Sistemas Informáticos Curso 2006-2007

Aceleración de procesamiento intensivo mediante plataformas de hardware reconfigurable

Autores:

Emilio Martínez Pacheco
Federico Yeray Navarro Polo
Antonio González Sánchez-Migallón

Dirigido por:

David Atienza Alonso

Facultad de Informática
Universidad Complutense de Madrid

Índice

1. Preámbulo

Autorización

Los autores de este proyecto, Emilio Martínez Pacheco, Federico Yeray Navarro Polo y Antonio González Sánchez-Migallón, autorizan mediante este texto a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos no comerciales, y mencionando expresamente a sus autores tanto el código, la documentación y/o el prototipo desarrollado.

Emilio Martínez Pacheco **Federico Yeray Navarro Polo** **Antonio González Sánchez-Migallón**

Agradecimientos

Nos gustaría enviar un agradecimiento especial a Pablo García del Valle, por su apoyo y ayuda a lo largo del curso en el transcurso del proyecto, y a Miguel Peón Quirós y Ángel Luis González Bravo, por la ayuda prestada en los momentos más difíciles al final del proyecto, así como a nuestro tutor, David Atienza Alonso. Sin ellos, no nos cabe duda de que no hubiéramos sido capaces de realizar este proyecto tal y como lo hemos hecho.

Resumen del proyecto

Aceleración de procesamiento intensivo mediante plataformas de hardware reconfigurable

El objetivo del proyecto es conseguir aprovechar la gran potencia y versatilidad que ofrece el hardware reconfigurable (FPGA) para liberar al procesador del PC de cálculos intensivos que demande una determinada aplicación. Para ello se ha creado una interfaz entre el host PC y la FPGA a través del bus PCI que permita la utilización de ésta última como una tarjeta aceleradora de propósito general.

El sistema, haciendo uso de la principal característica del hardware reconfigurable como es la flexibilidad, intentará conseguir una determinada aceleración, siempre de forma individual, de distintos perfiles de ejecución, como por ejemplo una red neuronal o un algoritmo específico. Idealmente está pensado para tareas que necesiten mínima comunicación con el exterior, esto es, algoritmos de cálculos pesados con transacciones de entrada/salida muy sencillas. Como ejemplo de un perfil determinado que cumpla estos requerimientos se ha elegido el algoritmo criptográfico AES (Advanced Encryption Standard), un esquema de cifrado por bloques adoptado como un estándar por el gobierno de los Estados Unidos.

Para la realización del proyecto se ha utilizado una FPGA “Spartan IIE”. En ésta se ha introducido un puente (Bridge PCI-Wishbone) entre el bus PCI y el sistema para facilitar la gestión del protocolo del primero, siendo más sencillo el manejo de sus señales. El sistema completo estará formado por distintos módulos que ejecuten cada uno de los perfiles de ejecución anteriormente citados.

El elemento central del proyecto entorno al que ha girado la mayor implementación del mismo ha sido la comunicación del host con el dispositivo a través del conector PCI existente en este último.

Abstract

Acceleration of intensive processing through platforms of reconfigurable hardware

The main target of the project is to get benefit of the great strength and versatility of reconfigurable hardware (FPGA) for freeing the PC multiprocessor from the task of making complex and intensive calculus. To achieve this, it has been created an interface through PCI port between host PC and FPGA, allowing the use of the FPGA like a general purpose acceleration card.

The system takes advantage of the most determinant feature of reconfigurable hardware, flexibility, for achieving acceleration in different execution profiles, like could be a neural network or a specific algorithm. The system is focused in tasks which require minimal communication between the nodes, in order to avoid non relative to calculus tasks, like data load. Examples of this could algorithms with heavy operations and simple I/O. A profile which could satisfy this requisite is the cryptographic algorithm AES, a block cipher scheme adopted like a standard by the government of United States.

For the development of the project it has been used a FPGA Spartan IIE. It has been programmed with a bridge PCI-Wishbone to allow a simplified communication between the implemented system and the PCI port. Besides the PCI Bridge, the system counts with different hardware modules which carry out the distinct execution profiles already mentioned.

The most relevant aspect of the project, and in which the most part of the work has been focused, is the communication between the host and the device through the PCI port.

Objetivos del proyecto

- Familiarizarse con la placa y el software a utilizar en el proyecto.
- Lograr comunicación entre PC y FPGA a través del bus PCI.
- Encontrar uno o varios algoritmos que sirvan para ilustrar la aceleración que se desea conseguir.
- Realizar diseños del algoritmo de ejemplo tanto en hardware como en software.
- Elaborar un controlador para Microsoft Windows, de tal forma que sea posible realizar la comunicación PC-FPGA en software propio.
- Realizar una comparativa entre la versión software y la versión hardware y analizar los resultados.

2. Entorno de trabajo

Introducción a las FPGA

Una FPGA[1] es un dispositivo hardware semiconductor que contiene componentes lógicos e interconexiones programables entre ellos, típicamente volátiles.

Las dos características fundamentales de las FPGA son:

- Funcionalidad configurable: pequeñas memorias de configuración que almacenan tablas de verdad cuyo contenido es seleccionado por un multiplexor.
- Interconexión configurable: segmentos metálicos unidos a través de transistores de paso controlados por una memoria de configuración. Una jerarquía de interconexiones programables permite a los bloques lógicos de una FPGA ser interconectados según la necesidad del diseñador del sistema.

Una FPGA es programable a nivel hardware, por lo que proporciona las ventajas de un procesador de propósito general y un circuito especializado. A cambio, estos dispositivos ofrecen en general un rendimiento inferior a los diseños realizados directamente sobre un circuito integrado. Además de ser más lentos por norma general, consumen más energía y no suele aceptar diseños excesivamente complejos debido a carencia de recursos para su elaboración.

La versatilidad propia de las FPGA convierte a estos dispositivos en una plataforma idónea para el desarrollo hardware, pues los cambios en el diseño se pueden probar de forma relativamente rápida, y una vez testeado, quemar el diseño en dispositivos ASIC[2]. Esta reducción de tiempos en las fases de testeo y diseño conduce también a una reducción de costes en el proceso global del desarrollo hardware.

La raíz histórica de las FPGA son los dispositivos de lógica programable compleja CPLD[3] de mediados de los ochenta.

La principal diferencia entre los CPLD y las FPGA son sus arquitecturas. Un CPLD tiene una estructura un poco más restringida, consistiendo en la unión de uno o más arrays lógicos que alimentan a un número pequeño de registros con entrada de reloj. Ello conlleva una flexibilidad reducida, con la ventaja de una mejor predicción de los tiempos de retraso. La arquitectura de las FPGA, por otro lado, cuenta con muchas más interconexiones. Esto los hace más flexibles, es decir, el rango de diseños prácticos en los cuales pueden ser usados es mayor.

Otra notable diferencia entre CPLD y FPGA es la presencia de funciones de más alto nivel (tales como sumadores y multiplicadores) dentro de las FPGA, además de memorias.

Una tendencia reciente ha sido combinar los bloques lógicos e interconexiones de las FPGA con microprocesadores y periféricos relacionados para formar un "Sistema programable en un chip". Esto puede hacerse bien a través de procesadores integrados

[1] Field Programmable Gate Array
[2] Application Specific Integrated Circuit
[3] Complex Programmable Logic Device

directamente en el hardware, o bien haciendo uso de procesadores integrados en la lógica de la FPGA, como MicroBlaze o PicoBlaze. Esta última opción tiene como desventaja la ocupación de parte de la FPGA, además de no ser tan rápidos o completos como los procesadores hardware en circuito integrado.

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".

Una FPGA se compone de una serie de elementos básicos, entre los que cabe destacar los siguientes:

- Celdas básicas de funcionalidad configurable (CLB)
- Bloques de interconexión configurable
- Celdas de entrada/salida (IOB)
- Memorias de configuración
- Circuitería de control de la configuración.

La tarea del programador consiste en definir la función lógica que realizará cada una de las celdas CLB, seleccionar el modo de trabajo de las celdas IOB y el interconexionado. En la práctica, se utilizan entornos de diseño que reúnen una amplia variedad de herramientas de desarrollo para realizar estas funciones, dejando las tareas a más bajo nivel, como el interconexionado o la definición de funciones de cada bloque lógico, a cargo de estas aplicaciones, que implementan sofisticados algoritmos que optimizan el uso de recursos en la FPGA. De igual forma, estos entornos incorporan simuladores que permiten garantizar el correcto funcionamiento de los módulos diseñados por el programador sin necesidad de volcarlos en la FPGA.

Así, con la ayuda de entornos de desarrollo especializados en el diseño de sistemas para FPGA, se puede diseñar hardware desde distintos niveles de abstracción, desde diagramas de estados y diseños de esquemáticos hasta lenguajes de programación de alto nivel. Estos últimos son conocidos como lenguajes HDL[4]. Los lenguajes más usados dentro de esta categoría son VHDL y Verilog; la utilización del primero es más extensa en Europa, mientras que el segundo está más extendido en Estados Unidos.

Cualquier circuito de aplicación específica puede ser implementado en una FPGA, siempre y cuando esta disponga de los recursos necesarios. Las aplicaciones donde más comúnmente se utilizan son aquellas en las que se puede explotar un alto grado de paralelismo, pudiéndose realizar módulos específicos que puedan tratar los datos con una rapidez que supere a la de un microprocesador de propósito general. Entre estas aplicaciones, se pueden destacar las siguientes:

- DSP (Digital signal process)
- Prototipos de ASIC
- Sistemas de tratamiento de imágenes o de visión por computador
- Bioinformática

[4] Hardware Description Language

- Simulación y depuración en el diseño de microcontroladores
- Simulación y depuración en el diseño de microprocesadores

A fecha de 2007, el mercado de las FPGA se encuentra principalmente dominado por dos fabricantes mayoritarios, Xilinx y Altera. En nuestro caso se ha utilizado una placa de prototipado del primero, con dos FPGA integradas (Virtex-2 y Spartan-2).

Placa de prototipado utilizada en el proyecto

La placa utilizada en el proyecto ha sido la Xilinx Virtex-II Pro XC2VP30, modificada por AVNET para su Virtex-II Pro Development Kit.

Esta placa incluye entre sus características más destacadas las siguientes:

- FPGA Xilinx Virtex-II Pro XC2VP30
- FPGA Spartan-IIE XC2S300E
- Memoria Micron DDR SDRAM SODIMM (128 MB expandibles a 1 GB)
- Memoria Micron Mobile SDRAM (32 MB)
- Memoria SRAM (2 MB)
- Tarjeta CompactFlash
- Dos microprocesadores PowerPC
- Puerto Ethernet 10/100/1000 Mb/s
- Dos puertos serie RS-232
- Puerto PCI universal (compatible con ranuras de 32 y 64 bit, y voltajes de 3.3V y 5.0V)
- Puente PCI para la Spartan-IIE

Aunque en principio la FPGA Spartan-IIE está destinada a albergar un módulo que hace de puente PCI para interconectar el resto de componentes, en nuestro caso nos hemos visto obligados a utilizar esta FPGA para realizar el diseño global del proyecto.

La FPGA Spartan-IIE es una solución de bajo coste que ofrece un aceptable rendimiento. El modelo concreto XC2S300E tiene una capacidad de hasta trescientas mil puertas lógicas, casi siete mil celdas, 64 Kb de memoria BRAM y más de trescientos bloques de entrada/salida. Cabe destacar también la compatibilidad con una amplia gama de estándares de entrada/salida, incluyendo el estándar PCI.

PCI

El bus PCI[5] es un bus de comunicaciones estándar de ancho de banda elevado e independiente del procesador. Es ampliamente utilizado para conectar dispositivos periféricos directamente a la placa base de un sistema, bien en forma de tarjetas de expansión, bien en forma de chips integrados en la propia placa. Aunque la especificación es abierta, se encuentra gestionada y comercializada por el grupo PCI SIG[6].

Este bus es muy común en los PC de sobremesa, donde sustituyó como bus de expansión a los antiguos buses ISA[7] y VLB[8] durante los años noventa. En la actualidad, se encuentra en proceso de desaparición, en beneficio del bus PCI-Express, que permite unas tasas de datos muy superiores.

A diferencia de los buses ISA, el bus PCI permite configuración dinámica de un dispositivo periférico. Esta es una diferencia importante respecto al bus ISA, que no permitía dicha funcionalidad, haciendo necesario el uso de jumpers externos para llevar a cabo la asignación de IRQ[9] y otra información relevante, que en caso de conflicto podían llegar a provocar la inestabilidad del sistema. En un sistema típico, el sistema operativo analiza el bus PCI en tiempo de arranque del sistema para averiguar qué dispositivos están presentes y qué recursos del sistema (memoria, líneas de interrupción,...) necesita cada uno. Entonces asigna dichos recursos y le pasa dicha información a cada dispositivo. Cada dispositivo PCI puede solicitar hasta seis espacios de memoria y de entrada/salida.

El espacio de configuración PCI también contiene una pequeña cantidad de información acerca del dispositivo, lo que se recoge en la cabecera del mismo. Estos datos son relativos al tipo de dispositivo, fabricante, etc., información que ayuda al sistema operativo en la elección de controladores o, en su defecto, a informar al usuario acerca de la presencia del dispositivo.

El bus ha pasado por varias versiones y revisiones que han ido ampliando su funcionalidad y características iniciales. Entre sus características básicas más importantes están las siguientes:

- Reloj de 33.33 MHz con transferencias síncronas
- Ancho de bus de 32 bits ó 64 bits
- Tasa de transferencia máxima de 133 MB/s en el bus de 32 bits
- Tasa de transferencia máxima de 266 MB/s en el bus de 64 bits.
- Espacio de dirección de 32 bits (4 GB)
- Espacio de puertos I/O de 32 bits
- 256 bytes de espacio de configuración.
- Configuración de 3.3 V ó 5 V, dependiendo del dispositivo

5 Peripheral Component Interconnect
6 PCI Special Interest Group
7 Industry Standard Architecture
8 VESA Local Bus
9 Interrupt ReQuest

- Posibilidad de compartir una (o varias) IRQ entre distintos dispositivos PCI, lo cual, dada la escasez de líneas de interrupción, resuelve un viejo problema de la arquitectura PC.

Versiones posteriores del estándar mejoran estas características. Por ejemplo, la versión 2.2 funciona a 66 MHz, permitiendo una tasa de transferencia máxima de 533 MB/s. Asimismo se rediseñó en parte el estándar, creándose una nueva versión conocida como PCI-X, que incrementaba las tasas de transferencia e incluía algunos cambios leves en el protocolo, aunque por regla general los dispositivos seguían siendo compatibles hacia atrás.

El bus PCI contiene un bus de alimentación, con las líneas +5, +3.3 +12 y -12 V.; un bus de direcciones (multiplexado); un bus de datos y un bus de control que incluye cuatro líneas de interrupciones, una de presencia de tarjeta, y líneas de control y test. No soporta DMA en el sentido tradicional del IBM PC, aunque dispone de análoga funcionalidad mediante *bus mastering*.

Ha sido diseñado pensando en sistemas de máximas prestaciones, e incluye todas las funcionalidades y características de los diseños más modernos (soporte para multiprocesador, transferencia a ráfagas -*burst mode*-, etc.) Presenta características que no eran usuales en los sistemas de bus anteriores, por ejemplo:

- **Configuración por software** (sin jumpers): PCI se creó pensando en el estándar PnP[10], por lo que los dispositivos PCI pueden ser configurados exclusivamente mediante software. Cada dispositivo PCI debe estar diseñado para solicitar de forma inequívoca los recursos que necesita (zona de memoria mapeada, direcciones E/S, canales DMA, interrupciones, etc.). Para ello tienen una memoria ROM que contiene las especificaciones de configuración y de aquí obtiene el sistema los datos necesarios en la etapa de arranque.
- **Identificación**: los dispositivos PCI deben identificarse a sí mismos, señalando su fabricante, modelo, número de serie y código de clase. Los códigos de fabricante son administrados por el PCI SIG. El código de clase proporciona un método de identificación, de modo que el controlador genérico del sistema operativo disponga de cierta información básica sobre el dispositivo PCI conectado, e incluso, en ausencia de un controlador específico, proporcionar algún control básico del dispositivo.
- **Diseño flexible**: en cualquier momento pueden añadirse nuevos códigos de fabricante o de clase. De hecho, la especificación ya ha pasado por varias mejoras y extensiones. Por ejemplo, el bus AGP[11] es una extensión de la especificación PCI; otras revisiones de la especificación incluyen el conector SmallPCI, el soporte para 64bits y las versiones de 3.3 V.
- **Independencia**: PCI no está ligado a ninguna plataforma particular; puede ser implementado virtualmente en cualquiera, además de la conocida arquitectura IBM-PC/x86. De hecho, ha sido adoptado por muchos fabricantes de otras arquitecturas, como por ejemplo Apple y SUN.

[10] Plug and Play
[11] Advanced Graphics Port

Espacio de configuración PCI

Una de las principales ventajas que tenía el bus PCI respecto a otros buses de entrada/salida era su mecanismo de configuración. Además de los habituales espacios de puertos y memoria mapeada, cada dispositivo en el bus dispone de un espacio de configuración. Se trata de 256 bytes que son direccionables conociendo 8 bit del bus PCI, 5 bit de dispositivo y 3 bit de número de función para el dispositivo. Esto permitiría, en teoría, hasta 256 buses, cada uno con 32 dispositivos con 8 funciones. Una simple tarjeta de expansión PCI puede responder como un dispositivo y debe implementar al menos la función número cero. Los primeros 64 bytes del espacio de configuración están estandarizados; el resto están disponibles para uso propio del fabricante.

Entre la información disponible en esta cabecera de 64 bytes, se puede destacar la siguiente:

- Vendor ID, define al fabricante del chip y es asignado por la organización PCI SIG.
- Device ID, identifica al dispositivo y es asignado por el fabricante.
- Subsystem Vendor ID y Subsystem Device ID acotan aún más la información acerca del dispositivo, indicando el fabricante de la tarjeta.
- El Command Register contiene una máscara de bits que especifica diversas características que puede ser activadas o desactivadas individualmente.
- El Status Register se utiliza para informar acerca de la compatibilidad con determinadas funciones o para informar del estado de una operación, como por ejemplo, cuando sucede un error.

Para poder configurar en un dispositivo los espacios de memoria o puertos, el firmware del sistema o el sistema operativo programan los BAR escribiendo comandos de configuración al controlador PCI. Dado que durante el arranque del sistema todos los dispositivos PCI se encuentran en un estado inactivo, no tienen asignada ninguna dirección con la que el sistema operativo o los drivers se puedan comunicar.

Cuando durante dicha configuración se lee la cabecera de un dispositivo con éxito, se asignan direcciones al dispositivo PCI. Estas direcciones permanecerán válidas durante el tiempo que el sistema permanezca en ejecución. Al apagarse el sistema, esta configuración se pierde y volverá a asignarse en el próximo arranque. Dado que este procedimiento se realiza de forma automática, se ahorra así al usuario la difícil tarea de configurar cualquier nuevo dispositivo hardware de forma manual, teniendo que modificar la configuración o cambiando interruptores en las tarjetas de expansión. Este método recibe el popular nombre de *Plug and Play*.

Estructura y funcionamiento del PCI

Como ya se ha comentado anteriormente, el bus PCI puede configurarse como un bus de 32 o 64 bits. La siguiente tabla define las líneas más importantes obligatorias en el PCI:

- *CLK* (reloj): señal de reloj que es muestreada en el flanco de subida.
- *RST#* (reset): hace que todos los registros y señales específicas del PCI pasen al estado inicial.

Señales de direcciones y datos

- *AD[31:0]: i*ncluye 32 líneas para datos y direcciones multiplexadas en el tiempo.
- *C/BE[3:0]:* Se utilizan para interpretar y validar las líneas de datos y direcciones.

Señales de control de interfaz

- *FRAME:* activada por el master para indicar el comienzo y la duración de una transferencia. Las activa al comienzo y la desactiva al final de la fase de datos.
- *IRDY:* Señal de *master* preparado (*Initiator Ready*). La proporciona el *master* actual del bus (el iniciador de la transacción). Durante una lectura, indica que el master está preparado para aceptar datos; durante una escritura indica que el dato válido está en AD.
- *TRDY*: señal de *slave* preparado (*Target Ready*). La activa el *slave* al principio de la transferencia, y la desactiva cuando no puede completar la transferencia en un solo ciclo de reloj
- *DEVSEL: s*eñal de *slave* (dispositivo) seleccionado (*Device Select*). Activada por el *slave* cuando ha reconocido su dirección.

Señales de arbitraje

- *REO: i*ndica al árbitro que el dispositivo correspondiente solicita utilizar el bus. Es una línea punto-a-punto específica para cada dispositivo.
- *GNT:* indica al dispositivo que el árbitro le ha cedido el acceso al bus. Es una línea punto-a-punto específica para cada dispositivo.

La actividad del bus consiste en transferencias entre dos elementos, denominándose maestro al que inicia la transacción. Cuando un maestro del bus adquiere el control del mismo, determina el tipo de transferencia que se producirá a continuación. Los tipos, entre otros son los siguientes:

- Reconocimiento de interrupción
- Lectura de E/S
- Escritura en E/S
- Lectura de memoria
- Escritura en memoria

Toda transferencia de datos en el bus PCI es una transacción única, que consta de una fase de direccionamiento y una o más fases de datos. La siguiente figura muestra la temporización de una operación de lectura. Todos los eventos se sincronizan en las transiciones de bajada del reloj, cosa que sucede a la mitad de cada ciclo de reloj.

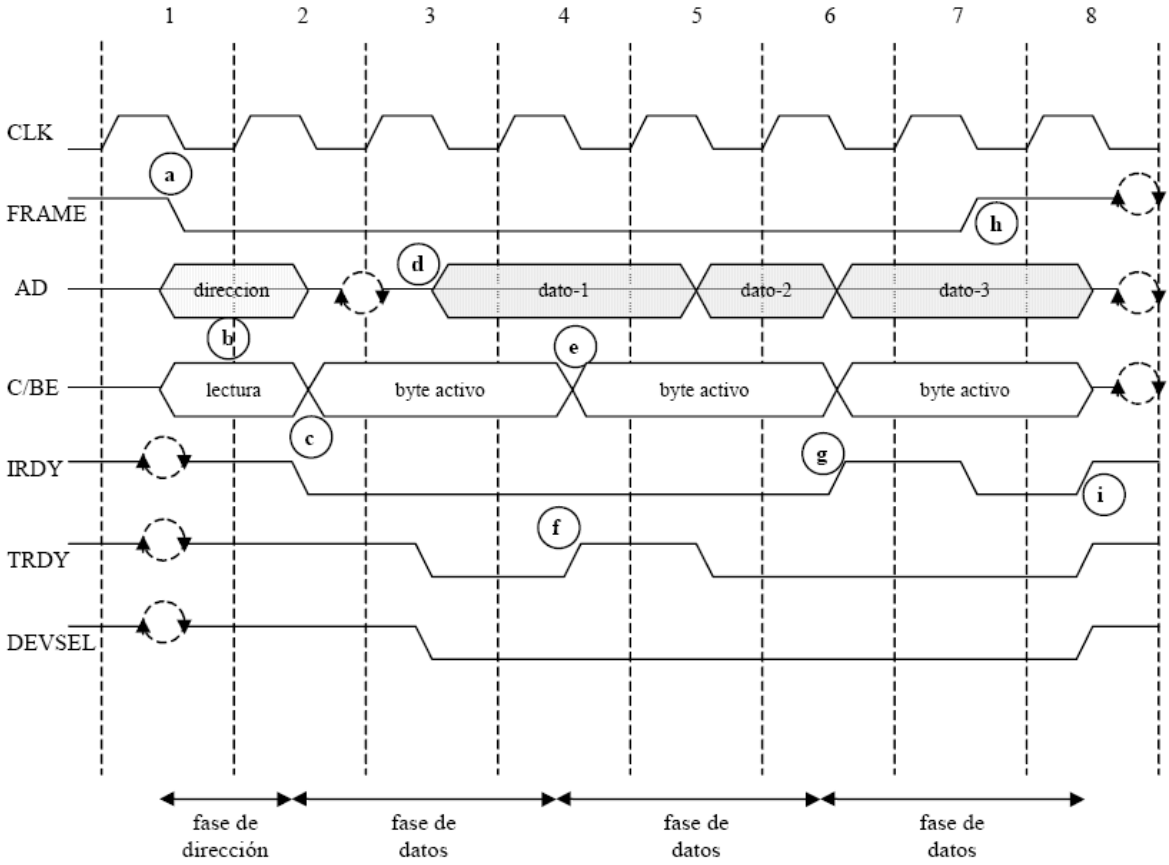

Figura 1. Cronograma de una transferencia en el bus PCI.

