
# UNIVERSIDAD CARLOS III DE MADRID

## ESCUELA POLITÉCNICA SUPERIOR

### DEPARTAMENTO DE INFORMÁTICA

### GRUPO DE ARQUITECTURA DE COMPUTADORES, COMUNICACIONES Y SISTEMAS

## *Implementación y evaluación de algoritmos de compresión en FPGAs*

PROYECTO FIN DE CARRERA

INGENIERÍA TÉCNICA INDUSTRIAL: ELECTRÓNICA INDUSTRIAL

**Autor**: Juan Luis Fernández Moya

**Tutores**: David Expósito Singh

Óscar Pérez Alonso

Marzo 2.012

**Proyecto Fin de Carrera**

**TÍTULO:** Implementación y evaluación de algoritmos de compresión en FPGAs

**AUTOR:** Juan Luis Fernández Moya

**TUTORES:** David Expósito Singh
Óscar Pérez Alonso

La defensa del presente Proyecto Fin de Carrera se realizó el día 7 de Marzo de 2.012, siendo calificada por el siguiente tribunal:

**PRESIDENTE:** Jesús Carretero Pérez

**SECRETARIO:** Javier Fernández Muñoz

**VOCAL:** Ricardo Vergaz Benito

Que acuerda otorgarle la calificación de:

**PRESIDENTE:**

**SECRETARIO:**

**VOCAL:**

# AGRADECIMIENTOS

¿Cómo expresar en pocas palabras todo lo que me gustaría decir?

En primer lugar quiero darles mi más sentido agradecimiento a David y a Óscar, que han sido unos tutores fabulosos. Me han ayudado en todo y me han prestado su atención siempre que los he necesitado. Sólo puedo expresar una infinita gratitud. Y tampoco puedo olvidarme de Jesús, el jefe, ni de sus inconfundibles saludos *"¿qué pacha 'letrónico?"*, que nunca perdía oportunidad de meterse un poco conmigo pero que siempre se interesaba por como me iba todo y que me animó a tirar p'alante cuando peor me pintaban las cosas con el proyecto. A los tres gracias.

A Alber, Varo, Gallo, Muñoz, Juanjo y Cristian, mis amigos de toda la vida. Porque para llegar a la universidad primero hay que pasar por la terrible adolescencia y el instituto, y son años difíciles que siempre es mejor pasar con unos buenos amigos, y ellos llevan junto a mí más de 10 años, incluso más de 20 algunos de ellos. Y tampoco puedo olvidarme de Mire y Junky, que aunque llegaron algunos años después también son parte de esto.

Por supuesto, a toda esta panda locos a quienes he tenido la suerte, el honor y el privilegio de conocer: Edgar, Sergio, Jose, Rodri, Sebas, Sopranis, Gemelo Dani, Gemelo Raúl, Esther, Chechu, Patxi, Alina, Germán, Josito, Solís, Laura, Milongas, Manu, Gabi, Paula, Alberto y Navidad. He pasado 4 años fantásticos estudiando electrónica junto a ellos y sin ellos no habría llegado hasta aquí. Hoy puedo decir que ellos han hecho que estudiar en la universidad haya valido la pena. Tampoco me olvido de otra gente estupenda como Itzi, Juanito, Dairon, Noe o mis compañeros del equipo de fútbol sala "Los Ingenieritos" y esa Copa del Rector que ganamos en 2.010. A todos vosotros, gracias.

Y sin duda, debo agradecerle a mi familia. A mis tíos y primos, que siempre me han animado a seguir.

A mi abuela Fina, que ya hace unos años que se fue, que la echo mucho de menos y que seguro que allá donde esté me estará viendo orgullosa de mí y feliz como siempre fue.

A mi hermana Sonia, que no puedo expresar con palabras todo el cariño que tengo por ella.

Pero sobretodo, sobretodo, si tengo que darle las gracias a alguien es a mis padres, Juan Antonio y Sagrario, que siempre confiaron en mí y lo han dado absolutamente todo para que yo pudiera llegar hasta aquí. Porque este éxito es realmente su éxito.

## MIL GRACIAS A TODOS

# RESUMEN

A partir de un algoritmo software de compresión de datos se ha realizado un diseño teórico del mismo en lenguaje VHDL que posteriormente ha sido implementado en un soporte hardware. La idea inicial era crear un algoritmo en VHDL que permitiera realizar una compresión simple de datos de tamaño de palabra arbitrario. Finalmente se amplió el diseño añadiendo nuevas líneas de código que permitieran implementar un nuevo algoritmo que realiza una doble compresión. Este proyecto ha sido realizado en el departamento de informática de la Universidad Carlos III de Madrid.

En primer lugar el algoritmo en VHDL fue creado y probado en simulación mediante la herramienta *Quartus II* del fabricante Altera. Una vez que se comprobó su funcionamiento teórico se procedió a su implementación en la FPGA *Spartan 3A* del fabricante Xilinx. Para ello se ha utilizado el entorno de desarrollo *Xilinx Platform Studio* (XPS) también de Xilinx.

La FPGA *Spartan 3A* viene integrada en la plataforma de desarrollo *Spartan 3A Starter Kit.* Dicha plataforma consta de un microprocesador MicroBlaze, el cual hemos configurado, y que se encarga del manejo de los datos. También hemos configurado una serie de dispositivos, denominados coprocesadores, encargados de realizar la compresión de datos. La conexión entre MicroBlaze y los coprocesadores se estable mediante buses de conexión llamados FSL.

El correcto diseño y funcionamiento del modelo hardware creado se ha validado mediante diferentes pruebas realizadas utilizando diversas tramas de datos y comprobando que efectivamente la compresión de los datos era correcta. Igualmente se ha hecho un estudio del rendimiento y los costes hardware del diseño, tales como memoria, biestables, DCMs, etc. También se ha comparado el rendimiento de nuestro algoritmo con el rendimiento de un algoritmo software previamente existente.

Finalmente se ha documentado un tutorial del manejo del programa XPS que permite una rápida familiarización con dicha herramienta para la implementación de futuros diseños en FPGA.

**Palabras clave**: Compresión, MicroBlaze, periférico, coprocesador, FLS, búffer, contador, Spartan 3A, algoritmo, *Xilinx Platform Studio*, *Quartus II.*

# SUMMARY

A theoretical design in VHDL language has been developed based on a software algorithm for data compression. This algorithm has subsequently been implemented in FPGAs devices. The initial idea was to create a VHDL algorithm that would make a simple compression of data of arbitrary word sizes. Finally, the design was extended implementing a new algorithm that performs a two phase compression. This project was performed at the Computer Science department of Carlos III University of Madrid.

First, the VHDL algorithm was created and tested in a simulation using *Quartus II* tool from Altera. Once the theoretical performance was obtained, it was implemented in a FPGA *Spartan 3A* from Xilinx. We used the development environment called *Xilinx Platform Studio* (XPS) from Xilinx.

The FPGA *Spartan 3A* is integrated into the development platform *Spartan 3A Starter Kit.* This platform consists of a MicroBlaze microprocessor, which we have set, and is responsible for managing the data. Also, we have set up several devices called coprocessors, responsible of carrying out the data compression. The connection between MicroBlaze and the coprocessors is established by FSL buses.

The hardware implementation has been validated by several tests using various data sets and checks for ensuring that data compression was performed correctly. We have also made a study of the performance and hardware costs of the design, measuring the ammount of memory, number of flip-flops and DCMs, etc. Also, the performance of our algorithm has been compared with the performance of a software implementation of the algorithm.

Finally we documented a management tutorial of XPS program that allows a fast familiarization with this tool for the implementation of future FPGA designs.

**Keywords**: Compression, MicroBlaze, peripheral, coprocessor, FLS, buffer, counter, Spartan 3A, algorithm, *Xilinx Platform Studio*, *Quartus II.*

# ÍNDICE

# LISTA DE FIGURAS

# LISTA DE TABLAS

# 1. INTRODUCCIÓN

En este primer capítulo de introducción se pretende ofrecer una visión global del proyecto realizado, y más concretamente de los motivos que nos han llevado a realizarlo y los objetivos que se han perseguido con su ejecución.

La idea principal de nuestro proyecto ha sido la creación de un algoritmo hardware de compresión de datos en código VHDL que sería implementado en una FPGA. Se comenzó realizando un algoritmo de compresión simple y posteriormente se amplió para hacer una compresión doble.

Los datos iniciales son divididos en tres campos, los cuales se comprimen de forma individual respecto a los otros dos. Una serie de contadores se encarga de llevar la cuenta del número de repeticiones de cada uno de los campos.

La implementación en FPGA se realiza mediante el programa *Xilinx Platform Studio*. Este programa también proporciona los recursos necesarios para la configuración de un microprocesador MicroBlaze, programado en lenguaje C. También nos permite la inclusión de coprocesadores, los cuales son programados a través del algoritmo VHDL diseñado.

Una vez diseñados los códigos en C y en VHDL se establece la conexión entre MicroBlaze y el coprocesador o coprocesadores creados a través de buses de conexión llamados FSL.

Por último XPS genera una serie de archivos necesarios que implementan la interactuación entre MicroBlaze y coprocesadores. Una vez generados, estos archivos son cargados en la FPGA.

Con el programa cargado en la FPGA se han realizado diferentes pruebas para comprobar su correcto funcionamiento y finalmente se han realizado pruebas de rendimiento y se ha hecho un análisis de los recursos hardware necesarios para el correcto funcionamiento.

## 1.1. MOTIVACIÓN

En la actualidad uno de los aspectos más importantes en las ciencias de la computación es el de la compresión de datos. La cantidad de datos e información que se manejan hoy en día en cualquier ámbito de la sociedad es tan grande que hacen indispensable su uso.

**La compresión de datos se utiliza fundamentalmente para dos tareas concretas, el almacenamiento de datos y la mejora en las comunicaciones**. La principal ventaja que se obtiene sin duda es la reducción del volumen de datos. La eliminación de datos redundantes y/o innecesarios y el uso de algoritmos son algunas de las técnicas que se usan para ello. Esto permite un mejor almacenamiento de datos, ya que al tener un volumen menor se ocupa menos espacio, con lo cual, el consumo de recursos es menor y además, se hace una mejor selección de la información útil. Por otro lado, también es un pilar fundamental en las comunicaciones. Una menor cantidad de datos supone menos tiempo en la transmisión de la información y nuevamente, disminuye los recursos necesarios para realizar la transmisión. Al respecto del tema, una contribución importante de aplicación de técnicas de compresión ha sido la tesis doctoral ***Técnicas de optimización dinámicas de aplicaciones paralelas basadas en MPI***, realizada por Rosa Filgueira en la universidad Carlos III de Madrid [MPI2010].

Sin embargo, la compresión de datos presenta algunas dificultades, en cuya solución se trabaja actualmente. **El problema fundamental es el tiempo extra de proceso que tiene asociada la compresión/descompresión de los datos**. La compresión de datos requiere una gran cantidad de tiempo y cuanto mayor sea el volumen de datos más lento será el proceso.

Además, supone un consumo de recursos informáticos que conllevan a su vez a un mayor consumo de energía, lo que puede suponer un alto coste económico y de medios.

La realización de este proyecto fin de carrera tiene dos motivaciones principales: **diseñar un algoritmo de compresión que sea rápido y eficiente y además, que reduzca al máximo posible el consumo de recursos.**

Consideramos que la compresión de datos es un tema muy importante y muy interesante que tratar, y por tanto hemos tratado de realizar un meticuloso trabajo que aporte algo nuevo al tema y que pueda ayudar a mejorar la gran labor realizada por otros anteriormente.

Con las premisas descritas anteriormente se ha optado por el uso de las FPGA, cuyo uso está cada vez más extendido. Se ha diseñado un código que realiza una compresión sin pérdidas de información y que realiza una doble compresión, lo que permite una gran reducción del volumen de datos. Además, su implementación en FPGA es un gran acierto, pues estos dispositivos permiten que la compresión de datos se realice a gran velocidad, y sin cómputo extra, es decir, sin utilizar otros recursos informáticos, lo que además supone un gran ahorro energético, pues las FPGA tienen un bajo consumo energético.

## 1.2. OBJETIVOS

En este proyecto fin de carrera se han abordado los siguientes objetivos:

- **Diseño de un algoritmo hardware en lenguaje VHDL** que permita la compresión de grandes cantidades de datos.
- **Implementación de los algoritmos hardware en FPGA**. Una vez se hayan terminado el diseño hardware y la interface software, la implementación debe realizarse con el entorno de desarrollo de Xilinx llamado *Xilinx Platform Studio* (XPS) para configurar un proyecto que incluyan el diseño de una FPGA y un microprocesador Microblaze.
- **Diseño de un interface software realizado en código C** que permita el correcto envío de los datos originales y la correcta recepción de los datos comprimidos desde los diferentes coprocesadores. Finalmente, también debe mostrar por pantalla los datos recibidos.
- **Evaluación con trazas**. Con el programa ya cargado en la FPGA se procederá a evaluar su comportamiento con diferentes cantidades de datos. Esto permitirá ver cuáles son los métodos más óptimos para realizar la compresión de datos.
- **Evaluación de recursos**. Se pretende calcular también la cantidad de memoria de la FPGA que se precisa para almacenar el programa, así como otros dispositivos tales como biestables o DCMs.
- **Comparativa de rendimiento**. Se hará una comparativa del rendimiento del algoritmo hardware diseñado frente a un algoritmo software de compresión ya existente.
- **Documentación del proceso. Finalmente, se realizará un tutorial paso a paso para la** implementación de algoritmos hardware en FPGA. Se busca explicar detalladamente con capturas de pantalla cómo se realiza la configuración de una FPGA, cómo se crea y configura un Microblaze, cómo se crea un coprocesador, cómo se añaden y modifican los códigos C y VHDL, cómo realizar de forma correcta las conexiones y cómo generar y cargar en una FPGA los archivos de programa. También se pretende explicar de qué manera se añaden puertos de entrada al Microblaze y puertos de entrada y salida de los coprocesadores y como poder conectarlos entre sí.

# 2. ESTADO DE LA CUESTIÓN

En este capítulo se pretende realizar una descripción en detalle acerca de las FPGAs y posteriormente se hablará sobre las diferentes técnicas de compresión de datos que existen.

## 2.1. FPGAs

En este apartado se van a explicar qué son y qué aplicaciones tienen las FPGA, se hablará también de la FPGA Spartan3, que es la utilizada en la realización de este proyecto, y finalmente se hablará del entorno de desarrollo *Xilinx Platform Studio*.

### 2.1.1. Qué son y en qué se utilizan

Un **FPGA (Field Programmable Gate Array)** es un dispositivo semiconductor formado por bloques de memoria cuya interconexión y funcionalidad se puede configurar mediante un lenguaje de programación específico, realizándose un código que sustituya a la electrónica digital. Las FPGA surgen como una evolución de los conceptos desarrollados para sistemas PAL y CPLD.

La lógica programable permite programar todo tipo de dispositivos de cómputo, desde puertas lógicas o sistemas combinacionales hasta complejos sistemas en un chip, e incluso hacerlos funcionar de forma paralela [W - FPGA] [ALT1040].

Las FPGA se aplican en funciones similares a los ASICs. Un **ASIC (application-specific integrated circuit)** es un circuito integrado hecho a la medida para un uso en particular, en vez de ser concebido para propósitos de uso general. Por ejemplo, un chip diseñado únicamente para ser usado en un teléfono móvil es un ASIC. Las FPGAs son más lentas, consumen más energía y no son capaces de abarcar sistemas tan complejos como ellos. Sin embargo, frente a los ASIC tienen la ventaja de ser reprogramables. Además, tienen un menor coste de desarrollo y adquisición y su tiempo de desarrollo también es menor.

Ciertos fabricantes desarrollan FPGAs que sólo pueden programarse una vez, colocando sus ventajas e inconvenientes a medio camino entre los ASIC y las FPGA programables [W - FPFGA].

No obstante, la programación de una FPGA está limitada por las características de la tarjeta y de la plataforma. También es necesario un software especial diseñado por el propio fabricante, como por ejemplo, la herramienta ***Xilinx Platform Studio*** diseñado por la compañía Xilinx para cargar la configuración de sus FPGAs [ALT1040].

Normalmente la configuración de una FPGA es volátil, por lo que se debe recargar cuando se le aplica energía o se requiere una función diferente. Generalmente la configuración se almacena en memorias PROM o EEPROM.

Las FPGA utilizan una red de bloques lógicos e interconexiones programados por el usuario que han sido soldados al circuito impreso por el fabricante, permitiendo desempeñar cualquier función lógica necesaria.

Los lenguajes de programación o **HDL (Hardware Description Lenguage)** más utilizados en la programación de FPGAs son VHDL, Verilog y ABEL [Alegsa].

Las FPGAs llevan comercializándose alrededor de 25 años, teniendo cada vez más áreas de aplicación (radioastronomía, bioinformática, criptografía, etc.). En las universidades su uso está muy extendido como una excelente herramienta didáctica [ALT1040].

Los mayores fabricantes de FPGAs son **Xilinx** (www.xilinx.com) y **Altera** (www.altera.com), las cuales tienen una cuota del 80% del mercado [Alegsa].

### ➢ Aplicaciones de las FPGA

Cualquier circuito de aplicación específica se puede implementar en una FPGA si ésta dispone de los recursos necesarios. Algunas de las aplicaciones más frecuentes de las FPGA son los **DSP (procesamiento digital de señales)**, sistemas aeroespaciales y de defensa, sistemas de imágenes en medicina, sistemas de visión para computadoras o bioinformática, entre otras. Se espera que en el futuro se cuente con herramientas de más alto nivel para programar las FPGA, permitiendo mejores resultados no sólo en campos como la medicina o la ingeniería, sino también en otros como el arte o la creatividad en general.

Existe código fuente disponible (bajo licencia GNU GPL) de sistemas como microprocesadores y microcontroladores, filtros, módulos de comunicaciones y memorias, entre otros. Estos códigos se denominan **cores**. Uno de los sitios web donde se puede encontrar una gran cantidad de cores es la página *Opencores* (www.opencores.org).

Una tendencia reciente ha sido combinar los bloques lógicos e interconexiones de los FPGA con microprocesadores y periféricos relacionados para formar un *sistema programable en un chip*. Ejemplo de tales tecnologías híbridas pueden ser encontradas en los dispositivos Virtex-II PRO y Virtex-4 de Xilinx, los cuales incluyen uno o más procesadores PowerPC embebidos junto con la lógica del FPGA. El FPSLIC de Atmel es otro dispositivo similar, el cual usa un procesador AVR en combinación con la arquitectura lógica programable de Atmel. Otra alternativa es hacer uso de núcleos de procesadores implementados mediante la lógica del FPGA. Esos núcleos incluyen los procesadores MicroBlaze y PicoBlaze de Xilinx, Nios y Nios II de Altera, y los procesadores de código abierto LatticeMicro32 y LatticeMicro8.

Muchas FPGA modernas soportan la reconfiguración parcial del sistema, permitiendo que una parte del diseño sea reprogramada, mientras las demás partes siguen funcionando. Este es el principio de la idea de la *computación reconfigurable*, o los *sistemas reconfigurables* [W - FPGA].

## 2.1.2. Descripción de la estructura interna de una FPGA

La arquitectura de un FPGA consiste en cinco elementos programables fundamentales:

- **CLB (bloques lógicos configurables)**: se pueden programar de diversas maneras logrando así una amplia gama de funciones lógica. Cada CLB está compuesto por cuatro slices (figura 1) y estos a su vez contienen las llamadas LUTs (en inglés, *Look up tables*), las cuales son elementos basados en memoria RAM que se pueden usar como biestables o *latches*. Las LUTs pueden tomar la forma de un bloque lógico e implementar multiplexores, o bien utilizarse como elementos de memoria (RAM distribuida) donde cada una tiene una capacidad de hasta 16 bits. También puede utilizarse como un registro de desplazamiento. Las LUTs son el elemento fundamental para la síntesis de funciones lógicas.

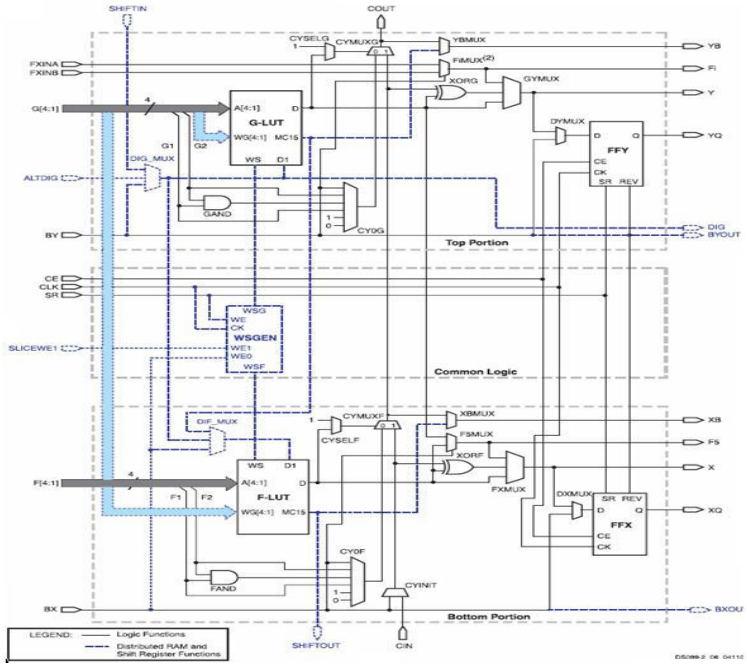


Figura 1. Estructura de un slice.

- **IOB (bloques de entrada-salida):** se encargan del flujo de datos desde y hacia el FPGA a través de los pines del chip. Soportan flujos de datos bidireccionales, operaciones tri-estado, y un total de 24 estándares de señales. Poseen además control digital de impedancias.

- **BRAM o RAM de bloque:** consiste en varios bloques (internos del FPGA) de 18 Kbits. Cada uno se comporta como un chip de memoria de doble puerto. Cada puerto tiene sus propias señales de control para las operaciones de lectura y escritura.

- **Multiplicadores:** son bloques dedicados que efectúan esta operación entre dos números de 18 bits cada uno (figura 2**Figura 2**). A la salida se obtiene un número de 36 bits. Se puede asociar un bloque multiplicador con un bloque de RAM, de manera que se obtiene un multiplicador síncrono con las salidas registradas. La cercanía física de los bloques multiplicadores y los bloques de RAM posibilita esta característica.

Haciendo multiplicadores en cascada es posible lograr la multiplicación de más de dos números e incluso de números de más de 18 bits.

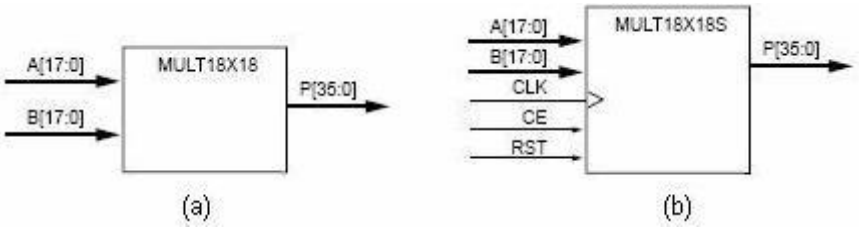


Figura 2. Multiplicador de 18 bits asíncrono. Multiplicador de 18 bits con salidas registradas.

- **DCM (manejador de reloj digital)**: por lo general, los miembros de la familia Spartan 3 poseen cuatro DCMs. Estos elementos están destinados a proveer una señal de reloj de elevada exactitud. Eliminan los cambios de fase en la señal de reloj, así como las desviaciones de esta señal producto de perturbaciones externas, de altas temperaturas u otros efectos. Para esto implementan un DLL (en inglés, *Delay-Locked Loop*). El DLL rastrea las desviaciones de la señal de reloj y a través de una realimentación logra eliminar el error en la señal original. El DCM es capaz de proveer al sistema de un conjunto de señales desfasadas con respecto a la señal de reloj original [MicroC - FPGA].

### 2.1.3. FPGA Spartan 3

En la realización de nuestro proyecto hemos utilizado la **FPGA *Spartan 3A*** de Xilinx, **la cual viene integrada en la plataforma *Spartan-3A FPGA Starter Kit***, también de Xilinx. Esta plataforma es una completa herramienta de desarrollo a la hora crear diseños para la FPGA Spartan 3A [TE].

Se ha optado por el uso de la Spartan3 ya que es la FPGA de referencia mundial a la hora de realizar un proyecto informático y el rendimiento de cualquier otra FPGA siempre es comparado con el de la Spartan 3. Aparte de esto, en el momento de iniciar este proyecto era la única FPGA de la que se disponía. No obstante, la elección habría sido la misma aunque se hubiese dispuesto de otros modelos de FPGA.

En la figura 3 se puede ver una imagen de la plataforma *Spartan 3A Starter Kit*.


Figura 3. Plataforma *Spartan 3A Starter Kit*

- **Características de la plataforma**
  - Es un completo prototipado de FPGA a bajo coste.
  - Soporta dispositivos de la familia Spartan 3A.
  - Dispositivos de Xilinx: Spartan-3A (XC3S700A-FG484) y plataforma Flash (XCF04S-VOG20C).
  - Relojes: oscilador de cristal de 50 MHz, zócalo de 8 pines para conexión de oscilador externo.
  - Memoria: Flash paralela de 32 Mbit, Flash SPI de 16 Mbit, SDRAM DDR2 32Mx16.
  - Interfaces analógicas: convertidor D/A de 4 canales, convertidor A/D de 2 canales, amplificador de señal.
  - Conectores e interfaces: Conector PHY 10/100 para Ethernet, puerto USB JTAG, 2 puertos serie de 9 pines RS-232, puerto para teclado o ratón PS/2-style, conector VGA con capacidad para 4096 colores, ruleta codificadora con pulsador, 4 interruptores deslizantes, 8 LEDs individuales, 4 pulsadores de contacto momentáneo, puertos de conexión de expansión Hirose de 100 pines y 2 conectores de expansión de 6 pines.
  - Display de 16 caractéres con 2 líneas de LCD.

### 2.1.4. Entorno de desarrollo Xilinx Platform Studio

Actualmente el mercado ofrece diversas posibilidades a la hora de utilizar un entorno de desarrollo para las FPGA. Los más utilizados hoy en día son *Xilinx Platform Studio* del fabricante Xilinx y *SOPC Builder* del fabricante Altera. Para nuestro proyecto hemos optado por el uso del primero de ellos, ya que la FPGA de que disponíamos, Spartan3, es también del fabricante Xilinx.

La herramienta ***Xilinx Platform Studio (XPS)*** permite diseñar una infraestructura hardware donde el núcleo del sistema sea un microprocesador. Cuando se habla de un procesador en un desarrollo basado en un FPGA, se habla de un procesador empotrado. O sea, puede ser una de las versiones de MicroBlaze o bien un PowerPC, como se han mencionado antes. XPS soporta una amplia gama de familias de FPGA de Xilinx; así mismo, incluye un número importante de modelos de placas de desarrollo. También es posible encontrar versiones compatibles con los Sistemas Operativos Windows, Solaris y Linux Red Hat.

En una plataforma se integran componentes hardware y software como periféricos (módulos IP mencionados anteriormente), según los requerimientos de la aplicación deseada. Existen controladores de memoria, interfaces de comunicaciones (incluso de alta velocidad como Fast Ethernet), generadores de señal de reloj, módulos de verificación y depuración, etc. Estos componentes se interconectan a través de sus señales de entrada-salida y de buses fundamentalmente. Son altamente configurables presentando una cantidad considerable de parámetros para flexibilizar su operación. Existen drivers para estos componentes que incluyen funciones mediante las cuales el desarrollador puede configurar e interactuar con los mismos. Los drivers pueden incluir funcionalidades como generación de interrupciones, DMA y funciones de auto chequeo.

Una plataforma en XPS se puede comenzar a construir mediante el asistente *Base System Builder*, una aplicación que guía paso a paso al desarrollador para seleccionar el procesador que se usará y algunas características de este como memoria cache, frecuencia de reloj, uso de depurador, entre otras. También puede añadirse un temporizador en esta etapa. Además se pueden seleccionar algunos controladores durante esta aplicación como el GPIO para el uso posterior de recursos de la placa (botones, interruptores, lámparas 7 segmentos, LEDs, entre otros).

El diseño basado en plataforma tiene la ventaja de poder utilizar la misma infraestructura para el diseño de otro prototipo con características y funciones bien diferentes. O sea, se cumple aquí también (como con los módulos IP) el principio de reusabilidad; ahora con la arquitectura hardware. Este principio implica un ahorro considerable de tiempo de ejecución de un proyecto.

Una vez que están incluidos los componentes e interconectados (incluyendo las señales de interrupción), se les asigna a cada uno un espacio de direcciones lógicas. Estas direcciones se pueden asignar manualmente o automáticamente por la herramienta. Se pueden definir parámetros explícitamente para cada componente en el fichero *system.mhs* y algunas condiciones específicas en el fichero *system.ucf* [MicroC - XPS].

## 2.2. TÉCNICAS DE COMPRESIÓN.

**En ciencias de la computación la compresión de datos es la reducción del volumen de datos tratables para representar una determinada información empleando una menor cantidad de espacio**. El proceso de compresión de datos se denomina *compresión*, y al contrario *descompresión*.

**El espacio que ocupa una información codificada** (datos, señal digital, etc.) **sin compresión es el cociente entre la frecuencia de muestreo y la resolución**. Por tanto, cuantos más bits se empleen mayor será el tamaño del archivo. No obstante, la resolución viene impuesta por el sistema digital con que se trabaja y no se puede alterar el número de bits a voluntad; por ello, se utiliza la compresión, para transmitir la misma cantidad de información que ocuparía una gran resolución en un número inferior de bits [W - AdC].

**La compresión de datos es el proceso de transformación de la información de una representación a otra**. En este proceso una pequeña representación de la original o una aproximación cercana a ella permiten poder recuperar la información original. Los procesos de compresión y descompresión suelen denominarse codificación y decodificación. La compresión de datos tiene importantes aplicaciones en áreas como el almacenamiento de datos o la transmisión de los mismos. Además del ahorro de compresión, se incluyen también otros parámetros de interés, como las velocidades de codificación y decodificación, el espacio de memoria requerido, la capacidad para acceder y decodificar archivos parciales y la generación y propagación de errores [MPI2010].

En ciencias de la computación y teoría de la información, el conjunto de la compresión de datos, la codificación de la fuente y la reducción del bit-rate, supone la codificación de información usando menos bits de los que usaría la representación original [W - AdC].

La compresión de datos es útil porque ayuda a reducir el consumo de grandes recursos, tales como espacio en disco duro o el ancho de banda de transmisión. En el lado negativo, los datos comprimidos deben ser descomprimidos para ser usados, y este proceso adicional puede ser perjudicial para algunas aplicaciones. Por ejemplo, el esquema de compresión para vídeos puede requerir un hardware de alto coste, el cual es necesario para que la descompresión sea suficientemente rápida para ver el video mientras es descomprimido (la opción de descomprimir el video completamente antes de verlo puede ser un inconveniente y requiere espacio de almacenamiento para el video descomprimido). Por lo tanto, el diseño de sistemas de compresión de datos implica ventajas y desventajas entre varios factores, incluyendo el grado de compresión, la cantidad de distorsión introducida en la imagen, y los recursos de computación requeridos para comprimir y descomprimir los datos [MPI2010].

La compresión es un caso particular de la codificación, cuya característica principal es que el código resultante tiene menor tamaño que el original. Se basa fundamentalmente en buscar repeticiones en series de datos para después almacenar solo el dato junto al número de veces que se repite. Así, por ejemplo, si en un fichero aparece una secuencia como "AAAAAA", ocupando 6 bytes se podría almacenar simplemente "6A" que ocupa solo 2 bytes, en algoritmo RLE .

En realidad, el proceso es mucho más complejo, ya que raramente se consigue encontrar patrones de repetición tan exactos (salvo en algunas imágenes). Por ello se utilizan diferentes tipos de algoritmos de compresión: por un lado, algunos buscan series largas que luego codifican en formas más breves. Por otro lado, algunos algoritmos, como el algoritmo de Huffman, examinan los caracteres más repetidos para luego codificar de forma más corta los que más se repiten. Finalmente también hay otros, como el LZW, que construyen un diccionario con los patrones encontrados, a los cuales se hace referencia de manera posterior.

La codificación de los bytes pares es otro sencillo algoritmo de compresión muy fácil de entender.

A la hora de hablar de compresión hay que tener presentes dos conceptos:

- **Redundancia**: Datos que son repetitivos o previsibles.
- **Entropía**: La información nueva o esencial que se define como la diferencia entre la cantidad total de datos de un mensaje y su redundancia.

La información que transmiten los datos puede ser de tres tipos:

- **Redundante**: información repetitiva o predecible.
- **Irrelevante**: información que no podemos apreciar y cuya eliminación por tanto no afecta al contenido del mensaje. Por ejemplo, si las frecuencias que es capaz de captar el oído humano están entre 16/20 Hz y 16.000/20.000 Hz, serían irrelevantes aquellas frecuencias que estuvieran por debajo o por encima de estos valores.
- **Básica**: Es la información relevante, que debe ser transmitida para que se pueda reconstruir la señal.

Teniendo en cuenta estos tres tipos de información, se establecen tres tipologías de compresión de la información:

- **Sin pérdidas reales**: Se transmite toda la entropía del mensaje (toda la información básica e irrelevante, pero eliminando la redundante).
- **Subjetivamente sin pérdidas**: Además de eliminar la información redundante se elimina también la irrelevante.
- **Subjetivamente con pérdidas**: se elimina cierta cantidad de información básica, por lo que el mensaje se reconstruirá con errores perceptibles pero tolerables (por ejemplo: la videoconferencia) [W - AdC].

