recepción de datos.

A continuación se muestra un ejemplo básico de una comunicación LocalLink en un único sentido:

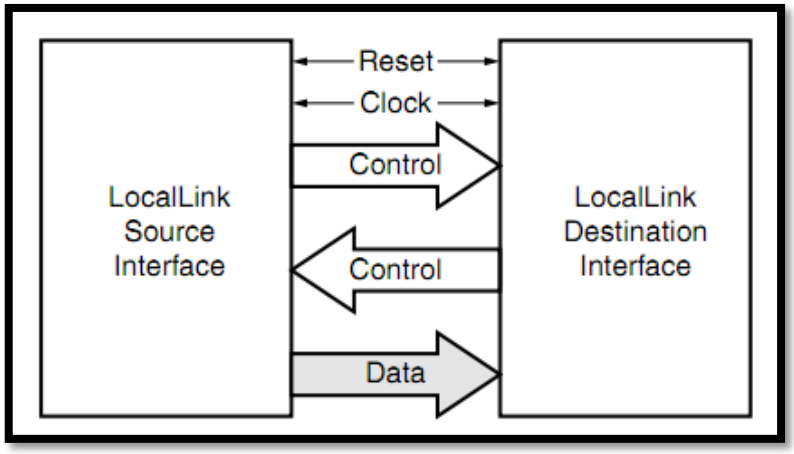


Figura 62. Ejemplo de comunicación LocalLink.

La especificación LocalLink es genérica, pudiéndose parametrizar para cada uso específico. De esta forma, una trama de LocalLink está formada por una cabecera, una carga útil o *payload* y una cola, siendo la cabecera y la cola opcionales.

A continuación se enumeran las señales que obligatoriamente deben aparecer en un canal unidireccional LocalLink genérico.

Tabla 9. Señales obligatorias de una interfaz LocalLink.

Señal	Dirección	Descripción
DATA	Fuente → Destino	Bus de datos de ancho parametrizable.
SRC_RDY_N	Fuente → Destino	Señal activa a nivel bajo. Indica si la fuente está preparada para enviar datos.
DST_RDY_N	Fuente ←Destino	Señal activa a nivel bajo. Indica si el destino está preparado para recibir datos.

SOF_N	Fuente → Destino	Señal activa a nivel bajo. Indica el comienzo de una trama.
EOF_N	Fuente → Destino	Señal activa a nivel bajo. Indica el fin de una trama.

En función de las opciones que se quieran usar de LocalLink, pueden ser, además, necesarias otras señales de control, que se comentan a continuación.

Tabla 10. Señales opcionales de una interfaz LocalLink.

Señal	Dirección	Descripción
SOP_N	Fuente → Destino	Señal activa a nivel bajo. Indica el comienzo de la carga útil.
EOP_N	Fuente → Destino	Señal activa a nivel bajo. Indica el final de la carga útil.
REM	Fuente → Destino	Indica los bytes útiles de la última palabra del bus de datos. Su ancho es de un bit por cada byte del bus de datos.
REM_SOF	Fuente → Destino	Indica el primer byte que corresponde al comienzo de la trama.
REM_SOP	Fuente → Destino	Indica el primer byte que corresponde al comienzo de la carga útil.
REM_EOP	Fuente → Destino	Indica el último byte que corresponde al final de la carga útil.
SRC_DSC_N	Fuente → Destino	Indica que la fuente está cancelando la actual trama por error o otras causas.
DST_DSC_N	Fuente ← Destino	Indica que el destino está cancelando la actual trama por insuficiente espacio o otras causas.
CH	Configurable	Indica el canal, en implementaciones en las que se pueda leer o escribir de diferentes canales.
CH_RDY_N	Configurable	Indica si el canal está disponible.
CH_SWAP_LOOKAHEAD_N	Configurable	Indica que el canal está a punto de vaciarse.
PARITY	Fuente → Destino	Indica la paridad calculada sobre todo el bus de datos.
PARITY_ERR_N	Fuente ← Destino	Indica que se ha detectado un error de paridad en el anterior dato.

4.2.1 Interfaz LocalLink de un bloque DMA

Como se indicó anteriormente, los bloques DMA del procesador usan un protocolo de comunicación LocalLink para enviar y recibir tramas de los periféricos. En este apartado se presentan los parámetros específicos de la implementación [55].

En particular, cada bloque DMA tiene dos interfaces LocalLink, uno para transmisión y otro para recepción, además de las señales de reloj y reset, comunes a ambas interfaces. En la figura 63 se muestra el diagrama de bloques de un canal DMA.

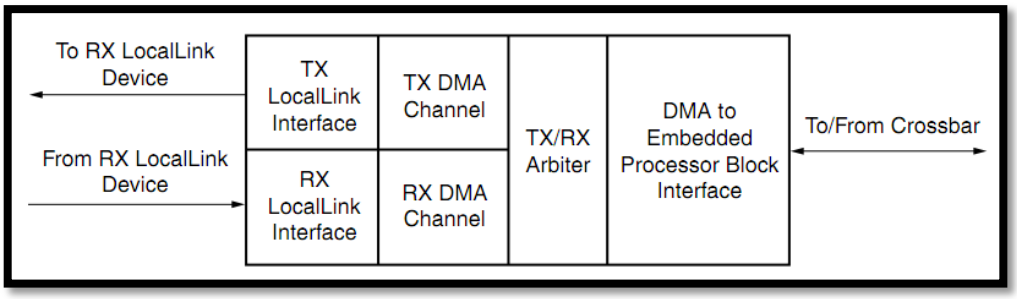


Figura 63. Interfaces LocalLink de un bloque DMA.

El dispositivo de transmisión del bloque DMA recoge los datos de una o varias regiones de la memoria, creando la carga útil a transmitir por la interfaz LocalLink de transmisión. Por su lado, el dispositivo de recepción recibe los datos desde la interfaz LocalLink de recepción y los almacena en una o varias regiones de la memoria. Estas zonas de memoria son señaladas por unos registros de control llamados descriptores.

Cada canal DMA es controlado por descriptores independientes, y sus correspondientes estructuras de datos, inicializadas por el PowerPC antes de que las operaciones del DMA comiencen. Entre otras cosas, estos descriptores controlan cuántos datos serán transferidos y la localización de los datos en el sistema de memoria.

Los descriptores pueden estar encadenados, permitiendo formar una secuencia de bloques de memoria separados, que se combinan en un único paquete al ser transmitido. De la misma forma, se puede dividir el paquete recibido en una secuencia de bloques de memoria separados.

El PowerPC inicializa el DMA creando una secuencia de descriptores en memoria, y luego escribiendo la dirección del primer descriptor en el registro DCR de Puntero de Descriptor Actual. Finalmente, la CPU comienza las operaciones del DMA escribiendo la dirección del último descriptor de la secuencia en el registro DCR de Puntero de Descriptor de Cola.

Esta última escritura hace que el DMA busque un nuevo descriptor de la dirección apuntada por el registro Puntero al Descriptor Actual. En el caso del canal de transmisión, el DMA comienza por buscar los datos de la dirección indicada en el descriptor y comienza el proceso de creación y envío de un paquete de datos. Después de transmitir todos los datos indicados por el descriptor actual, el DMA busca el siguiente descriptor, si existe, y continúa enviando los datos indicados por este descriptor.

En el caso del canal de recepción, el DMA espera por los datos recibidos del periférico externo, y comienza a copiarlos en la región de memoria apuntada por el descriptor. Si se reciben más datos, se busca el nuevo descriptor y se escriben en la región indicada por este. Este proceso continua hasta el final de la carga útil recibida. En la tabla 11 se muestra el formato de un descriptor.

Tabla 11. Formato de un descriptor de DMA.

#Palabra	Byte Offset	Descripción			
		Byte 3	Byte 2	Byte 1	Byte 0
0	0x00	Puntero al siguiente descriptor			
1	0x04	Dirección del buffer de datos			
2	0x08	Longitud del buffer de datos			
3	0x0C	Est/Cont	Datos definidos de la aplicación		
4	0x10	Datos definidos de la aplicación			
5	0x14	Datos definidos de la aplicación			
6	0x18	Datos definidos de la aplicación			
7	0x1C	Datos definidos de la aplicación			

El campo **puntero al siguiente descriptor** contiene la dirección del siguiente descriptor en el anillo de descriptores del canal. Este campo debe estar alineado a ocho palabras, es decir, sus últimos cinco bits deben ser cero. Por su parte el campo **dirección del buffer de datos** contiene un puntero al comienzo del bloque de datos al que apunta el descriptor, y su longitud se almacena en la siguiente palabra del descriptor. En la siguiente palabra, el primer byte contiene ocho bits de estado y control, entre los que se destacan el SOP (*Start of Payload*) y el EOP (*End of Payload*), los cuales indican si el descriptor actual es el comienzo y/o el final de un paquete. El resto de datos del descriptor pueden ser usados por la aplicación para diversas funciones. En la figura 64 se muestra un diagrama con las señales que aparecen en la interfaz LocalLink de transmisión del DMA.

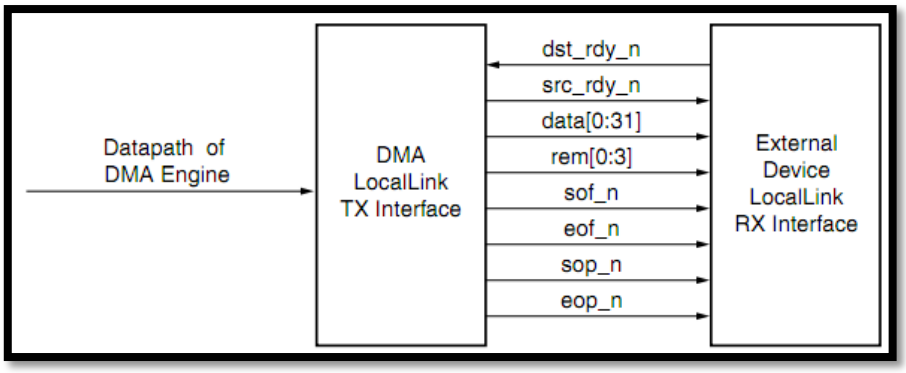


Figura 64. Diagrama de bloques de la interfaz de transmisión LocalLink de un DMA.

Como se observa en la figura 64, el bus datos para esta implementación de LocalLink tiene un ancho de 32 bits, y la señal REM es de 4 bits hace referencia a cada byte del bus de datos. Cada transmisión a través del LocalLink en el sentido origen DMA y destino el CE, está compuesta por una cabecera de ocho palabras, seguido de la carga útil. En la figura 65 se muestra la composición de un paquete.

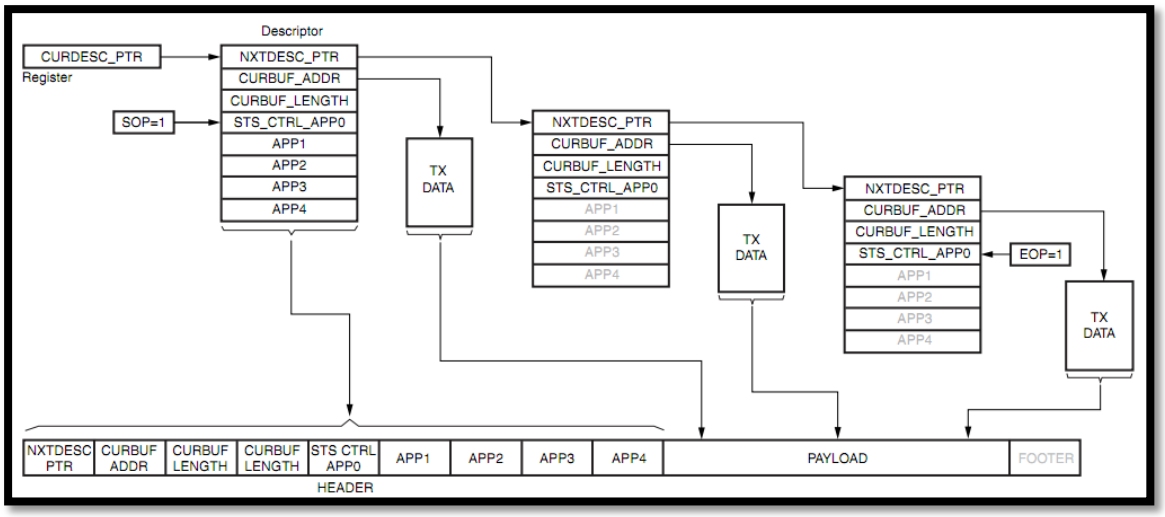


Figura 65. Creación de un paquete LocalLink.

En el caso de las transmisiones, no se envía cola en el paquete, mientras que la cabecera está formada por el primer descriptor. Como se observa de la figura 65, de los tres descriptores, el primero tiene el bit de comienzo de la carga útil a 1 (SOP=1), mientras que el tercero tiene el de fin de la carga útil (EOP=1). En la figura 66 se representa el diagrama temporal de una transmisión de LocalLink, en el que se observan los instantes temporales en los que se activan las señales de control.

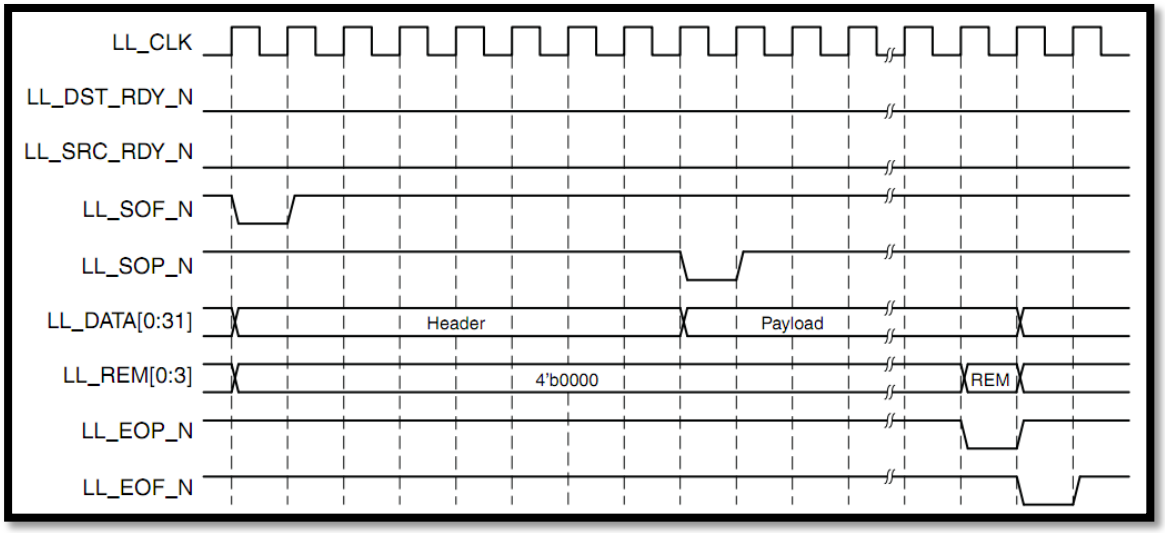


Figura 66. Diagrama temporal de una transmisión LocalLink de un DMA.

La interfaz de recepción LocalLink de un DMA es simétrica al de transmisión, siendo el periférico el que envía los datos hacia el bloque DMA. En la figura 67 se muestran las señales usadas.

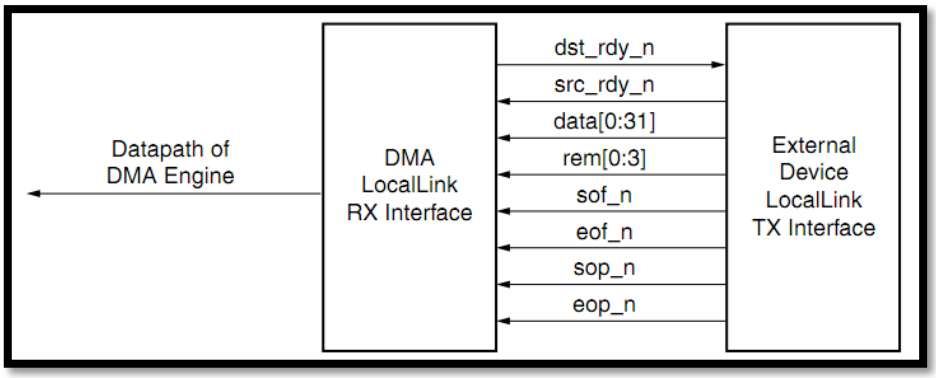


Figura 67. Diagrama de bloques de la interfaz de recepción LocalLink de un DMA.

Al contrario de lo que ocurre en las transmisiones LocalLink, los paquetes recibidos no tienen cabecera. En su lugar, están formados por la carga útil seguida de una cola de ocho palabras, compuesta también por los campos de un descriptor. A la recepción, los datos de la cola son obviados por el DMA, excepto aquellos definidos por la aplicación (los últimos 19 bytes). La figura 68 muestra una recepción de una trama que se divide en tres regiones de memoria indicadas por tres descriptores, y donde los datos recibidos en la cola se almacenan en el último descriptor. La figura 69 muestra el diagrama temporal del protocolo en modo recepción.

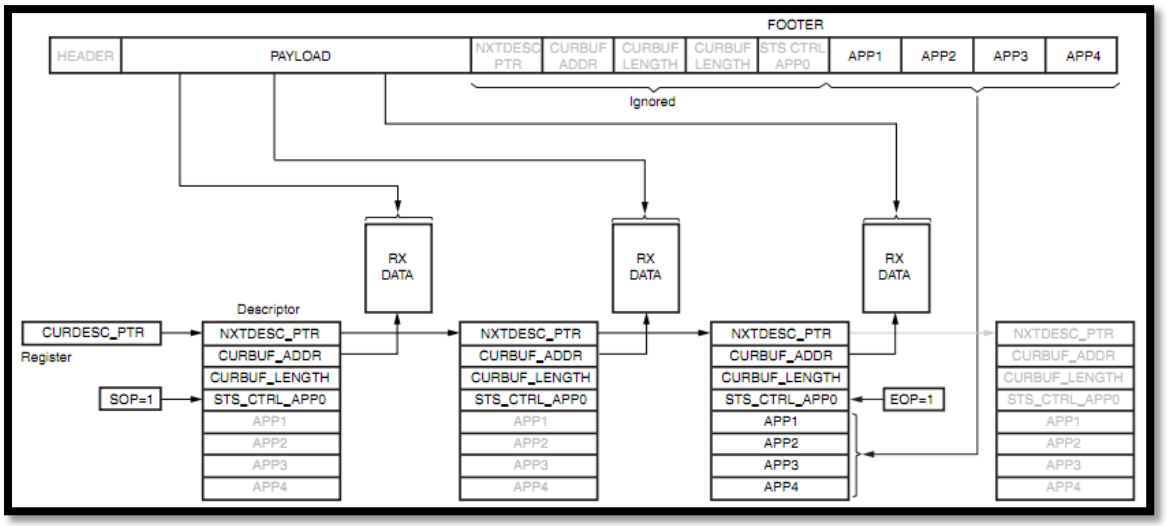


Figura 68. Partición de un paquete LocalLink.

Para ambos tipos de comunicación (transmisión y recepción), la señal REM indica cuales de los 4 últimos que bytes presentes en el bus de datos son válidos, ya que no ocurrirá siempre que los datos a transmitir estén alineados a palabras completas (tamaño múltiplo de 4). Así, cada bit de esta señal está asociado a uno de los bytes del bus de datos, siendo activa a nivel bajo, y pudiéndose solo recibir de forma consecutiva, es decir, o solo el primer byte es válido, o los dos

primeros, o los tres primeros o todos ellos. La tabla 12 muestra un ejemplo de los posibles valores de REM y su significado.

Tabla 12. Significado de la señal REM.

Bytes recibidos (B = Dato válido, X = Dato no válido)	REM (binario)	REM (hexadecimal)
BXXX	0111b	0x7
BBXX	0011b	0x3
BBBX	0001b	0x1
BBBB	0000b	0x0

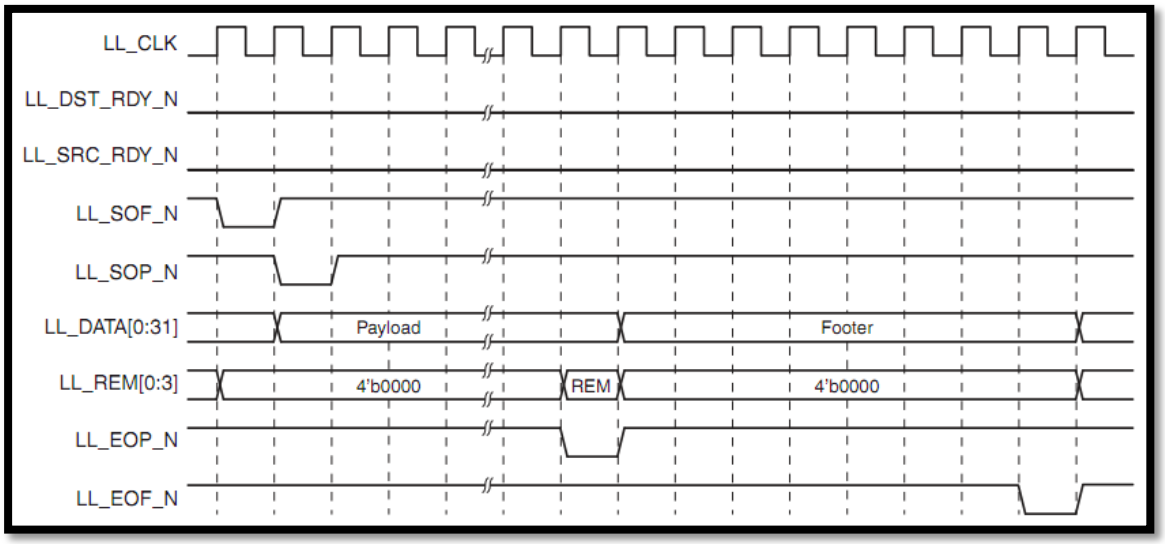


Figura 69. Diagrama temporal de una recepción LocalLink de un DMA.

Otra característica importante de las comunicaciones a través de LocalLink es que pueden ser pausadas y reanudadas en medio de un paquete. Estas pausas son controladas por las señales SRC_RDY_N y DST_RDY_N. Siempre que una de estas señales se desactive durante una transacción, significará que el dato en ese instante es inválido, y que el siguiente válido será aquel en el que la señal vuelva a activarse.

La causa para que el DMA deje de estar preparado pueden ser variadas, pero la más corriente es que se le retire su permiso para usar el *Crossbar*. Por otro lado, que el periférico deje de estar preparado para recibir datos puede deberse a que la FIFO de entrada se encuentre llena, al igual que puede dejar de estar listo para transmitir si la FIFO de salida se vacía.

El estudio realizado a nivel de ciclos de esta interfaz es muy importante para el desarrollo del presente proyecto, ya que el periférico debe cumplir minuciosamente todas las especificaciones, para una correcta comunicación con el bloque DMA en las transacciones de paquetes de datos.

5 Creación de la plataforma *hardware*

Ahora que se han explicado los diferentes IPs para implementar la funcionalidad de la plataforma, será necesario realizar su integración haciendo uso de la herramienta Xilinx Platform Studio. Durante la creación de un nuevo proyecto, esta permitirá hacer uso del asistente Base System Builder, el cual guiará al diseñador por con conjunto de pasos en el que definirá la plataforma básica sobre la que realizar un refinamiento más tarde así como integrar nuevos bloques IP.

5.1 Creación de la plataforma básica con el BSB

La primera de las opciones que el asistente permite elegir es el tipo de bus de comunicación. Aunque en esta memoria se ha hablado siempre del bus PLB, esto se debe a que el dispositivo FPGA usado contiene varios cores PowerPC 440 que incluyen este tipo de bus. Sin embargo, en las nuevas familias de FPGA de Xilinx, el tipo de conexión se ha modificado, siendo su actual estándar el AXI [56].

Una vez elegido el tipo de bus, será necesario especificar la placa de prototipado sobre la que se implementará el diseño. La herramienta permite elegir entre diferentes modelos de placas de diferentes fabricantes disponibles, así como la posibilidad de especificar una placa de desarrollo diseñada a medida en cuyo caso se especificará el modelo y encapsulado específico de la FPGA.

A continuación se especificará si el sistema a diseñar estará compuesto por un único procesador y sus periféricos, o por dos procesadores y sus correspondientes periféricos, independientes o compartidos. Es posible diseñar plataformas en el que coexistan microprocesadores, incluso de diferentes familias como pueden ser PowerPC y Microblaze. Para el presente proyecto se ha decidido hacer uso de un único procesador PowerPC de los disponibles en la FPGA. En la figura 70 se muestra el BSB en la que se realiza la selección del tipo de sistema (mono o multiprocesador).

Será necesario indicar la frecuencia de funcionamiento del procesador, el cual puede asignarse a un máximo de 400 MHz. Implantar la frecuencia máxima al PowerPC implica una reducción de la frecuencia del bus PLB hasta 100 MHz (para frecuencias más bajas del procesador, el bus PLB puede funcionar a 125 MHz), lo cual no es crítico para el diseño propuesto, ya que las transacciones críticas de datos se realizan a través de los bloques DMA y sus respectivas interfaces LocalLink. Se puede activar además aquí, la opción de añadir una unidad de coma flotante externa al bloque de procesador, conectado a este a través de la interfaz APU.

El siguiente paso para el diseño de la plataforma base es la inclusión de los controladores del sistema necesarios: *DDR2_SDRAM_DIMM0*, *Hard_Ethernet_MAC*, *RS232_Uart_1*, *SysACE_CompactFlash* y *xps_bram_if_cntlr_0*, activando además el uso del bloque DMA para el Hard_Ethernet_MAC.

Tras esta elección se activarán el uso de las cachés internas del PowerPC tanto para instrucciones como para datos, finalizando así el diseño de la plataforma base.

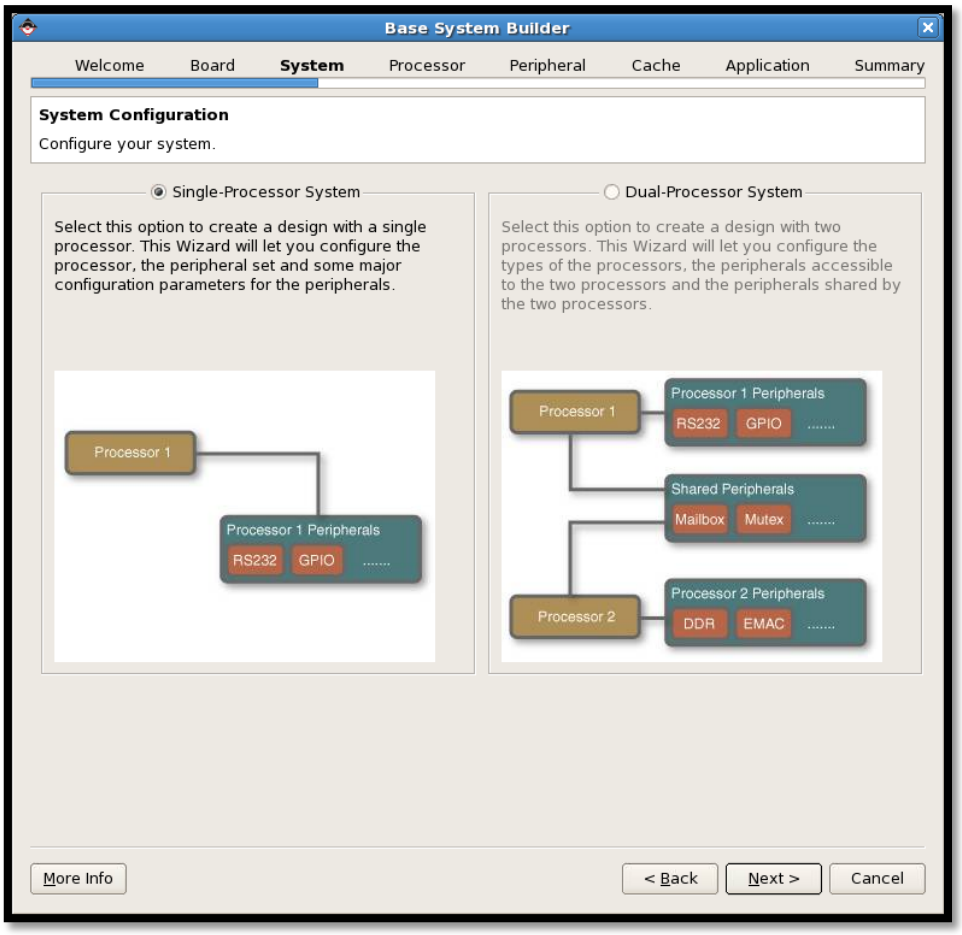


Figura 70. Selección del número de procesadores de la plataforma.

Con los pasos realizados, ya está disponible una plataforma *hardware* completamente sintetizable, sobre la cual puede ejecutarse una aplicación escrita en C/C++ y compilada con el compilador para el PowerPC 440. Tras realizar los pasos del BSB, la herramienta habrá añadido ciertos bloques IP necesarios para el correcto funcionamiento de la plataforma, como son el controlador de interrupciones (*xps_intc_0)*, el bloque de configuración vía JTAG (*jtagppc_cntlr_inst)*, el bloque de generación de reset (*proc_sys_reset_0)* y el generador de reloj (*clock_generator)*. Todos estos bloques ya han sido presentados anteriormente. En la figura 71 se muestran las instancias que aparecen como parte de la plataforma en este punto.

5.2 Creación e importación del periférico

Con el fin de poder depurar la plataforma al completo se ha decidido diseñar un pequeño y sencillo periférico capaz de recibir y enviar tramas a través de una interfaz LocalLink de forma que simule de forma muy básica al coprocesador de eventos que posteriormente se incluirá en este sistema [57].

En particular, se pretende que el periférico desarrollado reciba una trama a través del LocalLink, la almacene en una pequeña memoria interna, y una vez terminada la recepción, la vuelva a enviar en el mismo estado, a través de la interfaz LocalLink gemela.

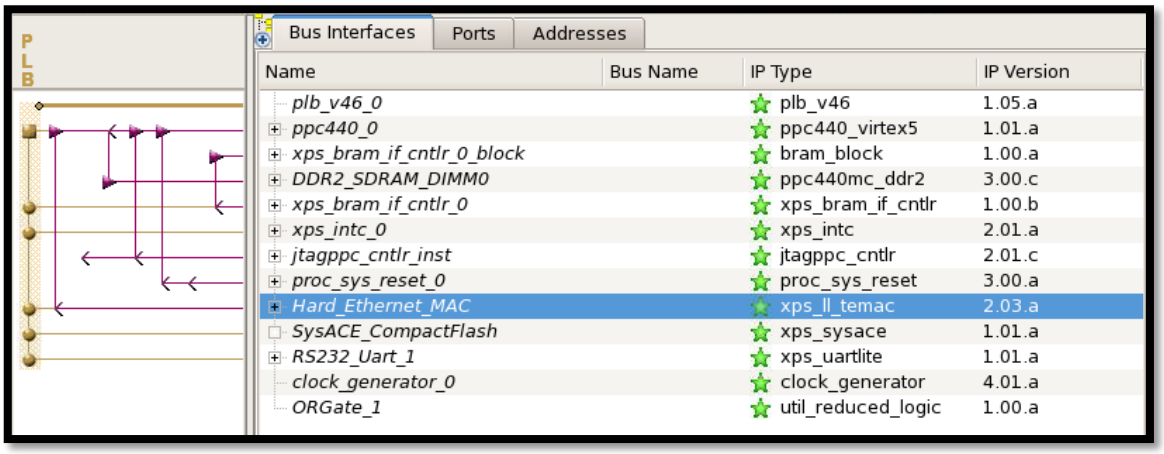


Figura 71. Instancias de la plataforma básica.

Atendiendo a los diagramas de tiempos de las especificaciones de LocalLink para el bloque DMA, el bloque estará compuesto por 9 estados, tal como se muestra en la figura 72.

El sistema comenzará en un estado **IDLE** en el que estará preparado para recibir, es decir, tendrá activa la señal **tx_dst_rdy_n**, pero no preparado para enviar, es decir, la señal **rx_src_rdy_n** estará desactivada. Es importante destacar que, por convención, se denota transmisión a una transacción en la que los datos tienen como fuente el bloque DMA y destino el periférico.

El bloque permanecerá en el estado **IDLE** hasta que el bloque DMA active la señal "fuente preparada para la transmisión", momento en el cual pasará al estado HEADER, donde se preparará para recibir la cabecera. En este estado permanecerá 8 ciclos, que es el número de palabras de la cabecera. Además, el contador solo se incrementará mientras la señal "fuente preparada" permanezca activa, ya que, como se comentó anteriormente, el bloque DMA puede pausar una transferencia y retomarla por el mismo ciclo.

El siguiente estado es el de la carga útil, el cual finalizará cuando se active la señal **EOP** (final de la carga útil), pasando a un estado de espera, hasta que el bloque DMA esté preparado para recibir los datos. Cuando lo esté, habrá dos estados consecutivos, el de comienzo de trama y el de comienzo de carga útil. Tras el envío de los datos, se enviará la cola que serán ocho palabras de ceros, ya que no se han utilizado los datos de aplicación de esta.

Para incorporar el coprocesador a la plataforma es necesario utilizar el asistente disponible en el XPS. Se indicarán los tipos de fichero que se vayan a vincular al IP (pueden haber descripciones HDL, *netlist* y ficheros de documentación), se indicarán si la descripción es en VHDL, Verilog o mixta, y se seleccionan los ficheros a añadir. De igual forma es necesario seleccionar las

interfaces de buses que tenga el periférico, entre las que el asistente reconoce: PLB, AXI, DCR. Como LocalLink no está entre las interfaces soportadas por el asistente, se desactivará la opción de buses y las de interrupciones. Es es el bloque DMA quien genera las interrupciones cuando recibe y envía tramas.

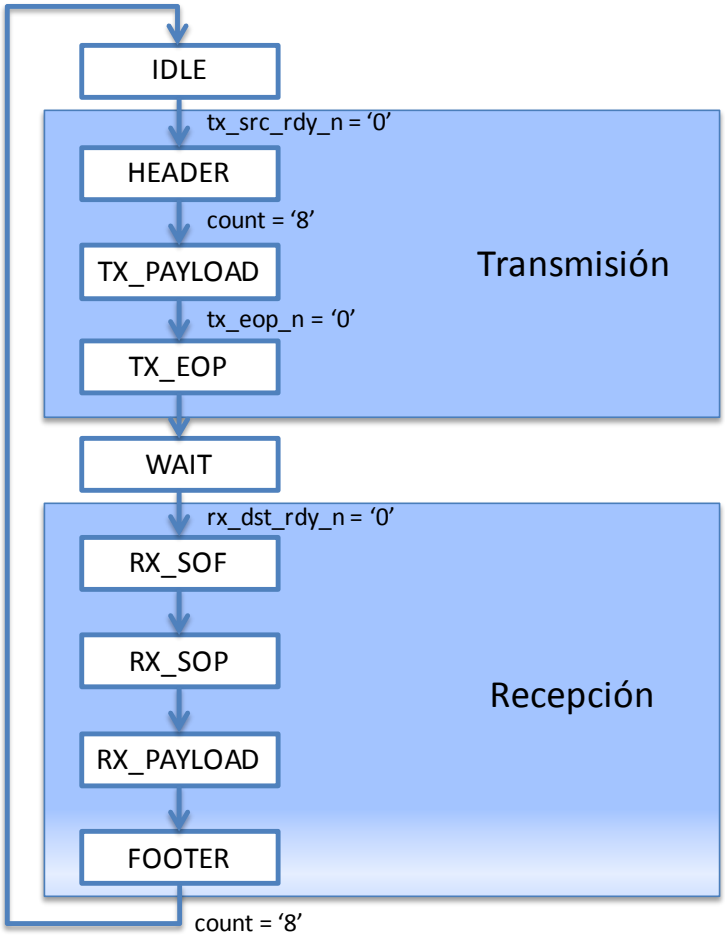


Figura 72. Diagrama de estados del bloque IP.

Una vez importado el periférico en el proyecto, se podrán incluir en el diseño. Para ello hay que definir la interfaz LocalLink como un bus en el fichero de definición de periférico del microprocesador (*Microprocessor Peripheral Definition – MPD*). Cada instancia de la plataforma tiene un fichero MPD asociado, que contiene sus parámetros, puertos, interfaces, etc. Al realizar la importación, la herramienta habrá creado un fichero MPD. Se deberá añadir la definición de la interfaz LocalLink, así como la asociación entre cada uno de los puertos, y los nombres específicos del bus (figura 73).