1. Los dispositivos interpretan las señales del bus en los flancos de subida, al comienzo del ciclo. A continuación, se describen los eventos significativos señalados en el diagrama:
2. Una vez que el *master* ha obtenido el control del bus, inicia la transacción:
   - activando *FRAME*, que permanece activa hasta la última fase de datos
   - situando la dirección de inicio en el bus de direcciones
   - situando la orden de lectura en las líneas *C/BE*.
3. El *slave* reconoce su dirección en las líneas *AD* al comienzo del ciclo de reloj 2.
4. El *master* deja libre las líneas *AD* del bus y cambia la información de las líneas *C/BE* para indicar qué líneas *AD* se utilizan para transportar datos (de 1 a 4 bytes). También activa *IRDY* para indicar que está preparado para recibir el primer dato (*).
5. El *slave* activa *DEVSEL* para indicar que ha reconocido su dirección. Después sitúa el dato solicitado en las líneas *AD* y activa *TRDY* para indicar que hay un dato válido en el bus.
6. El *master* lee el dato al comienzo del ciclo de reloj 4 y cambia las líneas *C/BE* según se necesite para la próxima lectura.

7. En este ejemplo el *slave* necesita algún tiempo para preparar el segundo bloque de datos para la transmisión. Por tanto desactiva *TRDY* para señalar al *master* que no proporcionará un nuevo dato en el próximo ciclo. En consecuencia el *master* no lee las líneas de datos al comienzo del quinto ciclo de reloj y no cambia la señal *C/BE* durante ese ciclo. El bloque de datos es leído al comienzo del ciclo de reloj 6.
8. Durante el ciclo 6 el *slave* sitúa el tercer dato en el bus. No obstante, en este ejemplo, el *master* todavía no está preparado para leer el dato. Para indicarlo, desactiva *IRDY*. Esto hará que el *slave* mantenga el tercer dato en el bus durante un ciclo de reloj extra.
9. El master sabe que el tercer dato es el último, y por eso desactiva *FRAME* para indicárselo al *slave*. Además activa *IRDY* para indicar que está listo para completar esa transferencia.
10. El *master* desactiva *IRDY*, haciendo que el bus vuelva a estar libre, y el *slave* desactiva *TRDY* y *DEVSEL*.

(*) Nota: En todas las líneas que pueden ser activadas por más de un dispositivo se necesita un ciclo de cambio (indicado por las dos flechas circulares) para que pueda ser utilizado por el dispositivo de lectura.

El bus PCI utiliza un esquema de arbitraje centralizado síncrono, en el que cada maestro tiene una señal propia de petición (REQ) y cesión (GNT) del bus. Estas líneas se conectan a un árbitro central. La especificación PCI no indica un algoritmo particular de arbitraje. El árbitro puede utilizar un procedimiento de *primero en llegar primero en servirse*, un procedimiento de *cesión cíclica* (*round-robin*), o cualquier clase de esquema de prioridad. El maestro del PCI establece, para cada transferencia que desee hacer, si tras la fase de dirección sigue una o más fases de datos consecutivas.

La siguiente figura es un ejemplo en el que se arbitra a cuál de los dispositivos A y B se cede el bus. Se produce la siguiente secuencia:

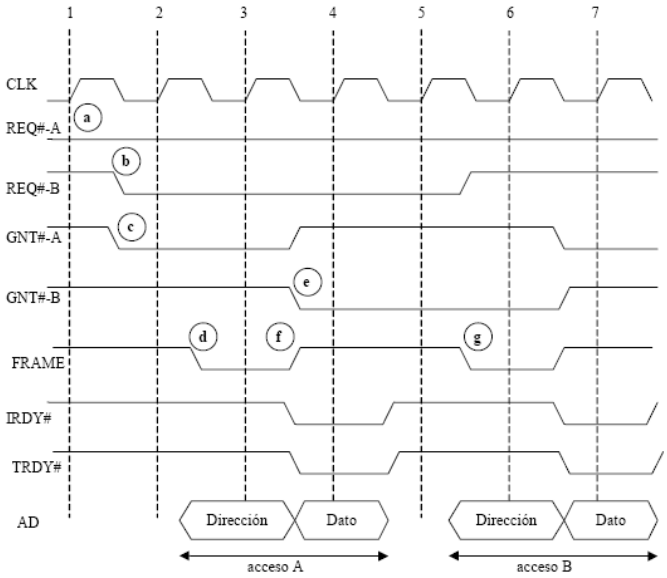

Figura 2. Cronograma de un proceso de arbitraje en el bus PCI.

1. En algún momento anterior al comienzo del ciclo de reloj 1, *A* ha activado su señal *REQ*. El árbitro muestrea esa señal al comienzo del ciclo de reloj 1.
2. Durante el ciclo de reloj 1, *B* solicita el uso del bus activando su señal *REQ*.
3. Al mismo tiempo, el árbitro activa *GNT-A* para ceder el acceso al bus a *A*.
4. El maestro del bus *A* muestrea *GNT-A* al comienzo del cielo de reloj 2 y conoce que se le ha cedido el acceso al bus. Además, encuentra *IRDY* y *TRDY* desactivados, indicando que el bus está libre. En consecuencia, activa *FRAME* y coloca la información de dirección en el bus de direcciones, y la orden correspondiente en las líneas *C/BE*. Además mantiene activa *REQ-A*, puesto que tiene que realizar otra transferencia después de la actual.
5. El árbitro del bus muestrea todas las líneas *GNT* al comienzo del ciclo 3, y toma la decisión de ceder el bus a *B* para la siguiente transacción. Entonces activa *GNT-B* y desactiva *GNT-A*. *B* no podrá utilizar el bus hasta que éste no vuelva a estar libre.
6. *A* desactiva *FRAME* para indicar que la última transferencia de datos está en marcha.
7. Pone los datos en el bus de datos y se lo indica al dispositivo destino con *IRDY*. El dispositivo lee el dato al comienzo del siguiente ciclo de reloj.
8. Al comienzo del ciclo 5, *B* encuentra *IRDY* y *FRAME* desactivados y, por consiguiente, puede tomar el control del bus activando *FRAME*. Además, desactiva su línea *REQ*, puesto que sólo deseaba realizar una transferencia.

Hay que resaltar que el arbitraje se produce al mismo tiempo que el maestro actual del bus está realizando su transferencia de datos. Por consiguiente, no se pierden ciclos de bus en realizar el arbitraje. Esto se conoce como *arbitraje oculto o solapado* (*hidden arbitration*).

VHDL

El principal lenguaje de descripción hardware que se ha usado para implementar el sistema ha sido VHDL (acrónimo que representa la combinación de VHSIC[12] y HDL).

VHDL es un lenguaje que se utiliza para describir circuitos a un alto nivel de abstracción. Está ampliamente aceptado, sobre todo en Europa, como un medio estándar de diseño. VHDL es el producto del programa VHSIC desarrollado por el Departamento de Defensa de los Estados Unidos a finales de la década de los 70. El propósito era crear un estándar para diseñar, modelar y documentar circuitos complejos de tal manera que un diseño desarrollado por una empresa pudiera ser entendido por otra y, además, pudiera ser procesado por software con propósitos de simulación.

VHDL es reconocido como un estándar de los lenguajes HDL por el instituto de Ingenieros en Electricidad y Electrónica – IEEE – como su estándar 1076 el cual fue ratificado en 1987, y por parte del Departamento de Defensa de los Estados Unidos como el estándar MIL-STD-454L. En 1993 el estándar IEEE-1076 se actualizó y un estándar adicional, el IEEE-1164 fue adoptado. Para 1996 el estándar IEEE-1076.3 se convirtió en un estándar de VHDL para síntesis, siendo éste el que se utiliza fundamentalmente en el diseño de sistemas digitales en Europa. Los estándares más utilizados en síntesis de circuitos por la mayoría de las herramientas de diseño son el IEEE-1164 y el IEEE-1076.3. En la actualidad, VHDL es un estándar de la industria para la descripción, modelado y síntesis de circuitos digitales en general y particularmente para programar PLD, FPGA, ASIC y similares.

Además para implementar el puente entre el bus PCI y nuestro sistema (Bridge PCI-Wishbone de OpenCores[13]) se ha utilizado el lenguaje de descripción hardware más utilizado en los Estados Unidos: Verilog. Este lenguaje soporta diseño, prueba e implementación de circuitos analógicos, digitales y de señal mixta a diferentes niveles de abstracción.

Verilog fue diseñado en 1985 por Phil Moorby y en 1989 fue adquirido por Cadence Design Systems. En 1990 Cadence Design Systems decide liberar el lenguaje y es cedido a OVI (Open Verilog International) que es formada en 1991. En 1993 bajo el subcomité de diseño automatizado de la IEEE se establece un grupo de trabajo para producir el estándar IEEE Verilog 1364. En 1995 es establecido el estándar IEEE 1364. El estándar es revisado en el 2001 y actualizado a IEEE 1364-2001. Debido a la evolución de los lenguajes de descripción de hardware nace Accelera[14] en el 2000 con la unión de OVI y VHDL International, y es ésta la que propone las últimas extensiones de Verilog. Tiene una sintaxis similar a la del lenguaje de programación C, de tal manera que resulte familiar a los ingenieros.

12 Very High Speed Integrated Circuit
13 http://www.opencores.org
14 http://www.accelera.org

3. Herramientas software utilizadas

Xilinx ISE

El Xilinx Integrated Software Environment es más un conjunto de herramientas que una aplicación por sí sola, constituyendo un verdadero entorno EDA[1]. ISE es el nexo de unión del paquete de aplicaciones que proporciona Xilinx para desarrollo, simulación y síntesis de hardware. El resto de herramientas se pueden utilizar desde ISE de forma directa o bien ejecutarse de forma separada.

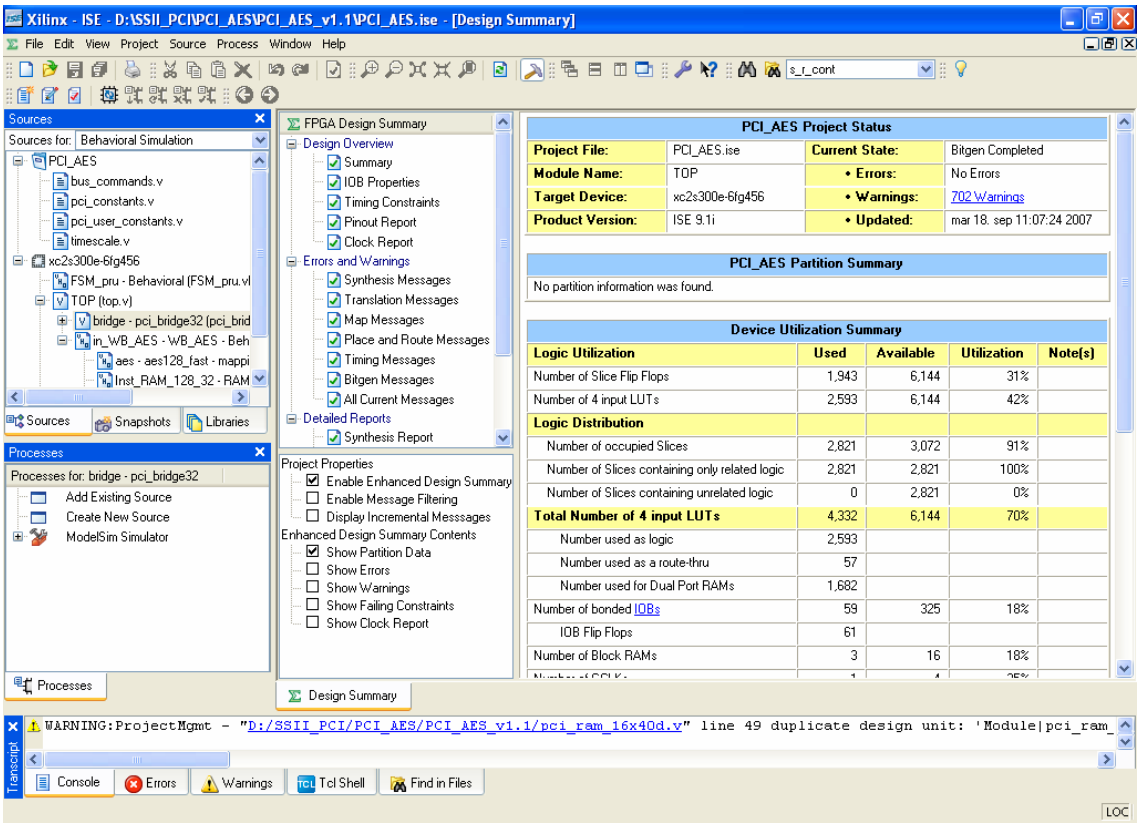

Figura 3. Captura de la aplicación Xilinx ISE.

Xilinx ISE se ha utilizado como medio para reunir los diversos módulos de código en VHDL y Verilog y poder sintetizar el proyecto completo. Asimismo, ha sido una herramienta de uso prácticamente diario dadas sus características, aunque finalmente fueran otras las herramientas encargadas de realizar otras tareas más concretas, como por ejemplo la simulación (ModelSim) o la configuración de la PROM (iMPACT), entre otras.

[1] Electronic Design Automation

Xilinx EDK

El Xilinx Embedded Development Kit es un conjunto de herramientas y módulos IP que capacita al usuario en la tarea de diseñar un sistema de proceso empotrado completo para su posterior implementacion en un dispositivo FPGA.

Esta herramienta facilita en gran parte el diseño de la arquitectura interna de un sistema, pues tan solo es necesario añadir los diversos módulos IP al mismo y enlazarlo con los respectivos buses del sistema. Xilinx EDK viene por defecto con una amplia gama de módulos IP ya disponibles para su uso, desde módulos para el control de comunicación externa como puertos serie RS232 a procesadores específicos para su uso en FPGA como Microblaze.

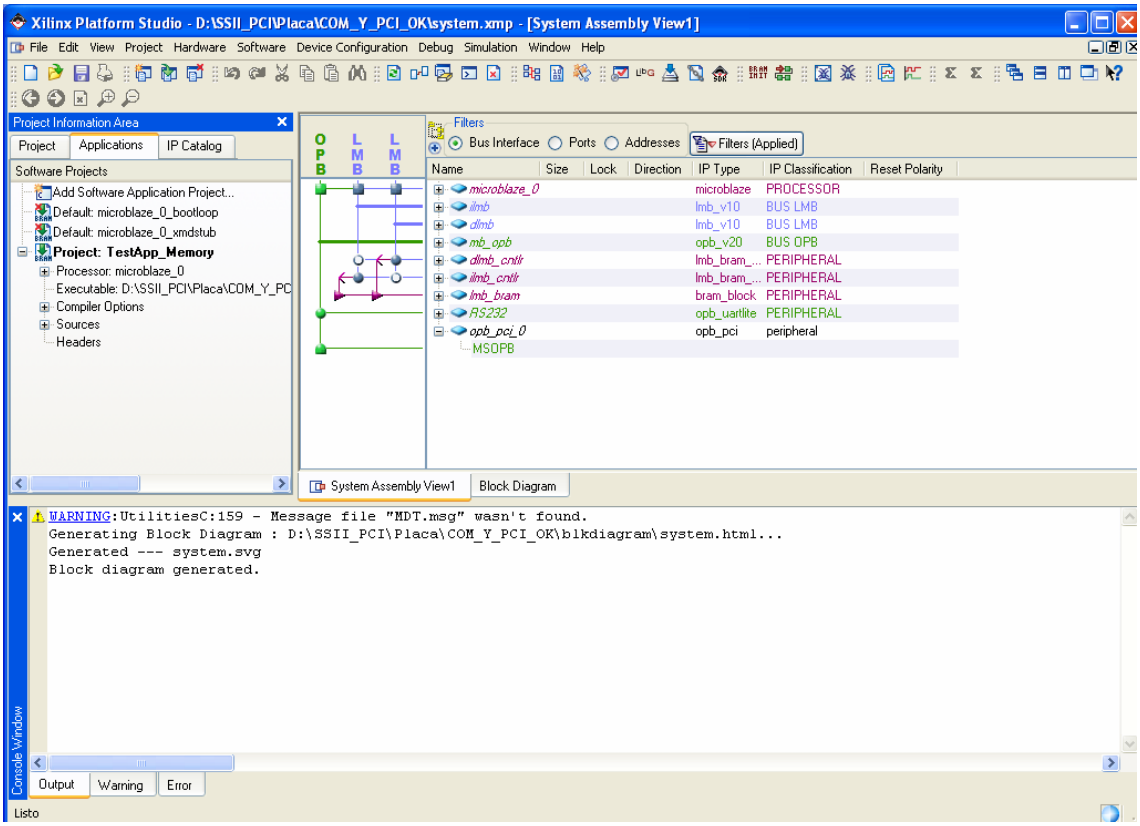

Figura 4. Captura de la aplicación Xilinx EDK.

Xilinx EDK se utilizó en las fases iniciales del proyecto debido a su facilidad de uso y a que satisfacía las primeras necesidades. No obstante, según fue aumentando de envergadura el proyecto, fue necesario empezar a utilizar el entorno Xilinx ISE.

iMPACT

iMPACT es una herramienta incluida en el paquete integrado de desarrollo hardware de Xilinx. Se utiliza para programar los chips con el código ya sintetizado.

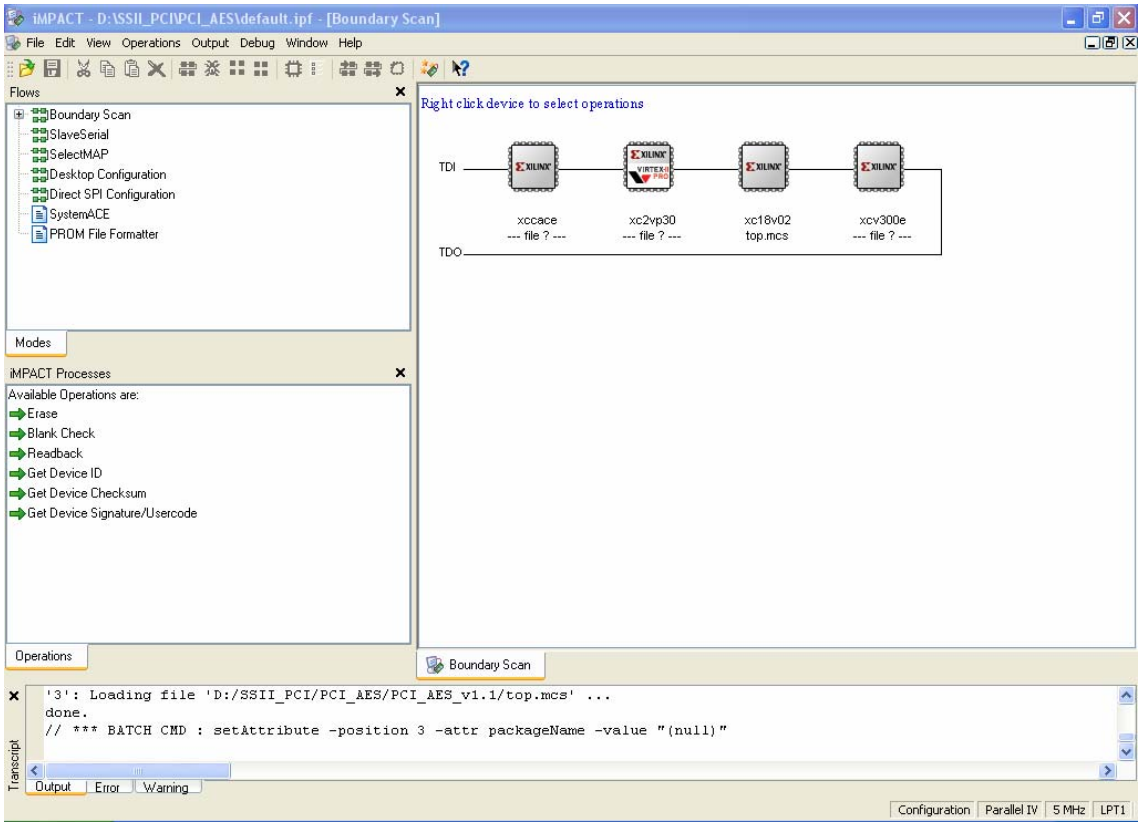

Figura 5. Captura de la aplicación Impact.

Esta aplicación se ha utilizado fundamentalmente para dos cometidos. El primero de ellos, preparar las imágenes de ROM con el contenido del proyecto; y el segundo de ellos, 'quemar' dichas imágenes en la PROM[2] de la placa, con el fin de poder configurar la FPGA Spartan II-E.