### ➢ Diferencias entre compresión con y sin pérdidas

El objetivo de la compresión es siempre reducir el tamaño de la información, intentando que esta reducción de tamaño no afecte al contenido. No obstante, la reducción de datos puede afectar o no a la calidad de la información, pudiéndose hacer dos tipos de compresión:

- **Compresión sin pérdida**: los datos antes y después de comprimirlos son exactamente iguales. En este caso una mayor compresión solo implica más tiempo de proceso. El bitrate siempre es variable. Se utiliza, por ejemplo, en la compresión de texto y en la de ficheros.
- **Compresión con pérdida**: se pueden eliminar datos para reducir aún más el tamaño de los datos, con lo que se suele reducir la calidad. En la compresión con pérdida el bitrate puede ser constante (CBR) o variable (VBR). Hay que tener en cuenta que una vez realizada la compresión, no se puede obtener la señal original, aunque sí una aproximación cuya semejanza con la original dependerá del tipo de compresión. Se utiliza principalmente en la compresión de imágenes, videos y sonidos [W - AdC].

# 3. ALGORITMO DE COMPRESIÓN

En este capítulo se va a realizar una descripción a alto nivel algoritmo diseñado, tanto para la compresión simple como para la doble. Posteriormente se explicará el algoritmo mediante un ejemplo de funcionamiento.

## 3.1. DESCRIPCIÓN DEL ALGORITMO DE COMPRESIÓN

La realización de este proyecto nace de la idea de comprimir datos representados en coma flotante. Un dato en coma flotante tiene 32 bits divididos en tres campos: signo (1 bit), exponente (8 bits), y mantisa (23 bits). **El objetivo fundamental de este proyecto es realizar un algoritmo que haga una doble compresión individual sin pérdidas de cada uno de los tres campos de datos representados en coma flotante.**

Aunque el objetivo a gran escala son los datos en coma flotante, **el algoritmo implementado se ha realizado para datos de 32 bits que se dividirán en tres campos de 12, 10 y 10 bits**, del 0 al 11, del 12 al 21 y del 22 al 31 respectivamente. No obstante, el algoritmo puede ser reconfigurado aumentar o disminuir tanto el número de campos como el número de bits de los mismos.

### 3.1.1. Algoritmo de compresión simple

Los datos originales de 32 bits son enviados por el microprocesador Microblaze a través de un bus FSL al coprocesador, configurado en lenguaje VHDL. Cada vez que recibe un dato lo primero que hace es dividirlo en tres campos, y además, se asigna un contador de 16 bits a cada uno de los tres campos. Dichos tres campos se utilizarán como datos independientes o *subdatos* y son comparados con el mismo *subdato* del siguiente dato. Cada vez que uno de los *subdatos* se repite, el contador suma 1. Una vez que deja de repetirse, el *subdato* es copiado a su búffer de datos correspondiente. Cuando los buffers de almacenamiento de datos están llenos, su contenido es copiado al buffer de salida correspondiente, también de 32 bits, y son enviados de vuelta al Microblaze. De igual manera, el valor de los contadores es copiado en su propio buffer de almacenamiento respectivo y el contador se resetea. Los contadores son de 16 bits y sus bufferes de almacenamiento son de 32 bits, por lo que cada dos contadores almacenados hay disponible un dato de contador que se copia a un nuevo registro y que posteriormente se someterá a la compresión doble. Finalmente se vacían los buffers de almacenamiento y de salida para poder introducir nuevos datos.

Cuando el Microblaze ha enviado todos los datos al coprocesador, si todos los datos han sido procesados y los buffers de almacenamiento no se han llenado completamente, estos se rellenan con ceros y continúa el proceso, copiándose al buffer de salida, enviándose y vaciándose los buffers.

En el apartado 3.2. se dará una explicación con mayor detalle de cómo se realiza una compresión simple utilizando un ejemplo con datos.

### 3.1.2. Algoritmo de compresión doble

Como ya se ha explicado, cuando dos contadores de la compresión simple son almacenados se dispone de un dato para realizar su compresión doble.

El funcionamiento de esta segunda compresión es prácticamente similar al de la primera, con la única diferencia de que los datos se procesan enteros sin dividirlos en campos. Los datos se van recibiendo y se comparan con el anterior, de manera que cada vez que se repita un dato un contador secundario de 16 bits suma 1. Cuando un dato deja de repetirse dicho dato es copiado directamente al buffer de salida y es enviado de vuelta al Microblaze. En este caso, como se comparan los datos completos, no hace falta almacenarlos en búffers, sino que se envían directamente, ya que el tamaño de los datos que se envían por FSL es de 32 bits.

Por otro lado y de igual manera que en la primera compresión, los contadores secundarios se copian al buffer de almacenamiento, hasta que se llena. Entonces se copia su contenido al búffer de salida y se envía al Microblaze. Al igual que en la primera compresión, como los contadores secundarios son de 16 bits y el buffer es de 32, cada dos contadores copiados se envía un dato de contador. Una vez que se envía un dato, el búffer se vacía, preparándose para almacenar nuevos datos. Finalmente, si ya se han recibido todos los datos y un búffer está incompleto, se rellena con ceros y se envía.

En el apartado 3.2. se dará una explicación con mayor detalle de cómo se realiza la compresión doble utilizando un ejemplo con datos.

## 3.2. EJEMPLO DE FUNCIONAMIENTO

A continuación mostraremos un pequeño ejemplo de cómo funcionaría todo el algoritmo de compresión doble, pero utilizando tan sólo datos originales de 10 bits que primero serán divididos en dos partes de 5 bits. Los búffers de almacenamiento serán de 16 bits en lugar de ser de 32 bits y los contadores serán de 8 bits en lugar de ser de 16 bits. Supongamos que desde MicroBlaze se envían los siguientes datos al coprocesador.

| Parte 1 (5 bits) | Parte 2 (5 bits) |
|---|---|
| 11111 | 11111 |
| 11111 | 11111 |
| 11111 | 10001 |
| 11111 | 10001 |
| 11111 | 11111 |
| 11111 | 11111 |
| 11111 | 10001 |
| 11111 | 10001 |
| 10001 | 11111 |
| 10001 | 11111 |
| 11111 | 10001 |
| 10001 | 10001 |
| 11011 | 11011 |
| 11011 | 11011 |

**Parte 1**

- *Compresión simple*

Búffer1: 11111/10001/11111/1/0001
0001/11011/0000000

Los datos se van copiando al búffer. Cuando el buffer se llena con 16 bits y un dato se queda a medias, se envía el buffer completo y una vez se vacía, se empiezan copiando los bits restantes del dato (color azul). Luego siguen copiándose el resto de datos de forma normal. Cuando no quedan más se rellena con ceros (color verde). Se envían a MicroBlaze.

Contador 1: 00000111/00000001
00000000/00000000
00000001/00000000

Los contadores son en binario. Hay 5 datos, el primero se repite 7 veces (“111” en binario), el segundo una vez (‘1’), el tercero y el cuarto ninguna, por lo que se copian ceros, y el último se repite una vez (‘1’). Como no quedan más datos se rellena con ceros.

- *Compresión doble*

Recibe los datos de contador 1.

Búffer de contador1: 0000011100000001
0000000000000000
0000000100000000

Los datos recibidos se copian enteros al buffer de salida y se envían a MicroBlaze.

Contador de contador1: 00000000/00000000
00000000/00000000

Recibe 3 datos y ninguno se repite por lo que se copian 3 ceros. Luego se rellena con ceros el último dato (color verde). Se envían al Microblaze.

**Parte 2**

- *Compresión simple*

Búffer 2: 11111/10001/11111/1/0001
0001/11111/10001/11/011
011/0000000000000

Funciona igual que la parte 1, se van copiando los datos y se rellena con ceros (color verde). Se envían a MicroBlaze.

Contador 2: 00000001/00000001
00000001/00000001
00000001/00000001
00000001/00000000

Todos los datos se repiten una vez, por lo que el contador vale '1'. Como el último dato está incompleto se rellena con ceros (color verde).

- *Compresión doble*

Recibe los datos de contador 2.

Búffer de contador2: 0000000100000001
0000000100000000

Los datos recibidos se copian enteros al buffer de salida y se envían a MicroBlaze.

Contador de contador2: 00000010/00000000

El primer dato se repite 2 veces ("10" en binario) y el último dato no se repite ninguna vez, por lo que se copia un cero y se llena con 0 por la derecha (color verde). Se envía al Microblaze.

# 4. DISEÑO HARDWARE

En este capítulo se va a realizar en primer lugar una breve explicación de las diferentes arquitecturas diseñada, así como los motivos de nuestra elección final para realizar el proyecto. Posteriormente se dará una descripción a alto nivel de la arquitectura elegida. Después se dará una breve descripción del microprocesador MicroBlaze y del bus FSL y se entrará a explicar en detalle el algoritmo hardware realizado en lenguaje VHDL. Finalmente se explicará la interfaz software realizada en código C.

## 4.1. ARQUITECTURAS DISEÑADAS

**La idea original de este proyecto era realizar únicamente un algoritmo de compresión simple** y a tal efecto se creó una arquitectura completamente funcional que puede estudiarse en el apartado 4.1.1., compuesta por un microprocesador MicroBlaze y un coprocesador. **Posteriormente se estudió la posibilidad de ampliar el algoritmo para realizar una doble compresión y viendo la viabilidad del proceso se decidió realizarlo**.

Para esta doble compresión en un primer momento **se ideó una arquitectura formada por cuatro coprocesadores comunicados entre sí y con el MicroBlaze a través de FSLs**. Con la base de la compresión simple, esta sería realizada en un coprocesador *primario* y la compresión doble sería realizada en los tres coprocesadores restantes, llamados *secundarios*.

**Sin embargo**, tras realizar numerosas pruebas y modificaciones tanto de arquitectura como de algoritmo VHDL, **no se han obtenido resultados satisfactorios por lo que se decidió desechar el modelo**. En las diferentes pruebas se trató de encontrar la causa de la disfunción, y aunque no podemos afirmar a ciencia cierta cuál es la causa, **creemos que existe un problema de sincronismo** entre el *primario* y los *secundarios*. No obstante, consideramos interesante la inclusión de esta arquitectura en el informe de este proyecto, como puede verse en el apartado 4.1.2., para que en el futuro pueda ser estudiado e implementado de forma correcta.

Como se ha explicado en el párrafo anterior, ante la idea de que el problema de sincronismo entre coprocesadores se retornó a la idea de **crear una arquitectura con sólo un coprocesador donde se realizase tanto la compresión simple como la doble**. Tras las pruebas pertinentes, el funcionamiento es plenamente correcto, por lo que **es el modelo final elegido para realizar este proyecto de compresión de datos**. En el apartado 4.1.3. puede estudiarse más a fondo su arquitectura.

### 4.1.1. Arquitectura de compresión simple.

Como ya se ha explicado, nuestro primer modelo se componía de **dos bloques, el procesador MicroBlaze y un coprocesador**, llamado *Compresor*. Como puede verse en la figura 4, MicroBlaze tiene un FSL de salida y 6 FSLs de entrada, y el coprocesador al revés, 1 de entrada y 6 de salida.

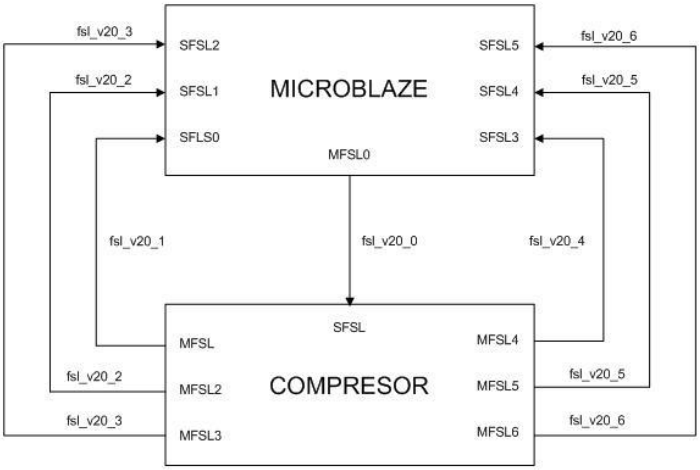


Figura 4. Arquitectura de modelo de compresión simple.

Los datos de entrada de 32 bits son recibidos en el *compresor* a través de un proceso llamado **"Recibir"**. Tres máquinas de estado llamadas **"MDE"** se encargan de dividir los datos en tres campos de 12, 10 y 10 bits y los comparan con los campos con el del dato siguiente. Por cada repetición de un campo un contador sumaba '1'. Cada máquina de estados tiene asociado un proceso **"Buffer"**, encargado de almacenar los datos y enviarlos a MicroBlaze, y un proceso **"Cuenta"**, que almacena los contadores y también los reenvía a MicroBlaze. Como hay tres procesos "Buffer" y tres procesos "Cuenta" el *compresor* tiene 6 salidas. Un esquema del código VHDL se puede ver en la figura 5.

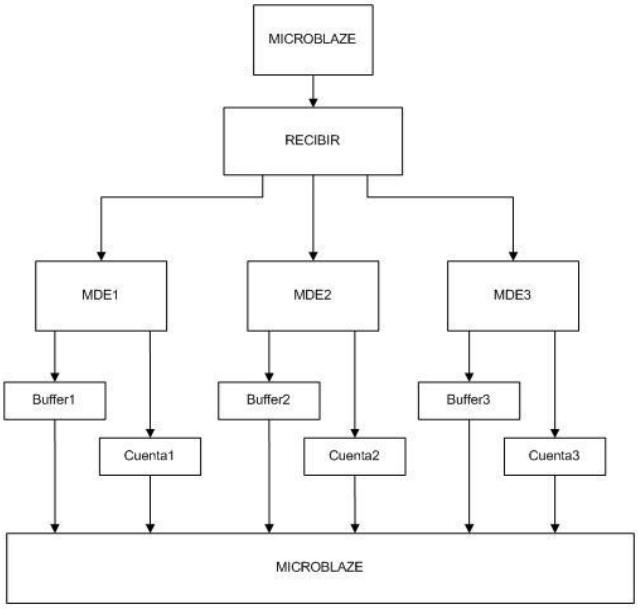


Figura 5. Esquema de código VHDL de modelo de compresión simple

### 4.1.2. Arquitectura de compresión doble con cuatro coprocesadores.

Para este segundo modelo la arquitectura se divide en **cinco bloques, MicroBlaze, un coprocesador *primario* y 3 coprocesadores *secundarios***, como puede verse en la figura 6. MicroBlaze tiene un FSL de salida y 9 de entrada, ya que en este caso, lo contadores de *primario* no son enviados a los *secundarios* donde son tratados como nuevos datos y a los que se añade un nuevo contador. De ahí que MicroBlaze tenga 9 entradas, 3 para datos originales, 3 para contadores de *primario* y 3 para contadores de *secundarios*.

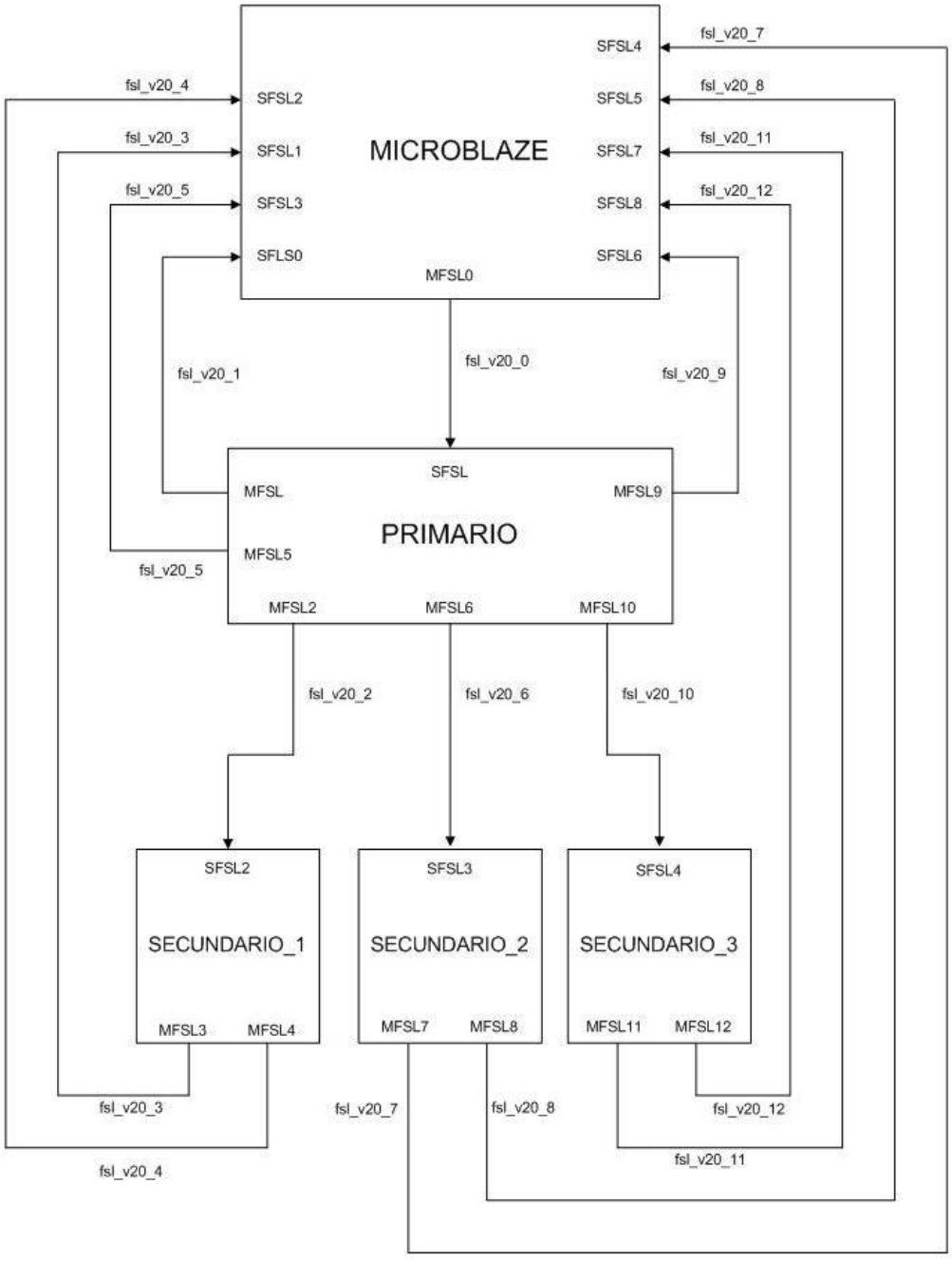


Figura 6. Arquitectura de modelo de compresión doble con cuatro coprocesadores.

En este modelo, el código VHDL de primario es exactamente igual que el del coprocesador del modelo simple, con la diferencia que los datos de los contadores se envían a los secundarios. Los secundarios tienen a su vez un código bastante similar, sólo que en este caso sólo hay una máquina de estados "**MDE**", un proceso "**Buffer**" y un proceso "**Cuenta**". No obstante, el funcionamiento es muy similar, excepto por un detalle. En primario se indica el número de datos que va a recibir, mientras que a los secundarios se les indica que han recibido todos los datos cuando en primario se activa la señal "**FSL_M_Control**", que se corresponde con la señal "**FSL_S_Control**" del secundario respectivo. En la figura 7**Figura 7** se puede ver un esquema del código VHDL

Como ya se ha dicho, este modelo en ciertas situaciones tiene un comportamiento incorrecto, creemos que debido a la falta de sincronización entre los coprocesadores y los buses FSL y por ello se ha desechado.

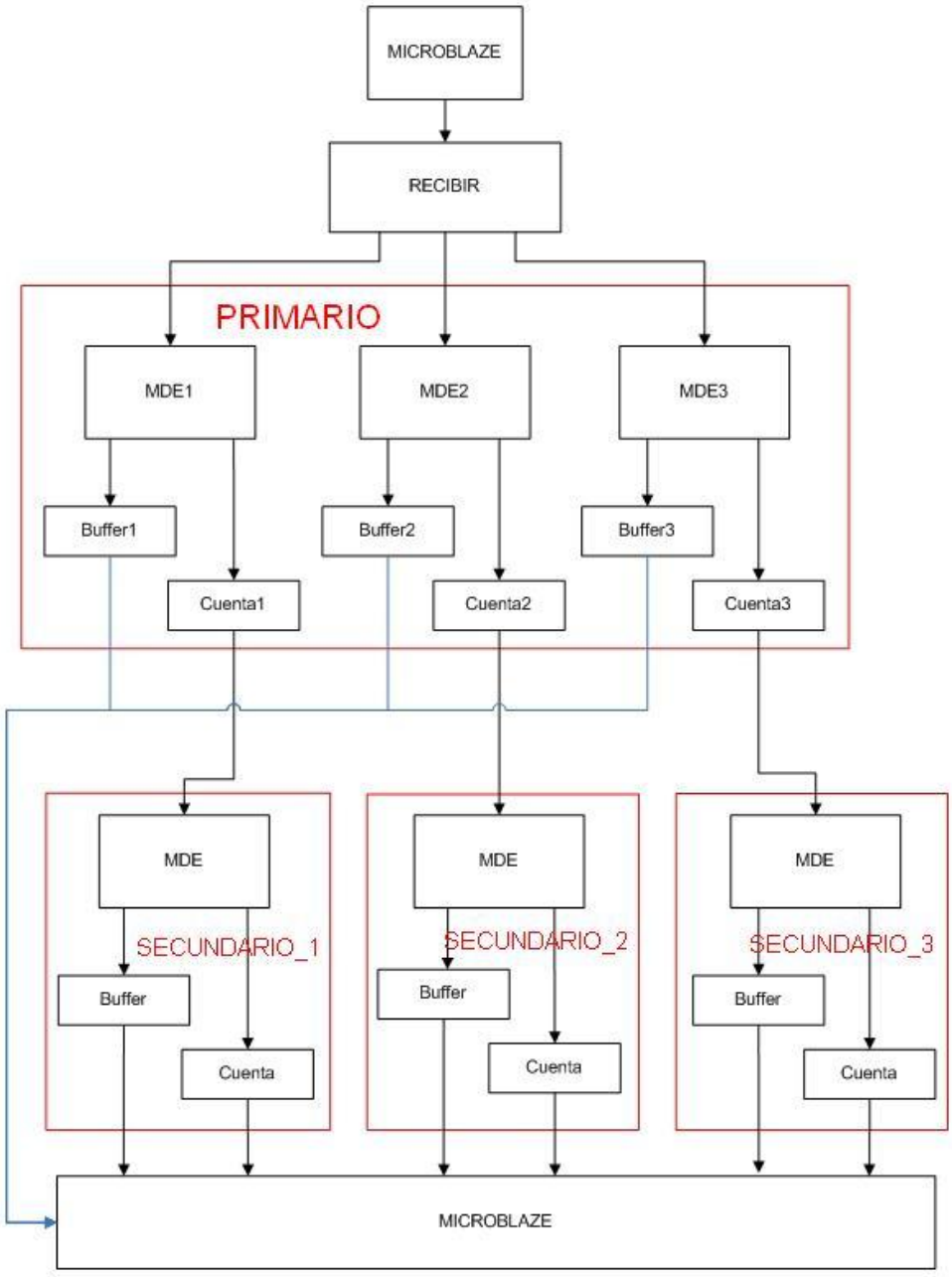


Figura 7. Esquema de código VHDL de modelo de compresión doble con cuatro coprocesadores.

### 4.1.3. Arquitectura de compresión doble con un coprocesador.

Como ya se ha explicado, esta ha sido la arquitectura final elegida para realizar nuestro proyecto. Se divide en **dos bloques, el procesador MicroBlaze y un coprocesador** llamado *Compresor.* En este caso, MicroBlaze tiene un FSL de salida y nueve de entrada, al igual que en el modelo de cuatro coprocesadores, y el coprocesador tiene uno de entrada y nueve de salida, tres de datos, tres de compresión simple de contadores y tres de compresión doble de contadores, como puede verse en la figura 8.

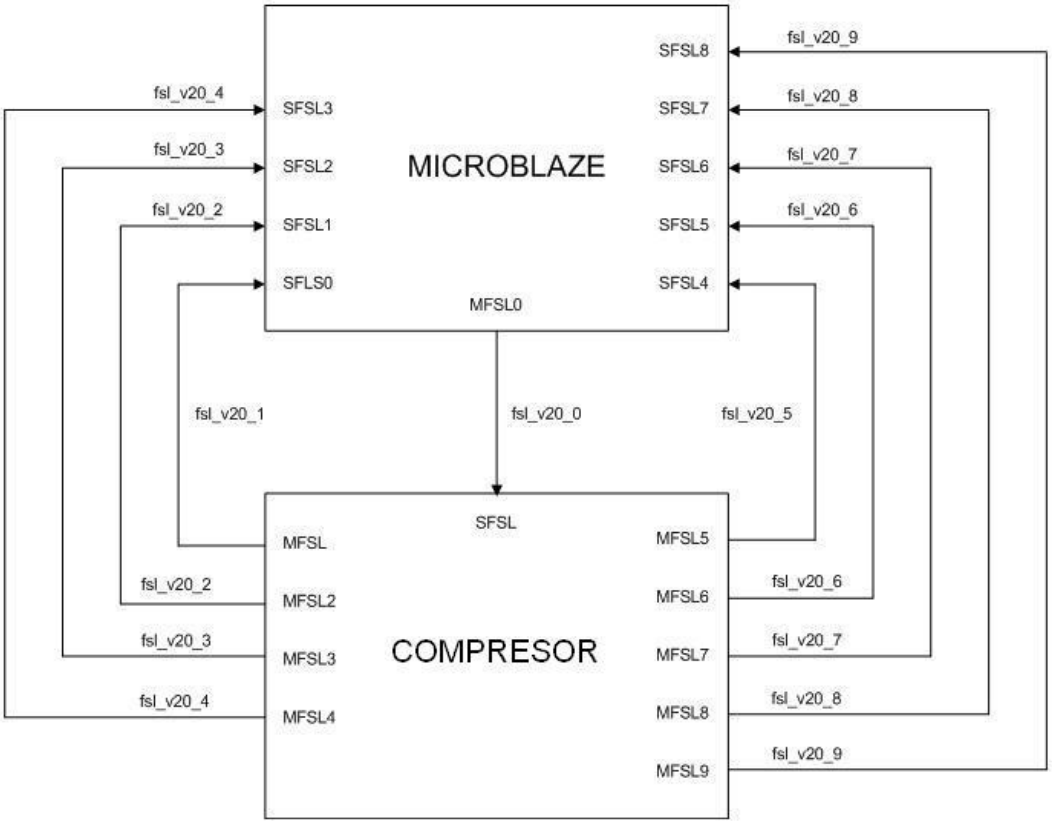


Figura 8. Arquitectura de modelo de compresión doble con un solo coprocesador.

Al ser este nuestro modelo final utilizado, en los apartados 4.2 y 4.3.3. se aportarán explicaciones con mucho mayor detalle.

## 4.2. DESCRIPCIÓN DE LAS ARQUITECTURAS EN ALTO NIVEL

En este apartado se pretende dar una idea general del funcionamiento completo del algoritmo VHDL realizado en el coprocesador.

### 4.2.1. Visión global del algoritmo hardware

En la figura 9 se puede ver un esquema de la distribución de máquinas de estado y procesos del código VHDL realizado.

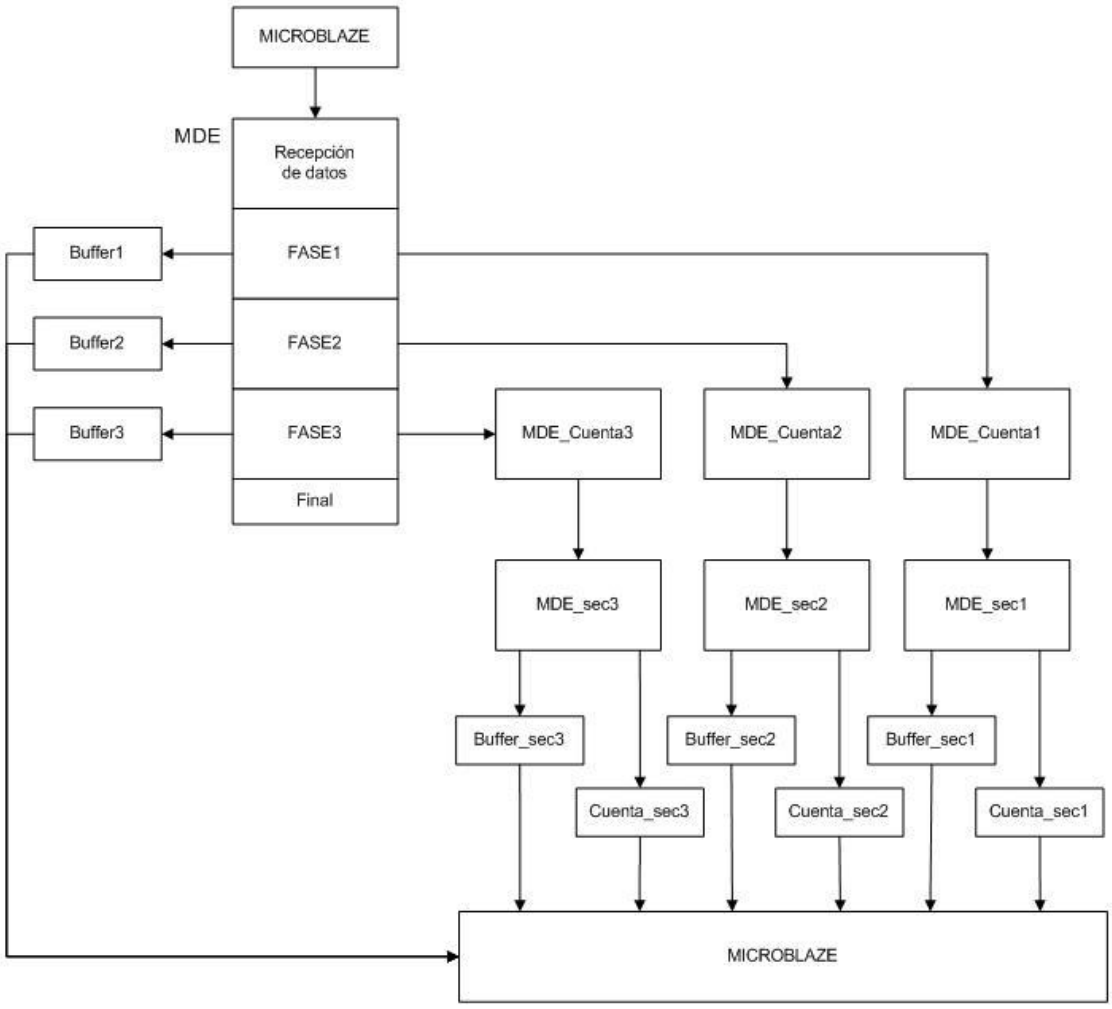


Figura 9. Esquema de código VHDL de modelo de compresión doble con un coprocesador.

El algoritmo comienza con una máquina de estados principal llamada **MDE,** encargada de hacer la compresión simple**. Se divide en cinco partes**. La primera es la **recepción de datos**. Las partes 2,3 y 4 las hemos denominado ***Fase1***, ***Fase2*** **y** ***Fase3***. Los datos recibidos desde MicroBlaze se dividen en 3 campos de 12, 10 y 10 bits respectivamente. Cada *fase* se encarga de uno de los campos. Las tres *fases* están formadas por estados de la "MDE" y tienen el mismo funcionamiento. Se encargan de hacer las comparaciones de los datos y de controlar

los contadores. Por último, la parte 5 nos indica el fin de la compresión y el retorno al inicio para resetear la "MDE" y que esté lista para recibir una nueva trama de datos.

Los procesos **Buffer1, Buffer2** y **Buffer3** son los encargados del almacenamiento y envío al MicroBlaze de los datos comprimidos en la compresión simple. Tienen un funcionamiento similar y cada uno está sincronizado con una *fase*.

**MDE_Cuenta1, MDE_Cuenta2** y **MDE_Cuenta3** son tres máquinas de estados encargadas de almacenar los contadores de la compresión simple y prepararlos para que se les aplique la compresión doble. Igualmente su funcionamiento es similar y también están sincronizados con su *fase* correspondiente.

**MDE_sec1, MDE_sec2** y **MDE_sec3** son tres máquinas de estados encargadas de realizar la compresión doble de los contadores almacenados en las "MDE_Cuenta". El funcionamiento de las 3 es igual y a su vez también es muy parecido al de la "MDE".

Los procesos **Buffer_sec1, Buffer_sec2** y **Buffer_sec3** se encargan del envío al MicroBlaze de los datos de la compresión doble. La arquitectura es igual para los tres y tienen un funcionamiento muy similar a sus homónimos de la compresión simple.

Por último están los procesos **Cuenta_sec1, Cuenta_sec2** y **Cuenta_sec3**. Se encargan del almacenamiento y del posterior envío al MicroBlaze de los contadores de la compresión doble.

### 4.2.2. Compresión simple

En la compresión simple el coprocesador recibe datos desde el MicroBlaze y luego envía datos de vuelta al MicroBlaze desde tres salidas diferentes. También almacena contadores sobre los que se aplicará la compresión doble. Como ya se explicó en el apartado anterior, la encargada de la compresión simple es la "MDE" principal, compuesta de tres *fases*.

La "MDE" comienza recibiendo el número de datos que va a enviar el MicroBlaze. Después, cada vez que recibe un dato aumenta en 1 la cuenta de datos hasta que es igual al número de datos, lo que significa que ya los ha recibido todos y activa la señal de todos los datos recibidos.