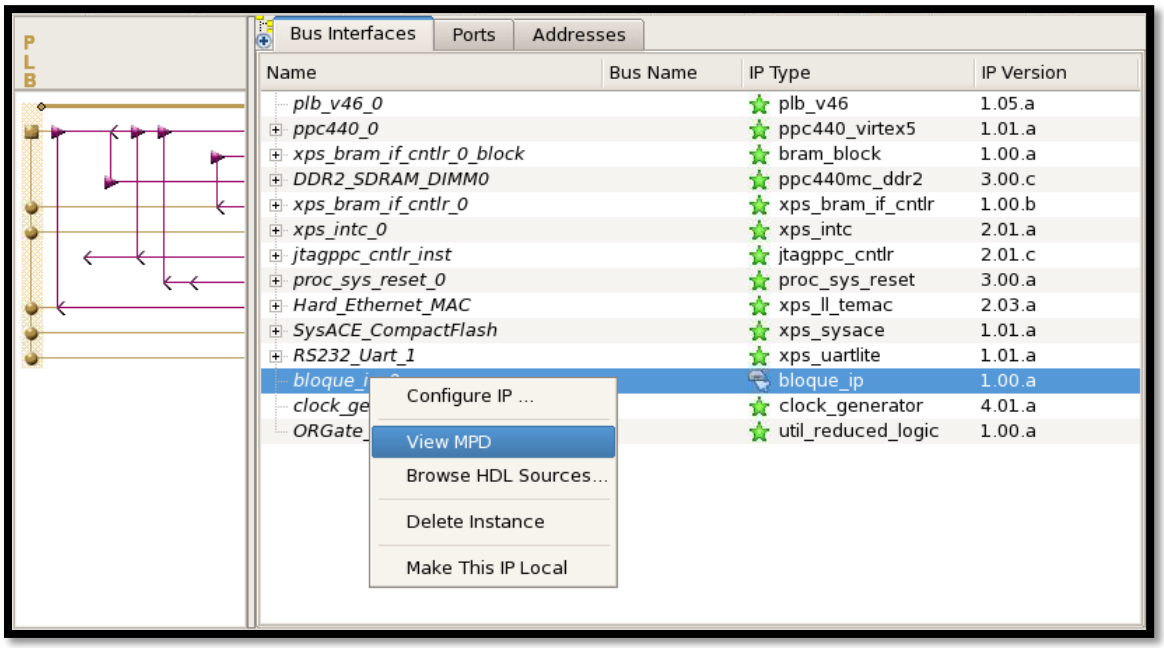


Figura 73. Fichero MPD de un periférico.

Para añadir la interfaz LocalLink, habrá que añadir la siguiente línea al fichero MPD.

```
BUS_INTERFACE BUS = LLINK0, BUS_STD = XIL_LL_DMA, BUS_TYPE = INITIATOR
```

Por otro lado, hay que vincular cada uno de los puertos del periférico, a la señal perteneciente a la interfaz. A continuación se muestran como quedan dichas sentencias.

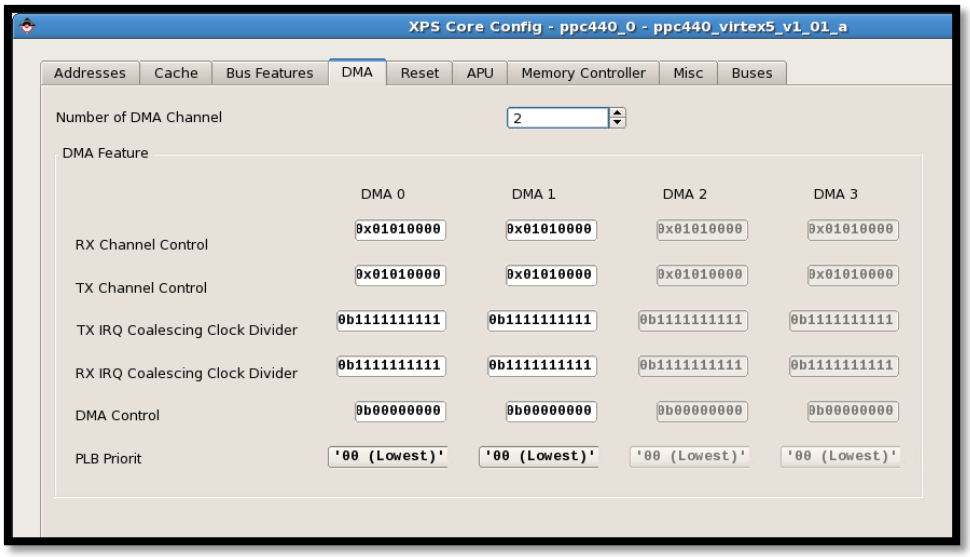
```
PORT Ll_Clk = "",                DIR = I, SIGIS = CLK, INITIALVAL = VCC
PORT Ll_Rst = "",                DIR = I

PORT tx_data = LL_Tx_Data,           DIR = I, BUS = LLINK0, VEC = [0:31]
PORT tx_rem = LL_Tx_Rem,             DIR = I, BUS = LLINK0, VEC = [0:3]
PORT tx_sof_n = LL_Tx_SOF_n,         DIR = I, BUS = LLINK0
PORT tx_eof_n = LL_Tx_EOF_n,         DIR = I, BUS = LLINK0
PORT tx_sop_n = LL_Tx_SOP_n,         DIR = I, BUS = LLINK0
PORT tx_eop_n = LL_Tx_EOP_n,         DIR = I, BUS = LLINK0
PORT tx_src_rdy_n = LL_Tx_SrcRdy_n,DIR = I, BUS = LLINK0
PORT tx_dst_rdy_n = LL_Tx_DstRdy_n,DIR = O, BUS = LLINK0

PORT rx_data = LL_Rx_Data,           DIR = O, BUS = LLINK0, VEC = [0:31]
PORT rx_rem = LL_Rx_Rem,             DIR = O, BUS = LLINK0, VEC = [0:3]
PORT rx_sof_n = LL_Rx_SOF_n,         DIR = O, BUS = LLINK0
PORT rx_eof_n = LL_Rx_EOF_n,         DIR = O, BUS = LLINK0
PORT rx_sop_n = LL_Rx_SOP_n,         DIR = O, BUS = LLINK0
PORT rx_eop_n = LL_Rx_EOP_n,         DIR = O, BUS = LLINK0
PORT rx_src_rdy_n = LL_Rx_SrcRdy_n,DIR = O, BUS = LLINK0
PORT rx_dst_rdy_n = LL_Rx_DstRdy_n,DIR = I, BUS = LLINK0
```

Una vez definida la interfaz LocalLink del periférico, este debe conectarse a un bloque DMA del bloque de procesador. Para ello, primero hay que configurar el PowerPC para activar un segundo DMA, ya que en el sistema base, viene por defecto un único DMA, el cual está conectado al TEMAC.

Para hacerlo, basta con abrir la configuración del PowerPC, y modificar el parámetro de número de DMAs. Activado el segundo DMA, queda realizar las conexiones de todos los puertos del periférico. Las conexiones, pueden realizarse desde la interfaz gráfica de la herramienta, o directamente en el fichero de especificaciones hardware del microprocesador (*Microprocessor Hardware Specification – MHS*) (figura 74). Puesto que ambas interfaces están conectadas a un mismo nombre, en este caso *bloque_ip_0_LLINK0*, significará que están conectadas entre sí.




```
BEGIN ppc440_virtex5
…
BUS_INTERFACE LLDMA1 = bloque_ip_0_LLINK0
…
END

BEGIN bloque_ip
…
BUS_INTERFACE LLINK1 = bloque_ip_0_LLINK0
…
END
```

Figura 74. Activación del segundo DMA del PowerPC y conexión a la interfaz LocalLink.

Además de la conexión del bus LocalLink, es necesaria la conexión de las señales de reloj y reset (el reset en el caso del DMA es interno al bloque de procesador, por lo que no es necesario conectar nada).

Para la señal de reset, se hará uso de una señal generada por el bloque destinado para este fin, es decir, el módulo de reset del sistema. Uno de sus puertos de salida se corresponde al reset para los periféricos. La conexión queda representada en el fichero MHS:

```
BEGIN bloque_ip
 …
 PORT Ll_Rst = sys_periph_reset
…
END
```

Para la conexión del reloj, se deberá generar una nueva fuente de reloj en el *clock_generator_0*, y conectarlo tanto a la entrada de reloj del DMA en cuestión y del periférico. Al igual que ocurre con las conexiones anterior, pueden realizarse desde la interfaz gráfica o directamente en el fichero MHS que debe quedar como a continuación.

```
BEGIN ppc440_virtex5
…
PORT CPMDMA1LLCLK = clk_ll_dma_100MHz
…
END

BEGIN clock_generator
…
PARAMETER C_CLKOUT5_FREQ = 100000000
PORT CLKOUT5 = clk_ll_dma_100MHz
…
END

BEGIN bloque_ip
…
PORT Ll_Clk = clk_ll_dma_100MHz
…
END
```

En este punto, el periférico está completamente integrado en la plataforma, y mediante el conjunto de funciones de comunicación con el DMA, pueden realizarse transferencias de datos desde una aplicación que se ejecute en el PowerPC 440.

5.3 Configuración del bloque TEMAC

Por defecto, durante la creación de una plataforma con un bloque TEMAC, sobre la placa de prototipado ML510 de Xilinx, el asistente BSB configurará por defecto la interfaz entre el TEMAC y el chip PHY externo como MII (*Media Independent Interface*) [58].

Esta interfaz permite transferencias a través de los protocolos *Ethernet* (10 Mbps) y *Fast Ethernet* (100 Mbps). Sin embargo, para la aplicación que se plantea en este proyecto, se tiene como requerimiento una alta tasa de eventos, y por lo tanto la interfaz de entrada de estos datos debe maximizarse, por lo que se hace necesario configurar el sistema para hacer uso de un protocolo *Gigabit Ethernet* (1000 Mbps).

El primer paso a realizar es modificar el tipo de interfaz en la configuración del TEMAC, que como todos los parámetros, puede realizarse a través de la interfaz gráfica o modificando el fichero MHS, para imponer un valor de *RGMII v2.0* [59].

```
BEGIN xps_ll_temac
…
PARAMETER C_PHY_TYPE = 3
…
END
```

La causa para no usar GMII (evolución directa de MII para dar soporte a Gigabit Ethernet) es que la placa ML510 no da soporte para esta interfaz, por motivos de conexiones de la PCB o porque el chip PHY no la soporte. Al realizar esta modificación, los puertos activos del bloque TEMAC cambian, ya que no coinciden con los de una interfaz MII. Los nuevos puertos deberán indicarse que son externos a la FPGA. Esto se muestra a continuación.

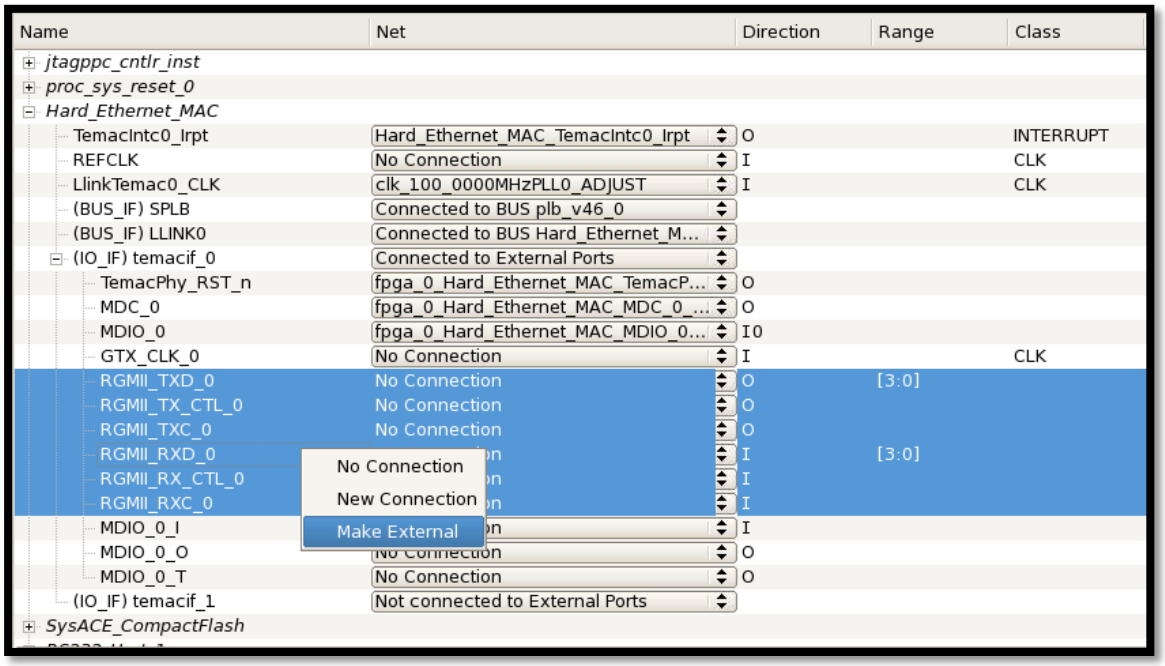


Figura 75. Asignación de señales externas del bloque TEMAC.

Otra modificación necesaria es que la fuente de reloj del TEMAC debe ser de 125 MHz. Para ello es necesario crear una nueva fuente de reloj de 125 MHz, que se conectará a la señal *GTX_CLK_0* del bloque TEMAC.

```
BEGIN xps_ll_temac
…
PORT GTX_CLK_0 = clk_RGMII_125MHz
…
END

BEGIN clock_generator
…
PARAMETER C_CLKOUT6_FREQ = 125000000
PORT CLKOUT6 = clk_RGMII_125MHz
…
END
```

La siguiente modificación a realizar se encuentra en el fichero de restricciones de usuario (*User Constraints File – UCF*). Este fichero contiene aquellas restricciones tanto de pines de entrada salida, como de latencia y posición en la FPGA para un TEMAC configurado con una interfaz MII. Habrá que comentar las líneas actuales asociadas a señales ya no existentes, para añadir las restricciones adicionales..

```
#NET "*Hard_Ethernet_MAC/TxClientClk_0"     TNM_NET = "clk_client_tx0";
```

```
#TIMEGRP  "mii_client_clk_tx0"    = "clk_client_tx0";
#TIMESPEC "TS_mii_client_clk_tx0" = PERIOD "mii_client_clk_tx0" 7500 ps
HIGH 50 %;
#NET "*Hard_Ethernet_MAC/RxClientClk_0"    TNM_NET = "clk_client_rx0";
#TIMEGRP  "mii_client_clk_rx0"    = "clk_client_rx0";
#TIMESPEC "TS_gmii_client_clk_rx0" = PERIOD "gmii_client_clk_rx0" 7500 ps
HIGH 50 %;
#NET "*Hard_Ethernet_MAC/MII_RX_CLK_0*"    TNM_NET = "phy_clk_rx0";
#TIMEGRP  "mii_clk_phy_rx0"    = "phy_clk_rx0";
#TIMESPEC "TS_mii_clk_phy_rx0 = PERIOD "mii_clk_phy_rx0" 40000 ps HIGH
50 %;
#NET "*Hard_Ethernet_MAC/MII_TX_CLK_0*" TNM_NET = "clk_mii_tx_clk0";
#TIMESPEC "TS_mii_tx_clk0" = PERIOD "clk_mii_tx_clk0" 40000 ps HIGH 50 %;
#INST "*mii0*RXD_TO_MAC*"     IOB = TRUE;
#INST "*mii0*RX_DV_TO_MAC"    IOB = TRUE;
#INST "*mii0*RX_ER_TO_MAC"    IOB = TRUE;
#NET "fpga_0_Hard_Ethernet_MAC_MII_RXD_0_pin(?)"  TNM = "mii_rx_0";
#NET "fpga_0_Hard_Ethernet_MAC_MII_RX_DV_0_pin"   TNM = "mii_rx_0";
#NET "fpga_0_Hard_Ethernet_MAC_MII_RX_ER_0_pin"   TNM = "mii_rx_0";
#TIMEGRP "mii_rx_0" OFFSET = IN 10 ns VALID 20 ns BEFORE
"fpga_0_Hard_Ethernet_MAC_MII_RX_CLK_0_pin";
#INST "*mii0*MII_TXD_?"       IOB = TRUE;
#INST "*mii0*MII_TX_EN"       IOB = TRUE;
#INST "*mii0*MII_TX_ER"       IOB = TRUE;

INST "*rgmii_rx_ctl_delay" IDELAY_TYPE = FIXED;
INST "*rgmii_rx_d*"        IDELAY_TYPE = FIXED;
INST "*rgmii_rxc0_delay"   IDELAY_TYPE = FIXED;

INST "*rgmii_rx_ctl_delay" IDELAY_VALUE = 26;
INST "*rgmii_rx_d*"        IDELAY_VALUE = 26;
INST "*rgmii_rxc0_delay"   IDELAY_VALUE = 0;
INST "*rgmii_rxc0_delay"   SIGNAL_PATTERN = CLOCK;
```

Por último, será necesario configurar los *jumpers* J50, J28 y J49 de la placa tal y como se muestra en la figura 76.

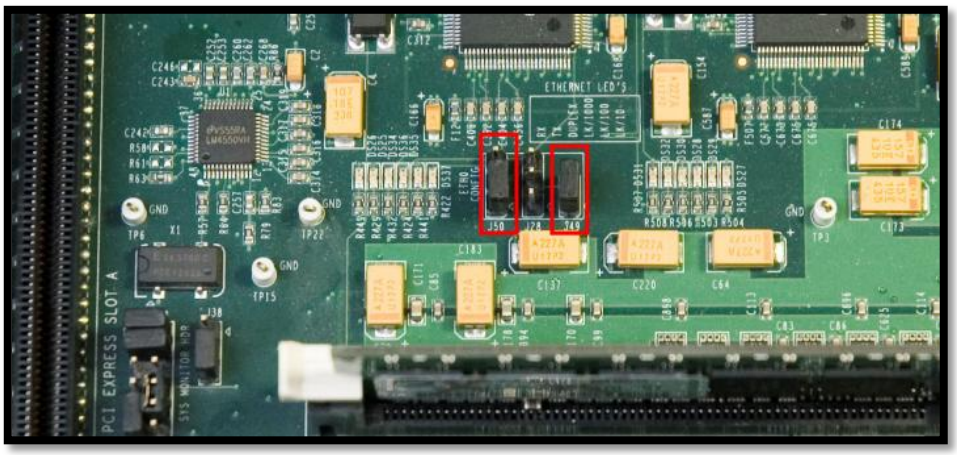


Figura 76. Jumpers de la placa ML510 para usar RGMII.

6 Creación de la plataforma *software*

Una vez definida la plataforma *hardware*, es necesario definir la plataforma *software*, es decir, el conjunto de librerías, *drivers*, y sistemas operativos, disponibles para el diseño de la aplicación a ejecutar en el microprocesador del sistema.

Al igual que ocurría con los bloques IP *hardware*, Xilinx proporciona diferentes recursos *software* sobre los que implementar la aplicación de usuario.

La librería de soporte estándar de C consta de la librería *libc*, que contiene las funciones estándar de las librerías *stdio*, *stdlib* y *string*. La librería matemática *libm* provee las rutinas estándar de operaciones aritméticas.

Se proporcionan además dos posibles sistemas operativos: la plataforma *Standalone* y *Xilkernel*.

Existe además una capa de abstracción de *hardware* (*Hardware Abstraction Layer* – HAL) que proporciona funciones comunes de entrada salida, excepciones y caché, para el Microblaze, el PowerPC 405 y el PowerPC 440. Esta capa tiene como función hacer portable el código desarrollado a distintas plataformas *hardware*, pudiendo modificarse el procesador y los periféricos manteniendo una gran parte del código sin alteraciones (figura 77).

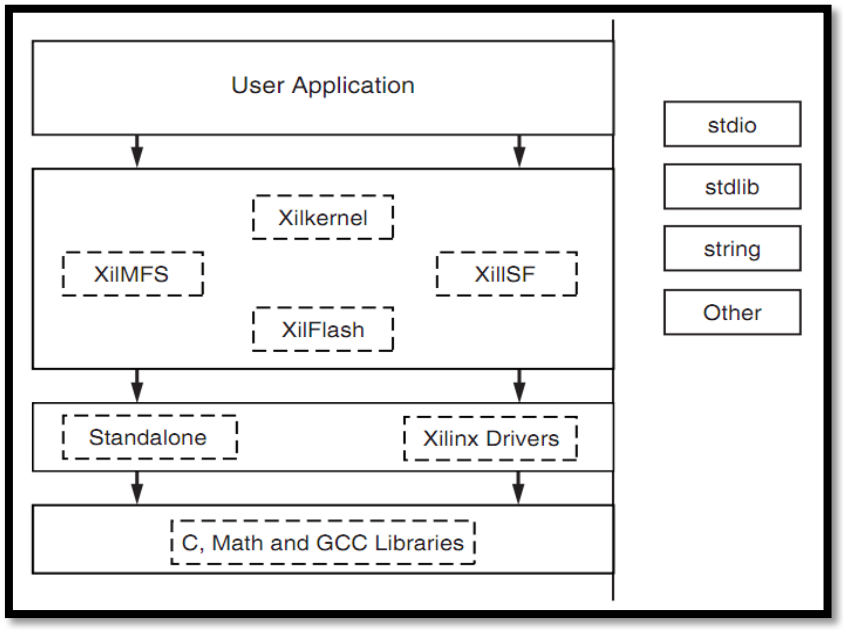


Figura 77. Estructura de capas del software del sistema.

Como se observa, una aplicación puede comunicarse con el *hardware* de varias maneras. En general, las librerías son independientes salvo algunas excepciones. Por ejemplo, *Xilkernel* usa la

plataforma *Standalone* internamente. La librería *LibXil* de *drivers* y la plataforma *Standalone* forman la capa más baja de abstracción *hardware* [60].

6.1 Librería lightweight IP (lwIP)

La librería lwIP proporciona a un sistema empotrado de capacidad la capacidad de comunicación TCP/IP mediante un conjunto de funciones que facilitan la labor de diseño del *software*. Esta librería sirve de adaptador para los bloques IP *hardware xps_ethernetlite* y *xps_ll_temac*, ambos son bloques *hardware* de Xilinx de tipo Ethernet MAC, y para ser ejecutado tanto en un Microblaze, en un PowerPC 405 como un PowerPC 440. Tiene dos modos de operación, cada cual tiene un conjunto de funciones asociadas.

- RAW API: Da acceso a la pila TCP/IP mediante *callbacks*.
- Socket API: Da acceso a la pila TCP/IP mediante el estilo de sockets BSD.

La librería da soporte a diferentes protocolos asociados a las redes TCP/IP, en particular a los siguientes:

- *Internet Protocol* (IP)
- *Internet Control Message Protocol* (ICMP)
- *User Datagram Protocol* (UDP)
- *Transmission Control Protocol* (TCP)
- *Address Resolution Protocol* (ARP)
- *Dynamic Host Configuration Protocol* (DHCP)

6.1.1 Funciones

A continuación se presentan las interfaces de algunas de las funciones que proporciona Xilinx para facilitar el uso de la librería lwIP y que se usarán en el diseño de la aplicación empotrada para proporcionar comunicación a través de la red TCP/IP.

- **`void lwip_init().`** Esta función tiene como objetivo la inicialización para las estructuras de datos de lwIP. Esta reemplaza llamadas específicas de inicialización de estados, sistema, memoria, y las capas ARP, IP, UDP y TCP.
- **`struct netif *xemac_add (struct netif *netif, struct ip_addr *ipaddr, struct ip_addr *netmask, struct ip_addr *gw, unsigned char *mac_ethernet_address, unsigned mac_baseaddr).`** Inicializa una interfaz de red de entre los bloques EMAC de la plataforma *hardware*. Se trata de un *wrapper* de la función `netif_add()`. Como parámetros de entrada tiene el puntero a la estructura de datos de la interfaz, la dirección IP a asociar a la interfaz, la máscara de subred, la dirección IP de la puerta de enlace, la dirección MAC del puerto en cuestión, así como la dirección base del periférico al que se asocie.
- **`void xemacif_input_thread (struct netif *netif).`** En el modo de operación Socket API, este hilo debe lanzarse en paralelo con la aplicación.

Cuando se recibe un paquete por la red, el EMAC los almacena en una cola. Este hilo tiene por objetivo leer los paquetes de esa cola y hacerlos disponibles para la librería.

- `void netif_set_default(struct netif *netif).` Hace que la interfaz seleccionada sea la interfaz por defecto.
- `void netif_set_up(struct netif *netif).` Activa la interfaz que se pasa como parámetro.

Además de estas funciones, lwIP da soporte a las funciones típicas de tratamiento de sockets BSD. Así, las funciones a usar para dicho objetivo son:

- **`lwip_socket()`**. Crea un nuevo socket de un tipo específico, identificado por un número entero, y asigna recursos del sistema para él.
- **`lwip_bind()`**. Se usa en el lado servidor, y asocia una dirección IP y puerto al socket al que se le realice.
- **`lwip_listen()`**. Se usa en el lado servidor y deja el socket en modo escucha para recibir peticiones de conexión.
- **`lwip_connect().`** Se usa en el lado cliente para realizar la conexión en caso de que sea TCP. Además asigna uno de los puertos libres.
- **`lwip_accept().`** Usado en el lado servidor para aceptar un intento de conexión, creando un nuevo socket que se asocia al cliente.
- **`lwip_send()`**. Sirve para enviar los datos a través de un socket.
- **`lwip_recv().`** Lee los datos de un socket.

6.2 Sistema Operativo Xilkernel

Xilkernel es un núcleo de sistema operativo integrado en el entorno de desarrollo de Xilinx. Soporta su personalización, pudiendo activar o desactivar funcionalidad para obtener un rendimiento óptimo tanto en memoria como en velocidad. Da soporte de todos los requerimientos de un núcleo empotrado, con una interfaz POSIX, y está preparado para ejecutarse, entre otros núcleos, en el PowerPC 440.

Las principales características que añade sobre la plataforma *Standalone* son:

- Hilos paralelos de ejecución, con planificación *Round-Robin* o estática.
- Sincronización de hilos: semáforos y exclusión mutua.
- Servicios de comunicación entre procesos: paso de mensajes y memoria compartida.
- Relojes *software*.

La aplicación de usuario se ejecuta, en una estructura de capas, sobre la capa del núcleo, usando parte, o todos los módulos de este.

Xilkernel está organizado en forma de librería de funciones del núcleo. Esto permite una etapa sencilla de enlazado durante la compilación de la aplicación. Para ello, basta con añadir Xilkernel a la plataforma *software*, configurarla y generarla, para luego poder hacer uso de sus funciones.

Una aplicación que use este núcleo debe comenzar su ejecución con una función **`main()`**, que realice la inicialización de los *drivers* necesarios. En ese punto deben activarse las excepciones *hardware*, necesarias para la correcta ejecución del Xilkernel. El siguiente paso es la realizaciónde una llamada a la función de inicialización del núcleo, **`xilkernel_main()`**. Esta función inicia el *kernel*, activa las interrupciones y devuelve el control a la aplicación de usuario mediante una tabla estática de hilos de ejecución. La figura 78 muestra un diagrama de capas donde se representa la jerarquía de una aplicación que haga uso del núcleo Xilkernel.

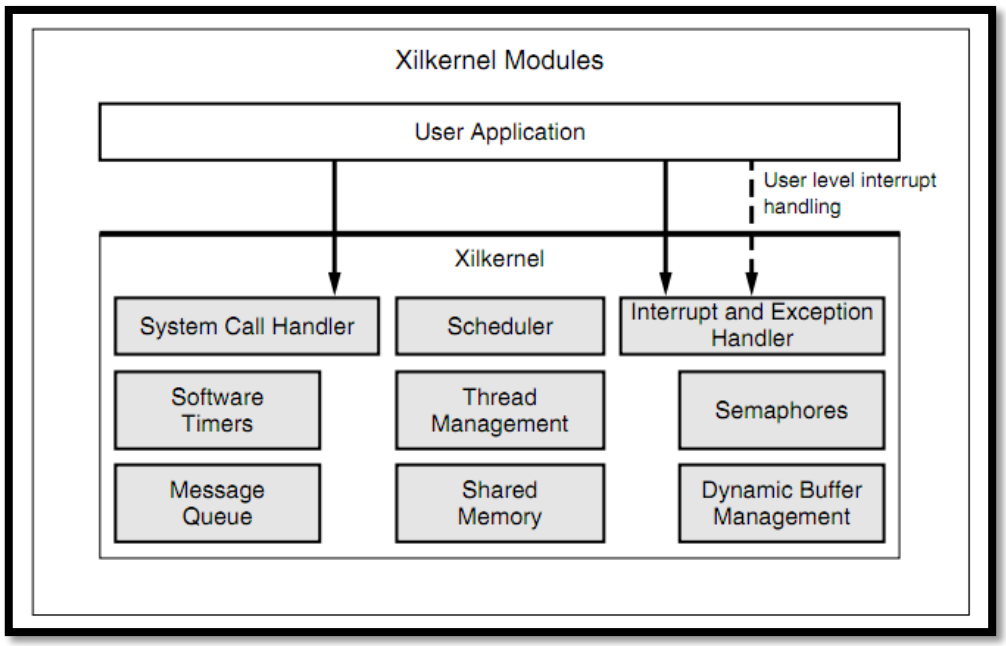


Figura 78. Diagrama de capas de Xilkernel.

6.3 Configuración de la plataforma *software*

Al igual que ocurría con la plataforma *hardware*, existe un fichero en el que queda descrita con totalidad la configuración de la plataforma, con el conjunto de librerías que se incluyen así como sus parámetros (*Microprocessor Software Specification* – MSS).

La primera librería a añadir es la lwIP, ya que es la que proporciona las funciones de interfaz con los Ethernet MACs, necesarios para recibir eventos de la red y enviar los resultados. Además hay que configurar el modo de operación que por defecto es RAW API. Por motivos de portabilidad del diseño se ha optado por usar el modo Socket API, que usa una interfaz de sockets BSD, haciendo que el *software* pueda ser fácilmente compilable en otras plataformas que usen sockets.

El usar el modo Socket API implica el uso de varios hilos de ejecución paralelos, puesto que uno es explícitamente para la lectura de paquetes de las colas del Ethernet MAC. Por ello, la elección del sistema operativo queda reducida a la configuración del núcleo Xilkernel, ya que la plataforma *Standalone* no da soporte a aplicaciones multi-hilo.

En el apartado de configuración de la plataforma software, habrá que indicar tanto el uso de Xilkernel como el de la librería lwIP. Y a su vez, indicar el modo de operación de la librería, y activar aquellas características del núcleo necesarias para que la librería pueda usarse. Estas características a activar son:

- Soporte para hilos paralelos.
- Planificador del núcleo.
- Soporte para temporizador para el núcleo.
- Soporte para semáforos.
- Incluir la función `kill()`, para matar procesos dinámicamente.
- Incluir la función `yield()`, para ceder el paso al siguiente proceso.

Además, deben estar configuradas las entradas y salidas estándar (stdin y stdout) para ser vinculadas con la UART del sistema, de forma que las funciones que escriban o lean de ellas (`printf()`, `scanf()`, etc.) envíen sus datos al transceptor serie y esto pueda ser recogido en otra máquina.

Además, se deben especificar ciertas opciones para el compilador (en particular para el *linker*), para que enlace las librerías. Si, como es el caso, además existe parte del código en C++, debe especificarse las siguientes opciones:

```
lwip4 xilkernel stdc++
```

Con esto, ya puede comenzarse con el diseño de la aplicación que se ejecuta en el microprocesador de la plataforma.

7 Aplicación empotrada

A la hora de usar las librerías, tanto Xilkernel como lwIP, Xilinx da unas directrices que especifican la estructura base del programa. Este esquema describe el orden de configuración de los diferentes módulos que conformen la plataforma, tanto *software* como *hardware*.

Para el uso del núcleo del sistema Xilkernel, por ejemplo, la secuencia de ejecución debe ser la siguiente [61]:

1. Iniciar el sistema de excepciones, en el que se debe indicar la rutina de servicio de las interrupciones IRQ, que corresponderá a la del controlador externo de interrupciones.
2. Inicializar Xilkernel, mediante la llamada a `xilkernel_main()`.
3. Configuración e inicialización de las estructuras de datos de Xilkernel, llevada a cabo por la función anterior.
4. Una vez finalizada la configuración, el Xilkernel lanzará en paralelo los hilos de ejecución de la tabla estática.

En dicha tabla estática estará, si fuese único, el hilo principal de ejecución de la aplicación, además de su prioridad si se quisiera especificar. La ejecución de este esquema se representa en la figura 79.

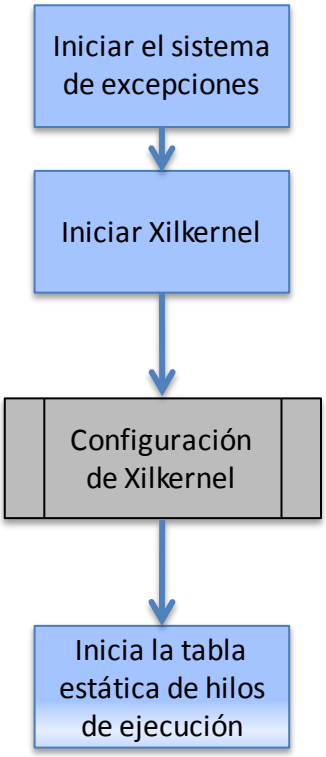


Figura 79. Esquema de aplicación con Xilkernel.

Por otro lado, para las aplicaciones que usen la librería lwIP, también se especifica la jerarquía de configuración. La primera es la inicialización de las estructuras de esta, para luego lanzar un hilo en el que se configure la interfaz Ethernet del sistema, tras lo cual deben lanzarse dos hilos. Uno será el que se lance en paralelo durante toda la ejecución y que tiene por objetivo mover los paquetes de la cola de recepción del TEMAC y hacerlos disponible a la librería de funciones de lwIP, y el segundo hilo será la aplicación que haga uso de dichas funciones [62].

El hilo de ejecución de la aplicación coincide con el de la configuración de la interfaz de red, pero ejecutado secuencialmente después de este, en lugar de hacer morir el hilo inicial para lanzar otro al final de este, como sí ocurre en el de configuración de lwIP (figura 80).

En la figura 80 se muestra de forma simplificada el flujo de ejecución de esta primera parte del *software* empotrado. En ambos esquemas, se ha representado como una nube las tareas de configuración de las librerías, aunque son accesibles por el diseñador para su estudio e incluso modificación.

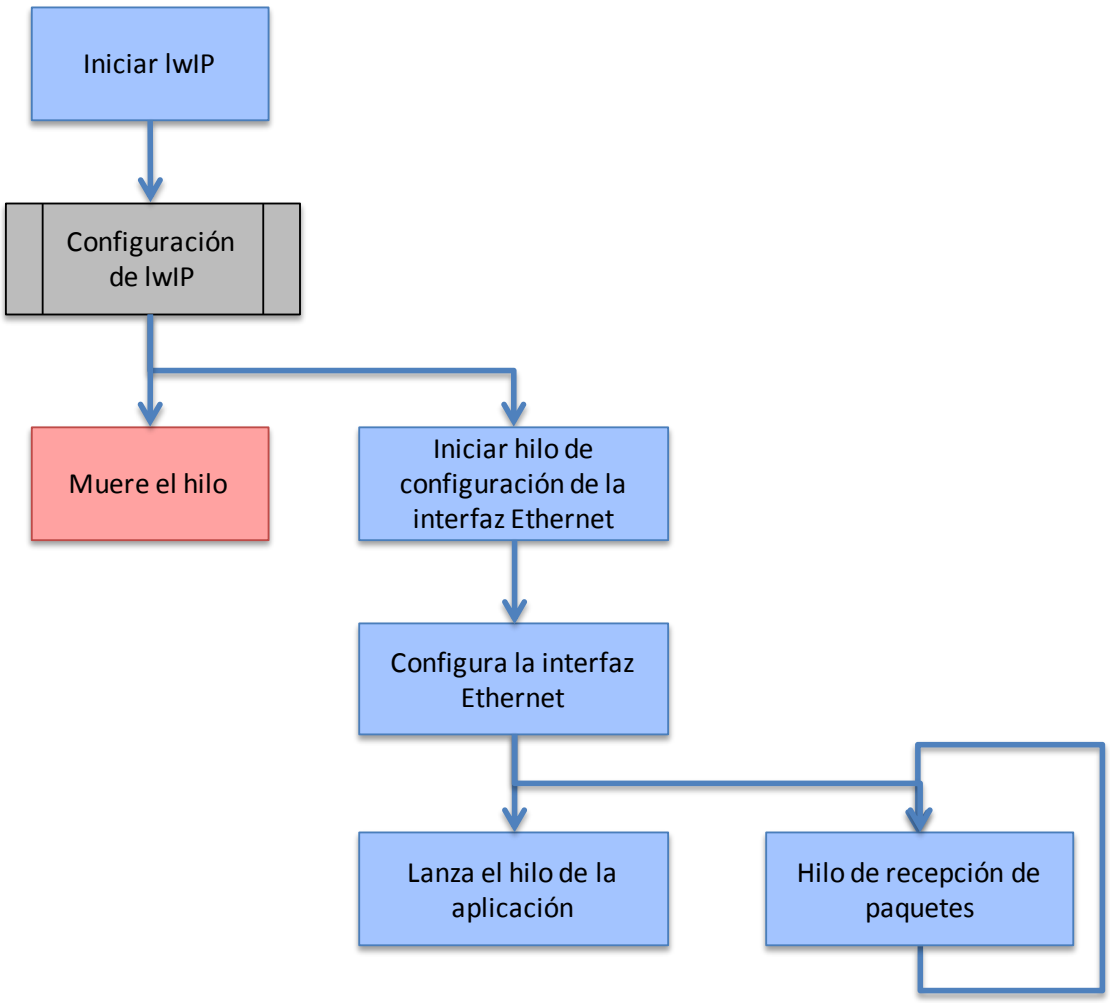


Figura 80. Esquema de aplicación con lwIP en modo Socket API.

A partir de este punto, el código que se ejecute a continuación puede usar todas las funciones activas de la librería Xilkernel (lanzar hilos, usar semáforos, etc.), así como de las funciones de lwIP (crear sockets, conectarlos, aceptar conexiones, etc).

En particular, la secuencia que deberá la aplicación principal consiste (figura 81):

1. Crear e inicializar los anillos de descriptores para las transacciones a través del bloque DMA (son las estructuras de control usadas para dichas transacciones).
2. Indicar las rutinas de servicio, para las interrupciones generadas por el DMA cuando se realice una transmisión con éxito así como para las recepciones.
3. Crear un socket y ponerlo a la escucha de una petición de conexión, para tras aceptarla comenzar con las tareas cíclicas de la aplicación.
4. Existirán a partir de aquí dos hilos de ejecución, uno por cada sentido de la comunicación.
    a. El primero, será el encargado de leer las tramas recibidas a través de la pila TCP/IP, preprocesarlas, y asignarle un descriptor para enviarle dichos datos al periférico a través del DMA.

b. El segundo hilo es el encargado de que, cuando se produzca una interrupción de trama recibida del periférico, realice un procesado y enviarla a través de la red a su destino.

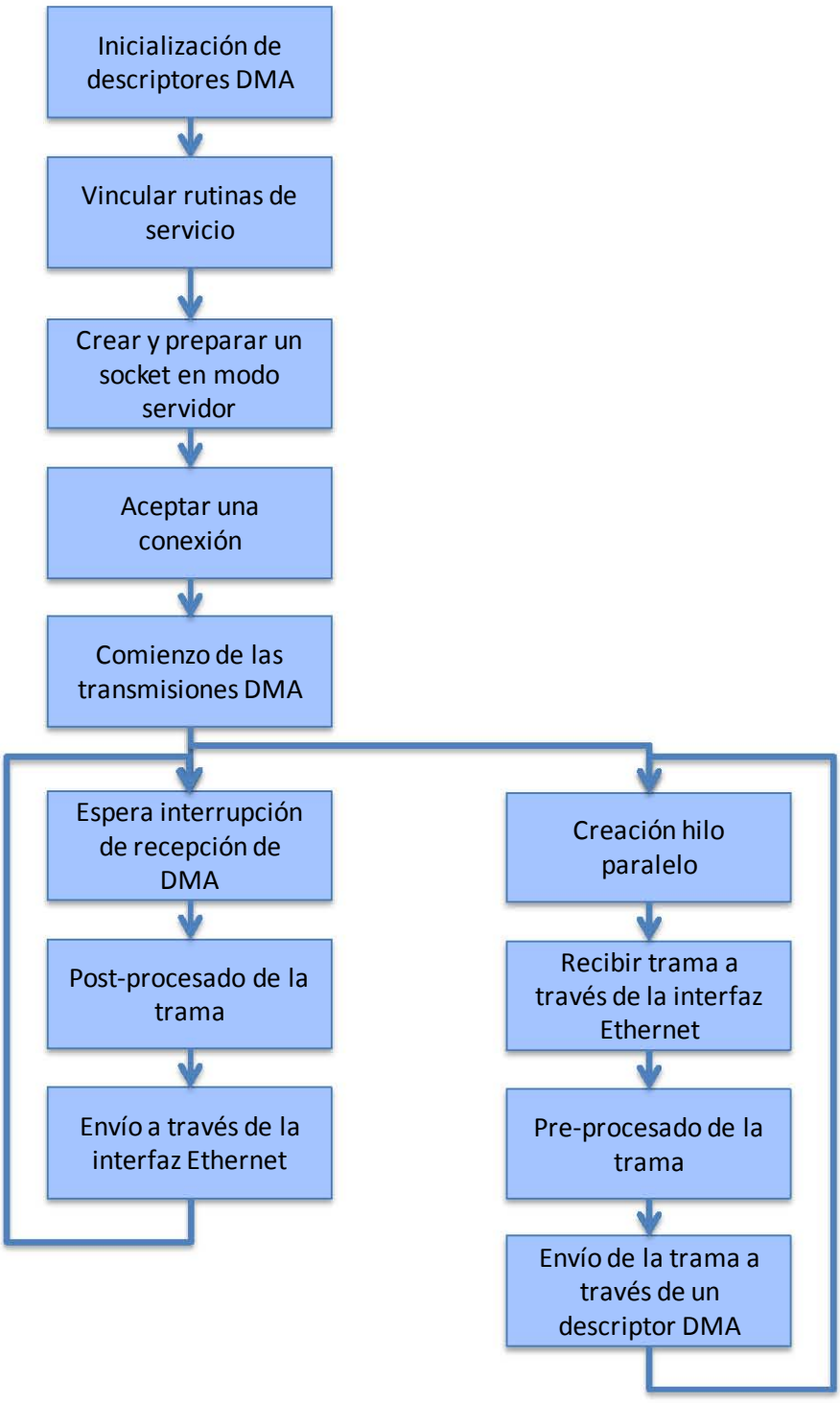


Figura 81. Flujo de ejecución de la aplicación.

7.1 Inicialización del sistema de excepciones

A continuación se detallarán las funciones de librería usadas para la inicialización del sistema de excepciones, así como una breve descripción del objetivo de cada una (figura 79).

```
#define INTC_DEVICE_ID              XPAR_INTC_0_DEVICE_ID
XIntc               Intc;

void SetupExceptionSystem(void) {
  XExc_Init();
  XExc_RegisterHandler(XEXC_ID_NON_CRITICAL_INT,
        XExceptionHandler)XIntc_InterruptHandler, &Intc);
  XExc_mEnableExceptions(XEXC_NON_CRITICAL);
  XIntc_Initialize(&Intc, INTC_DEVICE_ID);
  XIntc_Start(&Intc, XIN_REAL_MODE);
}
```

En general, todas las funciones de librería vendrán precedidas por un prefijo que indicarán la librería a la que pertenece (en el ejemplo anterior, XExc para la librería xexception_l.h y XIntc para xintc.h).

- **`XExc_Init().`** Inicializa los vectores de interrupción y asocia una rutina de servicio inerte a cada uno de los identificadores de interrupción.
- **`XExc_RegisterHandler().`** Asocia una rutina de servicio a un identificador de interrupción. En este caso en particular, se asocia una rutina de servicio a las interrupciones externas (no críticas) y se pasa además un tercer parámetro, el cual es un argumento de entrada para la rutina de servicio. Esta rutina, es única para todas las interrupciones externas, y su función es lanzar la rutina específica del dispositivo que haya generado la interrupción.
- **`XExc_mEnableExceptions().`** Permite las excepciones del tipo que se pase como parámetro.
- **`XIntc_Initialize().`** Inicializa la estructura de datos del controlador de interrupciones y la asocia al dispositivo físico de la plataforma *hardware*. Como parámetros de entrada tiene un puntero a la estructura de datos y el identificador único del controlador al que se quiere asociar. La salida queda desactivada al finalizar esta función.
- **`XIntc_Start().`** Activa la salida de interrupción del controlador de interrupciones. A partir de este punto, el controlador de interrupciones generará a su salida la señal IRQ cuando alguno de los periféricos conectados a su entrada así lo requieran.

7.2 Inicialización de Xilkernel

Para hacer uso del núcleo, es el primer paso a ejecutar tras activar las excepciones del microprocesador. Se realiza mediante una simple llamada (figura 79).

```
xilkernel_main();
```

Esta función se divide en dos partes. La primera inicializa el sistema, mediante la inicialización de la tabla de vectores de procesos, la creación del proceso base (PID = 0) y la creación de los procesos de la tabla estática. El segundo paso es la de comenzar la ejecución, mediante la llamada a la tarea base ubicada en la tabla estática de hilos.

7.3 Inicialización de lwIP

Al igual que ocurre con Xilkernel, la librería lwIP requiere de ciertas estructuras de datos que deben inicializarse antes de poder usar sus funciones (figura 80).

```
lwip_init();
```

La función inicializa los diferentes módulos que conforman la librería, en función de aquellos servicios que hayan sido, o no, activados (DHCP, ARP, TCP, IGMP, etc).

7.4 Configuración de la interfaz Ethernet

Antes de poder crear sockets, es necesario definir las interfaces de red de las que dispone el diseño. A continuación se muestra el código que defina una interfaz de red, la marca como predeterminada y la activa (figura 80).

```
void SetupNetIf (void *p) {
  struct netif *netif, server_netif;
  struct ip_addr ipaddr, netmask, gw;
  unsigned char mac_ethernet_address[] = { 0x00, 0x0a, 0x35, 0x01, 0xd8,
                                           0xd8 };

  netif = &server_netif;

  IP4_ADDR(&ipaddr,  192, 168,   0, 10);
  IP4_ADDR(&netmask, 255, 255, 255,  0);
  IP4_ADDR(&gw,      192, 168,   0,  1);

  xemac_add(netif,   &ipaddr,   &netmask,   &gw,   mac_ethernet_address,
            XPAR_LLTEMAC_0_BASEADDR)
  netif_set_default(netif);
  netif_set_up(netif);
  …
}
```

Estas funciones ya fueron explicadas anteriormente. Queda comentar, que como ocurría en el caso del controlador de interrupciones, en este caso también existe una estructura de datos vinculada al bloque *hardware* (netif).

7.5 Hilo de ejecución de recepción de paquetes

Como ya se adelantó anteriormente, es necesario, en el modo de operación Socket API, que se ejecute un hilo de ejecución paralelo que mueva los paquetes desde la cola de recepción a la librería (figura 80).

```
sys_thread_new("xemacif_input_thread",
                (void(*)(void*))xemacif_input_thread,
                netif,
                THREAD_STACKSIZE,
                DEFAULT_THREAD_PRIO);
```

Se debe pasar como parámetros de entrada, la función a ejecutar en el nuevo hilo, un puntero a datos de entrada, el tamaño de la pila y la prioridad estática del hilo, que servirá para la planificación de hilos del núcleo.

7.6 Inicialización del anillo de descriptores de DMA

Para realizar transacciones a través de un DMA, se hacen necesarias unas estructuras de datos que almacenan información de control sobre las tramas. Estas estructuras están unidas formando un anillo, que debe crearse e inicializarse (figura 81).

```
void SetupBDRing() {
  XLlDma_Initialize(&Dma, 0x98);  // Address DMA1


  DirFreeRx_na = malloc(XLlDma_BdRingMemCalc(BD_ALIGNMENT, RXBDCNT+1));
  DirFreeRx = (void*)((Xuint32)DirFreeRx_na + (BD_ALIGNMENT -
                (Xuint32)DirFreeRx_na % BD_ALIGNMENT));
  XLlDma_BdRingCreate(RxRingPtr, (Xuint32)DirFreeRx, (Xuint32)DirFreeRx,
                BD_ALIGNMENT, 1);

  DirFreeTx_na = malloc(XLlDma_BdRingMemCalc(BD_ALIGNMENT, TXBDCNT+1));
  DirFreeTx = (void*)((Xuint32)DirFreeTx_na + (BD_ALIGNMENT -
                (Xuint32)DirFreeTx_na % BD_ALIGNMENT));
  XLlDma_BdRingCreate(TxRingPtr, (Xuint32)DirFreeTx, (Xuint32)DirFreeTx,
                BD_ALIGNMENT, 1);
}
```

El primer paso consiste en enlazar la estructura de datos del DMA al bloque físico a través de su dirección única.

A continuación se generará el anillo de descriptores para cada interfaz LocalLink, es decir, un anillo de transmisión y otro de recepción. Una restricción a cumplir es que la dirección de comienzo debe estar alineada al tamaño de un descriptor, es decir, que los últimos cinco bits estén a cero. Al hacer una petición de memoria dinámica mediante la función `malloc()` no se puede asegurar que este tenga tal alineación, así que una forma de solucionarlo, es realizar la petición de memoria para el número de descriptores necesarios y uno más, que será el que permite la holgura para desplazar los descriptores. En la figura 82 se muestra como crear un anillo de tres descriptores, donde **DirFree_na** apunta al espacio de memoria del tamaño de cuatro

descriptores, y **DirFree** es un puntero que apunta al comienzo de la primera dirección alineada de dicho espacio. De esta forma se consigue el objetivo propuesto, a costa de desperdiciar el espacio de un descriptor, partido en dos subespacios de memoria, antes y después del espacio utilizado.

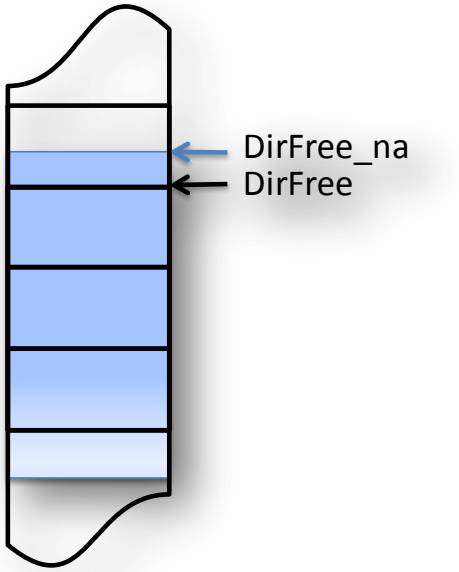


Figura 82. Alineamiento de los descriptores DMA.

7.7 Vinculación de las rutinas de servicio

Una vez vinculada la rutina de servicio del controlador de interrupciones al vector de excepciones del microprocesador, es necesario indicarle al controlador de interrupciones la rutina de servicio de cada periférico conectado a él (figura 81).

```
#define HWIP_RX_INTR_ID    XPAR_XPS_INTC_0_PPC440_0_DMA1RXIRQ_INTR
#define HWIP_TX_INTR_ID    XPAR_XPS_INTC_0_PPC440_0_DMA1TXIRQ_INTR

int SetupIntrHandlers()
{
  extern XIntc       Intc;
  extern XLlDma      Dma;

  XIntc_Connect(&Intc, HWIP_RX_INTR_ID, (XInterruptHandler) RxHandler,
                &Dma);
  XIntc_Connect(&Intc, HWIP_TX_INTR_ID, (XInterruptHandler) TxHandler,
                &Dma);

  XIntc_Enable(&Intc, HWIP_RX_INTR_ID);
  XIntc_Enable(&Intc, HWIP_TX_INTR_ID);

  return(XST_SUCCESS);
}
```

A través de la función `XIntc_Connect()` se enlaza un identificador de interrupción, con la rutina de servicio a ejecutar. En particular, en el caso del código anterior, a la interrupción de recepción de trama a través del DMA se le vincula la función `RxHandler()`, mientras que a la transmisión se vincula la función `TxHandler()`.

Es necesario definir qué hacer en cada rutina de servicio. Principalmente, en ambas deben leerse los descriptores, y posteriormente liberarlos para poder volver a hacer uso de ellos. Además, es importante siempre deshabilitar las interrupciones al comienzo de la rutina, ya que si no, se corre el riesgo de recibir una nueva interrupción entre el momento en el que se lee el estado del controlador y el momento en el que se envía la señal de acuse de recibo de este, perdiéndose tramas. Además, en el caso de la recepción, debe volverse a preparar un descriptor para recibir tramas.

```
void RxHandler(void *Callback) {
  u32                          IrqStatus;
  unsigned                     ReceivedBdCount;
  Xuint32                      R_FramePtr;

  XLlDma_Bd                    *BdPtr, *BdCurPtr;
  XLlDma_BdRing                RxRing, , *RxRingPtr;

  RxRingPtr = &XLlDma_GetRxRing(&Dma);
  XLlDma_BdRingIntDisable(RxRingPtr, XLLDMA_CR_IRQ_ALL_EN_MASK);

  IrqStatus = XLlDma_BdRingGetIrq(RxRingPtr);
  XLlDma_BdRingAckIrq(RxRingPtr, IrqStatus);

  ReceivedBdCount = XLlDma_BdRingFromHw(RxRingPtr, XLLDMA_ALL_BDS,
                                        &BdPtr);
  BdCurPtr = BdPtr;
  R_FramePtr = XLlDma_BdRead((BdCurPtr), XLLDMA_BD_BUFA_OFFSET);

  mensaje = (char *)R_FramePtr;
  memcpy(mensaje_salida, mensaje, longitud);
  mensaje_pendiente = 1;

  XLlDma_BdRingFree(RxRingPtr, ReceivedBdCount, BdPtr);
  XLlDma_BdRingAlloc(RxRingPtr, ReceivedBdCount, &BdPtr);

  BdCurPtr = BdPtr;
  XLlDma_BdSetStsCtrl(BdCurPtr, (XLLDMA_BD_STSCTRL_SOP_MASK |
                          XLLDMA_BD_STSCTRL_EOP_MASK));
  XLlDma_BdSetBufAddr(BdCurPtr, mensaje);
  XLlDma_BdSetLength(BdCurPtr, longitud);

  XLlDma_BdRingToHw(RxRingPtr, 1, BdCurPtr);
}
```

Como puede observarse, el primer paso de la rutina de servicio de interrupción consiste en obtener el anillo de descriptor mediante la función **`XLlDma_GetRxRing()`** y la deshabilitación de dicha interrupción, haciendo uso de la función **`XLlDma_BdRingIntDisable()`**.

El siguiente paso es recoger el valor del estado de la interrupción y enviar la señal de aceptación, mediante las funciones **`XLlDma_BdRingGetIrq()`** y **`XLlDma_BdRingAckIrq()`**.

A continuación, se realizará la lectura de los descriptores desde el bloque DMA, a través de la función **`XLlDma_BdRingFromHw()`**. Para acceder a los distintos campos de cada descriptor, se realizan llamadas a **`XLlDma_BdRead()`.**

Una vez obtenida la carga útil de la trama, esta se copia en un mensaje de salida, para su posterior post-procesado, y se activa el *flag* de dato pendiente de procesado.

Para finalizar, se deberá liberar los descriptores obtenidos (**`XLlDma_BdRingFree()`**), y, con el fin de poder seguir recibiendo tramas, se vuelve a enviar hacia el DMA un nuevo descriptor preparado para la recepción (**`XLlDma_BdRingToHw()`**).

El descriptor a enviar al DMA debe estar configurado con los bits de control, con la dirección del *buffer* donde almacenar los datos, su longitud y sus bits de comienzo y fin de la trama si fuese el caso (**`XLlDma_BdSetBufAddr()`**, **`XLlDma_BdSetLength()`** y **`XLlDma_BdSetStsCtrl()`**).

Las interrupciones de recepción no se habilitan al final de la rutina, sino que se rehabilitan una vez que los datos sean post-procesados, para que no se corrompan si una nueva trama es recibida antes de que el procesado termine.

```
void TxHandler(void *Callback) {
  unsigned       TransmitedBdCount;
  u32            IrqStatus;
  XStatus        Status;
  XLlDma_Bd      *BdPtr;
  XLlDma_BdRing  *TxRingPtr = &XLlDma_GetTxRing(&Dma);

  IrqStatus = XLlDma_BdRingGetIrq(TxRingPtr);
  XLlDma_BdRingAckIrq(TxRingPtr, IrqStatus);

  TransmitedBdCount = XLlDma_BdRingFromHw(TxRingPtr, XLLDMA_ALL_BDS,
                                          &BdPtr);
  XLlDma_BdRingFree(TxRingPtr, TransmitedBdCount, BdPtr);

  XLlDma_BdRingIntEnable(TxRingPtr, XLLDMA_CR_IRQ_ALL_EN_MASK);
}
```

La rutina de servicio para las interrupciones de transmisión es muy parecida a la de recepción, sin que se vuelva a enviar un descriptor al DMA (eso se debe hacer en el hilo de envío, tras recibir un dato de la red), y rehabilitando las interrupciones al final de esta.

7.8 Creación y preparación de un socket en modo servidor

Este paso difiere muy poco de la realización de un código equivalente sobre una máquina convencional. Como se adelantó en su correspondiente apartado, lwIP proporciona una API

basada en los sockets BSD, permitiendo portabilidad de código entre distintas plataformas (figura 81).

```
void SetupTcpRx () {
  extern int socket_cliente;
  int socket_recv;
  struct sockaddr_in address;
  socklen_t size;
  extern int puerto_escucha;

  socket_recv = lwip_socket(AF_INET, SOCK_STREAM, 0);

  address.sin_family = AF_INET;
  address.sin_port = htons(puerto_escucha);
  address.sin_addr.s_addr = INADDR_ANY;

  lwip_bind(socket_recv, (struct sockaddr *)&address, sizeof (address));

  lwip_listen(socket_recv, 1);
  size = sizeof(remote);

  socket_cliente = lwip_accept(socket_recv, (struct sockaddr*)&remote,
                               &size);
}
```

Tras la creación de un socket, este se asocia a una dirección IP mediante la función **`bind()`**. A continuación se pone a la escucha de una conexión mediante la función **`listen()`**, para luego crear el socket asociado a dicha conexión mediante la función **`accept()`**.

7.9 Comienzo de las transmisiones DMA

Para dar comienzo a las transmisiones DMA hacen falta dos pasos. El primero es habilitar las interrupciones de ambos sentidos, que por defecto se desactivan al inicio del programa. Además deben activarse los anillos de descriptores, que anteriormente han sido creados (figura 81).

```
  RxRingPtr     = &XLlDma_GetRxRing(&Dma);
  TxRingPtr     = &XLlDma_GetTxRing(&Dma);

  XLlDma_BdRingIntEnable(RxRingPtr,     XLLDMA_CR_IRQ_ALL_EN_MASK);
  XLlDma_BdRingIntEnable(TxRingPtr,     XLLDMA_CR_IRQ_ALL_EN_MASK);

  XLlDma_BdRingStart(TxRingPtr);
  XLlDma_BdRingStart(RxRingPtr);
```

7.10 Hilo de envío de eventos y estrategias

Uno de los hilos de ejecución tiene por objetivo recibir las tramas a través de la red, realizar un pre-procesado, si fuera necesario y enviar el resultado al periférico a través del DMA.

Para leer las tramas del TEMAC, se hace uso de la función típicas para socket, es decir la función **`recv()`**. En particular, en las tramas del modelo de referencia usado en este PFC, todas comienzan por una cabecera de seis bytes, los cuales indican el tipo de trama y su longitud. Es por

ello que, para la lectura de la trama, deben leerse primero esos primeros seis bytes, para conocer la longitud, y así leer el resto de la trama (figura 81).

```
bytes_ya_leidos = 0;


while (1) {
  bytes_ya_leidos += lwip_recv(socket_cliente,
                               (mensaje+bytes_ya_leidos),
                               (6-bytes_ya_leidos),
                                0);
  if (bytes_ya_leidos == 6) break;
}


p = (char *)&longitud_trama;
*p = mensaje[5]; p++;
*p = mensaje[4]; p++;
*p = mensaje[3]; p++;
*p = mensaje[2];

while (1) {
  bytes_ya_leidos += lwip_recv(socket_cliente,
                               (mensaje+bytes_ya_leidos),
                               (longitud_trama-bytes_ya_leidos),
                                0);
  if (bytes_ya_leidos == longitud_trama) break;
}

*longitud = longitud_trama;
```

La primera parte del código (en concreto, el primer bucle *while*) es el encargado de la lectura de la pila TCP/IP de la cabecera de la trama. El bucle finaliza cuando se hayan leído correctamente los seis bytes de cabecera. Esta cabecera tiene dos propósitos: conocer el tipo de trama recibida (evento o regla) y conocer la longitud total de la trama.

El segundo paso en la ejecución consiste en un *byte-swap*, debido a la diferencia de *Endianness* de los microprocesadores que generan las tramas y el PowerPC 440 integrado en la plataforma *hardware*. Los microprocesadores de la familia x86 son todos *Little-Endian*, es decir, que los datos de más de un byte (enteros de 16 bits, 32 bits o 64 bits; números en coma flotante de 32 o 64 bits, etc.) se almacenan con el byte menos significativo en la posición más baja de memoria. Sin embargo, el PowerPC 440 utiliza un bus de dato con formato *Big-Endian*, por lo que la posición más baja de la memoria almacena el byte más significativo. Para poder interpretar los datos, recibidos como una ristra de bytes, es necesario reordenarlos en memoria como se muestra en la figura 83.

Por último, se lee el resto de bytes en función de la longitud extraída de la cabecera. El procedimiento es equivalente al mostrado para la lectura de la cabecera.

El siguiente paso en este hilo consiste en realizar un pre-procesado de las tramas. Existen varias tareas a realizar en este paso.

- **Filtrado.** Es conveniente no enviar todos los eventos al procesador de eventos si no son necesarios. Pueden plantearse diferentes estrategias de filtrado. Por ejemplo, solo enviar los valores que hayan cambiado desde la última trama, enviar solo los valores asociados a las estrategias activas en el periférico, etc.
- ***Byte-Swap*.** Como ocurría con la longitud en la cabecera, los valores provenientes de la red estarán almacenados en formato *Little-Endian*, los cuales deberán modificarse a *Big-Endian* antes de enviarlos al periférico.

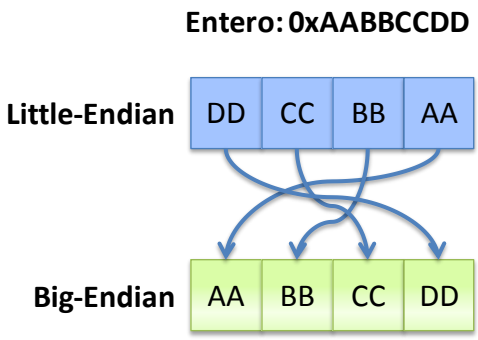


Figura 83. *Byte-Swap*

- **Mapeo de identificadores.** Se realiza una traducción de identificadores provenientes desde el generador de eventos mediante una tabla dinámica que asigne un valor numérico a cada identificador.
- **Conversiones de punto flotante a punto fijo.** Una restricción existente en las herramientas de síntesis de alto nivel impide trabajar en diseños *hardware* con valores en coma flotante, por lo que deberán traducirse a punto fijo. La distribución ha sido de, 17 bits para la parte entera (1 bit de signo y 16 de dígitos) y 15 bits de parte decimal. Para realizar la traducción, basta con tener en cuenta que un valor decimal con punto fijo con la distribución definida, equivale a un entero, si se desplaza a la izquierda 15 bits, o lo que es lo mismo, si se multiplica el valor por $2^{15} = 32768$. Por lo tanto, una forma de realizar la traducción a punto fijo es, multiplicar el valor flotante por 32768, y luego realizar un *typecasting* a entero de 32 bits.

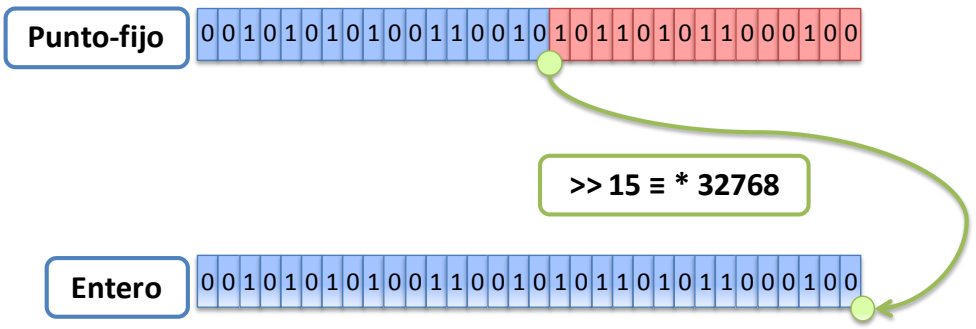


Figura 84. Traducción de punto flotante a punto fijo.

El último paso de ejecución de este hilo es enviar los datos al CE, o, siendo estrictos, preparando un descriptor con los datos a enviar por el DMA.

```
TxRingPtr = &XLlDma_GetTxRing(&Dma);

do {
  Status = XLlDma_BdRingAlloc(TxRingPtr, 1, &BdPtr);
} while (Status != XST_SUCCESS);

BdCurPtr = BdPtr;
XLlDma_BdSetStsCtrl(XLlDma_BdRingPrev(TxRingPtr, BdCurPtr),
            (XLLDMA_BD_STSCTRL_SOP_MASK | XLLDMA_BD_STSCTRL_EOP_MASK));
XLlDma_BdSetBufAddr(BdCurPtr, mensaje);
XLlDma_BdSetLength(BdCurPtr, longitud);

XLlDma_BdRingToHw(TxRingPtr, 1, BdPtr);
```

El código de preparación del descriptor se divide en tres segmentos. El primero consiste en asignar un nuevo descriptor. En caso de que no haya libres, se quedará en un bucle cerrado en espera de que uno se libere (o lo que es lo mismo, hasta que se produzca una interrupción de transmisión DMA). El segundo segmento asigna los campos de control del descriptor (dirección del buffer, longitud del mensaje y bits de comienzo y final de trama). Por último, el descriptor se hace disponible para el bloque *hardware* DMA.

7.11 Hilo de recepción de notificaciones y acciones

El hilo de recepción constituye el código que realiza la comunicación en el sentido contrario. El orden por lo tanto también queda alterado, comenzándose por la configuración de un descriptor DMA preparado para recibir la primera trama. Como ya se adelantó, en el modelo de referencia de procesador de eventos usado en el presente PFC, envía dos tipos de tramas de salida, las notificaciones y las acciones de mercado (figura 81).

```
RxRingPtr = &XLlDma_GetRxRing(&Dma);

XLlDma_BdRingAlloc(RxRingPtr, 1, &BdPtr);

BdCurPtr = BdPtr;

XLlDma_BdSetStsCtrl(BdCurPtr,
            (XLLDMA_BD_STSCTRL_SOP_MASK | XLLDMA_BD_STSCTRL_EOP_MASK));
XLlDma_BdSetBufAddr(BdCurPtr, mensaje);
XLlDma_BdSetLength(BdCurPtr, longitud);

Status = XLlDma_BdRingToHw(RxRingPtr, 1, BdPtr);
```

Esta preparación del descriptor se realiza, explícitamente solo una vez. Llevándose a cabo repetidas veces en la rutina de interrupción de recepción DMA.

La siguiente parte del código consiste en un post-procesado, si fuese necesario. En él se realizarán las acciones contrarias al pre-procesado (mapeado de identificadores, *byte-swap*, etc.).

Por último, el resultado del post-procesado, debe enviarse a través de la red, usando para ello la función típica de sockets BSD destinada a ello: `send()`.

```
lwip_send(socket_cliente, mensaje, longitud, 0);
```

8 Implementación

Una vez diseñada la plataforma queda realizar la implementación, es decir, seguir los pasos necesarios hasta programar la FPGA con el diseño propuesto. Para ello, Xilinx Platform Studio integra perfectamente las diferentes herramientas necesarias para realizar todas las etapas.

Lo primero a realizar es la generación del *netlist*, es decir, realizar la síntesis de aquellos bloques que hayan sido instanciados con su descripción HDL. Para ello, el paso a indicar al XPS es *"Generate Netlist"*.

El siguiente paso es generar el *bitstream*, es decir, el fichero de configuración de la FPGA. Para ello, XPS proporciona la opción *"Generate Bitstream"*, la cual realizará los distintos pasos hasta dicha generación:

- **NGDBuild**: Genera un fichero NGD que concentra la información del *netlist* y de las restricciones del diseño.
- **Map**: Realiza el mapeo del diseño sobre los recursos disponibles de la FPGA.
- **Place&Route**: Realiza la asignación de área y de la conectividad del diseño.

Como se comentará posteriormente en el capítulo 6 de integración y prototipado, en la fase de implementación del sistema completo se siguió un flujo de diseño más complejo a través de la herramienta de PlanAhead para este paso. Esta herramienta permite especificar estrategias específicas de colocado y ruteado que mejoren los resultados tanto de frecuencia como de consumo de recursos.

8.1 Resultados

A continuación se describen los resultados de la implementación en términos de ocupación. En él se indicarán el consumo de cada uno de los recursos disponibles.

Tabla 13. Recursos consumidos por la plataforma hardware.

	Slices	LUTs	Flip-Flops	BRAMs
Usados	4581	9572	8553	12
Totales	20480	81920	81920	298
Porcentaje	22%	11%	10%	4%

9 Conclusiones

En este capítulo se han presentado las necesidades generales del diseño, así como los bloques *hardware* que componen la plataforma y que dan solución a dichos requerimientos. Además se han expuesto algunos de los parámetros configurables y los valores de uso en el presente PFC. A continuación se ha explicado detalladamente la forma de importar un diseño *hardware* propio e integrarlo en la plataforma, así como distintas formas de interconectarlo.

Realizada la plataforma *hardware*, se ha descrito las diferentes alternativas de plataformas *software* que dispone Xilinx para el diseño con sus FPGAs así como su arquitectura de capas que permite personalizarla con el fin de solo incluir aquellas librerías necesarias.

Por último, se explica la estructura de la aplicación empotrada, definiendo su jerarquía, así como los pasos para la configuración de los recursos, tanto *software* como *hardware* de los que hace uso.

Capítulo 5:
Síntesis del coprocesador de eventos

1 Introducción

En este capítulo se describirán las etapas de síntesis de alto nivel y verificación del diseño RTL generado. Se pretende definir el flujo de diseño partiendo de una especificación funcional en SystemC, obtener un modelo sintetizado y optimizado, que pueda posteriormente integrarse en la plataforma para el desarrollo del sistema final.

Se parte, en este caso, como punto de partida, de un diseño SystemC, particionado y basado en la aplicación *software* original que pretender describir una misma funcionalidad mediante un diseño *hardware*.

Se pretende en este capítulo definir directrices sobre los pasos a seguir para la síntesis y verificación de un sistema *hardware* complejo compuesto por un alto número de bloques funcionales.

2 Criterios de optimización

En general, a la hora de realizar la transformación de una descripción funcional a una arquitectura, existen tres factores a optimizar: ciclos de latencia, periodo de reloj y uso de recursos. En general, estos factores dependen entre sí, de tal forma que disminuir el tiempo de ciclo de reloj puede ir asociado a un incremento en el uso de recursos o a un mayor número de ciclos de latencia. Esta dependencia entre los factores hace imposible optimizar todos conjuntamente, siendo necesario establecer un compromiso en función de los requisitos del sistema.

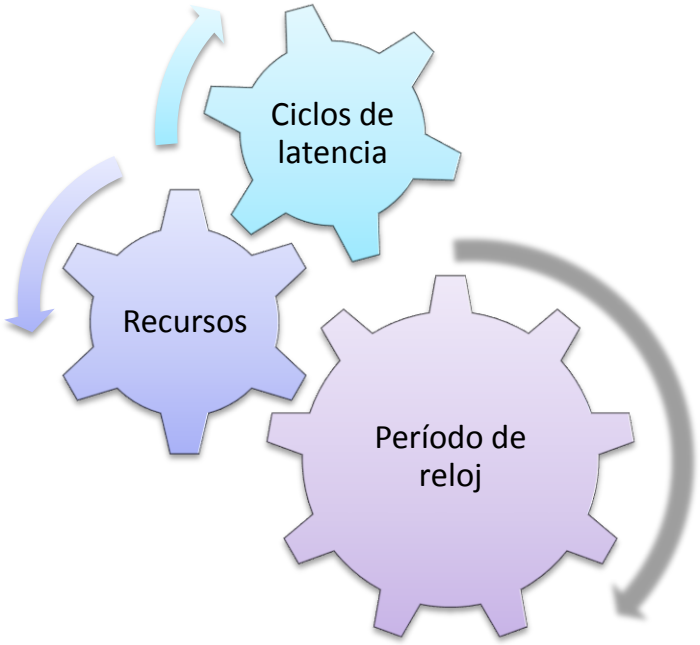


Figura 85. Factores de optimización.

3 Estrategia de síntesis

A la hora de realizar la síntesis, existen principalmente dos estrategias a valorar: *top-down* y *bottom-up*. Ambas se basan en la partición del sistema en módulos más simples y ligeros, que al interconectarlos consiguen la funcionalidad del módulo principal. En general, al módulo que integra e interconecta al resto se le conoce como "*top* del diseño".

La estrategia *top-down* consiste en crear un proyecto único para la síntesis, que tenga como módulo principal el *top*, el cual instancia el resto de bloques, produciendo que la propia herramienta los sintetice.

Por el contrario, una estrategia *bottom-up* se basa en la síntesis por separado de cada bloque integrado en el sistema para finalmente integrarlos todos mediante un módulo top dedicado únicamente a definir la conectividad entre ellos. Es necesario también tener en cuenta los efectos en los bordes de los módulos para facilitar la interconexión. Veremos más tarde que ambos

procesos se pueden combinar para llegar a un compromiso entre tiempo de síntesis y calidad de los resultados.

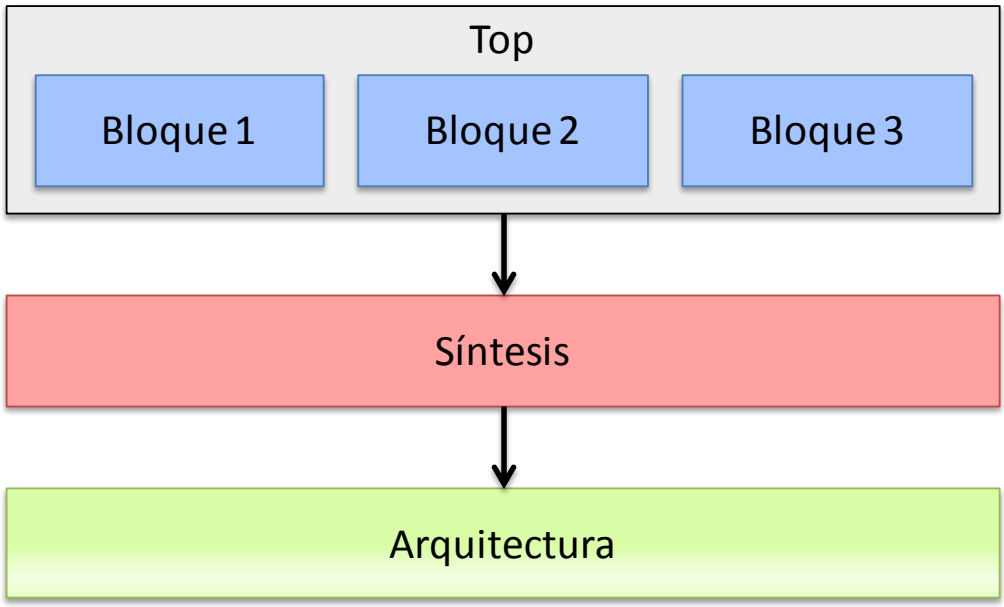


Figura 86. Estrategia *top-down*.

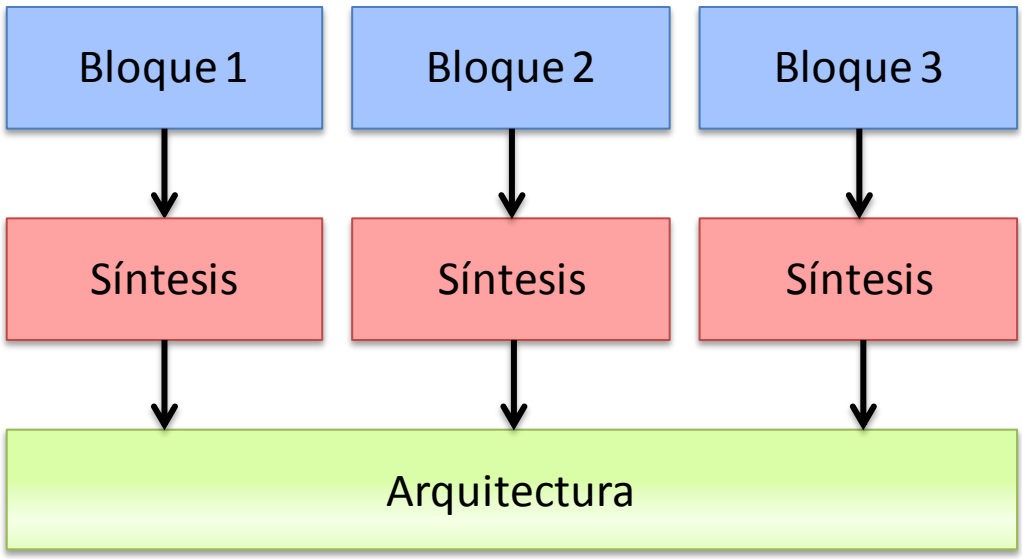


Figura 87. Estrategia *bottom-up*.

En el presente proyecto se ha optado por una tecnología *bottom-up* debido a las ventajas que ofrece y que se enumeran a continuación.

- **Paralelización de la síntesis**. Gracias a la creación de proyectos independientes para cada bloque, y aprovechando los servidores de cómputo, pueden lanzarse

simultáneamente la síntesis de un alto número de bloques, minimizando el tiempo de síntesis.

- **Menor tiempo de depurado**. Gracias a la separación de la síntesis de cada bloque, durante la etapa de depurado en la verificación RTL, pueden realizarse modificaciones en un único bloque y conseguir una nueva versión del diseño sin tener que sintetizar el resto de bloques.

4 Síntesis de alto nivel

Para la síntesis de alto nivel se ha optado por hacer uso de la herramienta C-to-Silicon (CtoS) de Cadence, la cual ofrece la posibilidad de realizar la síntesis partiendo de un diseño en SystemC, y obteniendo como resultado la descripción RTL del sistema en Verilog.

Como ya se adelantó en el capítulo de metodología, las herramientas de síntesis requieren de una guía que les proporcione cierta información sobre el objetivo del sistema, pues existen muchas posibles soluciones que cumplen con los requisitos de funcionalidad. Es por ello, que durante esta fase CtoS exige que se especifiquen ciertos parámetros, que se comentarán a continuación.

4.1 Parámetros de la síntesis

A continuación se enumerarán aquellos parámetros que deben especificarse durante la etapa de síntesis de alto nivel para la herramienta CtoS. Además se indicará la sintaxis para realizar esta especificación en la consola del programa o, lo que es más importante, para un script TCL (*Tool Command Language*) para futuras ejecuciones.

Frecuencia de reloj

Al comienzo de la etapa de síntesis es necesario indicar la frecuencia de reloj, así como su ciclo de trabajo (los valores se dan en términos de periodo de reloj y *offset* de flanco de bajada). Esta restricción guiará a la herramienta de síntesis para añadir más o menos ciclos de latencia así como a replicar o no lógica con el fin de alcanzar el objetivo.

```
define_clock -name nombrereloj -period periodo -rise 0 -fall flancobajada
```

Los valores de periodo, flanco de subida y flanco de bajada se indicarán en picosegundos, como a continuación se muestra en el ejemplo para una señal de reloj llamada *clock* que tiene un periodo de 10ns (frecuencia de reloj de 100MHz) y un ciclo de trabajo del 50%.

```
define_clock -name clock -period 10000 -rise 0 -fall 5000
```

Bucles combinacionales

En el diseño en SystemC pueden existir, y de hecho son simulables, bucles enteramente combinacionales que, según su definición, deberían ejecutarse en su totalidad en un único ciclo de reloj. Evidentemente, esto no suele ser el objetivo del bucle, aunque puede traducirse. Es por ello que ha de indicarse que hacer en estos casos. Puede optarse por varias soluciones:

- **Desenrollar el bucle**. Realiza todas las iteraciones una detrás de la otra en un único ciclo, a costa de replicar la lógica interna del bucle una vez por cada iteración. Favorece una minimización del número de ciclos, aumentando los recursos *hardware* necesarios, así como el periodo de reloj.
- **Romper el bucle.** Es el equivalente a añadir una sentencia `wait()` al final de cada iteración. Así, cada iteración se realiza en un ciclo de reloj. Es normalmente la acción por defecto.
- **Realizar un *pipeline*.** Rompe el bucle de forma que se tardan varios ciclos en realizarse, pero además, varias iteraciones se realizan a la vez, pero cada uno ocupando una etapa del bucle. Esta solución mejora el rendimiento para bucles con gran carga en cada iteración, pero es ineficiente para bucles sencillos al añadir la lógica adicional para el control del *pipeline*.

Para el presente proyecto se realizará una acción de rotura del bucle en todos aquellos bucles combinacionales existentes, para lo cual a continuación se muestra la sintaxis TCL.

```
set combo_loops [find_combinational_loops]
foreach loop $combo_loops {
    break_combinational_loop  $loop
}
```

Funciones no planificables

En la descripción del diseño en alto nivel se permiten las llamadas a funciones globales. En general estas son resueltas por el planificador de la herramienta mediante la síntesis de un bloque independiente y un mecanismo de comunicación mediante un protocolo de petición y respuesta. Sin embargo, pueden darse ocasiones en las que esto no ocurre, por diversas causas, como puede ser una latencia no constante o la llamada a la función desde distintos procesos concurrentes.

En estos casos, la herramienta obliga a realizar una acción de *inline*, es decir, modificar la llamada a la subrutina por una copia íntegra del código asociado a dicha función. Esta acción genera un aumento en el consumo de recursos. Ello requiere una modificación del código fuente en algunos casos, pero en otros es inevitable.

También se permite realizar un *inline* de las funciones que no producen estos conflictos. Sin embargo, llevar esta acción sobre todas las funciones del diseño suele conducir a unos costes de recursos demasiado elevados para luego poder ser implementado en un dispositivo físico. A continuación se muestra la sintaxis para realizar *inline* de una función específica.

```
inline /designs/nombre_proyecto/modules/nombre_modulo/behaviors/funcion1
```

Asignación de memorias

Si existen vectores de datos en la descripción SystemC del diseño, se debe decidir los recursos a utilizar para almacenarlos. Existen diferentes opciones, que se enumeran a continuación:

- **RAM interna**. Se usa como recursos los bloques BRAM o las LUTs de los SLICEM para almacenar la información. El uso de uno u otro depende de los ciclos de espera introducidos entre la disposición de la dirección y la lectura del dato.
- **Registros**. Todos los datos se almacenan en registros. Esta solución solo debe realizarse en casos de vectores de pocos elementos y en los cuales los elementos no son excesivamente grandes en términos de longitud de bits.
- **RAM de terceros**. Pueden imponerse como recurso de memoria la descripción HDL de una RAM de terceros.

El uso de registros o BRAM/LUTs dependerá de la disponibilidad de recursos del dispositivo y del uso que haga el resto del diseño. Por el alto uso de registros del diseño, en el presente proyecto se ha decidido por usar RAM interna para todos los vectores del diseño, como se muestra a continuación.

```
set memories [find $top_path/modules/*/arrays/*]
foreach mem $memories {
    allocate_builtin_ram $mem
}
```

Uso de DSPs

Para cada modulo, puede indicarse si debe o no usar DSPs. En general, y dependiendo de la naturaleza del diseño, puede ser interesante que un módulo haga uso de los DSPs para aquellas operaciones para los que han sido diseñados, y otros, menos importantes, usen lógica programable para realizarlas, a costa de un mayor número de ciclos de latencia.

```
use_dsp /designs/$modulo
```

Relajación de la latencia

Esta opción permite a CtoS incrementar el número de ciclos de latencia de una función, con el fin de facilitar la planificación de la misma. Es importante, no permitir esta acción sobre aquellas funciones dedicadas a implementar los protocolos de comunicación. Por ejemplo, no es conveniente permitir relajar la latencia de las funciones que hacen accesos a memoria, o que se dedican a leer y escribir sobre los puertos de control de comunicación, ya que añadir estados en medio de una de estas rutinas puede significar que el protocolo deje de ser correcto. A continuación se muestra como desactivar la relajación para todas las funciones del diseño.

```
set behaviours [find $top_path/modules/*/behaviors/*]
foreach beh $behaviours {
    set_attr relax_latency "false" $beh
}
```

4.2 Automatización de la síntesis

Como se adelantó anteriormente CtoS permite lanzarse en modo *batch*, haciendo uso de *scripts* TCL. De esta forma, realizar la síntesis durante la etapa de depuración es mucho más rápido, sin la necesidad de elegir todos los parámetros anteriormente descritos en cada síntesis.

Además, gracias a la estrategia elegida *bottom-up*, pueden lanzarse paralelamente varios procesos de síntesis simplemente lanzando la aplicación en modo *batch* y con el script previamente escrito. Para lanzar la herramienta en modo batch, se especificará el alias de la aplicación en dicho modo, el script a ejecutar y, opcionalmente, un fichero de log donde almacenar la salida del proceso.

```
ctos scripts/ctos.tcl -log log/ctos.log
```

En el fichero de script TCL existirán dos etapas diferenciadas. La primera es la que configura el proyecto de CtoS, indicando los ficheros fuente, la frecuencia de reloj, así como el dispositivo de prototipado final.

```
new_design $modulo
set_attr auto_write_models "true" /designs/$modulo
define_sim_config -model_dir "./model" /designs/$modulo
set_attr source_files [list src/$modulo.cpp] /designs/$modulo
set_attr compile_flags " -w -I../../include -I./src" /designs/$modulo
set_attr auto_save_dir $modulo /designs/$modulo
define_clock -name clock -period 10000 -rise 0 -fall 5000
set_attr implementation_target FPGA [get_design]
set_attr fpga_target [list Xilinx Virtex5 xc5vfx130t-1-ff1738]
[get_design]
set_attr fpga_install_path [exec which xst] [get_design]
set_attr fpga_work_dir "./fpga_work" [get_design]
```

La segunda parte del script es la que define los diferentes pasos de la síntesis previamente citados en este apartado. A continuación se muestra un ejemplo.

```
set combo_loops [find_combinational_loops]
foreach loop $combo_loops {
    break_combinational_loop  $loop
}
foreach mem [find $top_path/modules/*/arrays/*] {
    allocate_builtin_ram $mem
}
foreach beh [find $top_path/modules/*/behaviors/*] {
    set_attr relax_latency "true" $beh
}
schedule -effort high -passes 200 /designs/$modulo
allocate_registers
write_rtl -recursive -o [concat ./[string trim $model]] $top_path
/modules/$modulo
```

4.3 Resultados

A continuación se detallan algunos resultados obtenidos de la síntesis de alto nivel que pueden guiar al diseñador a una optimización del diseño en las fases más tempranas del flujo.

4.3.1 Control and Data Flow Graph

En el CDFG se representan los distintos caminos que puede seguir la ejecución del bloque diseñado, así como las acciones realizadas en cada etapa de cada uno de ellos. Pueden verse, además, los bucles y sus condiciones así como las bifurcaciones generadas en base a alguna condición. Es de especial interés para el diseñador esta representación del diseño ya que sirve para cuantificar las latencias del sistema para cada posible camino, pudiendo además insertarse ciclos de segmentación para igualar la latencia y simplificar las máquinas de estado de acceso a los recursos de la función representada. En esta vista puede además especificarse latencia máxima en ciclos entre dos puntos del código fuente. En la figura 89 se representa un ejemplo de este grafo.

Este análisis de noción temporal es realizado en la primera etapa de la síntesis de alto nivel y tiene como fin representar, de forma gráfica, el flujo de ejecución del bloque diseñado, permitiendo además añadir etapas en el mismo y partiendo otras etapas para modificar dicho flujo.

4.3.2 Análisis de ciclos

El análisis de ciclos representa la ruta crítica del bloque indicando los retardos de cada uno de los elementos que la componen, indicando además el *slack* de dicha ruta. Se incluyen además los retardos debidos al *fan-out* de la señal representada, mediante la separación de los bloques representantes de los elementos, como se muestra en la figura 88.

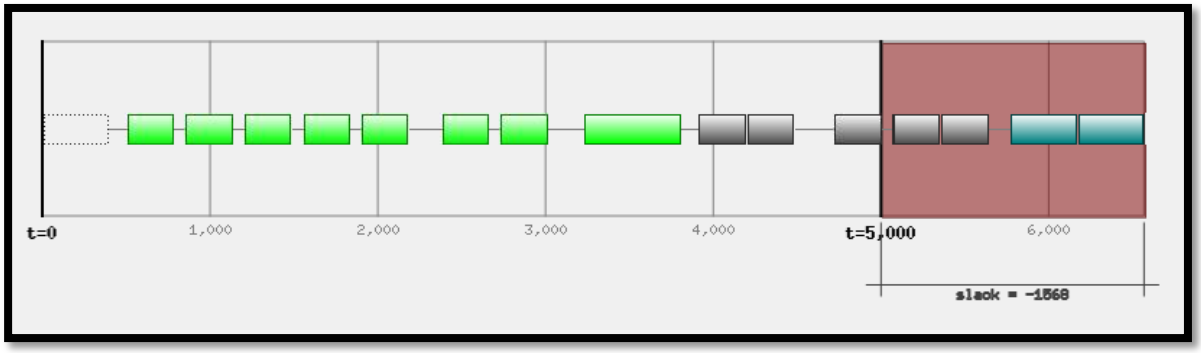


Figura 88. Análisis de ciclos.

En el ejemplo de la figura 88 el diseño está sintetizado con una frecuencia de funcionamiento de 200MHz, es decir, un periodo de reloj de 5 ns. Como puede observarse, existe un *slack* negativo de 1,56 ns, lo que implica que la frecuencia propuesta no es alcanzable con el diseño propuesto. La solución a este problema es reducir la frecuencia o partir la ruta crítica añadiendo un `wait()` en el fichero fuente en SystemC, o directamente desde CtoS.

Otros datos que se pueden observar son el tiempo de *hold* del *flip-flop* origen de la ruta crítica y los retardos de la propagación de la señal, representados mediante la separación de los componentes.

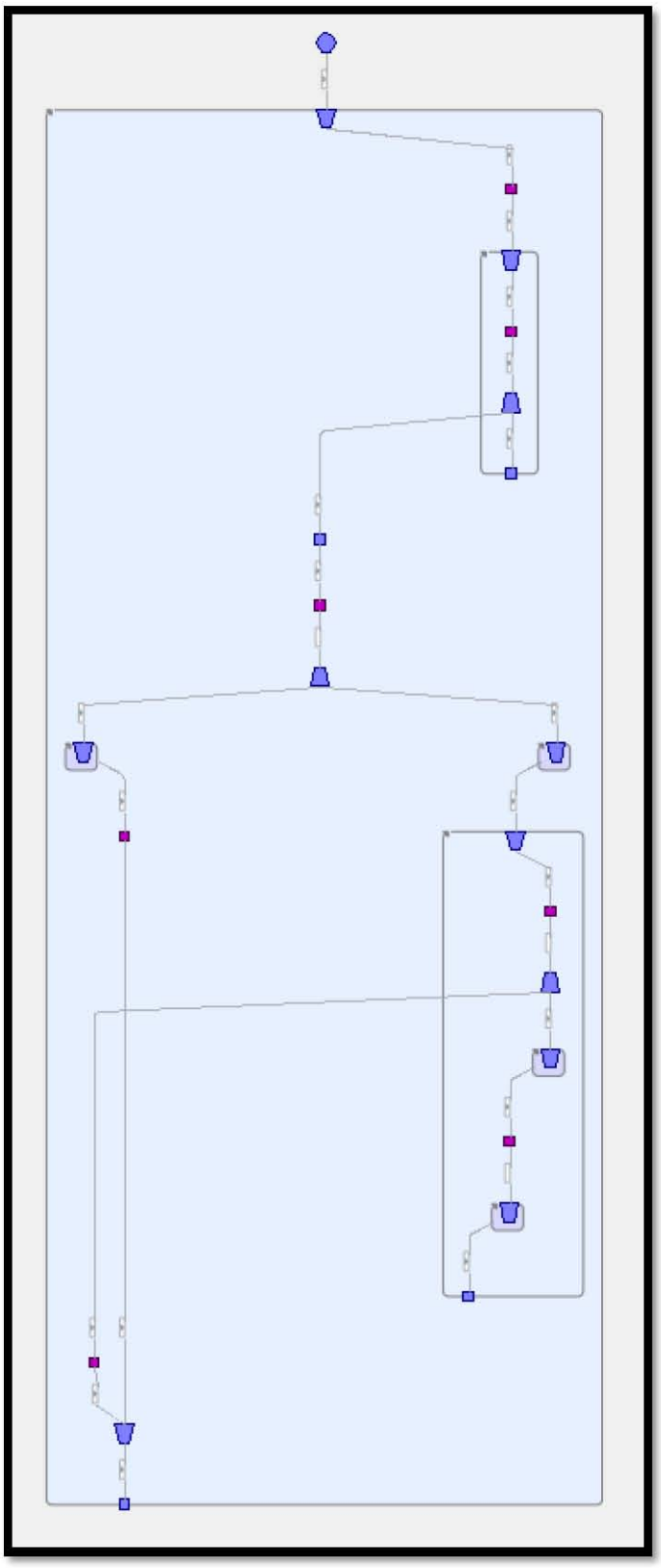


Figura 89. Vista de Control and Data Flow Graph.

4.3.3 Análisis de área

Aunque en un diseño sobre FPGAs los análisis de área no son significativos, al estar pre-establecidos los elementos básicos configurables que conforman el diseño físico, esto cambia en el diseño de ASICs donde es posible usar distintas librerías de células lógicas con un área diferente y que conforman el área total del diseño. Es por ello que la herramienta de síntesis de alto nivel permite, tras realizar el flujo de la misma, realizar un análisis de área consumida por cada sub-bloque del diseño, tal y como se muestra en la figura 90.

En ella, se muestra el consumo de área según en base a los elementos que constituyen el diseño, y para una función en concreto. Cabe recordar que la herramienta de síntesis generará un bloque para cada función del código fuente, generando además lógica adicional para las llamadas a cada función. Es por esto que el consumo de área puede visualizarse partido por funciones.

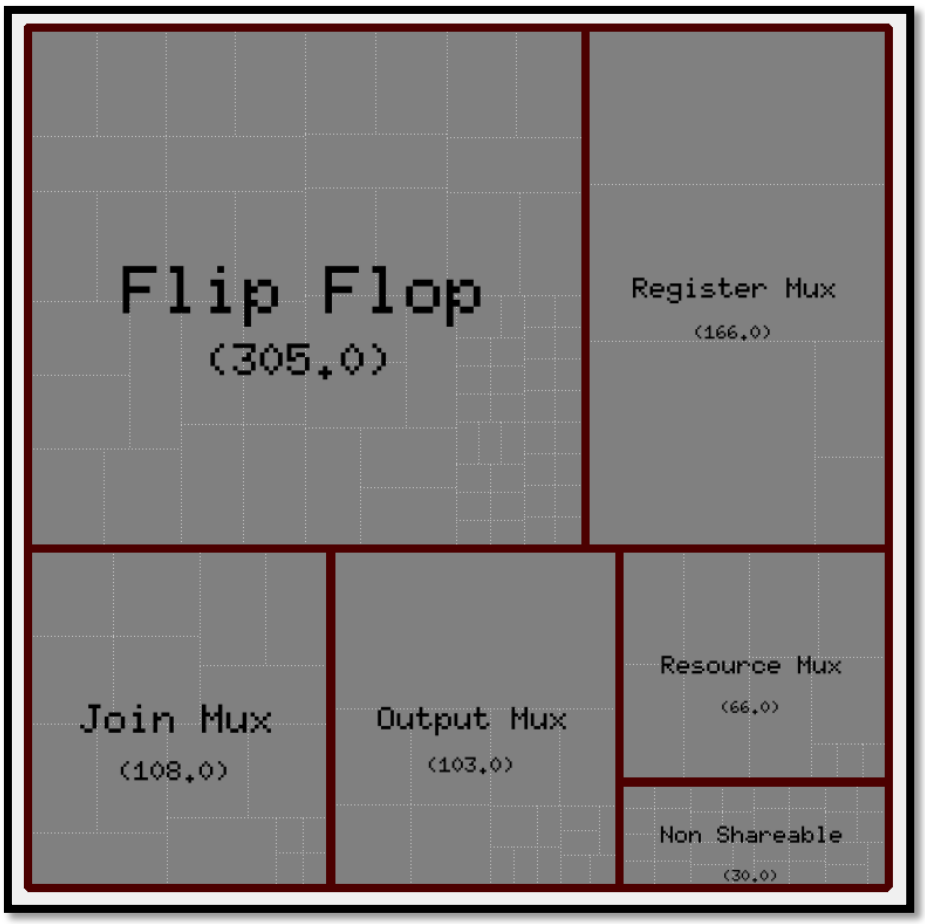


Figura 90. Análisis de área.

5 Verificación RTL

Una vez realizada la síntesis de alto nivel, el diseño pasa a estar constituido por un número de bloques, cada uno definido por su código en Verilog. Para comprobar que la síntesis ha sido

correcta, esto es, que el diseño escrito en HDL se comporta como se esperaba, se deben realizar simulaciones con los mismos eventos de entrada que el diseño en alto nivel. Si todo es correcto, deben generarse los mismos patrones de salida.

Partiendo de que el diseño en SystemC era correcto, es decir, que para los eventos de entrada proporcionados por el *testbench*, las salidas correspondían con las de la aplicación original, la verificación RTL pasa a basarse en una comparación entre las señales del modelo en alto nivel y el de nivel RTL.

Precisamente por esto, si los estímulos de entrada coinciden con los del *testbench* en alto nivel, la depuración del diseño a nivel RTL se simplifica notablemente, además de que se acelera, ya que basta con comparar las salidas y las formas de onda de ambas simulaciones.

Para este tipo de verificación, CtoS genera un *wrapper* en SystemC que instancia al modelo en Verilog generado, permitiendo luego realizar una simulación usando el mismo *testbench* que en SystemC. Este *wrapper* además, realiza dos instancias del diseño en cuestión, una del modelo original en alto nivel en SystemC y otro del sintetizado en Verilog. De esta forma, pueden presentarse en pantalla ambas señales y compararlas en el tiempo, haciendo que la etapa de verificación sea mucho más rápida.

A continuación se muestra la jerarquía de módulos a simular, primero en alto nivel (figura 91), y la reutilización del mismo *testbench* para simular el diseño sintetizado (figura 92).

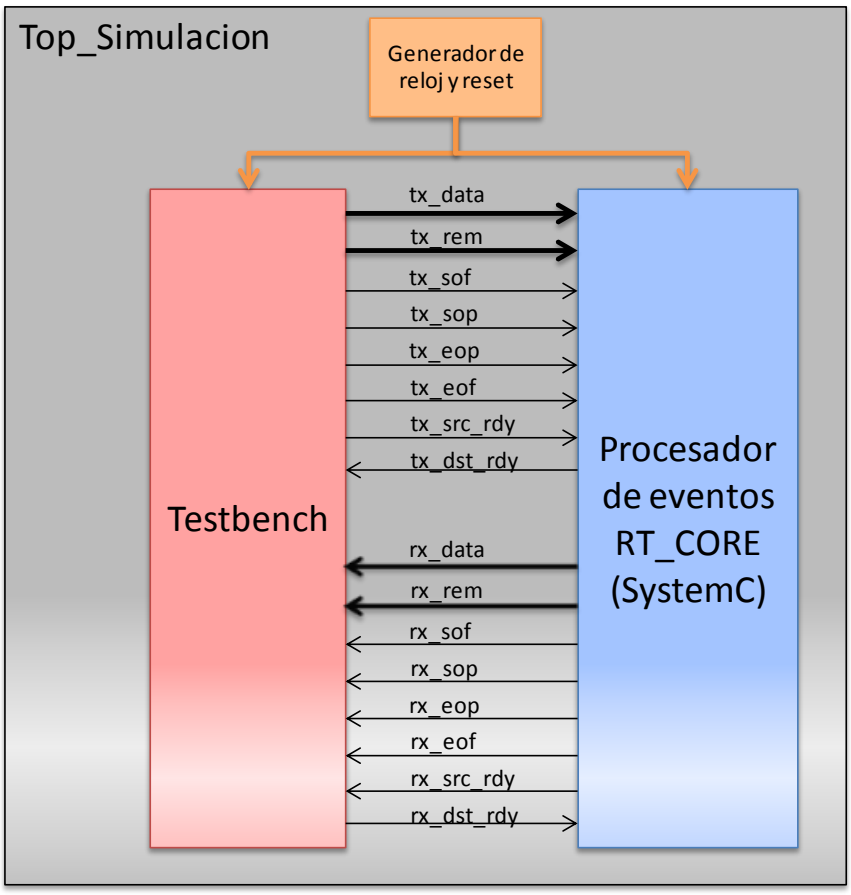


Figura 91. Esquema de simulación en alto nivel.

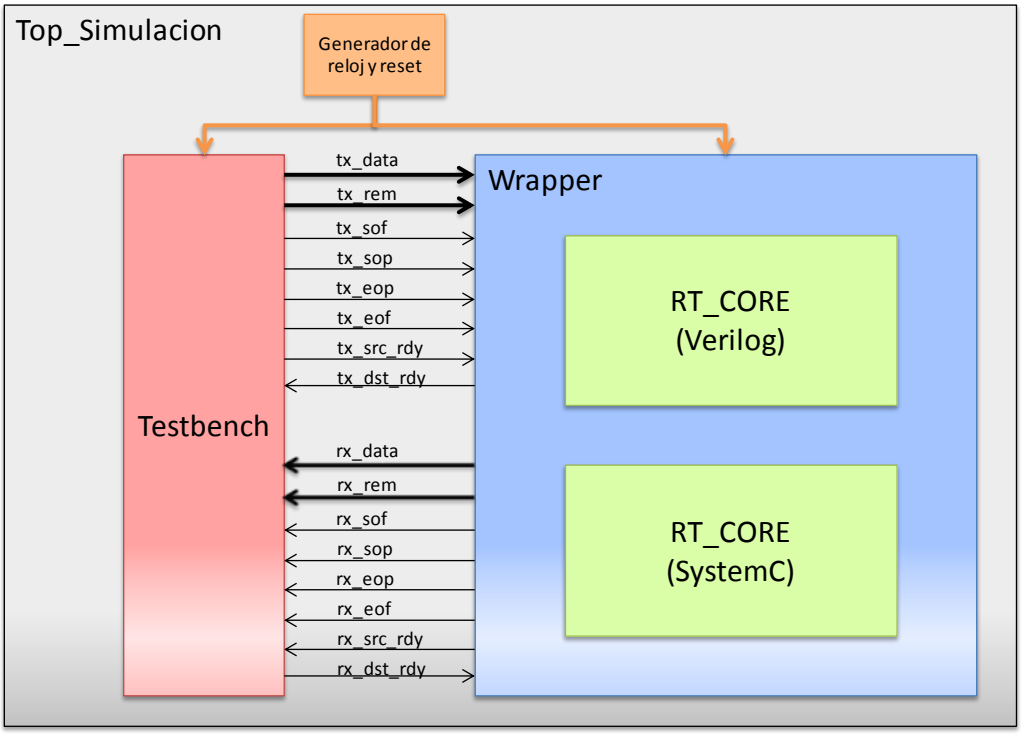


Figura 92. Esquema de simulación RTL.

Todo el proceso de verificación, a diferentes niveles de abstracción se realiza en el entorno de simulación de Cadence Incisive Enterprise Simulator (IES). Al igual que la herramienta de síntesis, permite lanzarse tanto en modo *batch* como en modo gráfico en el que se representan las señales como formas de onda y otras vistas gráficas y de depurado del diseño.

5.1 Diseño del testbench

Para el diseño del *testbench* cabe destacar que las tramas de entrada se encuentran almacenadas, individualmente en ficheros binarios y en el formato por el que son recibidas por la red. Es por ello que, teniendo en cuenta la necesidad de preparar las tramas para su posterior procesado (etapa de pre-procesado que en el sistema final será realizada por el procesador empotrado (PowerPC 440 de la FPGA), esta tarea deberá realizarla el *testbench* antes de enviar los datos al CE.

Además de leer los ficheros binarios y realizar el pre-procesado de las tramas, otro punto importante del *testbench* es que debe implementar el protocolo de comunicación del bloque a simular, en este caso LocalLink. Esto implica que el *testbench* pasa de ser un simple generador de tramas, a ser un sistema que se realimenta del dispositivo bajo test, parando las transmisiones cuando este deja de estar preparado para la recepción de nuevos datos. Además, deberá estar compuesto de dos hilos de ejecución ya que, como se comentó anteriormente, la interfaz del CE está compuesta por dos interfaces LocalLink, proporcionándole así una comunicación *full-duplex*.

La figura 93 representa en un diagrama de flujo la ejecución del *testbench*.

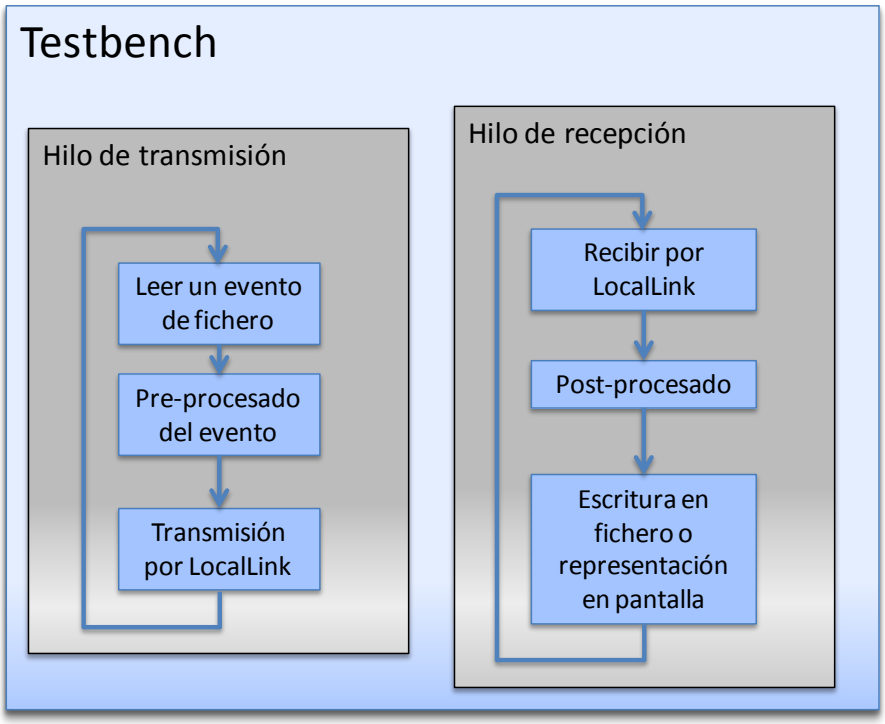


Figura 93. Estructura del *testbench* con interfaz LocalLink.

5.2 Simulación del diseño

Para la simulación y consecuente verificación del diseño generador por CtoS se ha usado el *wrapper* que instancia el modelo original en alto nivel y el diseño sintetizado por la herramienta. Las señales de cada uno se han añadido a la representación de la forma de onda con el fin de comprobar que coinciden.

A continuación se muestra una captura de la forma de onda en la que se muestran varias transmisiones y recepciones (figura 94). En particular se envía una regla, generándose notificaciones de creación de la regla en el sistema, y de las órdenes de ejecución que esta engloba. Las señales se ven duplicadas. El conjunto superior de señales se corresponden con el modelo RTL, mientras que las inferiores son sus análogas del modelo SystemC.

En la figura 94 se han introducido cinco marcadores, con nombres *TimeA*, *TimeB*, *TimeC*, *TimeD* y *TimeE*. La primera, marca el comienzo de transmisión de la regla, por parte del *testbench*, y con destino el modelo hardware bajo test. Los sufijos de *tx* y *rx* para denominar las señales de un sentido u otro de la comunicación han sido nombrados desde el punto de vista del bloque hardware, por lo que las señales con sufijo *rx* implican el sentido con origen el *testbench* (o el DMA en el sistema completo) y destino el coprocesador de eventos, y viceversa.

En las transmisiones, tanto de eventos como de tramas de reglas se han simulado las interrupciones en la comunicación que se producirán en el sistema real, debido a la compartición la matriz de interconexión interna del bloque de procesador, que impedirá que todos los envíos se realicen en una sola ráfaga. Al comienzo de la transmisión de la trama de reglas, puede observarse que la señal *ll_src_rdy_rx_n* se desactiva durante unos instantes, para luego seguir por

el mismo punto del paquete datos. Estas interrupciones deben ser previstas por el destino, poniéndose a la espera de que el origen vuelva a estar disponible, y seguir con la recepción de datos.

Puesto que el diseño en SystemC ya había sido testado, y comprobado su correcto funcionamiento, la correlación entre las señales del modelo RTL y este, implican un correcto funcionamiento del modelo sintetizado.

Como ya se adelantaba, la relajación de latencia en las opciones de síntesis de CtoS provoca que, en ocasiones, los instantes temporales en los que ocurren ciertos eventos en el sistema, no sean los mismos en el diseño original y en el sintetizado. En el instante *TimeE*, por ejemplo, puede observarse que comienza la transmisión de una notificación (trama de salida del coprocesador de eventos), mediante la activación de la señal *ll_sof_tx_n* en el conjunto de señales del modelo en Verilog. Sin embargo, su gemela en el modelo SystemC ya había sido activada unos instantes antes. Es decir, modelo en Verilog tiene una latencia mayor en su ejecución, lo que produce que se envíe más tarde la trama de salida.

Como es evidente, para la comprobación de correcto funcionamiento del modelo RTL, una vez comprobado de forma detallada en forma de onda, se realiza una simulación en modo batch.

Para ello, se modifica el *testbench* para que escriba en fichero todas las tramas de salida que genere el coprocesador de eventos. Así, basta con comparar los ficheros de salida, con los generados por el modelo SystemC.

Los tiempos de simulación en modo forma de onda, son muchísimo mayores que en línea de comandos, lo cual haría imposible realizar simulaciones de una gran cantidad de eventos.

5.3 Interacción RTL-SystemC

De forma ideal, cabe esperar que la herramienta de síntesis genere un diseño RTL que se comporte de forma idéntica con su diseño de entrada en alto nivel. Sin embargo, por la naturaleza de nivel de abstracción mayor, suele ocurrir que una descripción en alto nivel tenga diferentes posibles interpretaciones o simplemente sus descripciones no sean sintetizables por la herramienta, provocando discordancias entre los diseños.

Uso de variables globales

Aunque las variables globales de C están soportadas por SystemC y por la herramienta de síntesis CtoS, puede provocar errores funcionales en el diseño Verilog si se hace un uso incorrecto de ellas. El soporte de variables globales por CtoS va destinado a su uso dentro de un mismo proceso, de forma que la variable sea común tanto a la función principal ejecutada por el hilo de ejecución como las rutinas que sean llamadas desde esta. Sin embargo, en módulos en los que existan dos o más hilos paralelos (*SC_THREAD*), el uso de una variable global con objetivo de comunicar ambos procesos es erróneo y puede acarrear disfunciones en el diseño generado.

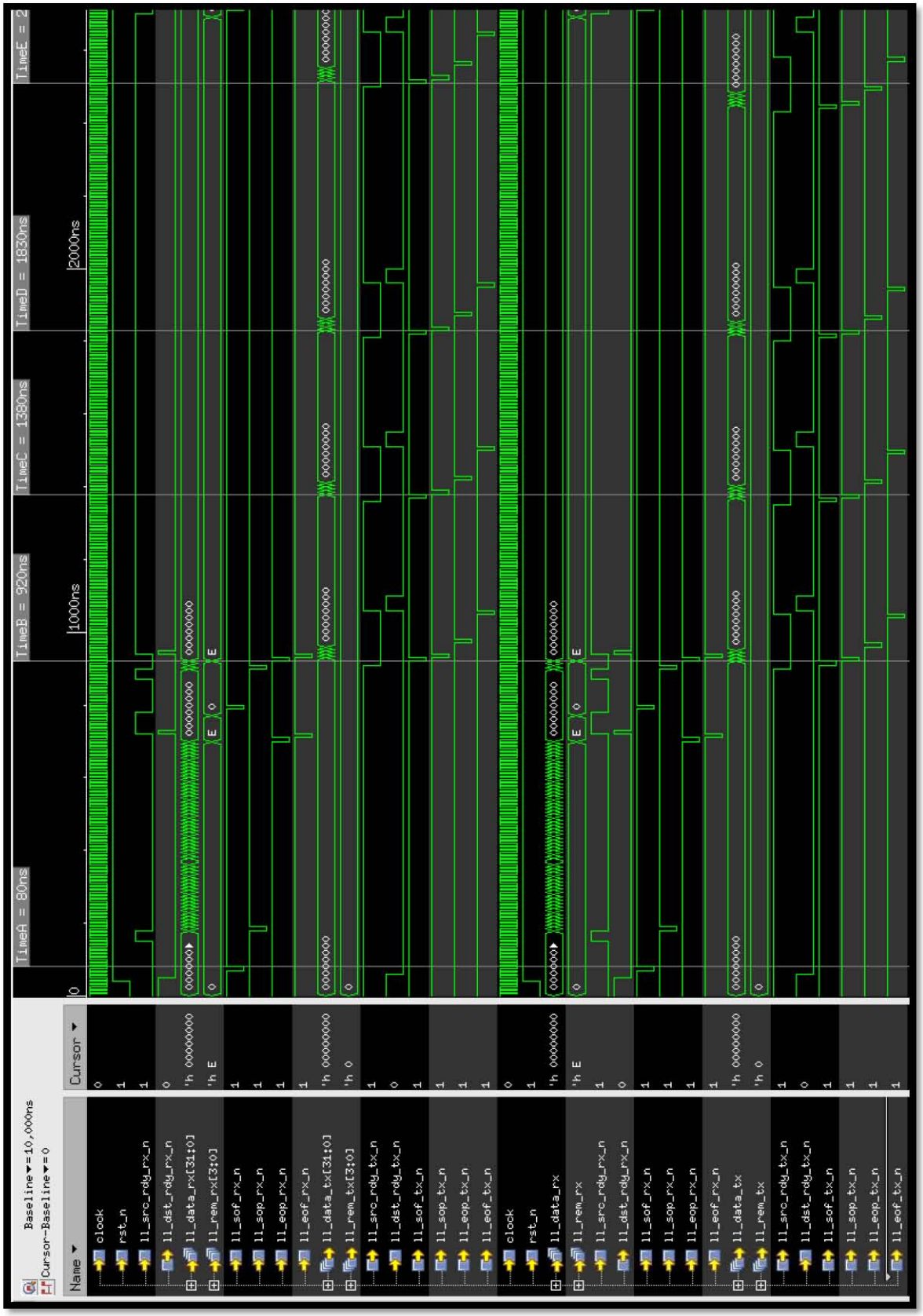


Figura 94. Simulación del diseño RTL y SystemC.

Para solucionar este problema, basta con hacer uso de canales de SystemC, como puede ser, en su modelo más simple, un *sc_signal* del tipo de dato a comunicar. Mediante el uso de este tipo de estructura de datos y sus métodos de lectura y escritura del mismo, la comunicación entre los procesos es segura.

Ancho de los datos

Al realizar asignaciones de datos de distinto ancho es importante que todas estén controladas por el diseñador, mediante las funciones de obtención de rango de los datos. Así, si se desean obtener los últimos n bits de un dato para almacenarlo en una variable de ese ancho, es importante hacerlo explícitamente.

Si por el contrario la asignación se hace de variable a variable, aunque es cierto que tanto el compilador de C como la herramienta de síntesis harán un truncamiento, no se tiene la certeza de que ambos cojan los mismo bits de la variable origen (podría pasar que la herramienta de síntesis cogiese los n bits más significativos y el compilador los n bits menos significativos). Esto provocará que la simulación del diseño en SystemC de un resultado y su diseño equivalente en Verilog de otro.

Relax Latency

Como ya se adelantó, la herramienta de síntesis CtoS tiene la opción de relajar las latencias, permitiendo al planificador alargar la ejecución de un módulo añadiendo registros intermedios. Aunque en el procesamiento de datos esto puede ser beneficioso (en general el planificador añadirá registros intermedios cuando encuentre un punto de latencia crítica que podría perjudicar a la frecuencia máxima del diseño), en las funciones de acceso a los puertos de salida o a las señales de control puede hacer que el diseño deje de funcionar correctamente.

Así, si se permite relajar la latencia, por ejemplo, en una función de control de lectura o escritura de una FIFO, esto puede provocar que se inserten datos duplicados en la cola, o que se extraigan varios elementos, lo cual corrompe los datos del sistema. Esto mismo ocurre por ejemplo, si se relaja la latencia en las funciones que gestionan las señales de control del protocolo de comunicaciones, como es en este caso, LocalLink.

Inicialización de las memorias

Realizar una inicialización de las memorias en SystemC es sencillo debido a su naturaleza del lenguaje C++. Una posibilidad, por ejemplo, es realizar dicha inicialización en el constructor del módulo. Otra solución es realizar la inicialización mediante un bucle combinacional en el ciclo de reset del módulo.

Sin embargo, ninguna de estas soluciones es válida para la síntesis, provocando que el diseño sintetizado parta de unas memorias sin valores definidos, frente a su diseño en alto nivel con valores previamente establecidos.

Para inicializar las memorias en Verilog, se ha usado la directiva `$readmemb` que permite inicializarla a través de un fichero externo que contenga los datos, separados por saltos de línea para cada posición de la memoria.

6 Síntesis lógica

Una vez comprobado el correcto funcionamiento del diseño a nivel RTL, el siguiente paso para poder implementarlo es realizar la síntesis lógica, consistente en realizar una transformación desde un nivel RTL al nivel de puertas lógicas, en este caso primitivas de Xilinx interconectadas formando un *netlist* completo en formato EDIF.

Este paso de síntesis lógica dentro del flujo de diseño está soportado dentro del entorno de Xilinx por la herramienta XST (Xilinx Synthesis Tecnology) [63] que permite realizar la síntesis lógica del CE en la plataforma generada desde XPS directamente a nivel RTL. Sin embargo, los resultados obtenidos por XST son, en general, peores a los de otros entornos especializados como es el caso de Synopsys Synplify Premier.

6.1 Estrategia *bottom-up*

Debido a la complejidad del diseño y a los tiempos de síntesis se ha optado por seguir la estrategia *bottom-up* ya adoptada durante la etapa de síntesis de alto nivel. Esta estrategia está soportada en Synopsys Synplify mediante un sistema de gestión de proyectos y subproyectos.

Para poner en práctica esta estrategia, es necesario crear un proyecto para cada módulo, un proyecto para el top y en este incluir como subproyectos, los proyectos anteriormente creados.

De esta forma, Synplify realizará la síntesis de cada módulo por separado, para luego realizar una optimización global de todo el diseño.

6.2 Estrategia *top-down*

Alternativamente es posible seguir una estrategia *top-down*, en la que se crea un único proyecto que incluye todos los ficheros de descripción RTL de los módulos, y señalando el jerárquicamente superior como top del diseño.

A partir de ahí, Synplify genera la estructura del diseño mediante las instancias realizadas en los distintos ficheros, sintetizando aquellos módulos que conformen el diseño completo. Los resultados optenidos

6.3 Comparativa entre estrategias

Con el creciente incremento de capacidad de elementos lógicos programables en las FPGAs los tiempos de síntesis e implementación han ido creciendo en sintonía. Esto produce que, en muchos casos, sea inviable seguir estrategias clásicas de diseño hardware [64].

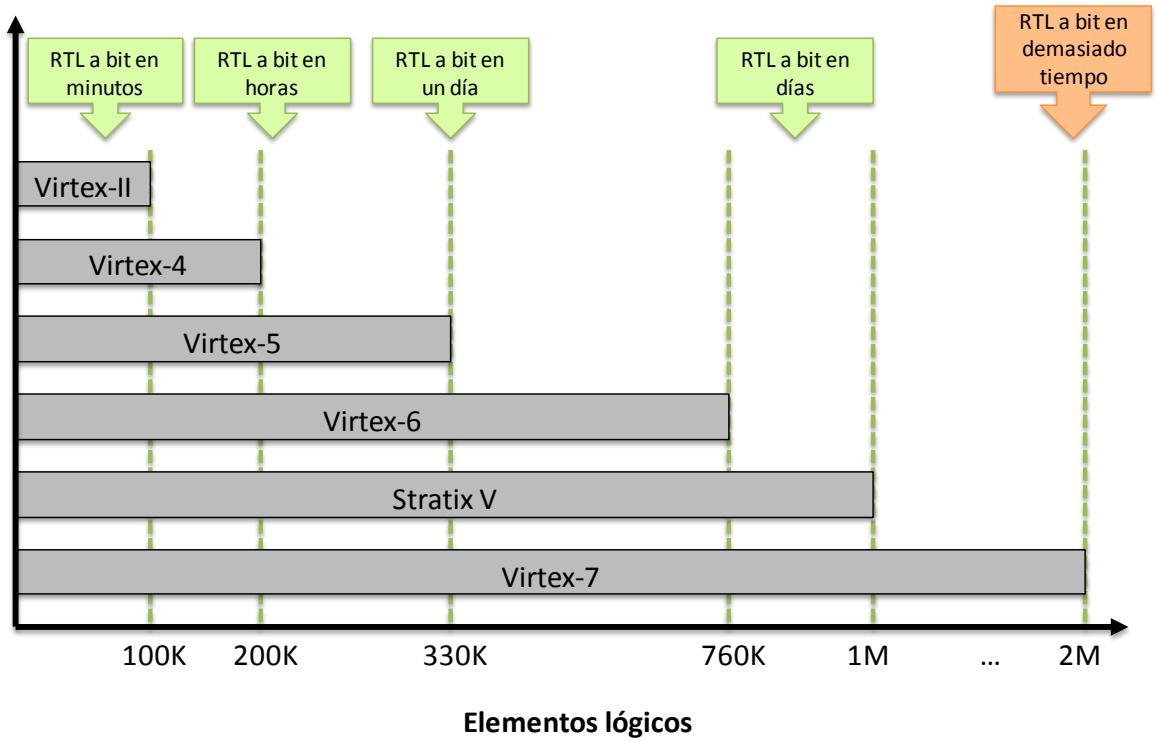


Figura 95. Tiempos de síntesis e implementación en familais de FPGA de Xilinx.

En casos donde el diseño es demasiado grande puede ocurrir que una estrategia *top-down* sea completamente inviable debido al tiempo necesario para realizar la síntesis o por los requerimientos de memoria que necesitaría la máquina donde se realizase. Por otro lado, en este tipo de diseños es normal que este sea partido en sub-módulos y desarrollados en paralelo, por lo que se precisan de modificaciones en el flujo de diseño para dar solución a estos requerimientos.

Una solución acertada para estos problemas ha sido la adopción de flujos *bottom-up* en el que cada sub-módulo es desarrollado mediante un flujo estándar de diseño, para luego ser integrado en un proyecto final con el fin de unirlos. Además, en la síntesis del sistema completo pueden lanzarse las síntesis de varios de estos sub-módulos en paralelo, reduciendo así los tiempos de dicha etapa del flujo. Esta estrategia tiene por defecto un empeoramiento en la calidad de los resultados finales.

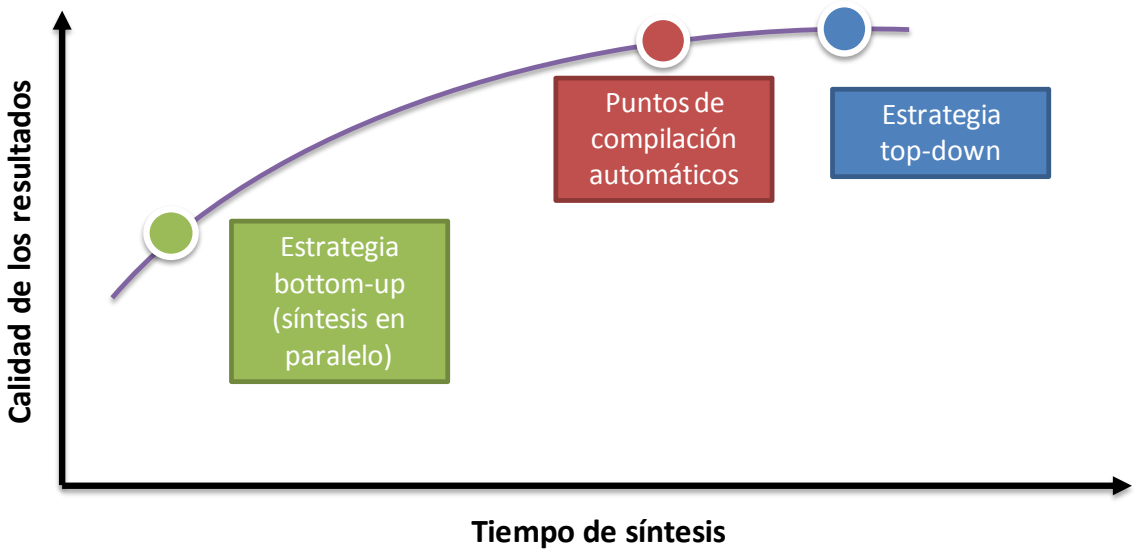


Figura 96. Calidad de los resultados frente a tiempos de síntesis.

En el caso de la herramienta de síntesis Synplify, existe además una estrategia de puntos de compilación automática con diseño incremental donde la propia herramienta crea particiones en el diseño de forma automática y, en donde es posible, mantiene los resultados de las síntesis anteriores de las particiones que no hayan sufrido cambios. Si además se activa el modo de multi-procesado cada partición o *Compile Point* es sintetizado en paralelo a los demás, reduciendo así los tiempos de síntesis.

En el caso de la estrategia *bottom-up* Synplify realiza una optimización global tras la síntesis de todos los sub-módulos del diseño, incrementando ligeramente el tiempo de síntesis pero consiguiendo una mejora notable en la calidad de los resultados, ahorrando recursos innecesarios en los bordes de los bloques.

6.4 Opciones de síntesis

A continuación se describen aquellas opciones a activar en las opciones de síntesis de la herramienta, con el fin de obtener los mejores resultados para el modelo de referencia con el que se trabaja.

Disable I/O Insertion

Consiste en impedir la inserción de bloques de entrada/salida (IOB) de la FPGA para las señales de entrada y salida del top del diseño. Para el caso que nos ocupa, es importante activarlo ya se trata de sintetizar un bloque IP que será integrado en la plataforma final que se diseña siguiendo otro flujo de diseño diferente. Por tanto las E/S del IP estarán conectadas a otras señales internas de la plataforma.

Por defecto, las herramientas de síntesis dispondrán de IOBs para los puertos del top, entendiendo que dichas señales estarán asociadas a pines de la FPGA, con el fin de conectarlas a dispositivos externos.

FSM Compiler

Se trata de un optimizador de máquinas de estado. Synplify ofrece la posibilidad de optimizar la lógica de estado siguiente de las instancias de máquinas de estado del diseño que encuentre, siguiendo una estrategia diferente de codificación de estados en función del número de estos, según la siguiente tabla.

Tabla 14. Codificación de estados de FSM Compiler.

Número de estados	Tipo de codificación
< 5	Secuencial
5 – 24	One-Hot
> 24	Gray

Resource Sharing

Permite compartir recursos con el fin de reducir el consumo del área del dispositivo de implementación, a costa de reducir la frecuencia.

Pipelining

Permite que varias operaciones se realicen a la vez sobre el mismo recurso, partiendo dicha ejecución en etapas, y permitiendo que cada dato se encuentre en una de ellas. En las tecnologías Virtex-5 de Xilinx, esta optimización va asociada a las memorias ROMs y a los multiplicadores del diseño.

Enable Advanced LUT Combining

Prepara el fichero *netlist* de salida para una posterior combinación de LUTs en los diseños sobre FPGAs de Xilinx.

7 Resultados

En este apartado se presenta la comparación ente los resultados de síntesis, en términos de consumo de recursos y frecuencia máxima de utilización, para las estrategias de síntesis *bottom-up* y *top-down*.

Además se presentarán los resultados para las FPGAs Virtex-5 FX130T y Virtex 5 FX70T de Xilinx, de las placas ML507 y ML510.

Los resultados, en los casos de estrategia *bottom-up*, serán además presentados mediante la partición realizada del diseño: interfaz, estado del sistema, núcleo de procesado y ejecutor de resultados.

La interfaz incluye la funcionalidad de control del protocolo LocalLink y las FIFOs de comunicación. El módulo de estado del sistema comprende todas las memorias internas, las cuales almacenan las reglas activas y por activar que actualmente ha recibido el sistema, así como el bloque encargado de reconocer una trama de reglas recibida por LocalLink y mapearla en las memorias internas. El núcleo de procesado es el encargado de, una vez recibido un evento, comprobar si los valores recibidos están asociados a una regla activa y realizar las operaciones necesarias. En caso de que se cumpla la condición, el ejecutor de resultados realizará las tareas asociadas a dicha regla.

7.1 Resultados estrategia *bottom-up*

Tabla 15. Resultado de la síntesis *bottom-up*.

Bloque	Ruta crítica (ns)	Frecuencia (MHz)	LUTs	DSPs	FlipFlops	BRAM
Interfaz	7,589	131,8	3218	0	1477	0
Estado	5,676	176,2	9890	0	14223	53
Núcleo	5,446	183,6	10120	4	18800	0
Ejecutor	6,779	147,5	11544	24	14484	0
Total	**7,589**	**131,8**	**35772**	**28**	**48984**	**53**
Sistema optimizado	**6,688**	**149,5**	**35772**	**28**	**36922**	**53**

Como puede observarse, tras la optimización global, Synplify es capaz de reducir notablemente el uso de registros innecesarios en los bordes de los bloques, mejorando así, además, la frecuencia del sistema, mediante la reducción de la ruta crítica.

Gracias a que la tecnología de las dos FPGAs de estudio es común, la síntesis lógica da los mismos resultados absolutos de consumo de recursos, aunque difieren en el relativo al total de recursos disponibles, como se muestra a continuación.

Tabla 16. Ocupación relativa de recursos de la síntesis bottom-up.

Dispositivo	LUTs	DSPs	FlipFlops	BRAMs
XC5VFX130T	39%	9%	41%	18%
XC5VFX70T	72%	22%	75%	36%

Como puede observarse, el consumo de LUTs y FlipFlops se encuentra entre el 70% y el 80% en el caso de la FPGA ensamblada en la placa ML507. Sin embargo, esto concierne únicamente al coprocesador de eventos, el que habrá que integrar en la plataforma diseñada anteriormente. Además, los FlipFlops y las LUTs están ubicados en *Slices*, por lo que es frecuente que, dos FlipFlops no puedan ubicarse en el mismo *Slice*, infrautilizando recursos.

Por otro lado, se puede analizar la distribución de recursos según la partición presentada anteriormente. A continuación se representa este análisis.

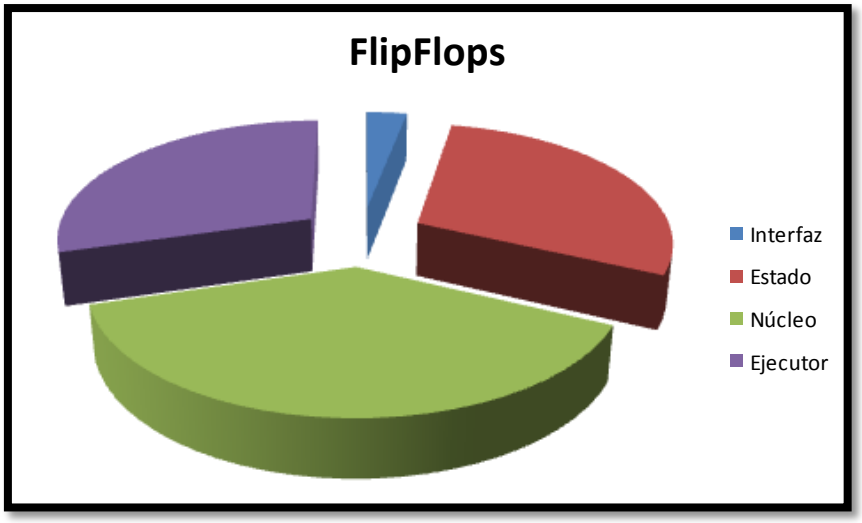


Figura 97. Distribución de FlipFlops.

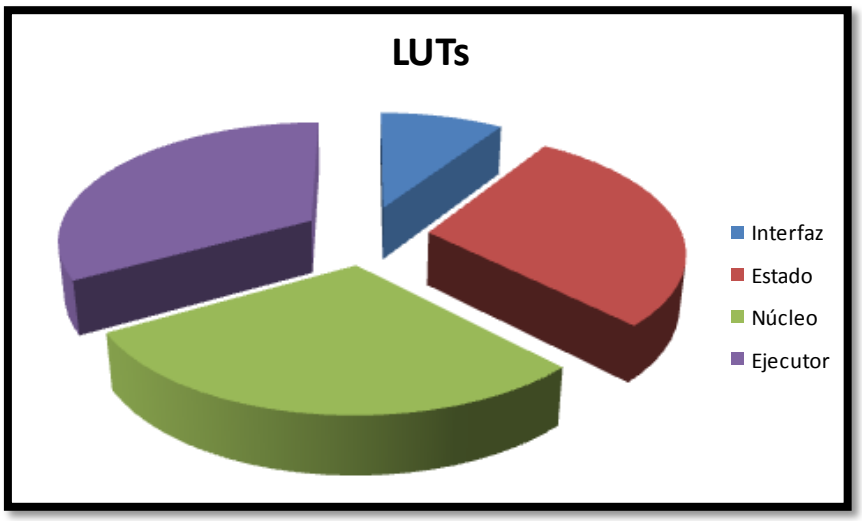


Figura 98. Distribución de LUTs.

Como cabía esperar, la mayor parte de lógica programable genérica, como son las LUTs y los FlipFlops se concentra en los bloques del núcleo de procesamiento y el ejecutor de resultados. Las LUTs y los FlipFlops del bloque de estado son los usados por el bloque responsable de modificar el estado del coprocesador cuando este recibe nuevas estrategias.

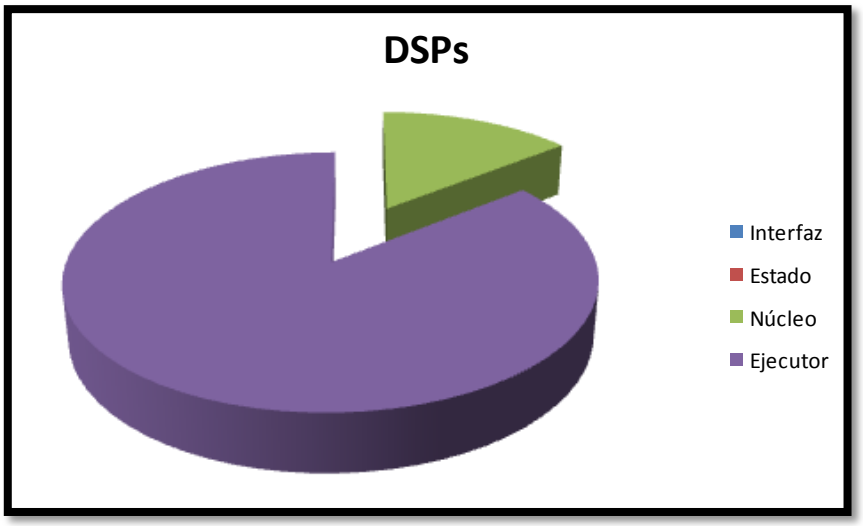


Figura 99. Distribución de DSPs.

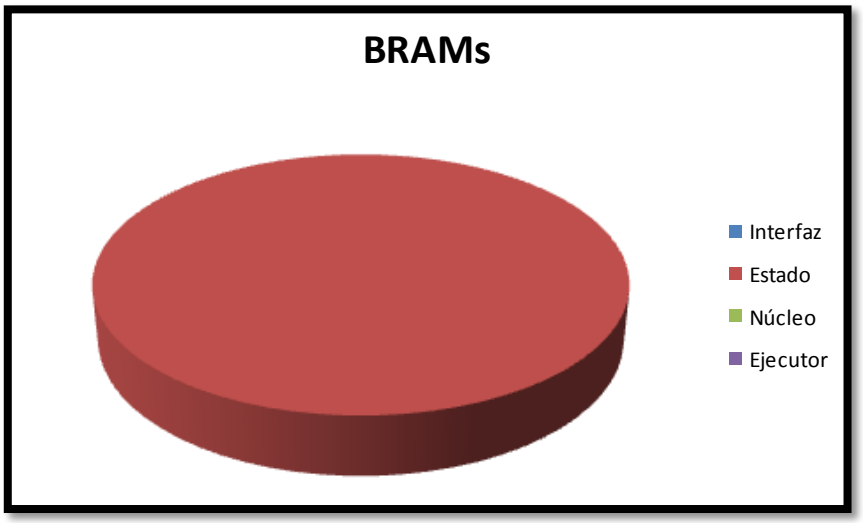


Figura 100. Distribución de BRAMs.

Los DSPs son usados únicamente por el núcleo de procesamiento y el ejecutor de resultados, pues son los que realmente tienen una mayor carga de cómputo. Las BlockRAMs están todas concentradas en el bloque de estado del sistema, que es quien almacena todas las estrategias, acciones e histórico de cotizaciones.

El uso intenso de LUTs del bloque de interfaz se debe a que usa memoria RAM distribuida (la cual, como se adelantó, ocupa LUTs de los SLICEM).

A continuación, se muestra además una comparativa entre la ruta crítica (y por lo tanto la frecuencia máxima) de cada bloque, con el fin de estudiar cual es el que limita el sistema global.

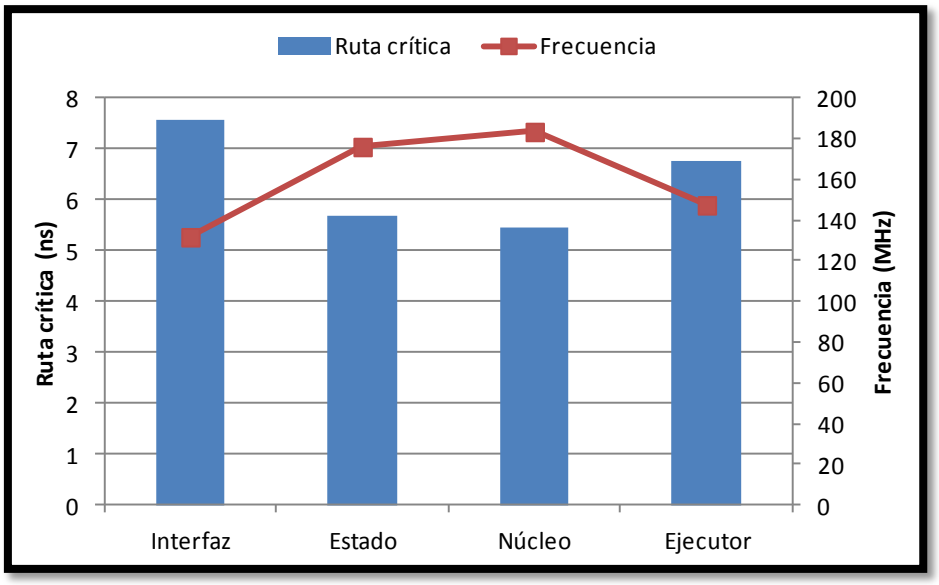


Figura 101. Frecuencia máxima de cada bloque.

En la figura 101 se observa que los dos bloques más lentos se encuentran en la interfaz y el ejecutor de resultados, por lo que serán los objetivos de optimización para futuras versiones. Cabe destacar que estas rutas críticas, y por lo tanto la frecuencia máxima asociada, son fruto de un análisis temporal estático previo al mapeo y ruteado del diseño, y puede verse empeorado una vez se conozcan el número de matrices de interconexión que deban atravesar las señales. Debido a la ya nombrada complejidad del proyecto, y su alta ocupación se producen situaciones en las que las rutas críticas son, en un 80% debido al retardo asociado a la interconexión asociado a la longitud de las conexiones, y que por lo tanto no están contempladas en la anterior gráfica.

7.2 Resultados estrategia *top-down*

A continuación se muestran los resultados de una síntesis con estrategia *top-down*, en la que se muestran las esperadas mejoras en cuestión de consumo de recursos arquitecturales frente a la síntesis *bottom-up*.

Tabla 17. Resultados de síntesis con estrategias bottom-up y top-down.

Estrategia	Ruta crítica (ns)	Frecuencia (MHz)	LUTs	DSPs	FlipFlops	BRAM
bottom-up	6,688	149,5	35772	28	36922	53
top-down	6,888	145,2	30708	28	32291	46

Puede observarse que, menos en el caso de los DSPs, el resto de resultados varían con la estrategia *top-down*, mejorando en el consumo de recursos, pero empeorando en la frecuencia máxima del diseño. A continuación se muestra el valor porcentual de estas diferencias.

Tabla 18. Diferencia de resultados entre estrategia top-down y bottom-up.

	Frecuencia (MHz)	LUTs	DSPs	FlipFlops	BRAM
Diferencia	**-2,88%**	**-14,04%**	**+0%**	**-12,54%**	**-13,21%**

Puede observarse, que la estrategia *top-down* ofrece una mejora significativa en el ahorro de recursos, manteniendo la frecuencia máxima de funcionamiento casi intacta. Estos resultados corroboran el análisis realizado en el apartado 6.3.

7.3 Representación

Los diseños pueden ser representados en Synplify tanto en una vista RTL, donde los elementos básicos son conceptuales como pueden ser RAMs, puertas lógicas o flip-flops, como en una vista tecnológica donde los elementos básicos son las células básicas de la librería con la que se implemente el diseño, como es en este caso, la librería de células de la familia Virtex-5.

En la figura 102 se muestra un ejemplo de una memoria en su vista RTL con la salida registrada, representando así el funcionamiento en modo registro de las BRAM de Virtex-5 ya presentadas en el capítulo 3.

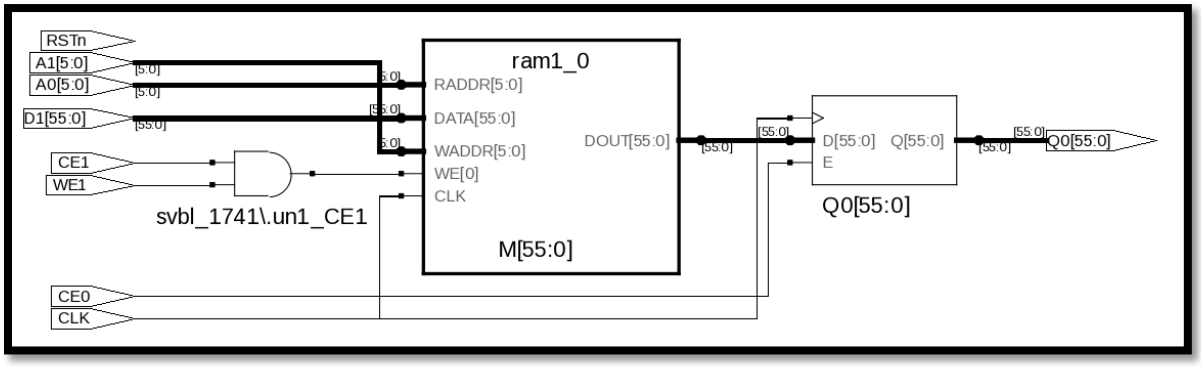


Figura 102. Vista RTL de una memoria interna.

Este diseño RTL puede sintetizarse de diferentes formas, pudiendo implementarse mediante los bloques de memoria RAM de la FPGA (BRAM), como LUTs de los SLICEM o como registros. En la figura 103 se muestra la vista tecnológica del mismo diseño de la figura 102 donde se ha implementado mediante un bloque BRAM de doble puerto, donde le registro de salida ha desaparecido al ser parte del propio elemento BRAM en su modo de funcionamiento registrado.

8 Conclusiones

En este capítulo de ha detallado el proceso de síntesis de un diseño partiendo de una descripción de alto nivel en SystemC, previamente partido mediante varias estrategias, obteniendo una representación de tipo *netlist*.

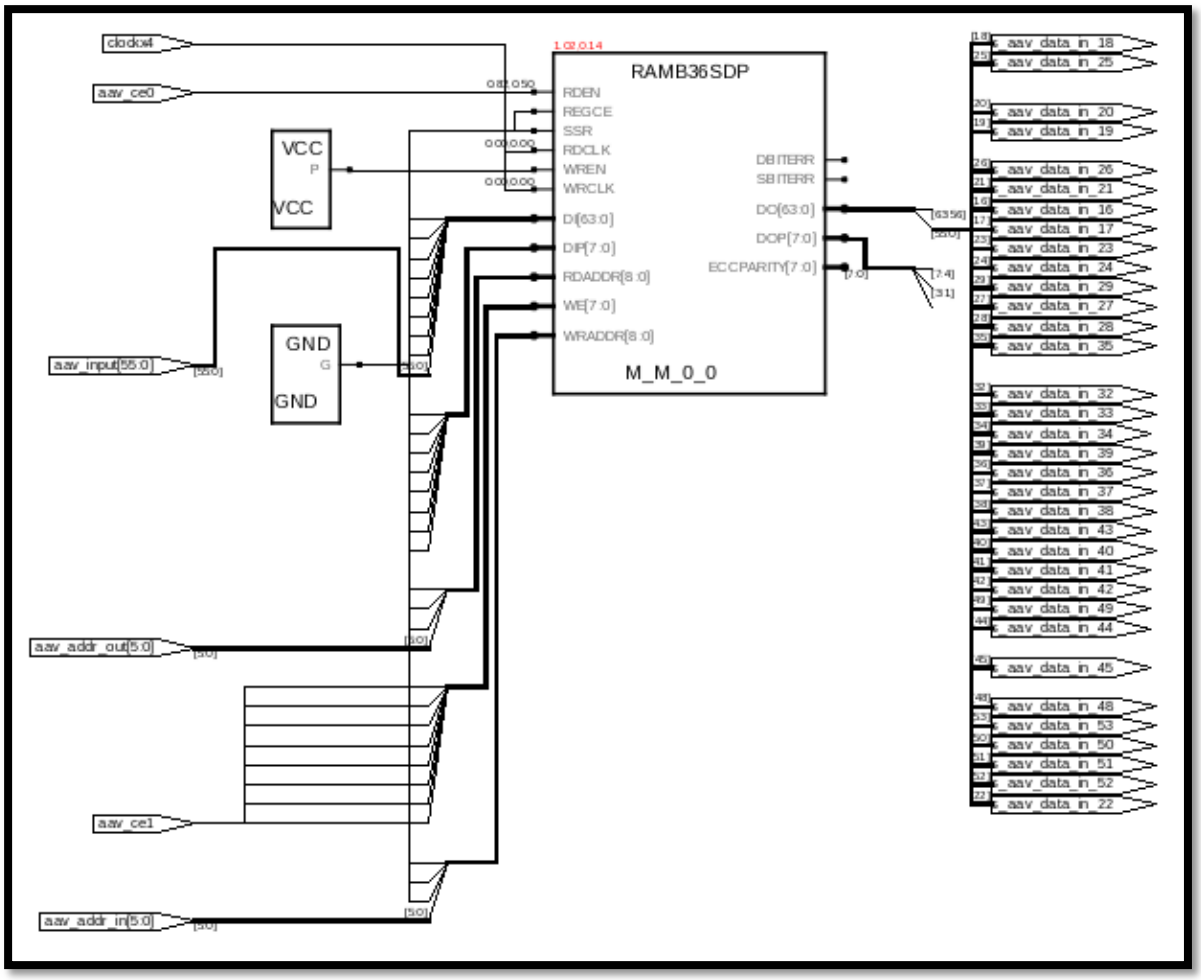


Figura 103. Vista tecnológica de una memoria implementada en BRAM.

Aunque los resultados de ocupación que se han mostrado son dependientes siempre del modelo de referencia sintetizado, el flujo de desarrollo seguido es común en mayor o menor medida, a cualquier diseño hardware que se pueda proponer.

El fin de este desarrollo es el de obtener un diseño sintetizado que integrar a la plataforma como coprocesador, de forma que pueda recibir tramas y enviar acciones o notificaciones al microprocesador del SoC propuesto en el capítulo 4.

Una vez diseñada la plataforma y sintetizado el modelo, en el Capítulo 6 se procederá a la integración de ambos y a su posterior prototipado para así finalizar lo que se podría catalogar como el flujo completo de un sistema electrónico.

Capítulo 6:
Integración y prototipado

1 Introducción

En el presente capítulo se describen los pasos llevados a cabo para integrar el modelo sintetizado del coprocesador de eventos en la plataforma previamente diseñada. Además se verán en este capítulo los pasos de una implementación en FPGA, asociados al mapeo y posicionado de los recursos en el dispositivo programable.

Una vez generado el fichero de configuración, se presenta además el *software* diseñado para la generación de tramas y envío a través de la red.

También se presentan los pasos necesarios para el depurado del sistema en tiempo de ejecución, mediante el uso de analizadores lógicos, y el *software* asociado. En ello se engloba, tanto la interconexión del analizador lógico a los pines que se quieran representar, así como al uso del *software* de captura y representación, donde se configuran además las condiciones de disparo.

2 Importación del coprocesador de eventos

El primer paso para la integración consiste en la importación del modelo sintetizado en la plataforma diseñada, como ya se hizo con el bloque IP de *echo*. Los pasos a seguir son exactamente los mismos, con la salvedad de que Xilinx no permite la importación de un *netlist* directamente como elemento integrante de la plataforma, sino que, en su lugar, debe importarse un *wrapper* escrito en VHDL o Verilog, que internamente instancie el fichero *netlist*.

Durante una de las etapas del asistente de importación de bloques IP de usuario, debe indicarse que existen, además de ficheros HDL, también de tipo *netlist*, para que estos sean importados al proyecto.

El *wrapper* debe realizar las conexiones entre los puertos del mismo, y los puertos internos del componente sintetizado, el cual será instanciado como una caja negra, indicándolo con las directivas que precise. Para el presente PFC el *wrapper* ha sido escrito en VHDL, tal y como se describe a continuación.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity rt_core is
    port(
        Ll_Clk                          : in  std_logic;
        Ll_Rst                          : in  std_logic;
        tx_data                         : in  std_logic_vector(0 to 31);
        tx_rem                          : in  std_logic_vector(0 to 3);
        tx_sof_n                        : in  std_logic;
        tx_eof_n                        : in  std_logic;
        tx_sop_n                        : in  std_logic;
        tx_eop_n                        : in  std_logic;
        tx_src_rdy_n                    : in  std_logic;
        tx_dst_rdy_n                    : out std_logic;
        rx_data                         : out std_logic_vector(0 to 31);
        rx_rem                          : out std_logic_vector(0 to 3);
        rx_sof_n                        : out std_logic;
        rx_eof_n                        : out std_logic;
        rx_sop_n                        : out std_logic;
        rx_eop_n                        : out std_logic;
        rx_src_rdy_n                    : out std_logic;
        rx_dst_rdy_n                    : in  std_logic
    );
end rt_core;

architecture arch of rt_core is

  component procesador_rtl
    port(
        clock                           : in  std_logic;
        rst_n                           : in  std_logic;
        ll_src_rdy_rx_n                 : in  std_logic;
        ll_dst_rdy_rx_n                 : out std_logic;
        ll_sof_rx_n                     : in  std_logic;
```

```
        ll_sop_rx_n             : in  std_logic;
        ll_data_rx              : in  std_logic_vector(31 downto 0);
        ll_rem_rx               : in  std_logic_vector(3 downto 0);
        ll_eop_rx_n             : in  std_logic;
        ll_eof_rx_n             : in  std_logic;
        ll_src_rdy_tx_n         : out std_logic;
        ll_dst_rdy_tx_n         : in  std_logic;
        ll_sof_tx_n             : out std_logic;
        ll_sop_tx_n             : out std_logic;
        ll_data_tx              : out std_logic_vector(31 downto 0);
        ll_rem_tx               : out std_logic_vector(3 downto 0);
        ll_eop_tx_n             : out std_logic;
        ll_eof_tx_n             : out std_logic
    );
  end component;

begin


 procesador_inst : procesador_rtl port map (
              clock  => Ll_Clk,
              rst_n  => Ll_Rst,
     ll_src_rdy_rx_n => tx_src_rdy_n,
     ll_dst_rdy_rx_n => tx_dst_rdy_n,
         ll_sof_rx_n => tx_sof_n,
         ll_sop_rx_n => tx_sop_n,
          ll_data_rx => tx_data,
           ll_rem_rx => tx_rem,
         ll_eop_rx_n => tx_eop_n,
         ll_eof_rx_n => tx_eof_n,
     ll_src_rdy_tx_n => rx_src_rdy_n,
     ll_dst_rdy_tx_n => rx_dst_rdy_n,
         ll_sof_tx_n => rx_sof_n,
         ll_sop_tx_n => rx_sop_n,
          ll_data_tx => rx_data,
           ll_rem_tx => rx_rem,
         ll_eop_tx_n => rx_eop_n,
         ll_eof_tx_n => rx_eof_n);

end arch;
```

Como se puede observar, las señales han sido conectadas de forma cruzada, para usar la nomenclatura del diseño de la plataforma, es decir, *tx* para designar las señales de comunicación asociadas al sentido de la comunicación con origen el DMA y destino el coprocesador de eventos, y *rx* para el sentido contrario.

En el *wrapper* basta con definir la caja negra (*blackbox*) y sus interfaces, y luego realizar la instancia en la que se conectan los puertos.

3 Conexión del analizador lógico

Con el fin de hacer posible la depuración del sistema una vez implementado, se hace uso del analizador lógico ya presentado en el Capítulo 4. En este apartado se detallarán los pasos para su conexionado y configuración, así como particularidades del mismo.

En primer lugar es importante hacer notar que el coprocesador de eventos ha sido instanciado como una caja negra, por lo que es imposible, *a priori*, acceder a las señales internas al mismo. De cara a la herramienta XPS, solo se podrán acceder a los puertos de entrada y salida del mismo.

3.1 Configuración del analizador lógico

La configuración del analizador lógico consiste en suministrar la información acerca del número de señales que se conectarán, así como su ancho. En principio, y puesto que desde XPS solo puede accederse a los puertos de entrada y salida del coprocesador de eventos, se hace uso de 16 señales (máximo número de señales soportadas por un bloque ILA).

Las 16 señales se conectan a cada una de las señales de la interfaz LocalLink del coprocesador, a excepción de las señales de reloj, reset y REM (ya que la longitud de las tramas está controlada internamente en los datos útiles). Quedan dos señales libres que son conectadas a las señales de petición de interrupción IRQ del DMA, una para transmisión y otra para recepción, que permite visualizar los instantes en los que se termina de enviar o recibir una trama.

El ancho de cada señal es de un bit, a excepción de las de datos que son de 32 bits en cada uno de los sentidos de la comunicación. Como ocurre en todas las instancias de bloques, la configuración de parámetros puede realizarse de forma simplificada e intuitiva a través de la interfaz gráfica que proporciona XPS, o manualmente en el fichero MHS. Las líneas que definen estas conexiones se describen a continuación.

```
BEGIN chipscope_ila
 PARAMETER INSTANCE = chipscope_ila_0
 PARAMETER HW_VER = 1.03.a
 PARAMETER C_NUM_DATA_SAMPLES = 1024
 PARAMETER C_TRIG0_TRIGGER_IN_WIDTH = 32
 PARAMETER C_TRIG1_UNITS = 1
 PARAMETER C_TRIG1_TRIGGER_IN_WIDTH = 1
 PARAMETER C_TRIG2_UNITS = 1
 PARAMETER C_TRIG2_TRIGGER_IN_WIDTH = 1
 PARAMETER C_TRIG3_UNITS = 1
 PARAMETER C_TRIG3_TRIGGER_IN_WIDTH = 1
 PARAMETER C_TRIG4_UNITS = 1
 PARAMETER C_TRIG4_TRIGGER_IN_WIDTH = 1
 PARAMETER C_TRIG5_UNITS = 1
 PARAMETER C_TRIG5_TRIGGER_IN_WIDTH = 1
 PARAMETER C_TRIG6_UNITS = 1
 PARAMETER C_TRIG6_TRIGGER_IN_WIDTH = 1
```

```
 PARAMETER C_TRIG7_UNITS = 1
 PARAMETER C_TRIG7_TRIGGER_IN_WIDTH = 32
 PARAMETER C_TRIG8_UNITS = 1
 PARAMETER C_TRIG8_TRIGGER_IN_WIDTH = 1
 PARAMETER C_TRIG9_UNITS = 1
 PARAMETER C_TRIG9_TRIGGER_IN_WIDTH = 1
 PARAMETER C_TRIG10_UNITS = 1
 PARAMETER C_TRIG10_TRIGGER_IN_WIDTH = 1
 PARAMETER C_TRIG11_UNITS = 1
 PARAMETER C_TRIG11_TRIGGER_IN_WIDTH = 1
 PARAMETER C_TRIG12_UNITS = 1
 PARAMETER C_TRIG12_TRIGGER_IN_WIDTH = 1
 PARAMETER C_TRIG13_UNITS = 1
 PARAMETER C_TRIG13_TRIGGER_IN_WIDTH = 1
 PARAMETER C_TRIG14_UNITS = 1
 PARAMETER C_TRIG14_TRIGGER_IN_WIDTH = 1
 PARAMETER C_TRIG15_UNITS = 1
 PARAMETER C_TRIG15_TRIGGER_IN_WIDTH = 1
 PORT CHIPSCOPE_ILA_CONTROL = chipscope_icon_0_control0
 PORT CLK = ll_clk_100MHz
 PORT TRIG0 = rt_core_LLINK0_LL_Tx_Data
 PORT TRIG1 = rt_core_LLINK0_LL_Tx_SOF_n
 PORT TRIG2 = rt_core_LLINK0_LL_Tx_EOF_n
 PORT TRIG3 = rt_core_LLINK0_LL_Tx_SOP_n
 PORT TRIG4 = rt_core_LLINK0_LL_Tx_EOP_n
 PORT TRIG5 = rt_core_LLINK0_LL_Tx_SrcRdy_n
 PORT TRIG6 = rt_core_LLINK0_LL_Tx_DstRdy_n
 PORT TRIG7 = rt_core_LLINK0_LL_Rx_Data
 PORT TRIG8 = rt_core_LLINK0_LL_Rx_SOF_n
 PORT TRIG9 = rt_core_LLINK0_LL_Rx_EOF_n
 PORT TRIG10 = rt_core_LLINK0_LL_Rx_SOP_n
 PORT TRIG11 = rt_core_LLINK0_LL_Rx_EOP_n
 PORT TRIG12 = rt_core_LLINK0_LL_Rx_SrcRdy_n
 PORT TRIG13 = rt_core_LLINK0_LL_Rx_DstRdy_n
 PORT TRIG14 = ppc440_0_DMA0TXIRQ
 PORT TRIG15 = ppc440_0_DMA0RXIRQ
END
```

Como se puede observar, es también necesario indicar el controlador del bloque de análisis lógico (esta información es autocompletada por la herramienta), el reloj de muestreo y el número máximo de muestras a capturar.

El reloj de muestreo debe ser el mismo con el que estén alimentadas las señales a capturar, para este caso, el mismo reloj de la interfaz LocalLink que tiene una frecuencia de 100 MHz. El número de muestras debe ser un compromiso entre facilidad de depurado, y consumo de recursos (y dificultad de conexionado, ya que los bloques BRAM están distribuidos en forma de columna en la FPGA).

Para una correcta elección de la longitud, es importante conocer la naturaleza de los datos a capturar. Si, por ejemplo, se conoce de antemano la longitud de las tramas que se comunicará, por la interfaz LocalLink, como es el caso, puede y debe tomarse la longitud inmediatamente superior en potencias de 2. Para este caso, y basando la decisión en la longitud de las tramas, se ha optado por una longitud de 1024 muestras.

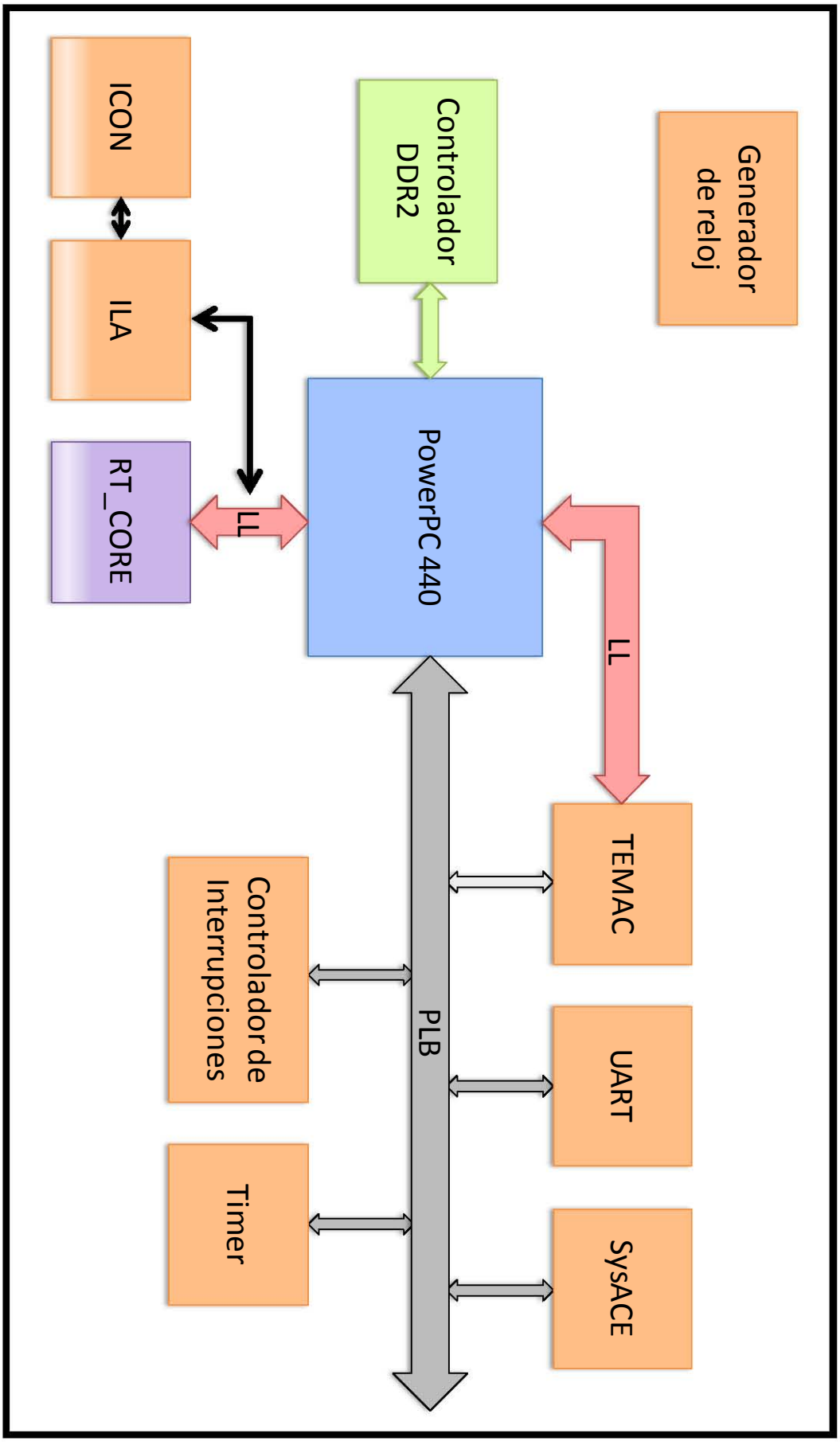


Figura 104. Diagrama de bloques del sistema.

4 Síntesis lógica

Al igual que ocurría en el diseño de la plataforma, el primer paso para la implementación consiste en realizar la síntesis del sistema. En ella se sintetizarán cada uno de los bloques del sistema, generándose un fichero netlist NGC. En el caso del coprocesador de eventos, la síntesis consisten en una simple conversión de EDIF a NGC, ya que se ha instanciado previamente ya sintetizado por Synplify.

Para realizar la síntesis, existe un comando en Xilinx Platform Studio, etiquetado como *Generate Netlist*, que hace uso de las herramientas CORE Generator y XST para generar las descripciones *hardware* de los bloques IP, y para sintetizarlos, respectivamente.

5 Síntesis física

Debido a la complejidad del sistema, realizar la síntesis física en XPS produce errores de posicionado, al indicar la herramienta que el diseño es muy complejo para el dispositivo. En su lugar, se hace uso de la herramienta PlanAhead de Xilinx, la cual proporciona diferentes estrategias, de mapeado, colocado y ruteado.

Los algoritmos de posicionado, parten en general de un colocado aleatorio, para luego comenzar con la optimización, tanto global como local. Esto provoca que dos ejecuciones con las mismas opciones y el mismo diseño, den resultados diferentes. En diseños críticos en consumo de recursos, esto puede suponer que dos ejecuciones, bajo las mismas condiciones, den resultados totalmente diferentes, pudiendo generar una un diseño correctamente colocado y la otra un mensaje de error indicando la imposibilidad de introducir el diseño en la FPGA.

PlanAhead permite importar varios ficheros de tipo *netlist* (EDIF o NGC) así como las restricciones de usuario, para crear un proyecto de implementación (síntesis física). Para el caso que ocupa a este PFC, los ficheros a importar son los generados por la síntesis lógica de XPS. Los ficheros asociados, se encontrarán en el directorio *implementation* dentro del directorio de proyecto de XPS, donde además se encontrará el fichero de restricciones de usuario (UCF).

5.1 Creación del proyecto

El primer paso para la implementación física con PlanAhead, es crear el proyecto para dicha herramienta. Para hacerlo, PlanAhead dispone de un asistente que guía al diseñador a través de varios pasos en el que se detallarán los parámetros del mismo.

Tras indicar el directorio base del proyecto de síntesis física, el siguiente paso es indicar el tipo de proyecto, o lo que es lo mismo, la etapa de diseño de la que se parte. Para el caso que se plantea en este proyecto, se selecciona la opción "*Specify synthesized (EDIF or NGC) netlist*", ya que los ficheros fuente de los que se parte están sintetizados pero no implementados. Se pueden

crear proyectos de PlanAhead a partir de ficheros fuentes HDL, y realizar la síntesis dentro de esta misma herramienta, o importar diseños ya implementados para realizar análisis y verificación.

El siguiente paso consiste en importar los ficheros fuente, así como indicar cuál es el fichero *top* del diseño. Como ya se adelantó, puesto que se quiere realizar la síntesis física del diseño sintetizado previamente por XPS, los ficheros fuentes se encontrará en el directorio *implementation* de dicho proyecto.

```
xps_project_directory/implementation/
```

Puesto que los ficheros ya están sintetizados, la información del dispositivo de prototipado es extraído de estos por la herramienta, aunque se pide la confirmación del diseñador. En la figura 105 se muestra cómo en la última etapa de creación del proyecto, ya la herramienta ha reconocido el dispositivo programable.

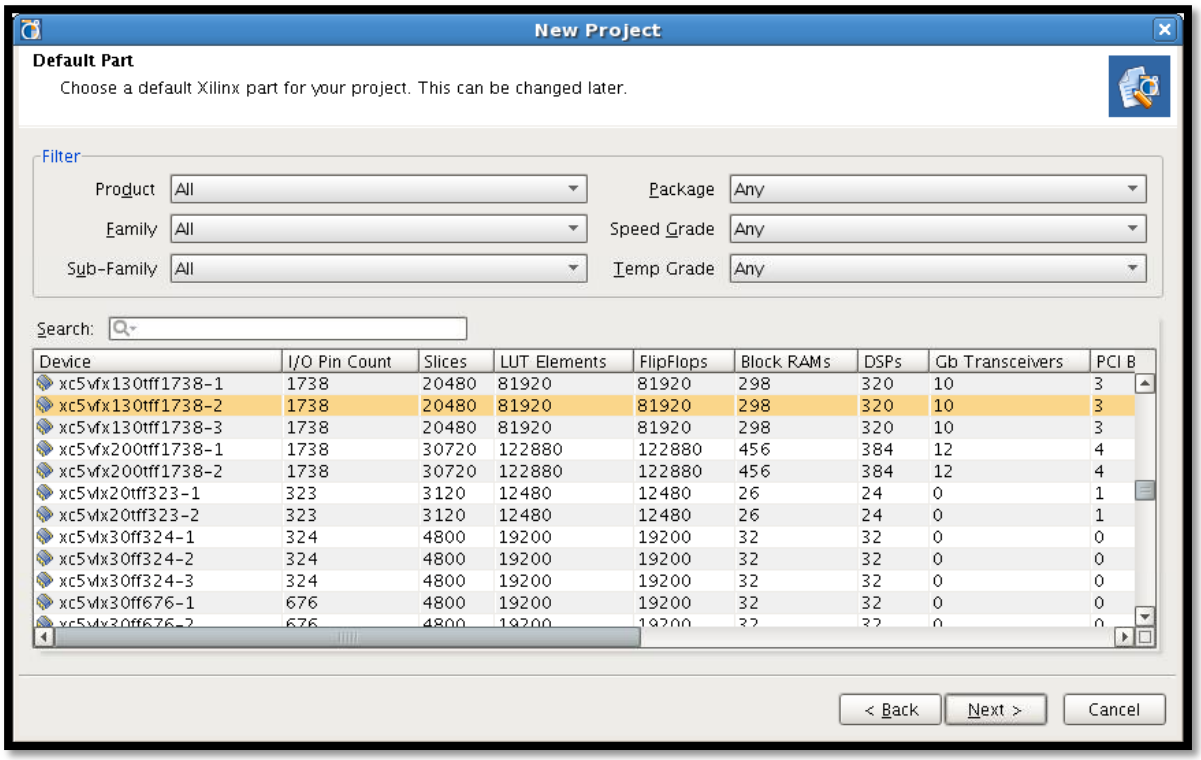


Figura 105. Dispositivo FPGA auto-identificado por la herramienta PlanAhead.

5.2 Generación de ejecuciones de implementación

Debido al gran consumo de recursos de la FPGA de este diseño y a la aleatoriedad en el comienzo de los algoritmos de posicionado, se ha recurrido a la utilidad de PlanAhead para lanzar varias implementaciones (en paralelo, o secuencialmente) con diferentes estrategias, para luego optar por aquella con un mejor análisis temporal.

Para poder trabajar con ejecuciones en paralelo y así reducir los tiempos de síntesis físicas, se ha optado por hacer uso de los servidores de cómputo disponibles. PlanAhead permite configurar

una lista de servidores, así como el número de trabajos en paralelo para cada uno. El primer paso consiste en invocar el asistente de lanzamiento de múltiples implementaciones, como se muestra en la figura 106.

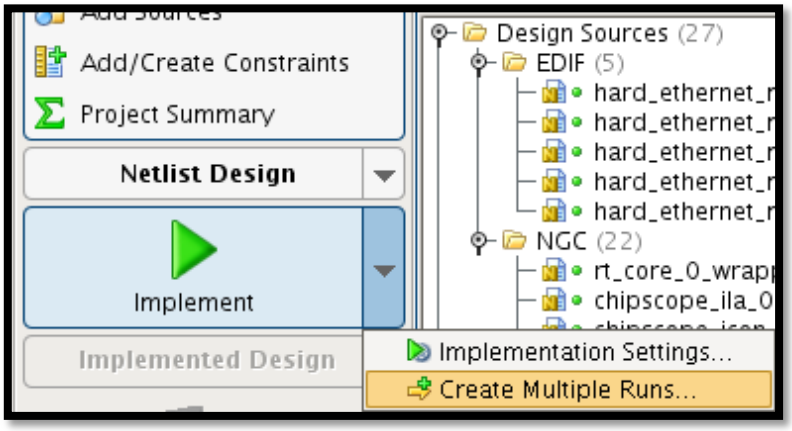


Figura 106. Lanzamiento de múltiples implementaciones.

Tras elegir el grupo de ficheros fuentes y de restricciones, se seleccionan el número de implementaciones, así como la estrategia de cada una de ellas. Por ejemplo, para este caso, se seleccionan nueve implementaciones, una por cada estrategia propuesta por la herramienta (figura 107).

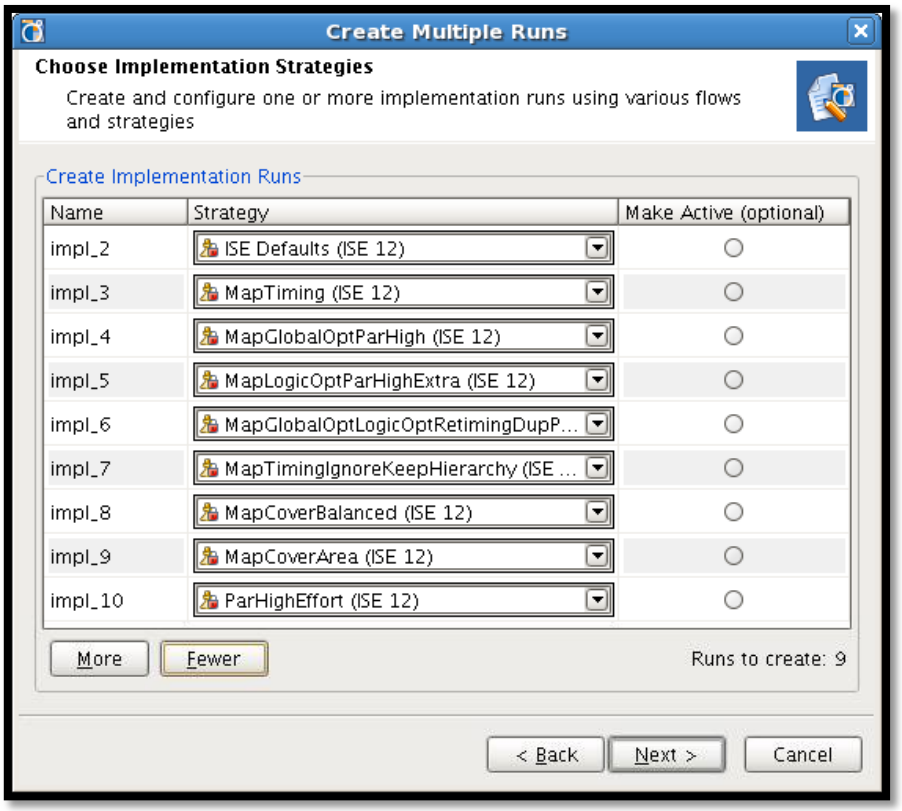


Figura 107. Estrategias de las implementaciones.

A continuación se decide si las implementaciones se realizan en local (en cuyo caso se seleccionan el número de trabajos paralelos) o si se lanzan en servidores remotos. En dicho caso, se puede configurar la lista de servidores, con su número de trabajos asociados, y una dirección email donde enviar un correo electrónico cada vez que una implementación termine (figura 108).

Las estrategias propuestas por PlanAhead son, en general, bastante diversas, con y sin optimización global, con optimización en área o en frecuencia, etc. Sin embargo, y debido a la aleatoriedad de la colocación inicial, no se ha observado una correspondencia directa, entre la estrategia y el éxito del algoritmo. Así, ha habido estrategias de optimización en frecuencia que consiguen un resultado correcto, y estrategias de optimización en área que terminan sin conseguir llegar a una solución y producen un error de "diseño demasiado complejo para la FPGA".

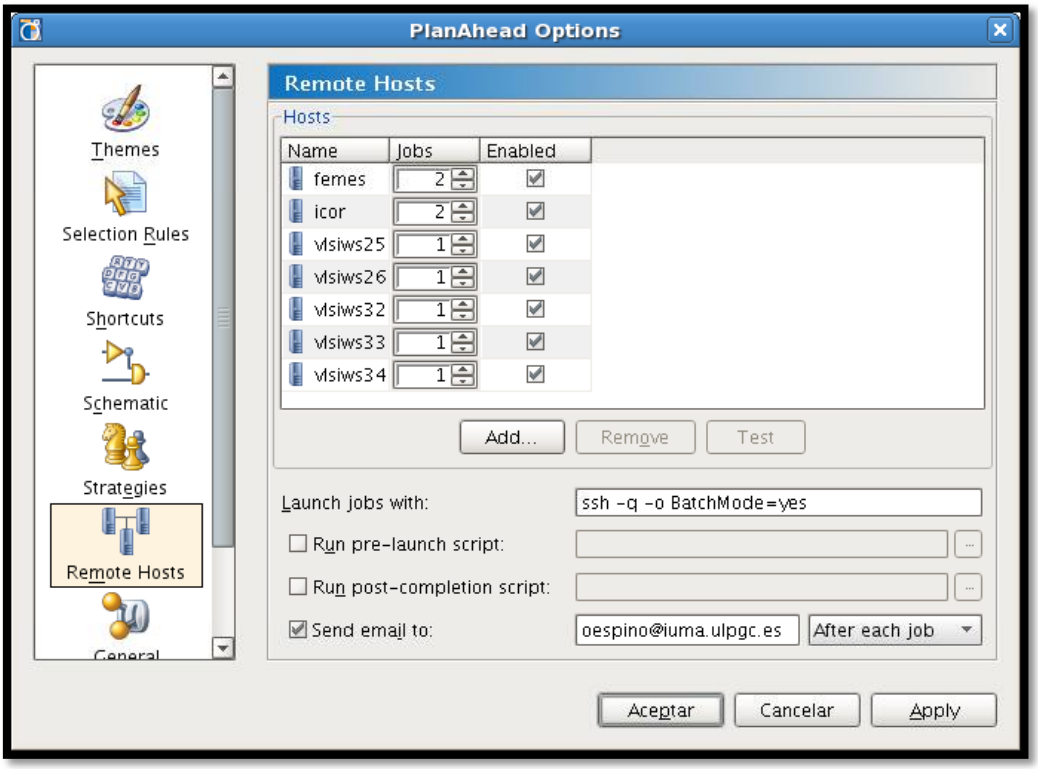


Figura 108. Servidores remotos.

En la figura 109, se observa como de nueve implementaciones, han conseguido finalizar cuatro, una sigue en el proceso de ruteado y otras cuatro han fallado al no conseguir realizar el posicionamiento de los recursos en el dispositivo, lo cual se representa por un "*MAP ERROR*".

En esa misma imagen, se muestra además los tiempos de síntesis física de cada estrategia, que ronda entre los 30 y los 120 minutos. Este punto resalta más si cabe, la importancia de realizar las ejecuciones en paralelo.

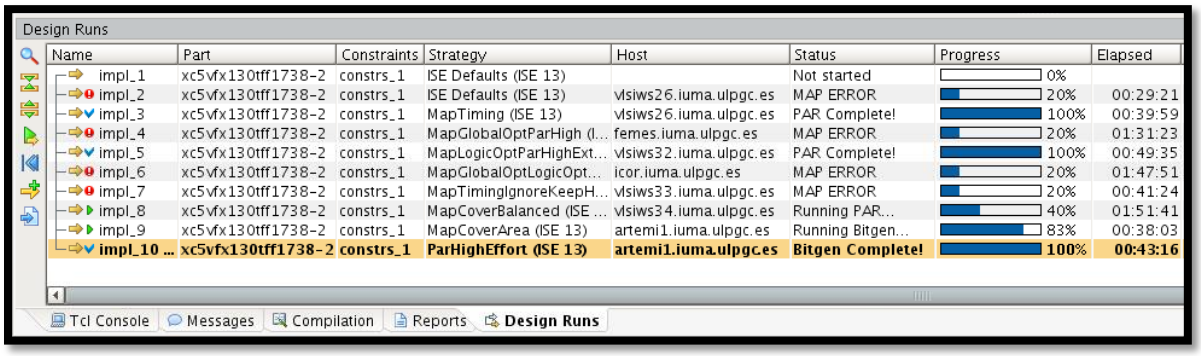


Figura 109. Resultado de las implementaciones paralelas.

5.3 Resultados de la síntesis física

Una vez realizada por completo la síntesis física, el diseño está listo para su volcado en la placa, y también para realizar análisis temporales más precisos, en el que se midan los retardos teniendo en cuenta la colocación de los recursos, así como la ruta que deberán seguir las distintas conexiones del sistema.

En este apartado se describen tres aspectos fundamentales de la solución proporcionada por la herramienta que son: el porcentaje de utilización de recursos de la FPGA, la colocación de dichos recursos en el mapa del dispositivo, y un análisis temporal en el que se muestran las rutas críticas, así como sus causas.

5.3.1 Recursos

Una vez realizada la síntesis física, se obtiene, además el número de LUTs y Registros utilizados, el número de *Slices* utilizados, algo que carece sentido hasta que los recursos son mapeados.

Tabla 19. Utilización de recursos.

Recursos	BRAMs	DSPs	LUTs	FlipFlops	Slices
Utilizados	62	28	38777	52982	**18562**
Totales	298	320	81920	81920	**20480**
Porcentaje	21%	9%	47%	65%	**91%**

Para hacer un análisis del área ocupada por el diseño así como de su complejidad, se debe observar el número de *Slices* utilizados, ya que el *Slice* es la unidad básica de programabilidad de las FPGAs de la familia Virtex-5 de Xilinx.

Para el modelo de referencia con el que se ha trabajado, y las opciones de síntesis propuestas, la ocupación de lógica programable se situa en un 91%. Es importante destacar que, alcanzar una ocupación del 100% es prácticamente inviable, debido a los recursos de conectividad del dispositivo. Alcanzar una ocupación muy alta depende, en parte, de lo mucho que se pueda adaptar el diseño a la arquitectura de la FPGA.

Centrando el estudio de ocupación en este hecho, otro punto de interés es la diferencia de ocupación entre *LUTs* y *FlipFlops*, y su equivalente en *Slices*. Dada la arquitectura de Virtex-5, cada *Slice* está formado por cuatro *LUTs* y cuatro *FlipFlops*, por lo que, podría considerarse que, dado un número de *LUTs* y *FlipFlops* usados, su equivalente en *Slices* se puede calcular como:

$$Slices = \frac{\max(LUTs, FFs)}{4}$$

Sin embargo, este cálculo es demasiado optimista, ya que está obviando las características de dependencia entre recursos debidos al diseño. La arquitectura interna de los *Slices* permite una gran variedad de soluciones de interconexión, con el fin de hacer lo más independientes posible los recursos internos a esta, sin embargo, nunca se puede llegar a una independencia total.

En un caso ideal, donde la arquitectura de la FPGA coincidiese perfectamente con las necesidades del diseño, el porcentaje de uso de *Slices* coincidiría con el máximo porcentaje de uso de *LUTs* y *FlipFlops*. Para el caso de la Tabla 19 el porcentaje de *Slices* coincidiría con el de *FlipFlops*, es decir, un 65%, permitiendo colocar todas las *LUTs* del diseño en los mismos *Slices* ya consumidos para *FlipFlops*.

Puede calcularse, a partir de los datos obtenidos, el número de *LUTs* y *FlipFlops* no usados, de *Slices* en uso, es decir, aquellos recursos que, se están contabilizando como usados, pues el *Slice* al que pertenece está en uso, pero que sin embargo, no está desempeñando ninguna función para el diseño que se presenta.

Tabla 20. Rentabilidad de uso de recursos.

Recurso	Slices	LUTs y FFs contabilizados	LUTs en uso	LUTs sin utilidad	FFs en uso	FFs sin utilidad
Cantidad	18562	74248	38777	35471	52982	21266

Como se deduce de la tabla 20, casi la mitad de las *LUTs* contenidas en los *Slices* del diseño se encuentran en desuso, lo mismo que ocurre con casi un tercio de los *FlipFlops*.

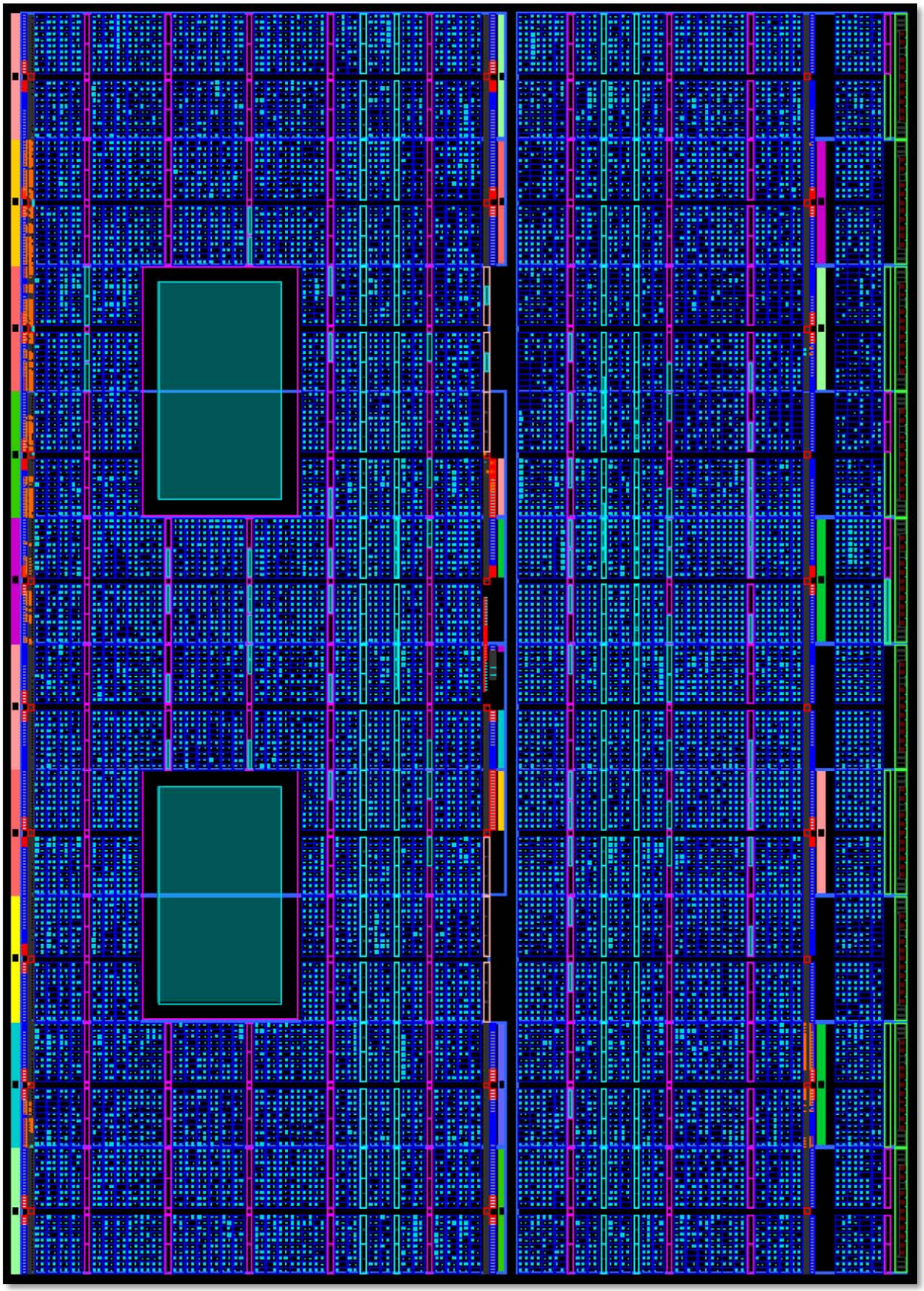


Figura 110. Recursos utilizados en la FPGA.

En la figura 110 se muestra, sobre un esquema que representa la FPGA y sus recursos en sus correspondientes localizaciones, aquellos recursos que están siendo usados. Cada cuadrado en la imagen representa un CLB (a su vez formado por dos Slices), los cuales están resaltados en color si están siendo usados, y en negro aquellos que están libres.

5.3.2 Colocación

Otro punto importante de la síntesis física, es la colocación de los recursos usados por el diseño sobre el mapa de la FPGA. Por la propia naturaleza de estos dispositivos, el diseño es el que debe adaptarse a la colocación previa de los recursos (al contrario de lo que ocurre en los diseños ASIC), lo cual provoca restricciones, sobre todo en aquellos recursos que no son tan abundantes. Mientras que los *Slices* de tipo M y L (los que contienen FlipFlops y LUTs) se distribuyen uniformemente en el dispositivo, los bloques de BRAM, DSPs, los microprocesadores empotrados, EMACs, etc., que son recursos más limitados, solo se encuentran en ciertas posiciones, lo que implica que aquellos módulos del diseño que tengan conexión directa con estos recursos, deben colocarse cercanos a ellos.

En la figura 111 se muestran resaltados con colores los cuatro bloques de la plataforma que más ocupación tienen. El resto no se han representado ya que quedan escondidos por el área de los cuatro mencionados. En dicha figura, puede verse que de los dos PowerPC 440, el que está en uso es el superior, quedando el segundo libre.

Otra imagen de interés es la de la colocación de los cuatro bloques internos en los que se divide el coprocesador de eventos sintetizado e integrado en la plataforma. Esta imagen se representa en la figura 112. En ella se observa que, la mayoría de bloques BRAM (columnas de borde magenta), están de color rojo, indicando que pertenecen al bloque de estado del coprocesador de eventos. Por otro lado, en amarillo se ha resaltado el bloque de interfaz, el cual es el más cercano al PowerPC, ya que es el que tiene conexión directa con él. El resto de lógica se divide entre núcleo de procesamiento, en azul cian, y el ejecutor de resultados en verde.

5.3.3 Análisis temporal

El último punto importante a analizar una vez el diseño se encuentra implementado, es el de análisis temporal. Se pretende con ello, medir el *slack* de todas las conexiones del diseño, prestando atención a aquellas con un valor negativo, en cuyo caso habría que hacer modificaciones en el diseño hasta cumplir las restricciones temporales, o modificar dichas restricciones si no hubiese más remedio.

Para este análisis, Xilinx dispone de la herramienta TRCE (Timing Reporter and Circuit Evaluator), que puede ser invocado desde PlanAhead una vez se ha implementado el diseño. La herramienta genera un análisis temporal estático basado en las restricciones temporales del diseño.

Una sencilla forma de representar el análisis es mediante un histograma, en donde cada barra representa un intervalo de valores de *slack* de las señales del diseño. Así el diseñador dispone, de forma representativa, de la información temporal de todas las interconexiones que conforman su diseño.

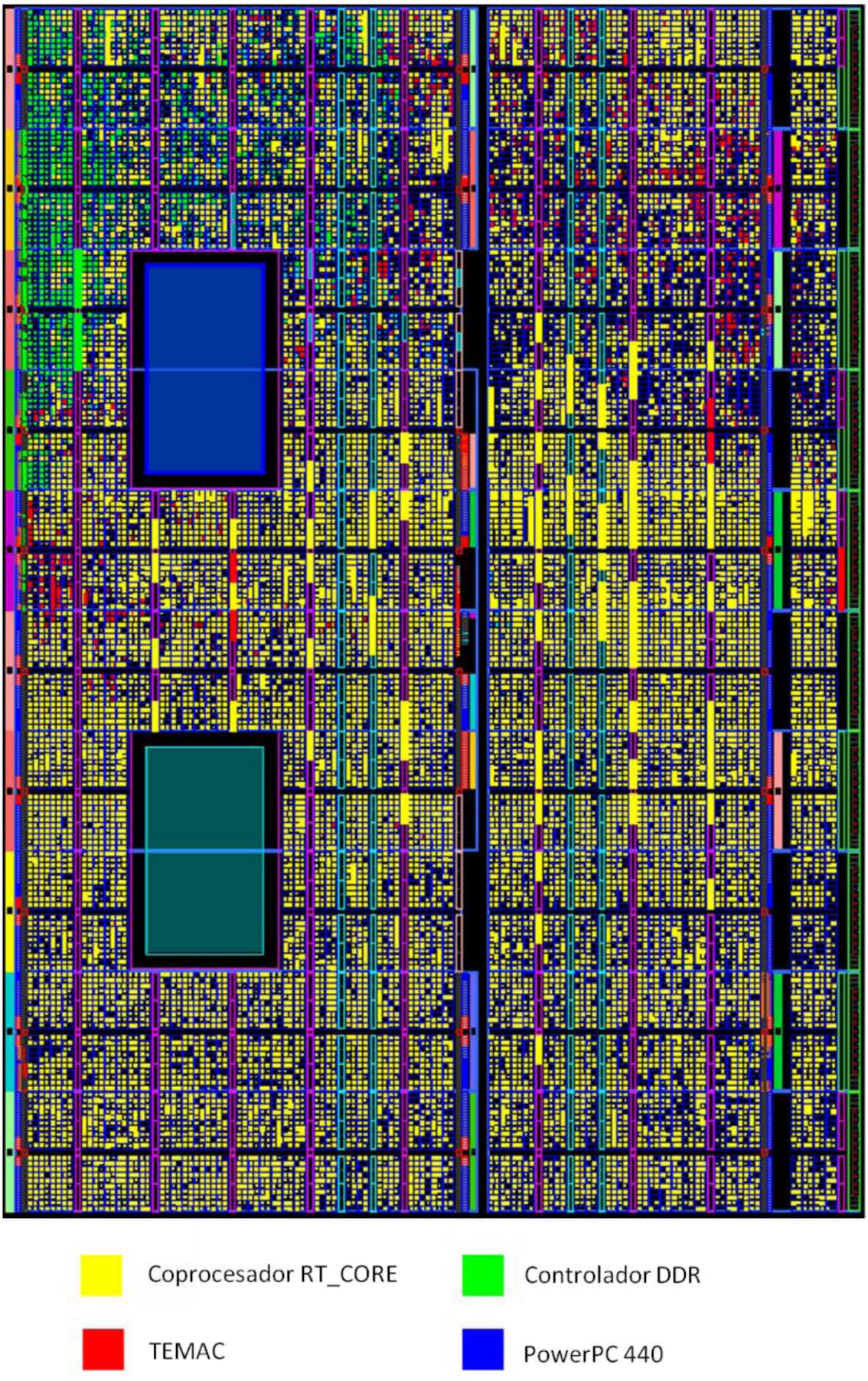


Figura 111. Colocación a nivel de plataforma.

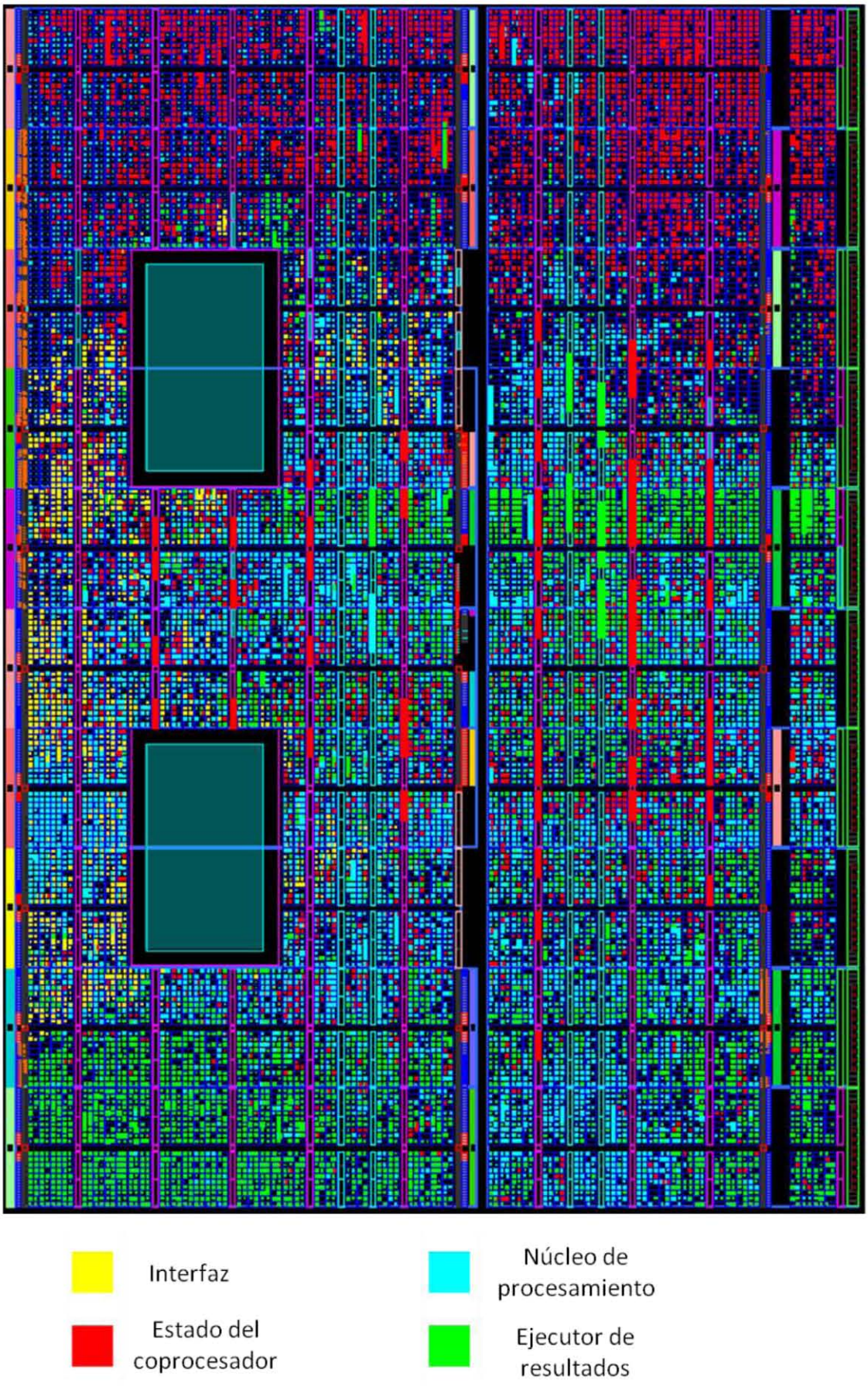


Figura 112. Colocación interna al coprocesador de eventos.

Durante la fase de implementación, PlanAhead presenta los resultados temporales mediante un *Timing Score*, un valor numérico que indica la suma en pico segundos, del valor absoluto de todos los *slacks* negativos del diseño.

En un diseño completamente ruteado y que cumple todas las restricciones temporales tendrá un valor de *Timing Score* nulo, o simplemente no aparecerá tal valor. En la figura 113 se muestra como representa PlanAhead un sistema con señales con *slack* negativo.

FMax (MHz)	Timing Score	Unrouted	Description
			ISE Defaults, including packing registers in IOs off
91.988	3164771	0	ISE Defaults, including packing registers in IOs off

Figura 113. Resultado de *Timing Score*.

Una vez conocidas las dificultades de la herramienta para cumplir las restricciones temporales, es de interés, conocer si se debe a unas pocas señales que tienen demasiada latencia, o si son muchas señales que se encuentran a poco de cumplir las restricciones. Para conocer este dato, se lanza el TRCE y se estudia el histograma generado. Un ejemplo con señales que no cumplen restricciones se muestra en la figura 114.

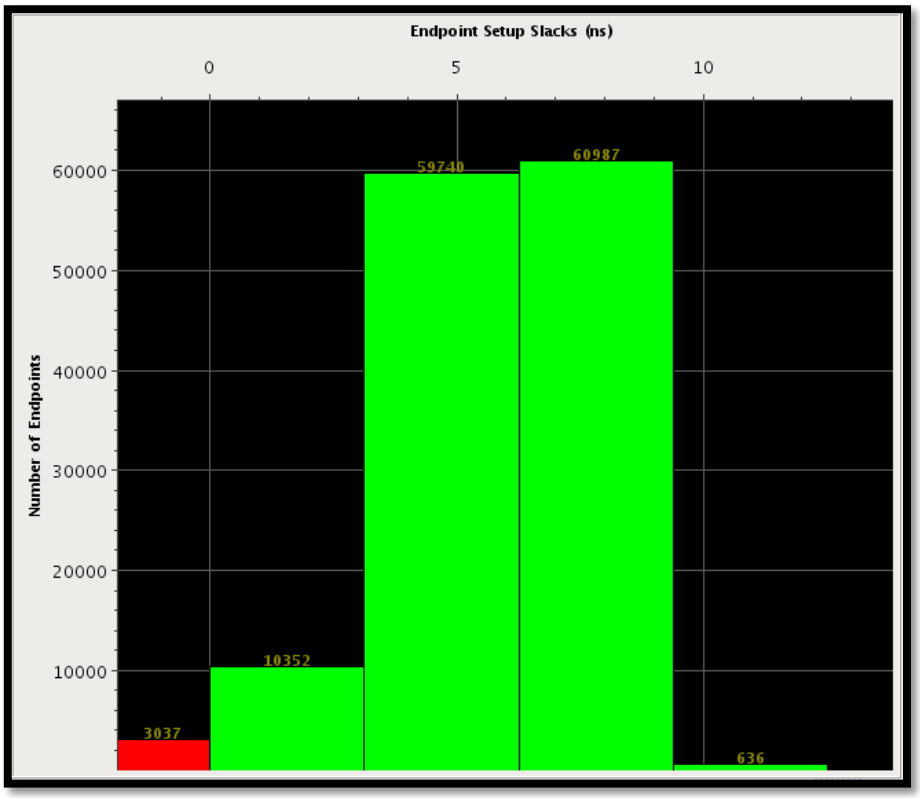


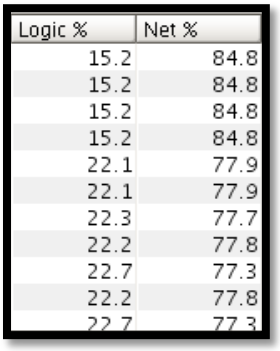
Figura 114. Histograma de *slacks*.

Una vez se ha realizado el análisis temporal de forma general al diseño es importante identificar las rutas donde se están produciendo los errores temporales, para poder aplicar las medidas oportunas. En el informe del análisis temporal, se detallan e identifican las señales que incumplen las restricciones, además de cierta información acerca de la causa de su retardo.

El retardo de una conexión entre elementos secuenciales se divide en los retardos introducidos por los elementos combinacionales que forman la ruta, y por las latencias debidas a la interconexión como son las matrices de interconexión.

En general, se toma como buen resultado de posicionado un 40% de retardo debido a la propia línea de interconexión, dejando un 60% debido a la lógica. Valores superiores implican diseños muy congestionados.

En el modelo de referencia usado en este PFC, existe una alta dependencia entre la mayoría de bloques internos al coprocesador de eventos. Esto provoca que, necesariamente, se produzcan conexiones entre recursos muy alejados en su colocación física de la FPGA. Esto provoca que, la frecuencia máxima del procesador venga, mayoritariamente restringida por los retardos de las conexiones, como se muestra en la figura 115.



Logic %	Net %
15.2	84.8
15.2	84.8
15.2	84.8
15.2	84.8
22.1	77.9
22.1	77.9
22.3	77.7
22.2	77.8
22.7	77.3
22.2	77.8
22.7	77.3

Figura 115. Distribución de retardos en las señales.

6 Generador de eventos

Para poder validar el sistema sobre la placa de prototipado, es necesario disponer de un generador de eventos, que suministre las tramas a la plataforma, y recoja las tramas de salida y las disponga en ficheros o simplemente las muestre por pantalla para un fácil análisis.

En el mejor de los casos, se parte de que las tramas de entrada están almacenadas, de forma independiente, en ficheros binarios. En este caso, el generador de tramas suministrará los eventos mediante un simple algoritmo secuencial que leerá del fichero fuente la ristra de bytes de cada evento, y los enviará por el socket previamente conectado a la plataforma de procesador.

Otro caso, bastante típico en ciertos procesadores de eventos, es que los eventos de entrada no estén previamente almacenados, sino que se tengan que extraer de la red mediante un analizador de paquetes (en la literatura relacionada, *sniffer*).

Para poder realizar tareas de obtención de tramas de red, existe la librería para C *libpcap*, destinada a dos funciones principales: acceder al medio y capturar paquetes, introduciendo una capa de *sniffer* en la aplicación; y ser capaz de extraer de dichos paquetes las distintas cabeceras de las capas OSI, así como la carga útil.

Esta librería es de especial interés para procesadores de eventos destinados a la seguridad de una red, donde los eventos no son tramas destinadas directamente al procesador en sí, sino que se tratan de mensajes entre elementos que conforman la red, y el procesador de eventos debe captarlos y detectar posibles amenazas.

En el desarrollo del presente proyecto se dio un caso que une las dos situaciones anteriormente descritas. Las tramas de ejemplo suministradas por el propietario de la aplicación de referencia se encuentran en formato PCAP, es decir, ficheros capturados directamente de la red, que incluyen las cabeceras de las capas OSI además de la carga útil.

Para simplificar las tareas de extracción de tramas útiles, tras un filtrado para eliminar todas las tramas carentes de datos (tramas ARP, DHCP, ICMP, etc.) se diseñó una sencilla aplicación de extracción de carga útil. Evidentemente, para poder acceder a los datos, se hizo uso de la librería *libpcap*, que permite emular la captura de datos de la red, mediante la lectura de un fichero PCAP. Cada carga útil extraída se almacenó en un fichero independiente en formato binario, para su posterior uso.

Tras dicha adaptación, los eventos se encuentran en ficheros independientes, que fueron usados para las verificaciones a nivel ESL en SystemC, a nivel RTL (ambos haciendo uso del mismo *testbench* que leían los datos de dichos ficheros) así como para la validación sobre la placa de prototipado, pues el generador de eventos lee directamente los ficheros binarios y los envía tal cual a través de la red haciendo uso de un socket, que se conecta a la dirección IP de la placa.

7 Depuración física

Una vez el diseño está completamente implementado, y se dispone de un generador de eventos que estimule el sistema, el siguiente paso es verificar la funcionalidad del prototipo. Para ello, se deberá programar la FPGA mediante el fichero binario de configuración, generado para este caso desde PlanAhead. Hecho esto, se podrá descargar el fichero ejecutable, de la aplicación empotrada, previamente compilada, y arrancar el sistema.

La depuración de un sistema *software* es un proceso muy documentado y sin dificultades tecnológicas, pues consiste en ejecutar paso a paso las instrucciones del código desarrollado, para lo cual los procesadores del mercado dan soporte.

Sin embargo, la depuración de un sistema digital durante su funcionamiento en tiempo real, requiere de tecnología adicional de captura de datos en los puntos de medida. Los dispositivos diseñados para este fin se conocen como analizadores lógicos, como son el caso de los bloques ILA instanciados en la plataforma.

Partiendo de que la aplicación empotrada ha sido previamente depurada, como ya se adelantaba en el capítulo 4, el primer punto de prueba debe ser la interfaz LocalLink entre el DMA y el coprocesador de eventos, para comprobar que las señales de control de la comunicación, así como los datos enviados, son los correctos, es decir, coinciden con las simulaciones previamente realizadas a los distintos niveles de abstracción.

Previa conexión de las señales de disparo del analizador lógico integrado, proceso explicado previamente en "*3. Conexión del analizador lógico"* se debe usar un software específico que, a través del cable de programación de la FPGA y de su interfaz JTAG, controla el analizador. La herramienta que se presenta es *ChipScope Analyzer*.

El proceso de medida a través de un analizador lógico, consiste en la definición de unas condiciones de disparo, mediante sentencias lógicas de las señales de disparo y valores asignados por el diseñador. Cuando se cumpla la condición de disparo, el analizador comenzará a almacenar los valores, de forma síncrona al reloj con el que se alimenta (en analizadores lógicos de alta gama es común usar relojes mucho más altos que los de las señales a capturar, con el fin de sobremuestrar la señal y capturar los cambios de estado de la misma, durante el tiempo de estabilización de la señal).

ChipScope permite dos modos de captura, el simple, y el de ventana. En el modo simple, cuando se cumple la condición, captura el número máximo de muestras que le permite el tamaño de su memoria interna (para este proyecto se configuró con un tamaño de 1024 muestras). En el modo ventana, se puede configurar un número de ventanas de captura, pero con un número de muestras inferior, dependiendo del número de ventanas seleccionadas. El tamaño de las ventanas de captura siempre será un múltiplo de dos (tanto para tres ventanas, como para cuatro ventanas, la longitud de la trama capturada será de 256 muestras).

Otro parámetro importante en la captura, es el desplazamiento de la muestra de disparo. Este valor, indica que posición ocupará la muestra en la que se produzco el disparo, respecto a la primera muestra capturada. Por ejemplo, puede darse el caso que, cuando se produce un evento en una señal, se quiera capturar los datos a partir de la muestra 100 después de dicho evento. En ese caso, el valor del desplazamiento (*Position*, en ChipScope) será de -100. Pueden además, indicarse desplazamientos positivos, si se quieren capturar las 100 muestras anteriores a que se produzca el evento (para ello, el analizador lógico se encuentra en un estado de escritura continua de los datos, siendo el evento el que indica que no los sobreescriba con los que se produzcan a partir de él).

En la figura 117 se muestra una captura de una transmisión a través de la interfaz LocalLink, en la que se envía una trama al coprocesador de eventos, mediante el bloque DMA. Los parámetros de captura se han configurado de tal forma que se almacenan las 500 muestras anteriores a la condición de disparo, por lo que el valor de *Position* es de 500. La condición de disparo será la activación de la señal de interrupción de transmisión, con lo que, en las 500 muestras anteriores se encontrarán las muestras de la transmisión. En la Figura 116 se observa la configuración de las condiciones de de disparo. En la parte superior, se indican los valores lógicos asociados a cada señal (1, 0 ó X). En la inferior, se enumeran las condiciones de disparo. En este caso, a la señal DMATXIRQ (M14), se le asocia el valor lógico 1 (ya que es una señal activa a nivel alto), y en la condición de disparo se indica M14 (se pueden indicar condiciones compuestas como M14 && M13).

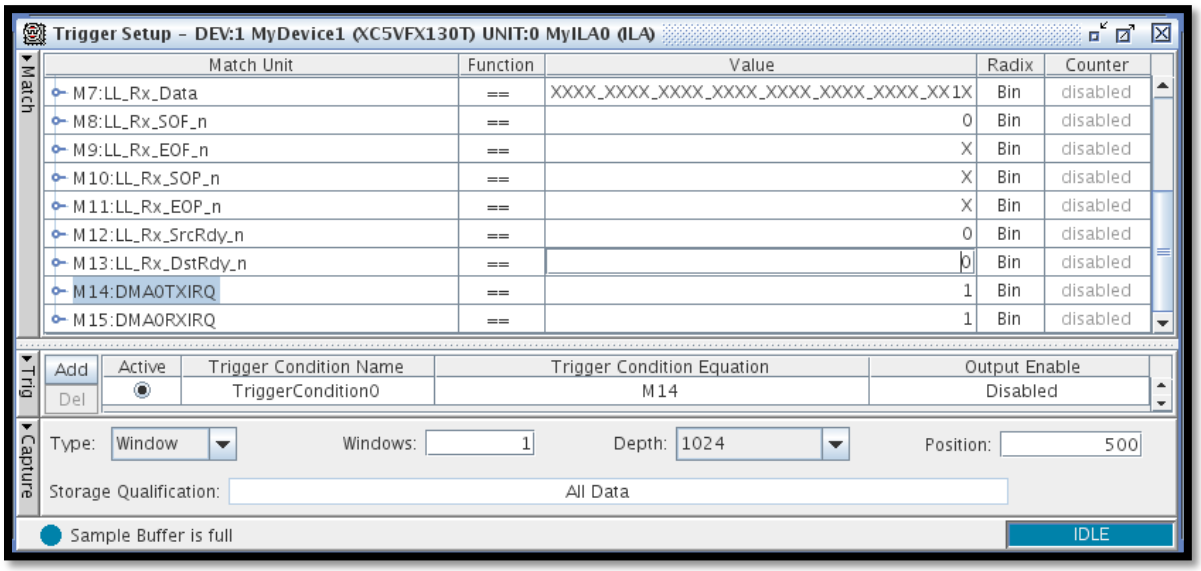


Figura 116. Configuración de las condiciones de disparo.

En la figura 117 se representa un ejemplo de captura de datos. El comienzo de la trama viene definido por la activación, a nivel bajo, de la señal *Start_Of_Frame* de LocalLink (SOF). Como puede observarse, tras las primeras ocho palabras, destinadas a la cabecera de la trama, el DMA interrumpe la transmisión (lo cual puede corroborarse prestando atención a la señal de *Source_Ready* de LocalLink (SrcRdy). Una vez reanuda la transmisión, comienza la carga útil activando la señal destinada a ello.

Para señalar el fin de la trama, el protocolo LocalLink usado por los DMA del bloque de procesador marca activar la señal de fin de carga útil (*End_Of_Payload*, EOP) en el mismo instante en el que se envía la palabra, para, en el siguiente ciclo, activar la señal de fin de trama (*End_Of_Frame*, EOF).

Este proceso de verificación, permite, en caso de que no se estén generando las interrupciones correctamente, comprobar si los datos que se están enviando como entrada al coprocesador de eventos eran los que cabía esperar, o si, el protocolo de comunicación que está implementando dicho coprocesador es correcto. Sin embargo, en caso de que no se generen las tramas de salida, es imposible comprobar a que se debe, ya que tal y como se ha presentado, solo se puede acceder a los puertos de entrada y salida, es decir, a las señales que componen la interfaz LocalLink del diseño.

7.1 Acceso a señales internas

Durante la depuración física del diseño volcado en la FPGA, puede ser necesario acceder a señales internas al coprocesador de eventos para capturar los datos a la entrada de ciertos módulos de granularidad más baja, como pueden ser las memorias, las FIFOs, o las señales de entrada y de salida de los bloques de procesamiento en los que se divide.

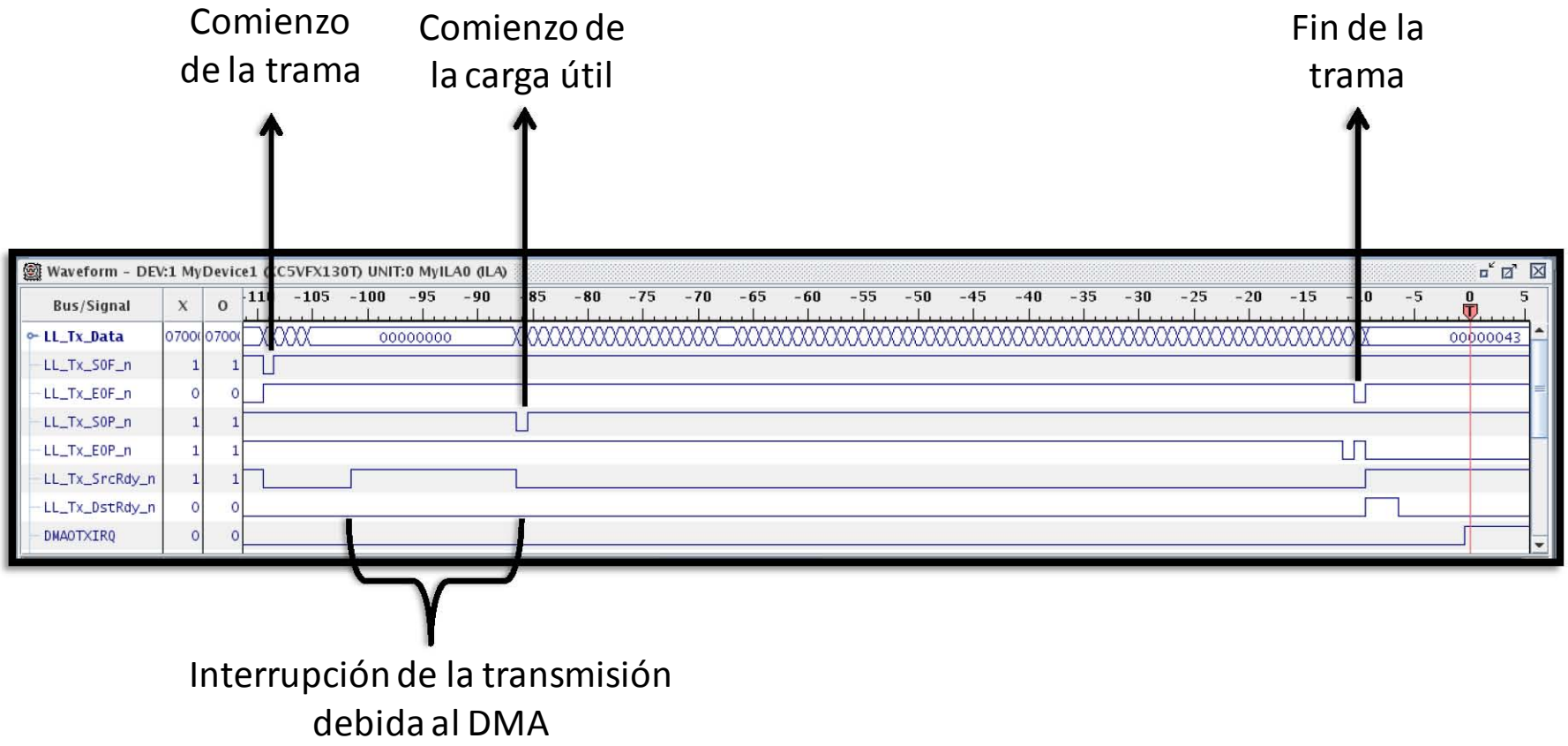


Figura 117. Captura de datos con ChipScope.

Como ya se comentó en el Capítulo 6, el coprocesador de eventos se integró como una caja negra por lo que desde la herramienta de diseño de la plataforma no será posible acceder a las señales internas.

La herramienta utilizada para dar solución a esta necesidad es FPGA Editor. Se trata de una herramienta de post-implementación que permite, a partir de un diseño ya implementado (mapeado, posicionado y ruteado) modificar ciertas conexiones manualmente sin tener que dar pasos atrás, y volver a generar el fichero de configuración de la FPGA.

De esta herramienta, el presente apartado hace alusión únicamente a su cualidad para conectar distintas señales a los puertos de entrada de los bloques ILA instanciados en la plataforma, sin describir el resto de utilidades que presenta.

La conexión de cada puerto de entrada del analizador lógico se realiza bit a bit, y en cada modificación la herramienta modifica el fichero NCD implementado, por lo que para modificar un alto número de señales (en el presente PFC, el número de bits es 78) se hace necesario escribir un *script* en el que se indiquen todas las conexiones. En la Figura 118 se muestra cómo se realiza la conexión de las señales de disparo del bloque ILA. En la búsqueda de la señal de destino de la conexión, pueden usarse expresiones regulares como las que se suelen utilizar en los *shell* de UNIX. A continuación se describe un ejemplo, en el que se conectan todas las señales del analizador, a las que previamente se han descrito, es decir, a los puertos de entrada y salida.

```
ila connect -force chipscope_ila_0/chipscope_ila_0/i_chipscope_ila_0/
trigger 0 rt_core_LLINK0_LL_Tx_Data[0]
ila connect -force chipscope_ila_0/chipscope_ila_0/i_chipscope_ila_0/
trigger 1 rt_core_LLINK0_LL_Tx_Data[1]
…
ila connect -force chipscope_ila_0/chipscope_ila_0/i_chipscope_ila_0/
trigger 31 rt_core_LLINK0_LL_Tx_Data[31]

ila connect -force chipscope_ila_0/chipscope_ila_0/i_chipscope_ila_0/
trigger 32 rt_core_LLINK0_LL_Tx_SOF_n
ila connect -force chipscope_ila_0/chipscope_ila_0/i_chipscope_ila_0/
trigger 33 rt_core_LLINK0_LL_Tx_SOP_n
ila connect -force chipscope_ila_0/chipscope_ila_0/i_chipscope_ila_0/
trigger 34 rt_core_LLINK0_LL_Tx_EOP_n
ila connect -force chipscope_ila_0/chipscope_ila_0/i_chipscope_ila_0/
trigger 35 rt_core_LLINK0_LL_Tx_EOF_n
…
ila connect -force chipscope_ila_0/chipscope_ila_0/i_chipscope_ila_0/
trigger 77 ppc440_0_DMA0RXIRQ
```

8 Pruebas de rendimiento

Con el fin de comprobar el rendimiento del sistema *hardware* propuesto, se han realizado un conjunto de medidas que probasen la calidad del diseño. Para poder comprobar dónde se encuentra el cuello de botella del sistema, se han realizado diferentes ejecuciones, cambiando los puntos de medida, como se presenta a continuación.

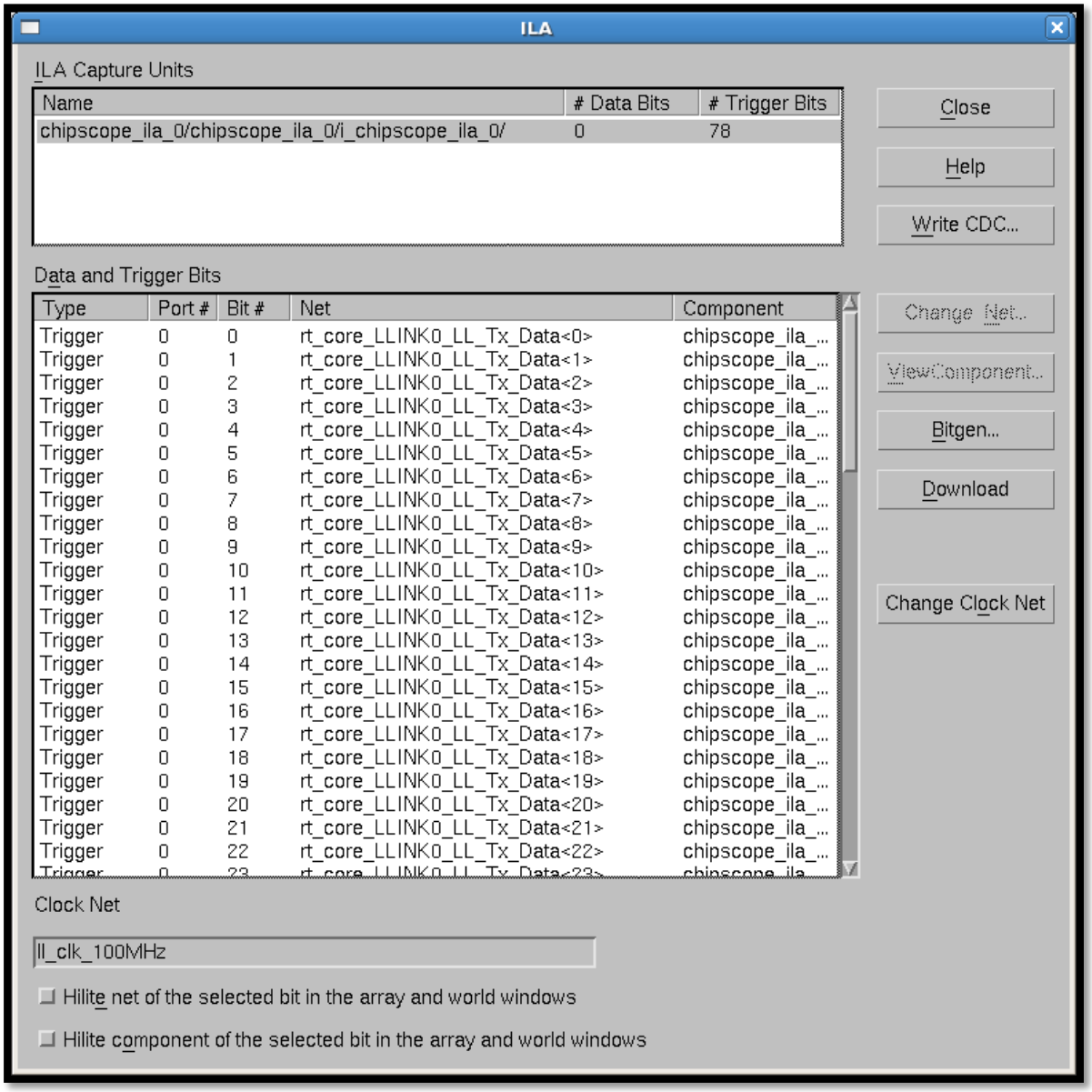


Figura 118. Conexión de puertos de un bloque ILA en FPGA Editor.

8.1 Bloque IP

Una necesidad básica a la hora de medir el rendimiento, es poder obtener marcas temporales, durante la ejecución del código. Para ello se hace uso de un bloque IP *hardware* añadido a la plataforma llamado XPS_Timer.

Este bloque permite, haciendo uso de las funciones de la librería asociada a este, acceder al valor del registro interno, y así contabilizar los ciclos de reloj que ha habido desde el último desbordamiento del mismo. Además, puede configurarse para que active una señal de interrupción cuando esto ocurra, para así, poder contabilizar tiempos de ejecución muy grandes conociendo el número de veces que se ha desbordado.

8.2 Puntos de medida

La librería con las funciones de acceso al bloque IP tiene por nombre `xtmrctr.h`, y alberga un conjunto de métodos que permiten inicializarlo, leer el valor del registro, así como otras utilidades que se le puede exigir a este tipo de bloques.

Una medida de rendimiento se podrá obtener tras contabilizar el tiempo transcurrido desde un punto de la aplicación hasta otro. Para conocer dicho tiempo, se realizará una medida de ciclos en un instante, y una segunda medida posterior, con lo que su diferencia indicará el tiempo transcurrido.

Para obtener las marcas de tiempo, se leerá el registro del *timer* o contador indicado, para lo que antes hay que configurarlo, indicando el valor de *reset* (aquel valor a partir del cual el contador vuelve a cero y activa la señal de interrupción), así como indicando la rutina de servicio vinculada a su interrupción.

La resolución de estas medidas será la del tiempo de ciclo de reloj del contador, en este caso aquel que proporciona la interfaz de bus PLB, es decir, de 10 ns.

Con el fin de analizar el cuello de botella del sistema, se realizarán medidas de rendimiento independientes para el corprocesador de eventos y para el pre-procesado *software* realizado en el PowerPC.

8.3 Rendimiento del coprocesador de eventos

Teniendo en cuenta que cada trama de eventos tiene una longitud variable tras su pre-procesado, debido al filtrado realizado en la aplicación empotrada, se realiza la medida de rendimiento como un resultado estadístico. Así pues, la tasa de eventos dependerá también de las reglas configuradas en el procesador de eventos así como de la tasa de aciertos que ocurran. Debido a estas dependencias, se realiza la medida como la latencia media de procesado, de un conjunto lo suficientemente grande.

Además, para evitar que el pre-procesado *software* interfiera en las medidas, se realizará el pre-procesado de todas las tramas, antes de tomar la medida de tiempo inicial. De esta forma, se conseguirá que en la trama a enviar al coprocesador de eventos se encuentren, almacenadas consecutivamente, un número alto de tramas previamente pre-procesadas.

Una vez las tramas se encuentren listas para el envío, se toma la primera medida de tiempo, y se envía la notificación al DMA (el descriptor ya explicado anteriormente) para que envíe toda la trama. Debido a la limitada capacidad de las colas de entrada del coprocesador, el cual es incapaz de almacenar a su entrada más de dos o tres tramas, la transacción LocalLink sufrirá cortes en la comunicación. Según el coprocesador vaya procesando tramas, liberará la cola para que el DMA pueda seguir enviando el resto.

De esta forma, en el momento en que se genere la interrupción de trama completamente enviada, se sabrá que se han procesado al menos, todas las tramas menos las dos o tres últimas que se encontrarán en las colas de entrada del coprocesador de eventos.

La segunda marca de tiempo se tomará en la rutina de servicio de la interrupción de transmisión, obteniendo así el tiempo de procesado.

Para las medidas, se pre-procesaron 40.000 tramas, y el intervalo de tiempo de procesado *hardware* fue de 80 ms. Así, el rendimiento del procesamiento *hardware* se obtiene como:

$$Rendimiento = \frac{40.000 \cdot 10\ eventos\ en\ promedio\ por\ trama}{80 \cdot 10^{-3}} = 5.000.000\ eventos/s$$

Estos resultados presentan una mejora de casi un orden de magnitud, frente a las medidas realizadas de la aplicación original que dieron como resultado una media de 600.000 eventos/s, siendo ejecutada sobre un servidor de cómputo cuyas propiedades se describen en la tabla 21.

Tabla 21. Características del servidor de cómputo.

Característica	Valor
Modelo CPU	Intel Xeon
Frecuencia CPU	2,67 GHz
Cantidad de memoria RAM	8 GB
Tipo de memoria RAM	DDR3

A partir de estas medidas se puede medir el factor de *SpeedUp* (valor proporcional a la aceleración alcanzada) por el modelo *hardware* sintetizado para la tecnología propuesta:

$$SpeedUp = \frac{5.000.000}{600.000} = 8,\hat{3}$$

8.4 Rendimiento del pre-procesado *software*

Para el rendimiento del pre-procesado, se realiza una medida de tiempo al recibir una trama, y se vuelve a tomar cuando esta va a ser enviada a través del LocalLink por el DMA. Tras realizar estas medidas de un gran número de tramas, se calcula el valor medio de pre-procesado (este también varía según la longitud final de la trama filtrada).

El resultado obtenido es que la latencia media por trama de cotización es de 124 µs. Si se eliminan las tareas de *byte-swapping* y de traducción de tipos de datoa *float* a *fixed*, las latencias mínimas obtenidas son de 27 µs. Esto implica que, la tarea de leer de un espacio de memoria los datos recibidos por la red y su escritura en otro espacio de memoria para preparar el *buffer* del descriptor del DMA consume más tiempo del que se dispone si se quiere mantener el mismo rendimiento que el coprocesador de eventos. En la tabla 22 se indican las latencias del preprocesado *software* original, el preprocesado consistente en traspasar los datos de entrada a un segundo *buffer*, y las del procesado *hardware*, para su comparación.

Tabla 22. Latencias de procesado del sistema.

	Pre-procesado software	Traspaso de datos en software	Procesado hardware
Latencia	124 µs	27 µs	2 µs

A tenor de los resultados, se comprende que las tareas de lectura y escritura de memoria son críticas en el procesador empotrado de la FPGA. Por lo que, en la tarea de recibir de la red las tramas de entrada y enviarlos al coprocesador de eventos, debe minimizarse las transacciones con memoria. Una solución consiste en crear el descriptor de DMA con los datos en la misma posición de memoria donde fueron ubicados en la recepción TCP/IP, eliminando así transacciones de memoria. Esta solución sin embargo, obliga a que los datos recibidos por la red estén previamente tratados para que el coprocesador *hardware* los procese.

9 Generación del sistema autónomo

El último paso, para disponer de un sistema autónomo, es disponer de algún sistema que permite auto-configurar la FPGA en el arranque de la placa, así como pre-cargar la aplicación empotrada en memoria en el mismo instante.

Para este cometido, se presentan los ficheros ACE, almacenados en la CompactFlash de la placa de prototipado. Dichos ficheros, son generados a partir de dos ficheros fuentes, el de configuración de la FPGA, y el ejecutable de la aplicación empotrada.

Para generar el fichero, se debe lanzar la herramienta XMD (*Xilinx Microprocessor Debugger*), indicándole como parámetros de entrada, el *script* TCL de generación de ficheros ACE, los ficheros fuente anteriormente citados, el fichero de salida y la placa de prototipado. A continuación se muestra un ejemplo para la placa ML510.

```
xmd -tcl genace.tcl -hw system.bit -elf exec.elf -ace rt.ace -board ml510
```

El fichero generado, se almacena en la tarjeta de memoria Flash de la placa, de donde la placa lo leerá al arranque. A partir de este punto, el sistema se auto-configurará, sin necesidad de cable de programación, una vez se suministre energía a la placa.

10 Conclusiones

En este capítulo se han presentado los pasos para integrar un bloque *hardware* sintetizado por herramientas externas, al flujo de diseño de sistemas empotrados de Xilinx. Además, se comentan las herramientas necesarias para el depurado *hardware* en tiempo real, junto con el flujo a realizar para poder realizar las capturas en cualquier punto del diseño.

Por último, se exponen los resultados de rendimiento del coprocesador de eventos, con un detallado análisis del cuello de botella del sistema, así como sus causas y posibles soluciones.

Capítulo 7:
Conclusiones y líneas futuras

1 Introducción

En este capítulo se presentan las conclusiones generales del presente Proyecto Fin de Carrera, así como posibles líneas de trabajo que parten de dichas conclusiones, con el fin de mejorar sus prestaciones, o para generalizar los resultados a otros ámbitos.

2 Conclusiones

En el presente proyecto se han abarcado los pasos del flujo de diseño para, a partir de una aplicación *software* de referencia, implementar un sistema empotrado, formado por un microprocesador y un coprocesador *hardware*, que implemente la misma funcionalidad que el modelo original.

Para ello, se ha trabajado con herramientas de síntesis de alto nivel, proceso que aún teniendo una vida amplia, no ha alcanzado a sustituir el flujo clásico de diseño *hardware* que parte, en general, de la realización de su descripción a nivel RTL con lenguajes HDL como Verilog o VHDL.

Además, se ha diseñado un *SoC* compuesto por un microprocesador y un conjunto de bloques independientes, interconectados mediante bus y DMA, todo en el mismo dado de silicio, en este caso una FPGA.

Se han realizado además, depurados físicos sobre el sistema implementado en placa, haciendo uso de un analizador lógico y su *software* asociado.

El resultado principal del proyecto, puede resumirse en un sistema completamente funcional, que responde a la funcionalidad esperada. El diseño ha sido testado para las estrategias y eventos de mercado proporcionados por los propietarios de la aplicación de referencia, produciendo las mismas tramas de salida.

Gracias al desarrollo de este proyecto, se han alcanzado, además, ciertas conclusiones acerca del desarrollo de sistemas empotrados para el procesado de eventos, como se detallan a continuación:

- La partición *software* del sistema debe minimizarse todo lo posible ya que, en cualquier caso, introduce las latencias máximas del sistema, produciendo un cuello de botella e impidiendo que la partición *hardware*, que es el núcleo del procesador, alcance unas tasas de procesado óptimas.
- Si se suprimen las latencias del pre-procesamiento *software*, se llega a la conclusión de que una interfaz Gigabit Ethernet es insuficiente para suministrar eventos a la velocidad que es capaz de procesar el núcleo *hardware* de procesamiento. En otras palabras, el cuello de botella del sistema se encuentra en el suministro de eventos al núcleo de procesamiento *hardware* del sistema.
- El rendimiento y gran parte del flujo de desarrollo es dependiente del procesador de eventos. Existen procesadores de eventos que solo requieren de una trama para su procesado, mientras que otros más complejos, hacen alusión a estados pasados, así como a otras tramas que ya se recibieron o se recibirán en el futuro. Además, la modificación de las condiciones de procesado en tiempo de ejecución añade complejidad a su diseño.
- Las herramientas de síntesis de alto nivel están en constante evolución, permitiendo cada vez más abstracción en el diseño a *hardware* a nivel de sistemas. La tendencia de dichas herramientas indican que en un futuro el diseño de sistemas *hardware* será cada vez más abstracto, como por ejemplo, en la etapa de comunicación con las librerías TLM, permitiendo diseñar sistemas mucho más complejos sin la necesidad de plantear las transacciones entre bloques a nivel de ciclos.

3 Líneas futuras

A partir de las conclusiones del presente proyecto, se plantean nuevas líneas de trabajo que parten de estas o que usan estas como base en su desarrollo. A continuación se indican algunas posibles:

- Implementar el diseño en FPGAs actuales, con nuevos microprocesadores y buses de conexión, con mayor número de recursos, para mejorar así los resultados de colocación y con ello la frecuencia de funcionamiento.
- Hacer uso de nuevas herramientas de desarrollo de sistemas empotrados como puede ser el kit de desarrollo *Vivado Design Suite* con el fin de estudiar las mejoras aportadas en la minimización del tiempo de desarrollo y de los recursos humanos necesarios.
- Replicar los núcleos de procesamiento, para lo cual se debe diseñar una unidad de emisión para evitar el procesamiento fuera de orden.
- Estudiar otro método de comunicación *software-hardware*, como puede usar placas de prototipado con posibilidad de conexión PCIe a una placa base, para así usar un microprocesador no empotrado con mayor capacidad de cálculo para el pre-procesado, y usar PCIe como bus de conexión con el bloque *hardware*.

BIBLIOGRAFÍA

[1] N. Nedjah, L. de Macedo Mourelle. *Co-Design for System Acceleration : A Quantitative Approach* .Universidad de Michigan: Springer, 2007.

[2] M. Gokhale, P. S. Graham. *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Birkhäuser, 2005.

[3] S. Levi, A. K. Agrawala. *Real-Time System Design.* Universidad de Michigan: McGraw-Hill Pub. Co., 1990.

[4] G. F. Mauer, "A fuzzy logic controller for an ABS braking system," en *Fuzzy Systems, IEEE Transactions on*, 2009.

[5] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, H. Jacobsen, "Efficient Event Processing through Reconfigurable Hardware for Algorithmic Trading," en *36th International Conference on very Large Data Bases (VLDB),* 2010.

[6] M. John, S. Smith, *Application-Specific Integrated Circuits.* Addison Wesley Professional, 2008.

[7] T. Grötker, *System Design with SystemC.* Boston: Kluwer Academic Publishers, 2002.

[8] M. Fingeroff, *High-Level Synthesis : Blue Book.* S.L.: Xlibris Corporation, 2010.

[9] P. Coussy, A. Morawiec, *High-Level Synthesis from Algorithm to Digital Circuit*.Springer, 2008.

[10] B. Cico and H. Rexha. "Implementing Codesign in Xilinx Virtex II Pro," presentedo en la cuarta Balkan Conference, 2009.

[11] *TMS320DM816x DaVinci Digital Media Processors: Technical Reference Manual* , Texas Instruments, Inc., 2011.

[12] M. Muraoka, H. Nishi, R. K. Morizawa, H. Yokota, H. Hamada. "Design methodology for SoC architectures based on reusable virtual cores," presentado en Design Automation Conference, 2004.

[13] Trong-Yen Lee, Yang-Hsin Fan, Yu-Min Cheng, Chia-Chun Tsai, Rong-Shue Hsiao. "Enhancement of hardware-software partition for embedded multiprocessor FPGA systems," presentado en Intelligent Information Hiding and Multimedia Signal Processing, 2007.

[14] Renesas Technology America, Inc. Renesas Application Specific Products**.** [En línea]**.** Disponible: http://www.arm.com/community/partners/display_product/rw/ProductId/2190/

[15] P. Bishop. Using ARM Processor-based Flash MCUs as a Platform for Custom Systems-on-Chip. [En línea]. Disponible: http://www.design-reuse.com/articles/13742/using-arm-processor-based-flash-mcus-as-a-platform-for-custom-systems-on-chip.html

[16] I. García. (Enero 2010). Procesador Inteligente de Eventos (IEP) con OpenESB. [En línea]. Disponible: http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=intelligent-event-processor-openesb

[17] J. Herrera, J. García, X. Perramon, "Aspectos avanzados de seguridad en redes: Mecanismos para la detección de ataques e intrusiones,".

[18] ActiBVA. (Agosto 2011). ¿Qué es el Algorithmic Trading?. [En línea]. Disponible: http://www.actibva.com/magazine/mercados-financieros/que-es-el-algorithmic-trading

[19] R. Ranjan. (2009). Algorithmic Trading - an Introduction. [En línea]. Disponible: https://sites.google.com/site/rajeevranjansingh/facts-and-figures-need-for-speed

[20] The Economist. (Junio 2007). Algorithmic trading: Ahead of the tape. [En línea]. Disponible: http://www.economist.com/node/9370718

[21] Edosoft Factory S.L. (Julio 2008). Robobroker. [En línea]. Disponible: http://www.edosoftfactory.com/images/stories/dossier_rb.pdf

[22] Real Academia Española. *Diccionario De La Lengua Española* (22º ed.) 2001.

[23] *EDK Concepts, Tools, and TechniquesA Hands-On Guide to Effective Embedded System Desig,* Xilinx, Inc., 2010.

[24] *Virtex-5 Family Overview*, Xilinx, Inc., 2009.

[25] *Virtex-5 FPGA User Guide,* Xilinx, Inc., 2012.

[26] *Virtex-5 FPGA: XtremeDSP Design Consideration*, Xilinx, Inc., 2012.

[27] *ML505/ML506/ML507 Evaluation Platform User Guide*, Xilinx, Inc., 2011.

[28] *ML510 Embedded Development User Guide*, Xilinx, Inc., 2011.

[29] S. Rigo, R. Azevedo, L. Santos. *Electronic System Level Design: An Open-Source Approach. Springer,* 2011.

[30] *IEEE Standard for Standard SystemC® Language Reference Manual*. IEEE Computer Society, 2012.

[31] D. C. Black, J. Donovan, SpringerLink. *SystemC: From the Ground Up.* Springer, 2009.

[32] M. Ganguly, *SystemC Overview*, Synopsys, Inc., 2001.

[33] V. de Armas Sosa, *Tutorial Verilog*, 2004.

[34] *Callgrind: a call-graph generating cache and branch prediction profiler*, Valgrind Developers, 2011. [En línea]. Disponible: http://valgrind.org/docs/manual/cl-manual.html.

[35] J. Weidendorfer, *kcachegrind: CallGraphViewer,* 2011. [En línea]. Disponible: http://kcachegrind.sourceforge.net/html/Documentation.html.

[36] *Cadence C-to-Silicon Compiler: User Guide*, Cadence Design Systems, Inc., 2011.

[37] *Synplify Premier User Guide*, Synopsys, Inc., 2011.

[38] *Introduction to SimVision,* Cadence Design Systems, Inc., 2011.

[39] *Embedded System Tools Reference Manual,* Xilinx, Inc., 2011.

[40] *PlanAhead: User Guide*, Xilinx, Inc., 2011.

[41] *ChipScope Pro Software and Cores: User Guide*, Xilinx, Inc., 2011.

[42] C. Stroud, *Overview of FPGA Editor*, Universidad de Auburn .

[43] *Virtex-5 FPGA Embedded Processor Block with PowerPC 440 Processor (Wrapper) (v1.01a): Product Specification*, Xilinx, Inc., 2011.

[44] *LogiCORE IP DDR2 Memory Controller for PowerPC 440 Processors (v3.00c): Product Specification,* Xilinx, Inc., 2011.

[45] *LogiCORE IP XPS Interrupt Controller (v2.01a): Product Specification,* Xilinx, Inc., 2010.

[46] *JTAGPPC Controller (v2.01c): Product Specification*, Xilinx, Inc., 2009.

[47] *LogiCORE IP XPS LL TEMAC (v2.03a): Product Specification*, Xilinx, Inc., 2010.

[48] *XPS SYSACE (System ACE) Interface Controller (v1.01a): Product Specification*, Xilinx, Inc., 2009.

[49] *XPS UART Lite (v1.01a): Product Specification,* Xilinx, Inc., 2009.

[50] *LogiCORE IP Clock Generator (v4.01a): Product Specification*, Xilinx, Inc., 2010.

[51] *ChipScope Pro Integrated Logic Analyzer (v. 1.00a, 1.01a, 1.02a, 1.03a): Product Specification,* Xilinx, Inc., 2010.

[52] *LogiCORE IP ChipScope Pro Integrated Controller (ICON) (v1.05a): Product Specification*, Xilinx, Inc., 2011.

[53] *LogiCORE IP Processor Local Bus (PLB) v4.6 (v1.05a): Product Specification,* Xilinx, Inc., 2010.

[54] *LocalLink Interface Specification*, Xilinx, Inc., 2005.

[55] *Embedded Processor Block in Virtex-5 FPGAs: Reference Guide*, Xilinx, Inc., 2010.

[56] *ML510 BSB2 Design Creation Using 11.1 EDK Base System Builder*, Xilinx, Inc., 2009.

[57] J. Lucero, *Reference System: Designing an EDK Custom Peripheral with a LocalLink Interface*, Xilinx, Inc., 2008.

[58] *Virtex-5 FPGA Embedded Tri-Mode Ethernet MAC: User Guide*, Xilinx, Inc., 2011.

[59] *ML510 BSB1 Design: Adding RGMII Standard IP Using 10.1 SP3 EDK*, Xilinx, Inc., 2008.

[60] *OS and Libraries Document Collection*, Xilinx, Inc., 2010.

[61] *Using EDK to Run Xilkernel on a PowerPC 440 Processor: Example Design,* Xilinx, Inc., 2011.

[62] S. MacMahon, N. Zang, A. Sarangi, *LightWeight IP (lwIP) Application Examples*, Xilinx, Inc., 2011.

[63] *XST User Guide*, Xilinx, Inc., 2009.

[64] A. Sutton, *Divide and Conquer: Faster FPGA Delivery using Hierarchical, Parallel Design Development*, Synopsys, Inc., 2011.

Presupuesto

1 Introducción

En este presupuesto se hace un análisis y balance de los costes tanto totales como parciales presentes en este proyecto, distinguiendo entre recursos materiales, costes de ingeniería, material fungible y costes de redacción del proyecto.

2 Recursos materiales

En este apartado se exponen los costes producidos por el uso de material *hardware* y *software* y el mantenimiento de los mismos. Tanto los equipos *hardware* como las herramientas *software* presentan un coste de amortización en función del período de tiempo usados.

2.1 Recursos hardware

Se describen a continuación los equipos hardware usados durante el desarrollo del presente proyecto, incluyendo características que lo definen y posteriormente se resume en una tabla el coste asociado a la amortización de los mismos. Los equipos de desarrollo lo componen una estación de trabajo para las labores de diseño y verificación, un *cluster* de servidores de cómputo para las tareas de síntesis (a cada nivel en el flujo de desarrollo) que permite síntesis paralela de varios bloques y la placa de prototipado para la implementación del diseño.

- **Estación de trabajo**: Sun Ultra 27 Workstation (Intel Xeon @2,67 GHz, 6GB RAM DDR3)
- **Servidores de cómputo**: SunFire X2200 (AMD Opteron @2,2 GHz, 4GB RAM DDR3)
- **Placa de prototipado**: Xilinx ML510 (Virtex-5 XC5VFX130T)

Tanto la estación de trabajo como los servidores de cómputo ejecutan como sistema operativo Red Hat Enterprise 5.

Tabla 23. Coste de recursos hardware.

Descripción	Coste (€)	Factor de amortización		Meses de uso	Coste asociado (€)
		Meses	Usuarios		
Sun Ultra 27 Workstation	1.200,00	12	1	12	1.200,00
Cluster SunFire X2200	12.504,00	12	5	9	1.875,60
Xilinx ML510	3.672,21	Dedicación total		-	3.672,21
				TOTAL:	6.747,81

2.2 Recursos software

Las herramientas software usadas para el desarrollo del proyecto tienen un coste asociado a sus licencias así como a su mantenimiento por parte de los suministradores. Estas herramientas

son suministradas por los proveedores en forma de paquetes que incluyen varias herramientas, por lo que todas las herramientas de un mismo suministrados estarán comprendidos en el mismo paquete. En la tabla siguiente se enumeran cada una de las herramientas así como el precio asociado a un año de utilización para el formato de licencia universitaria. En el caso de las herramientas de síntesis se incluye el precio de un *pool* de licencias para permitir la ejecución paralela de procesos de síntesis.

Tabla 24. Coste de recursos software.

Herramienta	Tipo de licencia	Coste (€)
Callgrind (Valgrind)	GNU GPL / Free software	0,00
Kcachegrind	GNU GPL / Free software	0,00
Paquete Cadence (Pool de licencias)	Universitaria	4.030,00
Paquete Synopsys (Pool de licencias)	Universitaria	250,00
Paquete Xilinx	Universitaria	200,00
	TOTAL:	4.480,00

A continuación se contabilizan los costes totales de recursos materiales, procedentes de recursos *hardware* y *software*.

Tabla 25. Coste de recursos materiales.

Descripción	Coste (€)
Recursos hardware	6.747,81
Recursos software	4.480,00
TOTAL:	11.227,81

3 Costes de ingeniería

Se ha invertido un total de 12 meses a jornada completa al desarrollo del presente proyecto. Durante el mismo se han realizado tareas de formación, de desarrollo y de documentación mediante la descripción en el presente documento. Para el coste de un ingeniero se ha contemplado la Tabla de Contrataciones publicada en el Boletín Oficial de la Universidad de Las Palmas de Gran Canaria, que atribuye un coste mensual de 1.921,64 € para un técnico en proyecto con titulación de Ingeniero.

Tabla 26. Costes de ingeniería.

Descripción	Meses	Coste mensual (€)	Coste total (€)
Formación	1	1.921,64	1.921,64
Desarrollo	9	1.921,64	17.294,76
Memoria	3	1.921,64	5.764,92
		TOTAL:	24.981,32

4 Material fungible

Se incluyen en este apartado los gastos asociados a consumibles utilizados durante el desarrollo del proyecto como pueden ser impresiones de documentación, CDs, DVDs, cables, etc. Se calcula un gasto total de 200€.

5 Costes de edición del proyecto

El coste de edición de la memoria, incluyendo folios, gastos de impresión y encuadernación asciende a 300€.

6 Resumen de costes

La tabla 27 muestra el coste total en función de los costes desglosados en los apartados anteriores.

Tabla 27. Resumen de costes

Descripción	Coste (€)
Recursos materiales	11.227,81
Costes de ingeniería	24.981,32
Material fungible	200,00
Costes de edición	300,00
Total antes de impuestos	36.709,13
5% IGIC	1.835,46
TOTAL:	38.544,59

D. Omar Espino Santana declara que el Proyecto Fin de Carrera: “Acelerador Hardware para el Procesado de Eventos en Tiempo Real (AHPETIR)” asciende a un total de **38.544,59 €**.

Fdo. Omar Espino Santana

En Las Palmas de Gran Canaria a 10 de mayo de 2012.

Pliego de condiciones

1 Introducción

En la realización de este proyecto se ha hecho uso de un conjunto de herramientas *software* y equipos *hardware* cuyas características y versiones se listan en los siguientes apartados así como la garantía de cada uno de ellos.

2 Equipos *hardware*

En la tabla 28 se muestran todos los recursos *hardware* utilizados.

Tabla 28. Equipos *hardware*.

Equipo	Descripción
Sun Ultra 27 Workstation	Estación de trabajo Intel Xeon @2,67 GHz, 6GB RAM DDR3 bajo Red Hat Enterprise Linux Server 5.3
Servidor SunFire X2200	Servidor de cómputo AMD Opteron @2,2 GHz, 4GB RAM DDR3 bajo Red Hat Enterprise Linux Server 5.3
Xilinx ML510	Placa de prototipado

3 Herramientas *software*

En la tabla 29 se muestran las herramientas software utilizadas especificando su versión.

Tabla 29. Herramientas *software*.

Aplicación	Versión	Fabricante	Descripción	Plataforma
Callgrind	3.5.0	Valgrind™ Developers	Herrmienta de perfilado.	Linux
Kcachegrind	0.4.6 kde	Valgrind™ Developers	Representación de perfil *software*.	Linux
C-to-Silicon Compiler	11.10-s200	Cadence	Entorno de síntesis de alto nivel.	Linux
Synplify Premier	F-2011.09-SP1-1	Synopsys	Entorno de síntesis lógica.	Linux
Incisive Enterprise Simulator	10.20-s073	Cadence	Entorno de simulación.	Linux
Xilinx Platform Studio	12.4	Xilinx	Entorno de diseño de sistemas empotrados.	Linux
PlanAhead	13.4	Xilinx	Entorno de síntesis física.	Linux
iMPACT	13.4	Xilinx	Herramienta de programación de FPGAs.	Linux
ChipsCope Analyzer	13.4	Xilinx	Herramienta de análisis lógico.	Linux
FPGA Editor	13.4	Xilinx	Herramienta de configuración de FPGAs.	Linux
Microsoft Office	2007	Microsoft	Paquete ofimático.	Windows 7

4 Garantía

El autor del módulo lo presenta “AS IS” (tal cual), sin garantía implícita de ningún tipo. Tamposo se responsabiliza de los daños que puedan causar el código presentado o sus archivos derivados a cualquier equipo o del uso que hagan de ellos terceras personas.

5 Cláusula de confidencialidad

El presente Proyecto Fin de Carrera se desarrolla dentro de las actividades del proyecto RT-CORE realizado conjuntamente por el Instituto Universitario de Microelectrónica Aplicada (IUMA) de la Universidad de Las Palmas de Gran Canaria y la empresa Edosoft Factory S.L. Por ello, el material usado y/o producido o referenciado en este proyecto puede estar sujeto a cláusulas de confidencialidad, lo que se declara a los efectos oportunos, según se especifica en el actual reglamento de Proyecto Fin de Carrera de la EITE.