[2] Programmable Read Only Memory

PciTree

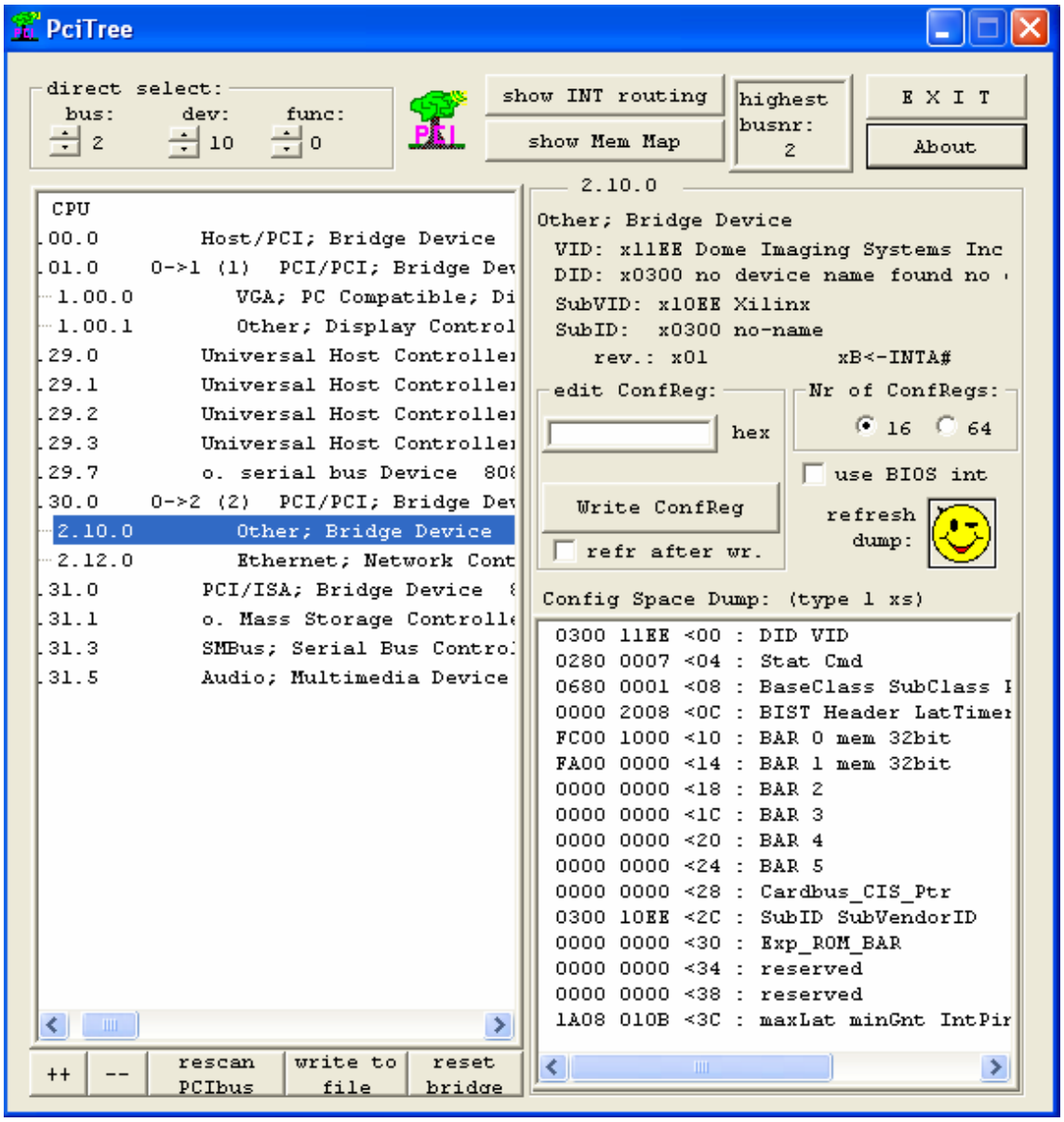

Figura 6. Captura de la aplicación PciTree.

PciTree es una herramienta gráfica shareware para el sistema operativo Windows que muestra, en forma de árbol, un listado de todos los dispositivos hardware conectados al bus PCI. Permite escritura y lectura de los registros de configuración, así como de los espacios de memoria definidos en cada dispositivo. Se ha empleado con fines de depuración principalmente, dado que no requiere de instalación de ningún controlador adicional para poder realizar estas operaciones.

ModelSim

ModelSim es un entorno de simulación para VHDL y Verilog, que incluye diversas herramientas para facilitar la depuración de especificaciones de hardware en dichos lenguajes. Es capaz de simular tanto especificaciones RTL como especificaciones a nivel de puertas lógicas. Otra importante característica es que permite la simulación de sistemas multicódigo, es decir, con módulos VHDL y Verilog mezclados.

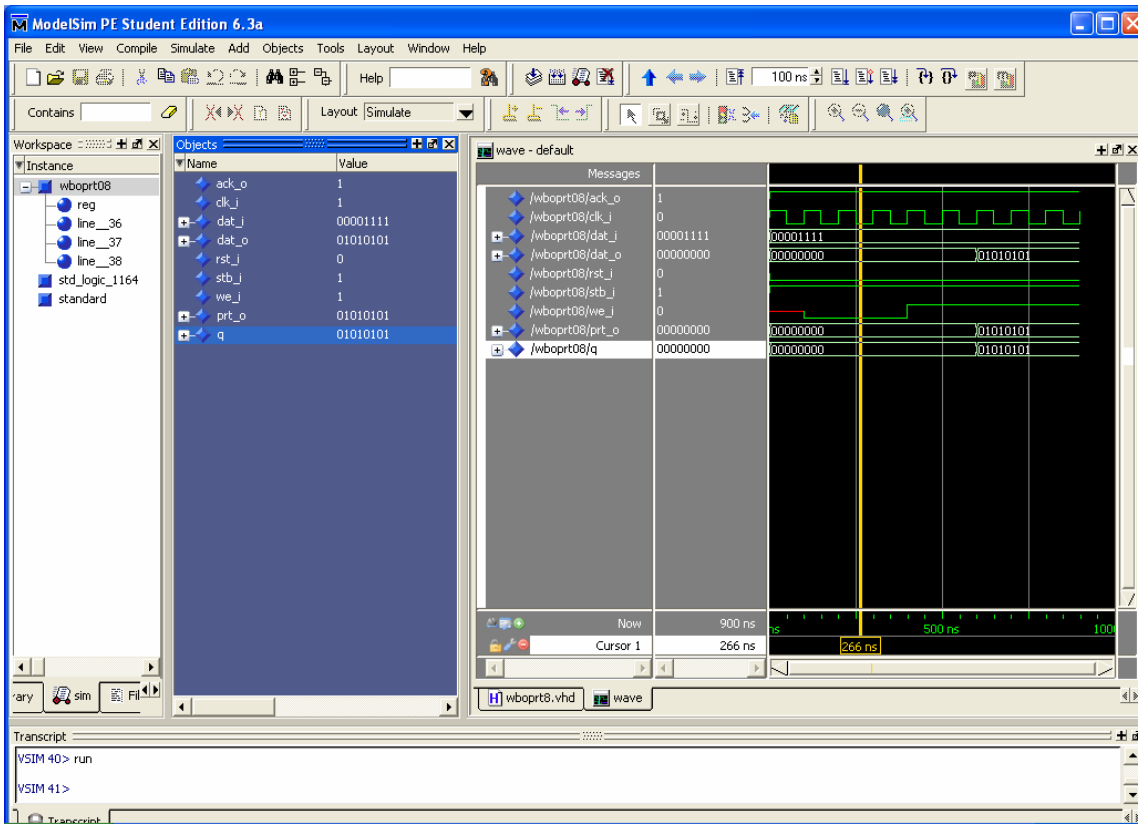

Figura 7. Captura de la aplicación ModelSim.

Se ha utilizado concretamente la versión para estudiantes de ModelSim 6.3 Personal Edition, suficiente para nuestros propósitos, consistentes en probar los diseños de los distintos módulos hardware antes de volcarlos a la placa, con el fin de garantizar su correcto funcionamiento lógico.

Microsoft Visual Studio 2003

Microsoft Visual Studio es un entorno integrado de desarrollo para sistemas Windows compatible con diversos lenguajes de programación, como C++, C#, ASP y Visual Basic.

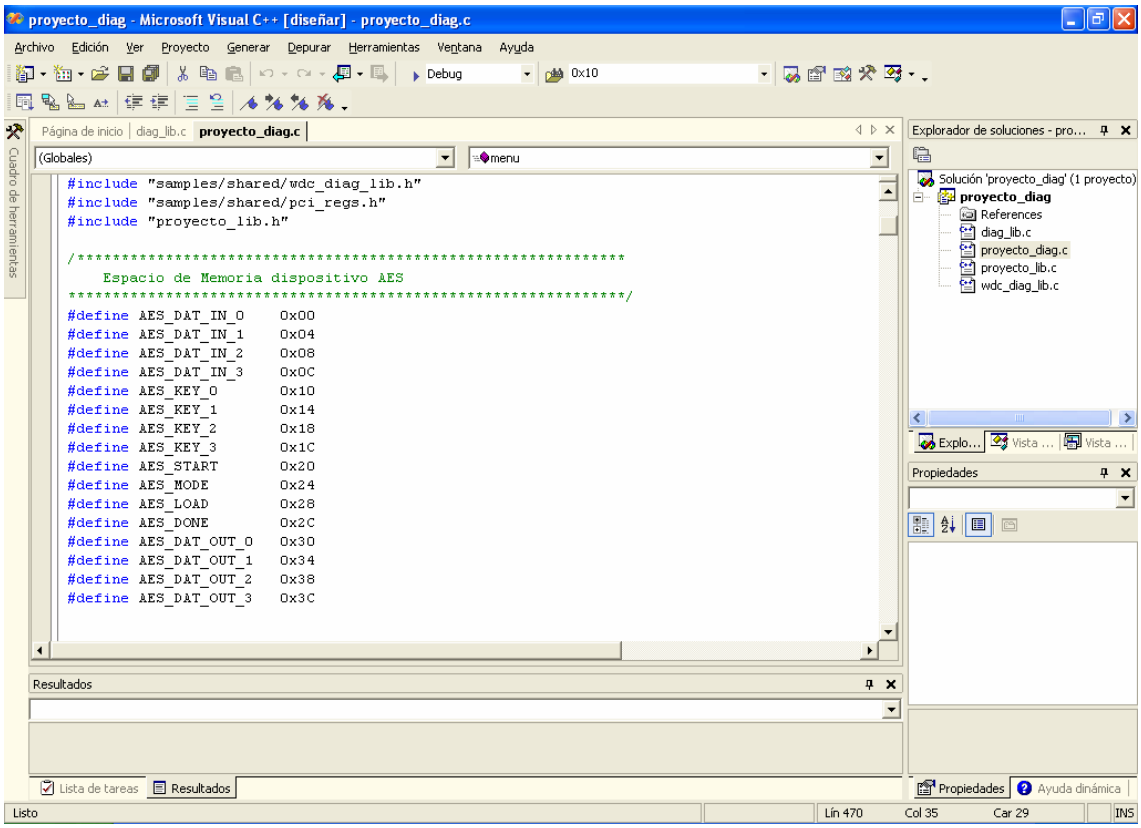

Figura 8. Captura de la aplicación Visual Studio.

En nuestro caso se ha utilizado como plataforma para desarrollar la aplicación software, pues contábamos con que la aplicación utilizada para el desarrollo del controlador (WinDriver) podía generar ejemplos de utilización del mismo para varios de los IDE más conocidos del mercado, entre ellos, la herramienta de Microsoft.

La aplicación desarrollada se ha realizado en lenguaje C, también compatible con el IDE de Microsoft.

Jungo WinDriver[3]

WinDriver es una herramienta que facilita la generación de controladores de dispositivos para diferentes sistemas operativos, como Windows, Linux, Solaris y VxWorks, entre otros.

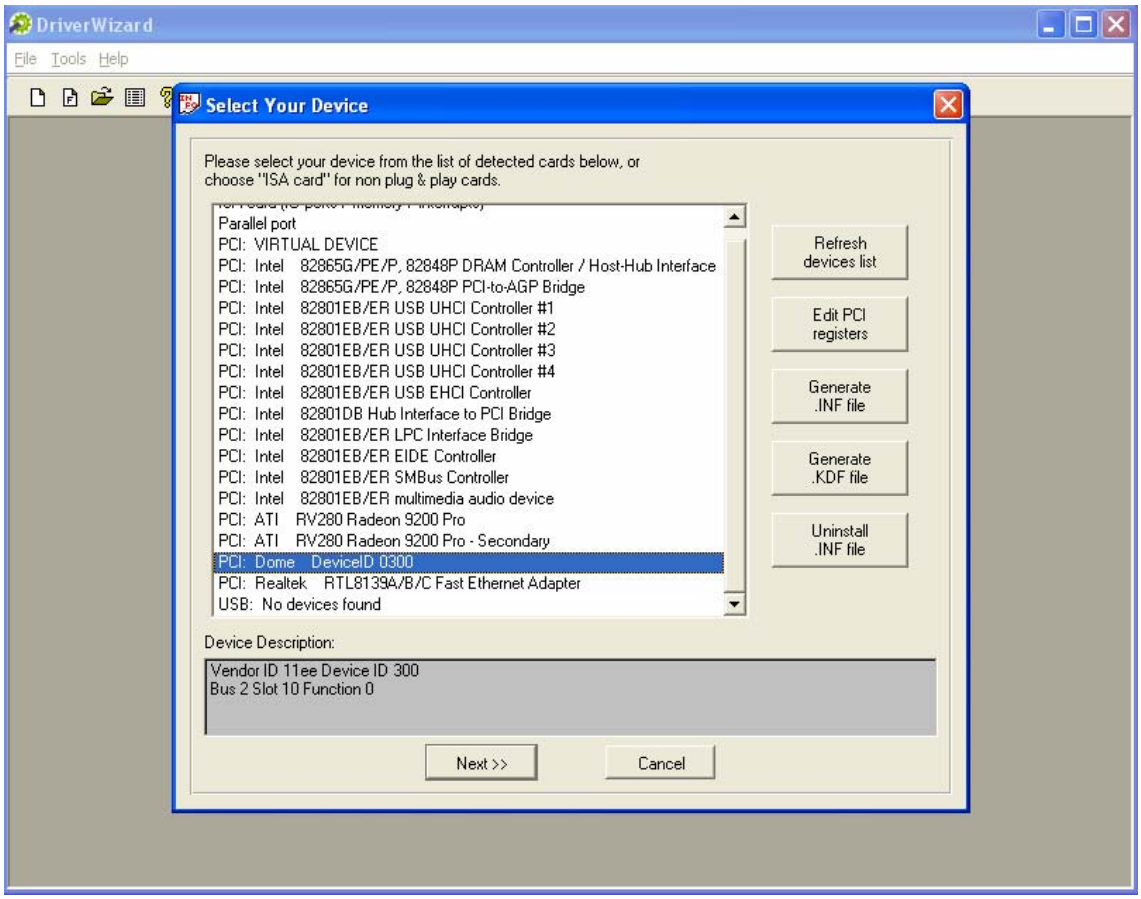

Figura 9. Captura de la aplicación WinDriver (selección de dispositivo).

A través de esta herramienta es posible crear controladores para dispositivos USB o PCI (aunque permite muchos otros estándares) sin necesidad de conocimientos de desarrollo a bajo nivel o del núcleo del sistema operativo.

WinDriver ofrece herramientas de diagnóstico hardware, acceso directo al mismo (bien a registros, bien a espacios de memoria), utilidades de depuración para los controladores y generación de bloques de código o aplicaciones de ejemplo para su posterior uso desde otras aplicaciones en diversos entornos de desarrollo y lenguajes.

[3] http://www.jungo.com/windriver.html

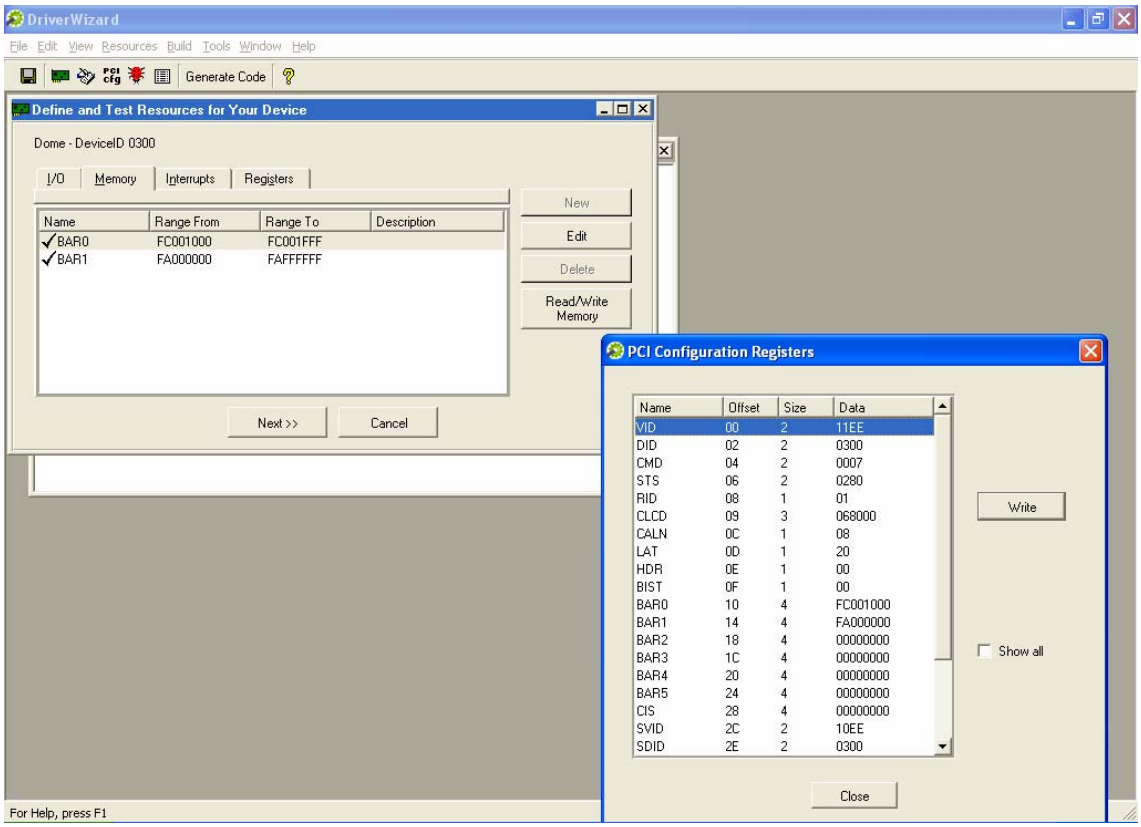

Figura 10. Captura de la aplicación WinDriver (definición de recursos).

La principal ventaja de generar controladores con una aplicación de este tipo consiste en poder abstraer elementos de desarrollo específicos del sistema operativo, permitiendo concentrarse en la funcionalidad del hardware.

En nuestro caso se ha utilizado para la creación del driver para el reconocimiento automático de la FPGA por el sistema operativo Windows y posibilitar así su uso en aplicaciones software.

4. Algoritmo propuesto: AES

Introducción

Como ejemplo de aplicación integrada en el sistema se eligió el algoritmo criptográfico AES[1]. Este algoritmo es el nuevo estándar de criptografía simétrica adoptado en el FIPS 197[FIPS1] (Federal Information Processing Standards).

Desde 1977 que apareció la primera versión del estándar FIPS 46, se asume como estándar el algoritmo DES[2], y sus posteriores revisiones en 1983, 1988, 1993, y 1999. Desde sus orígenes, han existido opiniones contrapuestas acerca de DES, sin embargo nunca se ha producido un ataque que descifrase por completo la clave a partir de la información pública. En realidad, el problema de DES era su corta longitud de clave, que lo iba comprometiendo poco a poco. En la última revisión de DES, en octubre de 1999, el algoritmo fue suplantado por TDES o 3DES, una versión múltiple de DES, designado como TDEA[3]. De hecho, en ese momento ya se tenían planes de encontrar un reemplazo definitivo a DES.

A pesar del gran número de algoritmos que en la época estaban presentes, entre ellos IDEA, RC5, skipjack, 3-way, FEAL, LOKI, SAFER, SHARK,... el NIST decidió convocar un concurso que tuvo como principales objetivos obtener un algoritmo simétrico que garantizase su seguridad para los próximos 20 años a partir del año 2000. En la convocatoria del 2 de enero de 1997 se admitieron 15 algoritmos; en agosto de 1998, en la primera conferencia AES, se discutieron los algoritmos sometidos y posteriormente en la segunda conferencia AES, en marzo de 1999, se realizaron los últimos comentarios. Finalmente, en agosto de 1999, se comunicaron los cinco finalistas: MARS, RC6, Rijndael, Serpent y Twofish.

En abril de 2000 se llevó a cabo la tercera conferencia AES, haciéndose los últimos análisis, para que finalmente el 2 de octubre del año 2000 se diera a conocer el ganador: el algoritmo Rijndael como AES[DAEM1]. Esto se asumió oficial el 26 de noviembre de 2001 en el FIPS 197. A partir de esa fecha hay conferencias especiales para analizar la situación actual de AES.

El algoritmo Rijndael fue elegido principalmente por garantizar seguridad, al ser inmune a los ataques conocidos, tener un diseño simple y poder ser implementado en la mayoría de los escenarios posibles, desde dispositivos con recursos limitados, como smart cards, hasta procesadores paralelos. El tiempo ha permitido que AES sea adaptado poco a poco, desde los protocolos más usados como SSL, hasta las aplicaciones más especializadas, como VoIP[4].

Aunque Rijndael está diseñado para manejar muchos casos de longitudes de claves y de bloques (puede ser especificado por una clave que sea múltiplo de 32 bits, con un mínimo de 128 bits y un máximo de 256 bits), finalmente AES definido en el estándar determina solo permitir los casos de bloques de 128 bits, y longitudes de claves de 128, 192 y 256. Por otra parte, la longitud de 128 bits garantiza la seguridad del sistema hasta después del año 2030.

1 Advanced Encryption Standard
2 Data Encryption Standard
3 Triple Data Encryption Algorithm
4 Voice over IP

La descripción de AES es simple si se estudian todos los elementos que lo forman. Ésta consiste en dos partes, la primera en el proceso de cifrado y la segunda en el proceso de generación de las subclaves. Una primera aproximación se muestra la siguiente figura:

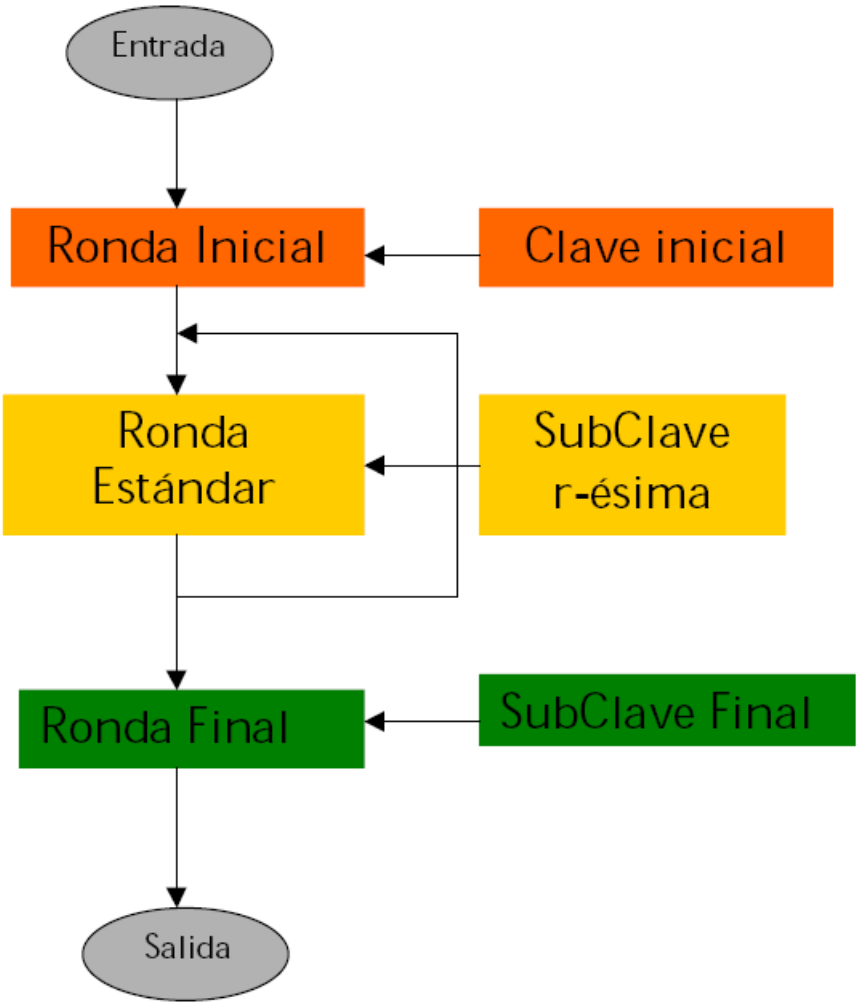

Figura 11. Diagrama de ejecución del proceso AES.