Cuando la "MDE" recibe un dato, lo primero que haces es dividirlo en tres partes, que llamaremos subdatos, y que son independientes entre sí. Después, contabiliza las veces que se repite cada subdato. Según se van recibiendo nuevos datos, se va comparando cada subdato con el anterior. Por cada repetición el contador correspondiente sumará uno. Cuando un subdato deja de repetirse se entra a la *fase* correspondiente. Durante una *fase*, el subdato es copiado al búffer de almacenamiento. También, de igual manera, el valor del contador se copia a otro búffer de almacenamiento y el contador es reseteado para comenzar a contar de nuevo. Las *fases* también se encargan de decidir si la "MDE", va a la siguiente *fase*, de si vuelve a leer datos o de si ya ha terminado la compresión simple y va al estado final.

Los procesos "Búffer" tienen la tarea de copiar los subdatos en el búffer de almacenamiento correspondiente cuando dejan de repetirse. También es el encargado de que cuando el búffer de almacenamiento se llena, copia su contenido en el búffer de salida, permitiendo borrarlo, y además envía el búffer de salida al MicroBlaze.

Las "MDE_Cuenta" tienen una tarea similar a los procesos "Búffer". Cuando un subdato deja de repetirse copia el contador correspondiente a su búffer de contador. Cuando se llena, concretamente cuando se copian 2 contadores, se activa la señal que indica a la "MDE_sec" correspondiente que hay un dato disponible para realizar la compresión doble. Una vez que la

"MDE_sec" ha leído el dato la "MDE_Cuenta" resetea el buffer de almacenamiento de contador y se prepara para almacenar nuevos contadores.

En el capítulo 4.3.3. *Definición de cada bloque funcional* se dará una explicación más detallada del funcionamiento de las máquinas de estados y de los procesos.

### 4.2.3. Compresión doble

La compresión doble es llevada a cabo por las "MDE_sec". Estas van recibiendo los contadores comprimidos por las "MDE_Cuenta" y los van comparando entre sí. Su funcionamiento es similar al de "MDE", con la diferencia de que los datos se comparan al completo y no se dividen en partes. También se asigna un contador secundario de 16 bits a cada dato.

La "MDE_sec" va comparando cada dato con el anterior. Si un dato se repite el contador suma 1. Cuando ese dato deja de repetirse éste es copiado a su búffer de almacenamiento y su contador al búffer de contador.

Cuando un dato deja de repetirse el proceso "Buffer_sec" copia el dato en el buffer de salida y lo envía al MicroBlaze.

De igual manera, cuando un dato deja de repetirse el proceso "Cuenta_sec" copia el contador secundario al buffer de almacenamiento. Cuando el buffer de llena se envía al MicroBlaze.

En total se envían al MicroBlaze seis tipos de datos. Por un lado, los tres buffers de datos de los tres procesos "Buffer_sec", y por otro, los tres buffers de contadores secundarios correspondientes a los tres procesos "Cuenta_sec".

En el capítulo 4.3.3. *Definición de cada bloque funcional* se dará una explicación más detallada del funcionamiento de Las máquinas de estados y de los procesos.

## 4.3. DESCRIPCIÓN DETALLADA DEL ALGORITMO HARDWARE

En este apartado se va a realizar una pequeña descripción del microprocesador MicroBlaze así como de la conexión a través del bus FSL. Finalmente también se hará a nivel de señal una explicación bien detallada de todo el código VHDL.

### 4.3.1. Definición de MicroBlaze

**MicroBlaze** es un procesador RISC (Set de Componentes de Instrucciones Reducidas) de 32 bits, desarrollado por Xilinx. Como microprocesador, Microblaze está implementado completamente en la memoria de propósito general y de lógica de fabricación de las FPGAs de Xilinx. El mismo es compatible con las familias Spartan y Virtex. MicroBlaze cuenta con una arquitectura Harvard con buses de 32 bits separados para acceso a datos e instrucciones y permite el acceso a estos recursos a través de memoria cache. Actualmente cuenta con interfaces para los buses OPB, LMB y PLB. Además cuenta con un protocolo XCL para el acceso a la memoria cache de datos e instrucciones cuando está habilitado este recurso.

Con muy pocas excepciones Microblaze puede emitir una nueva instrucción en cada ciclo de reloj, manteniendo el rendimiento de un solo ciclo en la mayoría de las circunstancias. El repertorio de instrucciones de MicroBlaze es el típico repertorio de un procesador RISC. En total cuenta con 87 instrucciones diferentes. Se incluyen operaciones de multiplicación hardware a partir de la familia Spartan 3. Este procesador cuenta con 32 registros de 32 bits cada uno, de los cuales 14 son de propósito general. Cuenta con cinco etapas máximo para la ejecución de instrucciones (arquitectura pipeline), con lo cual es capaz de tener lista una instrucción completa en cada ciclo de reloj [MicroC - MB].

Microblaze tiene un sistema de interconexión versátil para soportar una gran variedad de aplicaciones embebidas. El bus primario de E/S del Microblaze, llamado CoreConnect PLB Bus, es un bus de transacción basado en un sistema de memoria tradicional mapeado con capacidad maestro/esclavo. Una versión más moderna de Microblaze, soportada tanto en implementaciones de Spartan 6 como de Virtex 6, así como en la serie 7, soporta la especificación AXI. La mayoría de suministros proporcionados por el vendedor y las IP de terceros interactúan con la PLB (o a través de un PLB a un puente de bus OPB). Para acceder a la memoria local (BRAM) el Microblaze utiliza un bus LMB dedicado, el cual reduce la carga en otros buses. Los coprocesadores definidos por el usuario son soportados a través de una conexión dedicada tipo FIFO llamada FSL (“Fast simple link”). La interfaz del coprocesador puede acelerar computacionalmente algoritmos complejos mediante la descarga de partes o de la completa computación en un módulo hardware diseñado por el usuario y conectado a través de este bus [W - MB].

### 4.3.2. Definición de FSL

**Es un canal bus de comunicación punto a punto unidireccional utilizado para realizar una rápida comunicación entre dos elementos cualesquiera de diseño de una FPGA cuando se implementa la interfaz de un bus FSL**. La interfaz del FSL está disponible en el procesador Microblaze de Xilinx. Las interfaces son utilizadas para transferir datos hacia y desde el banco de registros en el procesador hardware que se ejecuta en la FPGA [Xilinx].

- **Características**
  - El FSL implementa una comunicación unidireccional punto a punto basada en un FIFO.
  - Proporciona un mecanismo para un mecanismo de comunicación no compartido y sin árbitro. Este se puede utilizar para una rápida transferencia de datos entre el maestro y el esclavo implementando la interface del FSL.
  - Proporciona un bit de control extra para controlar los datos transferidos. Este bit de control puede ser usado por la interfaz del esclavo para múltiples propósitos. Por ejemplo, descodificar la palabra que se transmite como palabra de control o usar el bit para indicar el comienzo o el fin de una transmisión de datos.
  - La profundidad del FIFO va desde un bit hasta 8 Kbits.
  - Soporta tanto el modo síncrono como asíncrono del FIFO. Esto permite al maestro y al esclavo trabajar a diferentes frecuencias.
  - Tiene soporte para el SRL16 y dos puertos LUT RAM o bloque de RAM basados en la implementación del FIFO.
- **Descripción funcional**

  En la figura 10 se muestra el diagrama de bloques del Bus FSL V20.

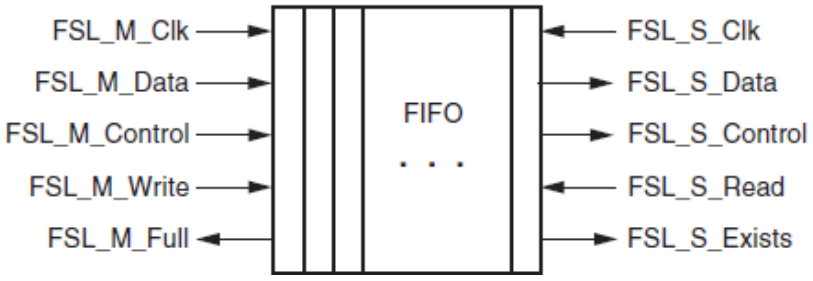


Figura 10. Diagrama de bloques del Bus FSL V20.

- **Mecanismos de interconexión de periféricos con FSL**

Los periféricos del FSL pueden ser creados como maestro o esclavo del bus FSL. Un periférico conectado a los puertos del maestro de un bus FSL envía las señales de datos y control al FSL. Todos los periféricos que actúan como maestro en el bus FSL deben crear una interface de bus del tipo MASTER para un bus FSL estándar en el archivo MPD (Microprocessor Peripheral Description). Además, el periférico debe formar conexiones por defecto para todos los puertos del maestro y debe seguir los requisitos de tiempo de operación de escritura tal como se explica en la figura 11.

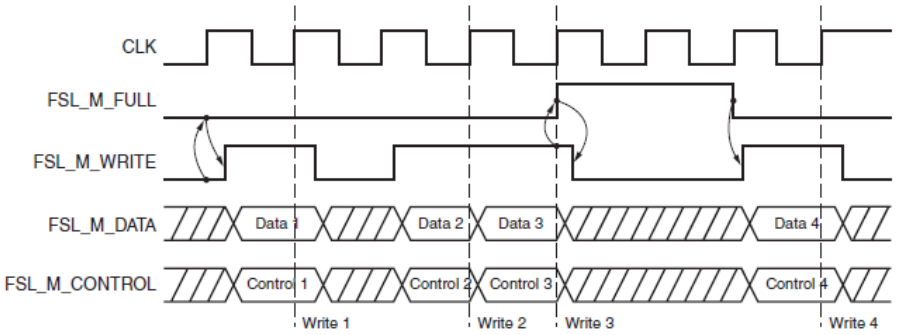


Figura 11. Operación de escritura en FSL.

Un periférico conectado a los puertos del esclavo del bus lee y recibe las señales de datos y control desde el FSL. Todos los periféricos configurados como esclavos deben crear una interface de bus del tipo SLAVE para un bus FSL estándar en el archivo MPD. Además, el periférico debe formar conexiones por defecto para todos los puertos del esclavo y debe seguir los requisitos de tiempo de operación de lectura tal como se explica en la figura 12 [Xilinx].

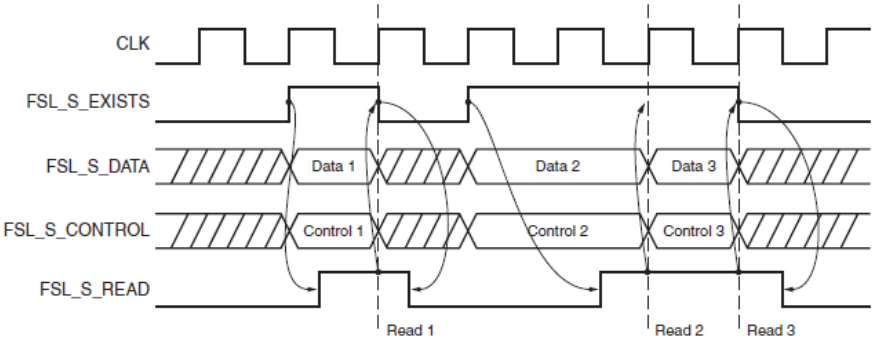


Figura 12. Operación de lectura en FSL.

## ➢ Interfaces del FSL de MicroBlaze Soft Processor

El procesador MicroBlaze tiene una lista de interfaces de FSL. La lectura y escritura de instrucciones del Microblaze se puede usar para transferir el contenido de un registro del MicroBlaze al bus FSL y viceversa. La configuración del FSL se puede usar en conjunto con cualquiera de las otras configuraciones de bus [Xilinx].

### 4.3.3. Definición de cada bloque funcional

En el apartado 4.2. se realizó una explicación a alto nivel del algoritmo VHDL. En este apartado se pretende dar una explicación mucho más detallada de todos los procesos, registros, entradas y salidas, etc. Existen también una serie de puertos de entrada y salida que no se han incluido en la explicación porque no se utilizan en nuestro algoritmo, pero el código exige sean incluidos en el código.

➢ **Puertos de entrada y salida**

- **FSL_Clk**: Entrada de reloj. 50 MHz.
- **FSL_Rst**: 1 bit. Entrada. Señal de Reset.

- **FSL_S_Read**: 1 bit. Salida. Se activa cuando se ha leído un dato de entrada y se pide el siguiente.
- **FSL_S_Data**: 32 bits. Entrada. Datos de entrada enviados desde MicroBlaze.
- **FSL_S_Exists**: 1 bit. Entrada. Cuando está activa indica que hay un dato esperando a ser leído en el FSL de entrada.

- **FSL_M_Write**: 1 bit. Salida. Cuando se activa se envía FSL_M_Data al MicroBlaze.
- **FSL_M_Data**: 32 bits. Salida. En él se copian los datos de "Buff1" ya codificados listos para ser enviados.
- **FSL_M_Full**: 1 bit. Entrada. Se activa si el FSL de salida está lleno indicando que no se pueden enviar datos. Si está desactivada permite el envío de datos.
- **FSL_M_Write2**: 1 bit. Salida. Cuando se activa se envía FSL_M_Data2 a Microblaze.
- **FSL_M_Data2**: 32 bits. Salida. En él se copian los datos de "registro_sec1" listos para ser enviados.
- **FSL_M_Full2**: 1 bit. Entrada. Se activa si el FSL de salida está lleno indicando que no se pueden enviar datos. Si está desactivada permite el envío de datos.
- **FSL_M_Write3**: 1 bit. Salida. Cuando se activa se envía FSL_M_Data3 al MicroBlaze.
- **FSL_M_Data3**: 32 bits. Salida. En él se copian los datos de "buff_cont_sec1" ya codificados listos para ser enviados.
- **FSL_M_Full3**: 1 bit. Entrada. Se activa si el FSL de salida está lleno indicando que no se pueden enviar datos. Si está desactivada permite el envío de datos.

- **FSL_M_Write4**: 1 bit. Salida. Cuando se activa se envía FSL_M_Data4 al MicroBlaze.
- **FSL_M_Data4**: 32 bits. Salida. En él se copian los datos de "Buff2" ya codificados listos para ser enviados.
- **FSL_M_Full4**: 1 bit. Entrada. Se activa si el FSL de salida está lleno indicando que no se pueden enviar datos. Si está desactivada permite el envío de datos.
- **FSL_M_Write5**: 1 bit. Salida. Cuando se activa se envía FSL_M_Data5 a Microblaze.
- **FSL_M_Data5**: 32 bits. Salida. En él se copian los datos de "registro_sec2" listos para ser enviados.
- **FSL_M_Full5**: 1 bit. Entrada. Se activa si el FSL de salida está lleno indicando que no se pueden enviar datos. Si está desactivada permite el envío de datos.
- **FSL_M_Write6**: 1 bit. Salida. Cuando se activa se envía FSL_M_Data6 al MicroBlaze.
- **FSL_M_Data6**: 32 bits. Salida. En él se copian los datos de "buff_cont_sec2" ya codificados listos para ser enviados.
- **FSL_M_Full6**: 1 bit. Entrada. Se activa si el FSL de salida está lleno indicando que no se pueden enviar datos. Si está desactivada permite el envío de datos.

- **FSL_M_Write7**: 1 bit. Salida. Cuando se activa se envía FSL_M_Data7 al MicroBlaze.
- **FSL_M_Data7**: 32 bits. Salida. En él se copian los datos de "Buff3" ya codificados listos para ser enviados.
- **FSL_M_Full7**: 1 bit. Entrada. Se activa si el FSL de salida está lleno indicando que no se pueden enviar datos. Si está desactivada permite el envío de datos.
- **FSL_M_Write8**: 1 bit. Salida. Cuando se activa se envía FSL_M_Data6 a Microblaze.
- **FSL_M_Data8**: 32 bits. Salida. En él se copian los datos de "registro_sec3" listos para ser enviados.
- **FSL_M_Full8**: 1 bit. Entrada. Se activa si el FSL de salida está lleno indicando que no se pueden enviar datos. Si está desactivada permite el envío de datos.
- **FSL_M_Write9**: 1 bit. Salida. Cuando se activa se envía FSL_M_Data9 al MicroBlaze.
- **FSL_M_Data9**: 32 bits. Salida. En él se copian los datos de "buff_cont_sec3" ya codificados listos para ser enviados.
- **FSL_M_Full9**: 1 bit. Entrada. Se activa si el FSL de salida está lleno indicando que no se pueden enviar datos. Si está desactivada permite el envío de datos.

### ➢ Señales y registros internos

- **num_datos**: 32 bits. Almacena el número de datos que van a enviar MicroBlaze.
- **cuenta_datos**. 32 bits. Aumenta en '1' cada vez que se recibe un dato. Cuando num_datos y cuenta_datos son iguales significa que ya se han recibido todos los datos.
- **registro1**. 11 bits. Recibe los bits de 0 a 11 del primer dato recibido, para luego ser comparados.
- **input1**. 11 bits. Recibe los bits de 0 a 11 del nuevo dato recibido y se compara con registro1. Si son iguales el contador suma 1. Si no, registro1 se copia en buff1 y después input1 se copia en registro1.
- **registro2**. 9 bits. Recibe los bits de 12 a 21 del primer dato recibido, para luego ser comparados.
- **input2**. 9 bits. Recibe los bits de 12 a 21 del nuevo dato recibido y se compara con registro2. Mismo funcionamiento que input1.
- **registro3**. 9 bits. Almacena los bits de 22 a 31 del primer dato recibido, para luego ser comparados.
- **input3**. 9 bits. Recibe los bits de 22 a 31 del nuevo dato recibido y se compara con registro3. Mismo funcionamiento que input1.
- **cont1, cont2, cont3**: 16 bits. Contadores. Cuando input es igual a registro el contador correspondiente suma '1'. Cuando son diferentes se copia el contador en su buff_cont correspondiente y después se pone a 0 de nuevo.
- **buff1**: 43 bits. Almacena los datos de registro1. Cada vez que buff1 se llena sus primeros 32 bits se copian en el buffer de salida, FSL_M_Data correspondiente. Después los bits de buff1 se desplazan a la izquierda 32 posiciones y se rellena con ceros por la derecha.
- **buff2**: 41 bits. Almacena los datos de registro2. Mismo funcionamiento que buff1.
- **buff3**: 41 bits. Almacena los datos de registro3. Mismo funcionamiento que buff1.
- **pos_buff1, pos_buff2, pos_buff3**: números naturales desde 0 hasta 43, 41 y 41 respectivamente. Indican en qué posición del buff correspondiente se copian los datos de registro. Cada vez que un registro se copia en su buff, pos_buff1 aumenta en 12 y pos_buff2 y pos_buff3 aumentan en 10. Y cuando un pos_buff es mayor que 31 se copian los 32 primeros bits de su buff correspondiente en el buffer de salida y al pos_buff se le restan 32.
- **buff_cont1, buff_cont2, buff_cont3**: 32 bits. Almacenan respectivamente los valores de cont1, cont2 y cont3. Como los contadores son de 16 bits, cada vez que se copian dos

contadores hay un dato disponible para ser enviado a la "MDE_sec" correspondiente y entrar a la compresión doble. Cuando se envía el buff_cont se resetea.

- **pos_cont1, pos_cont2, pos_cont3**: números naturales, únicamente toman los valores 0, 16 y 32. Indican en qué posición del buff_cont correspondiente se copian los datos de contador. Cada vez que se copia un contador en buff_cont, pos_cont aumenta en 16. Y cuando pos_cont vale 32 su buff_cont correspondiente se envía a la "MDE_sec" y pos_cont es reseteado.
- **env_buff1, env_buff2, env_buff3**: se corresponden con FSL_M_Write, FSL_M_Write4, FSL_M_Write7 respectivamente. Se activan para enviar los datos correspondientes por su FSL determinado.
- **env_cont1, env_cont2, env_cont3**: Se activan para indicar a la "MDE_sec" correspondiente que hay un dato listo para ser enviado.
- **fin**. 1 bit. Se activa cuando num_datos y cuenta_datos son iguales, indicando que MicroBlaze ya ha enviado todos los datos.
- **fin_f1, fin_f2, fin_f3.** 1 bit**.** Se activan cuando se envía el último dato del buff correspondiente al MicroBlaze.
- **aux**. número natural, desde 0 hasta 2. Se utiliza para tener una espera de 2 ciclos de reloj en los estados recibido1 y recibido2 de la MDE. Dichas esperas son necesarios para la correcta recepción de los datos.
- **registro_sec1, registro_sec2, registro_sec3**: 32 bits. Reciben el primer dato enviado por la MDE_Cuenta respectiva listo para la compresión doble.
- **input_sec1, input_sec2, input_sec3**: 32 bits. Reciben el nuevo dato enviado por la MDE_Cuenta respectiva, listo para la compresión doble, y se compara con el registro_sec correspondiente. Si son iguales el contador suma 1. Si no, registro_sec1 se copia en el buffer de salida y después input_sec se copia en registro_sec.
- **cont_sec1, cont_sec2, cont_sec3**: 16 bits. Contador. Cuando input_sec es igual a su registro_sec el contador suma '1'. Cuando son diferentes se copia el contador en buff_cont_sec y después se pone de nuevo a 0.
- **buff_cont_sec1, buff_cont_sec2, buff_cont_sec3**: 32 bits. Almacenan respectivamente los valores de cont_sec1, cont_sec2 y cont_sec3. Como los contadores son de 16 bits, cada vez que se copian dos contadores hay un dato disponible para ser enviado a MicroBlaze. Cuando el dato se copia al buffer de salida el buff_cont se resetea.
- **pos_cont_sec1, pos_cont_sec2, pos_cont_sec3**: número natural, únicamente toman los valores 0, 16 y 32. Indica en qué posición del buff_cont_sec se copian los datos de contador. Cada vez que se copia un contador en buff_cont_sec, su pos_cont aumenta en 16. Y cuando pos_cont_sec vale 32 su buff_cont_sec correspondiente se copia en el buffer de salida y pos_cont_sec se resetea a 0.
- **env_buff_sec1, env_buff_sec2, env_buff_sec3**: 1 bit. se corresponde con FSL_M_Write2, FSL_M_Write5 y FSL_M_Write7 respectivamnete. Se activan para enviar los datos de su FLS_M_Data correspondiente.
- **env_cont_sec1, env_cont_sec2, env_cont_sec3**: 1bit. se corresponde con FSL_M_Write3, FSL_M_Write6 y FSL_M_Write9 respectivamente. Se activan para enviar los datos de FSL_M_Data por su FSL determinado.
- **aux1, aux2, aux3**: 1 bit. Se activan cuando la MDE_sec recibe el último dato desde MDE_Cuenta.
- **c1, c2, c3**: 1 bit. Se activan para indicar a la MDE_sec que buff_sec va a enviar el último dato de la compresión doble a MicroBlaze. Una vez que se envía la MDE_sec vuelve al estado de inicio.

- **Máquina de estados *MDE***

En la figura 13 se puede observar un esquema completo de la máquina de estados "MDE". En la figura 14 se puede ver un esquema de lo que se han llamado *fase1*, *fase2* y *fase3*, que son una parte de la "MDE" y que por su complejidad ha sido preferible realizar un esquema aparte.

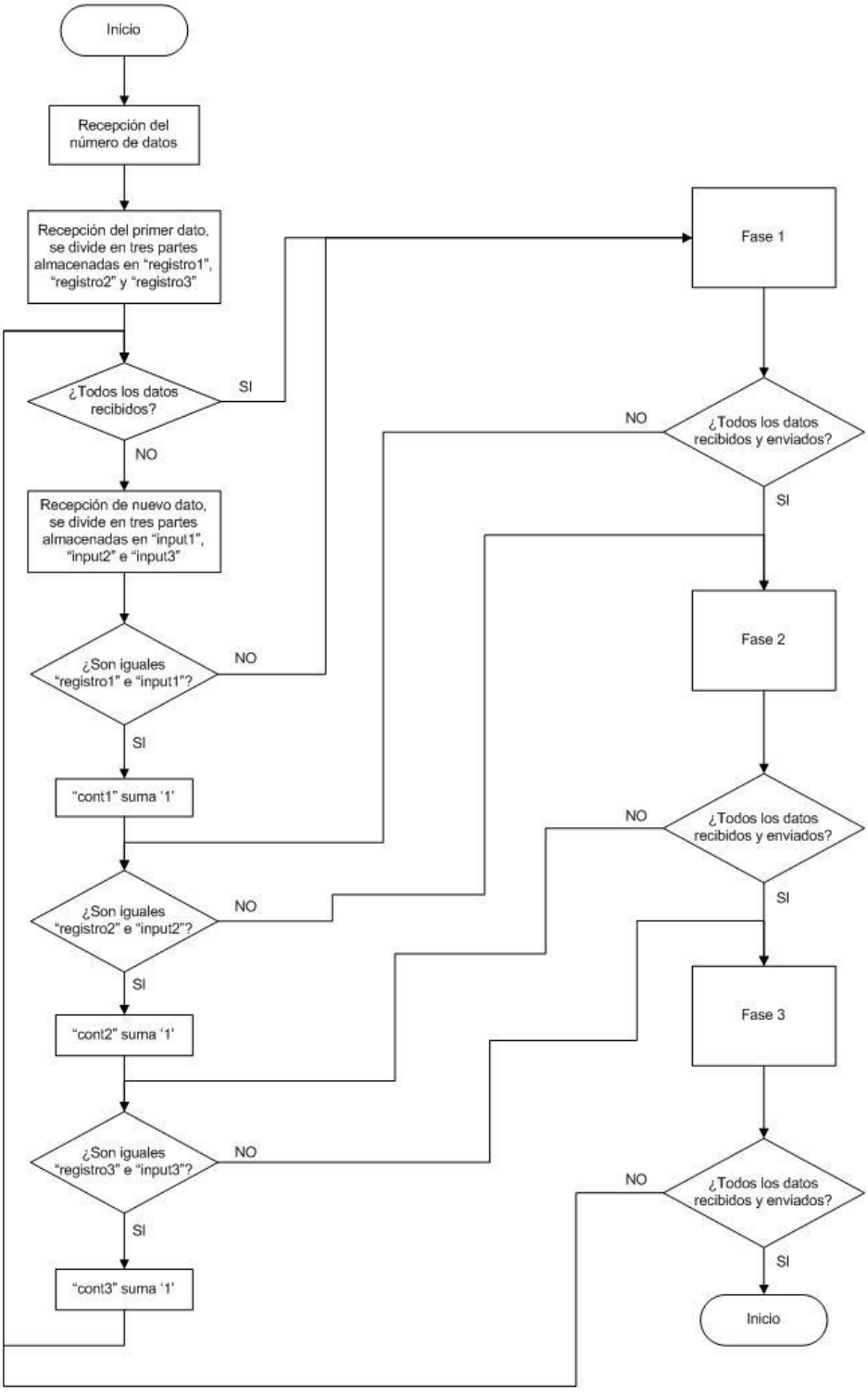


Figura 13. Máquina de estados *MDE*.

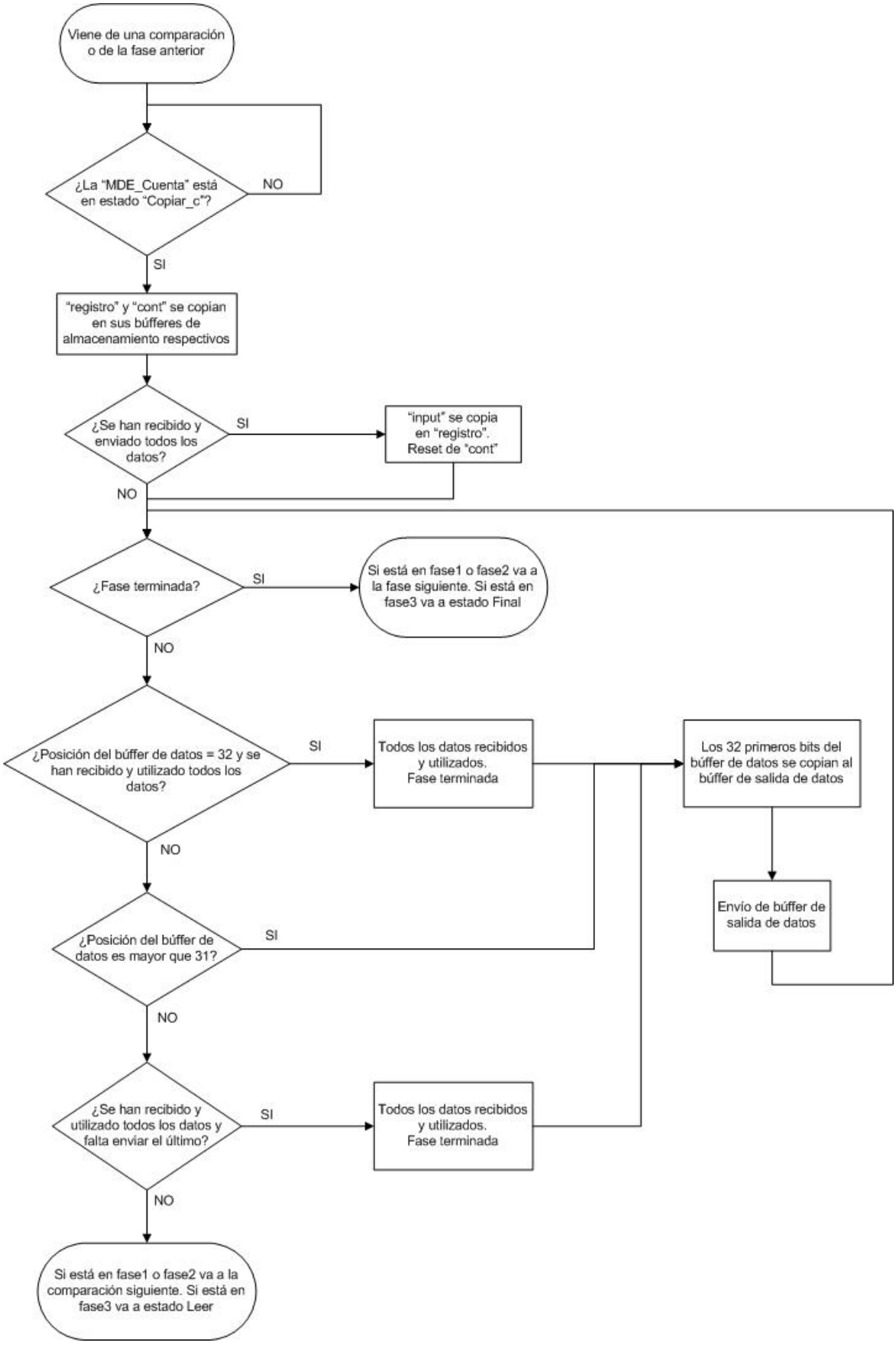


Figura 14. Esquema interno de *fase1*, *fase2* y *fase3.*

La MDE comienza en el estado **Inicio**, donde se hace un reset de todas las señales y registros, y automáticamente pasaría al estado **Numero_datos**. En este estado la MDE se mantiene hasta que la señal de entrada "FSL_S_Exists" está activada. Esto significa que hay un dato de entrada en el FSL esperando a ser leído. Este primer dato se almacena en el vector de 32 bits "num_datos" y es el número de datos que MicroBlaze va a enviar para ser comprimidos. Cuando recibe el dato se activa la señal de salida "FSL_S_Read" para indicar al MicroBlaze que puede enviarse el siguiente dato. A continuación la MDE va al estado **Recibido1**. En este estado se desactiva "FSL_S_Read" y se mantiene durante dos ciclos de reloj hasta que la señal aux valga 2 y va al estado "Primero". Esto se hace para dar tiempo a MicroBlaze a enviar el nuevo dato.

En el estado **Primero** vuelve a esperarse a que "FSL_S_Exists" valga '1'. Entonces el primer dato está listo para ser leído. De sus 32 bits los 12 primeros se almacenan en el vector "registro1", los 10 siguientes en "registro2" y los 10 últimos en "registro3". También existe un registro de 32 bits llamado "cuenta_datos", y cada vez que se lee un dato suma '1' hasta que valga lo mismo que "num_datos". Finalmente se activa "FSL_S_Read" y pasa al estado **Recibido2**. Al igual que en "Recibido1" se desactiva "FSL_S_Read" y cuando "aux" valga 2 pasa al estado "Leer".

En el estado **Leer** lo primero es comprobar si "cuenta_datos" es igual a "num_datos". En caso afirmativo siginifcaría que ya se han recibido todos los datos, por lo que se activaría la señal "fin" y la MDE iría al estado "Copiar1" de la *fase1*. En caso contrario, actuaría como el estado "Primero". Cuando "FSL_S_Exists" se ponga a '1' habrá un nuevo dato de entrada, pero esta vez los bits se copiarán en los vectores "input1", "input2" e "input3", de 12, 10 y 10 bits respectivamente. Se activa "FSL_S_Read" y MDE va al estado **Recibido3**. "FSL_S_Read" vuelve a ser desactivada y en este caso no se realiza espera porque la MDE va a realizar varias acciones antes de volver a leer un nuevo dato por lo que pasa directamente al estado "Comparar1".

A cada uno de los tres campos en que se dividen los datos se le asigna un contador de 16 bits, llamados "cont1", "cont2" y "cont3". En **Comparar1** se comprueba si "input1" y "registro1" son iguales. Si son iguales "cont1" suma '1' y va al estado "Comparar2". En caso contrario de ser diferentes "cont1" se mantiene y MDE va al estado "Copiar1" de la *fase1*.

**Comparar2** realiza lo mismo que su homónimo. Si "input2" y "registro2" son iguales "cont2" suma '1' y MDE va a "Comparar3". Si no va a "Copiar2" de la *fase2*.

Finalmente **Comparar3** se comporta igual. Si "input3" y "registro3" son diferentes MDE va a "Copiar3" de la *fase*3. Sin embargo, si son iguales "cont3" suma '1', pero en este caso la MDE vuelve al estado "Leer" y repite los pasos descritos anteriormente.

El funcionamiento de las tres *fases* es igual, por lo que sólo se explicará una de ellas de forma genérica. Están compuestas por cuatro estados, "Copiar", "Reset_cont", "Posicion" y "Enviar", numerados del 1 al 3 para cada una de las *fases*.

Hay que decir que para que el funcionamiento de todo el algoritmo sea correcto la MDE va sincronizada con las tres máquinas de estado MDE_Cuenta1, 2 y 3 a través de los estados de las *fases*.

Antes de entrar a la explicación es necesario hacer una indicación. El fin de una *fase* se indica con la puesta a '1' de la señal "fin_f", hecho que se produce cuando ya se han recibido todos los datos desde MicroBlaze y el proceso "Buffer" le ha enviado de vuelta todos los datos comprimidos. Durante la ejecución del algoritmo se puede entrar y salir varias veces de una misma *fase*, pero esta no termina hasta que se activa "fin_f". Tampoco hay que confundir la señal "fin" con la señal "fin_f".

Cada *fase* comienza en el estado **Copiar**. Mientras está en "Copiar" el proceso "Buffer" correspondiente copia el valor de "registro" en el buffer de almacenamiento "buff", que es de 43 bits para la *fase1* y de 41 bits en la *fase2* y la *fase3*, en la posición indicada por "pos_buff". Aquí la MDE espera a que la MDE_Cuenta correspondiente esté en el estado "Copiar_c". Cuando ambas máquinas de estado están en sus estados de "Copiar", la MDE_Cuenta copia el valor de "cont" al buffer de almacenamiento de contador "buff_cont" en la posición indicada por "pos_cont". Tras estas acciones la MDE va al estado "Reset_cont".

En **Reset_cont** primero se comprueba si la señal "fin" vale '1'. En ese caso la MDE pasa directamente al estado "Posicion". En caso contrario la MDE realiza dos acciones. Como el valor de "registro" ya se ha copiado al búffer de almacenamiento entonces el valor de "input" se copia en "registro". De igual forma, como el valor del "cont" ya se ha copiado a su búffer el "cont" se pone otra vez cero. Por otro lado, durante este estado el proceso "Buffer" aumenta en 12 o 10 el valor de "pos_buff" según corresponda. Finalmente la MDE va al estado "Posicion".

El estado **Posicion** tiene gran complejidad. Lo primero que comprueba es si la señal "fin_f" está activa, en cuyo caso la *fase* estaría terminada y la MDE iría a la siguiente *fase* o si está en la *fase*3 iría al estado "Final". En caso contrario lo primero que comprueba es si "pos_buff" vale 32 y "fin" está activa. Si esto se cumple significa que todos los datos enviados por MicroBlaze han sido copiados a "Buff" y que además hay justo 32 bits almacenados, por lo que se va a enviar el último dato al MicroBlaze. Se activaría "fin_f" y MDE pasaría a "Enviar".

Si "pos_buff" no es 32 se comprueba si "pos_buff" es mayor que 31. En caso afirmativo esto indicaría que hay más de 32 bits almacenados en "buff" por lo que puede enviarse un dato de salida y se pasaría al estado "Enviar".

Si "pos_buff" es inferior a 31 pero "fin" está activa significa que todos los datos han sido copiados a "buff", pero este no se ha llenado del todo. Entonces, el proceso "Buffer" copiaría los bits al búffer de salida y lo rellenaría con ceros por la derecha y la MDE iría a "Enviar" para enviar el último dato a MicroBlaze, por lo que también se activaría "fin_f".

Finalmente, si no se cumple ninguna de estas condiciones la MDE iría a "Comparar2" si fuera la *fase1*, a "Comparar3" si fuera la *fase2*, o a "Leer" si es la *fase3*.

El último estado de una *fase* es el estado **Enviar**. Aquí la MDE no realiza ninguna acción, simplemente espera a que su FSL de salida correspondiente esté vacío para que se pueda enviar el dato. Si el FSL está vacío la señal de entrada "FSL_M_Full" estará desactivada. En este mismo estado el proceso "Buffer" también comprueba si el FSL está vacío. Si lo está entonces se copian los 32 primeros bits de "buff" al buffer de salida "FSL_M_Data" correspondiente. También se pone a '1' "env_buff". Cuando esta última señal se activa se envía por el FSL el valor de "FSL_M_Data". Una vez que se ha enviado el dato la MDE regresa al estado "Posicion".

Por último, como ya se ha dicho, cuando la *fase*3 ha terminado la MDE va a **Final**. Esto significa que todos los datos de las tres *fases* ya han sido enviados. Aquí simplemente se mantiene un ciclo de reloj y regresa al estado de "Inicio", donde todas las variables son reinicializadas para esperar a recibir una nueva trama de datos.

Finalmente, si se produce un "reset" del sistema, la MDE va automáticamente al estado "Inicio".

- **Proceso *Buffer***

Cada una de las *fases* de la MDE tiene asociado un proceso "Buffer", que se encargar de enviar al MicroBlaze los datos de la compresión simple. En la figura 15**Figura 15** se puede ver un esquema detallado de forma genérica de uno de estos procesos. Posteriormente se dará una explicación detallada del mismo.

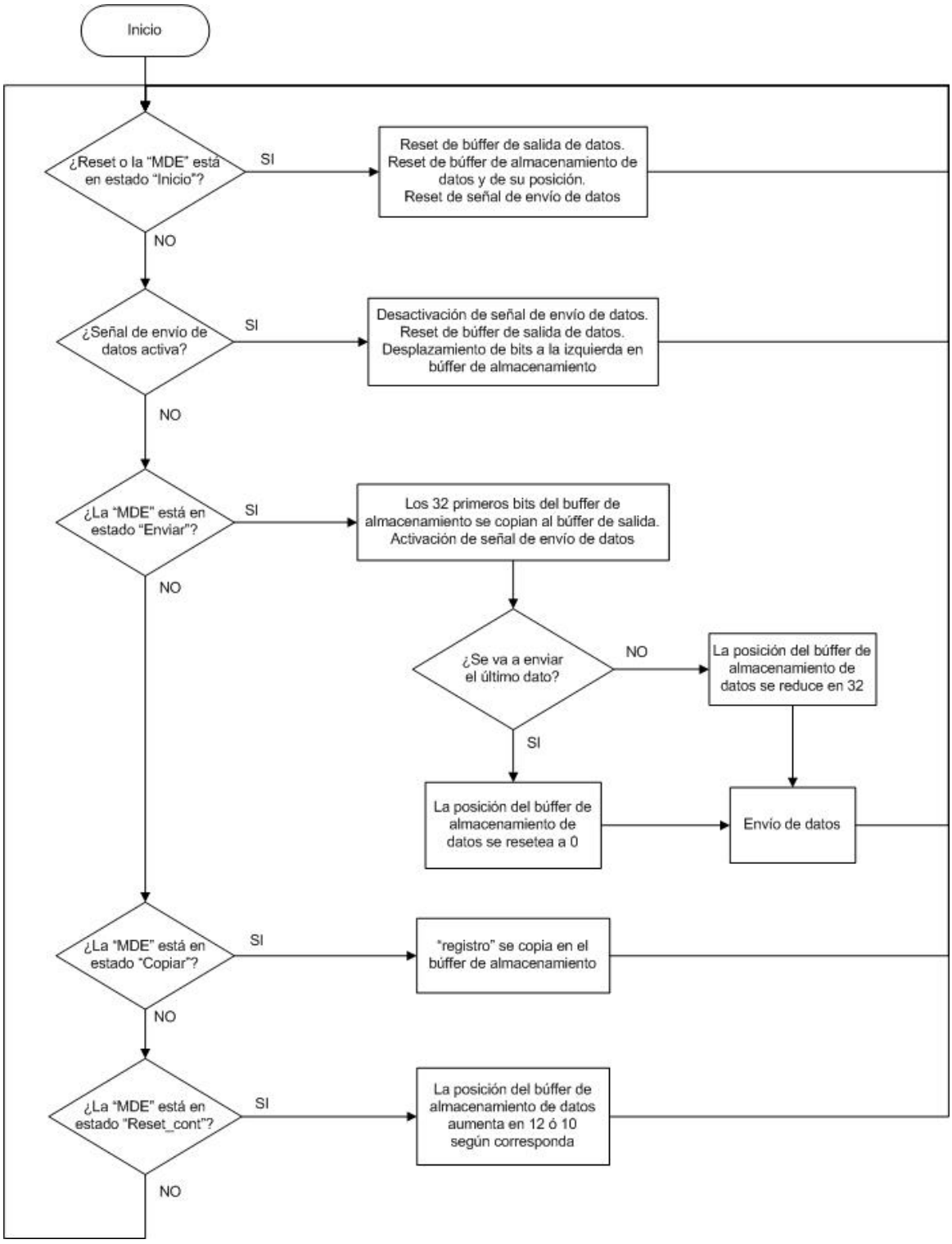


Figura 15. Esquema del proceso *Buffer*.

El proceso “Buffer” funciona según se cumplan diversas condiciones, teniendo algunas prioridad sobre otras.

**Si hay un reset del sistema o la MDE va al estado Inicio**, se resetean “buff”, “pos_buff”, “env_buff” y “FSL_M_Data”.

Como se ha dicho, **si env_buff está activa** se envía el “FSL_M_Data” correspondiente. Una vez ha sido enviada, el propio proceso “Buffer” desactiva “env_buff”, que sólo dura activa un ciclo de reloj. A la misma vez también se encarga de desplazar 32 bits hacía la izquierda los bits de “buff” y rellena con ‘0’ por la derecha. Finalmente también resetea “FSL_M_Data”.

**Si la MDE está en Enviar**, se comprueba si el FSL de salida está vacío. En ese caso se copian los primeros 32 bits de “buff” en “FSL_M_Data” y se activa “env_buff” para que sean enviados. Si el FSL de salida está lleno esperará a que se vacíe. Además, si se han enviado todos los datos el valor de “pos_buff” se resetea a 0, si no, se reduce en 32.

**Si la MDE está en Copiar**, el valor de “registro” se copia en “buff”, en la posición indicada por “pos_buff”.

Por último, **si la MDE está en Reset_cont**, el valor de “pos_buff” aumenta en 12 o 10 según corresponda.

Si no se cumple ninguna de las condiciones descritas comienza a comprobarlas todas nuevamente desde el principio.

- **Máquina de estados *MDE_Cuenta***

Al igual que ocurría con los procesos "Buffer" cada una de las *fases* de la MDE tiene asociada una MDE_Cuenta, con la que va sincronizada. A su vez, cada MDE_Cuenta también está sincronizada con una máquina de estados secundaria llamada MDE_sec. En la figura 16 se puede ver un esquema detallado de forma genérica de una MDE_Cuenta y posteriormente se realizará una explicación en detalle.

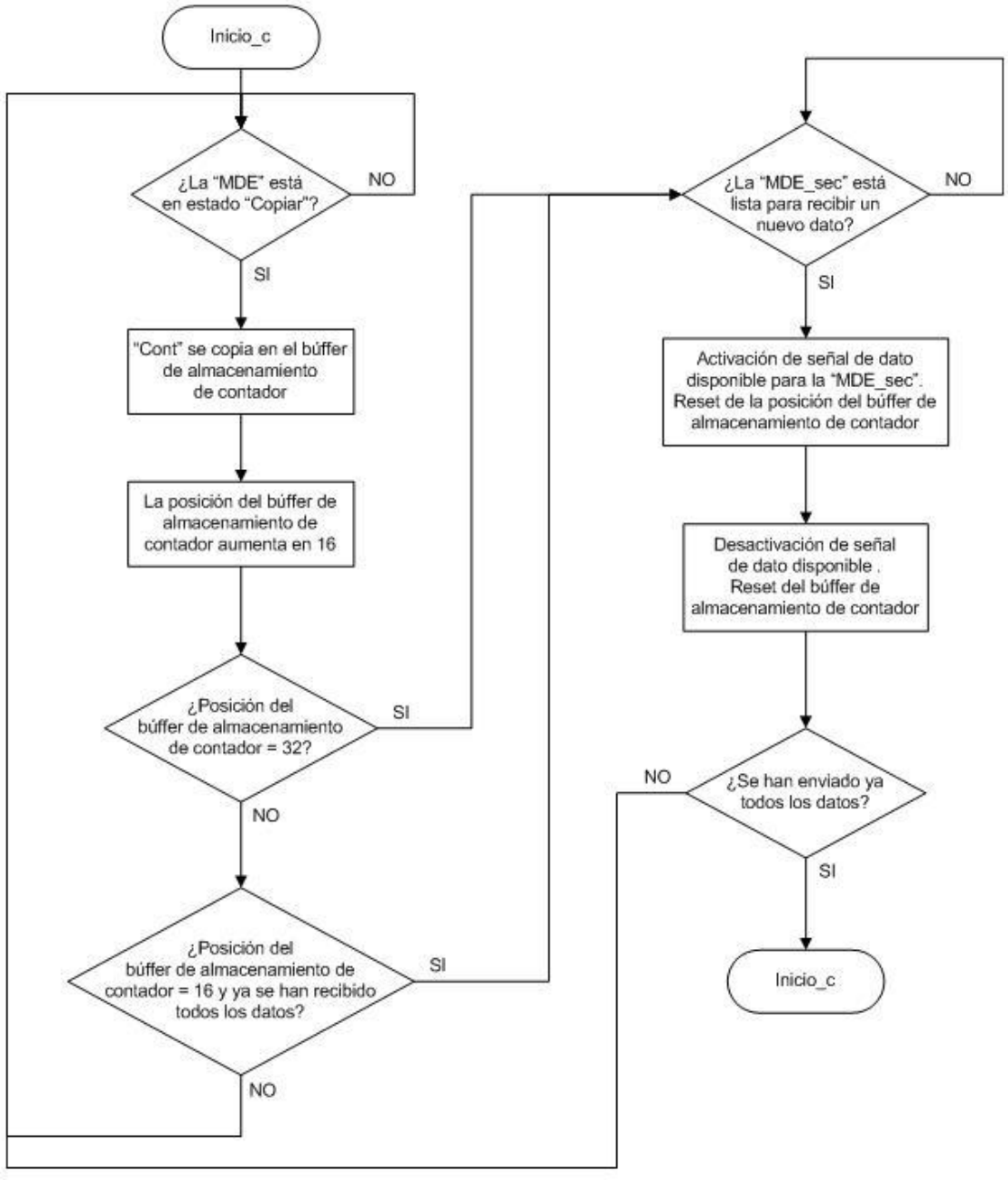


Figura 16. Máquina de estados *MDE_Cuenta*.

La MDE_Cuenta comienza en el estado **Inicio_c** donde se hace un reset de “pos_cont”, “buff_cont” y “env_cont”. Automáticamente pasa al estado “Copiar_c”.

En **Copiar_c** espera a que la *fase* correspondiente de la MDE esté en el estado “Copiar”. En ese caso el “cont” respectivo se copia en “buff_cont” en la posición indicada por “pos_cont”. MDE_Cuenta va a “Reset_cont_c”.

En **Reset_cont_c** se aumenta en 16 el valor de “pos_cont” y pasa al estado “Enviar_c”.

**Enviar_c** es un estado de gran complejidad. Hay que decir que “pos_cont” sólo toma los valores 0, 16 y 32. Lo primero que se hace es comprobar si “pos_cont” vale 32. En ese caso se mantiene esperando a que la MDE_sec correspondiente esté en uno de sus estados de leer datos, ya sea “Primero_sec” o “Leer_sec”. En ese caso MDE_sec está lista para recibir un nuevo dato, por lo que MDE_c activa “env_cont” y pasa al estado “Final_c”.

Si “pos_cont” no es 32 se comprueba si vale 16 y si “fin” vale ‘1’. Si eso ocurre significa que todos los contadores ya han sido copiados al búffer de almacenamiento y no va a llegar más datos. Entonces espera a que la MDE_sec correspondiente esté en uno de sus estados de leer datos, ya sea “Primero_sec” o “Leer_sec”. Cuando esto ocurre, el búffer de almacenamiento se rellena con ceros por la derecha y se activa “env_cont” para informar a la MDE_sec que tiene un dato disponible. MDE_Cuenta va al estado “Final_c”.

Pero si “pos_cont” no es 32 ni tampoco es 16 y “fin” no está activada entonces MDE_Cuenta regresaría a “Copiar_c”.

**Final_c** es el último estado de la MDE_Cuenta. En él “env_cont” es desactivada y “buff_cont” es reseteado porque la MDE_sec ya ha recibido el dato. Se comprueba si “fin” está activa. Eso significa que ya se han enviado todos los contadores, por lo que volvería al estado “Inicio_c” para resetear todas las variables y esperar un nuevo grupo de datos. Pero si “fin” vale ‘0’ entonces regresa a “Copiar_c” porque quedan contadores por ser enviados.

Por último, si hay un “reset” del sistema MDE_Cuenta va al estado “Inicio_c”.

- **Máquina de estados *MDE_sec***

Como ya se ha explicado a lo largo de la memoria, MDE_sec se encarga de realizar la compresión doble. Hay tres MDE_sec, sincronizadas con las tres MDE_Cuenta. También existen un proceso "Buffer_sec" y un proceso "Cuenta_sec" asociados a la MDE_sec. En la figura 17 se puede ver un esquema detallado de MDE_sec y posteriormente habrá una explicación en detalle.

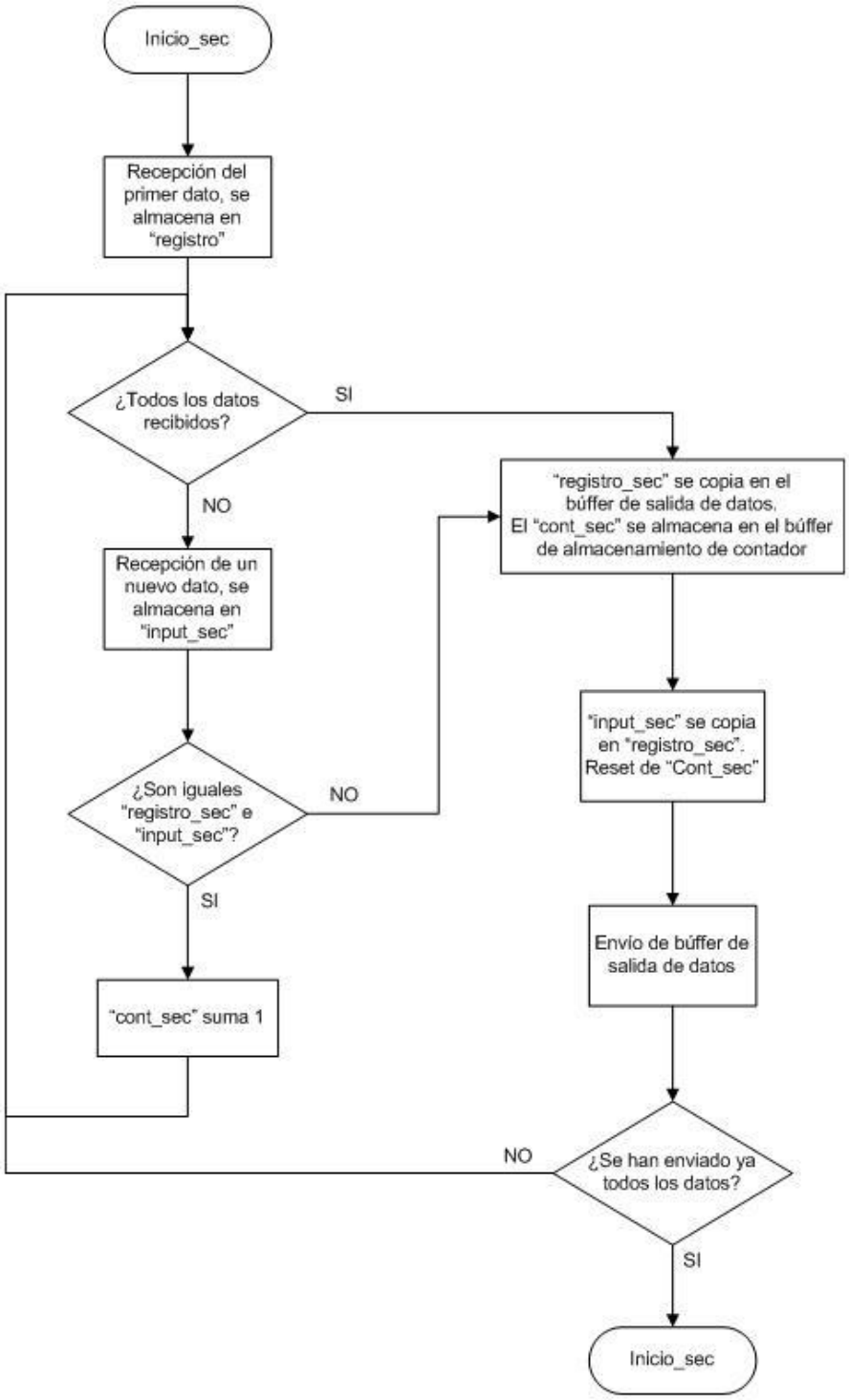


Figura 17. Máquina de estados de *MDE_sec*.

La MDE_sec tiene un funcionamiento muy similar al de la MDE, pero son menos complejidad. Comienza en el estado **Inicio_sec**, que inicializa todas las señales y va automáticamente al estado "Primero_sec".

En **Primero_sec** espera a que la MDE_Cuenta active "env_cont". Eso significa que hay un dato listo para la compresión doble. Entonces se copia el contenido de "buff_cont" en un vector de 32 bits llamado "registro_sec". A diferencia de MDE, aquí los datos no se dividen en partes, si no que se almacenan completos. También se comprueba si "fin" vale '1'. En ese caso ya no hay más datos para la compresión doble, por lo que se activa la señal "aux" correspondiente y se pasa al estado "Copiar_sec". Si "fin" vale '0' se pasa a "Leer_sec".

El estado **Leer_sec** lo primero que hace es comprobar si "aux" vale '1'. En ese caso no se va a recibir ningún dato, por lo que se activa la señal "c" y pasa al estado "Copiar_sec". Pero si "fin" vale '0' entonces espera a que "env_cont" valga '1'. Entonces se copian los 32 bits de "buff_cont" en un vector llamado "input_sec". También comprueba si "fin" vale '1'. En ese caso ya no se van a recibir más datos para la compresión doble, por lo que se activa "aux". Si no, "aux" se mantiene desactivada. Se pasa al estado "Comparar_sec".

A cada dato de la compresión doble se le asigna un contador de 16 bits, llamado "cont_sec", y que se va almacenando en un búffer de 32 bits. En el estado **Comparar_sec** se comprueba si "registro_sec" e "input_sec" son iguales. Si lo son "cont_sec" aumenta en '1' y se regresa al estado "Leer_sec" para recibir un nuevo dato. En caso de que sean diferentes "cont_sec" se mantiene y se pasa al estado "Copiar_sec".

En el estado **Copiar_sec** la MDE_sec no realiza ninguna acción y tras un ciclo de reloj pasa al estado "Reset_cont_sec". Sin embargo, cuando la MDE_sec está en "Copiar_sec" el proceso "Buffer_sec" copia "registro_sec" en el búffer de salida "FSL_M_Data" correspondiente. Como el dato se utiliza completamente se puede copiar directamente al búffer de salida en lugar de copiarlo primero al búffer de almacenamiento. Paralelamente, el proceso "Cuenta_sec" copia el "cont_sec" en su búffer de almacenamiento, "buff_cont_sec", en la posición indicada por "pos_cont_sec".

En el estado **Reset_cont** se copia el valor de "input_sec" en "registro_sec", eliminando su anterior valor, puesto que ya se ha copiado al búffer de salida. Igualmente, se resetea el "cont_sec", cuyo valor también ha sido copiado. Se pasa al estado "Enviar_sec". Durante "Reset_cont" el proceso "Cuenta_sec" aumenta en 16 el valor de "pos_cont_sec".

En el estado **Enviar_sec** la MDE_sec comprueba si el FSL de salida está vacío. Si está lleno "FSL_M_Full" valdrá '1', sino valdrá '0'. En este mismo estado el proceso "Buffer_sec" también comprueba si el FSL está vacío. Si está vacío entonces "Buffer_sec" activa la señal "env_buff_sec". Cuando esta última señal se activa se envía por el FSL el valor del "FSL_M_Data" correspondiente.

Durante "Enviar_sec" el proceso "Cuenta_sec" comprueba si "pos_cont" vale 16. Si además "aux" vale '1' entonces se copia el contenido de "buff_cont_sec" al búffer de salida correspondiente y se rellena con ceros por la derecha. Se activa "env_buff_cont" para enviarlo.

Desde "Enviar_sec" se pasa al estado **Final_sec**. Aquí se comprueba si la señal "c" vale '1'. La diferencia entre "aux" y "c" es que "aux" se activa cuando se recibe el último dato y "c" se activa cuando este último dato es enviado, indicando que MDE_sec ha terminado. Entonces, si "c" está activa MDE_Cuenta va al estado "Inicio_sec". Si no, regresa al estado "Leer_sec".

Si hay un reset del sistema la MDE_sec automáticamente va al estado "Inicio_sec".

- **Proceso *Buffer_sec***

Cada una de las MDE_sec tiene asociado un proceso “Buffer_sec”, que se encarga de enviar al MicroBlaze los datos de la compresión doble. En la figura 18 se puede ver un esquema detallado de forma genérica de uno de estos procesos. Posteriormente se dará una explicación detallada del mismo.

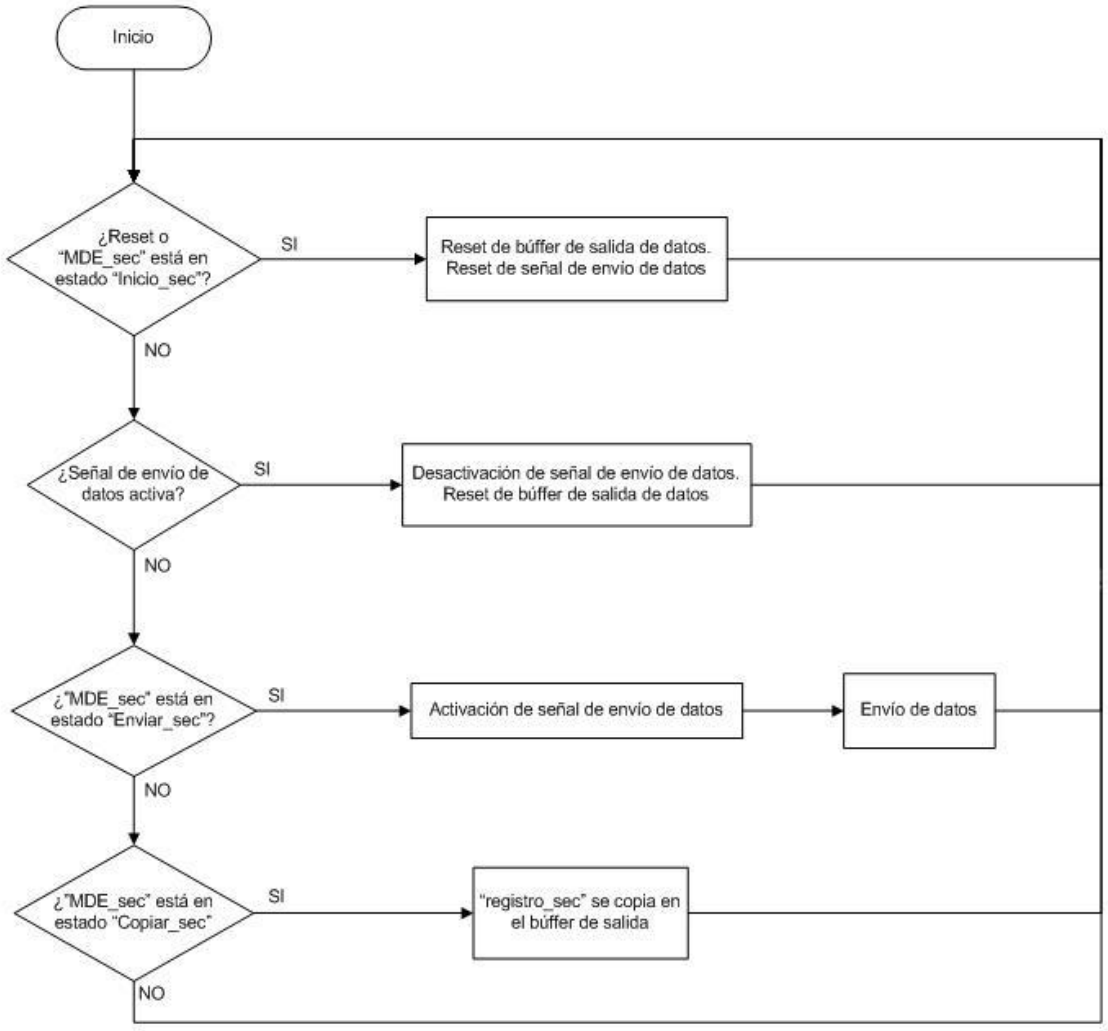


Figura 18. Proceso *Buffer_sec*.

El proceso “Buffer_sec” funciona dependiendo de si se cumplen unas condiciones, teniendo algunas prioridad sobre otras.

**Si hay un reset del sistema o la MDE_sec correspondiente está en estado Inicio_sec**, se resetean “env_buff_sec” y “FSL_M_Data”.

Como se ha dicho, **si env_buff_sec está activa** se envía “FSL_M_Data”. Una vez ha sido enviada, el proceso “Buffer_sec” desactiva “env_buff_sec”, que sólo dura activa un ciclo de reloj. Al mismo tiempo también resetea “FSL_M_Data”.

**Si la MDE_sec está en Enviar_sec**, “Buffer_sec” espera a que el FSL de salida está vacío, y entonces activa “env_buff_sec” para que los datos sean enviados.

Finalmente, **cuando la MDE está en Copiar_sec** se copia “registro_sec” en el búffer de salida “FSL_M_Data”.

Si no se cumple ninguna de las condiciones descritas comienza a comprobarlas todas nuevamente desde el principio.

### ➢ Proceso *Cuenta_sec*

Al igual que con el proceso "Buffer_sec", cada una de las MDE_sec tiene asociado un proceso "Cuenta_sec", que se encarga de enviar al MicroBlaze los contadores de la compresión doble. En la figura 19 se puede ver un esquema detallado de forma genérica de uno de estos procesos y finalmente se dará una explicación detallada del mismo.

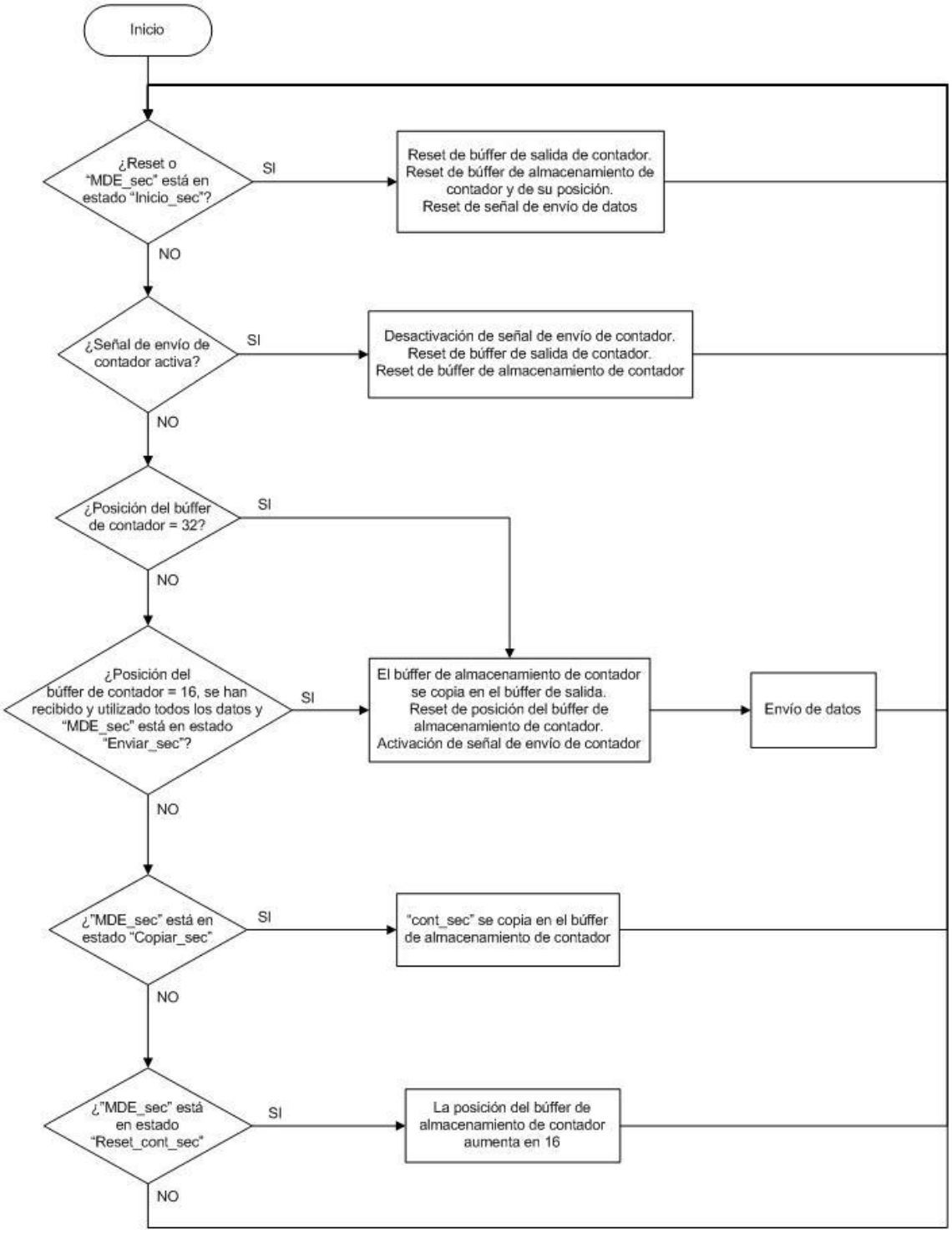


Figura 19. Proceso *Cuenta_sec*.