De manera un poco más detallada el algoritmo AES llega a ser:

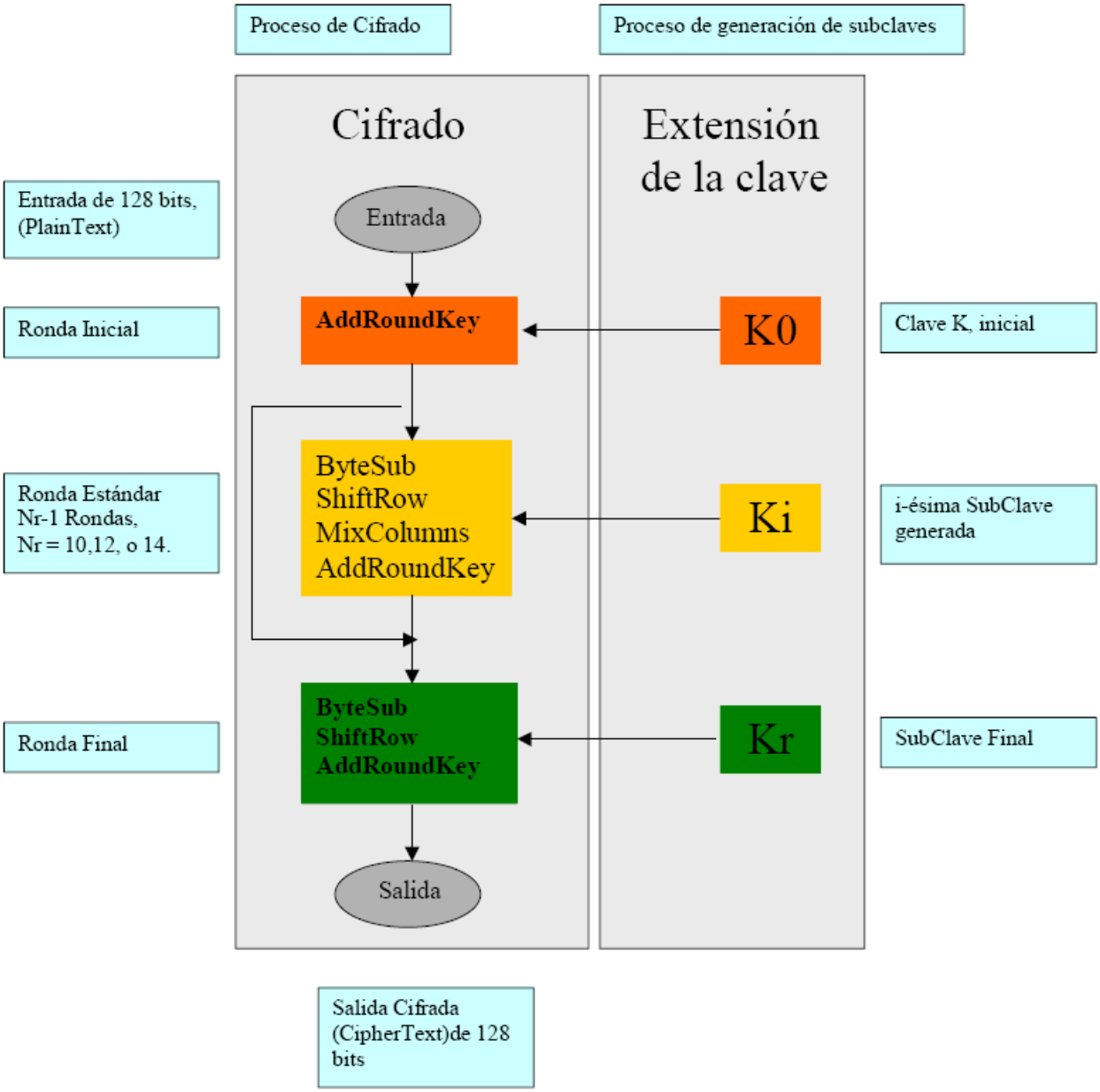

Figura 12. Diagrama detallado del proceso AES.

Por lo tanto la descripción de AES consiste de dos partes, en describir el proceso de "Cifrado", y el proceso de "Generación de las subclaves" o "Extensión de la clave K". El bloque de cifrado tiene una longitud de 128 bits, la longitud de la clave K varia de 128, 192 y 256 bits, en cada caso AES tiene 10,12, y 14 rondas respectivamente.

El proceso de cifrado consiste esencialmente en la descripción de las cuatro etapas básicas de AES: ByteSub, ShiftRow, MixColumns y AddRoundKey:

Etapa SubBytes

En la etapa SubBytes, cada byte en la matriz es actualizado usando la caja-S de Rijndael de 8 bits. Esta operación provee la no linealidad en el cifrado. La caja-S utilizada proviene de la función inversa alrededor del GF(28), conocido por tener grandes propiedades de no linealidad. Para evitar ataques basados en simples propiedades algebraicas, la caja-S se construye por la combinación de la función inversa con una transformación afín invertible. La caja-S también se elige para evitar puntos estables también cualesquiera puntos estables opuestos.

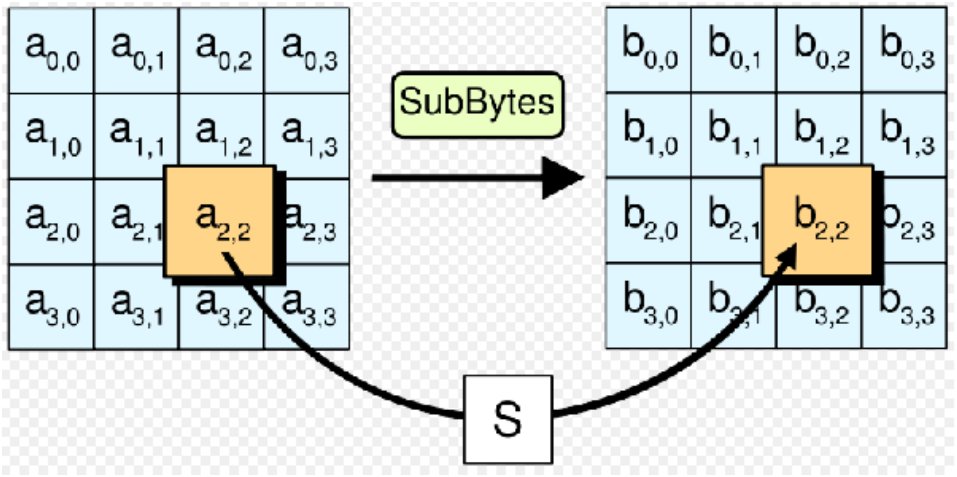

Figura 13. Fase SubBytes.

En la fase de SubBytes, cada byte en el state es reemplazado con su entrada en una tabla de búsqueda fija de 8 bits, *S*; $b_{ij} = S(a_{ij})$.

Etapa ShiftRows

El paso ShiftRows opera en las filas del state; rota de manera cíclica los bytes en cada fila por un determinado offset. En AES, la primera fila queda en la misma posición. Cada byte de la segunda fila es rotado una posición a la izquierda. De manera similar, la tercera y cuarta filas son rotadas por los offsets de dos y tres respectivamente. De esta manera, cada columna del state resultante del paso ShiftRows está compuesta por bytes de cada columna del state inicial (variantes de Rijndael con mayor tamaño de bloque tienen offsets distintos).

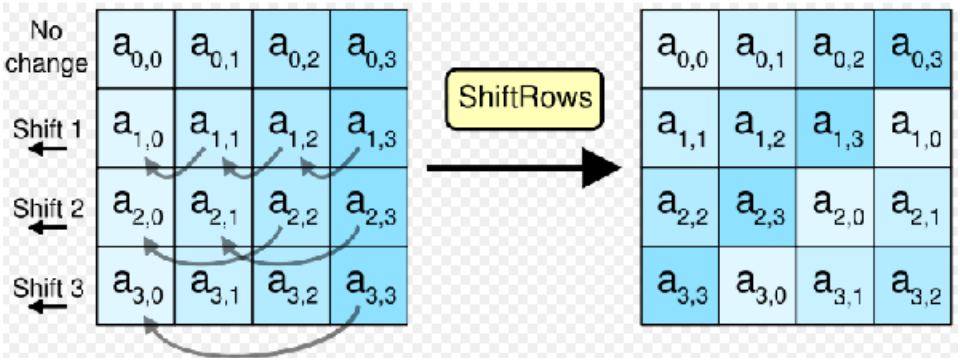

Figura 14. Fase SubRows.

En el paso ShiftRows, los bytes en cada fila del state son rotados de manera cíclica hacia la izquierda. El número de lugares que cada byte es rotado difiere para cada fila.

Etapa MixColumns

En el paso MixColumns, los cuatro bytes de cada columna del state se combinan usando una transformación lineal inversible. La función MixColumns toma cuatro bytes como entrada y devuelve cuatro bytes, donde cada byte de entrada influye todas las salidas de cuatro bytes. Junto con ShiftRows, MixColumns implica difusión en el cifrado. Cada columna se trata como un polinomio GF (2^8) y luego se multiplica el módulo $x^4 + 1$ con un polinomio fijo $c(x)$. El paso MixColumns puede verse como una multiplicación matricial en el campo finito de Rijndael.

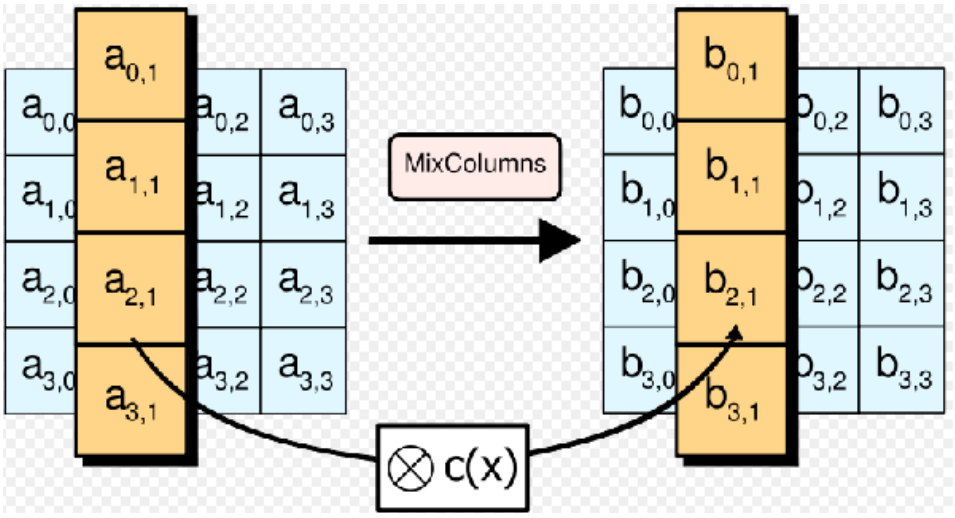

Figura 15. Etapa MixColumns.

En el paso MixColumns, cada columna del state es multiplicada por un polinomio constante *c(x)*.

Etapa AddRoundKey

En el paso AddRoundKey, la subclave se combina con el state. En cada ronda se

obtiene una subclave de la clave principal, usando la iteración de la clave; cada subclave es del mismo tamaño del state. La subclave se agrega combinando cada byte del state con el correspondiente byte de la subclave usando XOR.

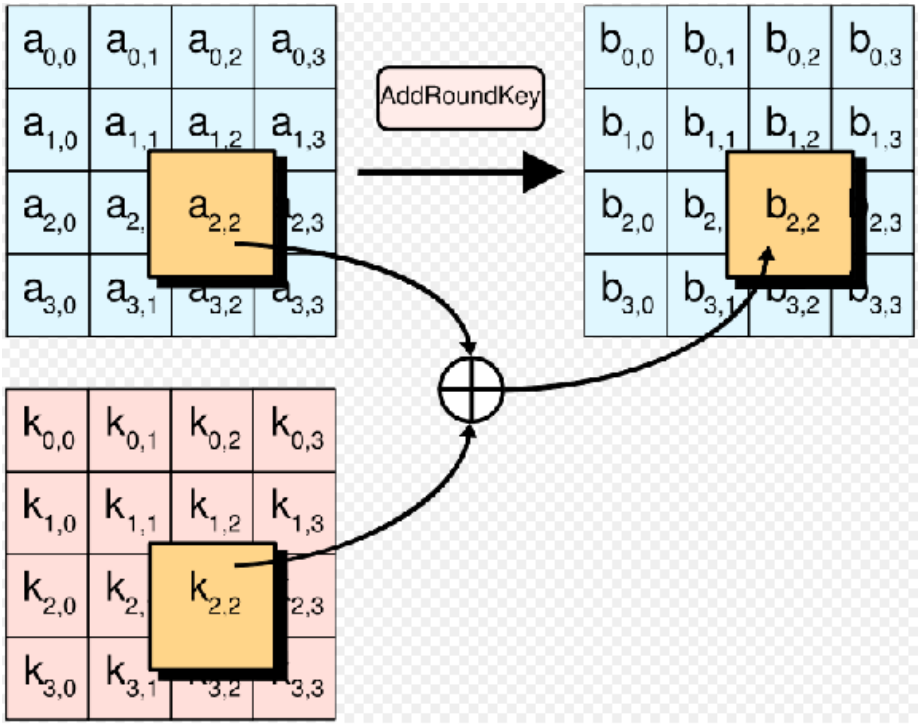

Figura 16. Etapa AddRoundKey.

En el paso AddRoundKey, cada byte del state se combina con un byte de la subclave usando la operación XOR.

Seguridad

Hasta 2005, no se ha encontrado ningún ataque exitoso contra el AES. La Agencia de Seguridad Nacional de los Estados Unidos (NSA) revisó todos los finalistas candidatos al AES, incluyendo el Rijndael, y declaró que todos ellos eran suficientemente seguros para su empleo en información no clasificada del gobierno de los Estados Unidos. En junio del 2003, el gobierno de los Estados Unidos anunció que el AES podía ser usado para información clasificada:

> "*The design and strength of all key lengths of the AES algorithm (i.e., 128, 192 and 256) are sufficient to protect classified information up to the SECRET level. TOP SECRET information will require use of either the 192 or 256 key lengths. The implementation of AES in products intended to protect national security systems and/or information must be reviewed and certified by NSA prior to their acquisition and use.*"

Este hecho marca la primera vez que el público ha tenido acceso a un cifrador aprobado por la NSA para información secreta. Es interesante notar que muchos productos

públicos usan llaves de 128 bits por defecto; es posible que la NSA de Estados Unidos sospeche de una debilidad fundamental en llaves de este tamaño, o simplemente prefieren tener un margen de seguridad para documentos secretos (que deberían conservar la seguridad durante décadas en el futuro).
El método más común de ataque hacia un cifrador por bloques consiste en intentar varios ataques sobre versiones del cifrador con un número menor de rondas. El AES tiene 10 rondas para llaves de 128 bits, 12 rondas para llaves de 192 bits, y 14 rondas para llaves de 256 bits. Hasta 2005, los mejores ataques conocidos son sobre versiones reducidas a 7 rondas para llaves de 128 bits, 8 rondas para llaves de 192 bits, y 9 rondas para llaves de 256 bits[FERG1].

Algunos criptógrafos muestran preocupación sobre la seguridad del AES. Piensan que el margen entre el número de rondas especificado en el cifrador y los mejores ataques conocidos es muy pequeño. El riesgo es que se puede encontrar alguna manera de mejorar los ataques y, de ser así, el cifrador podría ser roto. En el contexto criptográfico se considera "roto" un algoritmo si existe algún ataque más rápido que una búsqueda exhaustiva - ataque por fuerza bruta. De modo que un ataque contra el AES de llave de 128 bits que requiera 'sólo' 2^{120} operaciones sería considerado como un ataque que "rompe" el AES aún tomando en cuenta que por ahora sería un ataque irrealizable. Hasta el momento, tales preocupaciones pueden ser ignoradas. El ataque de fuerza bruta más conocido ha sido contra una clave de 64 bits RC5 por distributed.net[5].

Otra preocupación es la estructura matemática de AES. A diferencia de la mayoría de cifradores de bloques, AES tiene una descripción matemática muy ordenada. Esto no ha llevado todavía a ningún ataque, pero algunos investigadores están preocupados que futuros ataques quizá encuentren una manera de explotar esta estructura.

En 2002, un ataque teórico, denominado "ataque XSL", fue anunciado por Nicolas Courtois y Josef Pieprzyk, mostrando una potencial debilidad en el algoritmo AES. Varios expertos criptográficos han encontrado problemas en las matemáticas que hay por debajo del ataque propuesto, sugiriendo que los autores quizá hayan cometido un error en sus estimaciones. Si esta línea de ataque puede ser tomada contra AES, es una cuestión todavía abierta. Hasta el momento, el *ataque XSL* contra AES parece especulativo; es improbable que nadie pudiera llevar a cabo en la práctica este ataque.

Eficiencia

La eficiencia del algoritmo AES es otra de las tan anheladas propiedades con las que debe contar un algoritmo criptográfico. La eficiencia además de depender del diseño del algoritmo y de su implementación depende de la plataforma donde se ejecute. Obviamente las mejores eficiencias alcanzadas para cualquier algoritmo se alcanzan en la implementación de hardware dedicado. Se tiene entonces que la eficiencia del algoritmo se debe analizar separadamente tanto en software como en hardware.

Respecto a la eficiencia en software del algoritmo, el código de partida de AES es el conocido que muestra el funcionamiento de las funciones básicas de AES. Este código no tiene por objetivo ser el más eficiente; sin embargo, analizando cada una de sus

[5] http://en.wikipedia.org/wiki/Distributed.net

partes podemos saber cuales de ellas pueden ser optimizadas. La operación mas costosa en tiempo es el obtener inversos multiplicativos del campo GF (28), en este caso esta operación es precalculada y la operación es substituida por una consulta de caja-S. Con esta misma técnica varios cálculos del algoritmo son precalculados y entonces se evitan los cálculos reemplazándolos por consultas de tablas.

Respecto a la velocidad de los algoritmos la manera más simple de medirla es usando Mb/seg (cuántos millones de bits por segundo procesa). Los tres procesos más comunes en un algoritmo son el cifrado, el descifrado y el programa de claves. Estos tres procesos se miden de manera independiente.

Los siguientes datos nos dicen algunos resultados más representativos que se han obtenido a lo largo del estudio de Rijndael. Nos sirven como referencia para poder conocer las velocidades estándares con que se cuentan en nuestros días.

Plataforma	Autor	Longitud de clave	Velocidad de cifrado	Velocidad de descifrado
PIV 3.2GHz, ASM	H. Lipmaa[LIPM1]	128	1537.9.7 Mb/s	1519.9 Mb/s
PPro 200MHz, C	B.Gladman	128	70.7 Mb/s	71.5 Mb/s
PIV 1.8GHz, C++	C. Devine	128	646 Mb/s	646 Mb/s
PII 450MHz MMX, ASM	K. Aoki, H. Lipmaa	128	243 Mb/s	243 Mb/s

Tabla 1. Análisis de eficiencia del algoritmo SW Rijndael sobre distintas plataformas.

De los diseños de hardware damos algunos de los resultados también representativos de esta otra parte de la eficiencia con AES.

Tecnología	Autor	Longitud de clave	Velocidad de cifrado
ASIC	H. Kuo, I. Verbauwhede[HKUO1]	128	2.29 Gb/s
VLSI	H. Kuo, I. Verbauwhede[HKUO2]	128	1.82 Gb/s
FPGA	A. J. Elbirt[ELBI1]	128	2 Gb/s

Tabla 2. Análisis de eficiencia del algoritmo HW Rijndael sobre distintas tecnologías.

Implementación Hardware de AES en VHDL

El proceso de integración en el sistema del módulo hardware que implementa el algoritmo AES se ha dividido en dos fases:

- Búsqueda de un open-core que implementase el algoritmo y del que se detalla cómo funciona en el siguiente apartado.
- Conexión de este core al bus PCI de la FPGA a través de otro módulo open-core que actúa como puente: el “Bridge PCI-Wishbone”.

Detalles de la implementación del open core AES

El algoritmo está implementado en modo ECB (Electronic Codebook), que es la forma más simple de cifrado, consistente en dividir el texto plano original que se intenta cifrar en bloques y procesar cada uno de estos bloques de forma independiente. El otro modo de operación de un algoritmo criptográfico es el modo CBC (Chiper Block Chaining), en el que el procesamiento de cada uno de los bloques en que se dividió el texto plano original depende del cifrado de los bloques anteriores.

La arquitectura del módulo AES implementado en nuestro sistema se puede resumir en el siguiente diagrama de bloques:

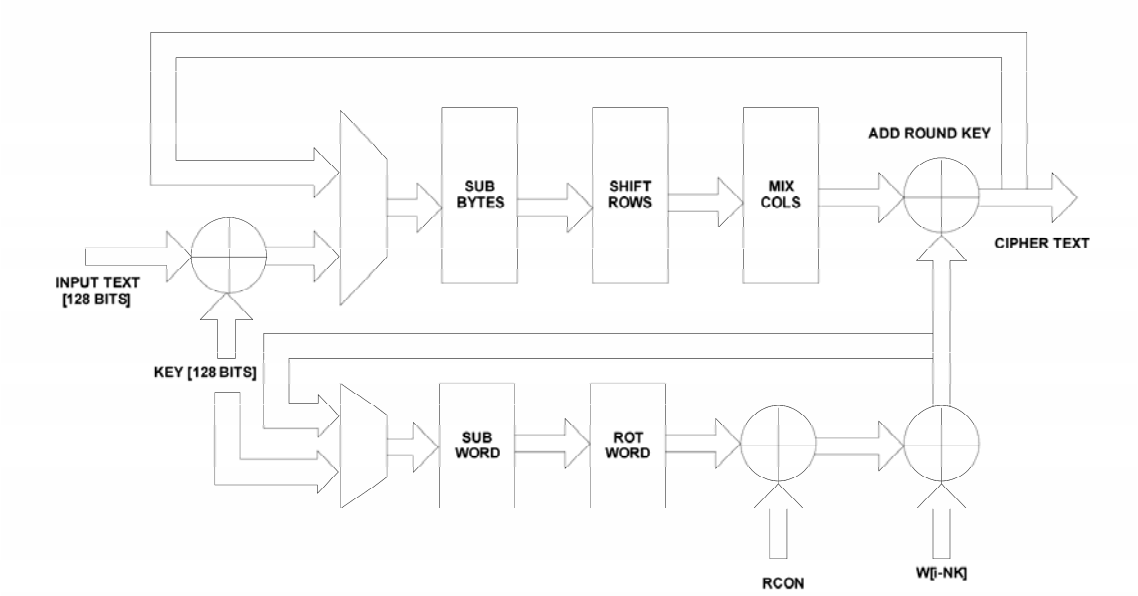

Figura 17. Diagrama de bloques estructural del core.

El proceso de descifrado sigue el mismo orden que la encriptación excepto por la inclusión de otra etapa de MixColumns en la generación de claves.

Secuencia de encriptado/desencriptado

Los datos de entrada (texto plano de 128 bits) y la clave (otros 128 bits) se dividen en dos bloques de 64 bits que consume el sistema en dos ciclos consecutivos. Para ello se utiliza la señal *load*. El primer bloque de 64 bits de los datos de entrada y el primer bloque de 64 bits de la clave se leen en el flanco de subida del ciclo siguiente tras haberse puesto a alta la señal de *load*. El otro bloque de datos de entrada de 64 bits y el resto de la clave son leídos en el flanco de subida del ciclo siguiente tras haberse puesto a baja la señal de *load*. Por lo tanto, el texto plano completo y la clave son cargados solo cuando la señal *load* hace una transición baja-alta-baja (básicamente un pulso de reloj).