Al igual que “Buffer_sec”, el proceso “Cuenta_sec” también funciona si se cumplen ciertas condiciones, teniendo algunas prioridad sobre otras.

**Si se produce un reset del sistema o la MDE_sec está en estado Inicio_sec**, se resetean “buff_cont_sec”, “pos_cont_sec”, “env_cont_sec”, y el “FSL_M_Data” correspondientes.

**Cuando env_cont_sec está activa** se envía “FSL_M_Data”. Una vez ha sido enviado, el propio proceso “Cuenta_sec” desactiva “env_cont_sec”, que sólo dura activa un ciclo de reloj. También se resetean “buff_cont_sec” y “FSL_M_Data”.

La posición del búffer del contador, “pos_cont_sec”, sólo puede valer 0, 16 y 32. **Cuando pos_cont_sec vale 32** significa que “buff_cont_sec” está lleno. Entonces, si “FSL_M_Full” vale ‘0’ significa que su FSL de salida está vacío. Entonces se copia “buff_cont_sec” en “FSL_M_Data”. También se activa “env_cont_sec” para hacer el envío del dato y “pos_cont_sec” es reseteado a 0.

También puede darse el caso de que **pos_cont_sec valga 16 pero aux valga ‘1’**. Si esto se cumple significa que ya se han recibido todos los datos pero aún queda un último contador por ser enviado. Entonces, **cuando la MDE_sec esté en el estado Enviar_sec** para enviar el último búffer de datos, el “buff_cont_sec” se rellenará con ceros por la derecha y se copia a “FSL_M_Data”. También se activa “env_cont_sec” para enviarlo y se resetea “pos_cont_sec”, pasando a valer 0 de nuevo.

**Si la MDE_sec está en Copiar_sec** se almacena “cont_sec” en su “buff_cont_sec”, en la posición indicada por “pos_cont_sec”.

Por último, **si la MDE_sec está en Reset_cont_sec** el valor de “pos_cont_sec” es aumentado en 16.

Si no se cumple ninguna de las condiciones descritas comienza a comprobarlas todas nuevamente desde el principio.

## 4.4. DEFINICIÓN DEL INTERFACE SOFTWARE

Para que la comunicación entre el MicroBlaze y el coprocesador pueda establecerse correctamente ha sido necesario realizar un interface software implementado en código C. Dicho código se encarga de fundamentalmente de 3 tareas, enviar los datos desde MicroBlaze al coprocesador y recibirlos correctamente desde este y mostrarlos por pantalla.

El código comienza con la declaración de las librerías que se van a utilizar, en nuestro caso **stdio.h** y **stdlib.h**. También se declaran otros archivos necesarios, externos al código C, que en nuestro proyecto son **compresor.h** y **xparameters.h**, que son dos archivos que se crean automáticamente cuando generamos el "bitstream" del proyecto con el programa XPS.

```
#include <stdio.h>
#include <stdlib.h>
#include "compresor.h"
#include "xparameters.h"
```

Después se definen las instrucciones de lectura y escritura de datos en el FSL. Se utiliza la instrucción **WRITE_COMPRESOR (val)** para escribir en el FSL. A lo largo del código sustituiremos "val" por el nombre del dato que queremos enviar. Como en nuestro proyecto sólo tenemos un FSL de salida desde MicroBlaze, lo numeramos con 0, "WRITE_COMPRESOR_0 (val)". También utilizamos la instrucción **READ_COMPRESOR (val)** para leer desde un FSL de entrada. Igualmente, sustituiremos "val" por el nombre de la variable en la que queramos almacenar el dato de entrada. Nuestro MicroBlaze tiene 9 FSL de entrada, por lo que los numeraremos desde "READ_COMPRESOR_0 (val)" hasta "READ_COMPRESOR_8 (val)". Mostraremos un ejemplo de cómo quedaría el código con un FSL de entrada y otro de salida, donde "Compresor" es el nombre del periférico cuyo archivo de C hemos utilizado para crear la interface de programa.

```
#define WRITE_COMPRESOR_0(val)  write_into_fsl(val, XPAR_FSL_COMPRESOR_0_INPUT_SLOT_ID)
#define READ_COMPRESOR_0(val)  read_from_fsl(val, XPAR_FSL_COMPRESOR_0_OUTPUT_SLOT_ID)
```

Aquí entonces comenzaría el **main** del programa. Lo primero que aparece en el "main" son las declaraciones de las señales, variables, registros, enteros, etc. que vamos a utilizar en nuestro programa, dándoles valores cuando sea necesario. Es aquí donde se deben declarar tanto el número de datos como los propios datos que van a ser enviados desde MicroBlaze al coprocesador.

```
main()
{       unsigned int A1, A2, A3, A4, B1, B2, B3, B4, datos;
        unsigned int output_0[8], output_1[8], output_2[8];
       int i, j;

datos = 9; //es el número de datos que se van a enviar
A1 = 0x23410040;
A2 = 0x23410080;
A3 = 0x23420040;
A4 = 0x23420080;
B1 = 0x23510040;
B2 = 0x23510080;
B3 = 0x23520040;
B4 = 0x23520080;
```

Una vez que tenemos declarados los datos, los enviamos utilizando "WRITE_COMPRESOR", como hemos explicado anteriormente. No importa si los datos declarados son en binario, decimal o hexadecimal, pues cuando se envían por el FSL se envían en formato binario de 32 bits. Primero se envía el número de datos y luego los propios datos.

```
WRITE_COMPRESOR_0(datos);
WRITE_COMPRESOR_0(B4);
WRITE_COMPRESOR_0(B4);
WRITE_COMPRESOR_0(A1);
WRITE_COMPRESOR_0(B4);
WRITE_COMPRESOR_0(B4);
WRITE_COMPRESOR_0(A1);
WRITE_COMPRESOR_0(B4);
WRITE_COMPRESOR_0(B4);
WRITE_COMPRESOR_0(A1);
```

Finalmente, utilizando "READ_COMPRESOR" leemos los datos que llegan a través de los FSL de entrada y los almacenaremos en la variable deseada. Es necesario indicarle al código el número exacto de datos que se van a recibir por cada uno de los FSL. Si no lo hacemos, se pueden cometer 2 errores. Si pedimos lectura de menos datos de los correctos estaremos perdiendo información. Si por el contrario pedimos que lea más datos de los correctos el código se quedará bloqueado, pues seguirá esperando a leer un dato que no va a llegar, impidiendo el avance y por tanto, impidiendo también la lectura de datos del siguiente FSL.

Así mismo, cada vez que un dato es leído es mostrado por pantalla utilizando la instrucción **xil_printf("X", dato)**. La "X" sirve para indicar que queremos que muestre el dato en formato hexadecimal. Si en su lugar pusiéramos "D" nos lo mostraría en decimal. Igualmente se pueden indicar otros formatos con otras instrucciones.

```
//Datos de buffer1
   for (i=0; i<2; i++)
   {
      READ_COMPRESOR_0(output_0[i]);
      xil_printf("Buffer1: %d %X \r\n",i,output_0[i]);
   }
```

Por último se pide que se muestre por pantalla el número de datos que fueron enviados desde MicroBlaze al coprocesador.

```
xil_printf("El numero de datos enviados es: %d \r\n",datos);
```

En la figura 20 se puede ver cómo queda la salida en la consola de un pequeño ejemplo en el que se envían los siguientes 9 datos:

```
23410040
23410040
23520080
23410040
23410040
23520080
23410040
23410040
23520080
```

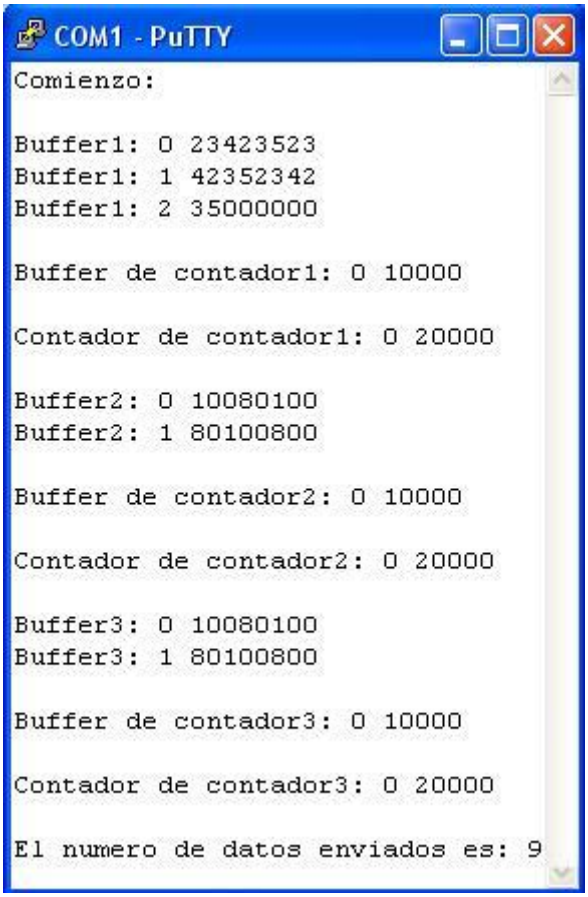


Figura 20. Ejemplo de programa con envío de 9 datos.

- Buffer1 son los datos comprimidos del primer campo de los datos en la compresión simple.
- Buffer de contador1 son los contadores comprimidos del primer campo de los datos en la compresión simple.
- Contador de contador1 son los contadores comprimidos del primer campo en la compresión doble.
- Buffer2 son los datos comprimidos del segundo campo de los datos en la compresión simple.
- Buffer de contador2 son los contadores comprimidos del segundo campo de los datos en la compresión simple.
- Contador de contador2 son los contadores comprimidos del segundo campo en la compresión doble.
- Buffer3 son los datos comprimidos del segundo campo de los datos en la compresión simple.
- Buffer de contador3 son los contadores comprimidos del segundo campo de los datos en la compresión simple.
- Contador de contador3 son los contadores comprimidos del segundo campo en la compresión doble.

# 5. ESTUDIO DEL RENDIMIENTO

Como ya se ha comentado a lo largo de esta memoria, la arquitectura escogida finalmente ha sido la compuesta por un MicroBlaze y un coprocesador y que realiza una doble compresión de datos. Esta elección viene motivada por dos motivos, su correcto funcionamiento y porque su consumo de costes hardware es menor.

En este capítulo se pretende hacer una descripción de las pruebas realizadas y de sus resultados, incluyéndose una comparativa en el rendimiento de nuestro algoritmo hardware frente a un algoritmo software existente previamente. Finalmente se hablará de los medios empleados para ello y también se hará un análisis del coste hardware de los modelos diseñados.

## 5.1. VALIDACIÓN DE LA ARQUITECTURA

En este apartado se va a empezar hablando de la herramienta utilizada inicialmente para diseñar el código y posteriormente se hará una descripción de las pruebas realizadas en la FPGA para la comprobación del correcto funcionamiento del algoritmo y las mediciones de rendimiento.

### 5.1.1. QUARTUS II. Simulación

**Quartus II** es una herramienta de software producida por Altera para el análisis y la síntesis de diseños realizados en HDL. Permite al desarrollador compilar sus diseños, realizar análisis temporales, examinar diagramas RTL y configurar el dispositivo de destino con el programador.

El empleo de Quartus II permite a los diseñadores utilizar los dispositivos HardCopy Stratix integrados en el programa para realizar pruebas sobre el código diseñado. De esta manera se puede prever y verificar su rendimiento, el cual resulta en promedio un 50 por ciento más rápido que su FPGA equivalente. **Adicionalmente**, otro motivo para el uso de Quartus II es su bajo precio en comparación con otras herramientas de diseño de ASIC [W - QII].

Aunque en el apartado 3.1. se dijo que para implementar el proyecto se había utilizado el entorno de desarrollo de Xilinx, **el uso de Quartus II tiene dos motivos fundamentales: la velocidad y la simulación.**

A diferencia del XPS, Quartus II realiza una rápida compilación y depuración de errores, mientras que XPS tarda alrededor de 10 minutos en compilar el código VHDL. Utilizando Quartus II el proceso se acorta considerablemente, llegando a durar menos de 30 segundos.

Además, posee una herramienta de simulación que permite ver el funcionamiento del código mediante diagramas de tiempo, donde se puede comprobar en qué momento se activan o desactivan las señales o ver donde se encuentra una máquina de estados. Xilinx también dispone de una herramienta de simulación, pero es necesario un banco de pruebas. Sin embargo esto en Quartus II no es necesario, puesto que también permite la activación y desactivación de las señales de entrada manualmente. Es una herramienta cuyo uso es rápido e intuitivo.

Por estas razones nuestro algoritmo VHDL fue diseñado primero en Quartus II y una vez que se vio que funcionaba correctamente en la simulación el código se implementó en la FPGA a través del XPS.

En la figura 21 se puede ver un pequeño ejemplo de cómo queda una simulación en Quartus II.

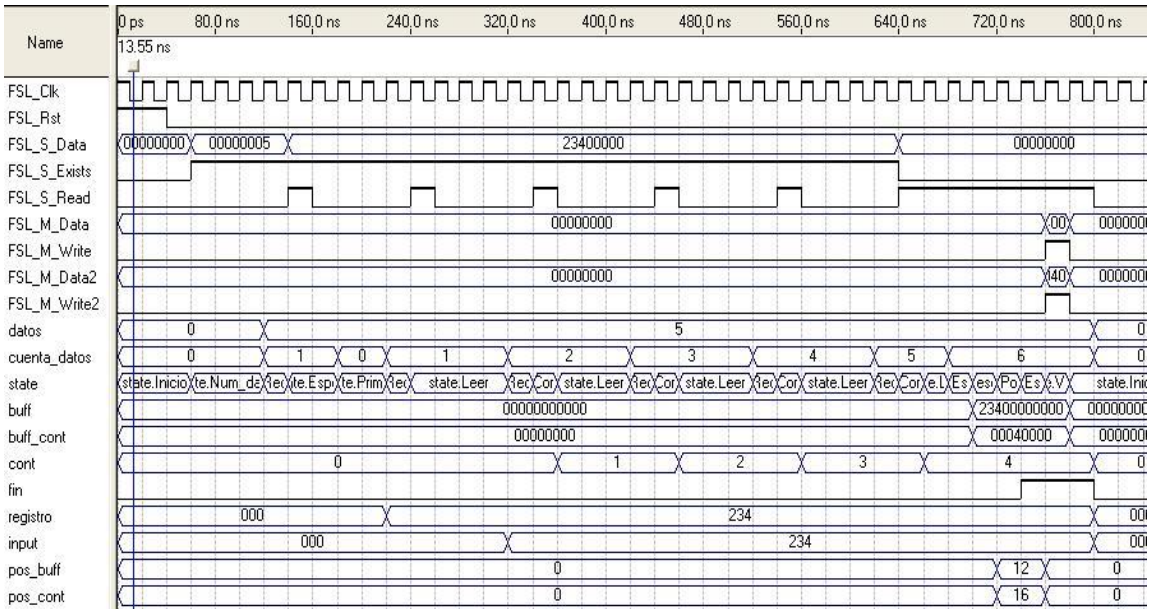


Figura 21. Ejemplo de simulación en Quartus II.

Esta simulación es un pequeño ejemplo de la compresión simple del primero de los tres campos en que se divide cada uno de los datos. Por el puerto de entrada "FSL_S_Data" se recibe primero un "5" en hexadecimal que se almacena en "datos" y posteriormente se reciben 5 veces el dato "23400000". "FSL_S_Read" es activada cada vez que se ha leído un dato y se pide el siguiente. En este caso la compresión se hace únicamente de los 12 primeros bit, es decir, de "234".

Como puede verse "cuenta_datos" va aumentado según se van recibiendo los datos y cuando llega a "5" significa que ya se han recibido todos. Así mismo aumentará hasta "6" para indicar que en ese momento ya han sido comparados todos los datos. También se puede ver que cuando todos los datos han sido comparados el "234" se copia en "buff" y se rellena con '0' por la derecha, y el contador, "cont", se copia en "buff_cont" y refleja un "0004" porque son las repeticiones del dato. También se rellena con '0' por la derecha. De igual forma, se ve como "pos_buff" y "pos_cont" aumentan en 12 y 16 respectivamente una vez que los datos han sido copiados.

Finalmente "buff" es copiado al puerto de salida "FSL_M_Data" y se envía con la activación de "FSL_M_Write". Igualmente "buff_cont" es copiado a "FSL_M_Data2" y es enviado con la activación de "FSL_M_Write2".

Una vez han sido enviados los datos la máquina de estados vuelve al estado "Inicio", terminando así el proceso de compresión.

### 5.1.2. Pruebas en la FPGA Spartan 3A

Una vez que el código fue probado en el simulador y se comprobó que funcionaba este fue implementado en la FPGA. **Para comprobar que el funcionamiento era correcto se probó a enviar desde MicroBlaze al coprocesador diferentes tramas de datos**. Se debe tener en cuenta que los datos enviados son divididos en tres campos cuando llegan al coprocesador.

A continuación se detallan las pruebas realizadas y los resultados obtenidos:

1. **Envío de 16 datos iguales en sus tres campos** para comprobación del correcto funcionamiento.

Se ha enviado 16 veces seguidas el dato "23410080" en hexadecimal. Por lo tanto, sólo se almacena un dato en cada uno de los búffers de almacenamiento correspondiente a los tres campos. El primero almacena los 12 primeros bits, "234" en hexadecimal (*buffer1*), el segundo almacena los 10 bits siguientes, "000100000" en binario (*buffer2*) y el tercero los 10 bits restantes, "0010000000" (*buffer3*). Luego se rellenan con '0' por la derecha hasta completar los búffers.

En la compresión simple se almacena un "000F" hexadecimal en los tres *búffers de contador*, porque los contadores son de 16 bits y cada campo llega repetido 15 veces. Como sólo llega un dato, sólo hay un dato de contador. Después se rellena con '0' por la derecha. Hay que decir que la consola no muestra los ceros por la izquierda.

Finalmente, a la compresión doble llega el único dato del contador de la compresión simple, el "000F0000" (*buffers de contador 1, 2 y 3*). Como no se repite ninguna vez los *contadores de contador* muestran "0" (*contador de contador 1, 2 y 3*).

El resultado de la prueba es correcto como puede verse en la figura 22.

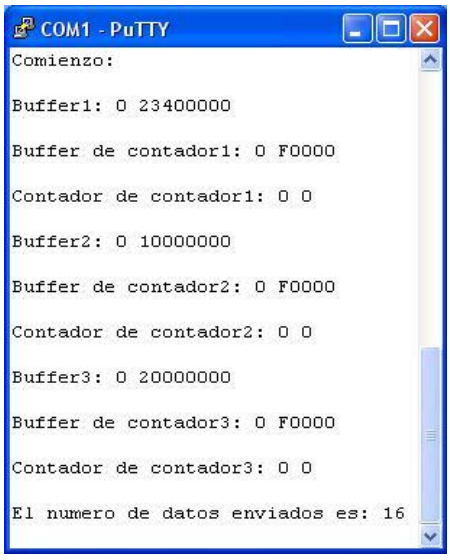


Figura 22. Resultado de prueba 1.

2. **Envío de 16 datos, diferentes todos del anterior en sus tres campos** para comprobación del correcto funcionamiento.

En esta prueba se envían los datos "23410080" y "23520040" de forma alternativa durante 8 veces. Con lo cual, al no repetirse ninguno de los campos, se copian 16 datos en cada uno de los búffers. El *buffer1* almacena "234" y "235" de forma alternativa, *buffer2*

"0001000000" y "0010000000" y *buffer3* "0010000000" y "0001000000". En este caso el *búffer1* se llena con 6 datos completos y *buffer2* y *buffer3* con 5 datos completos, por lo que no es necesario rellenar con 0 por la derecha.

Como llegan 16 datos diferentes, 16 veces se copian en los tres *buffer de contador* un "0000", y se almacenan de 2 en 2, por lo que habrá 8 datos iguales con "00000000" en los tres *buffer de contador*. Los 8 datos iguales llegan a la compresión doble, y como se repiten 7 veces, se copia "0007" y se rellena con 0 por la derecha. Por ello los tres *buffer de contador* sólo muestran un dato de "0" y los tres *contadores de contador* muestran "70000".

El resultado de la prueba es correcto como puede verse en la figura 23.

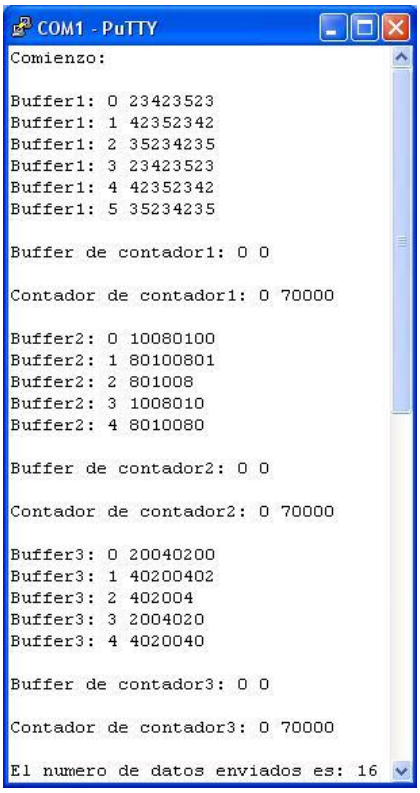


Figura 23. Resultado de prueba 2.

3. **Envío de 16 datos de forma aleatoria, donde alguno de los campos se repiten y dejan de repetirse aleatoriamente**, para comprobación del correcto funcionamiento.

   Para esta tercera prueba se han enviado los siguientes 16 datos:

1. 23410080
2. 23520040
3. 23510040
4. 23410040
5. 23420080
6. 23420080
7. 23510080
8. 23510040
9. 23420040
10. 23410080
11. 23420040
12. 23420080
13. 23520080
14. 23510080
15. 23510040
16. 23410040

Los datos se van copiando en los tres *buffers* cuando dejan de repetirse, tal y como ya se ha explicado. *Buffer1* almacena de forma alterna "234" y "235", *buffer2* almacena "0001000000" y "0010000000" y *buffer3* "0010000000" y "0001000000". De igual forma los contadores van contando las repeticiones de cada uno de los campos y se van copiando en los *buffers de contador*. Como puede verse, los tres *buffer de contador* muestran repeticiones aleatorias. Finalmente, ningún dato de *buffer de contador* se repite, y por ello los tres *contadores de contador* siempre muestran "0".

El resultado de la prueba es correcto como puede verse en la figura 24.

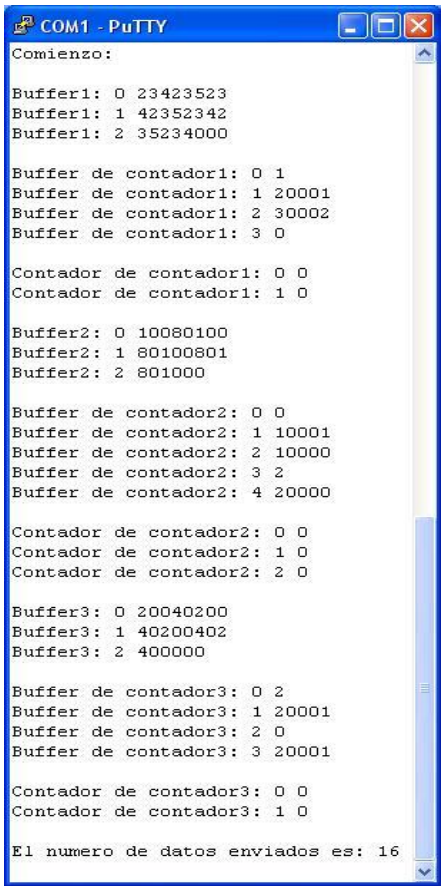


Figura 24. Resultado de prueba 3.

4. **Envío de 65538 datos iguales en sus tres campos** para comprobación de desbordamiento de contadores.

Al ser los contadores de 16 bits permiten contar hasta 65535 repeticiones en decimal. Si cuentan más de esa cifra el contador desborda y empieza a contar de nuevo desde 0. En este caso, como se han enviado tres datos de más, el primero de ellos hace que el contador se ponga en 0 y los otros dos hacen que el contador se cargue con un 2 en decimal, que sería "10" en código binario.

5. **Repetición de las pruebas 1, 2 y 3, enviando las mismas tramas de datos repetidas 5 veces** para comprobar el correcto reinicio de máquinas de estado y de procesos.

Se comprueba que se realiza un correcto reinicio de máquinas de estado, procesos y señales internas, obteniéndose los datos correctos en las 5 repeticiones.

## 5.2. ANÁLISIS DEL RENDIMIENTO DE LA ARQUITECTURA

Para realizar el cálculo del rendimiento de la arquitectura **se han repetido las pruebas 1,2 y 3 descritas en el apartado anterior, pero enviando las mismas tramas de datos repetidas 10, 50 y 100 millones de veces.**

Como ya se ha dicho, mediante el código C se puede pedir que se muestren los resultados por la pantalla a través de una consola. Se añadió una línea de código que indicase en qué momento comenzaban a enviarse los datos desde MicroBlaze hacia el coprocesador. De igual manera, cuando MicroBlaze termina de recibir todos los datos comprimidos desde los coprocesadores muestra por pantalla el número de datos enviados al coprocesador. Tomaremos esto como el fin de la compresión.

También ha habido que modificar el código C indicando que no se mostrasen por pantalla los datos recibidos por MicroBlaze, ya que esto ralentiza de manera notoria todo el proceso e impediría calcular el tiempo real.

Con el código C modificado se procede a cargar el programa en la FPGA y se activa la consola. Cuando en esta se mostrase el inicio del envío de datos se activaría un cronómetro manual para empezar a contar el tiempo y cuando en pantalla se mostrase el final de la compresión se detendría el cronómetro. Consideramos que aunque mediante éste método se puede cometer un error de cálculo de tiempo grande, es perfectamente válido para hacer una estimación del rendimiento.

Paralelamente estas mismas pruebas se han repetido con el mismo algoritmo programado en código C sobre un ordenador personal, realizándose en un equipo con las siguientes características:

- Intel Core i7, 2,97 GHz.
- S.O. Devian Squeeze.
- Bogomips: 5.866 instrucciones/seg.
- Memoria RAM: 4 GBytes.
- Memoria caché: 8 Mbytes.

En las tablas 1, 2 y 3 se muestran **los resultados obtenidos para las diferentes pruebas, tanto en hardware como en software**. Así mismo, en las figuras 25, 26 y 27 se muestra una **comparativa de rendimiento hardware frente a software en MBytes/seg.**

| | 10 millones de repeticiones - 160 millones de datos | | | |
|---|---|---|---|---|
| | Hardware | | Software | |
| | Tiempo (seg) | Rendimiento (Mbytes/seg) | Tiempo (seg) | Rendimiento (Mbytes/seg) |
| Datos iguales | 28,92 | 21,10 | 690 | 0,88 |
| Datos diferentes | 55,3 | 11,04 | 3.469 | 0,18 |
| Datos aleatorios | 57,33 | 10,65 | 6.920 | 0,09 |

Tabla 1. Análisis de rendimiento para 160 millones de datos.

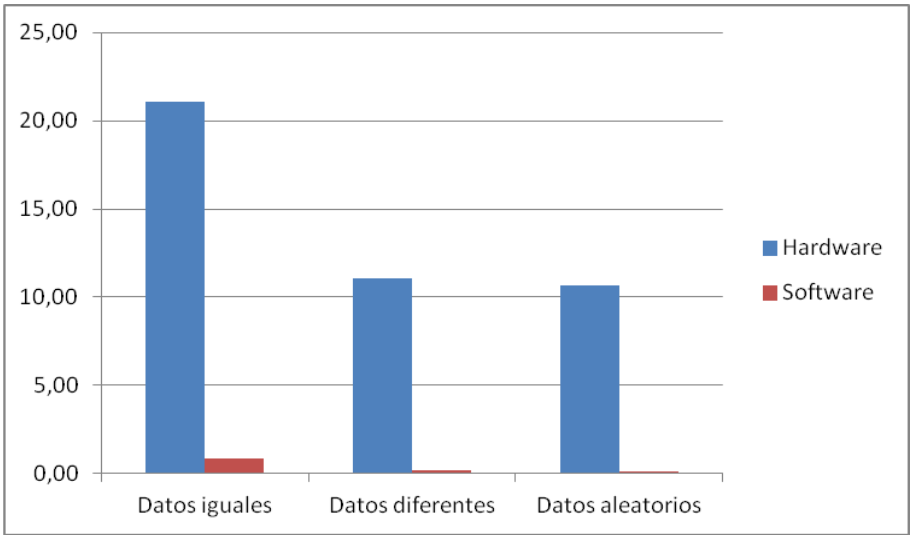


Figura 25. Comparativa de rendimiento en MBytes/seg para 160 millones de datos.

| | 50 millones de repeticiones - 800 millones de datos | | | |
|---|---|---|---|---|
| | Hardware | | Software | |
| | Tiempo (seg) | Rendimiento (Mbytes/seg) | Tiempo (seg) | Rendimiento (Mbytes/seg) |
| Datos iguales | 134,4 | 22,71 | 754 | 4,05 |
| Datos diferentes | 261,6 | 11,67 | 3.775 | 0,81 |
| Datos aleatorios | 268,2 | 11,38 | 7.543 | 0,40 |

Tabla 2. Análisis de rendimiento para 800 millones de datos.

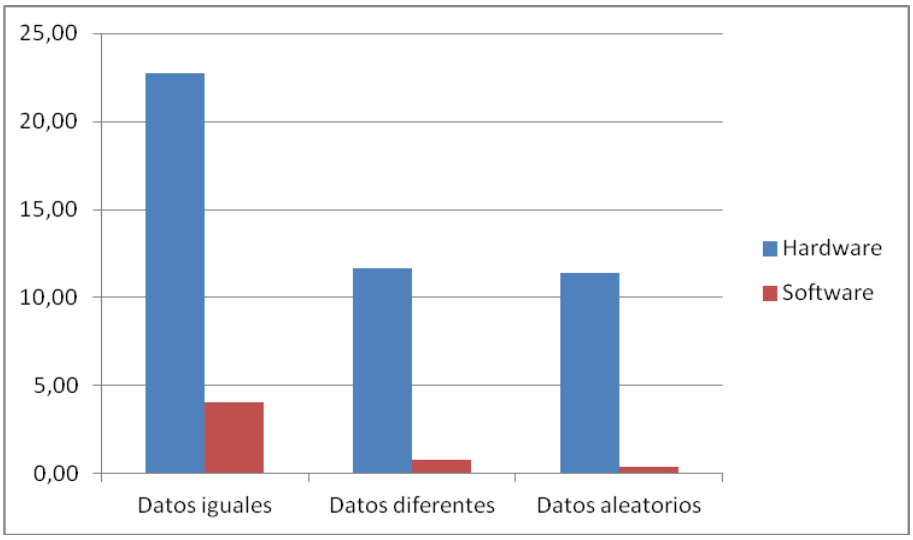


Figura 26. Comparativa de rendimiento en MBytes/seg para 800 millones de datos.

| | 100 millones de repeticiones - 1.600 millones de datos | | | |
|---|---|---|---|---|
| | Hardware | | Software | |
| | Tiempo (seg) | Rendimiento (Mbytes/seg) | Tiempo (seg) | Rendimiento (Mbytes/seg) |
| Datos iguales | 270,2 | 22,59 | 760 | 8,03 |
| Datos diferentes | 547,8 | 11,14 | 3.801 | 1,61 |
| Datos aleatorios | 560,4 | 10,89 | 7.576 | 0,80 |

Tabla 3. Análisis de rendimiento para 1.600 millones de datos.

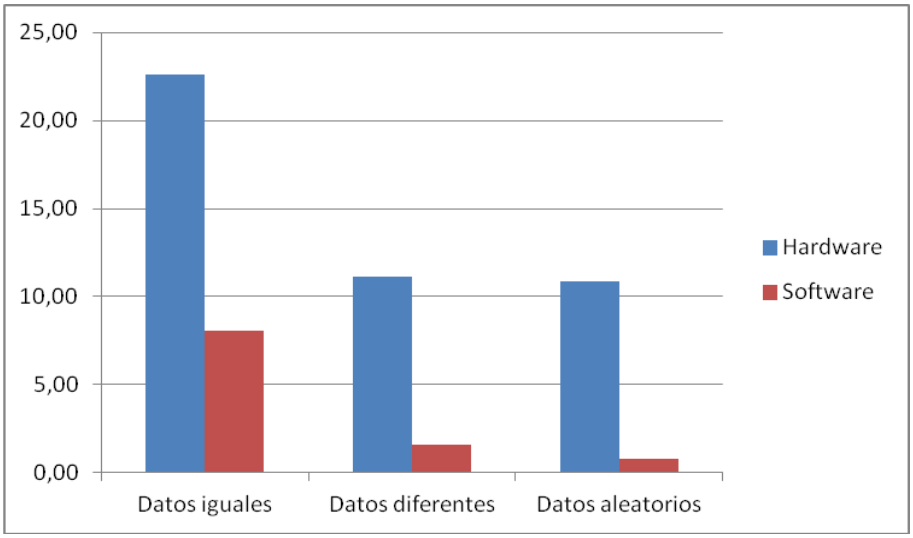


Figura 27. Comparativa de rendimiento en MBytes/seg para 1.600 millones de datos.