El proceso comenzará cuando se haga una transición baja-alta-baja (otro pulso de reloj) de la señal *start*. La salida será válida cuando la señal *done* se active, trece ciclos después del flanco de bajada de la señal *start*. La señal *done* continuará a alta hasta el próximo pulso de la señal *start*.

Este es un diagrama de tiempo que muestra perfectamente el proceso descrito:

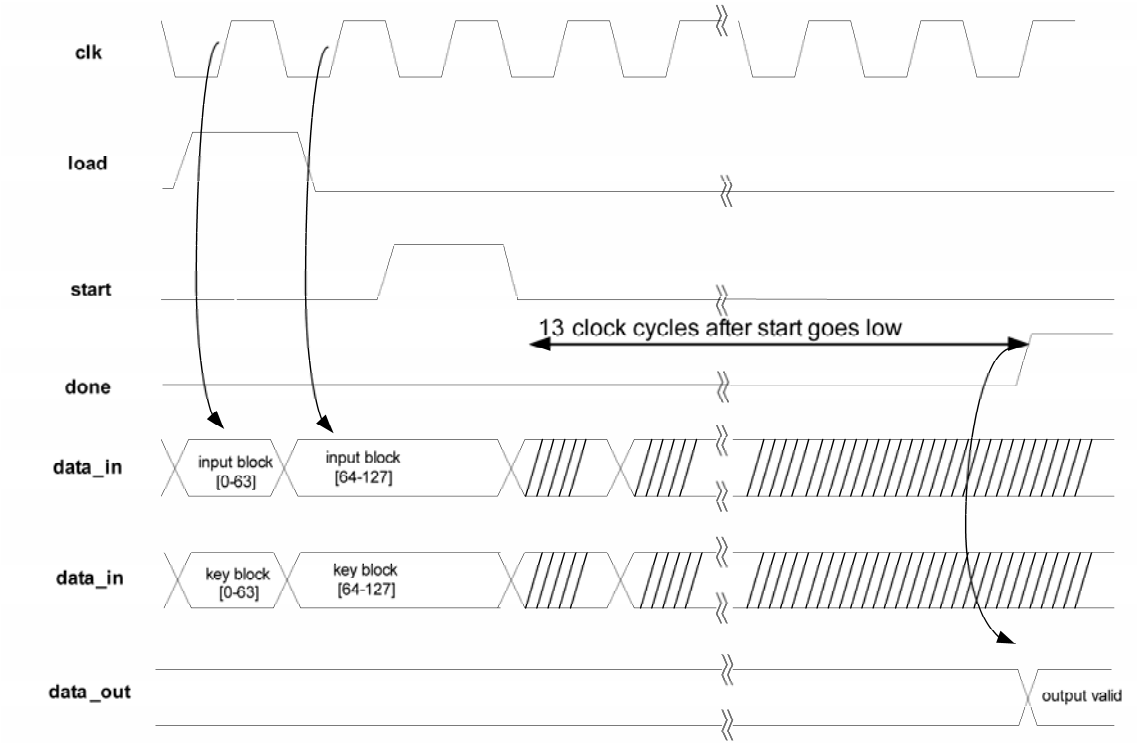

Figura 18. Diagrama de tiempos del proceso de cifrado/descifrado.

La arquitectura no está segmentada, por lo que solamente es capaz de hacer cifrado/descifrado en serie, es decir, uno detrás de otro. La salida solo es válida después de que la señal *done* se ponga a alta. Si se modifica la señal de *start* durante el proceso de cifrado/descifrado el proceso devolverá una salida errónea.

Implementación Software de AES en C++

Rijndael es el algoritmo de cifrado por bloques seleccionado por el NIST[6] como candidato para AES, sustituto del algoritmo DES.

El cifrado tiene longitud de bloque y clave variables. Con esta implementación son posibles hasta 9 combinaciones de longitud de clave y bloque (128, 192 ó 256 bits cada una).

Esta es una implementación de código abierto en C++ realizada por George Anescu[GANE1] en 2002. Está basada en una implementación previa en Java con el toolkit Cryptix realizada por Raif S. Naffah y Paulo S.L.M. Barreto[VRIJ1]. Esta implementación ha sido probada contra el test KAT[7] publicado por los autores del método, con resultado satisfactorio.

El código ofrece una clase *CRijndael* con una serie de funciones básicas para llevar a cabo los procesos de cifrado, descifrado y configuración. Un listado de las más importantes podría ser el siguiente:

- *MakeKey()*, que se utiliza para expandir una clave proporcionada por el usuario en una clave de sesión.
- *EncryptBlock()*, que se utiliza para cifrar un bloque del tamaño de bloque especificado.
- *DecryptBlock()*, que realiza la función inversa a la función anterior.
- *Encrypt()*, que se utiliza para cifrar grandes bloques de datos. Esta función puede operar en modo ECB, CBC o CFB.
- *Decrypt()*, que realiza la función inversa a la función anterior.

6 National Institute of Standards and Technology
7 Known Answer Test

Comparativas

Se han realizado mediciones de tiempos entre los tres modos de ejecución diferentes implementados en el sistema y para tres tamaños típicos diferentes de entrada de datos.

Los dos primeros modos de ejecución (Software y Hardware) hacen escrituras simples sobre los registros del hardware implementado. El primero de ellos (Software), se refiere al modelo de la máquina de estados, que está dirigido por el usuario (veáse Arquitectura-Módulo funcional para más información). El segundo (Hardware), se refiere al modelo dirigido por el propio sistema. Finalmente, el tercer modo de ejecución se refiere a la implementación con memorias intermedias que actúan como búffers para lecturas y escrituras en ráfaga por el bus PCI.

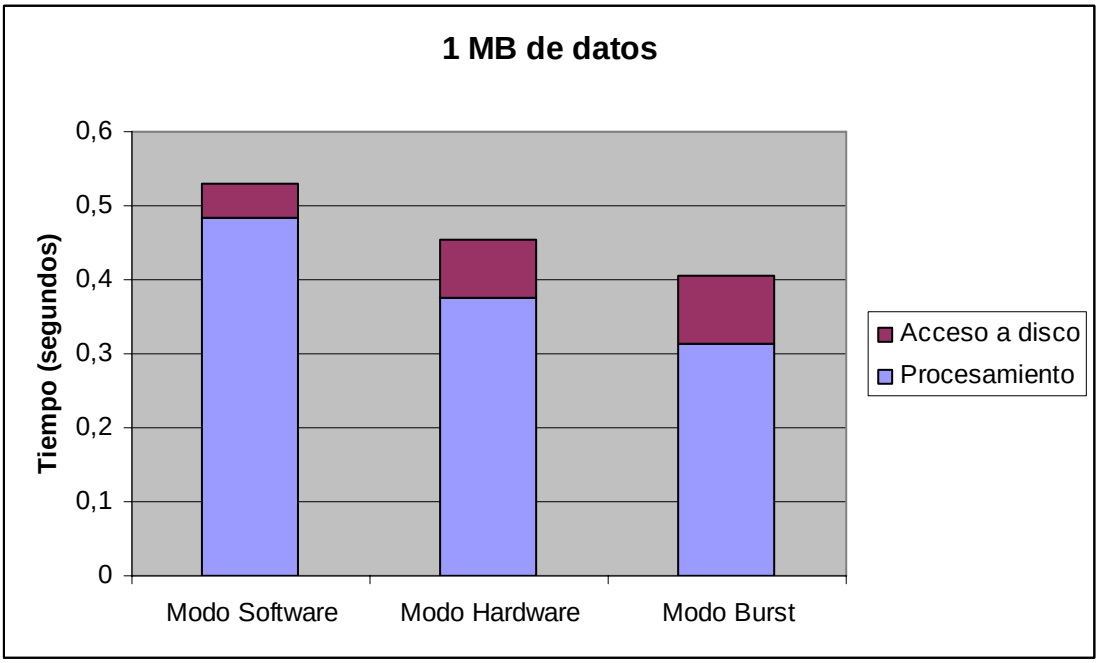

Gráfico 1. Comparativa entre los modos de ejecución implementados para un fichero de 1 MB.

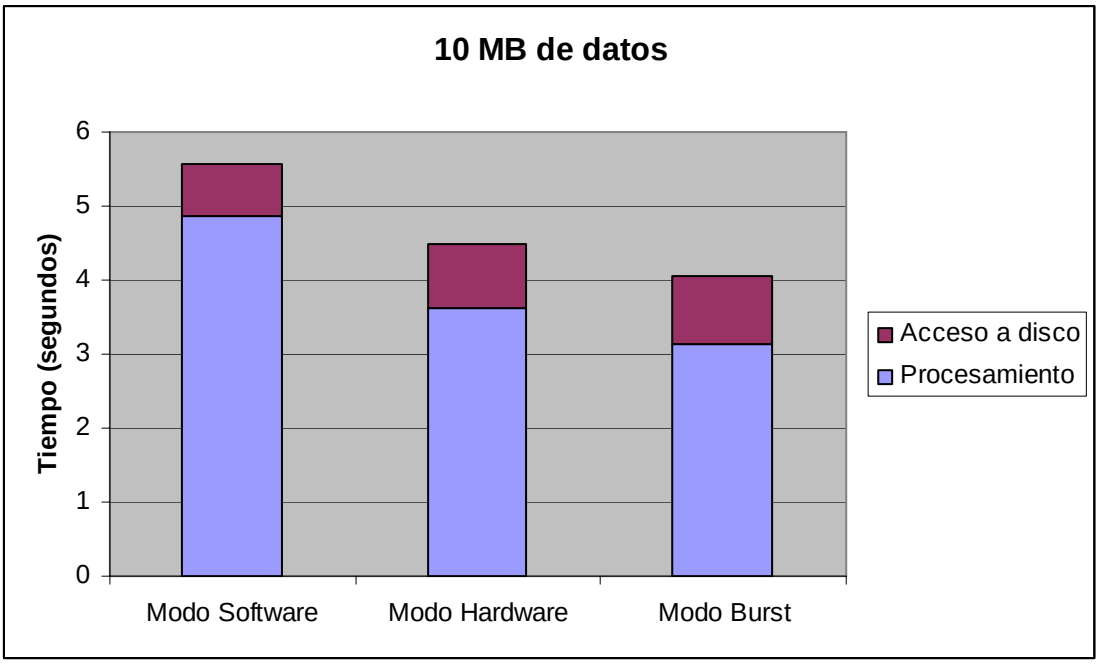

Gráfico 2. Comparativa entre los modos de ejecución implementados para un fichero de 10 MB.

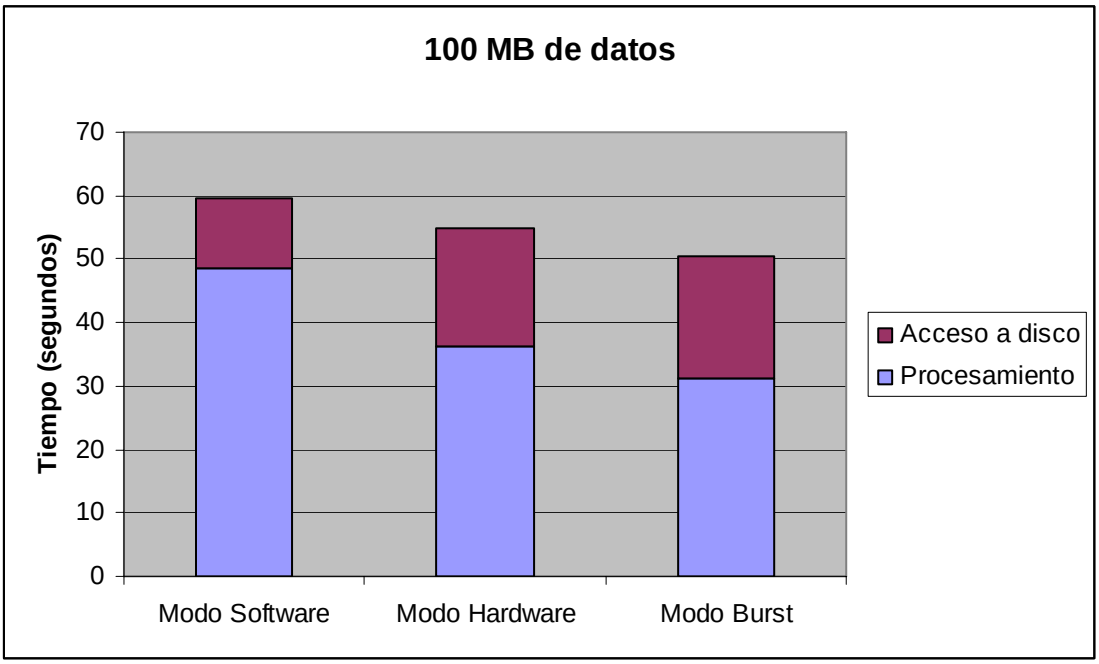

Gráfico 3. Comparativa entre los modos de ejecución implementados para un fichero de 100 MB.

5. Desarrollo del proyecto

Arquitectura

El sistema se compone de un diseño hardware implementado en una FPGA montada sobre una placa que se comunica con el PC a través del bus PCI. El principal objetivo de este proyecto ha sido conocer cómo se realiza esta comunicación e intentar minimizar el impacto de ésta en la ejecución de las aplicaciones propuestas.

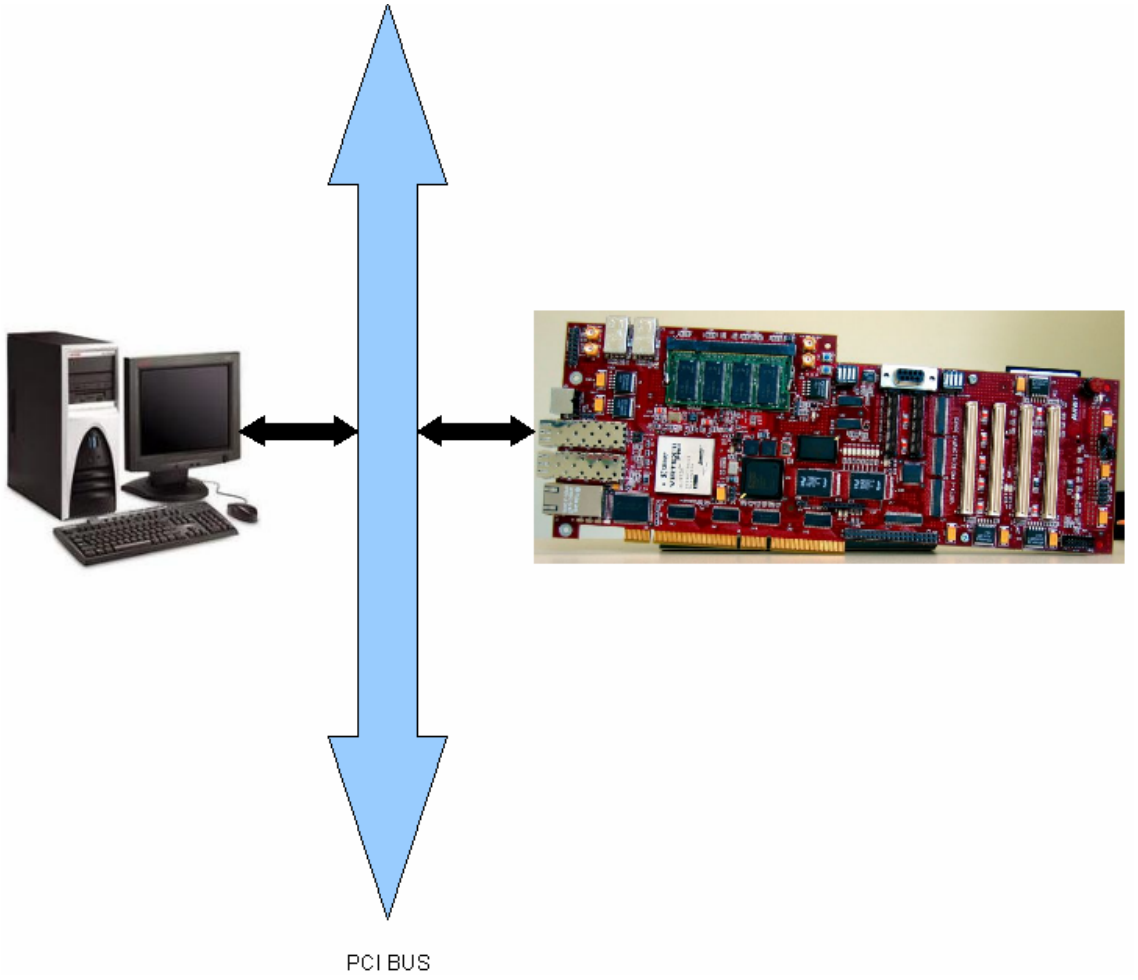

Figura 19. Esquema general del proyecto en el que se muestra la relación PC - PCI – FPGA.

Para ello, en primer lugar se tuvo que desarrollar un driver para que el sistema operativo sobre el que se ha trabajado (Microsoft Windows) reconociese la tarjeta como un dispositivo PCI. Para este punto fue de gran ayuda el programa WinDriver, el cual permite una creación rápida e intuitiva de controladores, lo que permitió configurar el dispositivo de forma sencilla y transparente, definiendo los espacios de direcciones de memoria sobre las que se iba a mapear el dispositivo hardware. Una vez instalado, y con ayuda de la misma aplicación, se desarrolló una aplicación software que, haciendo uso del citado driver, permitiese leer y escribir en el dispositivo usando los registros BAR[1], usando funciones ya definidas. Estos registros son los encargados de informar a la BIOS del número y tamaño de los espacios de memoria o de entrada/salida necesarios, y de

[1] Base Adress Register

identificar una zona de memoria del sistema mapeada sobre un determinado dispositivo PCI. En este caso, se utilizan dos registros BAR:

- **BAR 0**: destinado a la configuración del "Wishbone bridge", módulo del que se hablará más adelante.

- **BAR 1**: destinado a la entrada/salida, mapeado directamente en memoria (sin traducción de direcciones). Este hecho simplifica la comunicación, ya que de esta forma, para leer o escribir un dato en un registro del dispositivo, basta con tan solo leer o escribir la dirección de memoria asociada a dicho registro.

En cuanto a la parte hardware del proyecto, la parte más compleja fue la comunicación de la tarjeta con el PC a través del bus PCI. Para este cometido fue de gran ayuda el uso de un IP core[2] ("Wishbone bridge") que actúa de puente entre el bus PCI y un core con una interfaz Wishbone. El objetivo del "Wishbone bridge" es el simplificar el envío y la recepción de datos a través del bus PCI, encapsulando dichas funciones con una capa de interfaz Wishbone. Así, es posible utilizar la comunicación vía PCI sin necesidad de conocimientos profundos acerca de la implementación de dicho puerto y sin tener que lidiar con todas y cada una de las señales que intervienen en el protocolo del mismo.

Interfaz Wishbone

La interfaz Wishbone es una especificación abierta[WISH1] que define un método de interconexión entre módulos de diseño digital dentro de un circuito integrado. Esta arquitectura resuelve un problema básico en el diseño de circuitos integrados, que es cómo conectar IP cores entre sí de una manera simple, flexible y transportable. El interfaz Wishbone estandariza los interfaces utilizados por IP cores, lo que simplifica su interconexión para la creación de sistemas a medida sobre un chip (SoC[3]). La especificación Wishbone pertenece al dominio público, por lo que puede ser libremente copiada, distribuida y utilizada para el diseño y producción de componentes de circuitos integrados sin necesidad de pagar ningún tipo de licencia ni infringir ninguna patente.

La ventaja de pasar de un tipo de bus a otro radica en que la especificación Wishbone está pensada para interconectar diseños dentro de un mismo sistema. Las interfaces y los ciclos de transferencia de datos son iguales para todos los cores IP sin importar su función (controlador de memoria, interfaz PCI, registros, etc.). Por ejemplo, no es necesario conocer el funcionamiento del bus PCI (cuyo protocolo es mucho mas complejo que el protocolo de actuación del interfaz Wishbone) para hacer un diseño como el del presente proyecto. Esto, además de facilitar la tarea, permite la reutilización. Si por ejemplo ya existe un controlador de memorias con interfaz Wishbone, basta con diseñar la interconexión entre dicho controlador y el "Wishbone bridge" para conseguir un sistema funcional.

[2] Intellectual Property core
[3] System-on-Chip

Wishbone utiliza una arquitectura maestro/esclavo, lo que significa que aquellos módulos con interfaz maestro inician las transacciones de datos realizadas con los módulos esclavos.

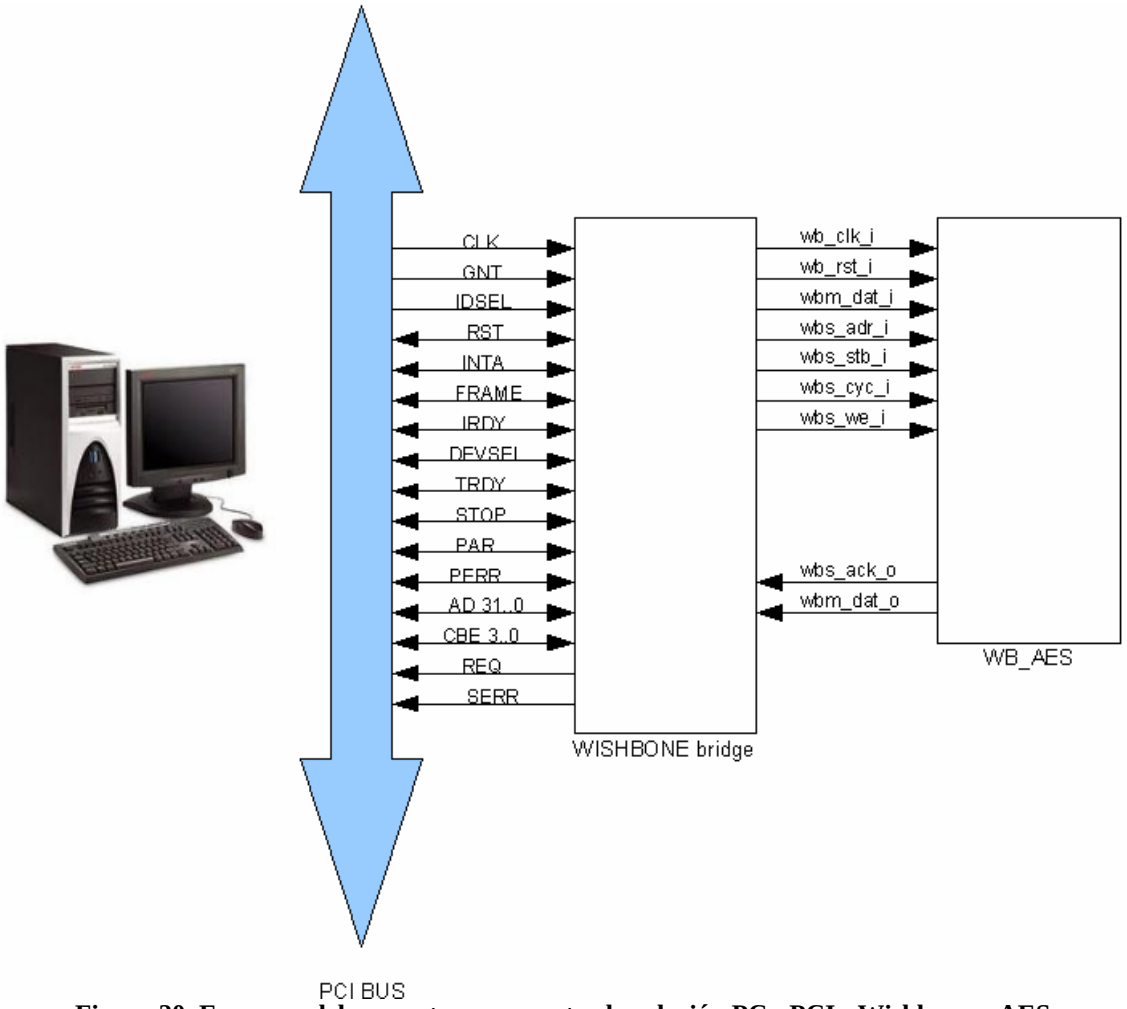

Figura 20. Esquema del proyecto que muestra la relación PC - PCI - Wishbone – AES.

A continuación, se describen las señales del interfaz Wishbone utilizadas por el "Wishbone bridge". Al configurar el módulo propio conectado al interfaz Wishbone como esclavo, el conjunto de señales que es necesario utilizar es mínimo. Todas ellas son activas a alta y la terminación *_i* y *_o* indica si son entradas o salidas al core, respectivamente.

wb_clk_i: reloj. La señal de reloj coordina todas las actividades para la lógica dentro de un dispositivo Wishbone. Todas las señales de salida son registradas en el flanco de subida de esta señal. Todas las entradas deben estar estables antes del flanco de subida de la señal de reloj.

wb_rst_i: reset. La señal de reset fuerza al interfaz Wishbone a reiniciarse. No implica el reinicio de los cores conectados al interfaz (aunque normalmente se usa para este cometido).

wbm_dat_i: bus de entrada de datos. Su ancho de palabra es de 32 bits.

wbs_cyc_i: indica que un ciclo válido del bus está en progreso. La señal está a alta durante la duración de todos los ciclos. Por ejemplo, durante un ciclo de transferencia de un bloque de datos, la señal estará a alta desde la primera hasta la última transferencia de datos.

wbs_stb_i: la señal de entrada *strobe*, cuando está a alta, indica que el dispositivo esclavo está seleccionado.

wbs_we_i: la señal de *write enable* indica si el ciclo actual es un ciclo de lectura o escritura. La señal está a baja durante los ciclos de lectura y a alta durante los ciclos de escritura.

wbs_adr_i: vector de dirección que permitirá a nuestro core determinar en función de los bits 5 al 2 del propio vector, qué registro de nuestro módulo se intenta leer o escribir.

wbm_dat_o: bus de salida de datos. Su ancho de palabra es de 32 bits.

wbs_ack_o: señal de salida *acknowledge*, cuando está a alta, indica la terminación de un ciclo normal (cuando tanto como *wbs_stb_i* como *wbs_cyc_i* están a alta).

En resumen, las únicas señales que realmente utiliza el módulo diseñado son las propias de los dispositivos slave:

Señales de entrada	*wb_clk_i, wb_rst_i, wbm_dat_i, wbs_adr_i, wbs_stb_i, wbs_cyc_i, wbs_we_i*
Señales de salida	*wbs_ack_o, wbm_dat_o*

Tabla 3. Resumen de señales del módulo esclavo Wishbone.

El resto de señales se ignoran, aunque están incluidas en nuestro diseño para preservar la especificación Wishbone.

Módulo funcional

El módulo funcional que se conecta al "Wishbone bridge" es el "WB_AES". Este módulo (configurado como un módulo esclavo, ya que simplifica en gran manera el uso de la interfaz) está compuesto por un microcontrolador (básicamente una máquina de estados) que, tras recibir las "señales de control Wishbone" y unos datos de entrada (dato de entrada, clave, señal de *start* y *load*, modo de carga de registros, etc), realiza unas determinadas operaciones sobre estos (cometido éste del modulo "AES_128_fast") y los devuelve al "Wishbone bridge" para que éste los escriba en el bus PCI (datos de salida y señal *done*).

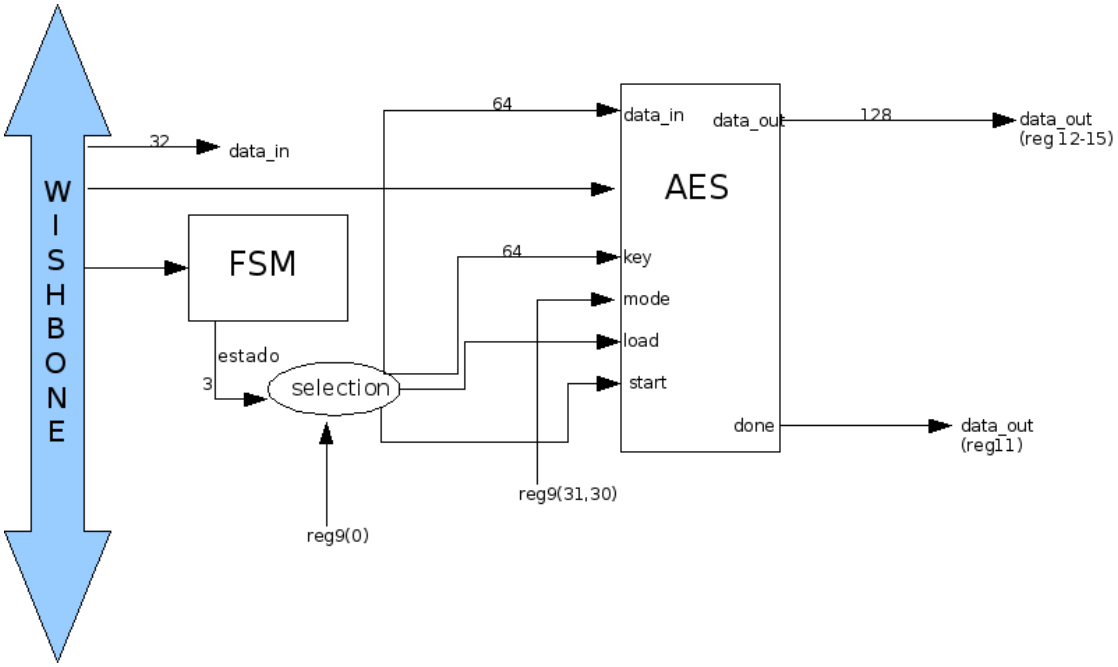

Figura 21. Arquitectura general del sistema.

Para almacenar los datos de entrada, operar sobre ellos y devolver los datos de salida, el módulo utiliza 16 registros de 32 bits que leerá y escribirá utilizando las señales de control del interfaz Wishbone (reloj, reset, write enable, strobe, etc). Para elegir el registro que se va a leer o escribir se utilizarán los bits 5 al 2 de la señal *wbs_adr_i*.

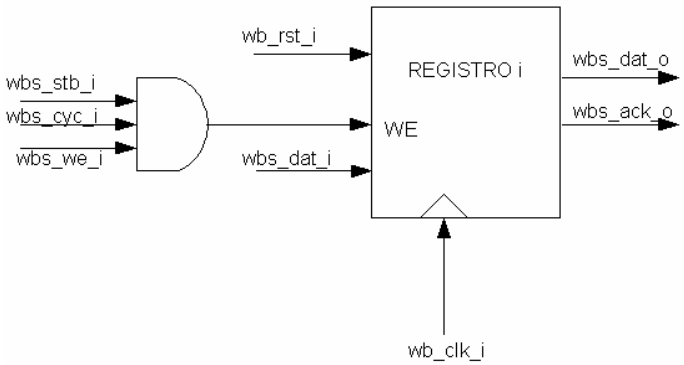

Figura 22. Señales y bloques lógicos del interfaz Wishbone.

Estos 16 registros están mapeados directamente en memoria a través del registro BAR1 de la siguiente forma:

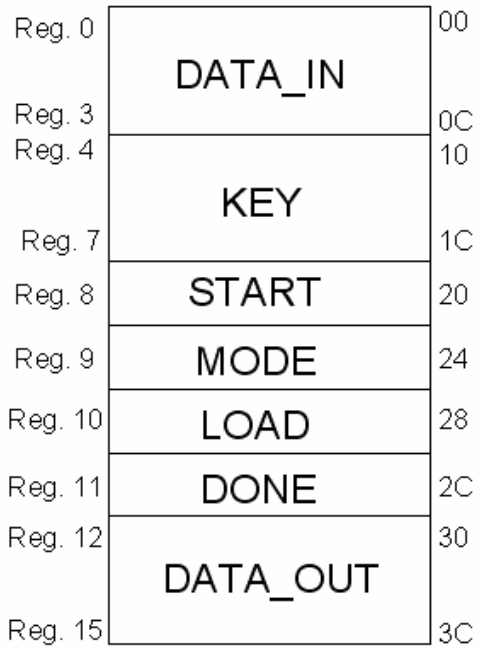

Figura 23. Distribución de la información en el espacio de memoria BAR1.

El usuario para leer o escribir estos registros del modulo "WB_AES" tan solo deberá hacer la respectiva operación en las posiciones de memoria asociadas a estos registros

El IP core "WB_AES" tiene dos formas distintas de funcionamiento en función del método de carga de los registros:

- Modo Software o modo Usuario.- el usuario, a través de la aplicación software:
    1) carga los primeros 64 bits de datos de entrada (registros 0 y 1 del módulo)
    2) y los primeros 64 bits de la clave (registros 4 y 5 del módulo),
    3) pone la señal *load* a alta (último bit del registro 10),
    4) carga los segundos 64 bits del dato de entrada y de la clave,
    5) pone la señal *load* a baja (de nuevo en los registros 0 y 1 para los datos y los registros 4 y 5 para la clave)
    6) introduce un pulso en la señal *start*, poniéndola a alta y a baja consecutivamente (último bit del registro 8)

    Tras estos pasos, el módulo "AES_128_fast" ejecutará la función indicada en los dos bits que definen la señal *mode* (dos últimos bits del registro 9) y tras 13 ciclos y tras activar la señal *done* (último bit del registro 11), escribirá los 128 bits del resultado en los registros 12 a 15.

    Cuando el usuario escoge este modo de cargar los registros a través de la aplicación software, el último bit del registro 9 se pondrá a 0 para que el módulo tenga conocimiento del modo en que se van a cargar sus registros.

- Modo Hardware o de Carga directa de registro.- el usuario tan solo tendrá que:
    1) escribir a través de la aplicación software los 128 bits de datos de entrada (se almacenarán los registros 0 al 4 de nuestro módulo)
    2) y los 128 bits de la clave (se almacenarán directamente en los registros 5 al 7) y
    3) poner a 1 el último bit de la señal *start* (registro 8)

    Tras estos pasos, se pone en funcionamiento el micro-controlador (maquina de estados) citado anteriormente que generará las señales de load y start adecuadas para que el módulo "AES_128_fast" ejecute convenientemente las operaciones indicadas en los dos bits mas significativos de la señal *mode* (registro 9).

    Cuando el usuario escoge este modo de cargar los registros a través de la aplicación software, el último bit del registro 9 se pondrá a 1 para que el módulo tenga conocimiento del modo en que se van a cargar sus registros.

    La máquina de estados es la siguiente:

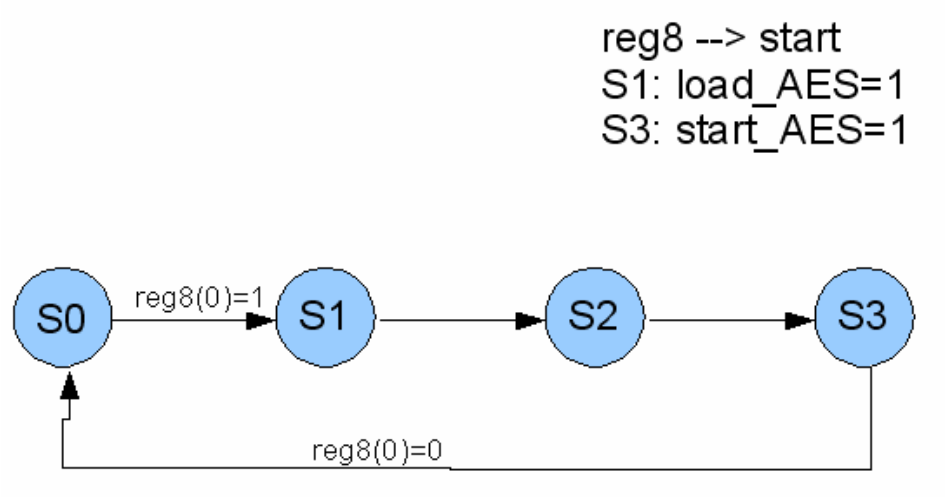

Figura 24. Máquina de estados del modo de ejecución Hardware.

Estado0: estado inicial en el que se permanece hasta que el usuario ponga a '1' la señal de *start* (último bit del registro 8)

Estado1: estado al que se pasa después de que el usuario haya puesto a '1' la señal de *start*. En este estado la señal de *load* se pondrá a '1' para que se carguen los primeros 64 bits del dato de entrada (registros 0 y 1) y de la clave (registros 4 y 5). Paso al Estado2.

Estado2: la señal de load se pondrá a '0' para que se carguen los segundos 64 bits del dato de entrada (registros 2 y 3) y de la clave (registros 6 y 7). Paso al Estado3.

Estado3 y Estado4: la señal *start* de entrada al módulo "AES_128_fast" se pone a '1' y a '0' en estos dos estados consecutivos (así se simula el pulso necesario para que el módulo "AES_128_fast" comience la ejecución del algoritmo).

Módulo funcional: implementación optimizada

Ante los resultados temporales obtenidos con la implementación l módulo funcional anteriormente descrita, en los que se observa que el mayor porcentaje de tiempo está dedicado a la comunicación (lectura y escritura de datos, escritura de la señal *load*...) se planteó la opción de mejorarla utilizando buffers de entrada/salida de 512 bytes. Por lo tanto ahora, el módulo funcional estará trabajando con bloques de datos de 512 bytes, ahorrándose el tener que manejar las señales de *load*, *start*, *done*... para cada procesado de 128 bits, como se hacía anteriormente.

Debido a que el interfaz Wishbone trabaja con un ancho de palabra de 32 bits para transmisión de datos, y a que el módulo "AES_128_fast" necesita ser alimentado con palabras de 64 bits devolviendo resultados de 128 bits, se han utilizado dos memorias distintas para el buffering de los datos:

- Una memoria para el almacenamiento de los datos que van a ser procesados, en la que se escriben datos de 32 bits y de la que se leen palabras de 64
- Una memoria para el almacenamiento de los datos que ya han sido procesados, en la que se escriben palabras de 128 bits y se leen de 32 bits.

La forma de funcionamiento de esta arquitectura es parecida a la anterior. Para almacenar la clave, la señal de *start*, el modo (operación que se quiere que realice el módulo "AES_128_fast") y la señal *done* se utilizarán también 7 registros de 32 bits mapeados directamente en memoria a través del registro BAR1 de la siguiente forma:

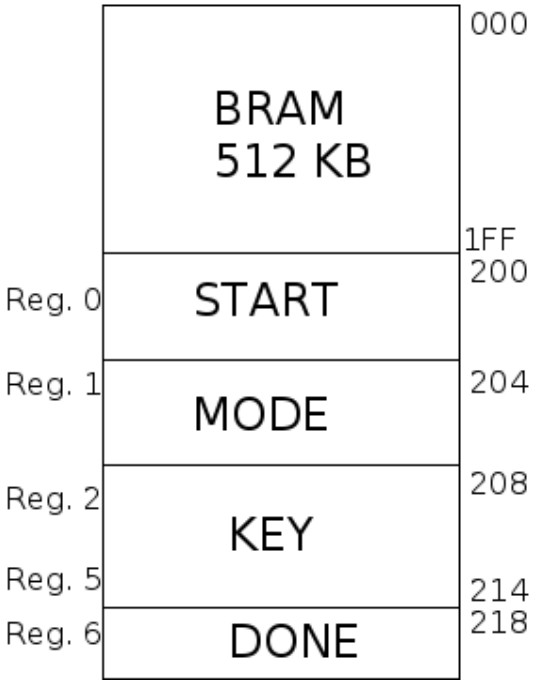

Figura 25. Distribución de la información en el espacio de memoria BAR1 (versión optimizada).

En los primeros 512 bytes estará mapeada la memoria utilizada para los buffers de entrada y salida de datos (ambas comparten espacio de direcciones).

La máquina de estados que actúa como controlador del sistema también ha tenido que ser modificada:

Estado0: estado inicial en el que se permanece hasta que el usuario ponga a ‘1’ la señal de *start* (último bit del registro 0)

Estado1: estado al que se pasa después de que el usuario haya puesto a ‘1’ la señal de *start*. En este estado la señal de *load* se pondrá a ‘1’ para que se carguen los primeros 64 bits del dato de entrada (estos 64 bits se obtienen del buffer de entrada. La dirección a la que se accederá para obtener estos datos viene determinada por el valor de un contador con el que se recorrerá este buffer) y de la clave (registros 2 y 3). Paso al Estado2.

Estado2: la señal de *load* se pondrá a ‘0’ para que se carguen los segundos 64 bits del dato de entrada (obtenidos al igual que en el estado anterior, pero esta vez de la dirección inmediatamente posterior) y de la clave (registros 4 y 5). Paso al Estado3.

Estado3 y Estado4: la señal *start* de entrada al módulo “AES_128_fast” se pone a ‘1’ y a ‘0’ en estos dos estados consecutivos (así se simula el pulso necesario para que el módulo “AES_128_fast” comience la ejecución del algoritmo).

Estado5 y Estado6: estados necesarios para esperar a que se actualice la señal de done. En el Estado6 el contador que hace de índice para leer palabras de 64 bits del buffer de entrada se verá incrementado.

Estado7: estado en el que se espera a que termine el módulo “AES_128_fast” (mientras que done=0 se permanece en este estado).

Estado8: estado en el que se escribe en el buffer de salida el dato de 128 bits que ha procesado el módulo “AES_128_fast”. En este estado se incrementa el contador que hace de índice para escribir palabras de 128 bits (dato procesado) en el buffer de salida. Si ya se han leído procesado todas las palabras de 64 bits del buffer de entrada se pasa al Estado9. En caso contrario se pasa al Estado1.

Estado9: Se pone a '1' la señal done (último bit del registro 6) que reconoce que el bloque de datos de 512 bytes ya ha sido procesado. Se permanecerá en este estado hasta que el usuario ponga a '0' la señal de *start* (último bit del registro 0) momento en el que se pasará al Estado0.

Aquí se representa visualmente la máquina de estados que se acaba de describir:

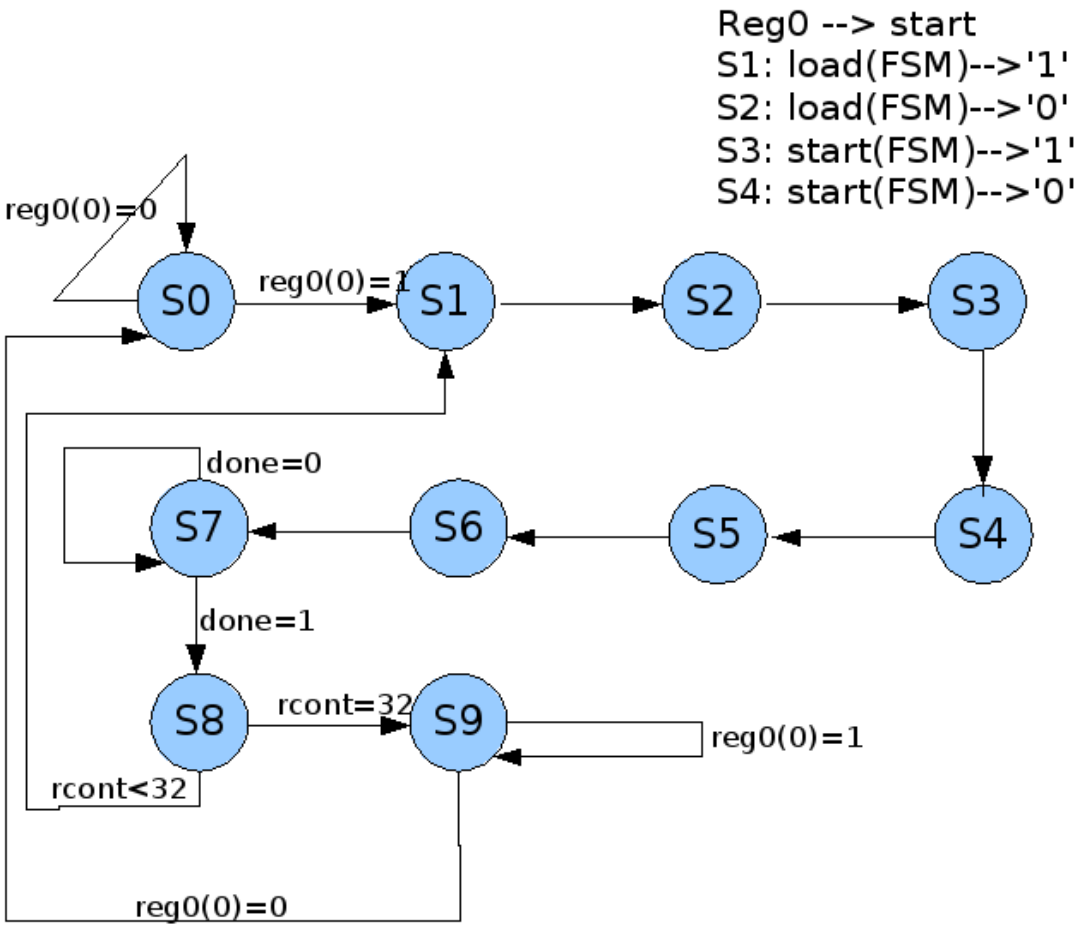

Figura 26. Máquina de estados de la versión optimizada.

Y este será el diagrama de bloques que describe la arquitectura de este módulo funcional optimizado usando dos buffers de entrada y salida.

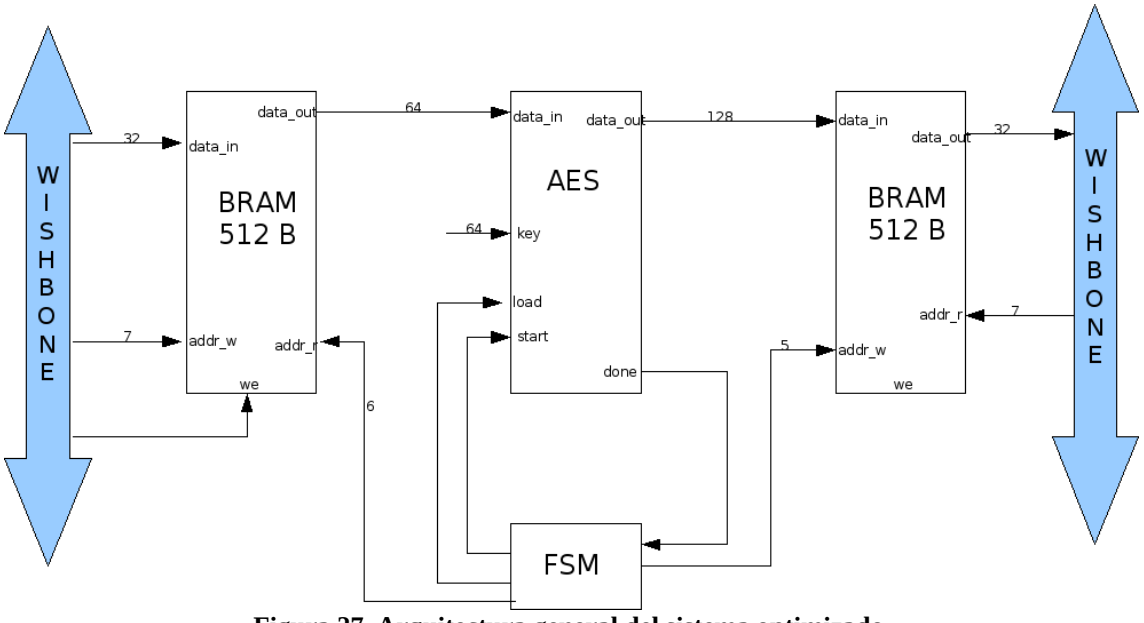

Figura 27. Arquitectura general del sistema optimizado.