Para mayor comodidad al referirnos a las pruebas, llamaremos prueba A cuando sean todos los datos iguales, prueba B cuando todos sean diferentes, y prueba C cuando sean datos aleatorios. A continuación se mostrará una gráfica de comparación de rendimientos entre las pruebas A, B y C (figura 28**Figura 28**).

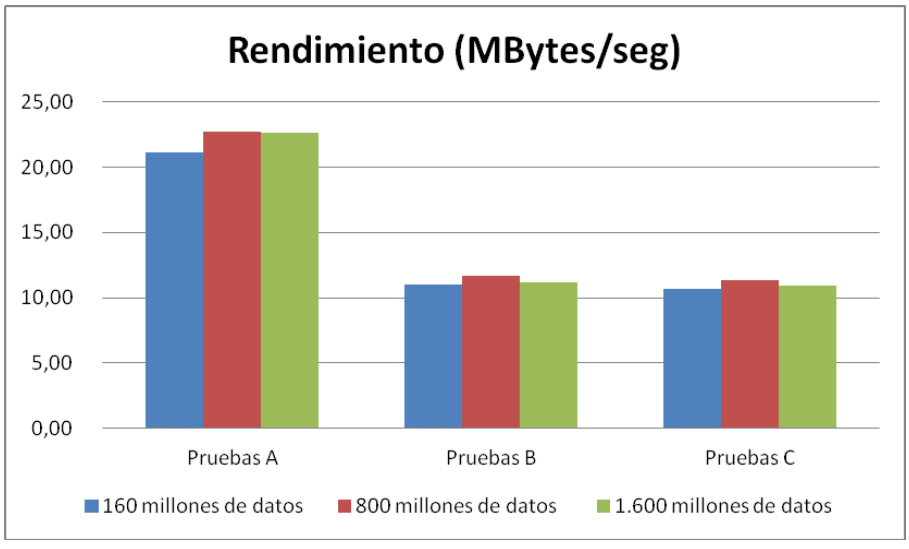


Figura 28.Comparativa de rendimientos para las pruebas A, B y C.

Como puede observarse en los resultados, y **centrándonos exclusivamente en el algoritmo hardware** se pueden extraer dos conclusiones fundamentales.

La primera es **la práctica linealidad en el rendimiento de nuestro proyecto,** como se observa en la figura 28. Si observamos, para las pruebas A se obtiene un rendimiento de entre 21 y 22,6 Mbytes aproximadamente. Para las pruebas B el rendimiento se sitúa entre 11 y 11,7 Mbytes y para las pruebas C se consigue un rendimiento de entre 10,6 y 11,4 Mbytes. No obstante ya se ha mencionado que las pruebas están sujetas a error y que lo correcto es tomarlas como una estimación.

La segunda conclusión también es clara, **cuantos más datos repetidos haya, mayor es el rendimiento**. Como puede verse, en las pruebas A el rendimiento es de casi el doble que en las pruebas B y C. Esto es completamente lógico, ya que para las pruebas A, durante la compresión simple sólo se genera un dato que enviar a la compresión doble, y por lo tanto, la compresión doble a penas si dura tiempo en comparación con la simple. Sin embargo, en la pruebas B y C, en la compresión simple se generan entre 4 y 8 datos a los que aplicar la doble compresión. De ahí que haya esas diferencias de rendimiento.

Por otro lado, si comparamos las pruebas B con las C, el rendimiento de las primeras es algo mayor. Esto se debe a que en las pruebas B, en la compresión simple se generan 8 datos iguales a los que aplicar la doble compresión. Sin embargo, en las pruebas C, pese a generarse únicamente 4 o 5 datos para la doble compresión, estos son todos diferentes. Por ello, si todos los datos de la doble compresión son iguales el tiempo es menor que si todos los datos son diferentes, pese a ser menos datos.

Finalmente, también se puede extraer otra conclusión. A tenor de la linealidad en los resultados obtenidos para distintas cantidades de datos, podemos afirmar que **con grandes cantidades de datos nuestro algoritmo mantiene su funcionamiento de forma totalmente correcta.**

Si realizamos una comparativa entre la implementación hardware y software de los algoritmos, apreciamos, en primer lugar y a tenor de los resultados queda bastante claro que nuestro algoritmo hardware tiene un rendimiento muchísimo más alto que el algoritmo software, llegando a ser de alrededor de 120 veces superior en la prueba C para 160 millones de datos.

Sin embargo, se han de añadir ciertos datos para explicar esta diferencia tan grande. La primera es que el algoritmo software no está optimizado y opera secuencialmente sobre cada campo, mientras que el algoritmo hardware opera en paralelo sobre todos los campos. La segunda es que en el algoritmo software la compresión doble no empieza hasta que ha terminado completamente la simple. En cambio, el algoritmo hardware puede realizar al mismo tiempo la compresión simple y doble de diferentes datos siempre que haya datos disponibles para ello. De ahí que el rendimiento sea tan superior.

## 5.3. ANÁLISIS DE COSTE HARDWARE DE LOS DISEÑOS

Como ya se explicó en el apartado 3, en este proyecto se han diseñado tres arquitecturas diferentes y en este apartado se pretende hacer una comparativa del coste hardware de las tres, en concreto de biestables, LUTs, bloques de E/S y DCMs.

Los bloques de E/S se utilizan para dar conectividad externa a la FPGA, ofreciendo distintos voltajes de conexión y dándole una fácil adaptación de impedancias. También incluye *buffering* de E/S.

Un DCM (del inglés *digital clock manager*) es una función que se utiliza para la manipulación y control de señales de reloj.

- **Compresor simple con un coprocesador**
  - Biestables: 3.519 utilizados de 11.776 29%.
  - LUTs de 4 entradas: 7.858 utilizadas de 11.776 66%
  - Bloques de E/S: 53 utilizadas de 372 14%
  - DCMs: 2 utilizados de 8 25%
- **Compresor doble con cuatro coprocesadores**
  - Biestables: 4.342 utilizados de 11.776 36%.
  - LUTs de 4 entradas: 11.220 utilizadas de 11.776 95%
  - Bloques de E/S: 53 utilizadas de 372 14%
  - DCMs: 2 utilizados de 8 25%
- **Compresor doble con un coprocesador**
  - Biestables: 3.774 utilizados de 11.776 32%.
  - LUTs de 4 entradas: 9.390 utilizadas de 11.776 79%
  - Bloques de E/S: 53 utilizadas de 372 14%
  - DCMs: 2 utilizados de 8 25%

Como puede observarse en los resultados, el compresor simple es el que menos biestables y LUTs utiliza, algo lógico sabiendo que sólo realiza la compresión simple.

Por otro lado, podemos comparar los dos modelos de compresión doble y observamos claramente que utilizando un único coprocesador se utilizan menos biestables y sobretodo muchísimas menos LUTs, habiendo una diferencia notable entre el 79% que utiliza frente al 95% que precisa el modelo de cuatro coprocesadores, viéndose que este último consume la práctica totalidad de LUTs disponibles en la FPGA. Esto probablemente sea debido a dos razones. La primera es que se reducen las líneas de código VHDL, así como el número de señales y compradores utilizados, y la segunda es que también se reduce el número de FSLs utilizados, ya que al sólo haber un coprocesador, únicamente son necesarios 10 FSL para comunicarse con el MicroBlaze, mientras que con cuatro coprocesadores son necesario 13 FSLs para comunicarse entre sí y con el MicroBlaze.

Finalmente, tanto el número de bloques de E/S, como el número de DCMs es el mismo para los tres modelos.

# 6. CONCLUSIONES Y TRABAJO FUTURO

En este capítulo se van a describir las conclusiones a las que se han llegado una vez se han terminado tanto el desarrollo como las pruebas ejecutadas durante la realización de este proyecto. Posteriormente se comentarán algunos de los posibles trabajos futuros que se pueden realizar basados en el proyecto realizado.

## 6.1. CONCLUSIONES

Tras la finalización del proyecto se han sacado diversas conclusiones acerca del trabajo realizado que tenemos a bien comentar a comentar a continuación.

La principal conclusión que se puede sacar es que **el objetivo principal de este trabajo, que era el diseño de un algoritmo software de compresión de datos de tamaño de palabra arbitrario en lenguaje VHDL y su posterior implementación en una FPGA, se han completado con un resultado satisfactorio**.

El principal problema encontrado a la hora de realizar el proyecto sin duda ha sido la familiarización con el entorno de desarrollo utilizado, *Xilinx Platform Studio* (XPS), pues se partía de un total desconocimiento del programa. No obstante, el manejo continuo del programa ha permitido adquirir los conocimientos necesarios para la realización del proyecto, permitiendo reducir el tiempo empleado en tareas que se han ido repitiendo a lo largo de la ejecución. En este sentido se puede decir que la familiarización con la herramienta es completa. No obstante, este programa tiene dos grandes inconvenientes. El principal es que el tiempo de síntesis de diseño es demasiado grande, sobretodo, cuando se hacen pequeños cambios. El segundo es que dispone de un programa de simulación de uso bastante complejo, ya que necesita una configuración de señales iniciales, lo cual dificulta su empleo.

Por estos dos motivos y porque existía un conocimiento previo de su manejo, es necesario mencionar que **se ha empleado también el programa *Quartus II,* cuyo uso no estaba previsto al inicio del proyecto, ha sido un total acierto**, pues nos ha permitido avanzar mucho más rápido en el desarrollo del algoritmo en VHDL gracias a su herramienta de simulación y a su mayor velocidad a la hora de realizar la síntesis del diseño. En conclusión, se puede afirmar que la utilización en conjunto de ambos programas ha sido fundamental para el correcto cumplimiento de este proyecto.

Como ya se ha comentado a lo largo de esta memoria, la idea original era realizar un código para realizar una compresión simple. Se creó una arquitectura basada en un microprocesador MicroBlaze y un coprocesador programado en VHDL que realizaría la compresión. Tras las pruebas realizadas se comprobó que el trabajo realizado era completamente funcional.

Posteriormente se añadió el objetivo de ampliar el código para hacer una doble compresión. En esta segunda parte del proyecto se encontraron diversas dificultades. En principio se iba a tomar como base la arquitectura de la compresión simple y se le añadirían tres nuevos coprocesadores que realizarían la compresión doble. Los coprocesadores estarían conectados entre sí y con MicroBlaze mediante buses de conexión llamados FSL. El primer problema surgió porque no se encontró la manera adecuada de configurar más de un FSL de salida del MicroBlaze, por lo que hubo que descartar este recurso. Sin embargo se configuraron más de un FSL de entrada al MicroBlaze, así como varios FSL de entrada o salida en los coprocesadores de manera exitosa.

Sin embargo, una vez configurado todo correctamente, surgió el principal problema que hemos tenido, que nos ha retrasado un mes intentando descubrir el error y subsanarlo, y que

desgraciadamente no se ha conseguido solucionar. El uso de cuatro coprocesadores para realizar la compresión doble no ha funcionado correctamente por lo que finalmente se buscó una alternativa.

La solución fue volver a la arquitectura principal, con un solo coprocesador que realizara tanto la compresión simple y la doble. Esta elección ha sido satisfactoria, puesto que se han logrado una arquitectura y un algoritmo completamente funcionales. Por ello concluimos que en el modelo de cuatro coprocesadores existían problemas de sincronización entre los coprocesadores que no hemos sido capaces de detectar.

A parte del algoritmo en VHDL, también se ha diseñado un código en lenguaje C, encargado del funcionamiento del MicroBlaze. Hemos obtenido resultados satisfactorios, puesto que se ha conseguido el correcto envío desde MicroBlaze y la posterior recepción de los mismos, además de mostrar los resultados por pantalla.

Tras realizar diferentes pruebas de funcionamiento se ha llegado a la conclusión de que la puesta en conjunto del algoritmo VHDL, la arquitectura diseñada y el código C para el control del MicroBlaze ha sido completamente satisfactorio.

Por otro lado, se ha realizado una evaluación de rendimiento y consideramos que la compresión se realiza de manera rápida y eficiente. Además, **se ha realizado un análisis de coste hardware y comparando el modelo de cuatro coprocesadores con el de uno sólo convenimos en que hemos acertado** también porque mientras el primer modelo utiliza el 36% de biestables y el 95% de las LUTs de 4 entradas disponibles, el modelo final reduce el número de biestables al 32% y sobretodo el número de LUTs al 79%.

Si nos atenemos a la comparativa del rendimiento hardware frente a software y aun teniendo en cuenta las explicaciones realizadas en el apartado 5.2. de posibles motivos para que exista esa gran diferencia de rendimientos podemos afirmar que se ha realizado un gran trabajo ya que para todas las pruebas realizadas nuestro algoritmo ha tenido un rendimiento muy superior al software y consideramos que hemos realizado un diseño muy eficaz.

Por último, también se ha realizado una documentación a modo de tutorial del manejo de la herramienta XPS. La idea era hacer un manual fácil de entender apoyado en capturas de pantalla. **Consideramos que con el trabajo realizado una persona con ciertos conocimientos de lenguaje C y de lenguaje VHDL y absoluto desconocimiento del XPS y del manejo de una FPGA puede llegar a desarrollar un diseño perfectamente funcional**, por lo que concluimos que este objetivo también se ha logrado.

A título personal he de admitir que me siento muy satisfecho con el trabajo realizado. Lo principal es que el proyecto encajaba perfectamente con el tipo de trabajo que quería realizar. La programación es un campo que me resulta muy interesante, sobre todo desde aprendí el manejo del lenguaje VHDL y de la herramienta Quartus II en la asignatura “Tecnología Electrónica II”. Mis conocimientos de ambas me han permitido avanzar a buen ritmo durante la realización del proyecto.

Por otro lado, he mejorado el manejo del lenguaje C, aunque en este proyecto el código realizado ha sido bastante sencillo, pero he sentado bases para trabajos futuros. De igual manera, considero que el aprendizaje del uso de XPS y del manejo de una FPGA son conocimientos muy prácticos y que pueden llegar a ser realmente útiles cuando me incorpore al mundo laboral.

La mayor satisfacción personal ha sido comprobar que el algoritmo desarrollado tiene un rendimiento altamente superior al algoritmo software existente y que me hace pensar que este proyecto ha sido un éxito.

Sin embargo, para mi realmente ha supuesto una decepción que el modelo de cuatro coprocesadores no haya funcionado, porque el algoritmo en VHDL es totalmente funcional, ya que es el mismo código que se ha implementado exitosamente en el modelo final con un solo coprocesador.

## 6.2. TRABAJOS FUTUROS

En este apartado se pretende realizar una serie de sugerencias de mejoras o trabajos futuros que puedan realizarse en base al trabajo realizado en este proyecto. Algunas de ellas no se han realizado por falta de tiempo, otras por falta de recursos o documentación y otras porque no entraban en el plan inicial de trabajo.
Nuestras propuestas son las siguientes:

1. Sin lugar a dudas, nuestra propuesta principal es conseguir el correcto funcionamiento del modelo de cuatro coprocesadores eliminando el problema de falta de asincronía entre los diferentes coprocesadores. También proponemos que se encuentre la correcta forma de configurar más de un FSL de salida de MicroBlaze.
2. Consideramos que sería muy interesante el desarrollo de un algoritmo de descompresión complementario a nuestro algoritmo de compresión y que posteriormente se implementen conjuntamente.
3. Proponemos también que se realice una mejora en el algoritmo VHDL que mejore el rendimiento, reduciendo tanto el tiempo de ejecución como los costes hardware.
4. Se había previsto integrar todo el diseño en Petalinux, pero por falta de tiempo provocada por los problemas con el modelo de cuatro coprocesadores no se ha podido realizar. Igualmente lo añadimos a las sugerencias.
5. También por falta de tiempo no se ha realizado la medición de energía consumida por la placa Spartan 3 durante la ejecución del programa y creemos que pueden obtenerse resultados interesantes para tratar de reducir costes energéticos.
6. Por último sugerimos la implementación de nuestro modelo en FPGAs diferentes de la Spartan 3 para posteriormente comparar tiempos, rendimientos, consumos de energía y recursos hardware, etc.

# 7. PRESUPUESTO

En este presupuesto se muestra una estimación de los gastos que ha conllevado la realización de este proyecto, quedando reflejados los costes de software y hardware, así como gastos del personal participante en el proyecto.

La duración completa del proyecto ha sido de 5 meses, abarcando desde Octubre de 2011 hasta Febrero de 2.012.

- **COSTES DE PERSONAL**

La realización de este proyecto ha contado con la participación de un ingeniero junior y dos ingenieros sénior durante sus 5 meses de duración.

El ingeniero junior ha trabajo 600 horas y se ha estimado que su salario es de 25 €/hora.

Los dos ingenieros sénior han tenido una participación de 100 horas cada uno y se ha estimado que su salario es de 35 €/hora.

| Concepto | Sueldo/hora | Horas | Total |
|---|---|---|---|
| Ingeniero junior | 16,00 € | 600 | 9.600,00 € |
| Ingeniero sénior | 25,00 € | 200 | 5.000,00 € |
| | | **TOTAL** | **14.600,00 €** |

Tabla 4. Costes de personal.

- **COSTES DE HARDWARE**

En este apartado se incluyen el coste de la plataforma de desarrollo *Spartan 3A Starter Kit* y el coste del ordenador utilizado, en cuyo precio se incluye la licencia del sistema operativo Windows XP.

| Concepto | Precio |
|---|---|
| Ordenador, incluyendo sistema operativo Windows XP | 517,90 € |
| Plataforma de desarrollo Spartan 3A Starter Kit | 143,21 € |
| **TOTAL** | **652,11 €** |

Tabla 5. Costes de Hardware.

- **COSTES DE SOFTWARE**

En este apartado se incluye el coste de las licencias de los diferentes programas utilizados para la realización del proyecto, que son *Quartus II, Xilinx Paltform Studio*, el cual está incluido en el pack *Xilinx ISE Design Suite*, y finalmente, *Microsoft Office Standard 2.007*.

Es necesario indicar que **las licencias adquiridas permiten la instalación de los programas en todos los equipos que se desee** y por lo tanto el precio podría dividirse de manera proporcional entre el número de equipos en los que fuesen a utilizarse dichos programas.

| Concepto | Precio |
|---|---|
| Quartus II | 2.995,00 € |
| Xilinx ISE Design Suite | 2.861,41 € |
| Microsoft Office Standard 2007 | 569,00 € |
| **TOTAL** | **6.425,41 €** |

Tabla 6. Costes de software.

- **PRESUPUESTO TOTAL**

Se establece un 7% sobre el total dedicado al riesgo del proyecto y un 15% de beneficio.

| Concepto | Precio |
|---|---|
| Costes de personal | 14.600,00 € |
| Costes de hardware | 652,11 € |
| Costes de software | 6.425,41 € |
| Total | 21.677,52 € |
| Riesgo (7%) | 1.517,43 € |
| Beneficio (15%) | 3.251,63 € |
| **PRECIO TOTAL DEL PROYECTO** | **26.446,58 €** |

Tabla 7. Presupuesto total.

El presupuesto total de ejecución de este proyecto asciende a la cantidad de **VEINTISEIS MIL CUATROCIENTOS CUARENTA Y SEIS EUROS CON 58 CENTIMOS incluyendo el impuesto sobre el valor añadido.**

**26.446,58 € (IVA incluido)**

# 8. BIBLIOGRAFÍA

**[Alegsa]**

Alegsa. Información acerca de las FPGA y sus aplicaciones.

*http://www.alegsa.com.ar/Dic/fpga.php*

**[ALT1040]**

Alt1040. Información acerca de las FPGA y sus aplicaciones.

*http://alt1040.com/2010/09/fpga-y-el-sorprendente-poder-del-hardware-reconfigurable*

**[MicroC - FPGA]**

Microcontoladores Pic. Información acerca de la estructura interna de una FPGA:

*http://www.microcontroladorespic.com/tutoriales/FPGAs/estructura-configuracion.html*

**[MicroC - MB]**

Microcontroladores Pic. Información acerca del microprocesador MicroBlaze:

*http://www.microcontroladorespic.com/tutoriales/FPGAs/Procesador-MicroBlaze.html*

**[MicroC - XPS]**

Microcontroladores Pic. Información acerca de la herramienta *Xilinx Platform Studio* (XPS) de Xilinx:

*http://www.microcontroladorespic.com/tutoriales/FPGAs/Procesador-MicroBlaze.html*

**[MPI2010]**

Información acerca de algoritmos de comunicación:

*Tesis doctoral "Técnicas de optimización dinámicas de aplicaciones paralelas basadas en MPI" realizada en 2.010 por Rosa Filgueira Vicente en la universidad Carlos III de Madrid.*

**[TE]**

Trenz Electronic. Información acerca de la FPGA *Spartan 3A*:

*http://shop.trenz-electronic.de/catalog/product_info.php?products_id=130*

**[W - AdC]**

Wikipedia. Información acerca de algoritmos de comunicación:

*http://es.wikipedia.org/wiki/Compresi%C3%B3n_de_datos*

*http://en.wikipedia.org/wiki/Data_compression*

**[W - FPGA]**

Wikipedia. Información acerca de las FPGA y sus aplicaciones.

*http://es.wikipedia.org/wiki/Field_Programmable_Gate_Array*

**[W - MB]**

Wikipedia. Información acerca del microprocesador MicroBlaze:

*http://en.wikipedia.org/wiki/MicroBlaze*

**[W - QII]**

Wikipedia. Información acerca de la herramienta *Quartus II* de Altera:

*http://es.wikipedia.org/wiki/Quartus_II*

**[Xilinx]**

Xilinx. Información acerca del bus de comunicación FSL:

*http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20.pdf*

# APÉNDICE A. PASOS EN LA IMPLEMENTACIÓN DEL DISEÑO DE UNA FPGA

En este apéndice se pretende explicar paso a paso como desarrollar un diseño mediante el programa XPS y su posterior implementación en una FPGA, todo ello con el uso de capturas de pantalla del programa.

El primer paso es abrir el programa. Al entrar normalmente sin pulsar nada aparece la ventana de la figura 29, donde elegiremos utilizar el asistente de creación de proyectos (*Base System Builder Wizard*), iniciar un proyecto totalmente desde cero o abrir un proyecto ya creado. Si no aparece la ventana, en la esquina superior izquierda pinchamos en *File - New Project*.

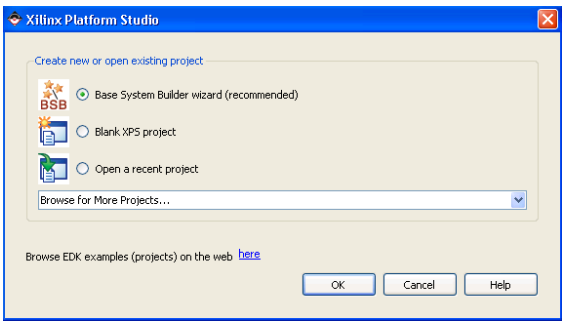


Figura 29. Asistente de creación de proyecto.

Seleccionamos *Base System Builder wizard* y pulsamos OK. Surge la ventana de la figura 30.

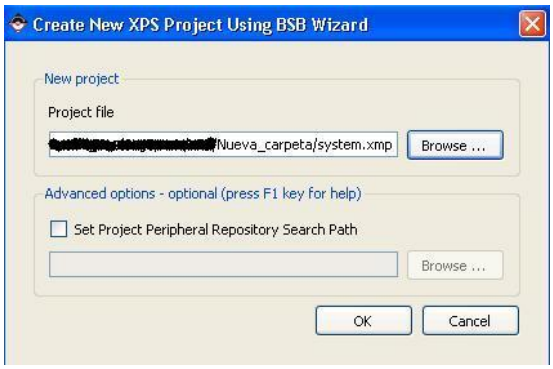


Figura 30. Seleccionar carpeta de destino de proyecto.

Se pincha en *Browse…* y se selecciona la carpeta donde se guardarán los archivos de nuestro proyecto. En nuestro caso hemos creado una carpeta llamada *Nueva_carpeta* (el nombre de la carpeta o subcarpeta elegida no puede contener espacios porque si no, no permitirá después abrir el proyecto). Así mismo ponemos nombre a nuestro proyecto, que por defecto será *system* (el nombre tampoco puede contener espacios). Lo demás se deja en blanco y pulsamos OK. Aparece la ventana de la figura 31.

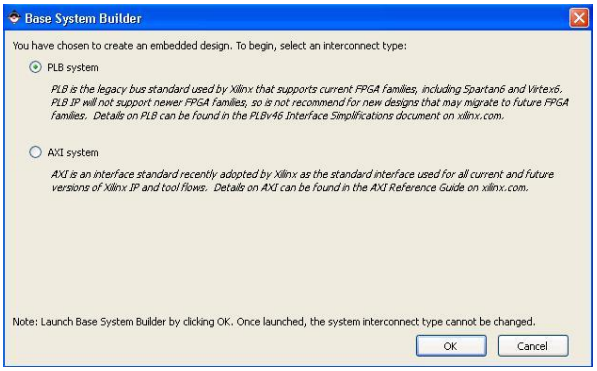


Figura 31. Tipo de interconexión.

Seleccionamos *PLB system* y pulsamos OK. A continuación surge la interfaz que aparece en la figura 32. Con esta interfaz configuramos la placa que vamos a utilizar, donde seleccionaremos la FPGA, sus periféricos, su velocidad, etc. Como podemos ver en la figura 32, en la pestaña *Welcome* seleccionamos *I would like to create a new design* y pulsaremos *Next*, en la esquina inferior derecha de la venta (no aparece en la figura 32).

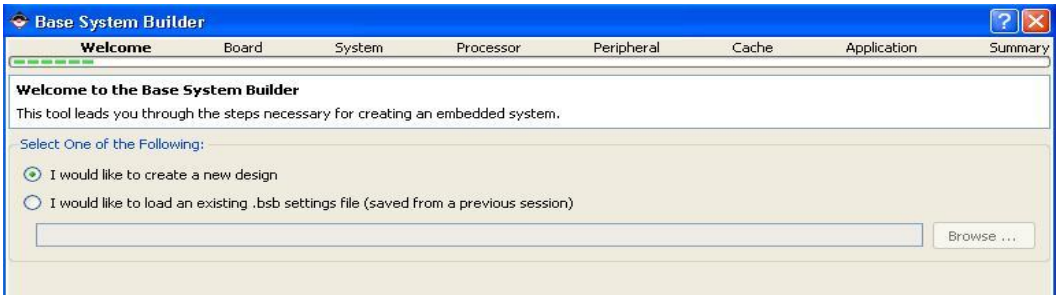


Figura 32. Pestaña *Welcome*.

Pasamos a la pestaña *Board* (figura 33) y aquí seleccionamos la placa que vamos a utilizar. Tenemos dos opciones, elegir una placa ya configurada por el fabricante (*I would like to create a system for the following developed board*) o crear nosotros una propia (*I would like to create a system for a custom board*). En nuestro proyecto hemos utilizado una placa *Xilinx - Spartan 3A Starter Kit - D*, ya configurada.

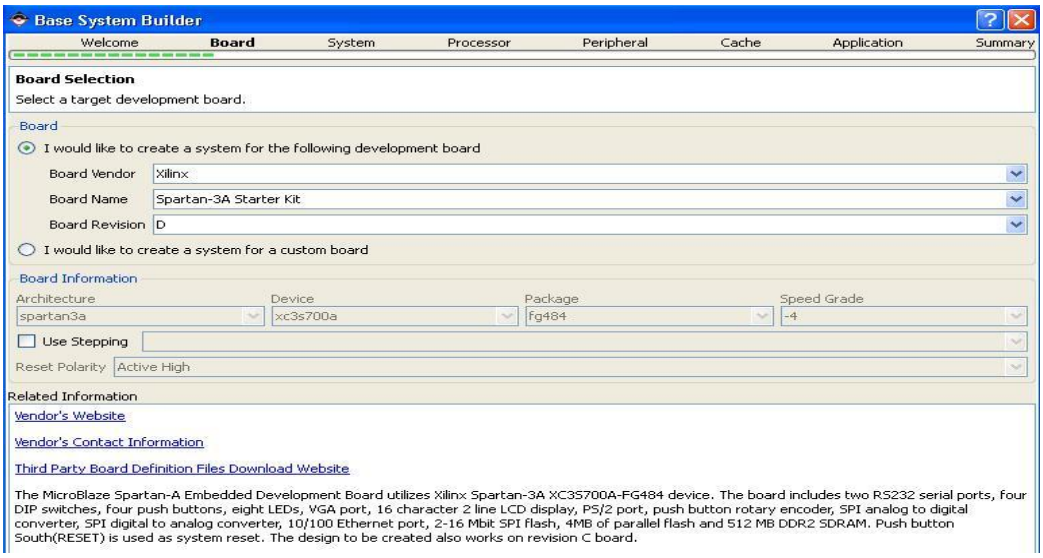


Figura 33. Pestaña *Board*.

Pulsamos *Next* y llegamos a la pestaña *System* (figura 34). Por defecto vendrá marcado *Single-processor System*, es decir, solo usamos un procesador. Con la opción *Dual-processor System* utilizaríamos dos procesadores. Pulsamos *Next* y aparece la pestaña *Processor* (figura 35). Aquí seleccionamos la frecuencia del reloj, el tipo de procesador, y el tamaño de la memoria local. En la figura 35 se observan las elecciones de nuestro proyecto. Nuevamente se pulsa *Next* y se llega a la pestaña *Peripheral* (figura 36). Como se ve, hay dos recuadros. En el de la izquierda aparecen los periféricos que trae por defecto la placa que hemos seleccionado. En el lado izquierdo aparecen otros periféricos que pueden ser añadidos, seleccionándolos y pulsando *Add*. En la figura 36 se aprecia qué periféricos hemos añadido, pues están sombreados en azul, y tras pulsar *Add* aparecen en la parte inferior del recuadro derecho.

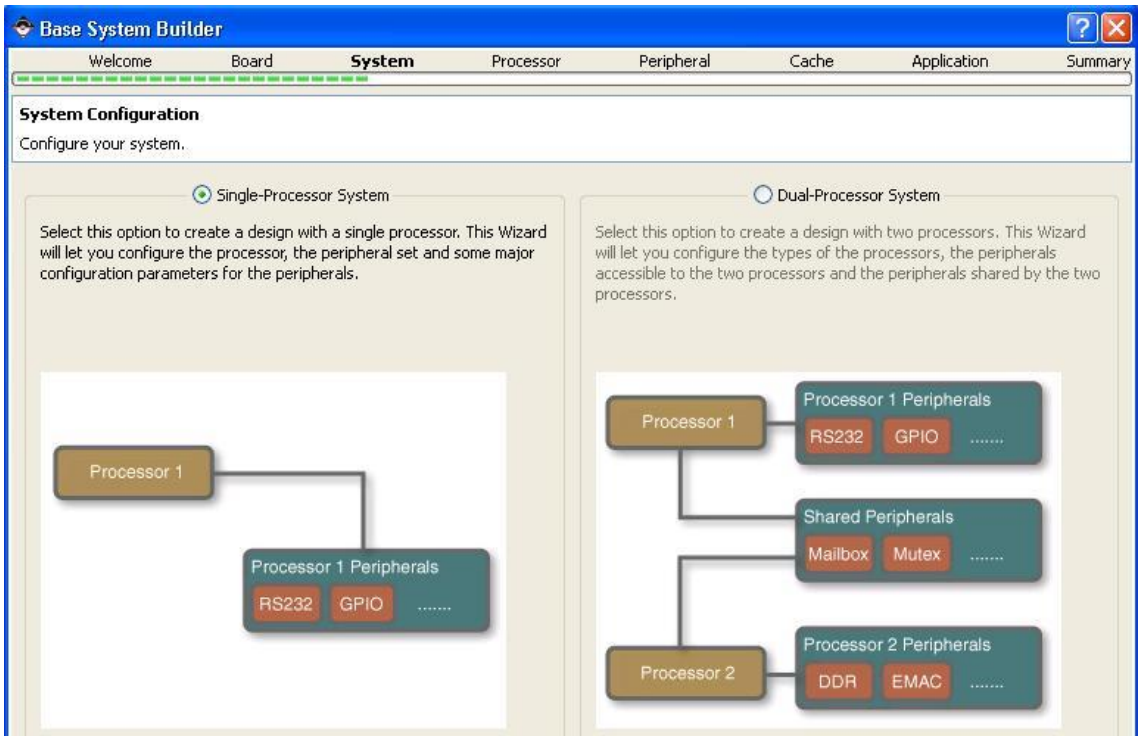


Figura 34. Pestaña *System*

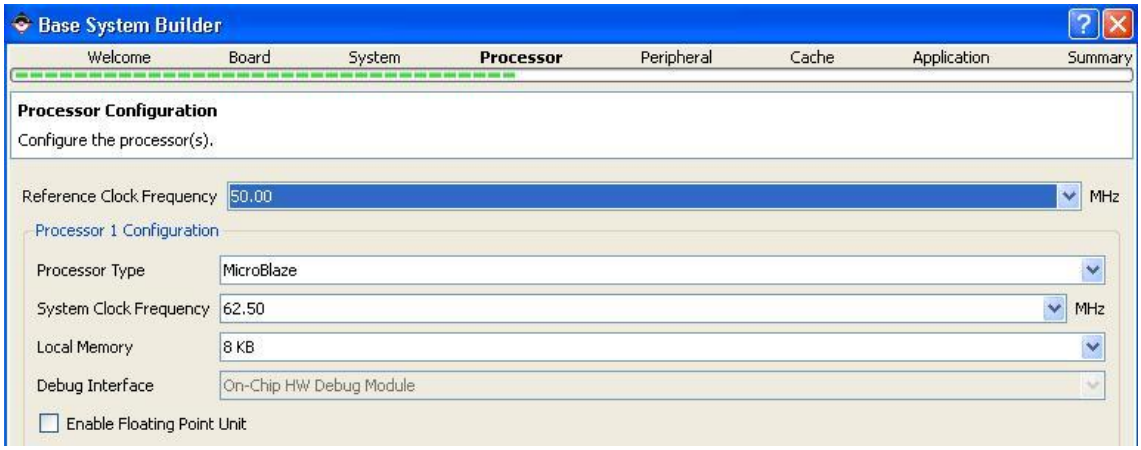


Figura 35. Pestaña *Processor.*

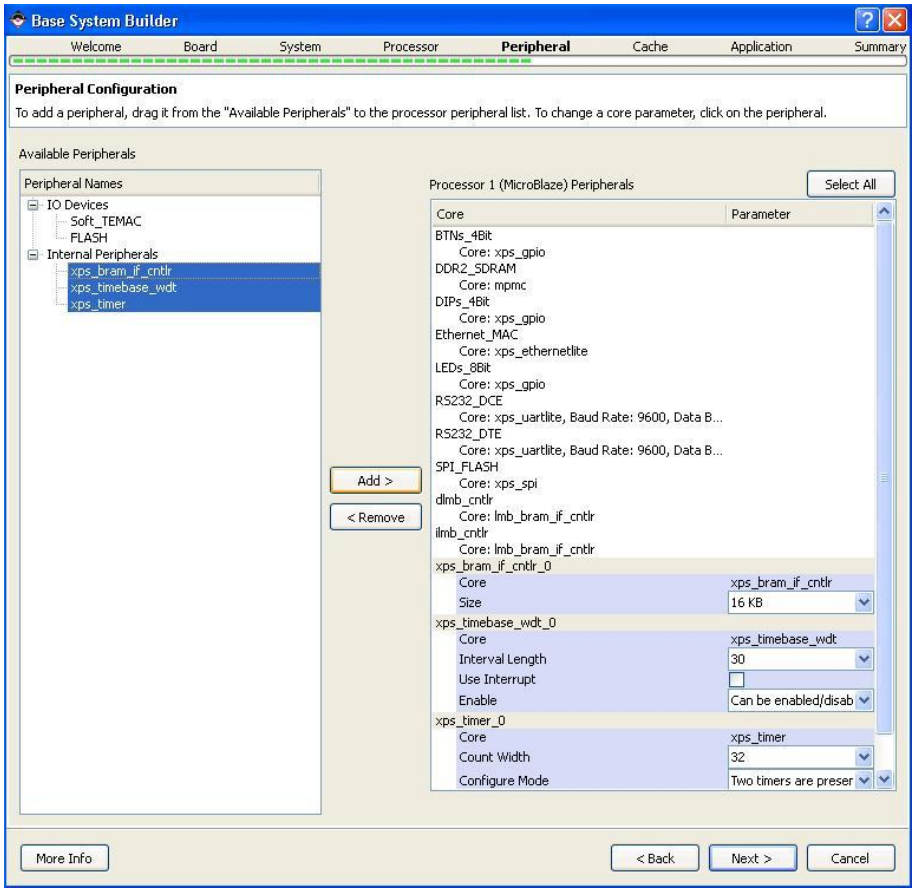


Figura 36. Pestaña *Peripheral*.

Tras seleccionar los periféricos se pulsa *Next*, y se llega a la pestaña *Cache*, donde no se selecciona nada y se pulsa directamente *Next*. En la pestaña *Application* tampoco seleccionamos nada y pulsamos directamente *Next*. Finalmente llegamos a *Summary* donde aparece un resumen de la completa configuración de nuestra placa. Pulsamos *Finish* donde antes aparecía *Next* y se cerrará la interfaz.

Es posible que surjan una o dos ventanas que nos ofrezcan diversas opciones, pero que no se han utilizado, por lo que daremos a cancelar. Así pues nos quedará una pantalla del programa como la de la figura 37. Ya tenemos totalmente configurada la placa que vamos a usar.

Como se ve en la figura 37, en el lado izquierdo podemos ver los proyectos software que tiene nuestro proyecto. El programa *Xilinx Platform Studio* proporciona dos programas software de test para comprobar que hemos configurado bien nuestra placa y que funciona. Ahora debemos crear nuestro propio proyecto software de aplicación. Seleccionamos la pestaña *Software* y *Add software application Project* (figura 38).

Aparece una ventana donde daremos nombre a nuestro proyecto software, en nuestro caso lo hemos llamado *Codificador* y por defecto aparece como procesador *microblaze_0* (figura 39). Pulsamos OK.

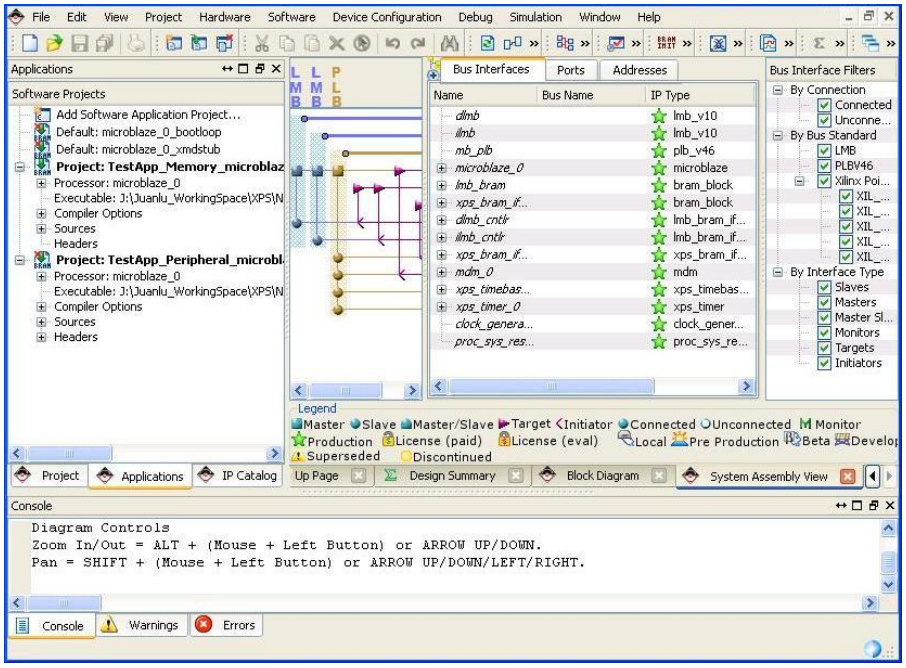


Figura 37. Captura de pantalla con la plataforma configurada.


Figura 38. Crear proyecto software de aplicación.

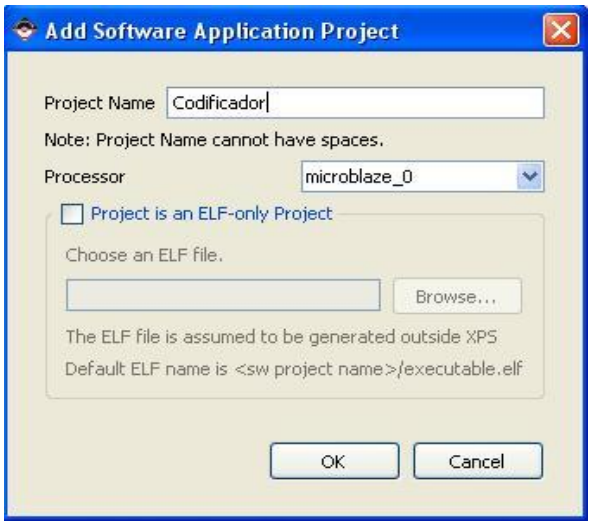


Figura 39. Nombre del proyecto de aplicación.

Ahora nuestro proyecto software aparece bajo los dos proyectos de test en el lado izquierdo. Así mismo en el recuadro central, en *System Assembly view* podemos observar que se ha añadido *microblaze_0*. Es momento de añadir los coprocesadores. Con el botón derecho pinchamos en *microblaze_0* y seleccionamos *Configure Coprocessor…* (figura 40).

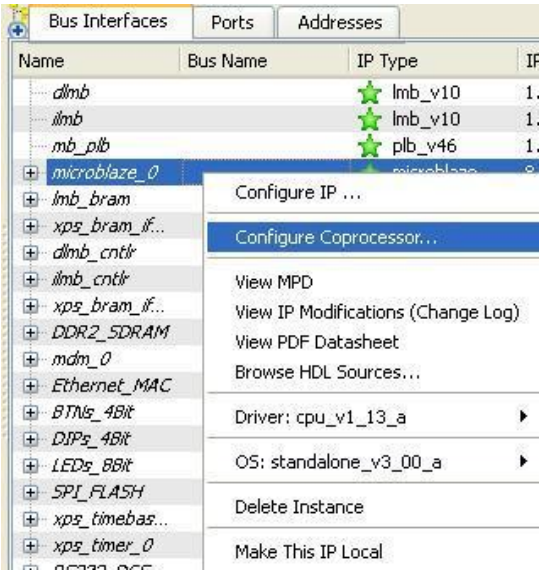


Figura 40. Configurar coprocesador.

Como es el primer coprocesador que añadimos al proyecto nos saldrá la advertencia que se puede ver en la figura 41. Nos dice que de momento no existe ningún coprocesador, por lo que nos ofrece la opción de crearlo como un periférico, con lo que pulsamos *Yes* y aparece la interfaz de creación o importación de un periférico (figura 42). Seleccionaremos *Create templates for a new peripheral* y pulsaremos *Next*.

Llegaremos os a una ventana como la de la figura 43. Aquí decidimos si queremos añadir el periférico a un repositorio EDK o a un proyecto XPS. Seleccionamos la segunda opción y por defecto nos aparece la carpeta de nuestro proyecto. Si no, pulsamos *Browse* y la buscamos.


Figura 41. Crear nuevo coprocesador.

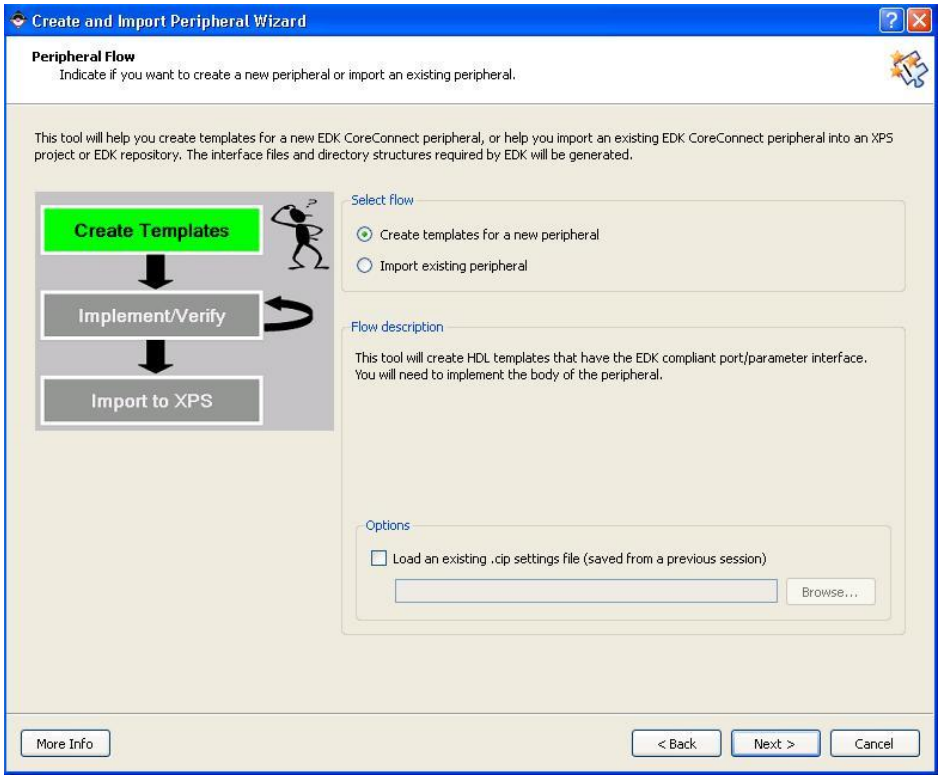


Figura 42. Crear plantilla para nuevo periférico.

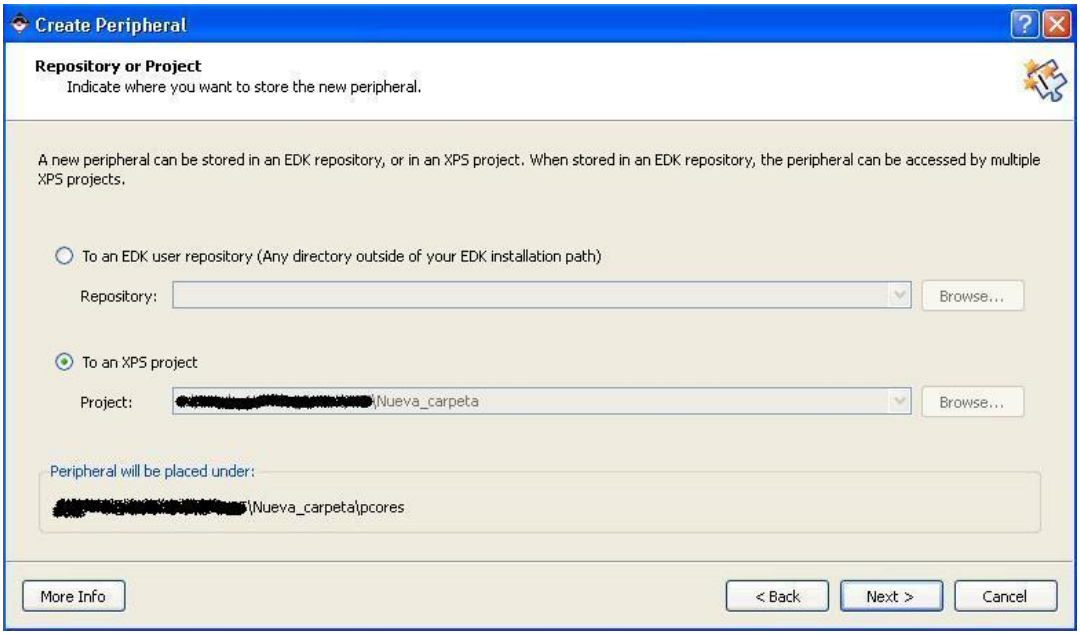


Figura 43. Añadir periférico a repositorio EDK o a proyecto XPS.

Debajo nos indica que el periférico se almacenará en una subcarpeta llamada *pcores*. Pulsamos *Next*.

Ahora daremos nombre a nuestro periférico, en nuestro caso *primario*. No se permiten letras mayúsculas ni espacios en el nombre. El resto de opciones las dejamos por defecto. Abajo se puede ver que para nuestro periférico se crea una librería lógica llamada *primario_v1_00_a* (figura 44). Pulsamos *Next*.

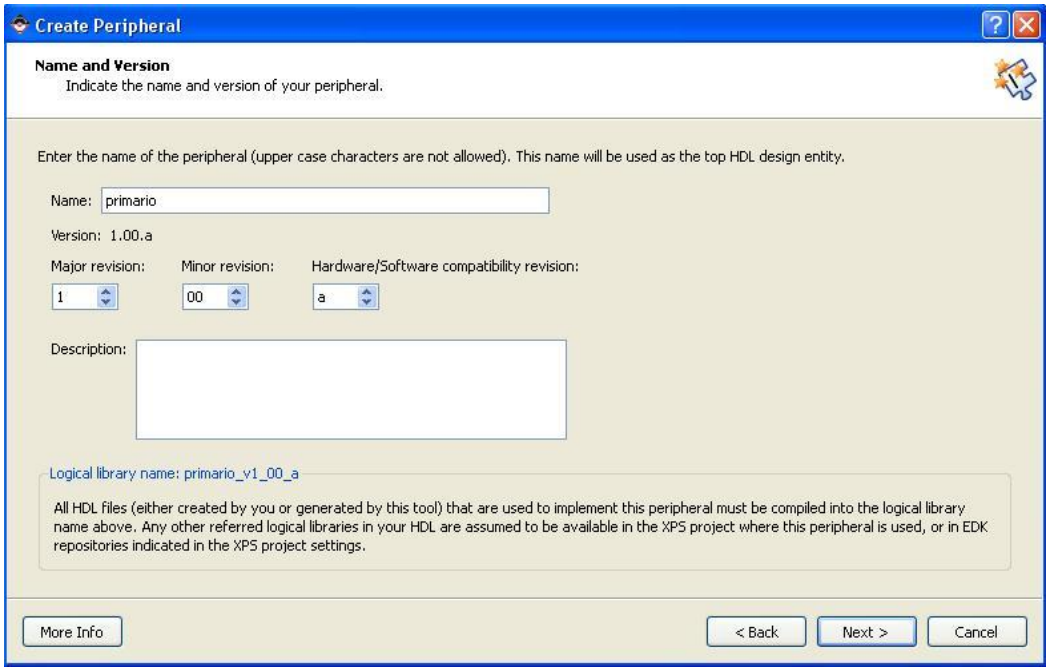


Figura 44. Nombre de periférico.

Lo siguiente es decidir qué tipo de bus vamos a utilizar, un Bus Local o un FSL. Seleccionamos FSL (figura 45) y pulsamos *Next*.

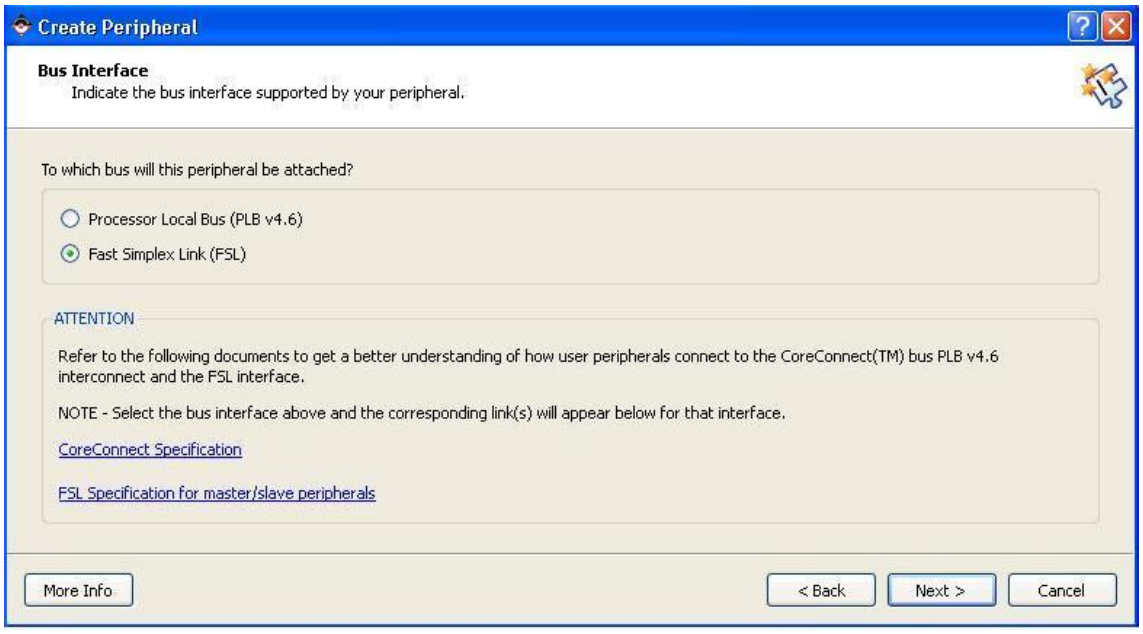


Figura 45. Seleccionar el tipo de bus.

En la siguiente ventana se configura la interfaz de entrada y salida del FSL (figura 46). Por defecto la interfaz de salida no está seleccionada, por lo que nosotros debemos seleccionarla. Así mismo debemos decir cuantas palabras de 32 bits se pueden enviar a la vez tanto en el FSL de entrada como en el de salida. En nuestro caso hemos puesto una palabra en ambos. Pulsamos *Next*.

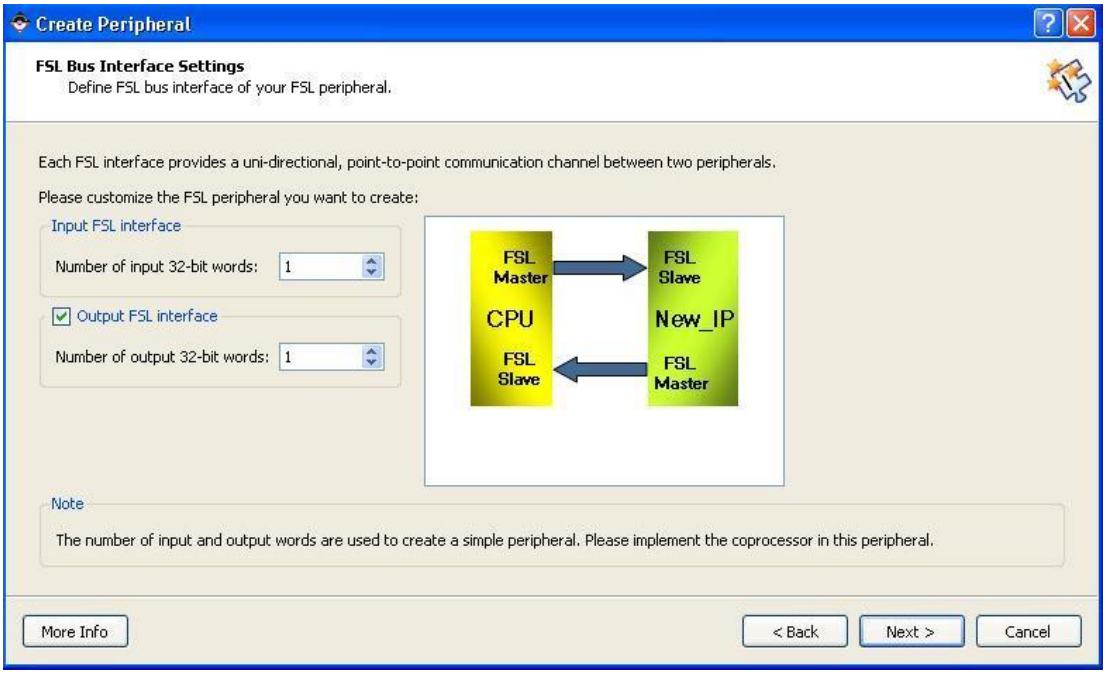


Figura 46. Ancho de banda del FSL.

Llegamos ahora a la ventana de la figura 47. Aquí, podemos seleccionar si queremos que se genere algún tipo de archivo específico acerca del diseño del periférico. Nosotros únicamente seleccionamos la última opción, *Generate template driver files to help you implement software interface*. Con esto se generará una carpeta llamada *driver* donde se alojarán los archivos referentes al periférico creado. Estos archivos nos facilitará la posterior configuración del periférico. Pulsamos *Next*.

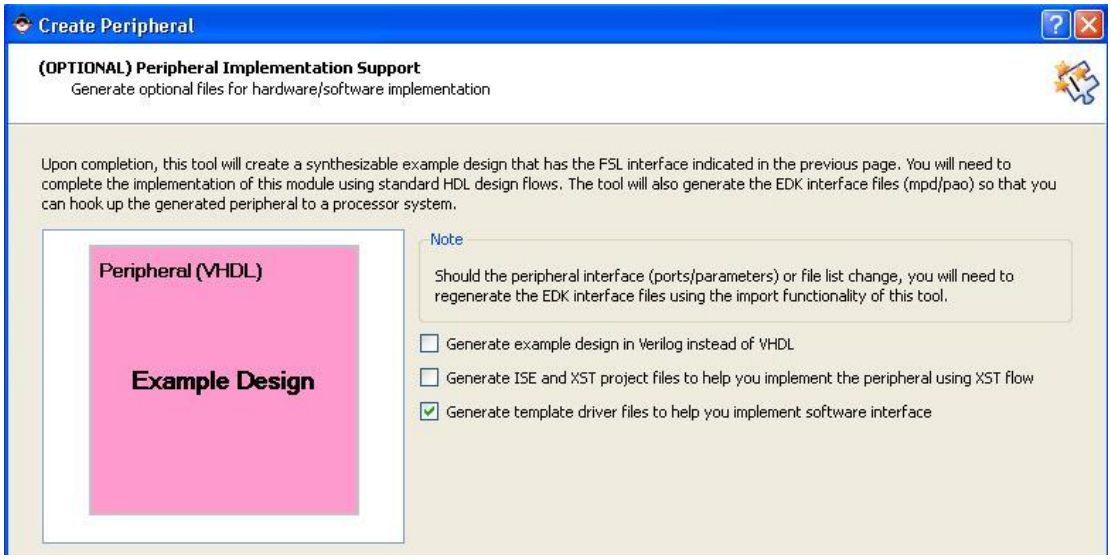


Figura 47. Generar archivos de periférico.

Aparece la penúltima ventana de creación del periférico (figura 48). Aquí se describe la función que va a tener el periférico y nos indica que se generará un *driver API* para el periférico. Por defecto aparecerá como en la figura 48 y no ha de modificarse nada, simplemente se pulsa *Next*.

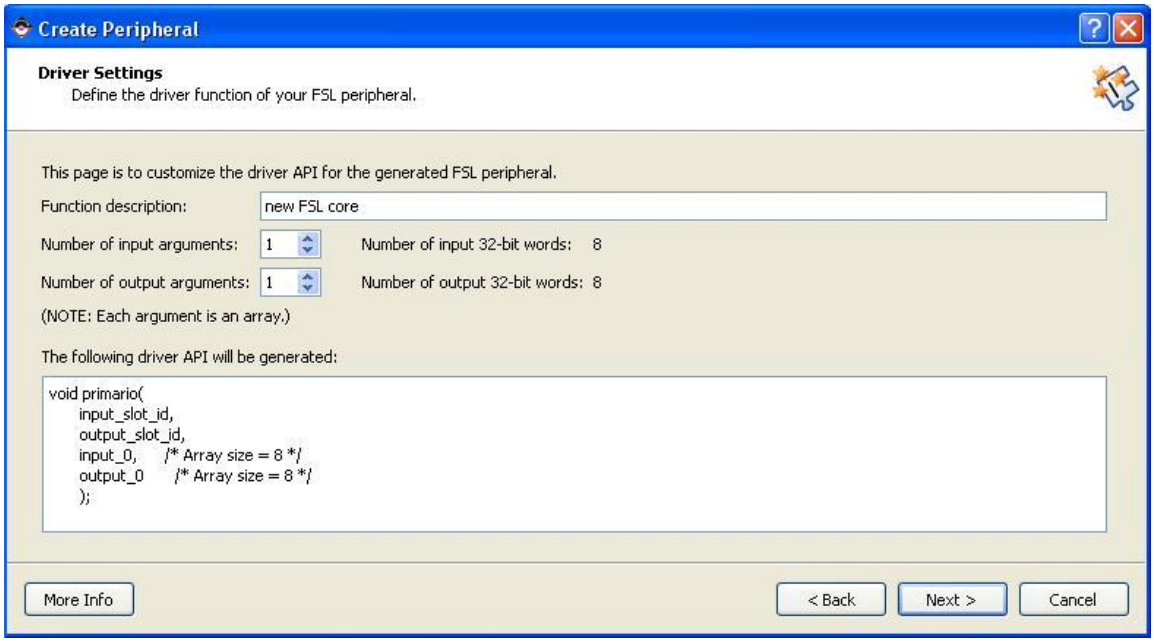


Figura 48. Función del periférico.

Finalmente llegamos al último paso de la creación de periférico. Como se ve en la figura 49, tenemos un resumen de la configuración del periférico. Pulsaremos *Finish* y el periférico estará listo para usarse.

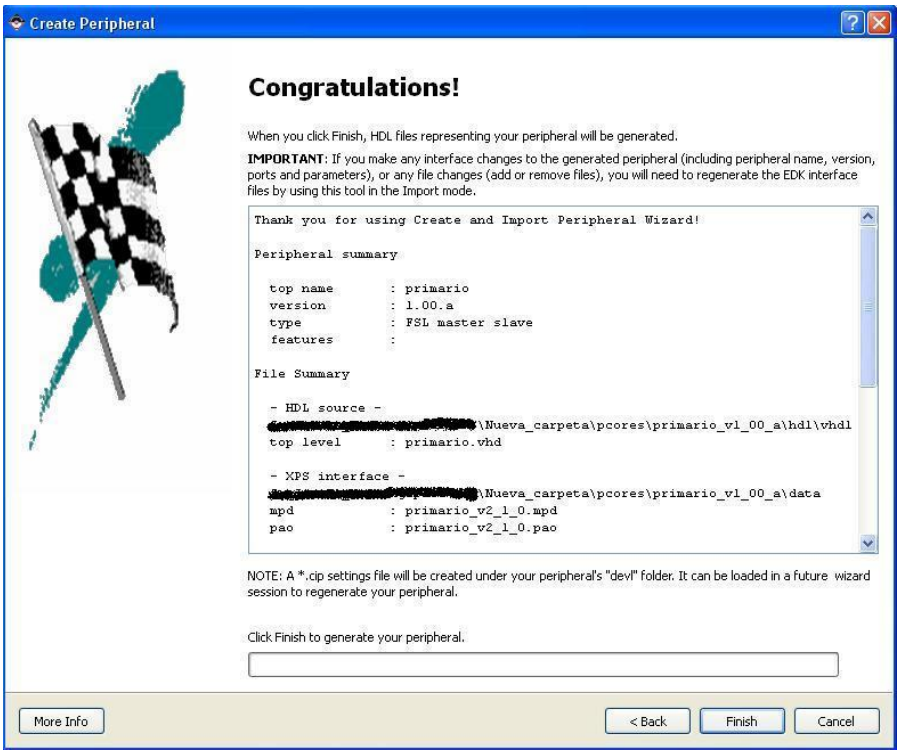


Figura 49. Periférico creado.

Una vez creado el periférico hemos de añadirlo al proyecto. En *System Assembly View* pulsaremos con botón derecho sobre *microblaze_0* y seleccionaremos *Configure Coprocessor…*, igual que en la figura 40. Aparecerá una interfaz como en la figura 50.

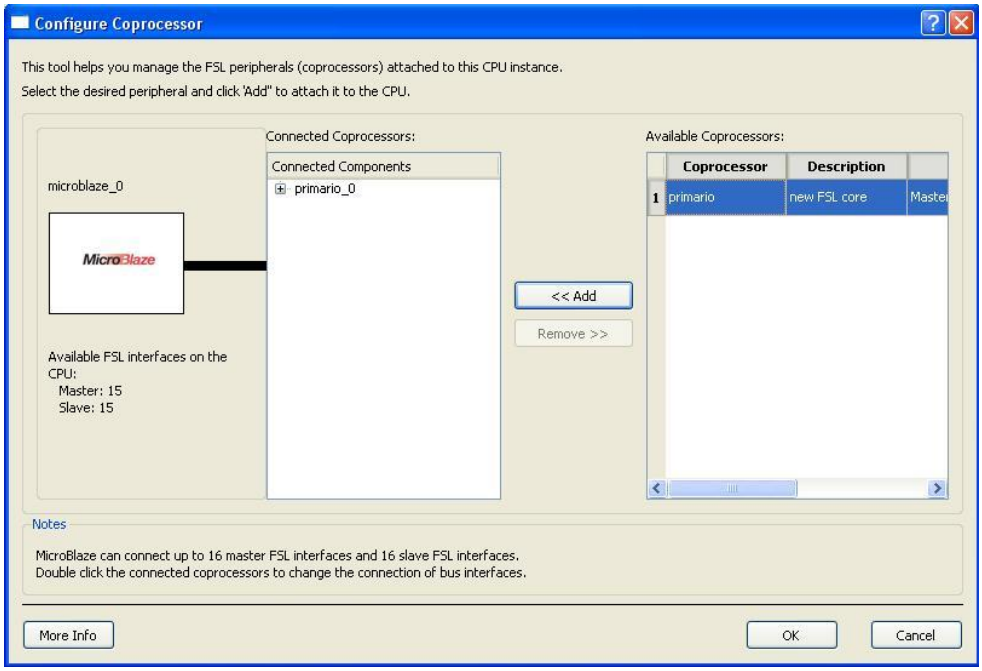


Figura 50. Creación de maestro y esclavo del FSL del coprocesador.

En el cuadro de la derecha seleccionaremos *primario*, pulsaremos *Add* y daremos a OK. Ahora el Microblaze ya tiene sus FSL esclavo y maestro, que deben ser conectados.

En *System Assembly View* abrimos *microblaze_0* y veremos que ya aparecen MFSL0 y SFLS0. En la pestaña de MFSL0 pulsaremos *New connection* y en la de SFSL0 también, como en la figura 51.

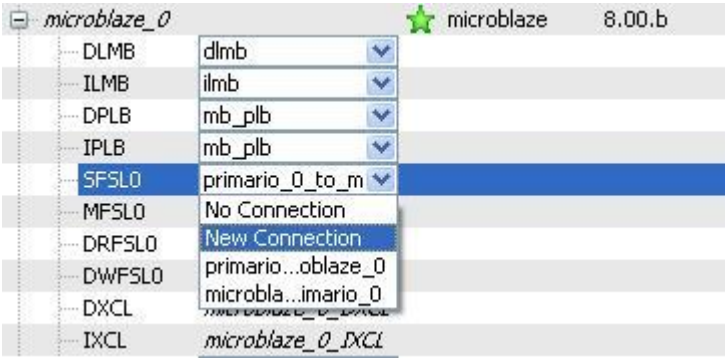


Figura 51. Conexión de coprocesador con MicroBlaze.

Veremos que por defecto nos aparecen *fsl_v20_0* y *fsl_v20_1* para cada uno de ellos (dependiendo del orden en que lo hagamos). Ya tenemos creados los FSL Maestro y Esclavo del MicroBlaze. Ahora hay que conectar el Maestro del MicroBlaze con el Esclavo del coprocesador *primario* y viceversa. Para ello, en *System Assembly View* abrimos *primario* y en la pestaña de MFSL seleccionamos *fsl_v20_1* y en la de SFLS seleccionamos *fsl_v20_0*. Quedará como en la figura 52. Ya están conectados completamente MicroBlaze y *primario*.

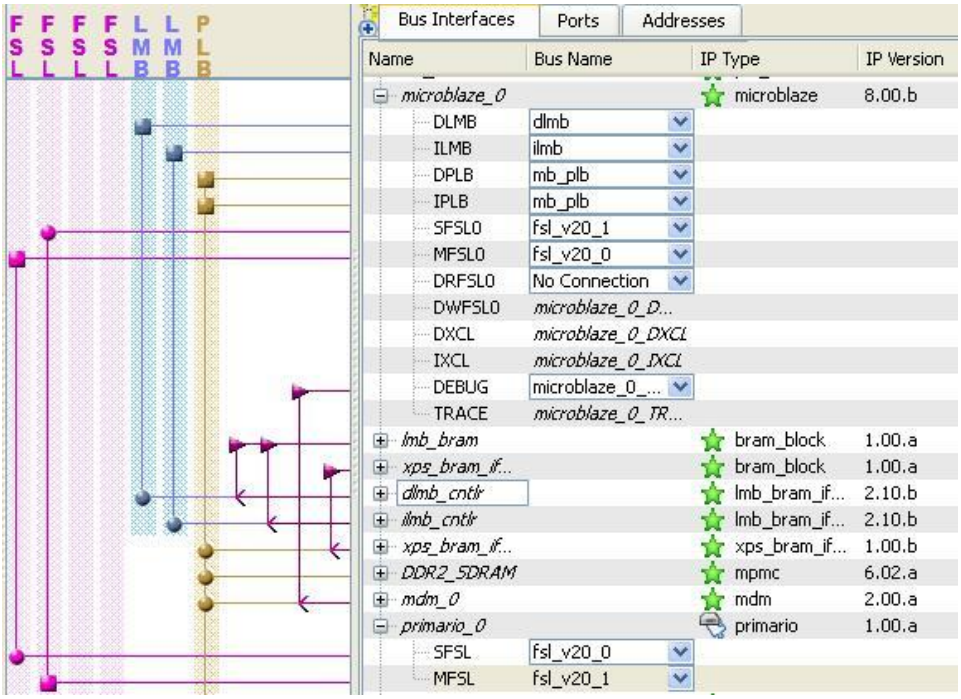


Figura 52. MicroBlaze y coprocesador conectados.

Tras haber realizado las conexiones ahora se debe configurar el funcionamiento del Microblaze y del coprocesador *primario*. El comportamiento del Microblaze se controla mediante un archivo en código C. Dicho código C puede estar almacenado en cualquier carpeta, pero nosotros vamos a utilizar el código C de ejemplo que se genera por defecto al crear el coprocesador *primario*, pero lo modificaremos. Lo primero es buscar el archivo. En el recuadro de la izquierda, donde aparecen los proyectos software, en el nuestro, *Project: Codificador*, con el botón derecho pinchamos en *Sources* y después en *Add Existing Files* (figura 53). Se abrirá una ventana de búsqueda y buscaremos archivo C que se encuentra en la siguiente ruta: *Nueva_carpeta/drivers/primario_v1_00_a/examples/primario_v2_1_0_app.c.*

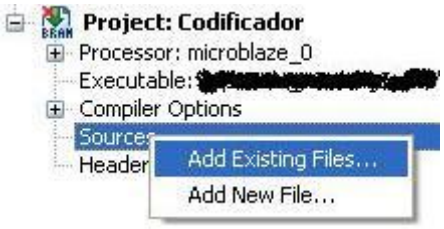


Figura 53. Añadir archivo en código C.

Ahora tendremos la oportunidad de abrir *Sources* y nos aparecerá el archivo C como se ve en la figura 54. Si pulsamos dos veces en el archivo este se nos abrirá en el lado derecho de la pantalla, viéndose como en la figura 55. Aquí desarrollaremos el código C que marcará el funcionamiento del Microblaze.


Figura 54. Código C añadido.

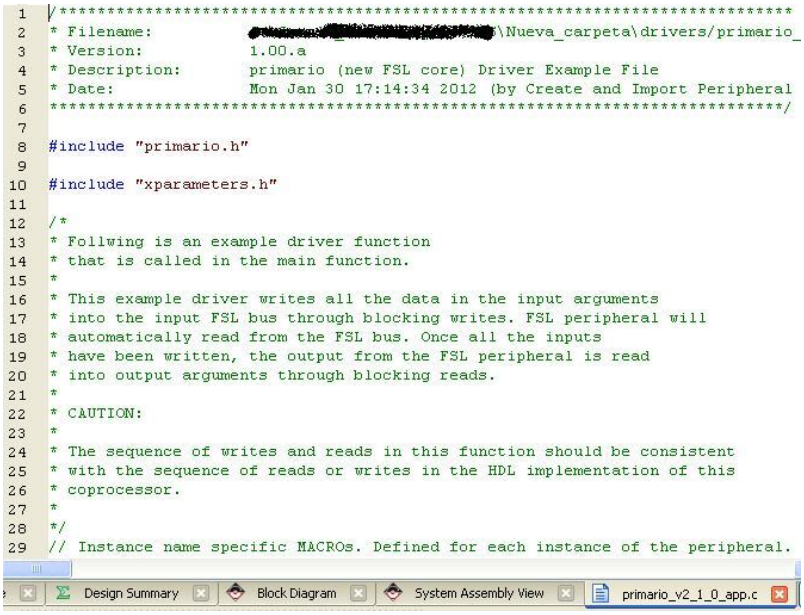


Figura 55. Apariencia inicial del código C.

El funcionamiento del coprocesador *primario* se controla mediante un código en lenguaje VHDL. Al crear el coprocesador se genera por defecto un código de prueba, que al igual que con el código C del Microblaze, nosotros lo vamos a usar y a modificar. Para abrir ese archivo, en *System Assembly View* pinchamos con el botón derecho en *primario_0* y seleccionamos *Browse HDL sources* (figura 56). Buscaremos el código VHDL en la siguiente ruta: *Nueva_carpeta/pcores/primario_v1_00_a/hdl/vhdl/primario.vhd*


Figura 56. Añadir archivo en código VHDL.

Una vez que encontramos el archivo se abrirá en el lado derecho como aparece en la figura 57.

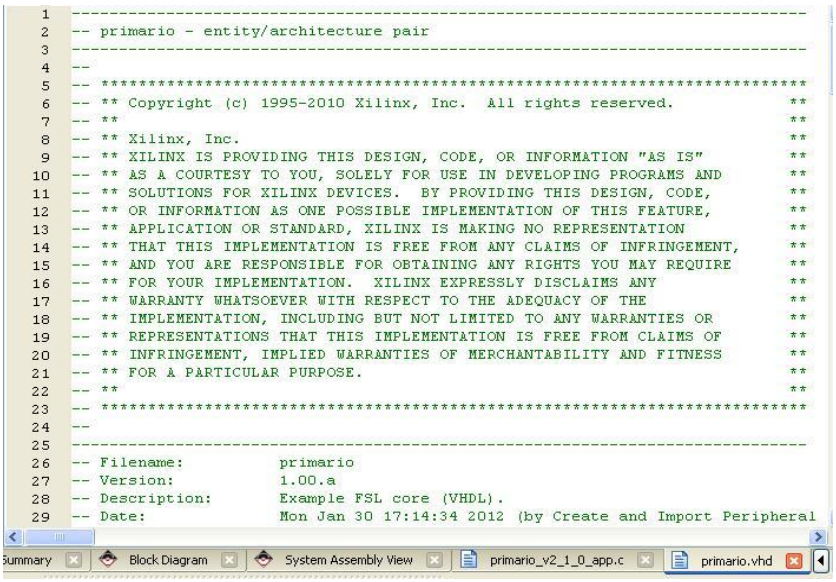


Figura 57. Apariencia inicial del código VHDL.

Ahora ya tenemos todo configurado y conectado y ya podemos abrir los códigos C y VHDL para modificarlos. Una vez que hayamos realizado todo esto hay que compilar el proyecto. Lo primero es, en el lado izquierdo, pinchar con el botón derecho sobre *Project: Codificador* y seleccionar *Mark to Initialize BRAMs*. Y ahora compilaremos todo el proyecto, la FPGA, los periféricos, el coprocesador y su código VHDL, etc, todo menos el código C de Microblaze, que eso de hace después. Seleccionaremos la pestaña *Hardware* de la parte de arriba y pulsaremos en *Generate Bitstream* (figura 58). Se inicia la compilación, que tarda alrededor de 10 minutos trabajando con un equipo *ADM Athlon II X2 250* con procesador de 3,01 Ghz y 2,75 GB de RAM (el tiempo suele ser el mismo para diferentes equipos). En la parte de abajo de la pantalla irán apareciendo mensajes de cómo va transcurriendo la compilación. Si hemos cometido algún error nos lo indicará y detendrá la compilación para que subsanemos el error. Si no seguirá hasta el final del proceso y cuando acabe aparecerá *Done*.


Figura 58. Generar *Bitstream*.

No obstante, el programa XPS tiene una pequeña pega. Si nos hemos equivocado y solucionamos el error o, después de haber compilado una vez, modificamos el código VHDL o cualquier otra parte del proyecto que no sea el código C, antes de volver a compilar con *Generate Bitstream* debemos eliminar los archivos creados previamente. Para ello, en la

pestaña *Project* pulsaremos *Clean all generated files* (figura 59). Cuando se hayan eliminado aparecerá *Done* y podremos volver a compilar, pero tendremos que volver a esperar 10 minutos.

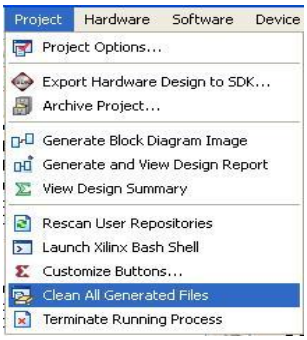


Figura 59. Eliminar los archivos generados.

Si la compilación es correcta pasaremos a compilar el código C. En *Project: Codificador* pulsaremos con el botón derecho y seleccionamos *Build Project* (figura 60). La compilación de código C es mucho más rápida, menos 30 segundos. Además, si sólo modificamos el código C no hace falta eliminar los archivos y generar el *Bitstream*, basta con volver a pulsar *Build Project*. Sin embargo, cada vez que se eliminan los datos previos y se genera el *Bitstream*, debemos volver a compilar también el Código C. Si todo es correcto, cuando termine la compilación aparecerá el mensaje *Done*.

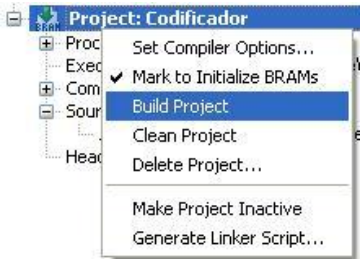


Figura 60. Compilar el código C.

Una vez hayamos terminado las dos compilaciones se debe crear el archivo que cargaremos en la FPGA. Para ello en la pestaña *Device Configuration* pulsaremos *Update Bistream* (figura 61). Cuando termine aparecerá *Done* y tendremos el archivo preparado para ser cargado. Finalmente, tenemos que cargar el programa en la FPGA. En nuestro caso, la FPGA que hemos utilizado nos permite cargarlo directamente desde el XPS pulsando *Download Bitstream* (figura 61). Hay otras placas de otros fabricantes que no permiten esta opción y han de ser cargadas mediante un programa secundario.


Figura 61. Cargar el programa en la FPGA.

A parte de todo lo explicado el programa XPS nos permite otras variantes como colocar varios FSL de entrada o salida en el coprocesador o más de un FSL de entrada en el Microblaze.

Estas acciones llevan el mismo procedimiento. Sin embargo, crear más de un FSL de salida del Microblaze lleva un proceso más complejo que no ha sido objeto de este proyecto fin de carrera.

Supongamos que queremos que el coprocesador tenga 2 FSL de salida que vayan a 2 FSL de entrada de Microblaze. El primer paso es cerrar completamente el programa XPS. Iremos a la carpeta de nuestro proyecto y abriremos el archivo *system.mhs* con un programa de tipo *Notepad++* o si no se dispone de ninguno con *Bloc de notas* también se puede. Este archivo debe ser modificado en varios puntos. Lo primero, buscamos las declaraciones del Microblaze (figura 62).

```
60  BEGIN microblaze
61   PARAMETER INSTANCE = microblaze_0
62   PARAMETER C_AREA_OPTIMIZED = 1
63   PARAMETER C_USE_BARREL = 1
64   PARAMETER C_DEBUG_ENABLED = 1
65   PARAMETER HW_VER = 8.00.b
66   PARAMETER C_FSL_LINKS = 1
67   BUS_INTERFACE DLMB = dlmb
68   BUS_INTERFACE ILMB = ilmb
69   BUS_INTERFACE DPLB = mb_plb
70   BUS_INTERFACE IPLB = mb_plb
71   BUS_INTERFACE DEBUG = microblaze_0_mdm_bus
72   BUS_INTERFACE SFSL0 = fsl_v20_1
73   BUS_INTERFACE MFSL0 = fsl_v20_0
74   PORT MB_RESET = mb_reset
75  END
```

Figura 62. Configuración inicial de puertos de MicroBlaze.

Vamos a tener 1 FSL de salida y 2 de entrada, por lo que en la línea 66 (los números de línea pueden variar de un proyecto a otro) debemos decir que el Microblaze va a utilizar 2 FSL en lugar de 1. Es decir, en el Microblaze una entrada y una salida forman un FSL completo, por lo que si añadimos una segunda entrada, es como añadir un FSL completo. Además, debemos declarar la nueva entrada y ponerle su nueva conexión, *fsl_v20_2*. Debe quedar como en la figura 63 (mirar las líneas 66 y 72). Sin embargo, no haría falta declarar la segunda salida porque no se va a utilizar (además, como ya se dijo, se configura de otra manera).

Hipotéticamente, si quisiéramos usar 4 entradas y 2 salidas, deberíamos colocar un 4 en la línea 66 y declarar las 3 nuevas entradas y la nueva salida.

```
60  BEGIN microblaze
61   PARAMETER INSTANCE = microblaze_0
62   PARAMETER C_AREA_OPTIMIZED = 1
63   PARAMETER C_USE_BARREL = 1
64   PARAMETER C_DEBUG_ENABLED = 1
65   PARAMETER HW_VER = 8.00.b
66   PARAMETER C_FSL_LINKS = 2
67   BUS_INTERFACE DLMB = dlmb
68   BUS_INTERFACE ILMB = ilmb
69   BUS_INTERFACE DPLB = mb_plb
70   BUS_INTERFACE IPLB = mb_plb
71   BUS_INTERFACE DEBUG = microblaze_0_mdm_bus
72   BUS_INTERFACE SFSL0 = fsl_v20_1
73   BUS_INTERFACE MFSL0 = fsl_v20_0
74   BUS_INTERFACE SFSL1 = fsl_v20_2
75   PORT MB_RESET = mb_reset
76  END
```

Figura 63. Configuración de puertos de MicroBlaze modificada.

En este mismo archivo, también debemos declarar la nueva salida del coprocesador *primario* así como declarar que existe la nueva conexión *fsl_v20_2* (observar las diferencias entre las figuras 64 y 65).

```
347  BEGIN primario
348   PARAMETER INSTANCE = primario_0
349   PARAMETER HW_VER = 1.00.a
350   BUS_INTERFACE SFSL = fsl_v20_0
351   BUS_INTERFACE MFSL = fsl_v20_1
352   PORT FSL_Clk = clk_62_5000MHz
353  END
354
355  BEGIN fsl_v20
356   PARAMETER INSTANCE = fsl_v20_0
357   PARAMETER HW_VER = 2.11.c
358  END
359
360  BEGIN fsl_v20
361   PARAMETER INSTANCE = fsl_v20_1
362   PARAMETER HW_VER = 2.11.c
363  END
```

Figura 64. Configuración inicial de puertos del coprocesador

```
347  BEGIN primario
348   PARAMETER INSTANCE = primario_0
349   PARAMETER HW_VER = 1.00.a
350   BUS_INTERFACE SFSL = fsl_v20_0
351   BUS_INTERFACE MFSL = fsl_v20_1
352   BUS_INTERFACE MFSL2 = fsl_v20_2
353   PORT FSL_Clk = clk_62_5000MHz
354  END
355
356  BEGIN fsl_v20
357   PARAMETER INSTANCE = fsl_v20_0
358   PARAMETER HW_VER = 2.11.c
359  END
360
361  BEGIN fsl_v20
362   PARAMETER INSTANCE = fsl_v20_1
363   PARAMETER HW_VER = 2.11.c
364  END
365
366  BEGIN fsl_v20
367   PARAMETER INSTANCE = fsl_v20_2
368   PARAMETER HW_VER = 2.11.c
369  END
```

Figura 65. Configuración modificada de puertos del coprocesador.

Observar que el nuevo FSL de salida de *primario*, MFSL2, se conecta al nuevo FSL de entrada del Microblaze, SFSL1, mediante la nueva conexión *fsl_v20_2*.

Está permitido que un FSL del Microblaze tenga el mismo nombre que un FSL de un coprocesador (por ejemplo, en ambos puede haber un SFSL2). Pero no puede haber varios FSL del mismo nombre en un coprocesador, ni tampoco repetirse si hay varios coprocesadores (por ejemplo, si hay 2 coprocesadores, no puede haber un SFSL1 en ambos).

Si tuviésemos más de un coprocesador, sus nuevos FSL y sus nuevas conexiones también se declararían en este archivo.

Tras haber declarado los nuevos FSL y las conexiones, se cierra el archivo *system.mhs* y se procede a configurar los puertos del coprocesador. Se debe abrir el archivo *primario_v2_1_0.mpd* que se encuentra en la siguiente ruta:
*Nueva_carpeta/pcores/primario_v1_00_a/data/primario_v2_1_0.mpd*

Este archivo se abre con *Notepad++* o con *Bloc de notas*. Originalmente nos aparecerá como en la figura 66.

Aquí hay que declarar el nuevo FSL de salida, con sus correspondientes puertos, así como declarar que el nuevo FSL se ve afectado por los puertos FSL_Clk, FSL_Reset. Observar el resultado final en la figura 67.

```
BEGIN primario

## Peripheral Options
OPTION IPTYPE = PERIPHERAL
OPTION IMP_NETLIST = TRUE
OPTION HDL = VHDL
## Bus Interfaces
BUS_INTERFACE BUS=SFSL, BUS_STD=FSL, BUS_TYPE=SLAVE
BUS_INTERFACE BUS=MFSL, BUS_STD=FSL, BUS_TYPE=MASTER

## Peripheral ports
PORT FSL_Clk = "", DIR=I, SIGIS=Clk, BUS=MFSL:SFSL
PORT FSL_Rst = OPB_Rst, DIR=I, BUS=MFSL:SFSL
PORT FSL_S_Clk = FSL_S_Clk, DIR=I, SIGIS=Clk, BUS=SFSL
PORT FSL_S_Read = FSL_S_Read, DIR=O, BUS=SFSL
PORT FSL_S_Data = FSL_S_Data, DIR=I, VEC=[0:31], BUS=SFSL
PORT FSL_S_Control = FSL_S_Control, DIR=I, BUS=SFSL
PORT FSL_S_Exists = FSL_S_Exists, DIR=I, BUS=SFSL
PORT FSL_M_Clk = FSL_M_Clk, DIR=I, SIGIS=Clk, BUS=MFSL
PORT FSL_M_Write = FSL_M_Write, DIR=O, BUS=MFSL
PORT FSL_M_Data = FSL_M_Data, DIR=O, VEC=[0:31], BUS=MFSL
PORT FSL_M_Control = FSL_M_Control, DIR=O, BUS=MFSL
PORT FSL_M_Full = FSL_M_Full, DIR=I, BUS=MFSL

END
```

Figura 66. Configuración inicial de archivo *.mpd*.

```
BEGIN primario

## Peripheral Options
OPTION IPTYPE = PERIPHERAL
OPTION IMP_NETLIST = TRUE
OPTION HDL = VHDL
## Bus Interfaces
BUS_INTERFACE BUS=SFSL, BUS_STD=FSL, BUS_TYPE=SLAVE
BUS_INTERFACE BUS=MFSL, BUS_STD=FSL, BUS_TYPE=MASTER
BUS_INTERFACE BUS=MFSL2, BUS_STD=FSL, BUS_TYPE=MASTER

## Peripheral ports
PORT FSL_Clk = "", DIR=I, SIGIS=Clk, BUS=MFSL:SFSL:MFSL2
PORT FSL_Rst = OPB_Rst, DIR=I, BUS=MFSL:SFSL:MFSL2
PORT FSL_S_Clk = FSL_S_Clk, DIR=I, SIGIS=Clk, BUS=SFSL
PORT FSL_S_Read = FSL_S_Read, DIR=O, BUS=SFSL
PORT FSL_S_Data = FSL_S_Data, DIR=I, VEC=[0:31], BUS=SFSL
PORT FSL_S_Control = FSL_S_Control, DIR=I, BUS=SFSL
PORT FSL_S_Exists = FSL_S_Exists, DIR=I, BUS=SFSL
PORT FSL_M_Clk = FSL_M_Clk, DIR=I, SIGIS=Clk, BUS=MFSL
PORT FSL_M_Write = FSL_M_Write, DIR=O, BUS=MFSL
PORT FSL_M_Data = FSL_M_Data, DIR=O, VEC=[0:31], BUS=MFSL
PORT FSL_M_Control = FSL_M_Control, DIR=O, BUS=MFSL
PORT FSL_M_Full = FSL_M_Full, DIR=I, BUS=MFSL

PORT FSL_M_Clk2 = FSL_M_Clk, DIR=I, SIGIS=Clk, BUS=MFSL2
PORT FSL_M_Write2 = FSL_M_Write, DIR=O, BUS=MFSL2
PORT FSL_M_Data2 = FSL_M_Data, DIR=O, VEC=[0:31], BUS=MFSL2
PORT FSL_M_Control2 = FSL_M_Control, DIR=O, BUS=MFSL2
PORT FSL_M_Full2 = FSL_M_Full, DIR=I, BUS=MFSL2

END
```

Figura 67. Configuración modificada de archivo *.mpd*.

Si en lugar de añadir un FSL de salida añadiésemos uno de entrada habría que duplicar los puertos FSL_S. Y con el mismo procedimiento se pueden añadir más de un FSL de entrada o salida, colocando los nombres nuevos.

Si hay varios coprocesadores está permitido que se repitan los nombres de los puertos, pero no se permite repetir el nombre del FSL, como ya se explicó anteriormente.

Es decir, si tenemos 2 coprocesadores, con un FSL de salida cada uno, pueden repetirse los nombres de los puertos de salida (FSL_M_Clk, FSL_M_Data, etc.), pero hay que indicar que pertenecen a su FSL correspondiente (supongamos que para el primero es MFSL y para el segundo MFSL2). Tampoco se permite que 2 conjuntos de puertos de entrada o salida de un mismo coprocesador pertenezcan al mismo FSL.

Si hay más de un coprocesador, cada uno tendrá su propio archivo .mpd en la ruta: *Nueva_carpeta/pcores/coprocesador_v1_00_a/data/coprocesador_v2_1_0.mpd*

En el lugar de coprocesador va el nombre de cada uno de ellos.

Una vez modificados los archivos .mhs y .mpd guardaremos los cambios y abriremos nuestro proyecto pulsando 2 veces sobre el archivo *system.xmp* de nuestra carpeta principal del proyecto, que tiene el logotipo del programa XPS.

Si hemos configurado bien los archivos el proyecto se abrirá correctamente. En caso contrario, aparecerá un mensaje de error, indicando cuál es el fallo. En ese caso, cerraremos el programa, iremos a los archivos, arreglaremos los fallos, guardaremos y volveremos a abrir el programa.

Una vez abierto *System Assembly View* deben aparecer los nuevos FSL y conexiones (figura 68).

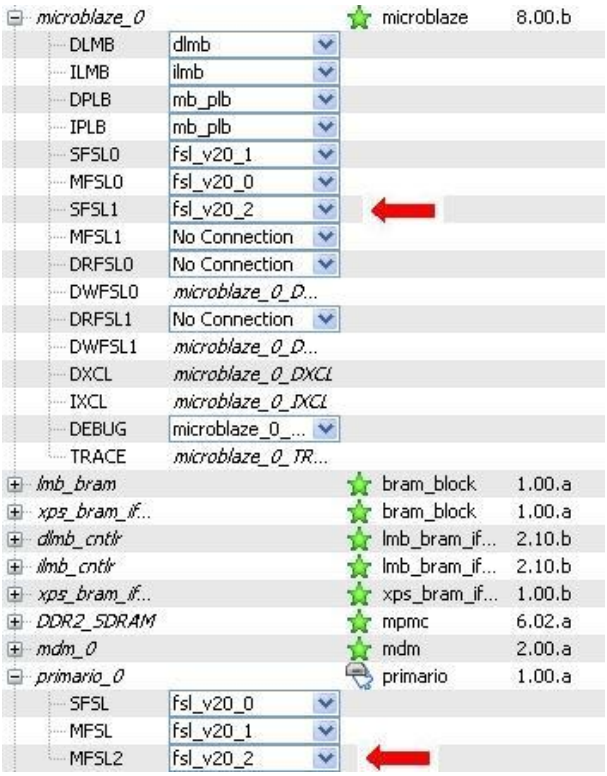


Figura 68. Apariencia de conexiones con más de un FSL de entrada o salida.

Ahora debemos modificar el código C y el código VHDL. Los abriremos de la forma anteriormente indicada.

El código VHDL original se ve como en la figura 69. En la parte de declaración de puertos de entrada y salida.

```
entity primario is
   port
   (
      -- DO NOT EDIT BELOW THIS LINE ----------------------
      -- Bus protocol ports, do not add or delete.
      FSL_Clk  : in  std_logic;
      FSL_Rst  : in  std_logic;
      FSL_S_Clk   : in  std_logic;
      FSL_S_Read  : out std_logic;
      FSL_S_Data  : in  std_logic_vector(0 to 31);
      FSL_S_Control  : in  std_logic;
      FSL_S_Exists   : in  std_logic;
      FSL_M_Clk   : in  std_logic;
      FSL_M_Write : out std_logic;
      FSL_M_Data  : out std_logic_vector(0 to 31);
      FSL_M_Control  : out std_logic;
      FSL_M_Full  : in  std_logic
      -- DO NOT EDIT ABOVE THIS LINE ----------------------
   );

attribute SIGIS : string;
attribute SIGIS of FSL_Clk : signal is "Clk";
attribute SIGIS of FSL_S_Clk : signal is "Clk";
attribute SIGIS of FSL_M_Clk : signal is "Clk";

end primario;
```

Figura 69. Configuración inicial de puertos en el código VHDL

En concordancia con lo modificado en el archivo *primario_v2_1_0.mpd* (figura 67) debemos declarar los nuevos puertos, quedando como se refleja en la figura 70.

```
entity primario is
   port
   (
      -- DO NOT EDIT BELOW THIS LINE ----------------------
      -- Bus protocol ports, do not add or delete.
      FSL_Clk  : in  std_logic;
      FSL_Rst  : in  std_logic;
      FSL_S_Clk   : in  std_logic;
      FSL_S_Read  : out std_logic;
      FSL_S_Data  : in  std_logic_vector(0 to 31);
      FSL_S_Control  : in  std_logic;
      FSL_S_Exists   : in  std_logic;
      FSL_M_Clk   : in  std_logic;
      FSL_M_Write : out std_logic;
      FSL_M_Data  : out std_logic_vector(0 to 31);
      FSL_M_Control  : out std_logic;
      FSL_M_Full  : in  std_logic;

      FSL_M_Clk2  : in  std_logic;
      FSL_M_Write2   : out std_logic;
      FSL_M_Data2 : out std_logic_vector(0 to 31);
      FSL_M_Control2 : out std_logic;
      FSL_M_Full2 : in  std_logic
      -- DO NOT EDIT ABOVE THIS LINE ----------------------
   );

attribute SIGIS : string;
attribute SIGIS of FSL_Clk : signal is "Clk";
attribute SIGIS of FSL_S_Clk : signal is "Clk";
attribute SIGIS of FSL_M_Clk : signal is "Clk";
attribute SIGIS of FSL_M_Clk2 : signal is "Clk";

end primario;
```

Figura 70. Configuración modificada de puertos en el código VHDL.

Ahora, tras añadir los puertos y modificar lo necesario del código VHDL, guardaremos, eliminaremos los datos anteriores y generaremos el nuevo *Bitstream* (figuras 59 y 60).

Mientras se compila el código VHDL modificaremos el código C. Lo abrimos y en su forma original aparecen las siguientes líneas como en la figura 71.

```
// Instance name specific MACROs. Defined for each instance of the peripheral.
#define WRITE_PRIMARIO_0(val)  write_into_fsl(val, XPAR_FSL_PRIMARIO_0_INPUT_SLOT_ID)
#define READ_PRIMARIO_0(val)  read_from_fsl(val, XPAR_FSL_PRIMARIO_0_OUTPUT_SLOT_ID)
```

Figura 71. Instrucciones de lectura y escritura iniciales del código C.

La instrucción *WRITE_PRIMARIO_0(val)* envía por el MFSL0 el dato que ocupe el lugar de "*val"* dentro del paréntesis. De igual manera, la instrucción *READ_PRIMARIO_0(val)* copia en el registro que ocupa la posición "*val*" el dato que se recibe por el SFSL0.

Como ahora tenemos un segundo FSL de entrada, debemos declararlo con una nueva instrucción, quedando como en la figura 72.

```
// Instance name specific MACROs. Defined for each instance of the peripheral.
#define WRITE_PRIMARIO_0(val)  write_into_fsl(val, XPAR_FSL_PRIMARIO_0_INPUT_SLOT_ID)
#define READ_PRIMARIO_0(val)  read_from_fsl(val, XPAR_FSL_PRIMARIO_0_OUTPUT_SLOT_ID)
#define READ_PRIMARIO_1(val)  read_from_fsl(val, XPAR_FSL_PRIMARIO_1_OUTPUT_SLOT_ID)
```

Figura 72. Instrucciones de lectura y escritura modificadas del código C.

La instrucción *READ_PRIMARIO_1(val)* copia en el registro que ocupa la posición "*val*" el dato que se recibe por el SFSL1, como se puede ver en la figura 72. Esto se hace tantas veces como nuevos FSL de entrada se crean.

Una vez que se ha generado el *Bitstream* pulsaremos *Build Project* para que se compile el código C. En esta ocasión nos dará error. Para subsanarlo abriremos el archivo *xparameters.h* con *Notepad++* o Bloc de Notas. Se encuentra en la siguiente ruta: *Nueva_carpeta/microblaze_0/include/xparameters.h*.

La carpeta *include* se crea al pulsar *Build Project*, por lo que no es posible evitar que nos dé el error.

Abriremos el archivo y buscaremos las líneas que aparecen en la figura 73.

```
187  /*****************************************************
188
189  #define XPAR_FSL_PRIMARIO_0_OUTPUT_SLOT_ID  0
190  #define XPAR_FSL_PRIMARIO_0_INPUT_SLOT_ID  0
191
192  /*****************************************************
```

Figura 73. Archivo *xparameters.h* original.

Ambas instrucciones se refieren a un FSL de entrada y otro de salida. Ya se explico anteriormente que para MicroBlaze un FSL completo consta de entrada y salida. Por lo que si

queremos otro FSL de entrada, aquí también se ha de declarar un nuevo FSL de salida aunque no se vaya a utilizar. Nos quedará como en la figura 74.

```
187  /*******************************************************
188
189  #define XPAR_FSL_PRIMARIO_0_OUTPUT_SLOT_ID  0
190  #define XPAR_FSL_PRIMARIO_0_INPUT_SLOT_ID  0
191  #define XPAR_FSL_PRIMARIO_1_OUTPUT_SLOT_ID  1
192  #define XPAR_FSL_PRIMARIO_1_INPUT_SLOT_ID  1
193
194  /*******************************************************
```

Figura 74. Archivo *xparamaters.h* modificado.

Por cada nuevo FSL completo hay que declarar un FSL de entrada y otro de salida. Como se ve, va en concordancia con lo reflejado en la figura 72.

Si modificamos el código VHDL y pulsamos *Clean all generated files* la carpeta *microblaze_0* será eliminada. Por ello, cuando volvamos a pulsar *Build Project* volveremos a tener el error y se tendrá que modificar de nuevo el archivo *xparameters.h* tal como se ha explicado.

Una vez que hemos compilado todo sin errores, ya se puede generar el archivo de carga y cargarlo en la FPGA (figuras 58 y 61).