6. Conclusiones

Después del desarrollo del proyecto, y a la luz de los resultados obtenidos, llegamos a la conclusión de que el objetivo inicial para el cuál se pensó el sistema es inviable en cuanto a la aceleración de procesamiento intensivo de datos, al menos con los recursos disponibles.

Por un lado, es patente la necesidad de usar mayores bloques de memoria que actúen como buffers en el sistema diseñado, con el fin de poder optimizar las transmisiones de datos en ráfagas, lo cual ahorra una importante cantidad de transacciones de control y petición sobre el bus de sistema escogido, en este caso, el bus PCI. No obstante, este aumento de rendimiento seguiría teniendo una cota máxima, el límite teórico (probablemente inferior en términos prácticos) del bus, estipulado en 133 MB/s. Es decir, una vez resuelto el problema del almacenamiento temporal, debería resolverse el problema del ancho de banda del canal, dirigiéndose bien a implementaciones posteriores del propio bus PCI (que podrían aumentar el ancho de banda hasta por encima de los 500 MB/s usando una longitud de 64 bits para los datos y una frecuencia de 66 MHz de funcionamiento del dispositivo), o bien directamente hacia el uso de otros buses de sistema más rápidos, como el sucesor del PCI, PCI-Express, cada vez más común en los PC de sobremesa y con velocidades de transmisión superiores a su predecesor.

Por otro lado, ha quedado clara la mala elección del algoritmo principal de ejemplo para la aplicación de la idea del proyecto. Con el fin de producir una aceleración clara, sería necesario acudir a algoritmos mucho más específicos, complejos y, sobre todo, con una menor necesidad de alimentación de datos. No parece clara a primera vista una aplicación fuera del ámbito científico o industrial, donde sí es probable encontrar más frecuentemente algoritmos que cumplan estas características. En la informática de usuario la entrada/salida juega un papel muy importante, y dadas las limitaciones en comunicación y las altas exigencias del caudal de datos necesarias para este tipo de sistemas, parece, al menos *a priori*, un obstáculo difícilmente salvable en el contexto actual.

Finalmente, es justo fijarse también en la diferencia de velocidad de cómputo que se plantea. Un procesador de propósito general actual (Pentium IV ó AMD 64) trabaja a una frecuencia superior en alrededor de tres órdenes de magnitud respecto a una FPGA de bajo coste, lo cual supone una dura comparación. Naturalmente, se pueden incrementar las prestaciones de las FPGA destinadas al uso planteado en el proyecto, pero el coste también se incrementa al mismo tiempo. Por ello, y redundando en el punto anterior, es vital una buena elección del algoritmo que se desea implementar para que los cálculos sean notablemente acelerados. Cualidades como paralelización y autoalimentación de datos que puedan, por un lado batir al procesador del sistema y por otro reducir al mínimo las transacciones entre el sistema y la placa externa.

7. Apéndices

a. Índice de figuras

b. Índice de tablas

c. Palabras clave

- Aceleración
- AES
- Bridge
- Bus
- Controlador W32
- FPGA
- ISE
- PCI
- Spartan II-E
- VHDL
- WinDriver
- Wishbone

d. Código fuente de la parte Software

La función cifrar() es donde se produce la mayor parte de comunicación con el driver. Para acceder al código completo de la aplicación y de las funciones de más bajo nivel del driver, ver el CD-ROM de documentación, en la ruta \software.

Primera aproximación (escritura en registros bloque a bloque)

```
int cifrar (WDC_DEVICE_HANDLE *phDev, char* entrada, char* salida, int* key,
            int modo,int oper) {
      int bytes_leidos = 0;
      char* buffer;
      unsigned int* dat_ptr;
      unsigned int* dat_sal_ptr;
      unsigned int done;
      long lCurPos, fileSize;
      int cont = 0;
      clock_t cuentaSegundos;
      clock_t contador2;

      FILE *fe, *fs;

      contador2 = iniciarCuenta();

      fe = fopen(entrada, "rb");    // r: read, b: binary

      if (!fe) {
            printf ("Error al abrir el fichero de entrada\n\n");
            return -1;
      }

      lCurPos = ftell ( fe );
      fseek ( fe, 0, 2 );
      fileSize = ftell ( fe );
      fseek ( fe, lCurPos, 0 );

      buffer = (char*) malloc (fileSize);

      fs = fopen (salida, "wb");

      if (!fs) {
            printf ("Error al abrir el fichero de salida\n\n");
            return -1;
      }

      bytes_leidos = fread (buffer, 1, fileSize, fe);
      if (bytes_leidos != fileSize){
            printf ("Error al leer el archivo de entrada\n");
            return -1;
      }
      printf(" Leidos %ld bytes\n",fileSize);

      dat_ptr = buffer;
      dat_sal_ptr = buffer;

      //Escribimos la clave en el dispositivo
      WDC_WriteAddr32(*phDev, 1, AES_KEY_0 , key[0]);
      WDC_WriteAddr32(*phDev, 1, AES_KEY_1 , key[1]);
```

```
WDC_WriteAddr32(*phDev, 1, AES_KEY_2 , key[2]);
WDC_WriteAddr32(*phDev, 1, AES_KEY_3 , key[3]);
// Seleccionamos modo hardware (1) o software
if (modo == 1)
      switch(oper){
            case 0:
                  WDC_WriteAddr32(*phDev, 1, AES_MODE, 0x00000001);
            break;
            case 1:
                  WDC_WriteAddr32(*phDev, 1, AES_MODE, 0x40000001);
            break;
            case 2:
                  WDC_WriteAddr32(*phDev, 1, AES_MODE, 0x80000001);
            break;
      default:
                  WDC_WriteAddr32(*phDev, 1, AES_MODE, 0xC0000001);
      }
else
      switch(oper){
            case 0:
                  WDC_WriteAddr32(*phDev, 1, AES_MODE, 0x00000000);
            break;
            case 1:
                  WDC_WriteAddr32(*phDev, 1, AES_MODE, 0x40000000);
            break;
            case 2:
                  WDC_WriteAddr32(*phDev, 1, AES_MODE, 0x80000000);
            break;
      default:
                  WDC_WriteAddr32(*phDev, 1, AES_MODE, 0xC0000000);
      }

//Aseguramos que start y load esten a 0
WDC_WriteAddr32(*phDev, 1,  AES_START, 0x0);
WDC_WriteAddr32(*phDev, 1, AES_LOAD, 0x0);

cuentaSegundos = iniciarCuenta();
while (cont <= (fileSize-16)){
      //Control SW (2) o HW (1)
      if (modo == 1){
            WDC_WriteAddr32(*phDev, 1, AES_DAT_IN_0 , *dat_ptr);

            dat_ptr++;
            WDC_WriteAddr32(*phDev, 1, AES_DAT_IN_1 , *dat_ptr);

            dat_ptr++;
            WDC_WriteAddr32(*phDev, 1, AES_DAT_IN_2 , *dat_ptr);

            dat_ptr++;
            WDC_WriteAddr32(*phDev, 1, AES_DAT_IN_3 , *dat_ptr);

            dat_ptr++;
            WDC_WriteAddr32(*phDev, 1,  AES_START, 0xffffffff);
      }
      else{
            WDC_WriteAddr32(*phDev, 1, AES_DAT_IN_0 , *dat_ptr);

            dat_ptr++;
            WDC_WriteAddr32(*phDev, 1, AES_DAT_IN_1 , *dat_ptr);

            dat_ptr++;
```

```
                WDC_WriteAddr32(*phDev, 1, AES_KEY_0 , key[0]);
                WDC_WriteAddr32(*phDev, 1, AES_KEY_1 , key[1]);

                WDC_WriteAddr32(*phDev, 1, AES_LOAD, 0xffffffff);

                WDC_WriteAddr32(*phDev, 1, AES_DAT_IN_0 , *dat_ptr);

                dat_ptr++;
                WDC_WriteAddr32(*phDev, 1, AES_DAT_IN_1 , *dat_ptr);

                dat_ptr++;
                WDC_WriteAddr32(*phDev, 1, AES_KEY_0 , key[2]);
                WDC_WriteAddr32(*phDev, 1, AES_KEY_1 , key[3]);


                WDC_WriteAddr32(*phDev, 1, AES_LOAD, 0x0);

                WDC_WriteAddr32(*phDev, 1,  AES_START, 0xffffffff);
                WDC_WriteAddr32(*phDev, 1,  AES_START, 0x0);
        }
        // Espera activa hasta que acabe el dispositivo
        do{
                WDC_ReadAddr32(*phDev, 1, AES_DONE, &done);
        }while (done == 0x0);

        WDC_ReadAddr32(*phDev, 1, AES_DAT_OUT_0 , dat_sal_ptr);

        dat_sal_ptr++;
        WDC_ReadAddr32(*phDev, 1, AES_DAT_OUT_1 , dat_sal_ptr);

        dat_sal_ptr++;
        WDC_ReadAddr32(*phDev, 1, AES_DAT_OUT_2 , dat_sal_ptr);

        dat_sal_ptr++;
        WDC_ReadAddr32(*phDev, 1, AES_DAT_OUT_3 , dat_sal_ptr);

        dat_sal_ptr++;

        WDC_WriteAddr32(*phDev, 1,  AES_START, 0x0);

        cont += 16;

    }
    printf("Codificacion terminada en %f segundos\n",
        detenerCuenta(cuentaSegundos));

    fwrite(buffer, 1, fileSize, fs);
    printf ("\n *** Guardando %s\n", salida);

    fclose (fe);
    fclose (fs);

    printf("Codificacion (+accesos a disco) terminada en %f segundos\n",
        detenerCuenta(contador2));
    printf ("\n *** Cifrado correctamente %s\n\n", salida);

    return 1;
}
```

Segunda aproximación (escrituras en ráfagas a memoria)

```
int cifrar (WDC_DEVICE_HANDLE *phDev, char* entrada, char* salida,
            int* key, int oper) {
      int bytes_leidos = 0;
      char* buffer;
      unsigned int* dat_ptr;


      unsigned int done;

      long lCurPos, fileSize;

      int cont = 0;
      clock_t cuentaSegundos;
      clock_t contador2;

      FILE *fe, *fs;

      contador2 = iniciarCuenta();

      fe = fopen(entrada, "rb");    // r: read, b: binary

      if (!fe) {
            printf ("Error al abrir el fichero de entrada\n\n");
            return -1;
      }
      //Comprobamos el tamaño del archivo
      lCurPos = ftell ( fe );
      fseek ( fe, 0, 2 );
      fileSize = ftell ( fe );
      //dejamos el apuntador al comienzo del archivo
      fseek ( fe, lCurPos, 0 );

      buffer = (char*) malloc (fileSize);


      fs = fopen (salida, "wb");

      if (!fs) {
            printf ("Error al abrir el fichero de salida\n\n");
            return -1;
      }

      bytes_leidos = fread (buffer, 1, fileSize, fe);
      if (bytes_leidos != fileSize){
            printf ("Error al leer el archivo de entrada\n");
            return -1;
      }
      printf(" Leidos %ld bytes\n",fileSize);


      dat_ptr = buffer;


      //Escribimos la clave en el dispositivo
      WDC_WriteAddr32(*phDev, 1, AES_KEY_0 , key[0]);
      WDC_WriteAddr32(*phDev, 1, AES_KEY_1 , key[1]);
      WDC_WriteAddr32(*phDev, 1, AES_KEY_2 , key[2]);
      WDC_WriteAddr32(*phDev, 1, AES_KEY_3 , key[3]);
```

```
	// Seleccionamos la operacion del modulo
	switch(oper){
		case 0:
			WDC_WriteAddr32(*phDev, 1, AES_MODE, 0x00000000);
		break;
		case 1:
			WDC_WriteAddr32(*phDev, 1, AES_MODE, 0x00000001);
		break;
		case 2:
			WDC_WriteAddr32(*phDev, 1, AES_MODE, 0x00000002);
		break;
	  default:
			WDC_WriteAddr32(*phDev, 1, AES_MODE, 0x00000003);
	}

	//Aseguramos que start este a 0
	WDC_WriteAddr32(*phDev, 1,  AES_START, 0x0);

	cuentaSegundos = iniciarCuenta();
	while (cont <= (fileSize-512)){
		WDC_WriteAddrBlock(*phDev, 1, AES_DAT_IN, AES_BLOCK_SIZE,
			dat_ptr, WDC_MODE_32, WDC_ADDR_RW_DEFAULT);
		WDC_WriteAddr32(*phDev, 1,  AES_START, 0x1);
		// Espera activa hasta que acabe el dispositivo
		do{
			WDC_ReadAddr32(*phDev, 1, AES_DONE, &done);
		}while (done == 0x0);

		WDC_ReadAddrBlock(*phDev, 1, AES_DAT_IN, AES_BLOCK_SIZE,
			dat_ptr, WDC_MODE_32, WDC_ADDR_RW_DEFAULT);

		WDC_WriteAddr32(*phDev, 1,  AES_START, 0x0);

		dat_ptr += 128;

		cont += 512;

	}
	printf("Codificacion terminada en %f segundos\n",
		detenerCuenta(cuentaSegundos));

	fwrite(buffer, 1, fileSize, fs);
	printf ("\n *** Guardando %s\n", salida);

	fclose (fe);
	fclose (fs);

	printf("Codificacion (+accesos a disco) terminada en %f
		segundos\n", detenerCuenta(contador2));
	printf ("\n *** Cifrado correctamente %s\n\n", salida);

	return 1;
}
```

e. Código fuente de la parte Hardware

Aquí se listan el módulo de funcionalidad y las dos implementaciones del módulo de comunicación. Para acceder a los listados completos del resto de componentes del proyecto y a los proyectos en sí mismos, sintetizables, ver el CD-ROM de documentación en las rutas \hardware\PCI_AES_v1.0 y \hardware\PCI_AES_v1.1, respectivamente.

Módulo de funcionalidad

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.aes_package.all;

entity aes128_fast is
port(
     clk       : in std_logic;
     reset     : in std_logic;
     start     : in std_logic; -- to initiate the encryption/decryption process after loading
     mode      : in std_logic_vector(1 downto 0); -- to select encryption or decryption
     load      : in std_logic; -- to load the input and keys.has to
     key       : in std_logic_vector(63 downto 0);
     data_in   : in std_logic_vector(63 downto 0);
     data_out  : out std_logic_vector(127 downto 0);
     done      : out std_logic
    );

end aes128_fast;

architecture mapping of aes128_fast is

signal data_in_reg0: state_array_type;
signal data_in_reg1: state_array_type;
signal data_in_reg2: state_array_type;
signal data_in_reg3: state_array_type;
signal key_reg0: state_array_type;
signal key_reg1: state_array_type;
signal key_reg2: state_array_type;
signal key_reg3: state_array_type;

signal load_d1 : std_logic;
signal start_d1: std_logic;
signal start_d2: std_logic;
signal round_cnt: integer range 0 to 15;
signal flag_cnt: std_logic;
signal done_d1 : std_logic;
signal done_d2 : std_logic;

begin
```

```
-- Loading the data and keys
process(clk,reset)
begin
  if(reset = '1') then
    key_reg0 <= (others =>(others => '0'));
    key_reg1 <= (others =>(others => '0'));
    key_reg2 <= (others =>(others => '0'));
    key_reg3 <= (others =>(others => '0'));
    data_in_reg0 <= (others =>(others => '0'));
    data_in_reg1 <= (others =>(others => '0'));
    data_in_reg2 <= (others =>(others => '0'));
    data_in_reg3 <= (others =>(others => '0'));
  elsif rising_edge(clk) then
    if(load = '1' and load_d1 = '0') then
      key_reg0     <= (key(63 downto 56),key(55 downto 48),key(47 downto 40),key(39
                    downto 32));
      key_reg1     <= (key(31 downto 24),key(23 downto 16),key(15 downto 8),key(7
                    downto 0));
      data_in_reg0 <= (data_in(63 downto 56),data_in(55 downto 48),data_in(47 downto
                    40),data_in(39 downto 32));
      data_in_reg1 <= (data_in(31 downto 24),data_in(23 downto 16),data_in(15 downto
                    8),data_in(7 downto 0));
    elsif(load_d1 = '1' and load = '0') then
      key_reg2     <= (key(63 downto 56),key(55 downto 48),key(47 downto 40),key(39
                    downto 32));
      key_reg3     <= (key(31 downto 24),key(23 downto 16),key(15 downto 8),key(7
                    downto 0));
      data_in_reg2 <= (data_in(63 downto 56),data_in(55 downto 48),data_in(47 downto
                    40),data_in(39 downto 32));
      data_in_reg3 <= (data_in(31 downto 24),data_in(23 downto 16),data_in(15 downto
                    8),data_in(7 downto 0));
    end if;
  end if;
end process;

done <= done_d2;

-- Registering start and load
process(clk,reset)
begin
  if(reset = '1') then
    load_d1  <= '0';
    start_d1 <= '0';
    start_d2 <= '0';
  elsif rising_edge(clk) then
    load_d1  <= load;
    start_d1 <= start;
    start_d2 <= start_d1;
  end if;
```

```
end process;

-- Initiator process
process(clk,reset)
begin
  if(reset = '1') then
    round_cnt <= 0;
    flag_cnt <= '0';
  elsif rising_edge(clk) then
    if((start_d2 = '1' and start_d1 = '0') or flag_cnt = '1') then
      if(round_cnt < 11) then
        round_cnt <= round_cnt + 1;
        flag_cnt <= '1';
      else
        round_cnt <= 0;
        flag_cnt <= '0';
      end if;
    end if;
  end if;
end process;

-- Completion signalling process
process(clk,reset)
begin
  if(reset = '1') then
    done_d1 <= '0';
    done_d2 <= '0';
  elsif rising_edge(clk) then
    if(start_d2 = '1' and start_d1 = '0') then
      done_d1 <= '0';
      done_d2 <= '0';
    elsif(round_cnt = 10) then
      done_d1 <= '1';
    end if;
    done_d2 <= done_d1;
  end if;
end process;

-- Output assignment process
process(clk,reset)
begin
  if(reset= '1') then
    data_out <= (others => '0');
  elsif rising_edge(clk) then
    if(done_d1 = '1' and done_d2 = '0') then
       case mode is
                    when "00" =>
                     data_out <=  (data_in_reg0(0) xor key_reg0(0))&
                          (data_in_reg0(1) xor key_reg0(1))&
                          (data_in_reg0(2) xor key_reg0(2))&
```

```
        (data_in_reg0(3) xor key_reg0(3))&
        (data_in_reg1(0) xor key_reg1(0))&
        (data_in_reg1(1) xor key_reg1(1))&
        (data_in_reg1(2) xor key_reg1(2))&
        (data_in_reg1(3) xor key_reg1(3))&
        (data_in_reg2(0) xor key_reg2(0))&
        (data_in_reg2(1) xor key_reg2(1))&
        (data_in_reg2(2) xor key_reg2(2))&
        (data_in_reg2(3) xor key_reg2(3))&
        (data_in_reg3(0) xor key_reg3(0))&
        (data_in_reg3(1) xor key_reg3(1))&
        (data_in_reg3(2) xor key_reg3(2))&
        (data_in_reg3(3) xor key_reg3(3));
    when "01" =>
      data_out <=  (data_in_reg0(0) + key_reg0(0))&
        (data_in_reg0(1) + key_reg0(1))&
        (data_in_reg0(2) + key_reg0(2))&
        (data_in_reg0(3) + key_reg0(3))&
        (data_in_reg1(0) + key_reg1(0))&
        (data_in_reg1(1) + key_reg1(1))&
        (data_in_reg1(2) + key_reg1(2))&
        (data_in_reg1(3) + key_reg1(3))&
        (data_in_reg2(0) + key_reg2(0))&
        (data_in_reg2(1) + key_reg2(1))&
        (data_in_reg2(2) + key_reg2(2))&
        (data_in_reg2(3) + key_reg2(3))&
        (data_in_reg3(0) + key_reg3(0))&
        (data_in_reg3(1) + key_reg3(1))&
        (data_in_reg3(2) + key_reg3(2))&
        (data_in_reg3(3) + key_reg3(3));
    when "10" =>
      data_out <=  (data_in_reg0(0) - key_reg0(0))&
        (data_in_reg0(1) - key_reg0(1))&
        (data_in_reg0(2) - key_reg0(2))&
        (data_in_reg0(3) - key_reg0(3))&
        (data_in_reg1(0) - key_reg1(0))&
        (data_in_reg1(1) - key_reg1(1))&
        (data_in_reg1(2) - key_reg1(2))&
        (data_in_reg1(3) - key_reg1(3))&
        (data_in_reg2(0) - key_reg2(0))&
        (data_in_reg2(1) - key_reg2(1))&
        (data_in_reg2(2) - key_reg2(2))&
        (data_in_reg2(3) - key_reg2(3))&
        (data_in_reg3(0) - key_reg3(0))&
        (data_in_reg3(1) - key_reg3(1))&
        (data_in_reg3(2) - key_reg3(2))&
        (data_in_reg3(3) - key_reg3(3));
    when "11" =>
      if (key_reg0(0) = "00000000") then
```

```
                        data_out <=
                data_in_reg0(1)&data_in_reg0(2)&data_in_reg0(3)&
                data_in_reg1(0)&data_in_reg1(1)&data_in_reg1(2)&data_in_reg1(3)&
                data_in_reg2(0)&data_in_reg2(1)&data_in_reg2(2)&data_in_reg2(3)&
                data_in_reg3(0)&data_in_reg3(1)&data_in_reg3(2)&data_in_reg3(3)&
                data_in_reg0(0);
                  else
                        data_out <=
                data_in_reg3(3)&data_in_reg0(0)&data_in_reg0(1)&data_in_reg0(2)&
                data_in_reg0(3)&data_in_reg1(0)&data_in_reg1(1)&data_in_reg1(2)&
                data_in_reg1(3)&data_in_reg2(0)&data_in_reg2(1)&data_in_reg2(2)&
                data_in_reg2(3)&data_in_reg3(0)&data_in_reg3(1)&data_in_reg3(2);

                    end if;
                when others =>
                        null;
                end case;
   end if;
  end if;
end process;

end mapping;
```

Módulo de comunicación v1.0 (con registros)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity WB_AES is
        PORT(
                wb_clk_i : IN std_logic;
                wb_rst_i : IN std_logic;
                wbm_dat_i : IN std_logic_vector(31 downto 0);
                wbm_ack_i : IN std_logic;
                wbm_err_i : IN std_logic;
                wbm_rty_i : IN std_logic;
                wbs_cyc_i : IN std_logic;
                wbs_stb_i : IN std_logic;
                wbs_sel_i : IN std_logic_vector(3 downto 0);
                wbs_we_i : IN std_logic;
                wbs_adr_i : IN std_logic_vector(31 downto 0);
                wbs_dat_i : IN std_logic_vector(31 downto 0);
                wbs_cab_i : IN std_logic;
                wbm_cyc_o : OUT std_logic;
                wbm_stb_o : OUT std_logic;
                wbm_sel_o : OUT std_logic_vector(3 downto 0);
                wbm_we_o : OUT std_logic;
                wbm_adr_o : OUT std_logic_vector(31 downto 0);
                wbm_dat_o : OUT std_logic_vector(31 downto 0);
                wbm_cab_o : OUT std_logic;
                wbs_dat_o : OUT std_logic_vector(31 downto 0);
                wbs_ack_o : OUT std_logic;
                wbs_err_o : OUT std_logic;
                wbs_rty_o : OUT std_logic;
                led : OUT std_logic;
                led1 : OUT std_logic_vector(7 downto 0)
                );
end WB_AES;

architecture Behavioral of WB_AES is

component aes128_fast
port(
    clk      : in std_logic;
    reset    : in std_logic;
```

```
    start     : in std_logic; -- to initiate the encryption/decryption process after loading
    mode      : in std_logic_vector(1 downto 0); -- to select encryption or decryption
    load      : in std_logic; -- to load the input and keys.has to
    key       : in std_logic_vector(63 downto 0);
    data_in   : in std_logic_vector(63 downto 0);
    data_out  : out std_logic_vector(127 downto 0);
    done      : out std_logic
   );

end component;

signal reg : std_logic_vector(7 downto 0);

signal reg_0 : std_logic_vector(31 downto 0);
signal reg_1 : std_logic_vector(31 downto 0);
signal reg_2 : std_logic_vector(31 downto 0);
signal reg_3 : std_logic_vector(31 downto 0);
signal reg_4 : std_logic_vector(31 downto 0);
signal reg_5 : std_logic_vector(31 downto 0);
signal reg_6 : std_logic_vector(31 downto 0);
signal reg_7 : std_logic_vector(31 downto 0);
signal reg_8 : std_logic_vector(31 downto 0);
signal reg_9 : std_logic_vector(31 downto 0);
signal reg_10 : std_logic_vector(31 downto 0);
signal reg_11 : std_logic_vector(31 downto 0);
signal reg_12 : std_logic_vector(31 downto 0);
signal reg_13 : std_logic_vector(31 downto 0);
signal reg_14 : std_logic_vector(31 downto 0);
signal reg_15 : std_logic_vector(31 downto 0);
signal DAT_O_s : std_logic_vector(31 downto 0);

signal data_out_s : std_logic_vector(127 downto 0);
signal s_key : std_logic_vector(63 downto 0);
signal s_data_in : std_logic_vector(63 downto 0);
signal done_s : std_logic;
signal fsm_load : std_logic;
signal fsm_start : std_logic;

signal s_load : std_logic;
signal s_start : std_logic;
signal s_mode : std_logic_vector(1 downto 0);


signal estado : std_logic_vector(2 downto 0);

signal wbs_dat_i_s : std_logic_vector(31 downto 0);

begin
```

```
s_mode <= reg_9(31)&reg_9(30);

aes : aes128_fast PORT MAP(
        clk => wb_clk_i,
        reset => wb_rst_i,
        start => s_start,
        mode => s_mode,
        load => s_load,
        key => s_key,
        data_in => s_data_in,
        data_out => data_out_s,
        done => done_s
);

process (wb_clk_i,wb_rst_i)

begin
        if (wb_clk_i'EVENT AND wb_clk_i='1') then
                if (wb_rst_i = '1') then
                        reg <= "00000000";
                else
                        if ((wbs_stb_i AND wbs_cyc_i) = '1') then
                                reg <= wbs_dat_i_s(7 downto 0);
                        end if;
                end if;
        end if;
end process;

process (wb_clk_i,wb_rst_i)
begin
        if (wb_clk_i'EVENT AND wb_clk_i='1') then
                        if (wb_rst_i = '1') then
                                reg_0 <= "00000000000000000000000000000000";
                                reg_1 <= "00000000000000000000000000000000";
                                reg_2 <= "00000000000000000000000000000000";
                                reg_3 <= "00000000000000000000000000000000";
                                reg_4 <= "00000000000000000000000000000000";
                                reg_5 <= "00000000000000000000000000000000";
                                reg_6 <= "00000000000000000000000000000000";
                                reg_7 <= "00000000000000000000000000000000";
                                reg_8 <= "00000000000000000000000000000000";
                                reg_9 <= "00000000000000000000000000000000";
                                reg_10 <= "00000000000000000000000000000000";
                                reg_11 <= "00000000000000000000000000000000";
                                reg_12 <= "00000000000000000000000000000000";
                                reg_13 <= "00000000000000000000000000000000";
                                reg_14 <= "00000000000000000000000000000000";
                                reg_15 <= "00000000000000000000000000000000";
                        else
                                -- ciclo de escritura
```

```
                                        if (wbs_stb_i = '1' AND wbs_cyc_i = '1' AND
                                                wbs_we_i = '1') then
                                                -- decodificacion de la direccion
                                                case wbs_adr_i(5 downto 2) is
                                                        when "0000" =>
                                                                reg_0 <= wbs_dat_i_s;
                                                        when "0001" =>
                                                                reg_1 <= wbs_dat_i_s;
                                                        when "0010" =>
                                                                reg_2 <= wbs_dat_i_s;
                                                        when "0011" =>
                                                                reg_3 <= wbs_dat_i_s;
                                                        when "0100" =>
                                                                reg_4 <= wbs_dat_i_s;
                                                        when "0101" =>
                                                                reg_5 <= wbs_dat_i_s;
                                                        when "0110" =>
                                                                reg_6 <= wbs_dat_i_s;
                                                        when "0111" =>
                                                                reg_7 <= wbs_dat_i_s;
                                                        when "1000" =>
                                                                reg_8 <= wbs_dat_i_s;
                                                        when "1001" =>
                                                                reg_9 <= wbs_dat_i_s;
                                                        when "1010" =>
                                                                reg_10 <= wbs_dat_i_s;
                                                        when "1011" =>
                                                                reg_11 <= wbs_dat_i_s;
                                                        when "1100" =>
                                                                reg_12 <= wbs_dat_i_s;
                                                        when "1101" =>
                                                                reg_13 <= wbs_dat_i_s;
                                                        when "1110" =>
                                                                reg_14 <= wbs_dat_i_s;
                                                        when "1111" =>
                                                                reg_15 <= wbs_dat_i_s;
                                                        when others => null;
                                                end case;
                                        end if;
                                end if;
                        end if;
        end process;

        REG_READ_PROC : process(wbs_adr_i, reg_0, reg_1, reg_2, reg_3, reg_4,
reg_5, reg_6, reg_7, reg_8, reg_9, reg_10, reg_11, reg_12, reg_13, reg_14, reg_15 ) is
  begin
                case wbs_adr_i(5 downto 2) is
                        when "0000" =>
                                DAT_O_s <= reg_0;
                        when "0001" =>
```

```
                        DAT_O_s <= reg_1;
                    when "0010" =>
                        DAT_O_s <= reg_2;
                    when "0011" =>
                        DAT_O_s <= reg_3;
                    when "0100" =>
                        DAT_O_s <= reg_4;
                    when "0101" =>
                        DAT_O_s <= reg_5;
                    when "0110" =>
                        DAT_O_s <= reg_6;
                    when "0111" =>
                        DAT_O_s <= reg_7;
                    when "1000" =>
                        DAT_O_s <= reg_8;
                    when "1001" =>
                        DAT_O_s <= reg_9;
                    when "1010" =>
                        DAT_O_s <= reg_10;
                    when "1011" =>
                        DAT_O_s <= "00000000000000000000000000000000"&done_s;
                    when "1100" =>
                        DAT_O_s <= data_out_s(127 downto 96);
                    when "1101" =>
                        DAT_O_s <= data_out_s(95 downto 64);
                    when "1110" =>
                        DAT_O_s <= data_out_s(63 downto 32);
                    when "1111" =>
                        DAT_O_s <= data_out_s(31 downto 0);
                    when others => null;
                end case;
end process REG_READ_PROC;

        FSM : process(wb_clk_i) is
        begin
                if wb_clk_i'event and wb_clk_i = '1' then
                        if wb_rst_i = '1' then
                                estado <= "000";
                        else
                                case estado is
                                        when "000" =>--estado inicial
                                                if (reg_8(0) = '1') then
                                                        estado <= "001";
                                                end if;
                                        when "001" => estado <= "010"; --load a '1'
                                        when "010" => estado <= "011"; --load a '0'
                                        when "011" => -- start a '1'
                                                estado <= "100";
                                        when "100" =>-- start a '0'
                                                if (reg_8(0) = '0') then
```

```
                                                estado <= "000";
                                            end if;
                                        when others => null;
                                    end case;
                                end if;
                        end if;
                end process FSM;
                fsm_load <= estado(0) and (not estado(1)) and (not estado(2));
                fsm_start <= estado(1) and estado(0) and (not estado(2));

        Select_in_key : process(estado,reg_0,reg_1,reg_2,reg_3,reg_4,reg_5, reg_6,reg_7) is
                begin
                        if (reg_9(0) = '0') then
                                s_data_in <= reg_0&reg_1;
                                s_key <= reg_4&reg_5;
                                s_load <= reg_10(0);
                                s_start <= reg_8(0);
                        else
                                s_load <= fsm_load;
                                s_start <= fsm_start;
                                case estado is
                                when "000" =>
                                        s_data_in <= reg_0&reg_1;
                                        s_key <= reg_4&reg_5;
                                when "001" =>
                                        s_data_in <= reg_0&reg_1;
                                        s_key <= reg_4&reg_5;
                                when "010" =>
                                        s_data_in <= reg_2&reg_3;
                                        s_key <= reg_6&reg_7;
                                when "011" =>
                                        s_data_in <= reg_2&reg_3;
                                        s_key <= reg_6&reg_7;
                                when others =>
                                        s_data_in <= reg_2&reg_3;
                                        s_key <= reg_6&reg_7;
                                end case;
                        end if;
                end process Select_in_key;



                wbs_ack_o <= (wbs_stb_i AND wbs_cyc_i);
                wbs_err_o <= '0';
                wbs_rty_o <= '0';
                wbs_dat_o <= DAT_O_s;
                led1 <= reg;

                -- las señales del WB no se usan (las fijamos a '0')
                wbm_cyc_o <= '0';
```

```
        wbm_stb_o <= '0';
        wbm_sel_o <= "0000";
        wbm_we_o <= '0';
        wbm_adr_o <= "00000000000000000000000000000000";
        wbm_dat_o <= "00000000000000000000000000000000";
        wbm_cab_o <= '0';
        led <= '0';


        --Transformacion de Little Endian a Big Endian
        wbs_dat_i_s <= wbs_dat_i;

end Behavioral;
```

Módulo de comunicación v1.1 (con memoria)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity WB_AES is
        PORT(
                wb_clk_i : IN std_logic;
                wb_rst_i : IN std_logic;
                wbm_dat_i : IN std_logic_vector(31 downto 0);
                wbm_ack_i : IN std_logic;
                wbm_err_i : IN std_logic;
                wbm_rty_i : IN std_logic;
                wbs_cyc_i : IN std_logic;
                wbs_stb_i : IN std_logic;
                wbs_sel_i : IN std_logic_vector(3 downto 0);
                wbs_we_i : IN std_logic;
                wbs_adr_i : IN std_logic_vector(31 downto 0);
                wbs_dat_i : IN std_logic_vector(31 downto 0);
                wbs_cab_i : IN std_logic;
                wbm_cyc_o : OUT std_logic;
                wbm_stb_o : OUT std_logic;
                wbm_sel_o : OUT std_logic_vector(3 downto 0);
                wbm_we_o : OUT std_logic;
                wbm_adr_o : OUT std_logic_vector(31 downto 0);
                wbm_dat_o : OUT std_logic_vector(31 downto 0);
                wbm_cab_o : OUT std_logic;
                wbs_dat_o : OUT std_logic_vector(31 downto 0);
                wbs_ack_o : OUT std_logic;
                wbs_err_o : OUT std_logic;
                wbs_rty_o : OUT std_logic;
                led : OUT std_logic;
                led1 : OUT std_logic_vector(7 downto 0)
                );
end WB_AES;

architecture Behavioral of WB_AES is

component aes128_fast
port(
    clk       : in std_logic;
    reset     : in std_logic;
```

```
    start     : in std_logic; -- to initiate the encryption/decryption process after loading
    mode      : in std_logic_vector(1 downto 0); -- to select encryption or decryption
    load      : in std_logic; -- to load the input and keys.has to
    key       : in std_logic_vector(63 downto 0);
    data_in   : in std_logic_vector(63 downto 0);
    data_out  : out std_logic_vector(127 downto 0);
    done      : out std_logic
   );

end component;

COMPONENT RAM_128_32
        PORT(
                clk : IN std_logic;
                dat_in : IN std_logic_vector(127 downto 0);
                w_adr : IN std_logic_vector(4 downto 0);
                r_adr : IN std_logic_vector(6 downto 0);
                we : IN std_logic;
                dat_out : OUT std_logic_vector(31 downto 0)
                );
END COMPONENT;

COMPONENT RAM_32_64
        PORT(
                clk : IN std_logic;
                dat_in : IN std_logic_vector(31 downto 0);
                w_adr : IN std_logic_vector(6 downto 0);
                r_adr : IN std_logic_vector(5 downto 0);
                we : IN std_logic;
                dat_out : OUT std_logic_vector(63 downto 0)
                );
END COMPONENT;


signal reg : std_logic_vector(7 downto 0);


signal DAT_O_s : std_logic_vector(31 downto 0);

signal data_out_s : std_logic_vector(127 downto 0);
signal s_key : std_logic_vector(63 downto 0);
signal s_data_in : std_logic_vector(63 downto 0);
signal done_s : std_logic;
signal fsm_load : std_logic;
signal fsm_start : std_logic;


signal s_mode : std_logic_vector(1 downto 0);
```

```
signal estado : std_logic_vector(3 downto 0);

signal wbs_dat_i_s : std_logic_vector(31 downto 0);

--señales RAM_32_64
signal we_RAM_32_64 : std_logic;
signal r_cont : std_logic_vector(4 downto 0);

--señales RAM_128_32
signal we_RAM_128_32 : std_logic;
signal sal_32 : std_logic_vector(31 downto 0);
signal w_cont : std_logic_vector(4 downto 0);

signal reg_start : std_logic_vector(31 downto 0);
signal reg_mode : std_logic_vector(31 downto 0);
signal reg_key0 : std_logic_vector(31 downto 0);
signal reg_key1 : std_logic_vector(31 downto 0);
signal reg_key2 : std_logic_vector(31 downto 0);
signal reg_key3 : std_logic_vector(31 downto 0);

signal DONE : std_logic;
--signal r_CE : std_logic;
--signal w_CE : std_logic;


signal s_r_adr : std_logic_vector(6 downto 0);
signal s_w_adr : std_logic_vector(6 downto 0);

signal s_r_cont : std_logic_vector(5 downto 0);
begin


        s_mode <=  reg_mode(1)& reg_mode(0);

        aes : aes128_fast PORT MAP(
                clk => wb_clk_i,
                reset => wb_rst_i,
                start => fsm_start,
                mode => s_mode,
                load => fsm_load,
                key => s_key,
                data_in => s_data_in,
                data_out => data_out_s,
                done => done_s
        );
        s_r_adr <= wbs_adr_i(8 downto 2);
        Inst_RAM_128_32: RAM_128_32 PORT MAP(
                clk => wb_clk_i,
                dat_in => data_out_s,
                dat_out => sal_32,
```

```
                w_adr => w_cont,
                r_adr => s_r_adr,
                we => we_RAM_128_32
        );

        we_RAM_32_64 <= wbs_stb_i  AND wbs_cyc_i  AND wbs_we_i AND (NOT
wbs_adr_i(9));
        s_w_adr <= wbs_adr_i(8 downto 2);

        s_r_cont <= r_cont&(NOT fsm_load);

        Inst_RAM_32_64: RAM_32_64 PORT MAP(
                clk => wb_clk_i,
                dat_in => wbs_dat_i_s,
                dat_out => s_data_in,
                w_adr => s_w_adr,
                r_adr => s_r_cont,
                we => we_RAM_32_64
        );

        process (wb_clk_i,wb_rst_i)

        begin
                if (wb_clk_i'EVENT AND wb_clk_i='1') then
                        if (wb_rst_i = '1') then
                                reg <= (others => '0');
                                reg_start <= (others => '0');
                                reg_key0 <= (others => '0');
                                reg_key1 <= (others => '0');
                                reg_key2 <= (others => '0');
                                reg_key3 <= (others => '0');
                                reg_mode <= (others => '0');
                        else
                                if ((wbs_stb_i AND wbs_cyc_i AND wbs_we_i AND
wbs_adr_i(9)) = '1') then
                                        reg <= wbs_dat_i_s(7 downto 0);
                                        case wbs_adr_i(4 downto 2) is
                                                when "000" =>
                                                        reg_start <= wbs_dat_i_s;
                                                when "001" =>
                                                        reg_mode <= wbs_dat_i_s;
                                                when "010" =>
                                                        reg_key0 <= wbs_dat_i_s;
                                                when "011" =>
                                                        reg_key1 <= wbs_dat_i_s;
                                                when "100" =>
                                                        reg_key2 <= wbs_dat_i_s;
                                                when "101" =>
                                                        reg_key3 <= wbs_dat_i_s;
                                                when others => null;
```

```
                        end case;
                    end if;
                end if;
            end if;
        end process;



        FSM : process(wb_clk_i) is
        begin
            if wb_clk_i'event and wb_clk_i = '1' then
                if wb_rst_i = '1' then
                    r_cont <= (others => '0');
                    w_cont <= (others => '0');
                    DONE <= '0';
                    estado <= "0000";

                else
                    case estado is
                        when "0000" =>--estado inicial
                            DONE <= '0';
                            if (reg_start(0) = '1') then
                                estado <= "0001";
                            end if;
                        when "0001" =>
                            --r_cont <= r_cont + 1;
                            estado <= "0010"; --load a '1'
                        when "0010" =>
                            --r_cont <= r_cont + 1;
                            estado <= "0011"; --load a '0'
                        when "0011" => estado <= "0100"; -- start a '1'
                        when "0100" => estado <= "0101";-- start a '0'
                        when "0101" => estado <= "0110";-- start a '0'
                        when "0110" =>
                            estado <= "0111";
                            r_cont <= r_cont + 1;
                        when "0111" =>
                            --esperamos a que se acabe la operacion
                            if (done_s = '1') then
                                estado <= "1000";
                            end if;
                        when "1000" =>
                            w_cont <= w_cont + 1;
                            if (r_cont = "00000") then
                                estado <= "1001";
                            else
                                estado <= "0001";
                            end if;
                        when "1001" =>
                            DONE <= '1';
```

```
                                        r_cont <= (others => '0');
                                        w_cont <= (others => '0');
                                        if (reg_start(0) = '0') then
                                                estado <= "0000";
                                        end if;
                                when others => null;
                        end case;
                end if;
        end if;
    end process FSM;
    fsm_load <= estado(0) and (not estado(1)) and (not estado(2)) and (not
estado(3));
    fsm_start <= estado(1) and estado(0) and (not estado(2)) and (not estado(3));
    we_RAM_128_32 <= (not estado(1)) and (not estado(0)) and (not estado(2)) and
estado(3);
    --r_CE <= (estado(0) xor estado(1)) and (not estado(2)) and (not estado(3));
    --w_CE <= we_RAM_128_32;

  Select_in_key : process(estado,reg_key0,reg_key1,reg_key2,reg_key3) is
    begin
                case estado is
                when "0000" =>
                        s_key <= reg_key0&reg_key1;
                when "0001" =>
                        s_key <= reg_key0&reg_key1;
                when "0010" =>
                        s_key <= reg_key2&reg_key3;
                when "0011" =>
                        s_key <= reg_key2&reg_key3;
                when others =>
                        s_key <= reg_key2&reg_key3;
                end case;
    end process Select_in_key;

  REG_READ_PROC : process(wbs_adr_i, sal_32,reg_start,reg_mode,reg_key0,
                reg_key1,reg_key2,reg_key3) is
  begin
                if (wbs_adr_i(9) = '0') then
                        DAT_O_s <= sal_32;
                else
                 case wbs_adr_i(4 downto 2) is
                        when "000" =>
                                DAT_O_s <= reg_start;
                        when "001" =>
                                DAT_O_s <= reg_mode;
                        when "010" =>
                                DAT_O_s <= reg_key0;
                        when "011" =>
                                DAT_O_s <= reg_key1;
                        when "100" =>
```

```
                        DAT_O_s <= reg_key2;
                    when "101" =>
                        DAT_O_s <= reg_key3;
                    when "110" =>
                        DAT_O_s <= "00000000000000000000000000000000"&DONE;
                    when "111" =>
                        DAT_O_s <= "00000000000000000000000000000000"&r_cont;
                    when others => null;
                end case;
               end if;
  end process REG_READ_PROC;

        wbs_ack_o <= (wbs_stb_i AND wbs_cyc_i);
        wbs_err_o <= '0';
        wbs_rty_o <= '0';
        wbs_dat_o <= DAT_O_s;
        led1 <= reg;

        -- las señales del WB no se usan (las fijamos a '0')
        wbm_cyc_o <= '0';
        wbm_stb_o <= '0';
        wbm_sel_o <= "0000";
        wbm_we_o <= '0';
        wbm_adr_o <= "00000000000000000000000000000000";
        wbm_dat_o <= "00000000000000000000000000000000";
        wbm_cab_o <= '0';
        led <= '0';


        --Transformacion de Little Endian a Big Endian
        wbs_dat_i_s <= wbs_dat_i;

end Behavioral;
```

f. Bibliografía

- Nicolas Courtois, Josef Pieprzyk, "Cryptanalysis of Block Ciphers with Overdefined Systems of Equations". ASIACRYPT 2002, pp267–287.
- Joan Daemen and Vincent Rijmen, "The Design of Rijndael: AES - The Advanced Encryption Standard". Springer-Verlag, 2002.
- Niels Ferguson, John Kelsey, Stefan Lucks, Bruce Schneier, Michael Stay, David Wagner and Doug Whiting, “Improved Cryptanalysis of Rijndael”. FSE 2000, pp213–230
- Sitio web del algoritmo Rijndael: http://www.iaik.tu-graz.ac.at/research/krypto/AES/
- Hans-Peter Messmer, “The indispensable PC hardware book”. Addison-Wesley, 2002
- Documentación de AVNET para la placa Xilinx Virtex II Pro

g. Referencias

[FIPS1] "FIPS-197", http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[DAEM1] "The Design of Rijndael" , 2001, J. Daemen, V. Rijmen, Springer Verlag.

[FERG1] "Improved Cryptanalysis of Rijndael", 2000, Niels Ferguson, John Kelsey, Stefan Lucks, Bruce Schneier, Michael Stay, David Wagner and Doug Whiting, pp213–230

[LIPM1] "Fast implementation of the AES Candidates", K.Aoki, H.Lipmaa, AES3papers-3-20.

[HKUO1] "A 2.29 Gb/s, 56 mW non-pipelined Rijndael AES Encryption IC in a 1.8 V, 0.18 um CMOS Technology", H. Kuo, I. Verbauwhede, P. Schaumont, Custom Integrated Circuits Conference.

[HKUO2] "Architectural Optimization for a 1.82 Gb/s VLSI Implementation of the AES Rijndael algorithm", H. Kuo, I. Verbauwhede, Cryptographic Hardware and Embedded Systems (CHES 2001), LNCS 2162, pp 51-64.

[ELBI1] "An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists", A.J. Elbirt, W. Yip, B. Chetwynd, C. Paar, NIST AES3papers-1-08.

[GANE1] http://www.codeproject.com/cpp/aes.asp

[VRIJ1] http://www.iaik.tu-graz.ac.at/research/krypto/AES/old/%7Erijmen/rijndael/

[WISH1] http://www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf