Universidad Autónoma de Madrid

Departamento de Ingeniería Informática

Coprocesadores Dinámicamente Reconfigurables en Sistemas Embebidos basados en FPGAs

Tesis Doctoral

Autor: Iván González Martínez

Director: Francisco J. Gómez

Escuela Politécnica Superior

Marzo 2006

Universidad Autonoma de Madrid

Computer Engineering Department

Dynamically Reconfigurable Coprocessors in FPGA-based Embedded Systems

PhD. Thesis

A MI FAMILIA
A MI NOVIA

In Memoriam of Prof. Javier Martinez

ESTA TESIS ESTÁ DEDICADA A JAVIER MARTÍNEZ, UNA PERSONA EXTRAORDINARIA

Resumen

Las nuevas características de los dispositivos FPGA actuales y su gran capacidad permiten el diseño de sistemas digitales complejos. Estos sistemas pueden incluir diseños específicos para una determinada tarea, y uno o varios microprocesadores, los cuales pueden estar implementados en la propia lógica reconfigurable o insertados como cores hardware. Sin embargo, la característica más interesante de las FPGAs y que les ha proporcionado actualmente su éxito en el marco de la investigación, es su capacidad de cambiar el diseño que internamente se les ha configurado. Es más, algunos dispositivos FPGA permiten cambiar solamente una parte de su configuración sin que este cambio afecte al funcionamiento del resto del diseño. Es lo que se conoce como Reconfiguración Parcial Dinámica. Esta nueva capacidad permite a los sistemas digitales resolver una de las mayores desventajas de los diseños hardware: la falta de flexibilidad. La reconfiguración parcial permite cambiar la funcionalidad del hardware del mismo modo que un microprocesador puede ejecutar varias tareas software, haciendo posible que partes del diseño puedan ser modificadas en tiempo de ejecución cuando la aplicación lo requiera.

El objetivo de este trabajo es el estudio de la reconfiguración parcial de los dispositivos FPGA y las diferentes metodologías y herramientas, algunas empleadas en trabajos de investigación previos y otras desarrolladas en estas tesis, que permiten realizar sistemas parcialmente reconfigurables. Como caso de estudio se propone el diseño de algoritmos de cifrado en hardware y la implementación de sistemas criptográficos. A los largo de esta tesis, la reconfiguración parcial se ha empleado en el diseño de coprocesadores de alto rendimiento, y para el desarrollo de sistemas embebidos en FPGA con coprocesadores dinámicamente reconfigurables. En primer lugar, un diseño del algoritmo IDEA fue optimizado, en rendimiento y recursos lógicos empleados, mediante el uso de multiplicadores por constante reconfigurables. En segundo lugar, se desarrolló un sistema basado en un procesador embebido cuya principal característica es su capacidad de auto-reconfiguración. El procesador puede reconfigurar la FPGA donde se encuentra implementado para cambiar el coprocesador cuando es necesario. Estas dos soluciones son complementarias: la primera emplea la reconfiguración para optimizar el rendimiento, y la segunda, para incrementar la flexibilidad del sistema permitiendo añadir nuevo hardware.

Como trabajo final, esta tesis muestra la ejecución de una aplicación estándar de seguridad sobre un sistema auto-reconfigurable: la aplicación SSH (Secure Shell) sobre el sistema operativo uClinux. Esta implementación en un dispositivo FPGA comercial de bajo coste demuestra el potencial y viabilidad de la reconfiguración parcial.

Summary

Modern FPGA devices allow engineers to build very complex digital systems. Nowadays, a typical design contains dedicated hardware modules for specific tasks, and one or more general-purpose processors. There are new FPGA families that include those processors as hard-cores, but they can always been implemented in the reconfigurable logic fabric as soft-core processors.

The benefits of reconfigurable computing are well known, being one of the most remarkable the possibility of changing previously configured designs. This is specially appreciated in the research area. In some devices it is possible to change part of their configuration while the rest of the circuit continues its normal operation. This characteristic is known as Dynamic Partial Reconfiguration, and it is exclusive of some FPGAs. This feature solves the main problem of digital systems implemented in hardware: Their lack of flexibility. As well as a processor can execute different software tasks, partial reconfiguration allows changing the functionality of the hardware in a digital system. It makes possible to modify at run-time a design when an application demands that.

The main goal of this work is to take advantage of the partial reconfiguration capabilities of FPGAs and to test different design tools and methodologies, some applied in previous research works, and others developed in this thesis. As a case study, the implementation of ciphering algorithms and the development of cryptography systems have been proposed. Along this thesis, partial reconfiguration is used to design high-performance cryptographic coprocessors, and to implement FPGA-based embedded systems featuring dynamically reconfigurable coprocessors. First, an implementation of the IDEA algorithm was optimised in performance and resource usage, taking advantages of reconfigurable constant multipliers. Second, a system based in an embedded processor was implemented, whose main characteristic is its self-reconfiguration capability. The processor can reconfigure the FPGA in which it is running to change the coprocessors, as demanded by the software applications. These two solutions are complementary: The first one uses reconfiguration to optimise performance, and the second one to increase the flexibility by allowing the addition of new hardware.

As final work, this thesis shows a standard secure application running on a self-reconfigurable system: The Secure Shell (SSH) on the uClinux operative system. This implementation on a commercial low-cost FPGA is used to demonstrate the potential and feasibility of partial reconfiguration.

Agradecimientos

A *Paco*, mi director, le debo el haber llegado hasta este momento. Ha sabido combinar la rectitud de un buen director de tesis, guiándome adecuadamente durante estos años, con una amistad que siempre he considerado fundamental. He de agradecerle doblemente el haber cumplido como amigo y compañero, y al mismo tiempo, conseguir que no me desviara demasiado de los objetivos de la tesis, y más, sabiendo lo "difícil" que soy en algunas ocasiones. Sin duda, su confianza y saber hacer han sido esenciales.

A *Sergio*, por su indudable labor y colaboración en todo momento.

A todos los amigos y compañeros que han compartido conmigo todos estos años, y cuyo apoyo y amistad han sido indispensables para mí: Ruben, Miguel, Juan, Ángel, Gustavo, Elías, Eduardo, Javi, Jorge, Tani, Alberto(s) y muchos más, que aunque no aparecen aquí, saben perfectamente lo agradecido que les estoy.

A los doctores *Andrea Boni* (UNIVERSITY OF TRENTO) y *Oswaldo Cadenas* (UNIVERSITY OF READING) por su colaboración en la revisión de la tesis con el objetivo de obtener la Mención de Doctorado Europeo. Por el mismo motivo, al profesor *Eduardo Sánchez* (ECOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE) por permitirme realizar una estancia pre-doctoral en su grupo de trabajo.

Índice general

Índice de figuras

Índice de tablas

Embebidos, empotrados o especializados

La traducción al español del término *embedded systems* siempre ha dado lugar a un amplío debate sobre cual es la palabra exacta en español que presenta un significado equivalente al término *embed* del inglés. Según la definición del diccionario Merriam-Webster:

embed Pronunciation: im-'bed
Function: verb
Inflected Form(s): **em·bed·ded; em·bed·ding**
transitive senses
1 a : to enclose closely in or as if in a matrix <fossils *embedded* in stone> **b :** to make something an integral part of <the prejudices *embedded* in our language> **c :** to prepare (a microscopy specimen) for sectioning by infiltrating with and enclosing in a supporting substance
2 : to surround closely <a sweet pulp *embeds* the plum seed>
intransitive senses : to become embedded

Según el diccionario de Lengua Española de la Real Academia de la Lengua Española:

embeber. *(Del lat. imbibêre).* **1. tr.** Dicho de un cuerpo sólido: Absorber a otro líquido. *La esponja embebe el agua.* **2. tr.** Empapar, llenar de un líquido algo poroso o esponjoso. *Embebieron una esponja en vinagre.* **3. tr.** Dicho de una cosa: Contener, encerrar dentro de sí a otra. **4. tr.** Dicho de una cosa inmaterial: Incorporar, incluir dentro de sí a otra. **5. tr.** Encajar, embutir, meter algo dentro de otra cosa. **6. tr.** Recoger parte de una cosa en ella misma, reduciéndola o acortándola. *Embeber un vestido, una costura.* **7. intr.** Encogerse, apretarse, tupirse. *El lino y la lana embeben al lavarlos.* **8. prnl.** embebecerse. **9. prnl.** Instruirse con rigor y profundidad en una doctrina, teoría, etc. **10. prnl.** Entregarse con vivo interés a una tarea, sumergirse en ella. **11. prnl.** *Taurom.* Dicho de un toro: Quedarse parado y con la cabeza alta cuando recibe la estocada.

empotrar. *(De potro).* **1. tr.** Meter algo en la pared o en el suelo, generalmente asegurándolo con fábrica. **2. tr.** Entre colmeneros, poner en un hoyo las colmenas para partirlas. **3. prnl.** Dicho de una cosa: Incrustarse en otra, especialmente al chocar con violencia contra ella. **U. t. c. tr.** *Empotró el coche en el escaparate.* **U. t. en sent. fig.**

especializar. 1. tr. Limitar algo a uso o fin determinado. **2. intr.** Cultivar con especialidad una rama determinada de una ciencia o de un arte. **U. t. c. prnl.**

Capítulo 1

Introduction

1.1. Overview

The first FPGA devices appeared in the mid-eighties and from then on, reconfigurable hardware has been present in many areas. At the beginning, reconfigurable devices were used as glue logic, to replace a set of discrete components or used to prototype different digital designs, even microprocessors, taking advantage of the possibility to change their configuration. As the FPGA technology improved, reconfigurable devices increased their integration level, and nowadays it is possible to find FPGAs with millions of equivalent logic gates. In present reconfigurable devices it is feasible to implement complete digital systems, that is, Systems-on-a-Chip (SoC) [1, 2]. This kind of systems has been traditionally associated with ASIC technology, so a new definition for these systems is necessary: Configurable System-on-Chip (CSoC) [3, 4]. The use of FPGAs to implement SoC solutions has intensified during the lasts years [5] and it has increased the interest for these devices. It is especially important the possibility to implement processors in reconfigurable logic, also know as soft-core processors (SCP) [6, 7, 8, 9] or alternatively, to use embedded hard-core processors [10, 11, 12, 13]. Besides, FPGAs combine the flexibility of general-purpose processors with the speed of custom hardware [14], making them suitable platforms to develop hundred of applications.

Currently, a revolution in research projects is taking place due to the capability of FPGAs to be reconfigured partially. This characteristic, nowadays available only in some devices from Xilinx and Atmel, is known as Partial Reconfiguration [15] and it means that small parts of the design can be modified without affecting the rest of the device. There are two improvements for Partial Reconfiguration: Dynamic Partial Reconfiguration and Self-Reconfiguration [15, 16]. The first one ensures that some parts of the device can be reconfigured while the rest of the device continues

working. The second one expands the Dynamic Partial Reconfiguration to include the possibility that the device can control its own reconfiguration.

The history of the Dynamic Partial Reconfiguration begins with the first partially reconfigurable device, the XC6200 family from Xilinx [17]. One of the first systems that made use of this device was the Flexible URISC architecture [18], which presented a minimal processing architecture based on only one instruction: "move to memory from memory". Using this instruction and reconfiguring the different modules connected to the system bus, it was possible to run different applications. URISC was the first self-reconfigurable system because the configuration memory of the device was included in the memory system. Nowadays, the XC6200 family is obsolete and the new Virtex [19] (including Virtex-II and the most modern Virtex-4) families from Xilinx are the alternative.

Unlike XC6200 family, the partial reconfigurable capability in Virtex families is neither well-documented nor supported by the traditional design tools. That is the reason for proposing different design flows and frameworks [20, 21, 22] to employ dynamic and partial reconfiguration, which have obtained interesting and hopeful practical results [23, 22, 21, 24]. For example, in [21, 24], the reconfiguration capability of Virtex FPGAs is used to implement an audio decoder system based on a fixed soft core processor and one dynamically reconfigurable coprocessor. Depending of the audio format, the coprocessor is dynamically reconfigured with the correct decoder core. Other researches apply the same idea to build dynamic reconfigurable hardware modules [25, 26] oriented to the rapid prototyping of routers [27] and firewalls [28].

The Self-Reconfiguration capability has also been the objective of some research works in order to develop self-reconfigurable CSoCs. One system is self-reconfigurable when it has the capability to generate, at run-time, a configuration bitstream and use it to modify its own configuration. Several systems have been implemented in the last years [29, 30, 31, 32, 33, 34]. However, the most interesting contribution in this area was presented in the work "Self-Reconfiguring Platform" [16, 35], where the reconfigurable capability can be controlled by the embedded processor inside the FPGA. Although this system is under development, some research works have used this approach in industrial designs [36] and also, a framework is available [22] to apply this capability in Xilinx Virtex-II FPGAs.

Finally, another interesting research topic in dynamically reconfigurable systems is the development of operative systems for reconfigurable hardware [37, 38, 39, 40, 41]. The goal is to control different hardware tasks using real-time standard operative systems executed in embedded processors inside the device. The OS manages the context-task switching of software and hardware.

1.2. Motivation and thesis goal

In this thesis, the development of embedded systems based on reconfigurable hardware for secure communication applications is proposed. These systems are used as testbench for measuring the potential of the new capabilities that are available in present FPGAs and also as an example of the different design methodologies.

Until a few years ago, the PC had been the reference platform to implement most digital systems. Recently, however, with the emergence of new technologies, communication applications require new capabilities to be adapted better to the environment where they are used, emphasising their pervasiveness and mobility. However, mobile systems are limited by the weight and size of the device, as well as their power consumption, so it is necessary to integrate all the features in a small and thrifty solution, offering a relatively high power of computation. That is the reason why embedded systems are nowadays ideal candidates to develop communication applications. Examples of such applications are cellular phones, faxes, pagers, and Internet solutions such as modems, multi-service network platforms that allow the implementation of IP telephony, Digital Subscriber Line (DSL) technologies, and some e-commerce devices, to enumerate a few. Many of these applications also rely heavily on security mechanisms, including security for wireless phones, faxes, wireless computing, pay-TV, and copy protection schemes for audio/video consumer products. Note that most of those embedded applications will be wireless, which makes the communication channel especially vulnerable so the need for security is even more evident.

This functionality integration of both communications and computation requires data processing in real time, and embedded systems have shown to be good solutions for many applications. However, the implementation of cryptographic systems presents several requirements and challenges. First, the performance of the algorithms is often crucial. Encryption algorithms need to reach the transmission rates of the communication links. However, fast running encryption might mean high product costs, and traditionally, higher speeds were achieved using custom hardware devices (ASICs). Additionally to the performance requirements, to guarantee a high security level is a remarkable challenge. An encryption algorithm running on a general-purpose computer has the limitation of the physical security, because the secure storage of keys in memory is difficult on most operative systems. In contrast, hardware encryption devices can be securely encapsulated to prevent possible attackers from tampering with the system. Thus, custom hardware is the selected platform for many secure protocol designers. Hardware solutions, however, come with the well-known drawbacks of reduced flexibility and potentially high costs. These drawbacks are especially prominent in security applications which are designed using new security protocol paradigms. Many of the new security protocols separate the choice of cryptographic algorithm from the design of the protocol: Users negotiate the algorithm to be used for a particular secure

session. To support these type of applications, the new devices need not only to support a single cryptographic algorithm and protocol, but also be "algorithm agile", that is, be able to select from a variety of algorithms. For example, IPSec (the security standard for the Internet) [42] allows to choose out of a list of different symmetric as well asymmetric ciphers. Hardware-accelerated embedded processors are currently an integral part of many communications devices. When their flexibility to be programmed is combined with their ability to perform arithmetic operations at moderate speeds, it is easy to understand why they are very promising platforms to implement cryptographic algorithms [43].

FPGAs have been widely used to implement cryptography algorithms because these devices present some interesting advantages for this kind of applications [44, 45]. At the first time, FPGA-based solutions were traditionally implemented as coprocessors of general-purpose microprocessors. Currently, with the arrival of the new technologies that look for small and powerful solutions, modern FPGAs can be used as an alternative of conventional embedded systems, creating complete secure solutions based on an embedded processor and some coprocessors. These FPGA-based SoC solutions present interesting advantages compared with the traditional embedded systems. The same flexibility of general-purpose processors is achieved by implementing one or more processors in a FPGA. Also, the possibility to reconfigure the hardware configuration of the FPGA allows having a solution well-adapted to the current secure protocols and requirements of the application. Reconfigurable devices show similar security levels than hardware solutions, and they also fit with the security rules required in embedded systems [46].

In addition to all previous advantages, it is possible to benefit from the partial reconfiguration of these devices. This feature allows FPGAs to offer a flexibility and adaptability never seen before, which is particularly interesting for secure applications whose protocols dynamically negotiate the encryption algorithm from a set of standards. This is the main goal of this thesis: *To develop FPGA-based embedded systems that make use of partial reconfiguration to reconfigure dynamically the coprocessor unit.* This goal can be achieved by defining some secondary objectives:

- Evaluation of the performance of cryptographic coprocessors in FPGA. Improvements related with the technology evolution of reconfigurable devices.
- Development of CSoCs based on FPGA-embedded processors to evaluate the soft-core and hard-core solutions, and to study the possibilities offered by these processors to execute ciphering algorithms.
- Use of partial reconfiguration on the developed CSoCs. The objective is to obtain a self-reconfigurable embedded system that dynamically reconfigures one part of itself (coprocessor).

- Development of an application that makes use of the self-reconfigurable capabilities for its application to embedded secure systems.

1.3. Technological framework and methodologies

The evolution of the FPGA technology and tools for partial reconfiguration has been constant during the last years. The defined objectives at the beginning of this work, in the year 2001, were too demanding for the technology available at that moment. Therefore this thesis has been adapting to the evolution in the area, using the resources available in each moment. The different stages in this work are related to the evolution of the tools and methodologies in partial reconfiguration. This follow-up has allowed the author to apply different technologies in different implementations, obtaining finally the main goal.

The first designs in the thesis were the development of specific circuits using the traditional hardware description languages such VHDL or Verilog, to create a set of cryptographic coprocessors. The idea behind the development of these circuits was to use them together with an external general-purpose microprocessor. For this work, a platform that combined a microprocessor with reconfigurable devices was used [47]. In this initial stage, the IDEA algorithm was selected to test the performance, due an architecture that includes most of the components used by cryptographic applications: module multiplication, permutations, logical operations, etc. Different approaches were studied according to the limitations of the reconfigurable platform chosen.

After this initial study, the use of the advanced reconfiguration capabilities available in the Virtex family was taken into account. When the JBits tool was released by Xilinx, in 2002, it made possible to utilise partial reconfiguration in Virtex FPGAs. Using this tool, a new implementation of the IDEA algorithm using reconfiguration was created. The development of this implementation allowed the author to verify the potential of the dynamic partial reconfiguration, but also to realize the limitations of the JBits tools when it is necessary to implement complex designs. In order to solve this problem, in future developments a new methodology was proposed, which is based in the combination of the traditional tools for developing the circuit, and the JBits tool for reconfiguring it. With this new methodology, better results in performance and area usage than employing only JBits were obtained (actually, the best one to date).

In parallel with the previous development, Xilinx presents its new Virtex-II family. The main characteristic of this family is the availability of embedded multipliers inside the device. The IDEA algorithm is, again, a good candidate to evaluate the potential of this new family, being necessary to compare the obtained results in the new family with the previous results in Virtex using partial reconfiguration.

The next step in the evolution of the FPGAs was the availability of hard-core embedded processors in some new devices [10, 11, 12, 13], and the tools to implement processors in reconfigurable logic [6, 7, 8, 9]. Although at the beginning these tools were very limited, it was finally possible to implement systems based in embedded processors with custom coprocessors. In this way, the external processor employed in the first experiments was replaced by another one placed inside the FPGA. Moreover, the high flexibility provided by the processors implemented in reconfigurable logic was also evaluated, and different cryptographic cores were developed for these tests.

The last step in this work is to use partial reconfiguration to change the different coprocessors, allowing the processor to employ any of them when demanded by the software applications. This solution avoids implementing all the different coprocessors in the device. Although it is possible to use the JBits tool, the choice was to follow the Xilinx application note oriented to the partial reconfiguration of cores [20]. This methodology proposed by Xilinx was suitable, provided that some improvements were done, for designing a system based on a processor with different dynamically reconfigurable coprocessors. The first prototype needed that the reconfiguration process was done by hand. An improvement of this system made possible that the processor employed one of the configuration interfaces available in the FPGA in order to reconfigure its own coprocessors. This feature is known as self-reconfiguration, and the target system is called self-reconfigurable system. In parallel with the availability of embedded processors in FPGA, different operative systems were ported to these new processors [48, 49]. The possibility to employ an operative system with FPGA-based processors converts FPGAs into suitable platforms to implement different applications, replacing other digital devices. By means of one of these operative systems it was possible to implement an application on a self-reconfigurable system that uses partial reconfiguration to change the coprocessors it used.

1.4. Application and importance of the contributions

The different results obtained during this thesis have been presented in international conferences and research journals. The increasing number of related publications in this area shows the interest that FPGA-based embedded systems and partial reconfiguration have obtained in the last years.

First, reconfigurable hardware has been used for the acceleration and optimisation of cryptographic algorithms. Typically, improvements in these algorithms are obtained by means of custom operational units or by using the new resources available in modern devices. This thesis follows a different approach, were the benefits of partial reconfiguration are applied to optimise the architecture of the algorithm implementation. This novel approximation allowed us to obtain, to the best

of our knowledge, the fastest and most optimised FPGA implementation of the IDEA ciphering algorithm to date.

Additionally, the capability of modern FPGAs to implement CSoCs with embedded processors has incremented the interest in partial reconfiguration. The use of these systems in standard security applications, which entail an extensive set of cryptographic algorithms, requires a great flexibility, and partial reconfiguration can offer it. In this thesis the first run-time partially reconfigurable system implemented on a commercial low-cost FPGA has been presented. On this platform, a real application (SSH on uClinux) has been executed, using the self-reconfigurable capabilities available on it.

Finally, the results showed in this thesis are extensive to other application areas. The different partial reconfiguration methodologies, dynamically self-reconfigurable systems, etc. can be easily used in digital processing, mobile communications, multimedia, etc.

1.5. Thesis structure

The chapters of the thesis are related to the different stages involved in this research. In this way, the next chapter states the research area, and the evolution of the technology and the achieved steps are shown in chapters 3 to 5. Finally, conclusions are shown in chapter 6, and complementary information is given in 2 additional appendixes.

Chapter 2 introduces the different aspects related to reconfigurable systems and FPGAs. The reconfiguration capabilities of present devices and the methodologies applied in the different research works developed during the recent years are mentioned. An introductory background of FPGA-based cryptography is also included too.

Chapter 3 shows the advantages that partial reconfiguration offers to optimise and adapt cryptographic algorithms to reconfigurable hardware. In particular, the IDEA algorithm has been selected as case-study for the different design tests. The methodologies that allow the designer to use partial reconfiguration have been applied and the obtained results have been compared with other implementations available in the bibliography.

Chapter 4 details the design of embedded systems based on FPGAs using processors in reconfigurable logic. The flexibility offered by these processors to configure or modify their architecture has been analysed. It is particularly interesting the improvement that supposes the connection of dedicated hardware cores as coprocessors, in order to increase the performance of the different cryptographic algorithms. The potential to implement multiprocessor architectures in FPGAs has also been explored.

Chapter 5 focuses on the development of self-reconfigurable embedded systems in FPGAs. The capability of run-time reconfiguration in this type of systems has been studied and a self-reconfigurable platform has been developed. As a final result, an application running on this system has been presented as a practical example of partial reconfiguration: the Secure Shell (SSH) on uClinux. SSH allows the secure communication between different systems. This application shows the advantages provided by the possibility of dynamically changing the coprocessors when an application demands that. It also proves that it is possible to implement a standard application using self-reconfiguration on a low-cost commercial device.

Chapter 6 enumerates the main conclusions, summaries the results of this thesis, and proposes the future work.

Capítulo 2

Sistemas Reconfigurables y Criptografía en FPGAs

2.1. Introducción

Tradicionalmente, para el desarrollo de sistemas computacionales se han planteado dos posibilidades bien definidas. Por un lado la posibilidad de "ejecución software", donde la tarea se realiza mediante la realización de un conjunto de instrucciones. Se trata de una solución muy flexible porque permite ejecutar diferentes tareas aunque la eficiencia es menor dada la funcionalidad general del microprocesador. Por otro lado la posibilidad de "ejecución hardware", donde se desarrolla un circuito especifico para la tarea a realizar. Esta solución presenta un mejor rendimiento dada la especialidad del diseño y el posible paralelismo de la aplicación, pero reduce la flexibilidad del sistema debido a la imposibilidad de cambiar la funcionalidad una vez fabricado.

Los dispositivos reconfigurables, y por ello los sistemas reconfigurables, se presentan como una solución intermedia entre el uso de microprocesadores de propósito general y el diseño de circuitos específicos (ASIC). En una situación ideal los sistemas reconfigurables combinan lo mejor de las dos soluciones anteriores: la flexibilidad del software que se ejecuta en un microprocesador de propósito general y la velocidad del hardware específico [14]. La figura 2.1 ilustra la situación de los sistemas reconfigurables.

Procesadores vs. Sistemas reconfigurables Los procesadores modernos están formados por pequeñas unidades de ejecución que presentan una fuerte multiplexación para soportar la demanda de computación. Sin embargo, en cada momento solo una pequeña cantidad de los recursos hardware

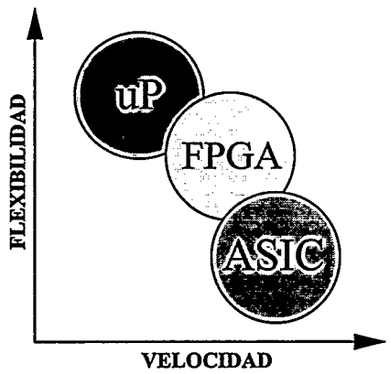


Figura 2.1: Microprocesadores, ASICs y Sistemas reconfigurables

disponibles se emplean de forma activa. Además, los microprocesadores presentan una arquitectura fija, lo que implica que la decodificación secuencial de instrucciones, el acceso a la memoria y la existencia de una unidad de control fija, limitan el rendimiento que puede ser alcanzado para una aplicación. Por el contrario, en los sistemas reconfigurables las operaciones de computación se implementan mediante la ejecución paralela de varias unidades de ejecución diseñadas específicamente, en vez de ejecutarse de forma secuencial como las instrucciones en un microprocesador (figura 2.2). De este modo, los sistemas reconfigurables pueden presentar un alto potencial computacional y un reducido conjunto de instrucciones. Al contrario de lo que se podría pensar, el rendimiento de los microprocesadores no se puede incrementar fácilmente mediante la incorporación de unidades funcionales fijas, porque el alto rendimiento deseado es proporcional al área consumida por estas unidades, lo que no permite la construcción de sistemas de bajo coste para un amplio grupo de aplicaciones [50]. Por ello, una probable tendencia de los microprocesadores futuros será incorporar adaptabilidad al hardware. La lógica reconfigurable soportará la ejecución paralela de un conjunto de instrucciones, y los principales bloques funcionales encontrados en los microprocesadores actuales deberán ser correctamente interconectados de una manera reconfigurable. Es más, la localidad de las operaciones a ejecutar podría mostrar suficiente regularidad para el uso de una implementación reconfigurable frente a un microprocesador [51].

ASICs vs. Hardware reconfigurable En el otro lado del espectro de computación aparecen los ASICs, que son circuitos diseñados específicamente para aplicaciones determinadas y por tanto, presentan un rendimiento superior para un conjunto restringido de tareas computacionales debido

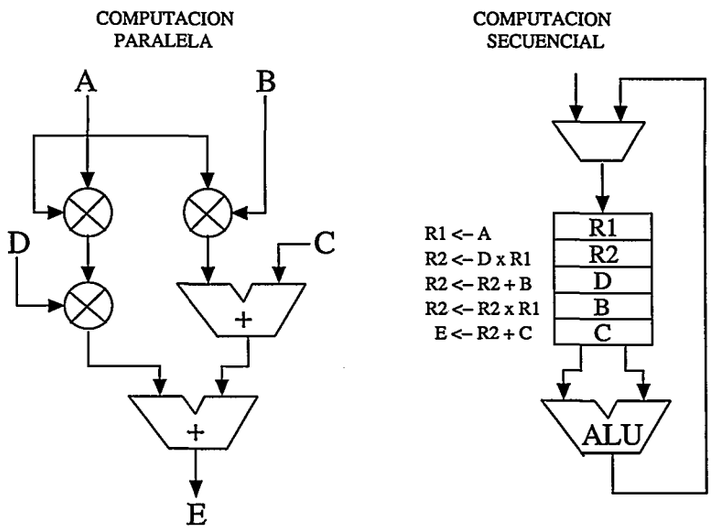


Figura 2.2: Computación paralela vs. secuencial

a su funcionalidad fija. Por ejemplo, en ciertas aplicaciones de procesado de señal los ASICs ofrecen un rendimiento superior a las mejores FPGAs en un factor dos [52, 53]. Una desventaja clara de los ASICs es la imposibilidad de optimizaciones post-diseño y actualizaciones para mejorar el rendimiento [54]. La razón fundamental de la popularidad de los ASICs en aplicaciones comerciales es bien conocida: cualquier algoritmo del que se quiera obtener el máximo rendimiento debe ser implementado en hardware [55]. Los ASICs además, dominan el segmento del bajo consumo, donde los sistemas reconfigurables no son competitivos actualmente. La industria de semiconductores tiene contemplados tradicionalmente a los ASICs como la única alternativa económica para los productos comerciales en masa, y los primeros dispositivos reconfigurables no han invadido el territorio de los ASICs. Sin embargo, esta situación ha cambiado con la aparición de las modernas FPGAs de millones de puertas que permiten niveles de integración sin precedentes. Los dispositivos reconfigurables deben hacer valer sus ventajas sobre los ASICs, por ejemplo en la reducción de costes, tiempo de diseño, y su flexibilidad para implementar diferentes funciones usando el mismo dispositivo [52].

2.2. Dispositivos reconfigurables

2.2.1. FPGAs

Desde hace dos décadas, las FPGAs han estado presentes en todo tipo de aplicaciones de red, telecomunicaciones, procesamiento de vídeo, etc. Estos dispositivos son circuitos electrónicos formados por un array de bloques lógicos programables y un conjunto de interconexiones también programables para unir esos bloques que permiten a los diseñadores de sistemas desarrollar soluciones que requieren de la velocidad del hardware a medida, manteniendo el beneficio de la flexibilidad de las soluciones basadas en microprocesadores.

Aunque en un inicio las FPGAs ya eran reconfigurables, los fabricantes de FPGAs añadieron a estos dispositivos la capacidad para reconfigurarse parcialmente, lo que se conoce como Reconfiguración Parcial [15]. Tradicionalmente si se deseaba cambiar el diseño de la FPGA se necesitaba parar el dispositivo y cargar la nueva configuración con la consecuente pérdida del diseño anterior - esto se conoce como Reconfiguración Estática. Aunque es aceptable para aquellos usuarios que emplean la FPGA como *glue-logic*, limita su uso si queremos que la FPGA se comporte como acelerador de algoritmos cuando la naturaleza de las tareas a ejecutar es dinámica. Además si se quiere actualizar la configuración del diseño es necesario enviar la configuración completa, lo que significa una gran cantidad de megabytes para diseños que contengan varios miles de puertas. Con la llegada de los dispositivos FPGA modernos, a partir de la familia Virtex de Xilinx [19], ya es posible reconfigurar parte del dispositivo sin tener que parar la FPGA y restablecer la configuración. Estos dispositivos permiten nuevas posibilidades en el diseño de circuitos evolutivos, tolerancia a fallos, etc.

Al mismo tiempo que se ampliaba la capacidad de reconfiguración de las FPGAs también ha crecido su densidad, y del mismo modo ha aumentado el tamaño y número de funciones que es posible implementar. En la actualidad, además de la lógica programable, las FPGAs integran nuevos componentes como procesadores, memorias, unidades aritméticas y mecanismos de comunicación a alta velocidad con elementos externos. Todo este avance tecnológico ha permitido la implementación en un único dispositivo reconfigurable de sistemas complejos, que junto con un incremento del interés en el hardware reconfigurable, ha dado lugar a diferentes tipos de sistemas:

- System-on-Chip (SoC): Circuito integrado formado por diversos módulos VLSI con distinta funcionalidad que interconectados entre sí ofrecen una funcionalidad específica para una aplicación.

- System-on-Programable-Chip (SoPC): Se aplica este término específicamente cuando el dispositivo utilizado para realizar el SoC es reconfigurable. En los SoPC no se utiliza la

capacidad de reconfiguración dinámica que puedan disponer estos integrados, sino que únicamente las facilidades que ofrecen estos dispositivos en la fase de desarrollo y posteriores actualizaciones del sistema.

- Configurable-System-on-Chip (CSoC) [3]: Mediante este término se definen los sistemas SoC en los que se hace uso de la capacidad de reconfiguración de los mismos para aplicaciones de computación reconfigurable. Pueden incluirse bajo la denominación CSoC tanto los sistemas que admiten diferentes configuraciones estáticas según ciertos condicionantes, como los que utilizan la reconfiguración parcial dinámica para modificar en tiempo de ejecución una sección hardware.
- Multiprocessor-Configurable-System-on-Chip (MCSoC): Se aplica esta definición a los sistemas CSoC que incluyan varias unidades procesador funcionando de forma simultánea.

2.2.2. Tecnología de reconfiguración

La reconfiguración de las FPGAs se basa en tecnología SRAM (Static Random Access Memory), de modo que los diferentes puntos de configuración se encuentran conectados a bits de la memoria, lo que hace posible la reconfiguración del dispositivo [56]. En función del contenido de la memoria programable se determina las interconexión entre los elementos computacionales y las funciones que estos elementos realizan. Usualmente la memoria de configuración almacena solo una configuración o *bitstream*, pero ciertos dispositivos pueden almacenar más de una [57], y los diferentes contextos de configuración pueden ser cambiados en unos ciclos de reloj. El contenido de las memorias SRAM es volátil lo que significa que la FPGA tiene que ser configurada, normalmente por una memoria externa no volátil, cada vez que se activa la alimentación. Los dispositivos FPGA admiten diversos sistemas para cargar el *bitstream* en la memoria SRAM de configuración [58, 59].

La reconfigurabilidad de las FPGAs permite cambiar la funcionalidad del hardware durante la ejecución pero presenta ciertas desventajas, ya que una gran parte de los recursos del chip se dedican a realizar esta función. Sin embargo, esto no debe ser el motivo para medir el coste y beneficio de la reconfiguración. A continuación se presentan brevemente los diferentes conceptos de reconfiguración que se manejan habitualmente en las aplicaciones con hardware reconfigurable:

- Reconfiguración Parcial: La reconfiguración parcial significa que el dispositivo puede ser reconfigurado mientras se encuentra activo [15]. Esto implica que la configuración almacenada en el dispositivo puede ser actualizada de forma selectiva, sin interferir a la operación del resto del circuito. Por ejemplo, en la familia Virtex porciones de la configuración de la FPGA tan pequeñas como un bit puede ser modificadas sin alterar el resto del dispositivo.

La reconfiguración parcial es un concepto importante, pero el problema de su aplicación práctica y la disponibilidad de herramientas de desarrollo que permitan su manejo no ha permitido su uso de forma extendida.

- Reconfiguración en Tiempo de Ejecución (-RTR- Run-Time Reconfiguration): La mayoría de las aplicaciones que se ejecutan en sistemas basados en FPGA emplean una única configuración por FPGA. Estas aplicaciones configuran la FPGA antes de iniciarse y mantienen la configuración hasta que la aplicación termina. Es decir, la funcionalidad del circuito no cambia mientras la aplicación se está ejecutando. Es lo que se conoce como Reconfiguración en Tiempo de Compilación (-CTR- Compilation-Time Reconfiguration) [60], porque la configuración completa del dispositivo se determina en el "momento de compilación" y no cambia durante la operación del sistema. Otra estrategia es implementar una aplicación con múltiples configuraciones por FPGA. En este escenario la aplicación se divide en operaciones que no necesitan operar concurrentemente. Cada operación se implementa como una configuración diferente que puede ser cargada en la FPGA durante la ejecución de la aplicación cuando es necesaria. Esto es lo que se conoce como Reconfiguración en Tiempo de Ejecución [60]. Al contrario que las aplicaciones CTR, que solo configuran la FPGA una vez durante toda la operación del sistema, las aplicaciones RTR normalmente reconfiguran varias veces la FPGA durante la operación normal de una única aplicación.

- Reconfiguración Dinámica y Auto-Reconfiguración: La Reconfiguración Dinámica y la Auto-Reconfiguración son dos de las formas más avanzadas de reconfiguración en FPGAs [35]. La reconfiguración dinámica implica que parte del dispositivo se está reconfigurando mientras se asegura el correcto funcionamiento de la parte que no está cambiando. La auto-reconfiguración extiende el concepto de reconfiguración dinámica. En este caso se asume que el dispositivo emplea parte de su circuito para el control de la reconfiguración de las otras partes de la FPGA. Es obvio que se debe garantizar la integridad del circuito de control durante la reconfiguración, por tanto, por definición la auto-reconfiguración es una forma especializada de reconfiguración dinámica. Tanto la reconfiguración dinámica como la auto-reconfiguración requieren de un mecanismo externo para configurar la FPGA cuando está se alimenta por primera vez, o el dispositivo se reinicia.

Las FPGAs que actualmente soportan reconfiguración dinámica son familias de dispositivos de los fabricantes Xilinx y Atmel. Particularmente, nos centraremos en los dispositivos del fabricante Xilinx. Los dispositivos empleados a lo largo de esta tesis se corresponden con FPGAs de las familias Virtex, Virtex-E, Virtex-II, Virtex-II Pro y Spartan-3. Esta última es una versión de bajo coste de la familia Virtex-II.

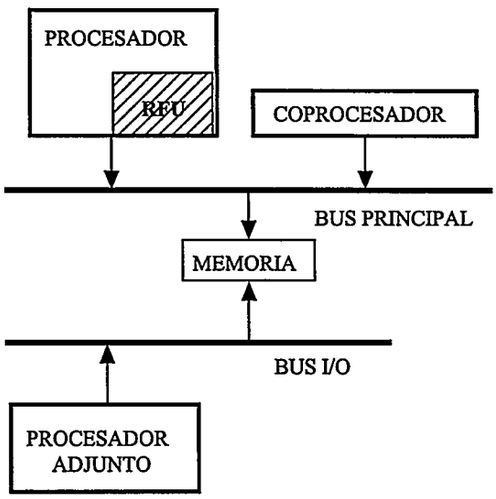


Figura 2.3: Tipos básicos de microprocesador reconfigurable

2.3. Sistemas reconfigurables

La tecnología FPGA ha sido empleada durante años en el desarrollo de sistemas reconfigurables experimentales. Básicamente estos sistemas se basan en la combinación de procesadores de propósito general y lógica reconfigurable. En función del mecanismo de conexión entre el procesador principal y la lógica reconfigurable, tal y como se muestra en la figura 2.3, podemos clasificar a los sistemas reconfigurables en tres clases principales [61]:

- Procesadores adjuntos.
- Coprocesadores.
- Unidades funcionales reconfigurables (RFUs).

Los requerimientos comunes para todas las clases de procesadores reconfigurables son un extenso camino de datos, pequeños tiempos de reconfiguración, y la posibilidad de reconfigurar el procesador durante la ejecución [62]. En todos los casos, el procesador convencional se emplea normalmente para soportar el volumen de funcionalidad requerida al implementar un algoritmo, mientras que el procesador reconfigurable se emplea para acelerar solo la parte computacionalmente más crítica del núcleo del programa [63].

A continuación se presentan algunos de los sistemas reconfigurables resultado de investigaciones en este área durante los últimos años.

2.3.1. Procesadores adjuntos

En este tipo de sistemas la lógica reconfigurable se encuentra conectada al bus de E/S o a la memoria del procesador principal. Al contrario que las RFUs y los coprocesadores, no se extiende el set de instrucciones del procesador.

La primera referencia a un sistema de este tipo la encontramos en los sistemas **Splash** [64] que datan de mediados de los 90. Se trataba de una placa de expansión para estaciones de trabajo formada por 32 FPGAs 3090 de Xilinx. El objetivo de estas placas era dar soporte a operaciones de procesamiento a nivel de bit en aplicaciones de alto coste computacional. Una mejora posterior, **Splash2**, empleaba dispositivos más modernos de la familia XC4000 de Xilinx. Esta misma filosofía de conexión a estaciones de trabajo fue empleada en el desarrollo de la plataforma **DECPeRLE-1** [65] basada en 16 FPGAs Xilinx XC3090 conectadas a 4 buses los cuales a su vez estaban conectados a otras FPGAs, del mismo tipo, empleadas como *data switches*.

Por otro lado, también se desarrollaron sistemas completos como la plataforma **PRISM-I** (Processor Reconguration through InstructionSet Metamorphosis) [66] donde la plataforma hardware de pruebas consiste en una placa con un microprocesador Motorola 68010 a 10 MHz, una segunda placa con 4 FPGAs Xilinx 3090 y un bus de 16 bits para conectar las dos placas. El microprocesador implementa las funciones estándar y las partes que demandan más computo son identificadas y ejecutadas en hardware. Otro ejemplo similar a **PRISM-I** es la plataforma **Labomat3** [47], que también emplea un procesador Motorola MC68360 al que se encuentran conectadas dos FPGAs de Xilinx, una XC4013E y una XC6216. Esta plataforma se desarrolló con propósitos docentes en el área del codiseño hardware/software.

2.3.2. Coprocesadores

Los procesadores de propósito general para los modernos computadores de escritorio dedican mucha más área del silicio a aceleradores específicos que al propio procesador [62]. Esto ha desenfocado en una mayor diferencia entre los microprocesadores convencionales y los sistemas reconfigurables, lo cual ha incrementado la investigación en microprocesadores reconfigurables, y además, ha dado lugar a la comercialización de productos relacionados de forma satisfactoria.

Los microprocesadores reconfigurables pueden ser configurados para una tarea computacional específica después de que el hardware físico haya sido diseñado, por ejemplo, reconfigurar de forma adaptativa su set de instrucciones. El trabajo en procesadores reconfigurables se encuentra en una primera fase de investigación, con cambios importantes sin solucionar [67], especialmente en

mejorar los modelos híbridos de programación CPU/FPGA [68]. Sin embargo, existen dominios de aplicación prometedores para los procesadores reconfigurables, como comunicaciones, multimedia y criptografía [69].

En esta configuración, la lógica reconfigurable es parte del procesador y se encuentra localizada cerca del mismo. Tanto **HARP** [70] como **Garp** [71, 72] pueden ser considerados como coprocesadores reconfigurables. La plataforma **HARP** consiste en un procesador RISC (Reduced Instruction Set Computer) de 32 bits (un T805), con 4 MB de DRAM, que tiene acoplada una FPGA Xilinx XC3195A con su propia memoria local. En la arquitectura **Garp** se combina un microprocesador MIPS con lógica reconfigurable en el mismo silicio, lo que permite la reprogramación parcial mediante un set de instrucciones MIPS extendido. **Spyder** y **RENCO** son otros dos ejemplos de sistemas procesador con una parte coprocesador reconfigurable [73].

2.3.3. Unidades funcionales reconfigurables

En esta clase de sistemas, la lógica reconfigurable se encuentra localizada dentro del procesador y el procesador trata a la RFU como una unidad estándar más del camino de datos. La mayor parte de la investigación en procesadores reconfigurables se ha concentrado en las unidades funcionales reconfigurables (RFUs), donde el decodificador de instrucciones maneja instrucciones para la unidad reconfigurable como si fuera una de las unidades estándar del procesador [61].

El **Procesador Nano** reconfigurable [74] fue implementado en una FPGA Xilinx de la familia 3000. Un conjunto de instrucciones a medida fueron desarrolladas para una aplicación específica, y el procesador de control fue implementado dentro de la FPGA en vez de emplear un procesador estándar. El procesador reconfigurable **DISC** (Dynamic Instruction Set Computer) [75, 76] tenía un set de instrucciones que se podía modificar, y era implementado mediante una reconfiguración a la demanda. Un inconveniente de este esquema es la sobrecarga causada por la continua reconfiguración de los módulos. La arquitectura **MorphoSys** [77] tenía como objetivo las aplicaciones de computación intensiva y paralelizable, y combinaba un procesador de control TinyRISC con un array de celdas reconfigurables. El modelo de ejecución estaba basado en el particionado de aplicaciones en tareas paralelas y secuenciales. El procesador reconfigurable **OneChip** [78, 79, 80] integraba una unidad funcional reconfigurable en el pipeline de un procesador RISC con soporte para el uso de múltiples instrucciones simultáneamente y ejecuciones fuera de orden. El procesador reconfigurable **Chimaera** [63] habilitaba un conjunto de instrucciones multi-operando a medida que garantizaban el acceso directo a los registros del procesador principal. La necesidad de un sistema operativo se tuvo en cuenta desde el inicio de la arquitectura **Proteus** [81], cuyo objetivo era extender un procesador ARM con unidades funcionales reconfigurables. El llamado paradigma del procesador polimórfico, que permite al programador modificar la funcionalidad de procesador

y el hardware sin modificaciones de la arquitectura y diseño, es el objetivo de la investigación en el proyecto de procesador reconfigurable **MOLEN** [82]. En esa misma línea se propone el desarrollo de procesadores con instrucciones variables para FPGAs conocidos como **FIPs** (Flexible Instrucciones Processors) [83, 84]. El FIP es un esqueleto de procesador y un conjunto de parámetros que permiten obtener diferentes implementaciones de procesador según la aplicación. En [84] se propone además la aplicación de la reconfiguración en tiempo real para modificar sus características, donde la adaptabilidad permite una evolución automática y el refinamiento del sistema para mejorar sus condiciones en tiempo de ejecución.

SoC y lógica reconfigurable El desarrollo de sistemas basados en unidades funcionales reconfigurables ha llevado al desarrollo de sistemas SoC que integran secciones reconfigurables. Estos sistemas intentan aprovechar las ventajas de la tecnología ASIC y las ventajas de la lógica reconfigurable. Un ejemplo de este tipo de sistemas es el **FIPSOC** [85] que integraba en el mismo chip un microcontrolador 8051 y una área reconfigurable con módulos analógicos. Lo más destacado de este sistema es el soporte de reconfiguración dinámica mediante la posibilidad de disponer de dos contextos en la memoria de configuración, de modo que mientras uno se encuentra activo, el otro se puede modificar. Mediante cambios de contexto se puede modificar la funcionalidad de la lógica reconfigurable desde el procesador principal. El procesador **Pleiades** [86] es otro ejemplo de este tipo donde se combina un microprocesador con un conjunto de unidades programables. El chip **Maia** [87] integra en el mismo chip un procesador ARM8 con 21 unidades especializadas para el procesamiento de comunicaciones a alta velocidad, ALUs, generadores de direcciones, memorias embebidas y un core FPGA de bajo consumo. Otro sistema muy relacionado con **Maia** es **DReAM** [88, 89], un sistema dinámica y parcialmente configurable formado por módulos denominados RPUs (Reconfigurable Processing Units). La principal función de los RPUs es la manipulación de datos para las secciones de procesamiento digital de señal. Finalmente, la arquitectura **XPP** [90, 91], al igual que en los dos casos anteriores, consiste de una matriz reconfigurable para ser integrada en SoC.

2.4. Configurable System-on-Chip

A finales de los 90 los fabricantes de FPGAs comenzaron a introducir microprocesadores en las FPGAs para el diseño de SoC. Algunos dispositivos incluyen uno o más microprocesadores hard-core implementados directamente en el chip, junto a eficientes mecanismos para la comunicación entre el microprocesador y la lógica reconfigurable de la FPGA. Atmel y Triscend [10, 11] fueron los primeros en disponer de estos dispositivos. Más recientemente, Altera desarrolló los dispositivos Excalibur empleando un procesador ARM9 en una FPGA de un millón de puertas

[13] y Xilinx ofrece el dispositivo Virtex-II Pro que incorpora uno o más procesadores PowerPC en FPGAs con decenas de millones de puertas [12].

Mientras las plataformas microprocesador hard-core/FPGA ofrecían un excelente encapsulado y ventajas de comunicación, se comenzó a emplear soluciones basadas en soft-cores, es decir, microprocesadores que los diseñadores pueden implementar usando FPGAs estándar. A estos procesadores se los conoce como "Soft-Core Processors" (SCPs) y ofrecen principalmente las ventajas de la flexibilidad y bajo coste. Muchos fabricantes de FPGAs ofrecen este tipo de diseños. Altera dispone del procesador NIOS y el más reciente NIOS II [6]. Xilinx, por su parte, presenta los procesadores PicoBlaze y MicroBlaze [7]. Paralelamente a las SCPs comerciales, también se encuentran disponibles algunos de libre distribución que son ampliamente utilizados, como LEON de Gaisler Research [8] y OpenRISC de OpenCores.org [9].

Los SCPs ofrecen a los diseñadores una enorme flexibilidad durante el proceso de diseño, permitiendo configurar el procesador para adaptarlo a lo que necesitan en sus sistemas y rápidamente integrar el procesador en cualquier FPGA. Por ejemplo, es posible añadir instrucciones a medida o incluir/eliminar unidades operacionales de la arquitectura del procesador. Al contrario que las FPGAs que disponen de microprocesadores hard-core, los procesadores soft-core permiten incorporar un número variable de procesadores en una misma FPGA dependiendo de las necesidades de la aplicación. Mientras que algunos sistemas embebidos requieren de unos pocos procesadores, otros diseños pueden incluir hasta 64 procesadores [92].

Desafortunadamente, los procesadores soft-core implementados en FPGAs presentan ciertas desventajas como un alto consumo y un rendimiento inferior comparado con los procesadores hard-core. Para reducir la relación entre el rendimiento y el consumo, un diseñador puede emplear el particionamiento hardware/software. Este particionamiento consiste en dividir una aplicación que inicialmente se ejecuta completamente en software, en un microprocesador, en coprocesadores hardware. Algunos trabajos realizados sobre MicroBlaze [93] han demostrado que puede ser comparable y competitivo con los microprocesadores hard-core existentes, mientras mantiene toda las ventajas de la flexibilidad asociada a los procesadores soft-core.

2.5. Sistemas auto-reconfigurables basados en FPGAs

El concepto de reconfiguración dinámica se aplica a los dispositivos capaces de mantener su funcionamiento sin interrupción mientras ciertas partes del dispositivo están siendo configuradas [94]. Este tipo de dispositivos se denominan reconfigurables dinámicamente para diferenciarlos de los dispositivos reconfigurables que no soportan un cambio de configuración en tiempo de ejecución. Por otro lado, el concepto de auto-reconfiguración se define como la capacidad de un

dispositivo para generar en tiempo de funcionamiento los bits de configuración y usarlos para modificar su propia configuración [29, 30]. De estas definiciones se deduce que un sistema auto-reconfigurable tiene al menos dos partes, una sección fija con la lógica necesaria para determinar la siguiente configuración aplicable, y una o más partes, sobre las que se aplica la reconfiguración parcial dinámica, para realizar las diferentes tareas de la aplicación.

La auto-reconfiguración requiere una serie de elementos que hagan posible su aplicación. En [95] se propone una infraestructura teórica para la gestión de la reconfiguración dinámica en tiempo de ejecución en diseños reconfigurables. En esta infraestructura existe un gestor de reconfiguración formado por tres módulos: un monitor, un cargador y un elemento para el almacenamiento de la configuración. El monitor dispone de la información sobre el estado de configuración y cuando se requiere un cambio de configuración, bien porque se ha recibido una solicitud desde la aplicación o bien desde la FPGA, el monitor indicará al cargador que instale un nuevo circuito en una localización determinada de la FPGA. El flujo de diseño para la generación de los controladores de reconfiguración está asistido por un conjunto de herramientas propias [96] y soportado por un modelo de componentes reconfigurables. El modelo propuesto [97] es un ejemplo representativo de abstracción teórica dentro del área de la lógica reconfigurable.

2.5.1. Reconfiguración parcial dinámica

El objetivo más importante en los sistemas diseñados con reconfiguración parcial es maximizar la cantidad de circuito estático que permanece sin cambiar cuando se reconfigura el dispositivo. Esto se lleva a cabo particionando la aplicación en bloques funcionales que son, las mayor parte de las veces, comunes a todas las configuraciones [98]. Este es el primer paso en el desarrollo de sistemas reconfigurables parcialmente, el segundo paso es mapear físicamente las configuraciones en el dispositivo.

Los sistemas reconfigurables parcialmente mapean muchos circuitos para compartir los recursos del dispositivo, cada uno de los cuales puede ser descompuestos en secuencias de mapeo uno a uno. La visualización de la configuración particionada, los efectos de la sobrecarga de la reconfiguración, la superposición espacial entre los bloques en diferentes configuraciones y la ejecución de los diseños, y la planificación de la configuración han sido intentados por la herramienta **DYNASTY** [99].

La modificación dinámica de circuitos implementados sobre lógica reconfigurable se comenzó a estudiar de forma intensiva en los años finales de la década de los 90, y particularmente se ha trabajado en metodologías basadas en entornos desarrollados en lenguaje de programación Java. **JHDL** ("Just another Hardware Description Language") [100] fue de los primeros intentos de herramientas de diseño para sistemas de reconfiguración parcial que incluía un programa supervisor

para controlar la plataforma. El objetivo era integrar un control de supervisión con la descripción del circuito, y permitir a los diseñadores la organización de manera rápida de los cambios dinámicamente, usando abstracciones de programación estándar encontradas en los lenguajes orientados a objetos. **JBits** es otra herramienta que da soporte a la reconfiguración parcial, basada en Java [101, 102]. Desarrollada por Xilinx, se trata de un conjunto de clases Java que proveen un API (Application Program Interface) para el manejo del *bitstream* de las FPGAs de Xilinx. La interfaz permite controlar individualmente, desde el software, todos los recursos configurables.

Al margen del lenguaje Java, también han aparecido otras herramientas como **PARBIT** (PARtial BITle Transformer) [103]. Esta herramienta permite la transformación del *bitstream* de configuración en *bitstreams* parciales mediante la lectura de los *frames* de configuración del *bitstream* original, copiando solo los bits de configuración relacionados con el área definida por el usuario al *bitstream* parcial.

Por otro lado, el conexionado es otro de los elementos más complejos que se abordan en el área de la reconfiguración parcial dinámica. Las herramientas de los fabricantes ofrecen mecanismos para especificar la localización de los elementos lógicos de la matriz reconfigurable, pero no proveen de métodos para guiar el rutado, de forma que una determinada ruta que conecte una sección fija con una sección dinámica siempre utilice la misma línea en cualquiera de los módulos configurables en esa sección. Para solucionar esa problemática, algunas investigaciones [104] proponen modelos de conexión para el rutado de unidades reconfigurables orientadas a distintos tipos de computación, paralela y secuencial con procesador, o de forma más general a cores. Estos cores especiales denominados *Sitcher cores* presentan únicamente recursos de rutado, de modo que se puede abstraer el conexionado real del dispositivo. Soluciones similares se plantearon en años posteriores, como las *bus macro* de Xilinx [20], para interconectar cores dinámicamente.

Finalmente, uno de los mayores inconvenientes de la reconfiguración dinámica es el tiempo necesario para la reconfiguración, que en muchas aplicaciones hace que su aplicación sea inviable [105, 106]. Por ello, la modificación parcial dinámica supone uno de los avances más prometedores en el área de los sistemas reconfigurables.

Reconfiguración dinámica mediante JBits

El API JBits de Xilinx se desarrolló como una herramienta exclusivamente de investigación que ha dado lugar una gran parte de las investigaciones relacionadas con la reconfiguración parcial dinámica en FPGAs. JBits da soporte para reconfiguración en tiempo de ejecución (-RTR- Run-Time-Reconfiguration) a los dispositivos Virtex y Virtex-II de Xilinx, siendo una evolución del sistema JERC6K [107] desarrollado para la antigua familia de dispositivos XC6200, y posteriormente ampliado a la familia XC4000.

JBits permite el acceso a los datos de configuración del dispositivo, lo que hace posible la modificación en tiempo de ejecución tanto de la lógica como del rutado. El modelo de programación usado por JBits se basa en un array de dos dimensiones de CLBs (Congurable Logic Blocks), la unidad de programación básica de los dispositivos Xilinx, y del resto de bloque funcionales disponibles como memorias y multiplicadores, estos últimos disponibles a partir de la familia Virtex-II. JBits puede manejar estos bloque funcionales, pudiendo replicarlos, implementarlos condicionalmente, etc. Los detalles sobre el manejo de los recursos, llamado sistema XBI se exponen en [108]. Desde JBits es abordable la construcción e instanciación de *Run-Time Parametrizable Cores* [109], de forma que se puedan realizar diseños con cores parametrizables en tiempo de funcionamiento. Del mismo modo, para cada conjunto de CLBs se asignan un grupo de elementos de rutado para realizar las conexiones. Pero la labor de interconectar esos cores de forma dinámica no estaba resuelta en las primeras versiones de JBits, hasta la llegada de JRoute [110].

El sistema JBits inicialmente ha estado restringido por la falta de mecanismos para el diseño de diferentes tipos de circuitos, especialmente circuitos de control, máquinas de estado, etc. [24]. El flujo de diseño original mediante JBits se ha ido modificando con el fin de integrar diseños de circuitos desarrollados mediante herramientas de síntesis convencionales basadas en lenguajes de descripción hardware. De esta forma se ampliaban los tipos de circuitos que se podían diseñar con soporte para reconfiguración dinámica.

A partir de JBits se han desarrollado otras herramientas interesantes como los *Run-Time Parametrizable Cores*, mencionados anteriormente, que permite crear cores en tiempo de ejecución y emplearlos dinámicamente para modificar el circuito existente, JRTR [111], que añade un soporte sencillo y directo para la reconfiguración parcial que anteriormente no estaba disponible en JBits, y JBitsDiff [112] que permite la extracción de información del circuito directamente del *bitstream* de configuración y producir cores pre-rutados y pre-emplazados para su uso en tiempo de ejecución.

Reconfiguración dinámica mediante herramientas estándar

El uso de metodologías basadas en las herramientas estándar para la manipulación directa del *bitstream,* permite aprovechar que se encuentran más preparadas para optimizar los circuitos, y además, son preferidas en general por los diseñadores ya que el diseño inicial puede partir de descripciones en lenguajes HDL. Los ejemplos más significativos de estas metodologías son los dos flujos de diseño para reconfiguración parcial dinámica propuestos por Xilinx [20]. Ambos flujos se basan en la utilización de sus herramientas para diseños convencionales. En situaciones donde solo se requiere cambiar una pequeña sección del hardware para modificar la funcionalidad, y por tanto el diseño debe estar preparado para la reconfiguración parcial, se pueden realizar los cambios

directamente en el fichero que agrupa la información de emplazamiento y rutado de los elementos. Esta tipo de modificación se corresponde con el primer flujo de diseño, y se denomina *Partial-Based*. Para llevar a cabo estos pequeños cambios se emplea el programa FPGA Editor, el cual muestra todos los elementos configurables de la FPGA de forma gráfica. Para cada contexto se obtiene un fichero modificado con pequeñas variaciones respecto al original. El programa que genera los *bitstreams* de configuración, *bitgen*, es capaz de generar *bitstreams* parciales únicamente con los cambios que se deben aplicar para configurar los diferentes contextos en base a las diferencias entre los ficheros modificados y el original.

Cuando las modificaciones que se deben realizar mediante la reconfiguración parcial dinámica son importantes, por ejemplo cuando queremos reconfigurar partes del diseño, Xilinx propone el segundo flujo de diseño, denominado *Module-Based*. En este caso, aunque las herramientas son las mismas, la forma de obtener los *bitstreams* es muy diferente. Es necesario realizar el emplazamiento y rutado de cada sección, definidas como dinámicas y estáticas, de la FPGA por separado y obtener los correspondientes *bitstreams* parciales además de un *bitstream* total para la carga inicial. Para mantener la integridad de las conexiones entre la sección estática y las dinámicas, y la del resto de módulos dinámicos que puedan ser cargados en una sección dinámica, se requiere cumplir unas pautas muy estrictas en el flujo de diseño que incluyen la utilización de elementos pre-rutados denominados *bus macro* [20], cuidar la distribución de los pines de salida y tener en cuenta las limitaciones impuestas a las señales globales, entre otras.

2.5.2. Reconfiguración parcial dinámica de cores

El uso de cores reusables en sistemas digitales [113] proviene del mundo de los ASICs. Los cores para ASICs suelen disponer de registros internos de configuración que permiten cambiar la funcionalidad de los mismos, de forma que se dota de flexibilidad a los diferentes módulos y de ese modo se pueden cubrir diferentes ámbitos de aplicación. Con el incremento de los recursos lógicos en las FPGAs, el diseño basado en cores se ha extendido a estos dispositivos permitiendo que puedan diseñarse en ellos sistemas digitales completos no solo basados en un único tipo de circuitos, sino que pueden incluir varios procesadores [114]. Sin embargo, esta característica que se muestra claramente beneficiosa para los ASICs, presenta en las FPGAs un serio problema: se considera redundante el hecho de que la FPGA pueda modificar el diseño interno vía reconfiguración, y que los cores dispongan de varios modos de funcionamiento seleccionables mediante registros, lo que supone que existan partes inactivas en el circuito. Además, en el caso de la FPGA, el coste en recursos de silicio en mayor que en los ASICs [115] lo que hace razonable que se busquen soluciones a esta redundancia. Una de las posibles soluciones se ha presentado anteriormente, los *Run-Time Parametrizable Cores*, como sustitución de las descripciones para ASICs en

los dispositivos con capacidad de reconfiguración dinámica.

Como aplicación práctica de la reconfiguración de cores cabe destacar los trabajos presentados en [21, 24]. En estos trabajos se combinan las diferentes técnicas presentadas anteriormente para la modificación parcial de circuitos en una aplicación real sobre dispositivos Virtex [21], que consiste en un sistema de decodificación de audio realizado mediante una FPGA reconfigurable dinámicamente. Para ello se definen en la FPGA una sección fija, en la que se implementa un microprocesador LEON [8] y otra sección modificable de forma dinámica. La aplicación consiste en que el microprocesador reciba la secuencia de audio a través de la red local y la envíe al core coprocesador de la sección dinámica para realizar la decodificación del *stream*. Dependiendo del tipo de formato de audio, el coprocesador de la sección dinámica es reconfigurado dinámicamente. Para llevar a cabo el diseño de la aplicación se emplea un flujo mixto [24], basado en las herramientas estándar de diseño y JBits. En un primer paso se crean dos diseños completos con el mismo microprocesador y diferente coprocesador mediante las herramientas convencionales. Empleando la herramienta JBitscopy se extrae el core coprocesador del segundo diseño y se integra en el *bitstream* completo del primer diseño. A partir de esta configuración, JBits puede generar un *bitstream* parcial que solo contiene el área del coprocesador y que posteriormente se puede aplicar a esa misma área del diseño. Para restringir los recursos de rutado, en las conexiones entre las secciones estáticas y dinámicas se emplean interfaces localizadas en posiciones fijas basadas en 2 CLBs sin operación lógica definida denominadas *feed-through*. Cada una de estas interfaces se establece en una posición específica de modo que se dispone de un cierto control del rutado, siguiendo la misma idea que el *bus macro* de Xilinx. Lo más destacado de esta metodología es que permite beneficiarse del nivel de abstracción utilizado en JBits para una manipulación de los *bitstreams* más comprensible [21], y al mismo tiempo, evitar las dificultades que supone emplear JBits para el diseño del sistema, de modo que se puedan emplear las herramientas estándar de diseño más preparadas para diseñar con facilidad y optimizar los recursos lógicos y de rutado.

De forma paralela a estas investigaciones, se han producido avances en el área de la reconfiguración dinámica de módulos hardware para comunicaciones desarrollados sobre la plataforma **Field Programmable Port eXtender (FPX)** [26, 25], orientada al diseño rápido de prototipos de routers [27] y firewalls [28]. Estos módulos son cores reconfigurables dinámicamente y se denominan *Dynamic Hardware Plugins (DHPs)* [116]. Al igual que en la metodología presentada anteriormente en [24], el *bitstream* global se extrae de la configuración de la zona de la matriz lógica que contiene el DHP mediante una herramienta específica denominada PARBIT [117]. Para solucionar la problemática del rutado se propone el uso de unos puntos fijos de interconexión para conectar los DHPs a la infraestructura lógica de la FPGA denominados *Gasket Interfaces* [118]. De esta forma se mantiene el mismo rutado para la interconexión con la sección estática del diseño cuando se realiza el intercambio.

2.5.3. Auto-reconfiguración en plataformas FPGA

La Auto-Reconfiguración Parcial Dinámica en FPGAs se comenzó a desarrollar en la década de los 90 principalmente con la aparición de los primeros dispositivos que la soportaban, las FPGAs de la familia XC6200 [17] de Xilinx. Uno de los primeros sistemas en emplear estos dispositivos fue la arquitectura **URISC** [18], en la que se empleaba un procesador Ultimate-RISC [119] como controlador de un sistema coprocesador FPGA dinámicamente reconfigurable. El URISC presentaba una arquitectura mínima de procesamiento con una única instrucción muy simple: "mover de memoria a memoria". Con esta única instrucción el proceso de computación se realizaba mediante el movimiento de datos entre los diferentes módulos conectados al bus del sistema. Estos módulos, denominados SPL (Swappable Logic Units), se reconfiguraban para cargar diferentes unidades de procesamiento. En el mapa de memoria del sistema se incluía, además de la memoria de datos y programa, la memoria de configuración de la FPGA, lo que dotaba a esta arquitectura de la capacidad de auto-reconfiguración. También empleando estos mismos dispositivos se propone en [94] un sistema auto-reconfigurable, llamado **"Self controlling DRL system"**, que además ofrece uno de los primeros ejemplos de aplicación auto-reconfigurable sobre lógica reconfigurable dinámicamente con un sistema modularizado.

Ya en la actualidad, las plataformas FPGA comerciales para el diseño de SoC están obteniendo un enorme éxito [5]. Los diseños realizados con estas plataformas implementados sobre dispositivos reconfigurables se benefician de la capacidad de configuración de los mismos, básicamente en la etapa de diseño y en posteriores actualizaciones. Es por ello que se están desarrollando y aplicando flujos de diseño específicos [20] para la aplicación de la reconfiguración parcial dinámica a cores con interesantes y prometedores resultados prácticos [23, 22]. Además, en los últimos años los fabricantes de dispositivos reconfigurables han incluido notables mejoras respecto a las capacidades de reconfiguración parcial y dinámica, siendo las posibilidades cada vez más prometedoras en este campo. Destacan la integración de sistemas de acceso a los puertos de configuración desde el interior de los propios dispositivos como el módulo ICAP de Xilinx [16] para dispositivos Virtex-II, o el control integrado de la reconfiguración dinámica del FPSLIC de Atmel [120].

Para poder hacer uso de estas posibilidades en los diseños actuales basados en cores se requieren metodologías, entornos y herramientas que faciliten el uso de estas posibilidades. En este campo concreto caben destacar los trabajos sobre el desarrollo de la plataforma de auto-reconfiguración **"Self-Reconguring Platform (SPR)"** [16, 35]. SPR es una plataforma para dispositivos Virtex-II y Virtex-II Pro que permite controlar de forma dinámica la reconfiguración de la FPGA bajo el control del microprocesador embebido en el CSoC. En este caso, la reconfiguración y el procesamiento se realizan mediante el procesador incluido en el mismo integrado que la lógica configurada. El hardware de control de la reconfiguración está formado por el componente ICAP y un

bloque de memoria RAM (BlockRAM) que se emplea como caché de datos de configuración. En cuanto al software, éste se encuentra divido en capas, de forma que se puedan diferenciar niveles dependientes del hardware y no dependientes del mismo. Este software se engloba en lo que se conoce como el API XPART (Xilinx Partial Reconfiguration Toolkit) [16] que provee métodos para transferir información entre la memoria de configuración y la caché, y accesos a ésta. El conjunto de herramientas XPART define métodos para realizar la modificación dinámica de recursos a través del ICAP por lo que se habilita la posibilidad de desarrollar sistemas auto-reconfigurables totalmente integrados.

Aunque este avanzado sistema de aplicación de auto-reconfiguración aún se encuentra en fase de desarrollo, se está comenzando a aplicar en diseños industriales. Un ejemplo representativo es el router reconfigurable dinámicamente para redes ópticas presentado en [121]. En [22] también se propone un *framework* para la aplicación de la auto-reconfiguración a diseños implementados en FPGAs Virtex-II de Xilinx, utilizando la interfaz ICAP y los flujos de diseño propuestos por Xilinx. El prototipo incluye soporte para una interfaz de comunicación serie con un host externo y un módulo criptográfico Blowfish. Los resultados obtenidos a partir de este trabajo demuestran la viabilidad de la tecnología de reconfiguración parcial dinámica, ofreciendo unos tiempos de modificación de un 13 % del área de la FPGA en unos 5 segundos. Este tiempo incluye la recepción de los datos de configuración a través de una canal lento de comunicación serie, procesamiento de seguridad, verificación y aplicación de los mismos.

Más recientemente, se ha presentado un sistema que también explota las posibilidades de la reconfiguración parcial dinámica de los dispositivos Virtex de Xilinx aplicada a la electrónica de automoción [31]. Además se incluyen elementos de prioridad dinámica como una primera aproximación hacia soluciones para toma de decisiones adaptativas relativas a la reconfiguración. La reconfiguración propuesta parte de la premisa de que las distintas funciones de los cores del sistema electrónico del automóvil no necesitan estar simultáneamente activas, siendo posible definir un subconjunto que se carguen en el dispositivo reconfigurable aplicando una estrategia de multiplexado en el tiempo bajo demanda [32]. La FPGA reconfigurable parcial y dinámicamente incluye un controlador de reconfiguración implementado, al igual que en la plataforma SRP [16, 35], mediante el procesador MicroBlaze. Una aportación destacable de esta propuesta es la capacidad del sistema de guardar el estado y diversos parámetros de una función cuando es sustituida por otra. El mecanismo de salvaguarda es administrado por el controlador enviándose los datos de cambios de contexto a través del bus bajo el control del arbitrador. La comunicación entre los distintos módulos-función reconfigurables se realiza mediante un sistema Network-on-Chip (NoC) basado en comunicación serie [122] y los módulos reconfigurables acceden a las líneas de comunicación de la NoC a través de las *bus macro* de Xilinx.

Otra propuesta realizada para el control de la reconfiguración parcial dinámica aplicada a dise-

ños basados en cores en FPGAs es la denominada **Fixed core Processor connected to Reconfigurable Coprocesor (FiPRe)** [33, 34]. Esta propuesta incluye la implementación de coprocesadores reconfigurables, donde estos coprocesadores se encargan de acelerar tareas específicas, siendo posible su inserción o sustitución en tiempo de funcionamiento del sistema. Además, para facilitar el diseño de software, incluyen el core de un microprocesador que dispone de un conjunto de instrucciones fijo. Este microprocesador, denominado R8R, está derivado del R8 [123] al cual se le han añadido cinco instrucciones para controlar los coprocesadores reconfigurables. Además de utilizar las herramientas de diseñó convencionales que oferta este fabricante, emplean herramientas auxiliares específicamente diseñadas por el grupo de investigación [36], como ocurre en la mayoría de los sistemas presentados. La sección no reconfigurable está compuesta por el procesador R8R junto con un Controlador de Configuración (CC) implementado en hardware. Este controlador es un módulo esclavo del host externo para las operaciones de transferencia de configuraciones desde éste y del R8R que lo emplea para acceder a la interfaz de configuración interna ICAP [16].

Para finalizar el estado del arte de los sistemas reconfigurables, es necesario mencionar las aportaciones más recientes en esta área relativas a sistemas operativos para hardware reconfigurable [37]. En esta línea, se presenta en [38, 39] una infraestructura de bus TCB (Task Communication Bus), señales y lógica para que un sistema operativo de hardware reconfigurable pueda gestionar múltiples tareas hardware en una FPGA Virtex-II de Xilinx. El sistema operativo se ejecuta en un procesador embebido en la propia FPGA, disponiendo de elementos similares a los utilizados en los sistemas operativos de tiempo real para sincronización como son FIFOs, semáforos, temporizadores, colas de mensajes, etc. La infraestructura propuesta se encuentra orientada a cumplir con los requisitos tecnológicos impuestos por la tecnología Virtex de Xilinx para soportar la reconfiguración parcial dinámica: reconfiguración en una dimensión (por columnas) [40] y uso de *bus macro* para interconectar las rutas que unen secciones reconfigurables. También en la misma línea de sistemas operativos con soporte para tareas hardware hay que destacar el modelo presentado en [41]. Aunque la reconfiguración parcial dinámica todavía no ha sido implementada usando este modelo, esta propuesta conjuga elementos claves en la evolución de los sistemas CSoC como son el uso de interfaces estándar y sistemas operativos de tiempo real ejecutándose en procesadores embebidos en el mismo dispositivo.

2.6. Criptografía en FPGAs

Con la aparición de las FPGAs se ha facilitado la investigación en el diseño de sistemas criptográficos, y el estado de arte actual de la implementación de estos algoritmos indica que se está alcanzando su madurez [44]. Los diseños en FPGA requieren de optimizaciones a nivel de arqui-

tectura y de algoritmo [124], lo que significa que desarrollar un sistema criptográfico sobre estos dispositivos necesita de amplios y efectivos conocimientos prácticos y teóricos en el diseño con hardware reconfigurable.

2.6.1. Criptografía

La criptografía envuelve el estudio de las técnicas matemáticas que permiten alcanzar los objetivos para proveer los servicios resumidos a continuación [125]:

Confidencialidad Es un servicio usado para mantener el contenido de la información solo accesible a quienes tengan autorización para ello. Este servicio incluye tanto la protección de los datos trasmitidos entre dos puntos durante un periodo de tiempo como la protección del flujo de tráfico frente al análisis.

Integridad Es un servicio que requiere que el sistema computador activo y la información transmitida puedan ser modificados solo por los usuarios autorizados. Las modificaciones incluyen escrituras, cambios, cambios de estado, eliminaciones, creación, y el retardo o retransmisión de los paquetes.

Autenticación Es un servicio que están relacionado con asegurar que el origen del mensaje está correctamente identificado, es decir, la información repartida en un canal debe ser autentificada por el origen, fecha de origen, contenido de los datos, tiempo de envío, etc. Por estas razones este servicio se divide en dos clases: autentificación de la entidad y autentificación del origen de los datos. Se debe tener en cuenta que la segunda clase de autentificación provee de forma implícita la integridad de los datos.

Confianza Este es un servicio que previene tanto al emisor como al receptor de una transmisión de la denegación de compromisos o acciones previas.

Existen dos clases de algoritmos en criptografía: algoritmos de clave privada, o de clave simétrica, y algoritmos de clave pública. Como complemento a los algoritmo de cifrado, en muchos sistemas se emplean además los algoritmos de hash, principalmente, para la integridad de los datos.

2.6.2. Algoritmos de cifrado

En el mundo de la criptografía, una clave es un valor numérico generado mediante computación que los algoritmos de cifrado usan para el cifrado y descifrado de datos. Al no ser la clave de longitud fija, cuanto mayor sea ésta mayor seguridad provee. Existen dos tipos de claves [126,

127], privada y pública, por lo que también existen dos métodos de cifrado: de clave privada y de clave pública.

La criptografía de clave privada fue el primero de los métodos criptográficos basados en clave. También se la conoce como criptografía simétrica porque se emplea la misma clave para el procedimiento de cifrado y de descifrado. La clave debe ser compartida por los diferentes participantes de la operación, de modo que en el origen se puedan cifrar los datos mediante la clave propuesta, y en el destino se puedan descifrar los datos con la misma clave. El problema de este método es que si la clave es conocida por otras personas, entonces tendrán acceso a todos los datos.

La criptografía de clave pública usa un mecanismo diferente. También se la conoce como criptografía asimétrica porque se emplean diferentes claves para el cifrado y descifrado. Con la criptografía de clave pública, existen dos claves disponibles, la clave privada que solo el usuario conoce, y la clave pública a la que todo el mundo tiene acceso. Ambas claves son generadas al mismo tiempo empleando el mismo algoritmo. Con este sistema si alguien quiere enviar un mensaje, lo cifra con la clave pública del destinatario, y entonces solo éste es capaz de descifrar el mensaje con su clave privada. Las claves privadas nunca son intercambiadas o enviadas a través de la red. Debido a que la clave privada permite la identificación de su propietario, las firmas digitales se basan en criptografía de clave pública.

A lo largo de esta tesis se emplean algoritmos de clave privada y algoritmos de hash, y todos ellos han sido y son ampliamente implementados sobre diferentes sistemas, tanto reconfigurables como basados en microprocesadores de propósito general, ya que son parte fundamental de la mayoría de aplicaciones de seguridad. A continuación se realiza un descripción detalla de cada uno de ellos.

Algoritmos de clave privada

Los algoritmos de clave privada o simétrica son comúnmente divididos en cifradores de bloques y cifradores de flujo [126]. En los cifradores de bloque el mensaje es dividido en cadenas, llamadas bloques, de longitud fija y se cifra un bloque cada vez. Ejemplos de cifradores de bloque son DES (Data Encryption Standard) [128], IDEA (International Encryption Standard) [129], y AES (Advanced Encryption Standard) [130]. Los cifradores de flujo operan con un único bit de texto sin cifrar al tiempo. En cierto modo, son cifradores de bloque que operan sobre bloques de longitud igual a uno. Son útiles porque la transformación de cifrado puede cambiar para cada símbolo del mensaje a ser cifrado. En particular, son muy útiles en situaciones donde los errores de transmisión son muy probables, porque no propagan los errores. Además, pueden ser usados cuando los datos son procesados símbolo a símbolo debido a la falta de memoria del sistema o que el buffer es limitado. Un ejemplo de este tipo de algoritmos es RC4 [126].

Es importante destacar que el objetivo al diseñar los modernos cifradores de clave simétrica actuales, como AES, ha sido optimizar los algoritmos para una implementación eficiente tanto en software como en hardware, al contrario que DES, que fue diseñado pensando en una implementación únicamente hardware. Estos criterios de diseño son evidentes si se analiza el rendimiento del algoritmo AES sobre diferentes plataformas. Las operaciones internas del algoritmo AES se pueden dividir en operaciones de 8 bit, lo cual es importante porque la mayoría de los algoritmos de cifrado se ejecutan sobre *smart cards*, que tradicionalmente emplean CPUs de 8 bits. Además, es posible combinar ciertos pasos para obtener un rendimiento aceptable en el caso de plataformas de 32 bits. Al mismo tiempo, las implementaciones de AES en hardware pueden alcanzar fácilmente los Gbits/s cuando se emplean ASICs o FPGAs.

Como detalle final destacar que uno de los mayores problemas de los algoritmos de clave simétrica es la necesidad de encontrar métodos eficientes para el intercambio de claves de forma segura. Esto se conoce como el problema de la distribución de la clave. En 1977, Diffie y Hellman propusieron un nuevo concepto que fue llamado criptografía de clave pública, y que revolucionó la criptografía tal y como se conoce hasta ahora.

Algunos de los algoritmos de clave simétrica más representativos de los empleados en la actualidad son:

IDEA El International Data Encryption Algorithm (IDEA) fue propuesto por Lai y Massey en 1990 [129]. El diseño de IDEA fue basado en el concepto de la mezcla de operaciones de diferentes grupos algebraicos, y opera con tres grupos de operaciones en parejas de subbloques de 16 bits: XOR (exclusive OR) de 2 bloques de 16 bits, suma de enteros módulo 2^{16}, y multiplicación de enteros modulo $2^{16}+1$. De todas las operaciones, la multiplicación es la operación más costosa para implementar en hardware o en software.

En IDEA, tanto el texto a cifrar con el resultado son bloques de 64 bits, y la clave es de 128 bits. El proceso de cifrado y descifrado es esencialmente el mismo, pero solo cambian los diferentes subbloques de la clave (sub-claves). IDEA se hizo popular debido a su uso en el paquete de seguridad de correo electrónico Pretty Good Privacy (PGP), y fue propuesto para ser usado como un *benchmark* general de diseño en dispositivos reconfigurables [131]. Hasta la llegada del algoritmo AES (Advanced Encryption Standard), IDEA se presentaba como el algoritmo de cifrado simétrico más seguro disponible al público [126].

DES y Triple-DES El algoritmo DES [128] está basado en el algoritmo Lucifer [132], diseñado por IBM a principios de los setenta. Después de algunas modificaciones de la NSA (National Security Agency), fue publicado en 1977 por el NBS (National Bureau of Standards), una sección del departamento de defensa de los EEUU. DES utiliza claves de 56 bits y un cifrado de bloques

de 64 bits. El algoritmo DES cifra la información por bloques, es decir, el texto en claro debe ser dividido en bloques de 64 bits que serán cifrados uno tras otro. DES utiliza claves de 56 bits, aunque estas suelen distribuirse en forma de números de 64 bits. De estos 64 bits, uno de cada ocho, es utilizado como bit de paridad (64-8=56). Desde la aparición del algoritmo DES han aparecido algunas críticas sobre el criptosistema. La principal es la longitud de la clave, de solo 56 bits, que hacen posible los ataques de fuerza bruta. Por otra parte, la arbitrariedad de las S-boxes podría suponer la existencia de una clave maestra que permitiese descifrar cualquier mensaje.

Una forma de solucionar el problema de la longitud de la clave es el uso del cifrado triple, conocido como Triple DES [128, 133]. Este algoritmo admite variantes que consisten básicamente en aplicar el algoritmo DES tres veces consecutivas. Es demostrable que esta operación no es equivalente a utilizar una sola vez DES con una clave diferente y por tanto mejora la resistencia a los ataques de fuerza bruta. La clave utilizada por Triple DES es de 128 bits (112 de clave y 16 de paridad), es decir, dos claves de 64 bits (56 de clave y 8 de paridad) de los utilizados en DES. El motivo de utilizar este de tipo de clave es la compatibilidad con DES. Si la clave utilizada es el conjunto de dos claves DES iguales el resultado será el mismo para DES y para Triple DES. Otra forma de utilizar Triple DES es con una clave de 192 bits (168 bits de clave y 24 bits de paridad). En este caso se cifrará primero con k1, a continuación con k2 y finalmente con k3. Para ser compatible con DES es necesario que k1=k2=k3.

Blowfish Blowfish [134] es un algoritmo de cifrado simétrico por bloques diseñado en 1993 por Bruce Schneier como una alternativa rápida a los algoritmos de cifrado actuales. Desde entonces, se ha analizado profundamente y ha ido ganando aceptación como un algoritmo de cifrado potente. Blowfish no está patentado y es gratuito para cualquier utilización. Blowfish utiliza una clave de longitud variable, de 32 bits a 448 bits, y cifra bloques de 64 bits de longitud. El algoritmo consta de dos partes: una parte de expansión de la clave y la otra parte de cifrado de datos. La expansión de la clave convierte una clave de hasta 448 bits en un conjunto de subclaves con un total de 4168 bytes. El cifrado de los datos se realiza mediante la aplicación de una red Feistel de 16 etapas. Cada etapa consiste en una permutación dependiente de la clave, y de una sustitución dependiente de la clave y los datos. Todas las operaciones son XORs y sumas de 32 bits. Las únicas operaciones adicionales son cuatro tablas indexadas por etapa.

Rinjdael Rijndael es un cifrador de bloques diseñado por Joan Daemen y Vincent Rijmen como algoritmo candidato para el AES [135, 130, 136]. El algoritmo presenta una longitud de bloque y clave variables. Actualmente se especifica como usarlo con claves de longitud 128, 192, o 256 bits para cifrar bloques de longitud de 128, 192 o 256 bits (todas las combinaciones de longitud de clave y longitud de bloque son posibles). Tanto la longitud de bloque como la longitud de la clave

pueden ser extendidos fácilmente a múltiplos de 32 bits. Rijndael puede ser implementado muy eficientemente en un amplio rango de procesadores y en hardware.

RC4 El algoritmo RC4 [137, 126] es un algoritmo simétrico de cifrado de flujo desarrollado en 1987 por Ronald Rivest y mantenido en secreto por el RSA Data Security. El 9 de Septiembre de 1994, el algoritmo fue publicado de forma anónima en Internet. RC4 usa una clave de longitud variable, entre 1 y 256 bytes, para inicializar una tabla de estados de 256 bytes. La tabla de estados se emplea para la generación de bytes pseudo-aleatorios que conforman un flujo pseudo-aleatorio al que se le aplica la función XOR con los datos a cifrar para obtener los datos cifrados. Cada elemento de la tabla de estados es intercambiado al menos una vez. La clave del algoritmo RC4 está actualmente limitada a 40 bits debido a restricciones de exportación, pero en algunas ocasiones se emplea una clave de 128 bits. Tiene la capacidad de usar claves entre 1 y 2048 bits. RC4 es usado en muchas aplicaciones comerciales como son Lotus Notes y Oracle Secure SQL. Además, forma parte de la especificación en telefonía celular (WEP).

Algoritmos de hash

Además de los algoritmos criptográficos, los algoritmos de hash se emplean en las aplicaciones de seguridad actuales normalmente para asegurar la integridad de los datos enviados, por lo que además de cifrar los datos, las aplicaciones se aseguran de que no han sufrido cambios durante la transmisión.

Los algoritmos de hash son algoritmos que trasforman los datos en un resumen irreversible. Es decir, si pasamos los datos por un algoritmo hash obtendremos un resumen único de esos datos, pero a partir del resumen no seremos capaces de volver a obtener los datos. Las ventajas de este tipo de algoritmos son la imposibilidad computacional de reconstruir la cadena original a partir del resultado, y también la imposibilidad de encontrar dos cadenas de texto que generen el mismo resultado. Esto nos permite usar el algoritmo para transmitir contraseñas a través de un medio inseguro. Simplemente se cifra la contraseña, y se envía de forma cifrada. En el punto de destino, para comprobar si el *password* es correcto, se cifra de la misma manera y se comparan las formas cifradas.

Las funciones de comprobación aleatoria son similares a las de cifrado, de hecho, algunas de ellas son funciones de cifrado con ligeras modificaciones. La mayoría de estas funciones toma un bloque de datos y lo somete reiteradamente a una sencilla función de desordenación (*scrambling*) para alterar sus elementos. Si esta operación se repite un cierto número de veces, no existe forma práctica conocida de predecir el resultado. Es imposible modificar un documento de un modo determinado y estar seguro de que la función de comprobación aleatoria producirá el mismo resul-

tado. Este tipo de firma utiliza una función de comprobación aleatoria criptográficamente segura, como Message Digest 5 (MD-5) [138] o Secure Hash Algorithm (SHA) [139], para producir un valor de comprobación aleatorio a partir de un archivo. El procedimiento de comprobación aleatoria encadena su clave secreta. El destinatario también tiene una copia de la clave secreta y la utiliza para evaluar la firma.

Algunos de los algoritmos hash más representativos de los empleados en la actualidad son:

MD5 El algoritmo MD5 fue desarrollado por Ronald L. Rivest del MIT [138]. Básicamente se trata de una función de cifrado tipo hash que acepta una cadena de texto como entrada, y devuelve un numero de 128 bits. Se basa en que es computacionalmente imposible que se produzcan dos mensajes que tengan el mismo resultado. El algoritmo MD5 está especialmente diseñado para aplicaciones de firma digital, donde el tamaño del fichero debe ser comprimido de una manera segura antes de ser cifrado usando un clave privada mediante un algoritmo de cifrado de clave pública como RSA. En esencia, MD5 es un modo de verificar la integridad de los datos, siendo más eficaz que un *checksum* y otros métodos comúnmente empleados.

SHA-1 y SHA-256 SHA-1 [139] es el algoritmo de hash más ampliamente aceptado. La versión original de este algoritmo, SHA, fue desarrollada por la Agencia de Seguridad Nacional (National Security Agency -NSA-), y revisado en 1995 para incrementar la seguridad incluso antes de que cualquier defecto fuera encontrado en investigaciones abiertas. SHA-1 fue introducido como estándar federal al mismo tiempo que un algoritmo de cifrado secreto de 80 bits llamado Skipjack y el Digital Signature Standard (DSS). Los parámetros de seguridad de todos estos estándar fueron elegidos de modo que garantizara el mismo nivel de seguridad, en el rango de las 2^{80} operaciones, como requería el mejor ataque conocido. Después de la introducción del nuevo estándar de cifrado de clave secreta, AES, con tres tamaños de clave, 128, 192 y 256 bits, la seguridad del SHA-1 no coincidía con la garantizada por el estándar de cifrado. Por lo tanto, la NSA realizó un esfuerzo por desarrollar tres nuevos algoritmos de hash, manteniendo la seguridad del AES con 128, 192 y 256 bits de clave respectivamente. Este esfuerzo dio como resultado la introducción de tres nuevos algoritmos de hash referidos como SHA-256, SHA-384 y SHA-512 [139].

2.6.3. Sistemas criptográficos en FPGAs

Los dispositivos reconfigurables presentan varias ventajas para la implementación de algoritmos de criptografía [45, 44]:

- Agilidad de los Algoritmos: este término se refiere al intercambio de algoritmos durante la operación de la aplicación. Se puede observar que la mayoría de los protocolos de seguri-

dad, como SSL [140] o IPSec [42], son independientes del algoritmo y permiten múltiples algoritmos de cifrado. Algunas de las ventajas de estos protocolos son: 1) la habilidad para eliminar algoritmos rotos 2) elección de los algoritmos de acuerdo a ciertas preferencias (por ejemplo, personales o por motivo de patentes) 3) habilidad para añadir nuevos algoritmos. Aunque la agilidad de los algoritmos resulta costosa para el hardware tradicional, las FPGAs pueden ser programadas *"on the fly"*.

- Actualización de los Algoritmos: desde el punto de vista criptográfico la actualización de algoritmos es necesaria porque un algoritmo actual puede ser roto (por ej. DES), que expire un estándar (por ej. DES) o que se cree un nuevo estándar (por ej. AES). Suponiendo algún tipo de conexión a redes como Internet, los dispositivos equipados con FPGAs pueden actualizarse con la nueva configuración. Esta posibilidad diferencia claramente este tipo de sistemas de los ASICs, en los que la actualización es prácticamente imposible.
- Eficiencia de la Arquitectura: en ciertos casos las arquitecturas hardware puede ser mucha más eficientes sin son diseñadas para un conjunto específico de parámetros. Estos parámetros en los algoritmos de criptografía pueden ser por ejemplo la clave, el coeficiente empleado, el tamaño de bloque, etc. Generalmente hablando, cuanto más específico un algoritmo es implementado más eficiente puede llegar a ser. Una implementación muy eficiente del algoritmo IDEA basada en claves fijas fue presentada en [141]. Con claves fijas, la principal operación del algoritmo IDEA se convierte en una multiplicación por constante que es mucho más eficiente que una multiplicación estándar.
- Eficiencia de Recursos: la mayoría de los protocolos de seguridad son híbridos, por ej. IPSec, SSL, TSL [142]. Esto implica que un algoritmos de clave pública se emplea para la transmisión de la clave de sesión. Después de que la clave ha sido establecida se emplea un algoritmo de clave privada para el cifrado de los datos. Ya que los algoritmos no se emplean simultáneamente, el mismo dispositivo FPGA se puede emplear mediante reconfiguración en tiempo de ejecución.
- Modificación del Algoritmo: existen aplicaciones que requieren modificaciones de los algoritmos de criptografía estándar, por ej., usando S-boxes o permutaciones propietarias. Estas modificaciones son fáciles de implementar mediante hardware reconfigurable. Un ejemplo donde un algoritmo estándar fue ligeramente modificado es el cifrado de las contraseñas en los sistemas UNIX [125]. Además, en muchas ocasiones las primitivas de criptografía o sus modos de operación deben ser modificados de acuerdo a la aplicación.
- Rendimiento: los procesadores de propósito general no están optimizados para una ejecución rápida, especialmente en el caso de los algoritmos de clave pública [141, 143], porque

las instrucciones para el manejo de operaciones de aritmética modular emplean operandos muy grandes. Las operaciones aritméticas modulares incluyen por ejemplo exponenciación para RSA [144] y multiplicación, raíz cuadrada, inversión, y suma para criptosistemas de curva elíptica (ECC) [145, 146]. Aunque típicamente son más lentas que las implementaciones en ASIC, las implementaciones en FPGA tienen el potencial de ejecutarse más rápido que las implementaciones software.

- Eficiencia de Coste: los dos factores a tener en cuenta respecto al coste cuando se analiza la eficiencia de las FPGAs son el coste de desarrollo y el coste de los dispositivos. El coste para el desarrollo de un determinado algoritmo es mucho menor que el caso de los ASICs porque se puede emplear la estructura de la FPGA (por ej. *look-up tables*), y porque es posible testear el dispositivo tantas veces como sea necesario sin coste añadido. Esto permite disponer del diseño en un tiempo corto, lo que actualmente supone una enorme reducción de coste. Los precios de cada dispositivo no son significativos con respecto a los costes de desarrollo. Sin embargo, para un alto volumen, las soluciones ASIC suelen ser una opción más eficiente.

Todas las ventajas anteriores no solo son aplicables a los algoritmos de criptografía. También pueden ser empleadas en diferentes contextos y aplicaciones.

Capítulo 3

El Algoritmo de Cifrado IDEA en FPGA

3.1. Introducción

Cada vez es más común que los sistemas de comunicaciones incrementen su velocidad de transmisión. Esto obliga a las diferentes soluciones basadas en los actuales protocolos de seguridad a garantizar el cifrado de datos a esas velocidades. Para ello en muchas ocasiones no es suficiente una solución basada en software, y es por ello que se hace necesario desarrollar soluciones basadas en hardware. Tradicionalmente la mayor parte de las soluciones se basan en ASICs, es decir, soluciones hardware a medida. Sin embargo, con la llegada de los nuevos protocolos de seguridad que requieren la negociación y manejo de un amplio conjunto de algoritmos de cifrado, muchas de las soluciones basadas en los diseños hardware actuales no son suficientes. Es por ello que el hardware reconfigurable se ha presentado como una alternativa interesante.

Las implementaciones de algoritmos criptográficos sobre hardware reconfigurable se realizan en la mayoría de los casos empleando HDLs (Hardware Description Languages). Sin embargo, los progresos empleando este tipo de lenguajes se encuentran asociados a la mejora de la tecnología de los circuitos reconfigurables, como son una mayor cantidad de recursos lógicos, mayor velocidad en las líneas de conexión o un mejor rutado. También es posible la optimización de las unidades operacionales que los involucran, para que sean eficientemente implementadas sobre el dispositivo reconfigurable. Sin embargo, la posibilidad que ofrecen estos dispositivos para ser reconfigurados parcialmente no ha sido explotada de forma habitual en los desarrollos sobre FPGA, y solo ahora se está comenzado a tener conciencia del potencial disponible.

Una de las primeras implementaciones de un algoritmo criptográfico que hace uso de la reconfiguración parcial de las FPGAs fue llevada a cabo por Patterson [147]. Se trata de una implementación del algoritmo DES completamente desarrollada con JBits. El objetivo de la implementación era adaptar el algoritmo DES a la clave de cifrado seleccionada, de modo que fuera posible eliminar toda la lógica asociada a la generación de subclaves, reduciendo parte de los recursos necesarios al estar el diseño especialmente desarrollado para una clave específica. Posteriormente, al seleccionarse una nueva clave se genera un nuevo diseño, y se reconfigura el dispositivo con los cambios necesarios.

En este capítulo se abordan diferentes implementaciones del algoritmo IDEA que tienen como objetivo obtener los mismos resultados con IDEA que en el trabajo de Patterson con DES y al mismo tiempo ofrecer una solución aplicable en plataformas comerciales con un alto rendimiento. El algoritmo IDEA se ha considerado un *benchmark* adecuado para la implementación de algoritmos de criptografía en hardware reconfigurable [131]. Además, es mejor candidato que el algoritmo DES para aplicar esta metodología ya que, mientras que en el caso del algoritmo DES las subclaves solo afectan a la operación XOR de cada round, en el algoritmo IDEA las subclaves afectan a las operaciones de multiplicación, por lo que se reduce drásticamente el tamaño del diseño final. Además, la aparición de la familia Virtex-II con multiplicadores embebidos ofrece nuevas posibilidades a la hora de implementar el algoritmo IDEA, del cual la multiplicación es la operación dominante. Es por ello que se ha estudiado su implementación en esta nueva familia de FPGAs, y se ha comparado con las implementaciones basadas en reconfiguración parcial.

3.2. El algoritmo IDEA

El algoritmo de cifrado IDEA fue propuesto por Lai y Massey en 1990 [129]. Se trata de un algoritmo simétrico que actúa cifrando grupos de 64 bits binarios empleando una clave de 128 bits [131, 148]. Esto quiere decir que con un bloque de 64 bits y una clave de 128 bits se obtiene un nuevo bloque de 64 bits. El algoritmo consta solo tres operaciones : XOR, suma módulo 2^{16}y multiplicación módulo $2^{16}+1$, y consiste en 8 rounds computacionalmente idénticos seguidos por una transformación de salida. El round n-ésimo recibe 64 bits de entrada como cuatro subbloques de 16 bits y produce cuatro bloques de 16 bits, como se muestra en la figura 3.1.

Para realizar el proceso de cifrado es necesario generar 52 subclaves de 16 bits, empleando seis subclaves por round y cuatro subclaves en la última transformación. La generación de las subclaves se realiza a partir de los 128 bits de clave, los cuales se dividen inicialmente en 8 subclaves de 16 bits que se corresponden con las primeras subclaves. Posteriormente los dígitos de la clave de 128 bits son rotados 25 veces a la izquierda para obtener la nueva clave, la cual se vuelve a dividir en

Ref.	Circuito	Características	Rendimiento (Mbits/s)
[149]	VINCI	ASIC@25MHz CBC	117,8
[150]	IDEACrypt	ASIC@40MHz	300,0
[151]	TMX320C6x	DSP@200MHz	53,1
[131]	XC4005	FPGA@7.3MHz	0,447
[151]	XC4000	4 FPGAs@33Mhz	528,0
[141]	PipeRench	@100Mhz	126,6
[152]	HIPCrypto	8 dispositivos@53Mhz	3400,0
[153]	Leong et al.*	XCV1000-6@141MHz	2350,0
[154]	CryptoBoost	XCV1000-4 CBC@13.1MHz	102,3
[155]	CryptoBoost II	XCV1000-4@66MHz	4300,0
[155]	CryptoBoost III*	XCV1000-4 CBC@200 MHz	200,0

* Uso de aritmética serie

Tabla 3.1: IDEA en la bibliografía

las siguientes subclaves de 16 bits. Este último paso se repite hasta que se generan las 52 subclaves necesarias.

Aunque el cifrado y descifrado presentan un esquema computacional idéntico, se generan las subclaves de forma diferente para la misma clave de 128 bits. Las subclaves para el descifrado se obtienen a partir de la subclaves de cifrado, calculando el inverso multiplicativo de la subclaves empleadas en las multiplicaciones, y el inverso aditivo de las subclaves empleadas en las sumas. Más detalles sobre la generación de subclaves se pueden encontrar en [126]. El proceso de expansión de subclaves ocurre solo cuando una clave es elegida y el resultado puede salvarse para ser usado después. Por ello, la expansión de subclaves requiere solo una pequeña fracción del total del tiempo de computación.

Existen muchas referencias en la bibliografía sobre implementaciones del algoritmo IDEA en hardware reconfigurable. La primera implementación VLSI de IDEA se desarrolló en el Integrated Systems Laboratory del ETH de Zurich, y se obtuvo un rendimiento de 115 Mb/s [129]. Posteriormente han sido muchas las implementaciones que se han desarrollado. Algunas de ellas se muestran en la tabla 3.1, donde también se muestran soluciones en ASICs o DSPs. En cuanto al uso del hardware reconfigurable las sucesivas mejoras de rendimiento se deben a la evolución de la tecnología, lo que supone emplear dispositivos más modernos. En otras ocasiones la mejora se debe a optimizaciones del algoritmo, como el uso de aritmética serie o mejoras sobre el multiplicador.

Centrándonos en los resultados que presentan un mayor rendimiento, destacan la implemen-

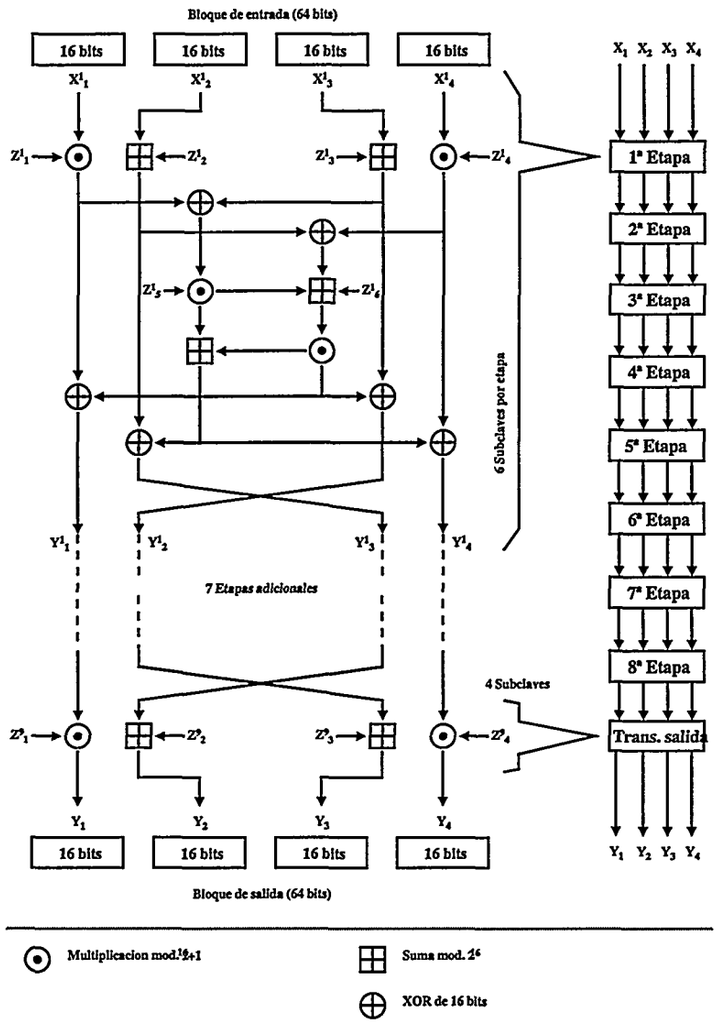


Figura 3.1: Algoritmo IDEA: Flujo de datos

Algoritmo 1 Algoritmo Low-High

```
CONST static uint16
   mul(register uint16 a, register uint16 b) {
        register word32 p;
        if(a) {
           if(b) {
              p = (word32)a*b;
              b = low16(p);
              a = p >>16;
              return b -a + (b <a);
           }
           else return 1 - a;
        }
        else return 1 - b;
   }
```

tación serie-paralela descrita en [156] cuyo rendimiento es de 5,25 Gbits/s en una FPGA Xilinx XCV1000, o una implementación completamente segmentada sobre una FPGA Xilinx XCV1000 que presenta un rendimiento de 6,78 Gbits/s [157].

De igual modo, el algoritmo IDEA ha sido también ampliamente implementado en software. Como referencia del rendimiento sobre procesadores de propósito general destacan los 4,6 Mbits/s en un procesador Pentium@90MHz y los 261,8 Mbits/s en un PIII@450MHz, que mejoran a 440 Mbits/s en un PIII@800MHz. Estos resultados se han obtenido a partir del programa IDEA distribuido por ASCOM. También es posible adaptar el código fuente para sacar partido de la arquitectura del procesador, y con estas mejoras recientemente se han reportado rendimientos de hasta 550 Mbits/s para código optimizado para Itanium@733MHz [158]. En [159] se pueden encontrar otros resultados de implementaciones de IDEA en software.

3.3. Implementación de IDEA en VHDL

De las operaciones que envuelven el algoritmo IDEA únicamente la operación multiplicador módulo $2^{16}+1$ presenta gran complejidad, ya que al ser implementada en hardware se requiere de una gran cantidad de recursos para el diseño de los multiplicadores, siendo esta operación el objeto de estudio y optimización si se quiere conseguir un alto rendimiento. Para entender el funcionamiento del multiplicador módulo $2^{16}+1$, en el algoritmo 1 se muestra la descripción en lenguaje de alto nivel de uno de los algoritmos que lo implementan, el algoritmo Low-High [160].

Aunque este algoritmo es relativamente sencillo, la parte central del mismo emplea una multi-

Implementación	Slices	Dispositivo	Frecuencia	Throughput
Combinacional	6863	55 % XCV1000-6	1,4 MHz	0,08 Gbits/s
Segmentada	7410	60 % XCV1000-6	24,5 MHz	1,50 Gbits/s

Tabla 3.2: Implementaciones de IDEA en VHDL

plicación de 16 bits que requiere de una gran cantidad de recursos lógicos. En el caso de la FPGA XC4013E de la plataforma Labomat3 (apéndice A), como ejemplo de dispositivo antiguo, el multiplicador combinacional ocupa 307 CLBs. Si se tiene en cuenta que el algoritmo IDEA emplea un total de 32 multiplicadores, se hacen necesarios unos 9824 CLBs de la FPGA XC4000 o lo que es lo mismo 4912 CLBs de la familia Virtex (se mantiene la equivalencia 1 Slice = 1 CLB 4000). Por ello, es necesario plantear diferentes técnicas a la hora de implementar el algoritmo IDEA en dispositivos de pequeño tamaño como se muestra en el apéndice A.

Con el paso del tiempo, las FPGAs han alcanzado capacidades muy superiores a las presentadas en la plataforma anterior, por lo que es necesario disponer de nuevos resultados sobre dispositivos más modernos. En el momento del desarrollo de este trabajo, se optó por una FPGA Virtex XCV1000. Dos diseños iniciales fueron realizados sobre este dispositivo para ser tomados como referencia. Ninguno de ellos incluye la generación de subclaves, que son introducidas mediante un protocolo de bus sencillo, ni tampoco ninguna mejora excepcional, simplemente se basan en la descripción del algoritmo IDEA (figura 3.1) en lenguaje VHDL. Los diferentes resultados se muestran en la tabla 3.2, y se corresponden con un diseño combinacional completo del algoritmo IDEA, y un diseño segmentado a nivel multiplicador y módulo con un total de 58 etapas de segmentación. Como se puede apreciar en estos resultados, un gasto adicional de un 5 % en área, para introducir los registros de segmentación, permite obtener una mejora considerable. El rendimiento obtenido por la solución segmentada se compara satisfactoriamente con la mayoría de las implementaciones mostradas en la tabla 3.1 aunque se encuentra muy por debajo del rendimiento ofrecido por los diseños más actuales [156, 157, 161].

3.4. Metodología de diseño

Muchas de las operaciones del algoritmo IDEA [155, 157, 153, 156] emplean los datos a cifrar y la clave de cifrado. Esto no es exclusivo de IDEA, ya que muchos algoritmos simétricos operan de igual forma, como DES [162, 147] y Rijndael [135]. En este contexto si la clave permanece fija, una gran cantidad de operaciones del algoritmo serán por constante. Esto puede resultar interesante porque supone un beneficio tanto en área como en rendimiento, al ser ampliamente conocido que

las operaciones por constante son más rápidas y que ocupan menos área [163, 164]. El uso de esta metodología presenta dos ventajas:

- Las operaciones aritméticas basadas en constante son muy rápidas y ocupan poca área, lo que es realmente acertado aplicado a FPGAs.
- La generación de subclaves puede ser eliminada porque esta operación la puede desarrollar el microprocesador asociado, que calcula las nuevas constantes y reconfigura la FPGA.

Sin embargo, un hardware de cifrado que mantenga la clave fija no tiene mucho sentido porque ofrece una seguridad muy pobre. Sin embargo, es normal que la clave se mantenga constante durante un tiempo relativamente largo. Por ejemplo, en una conexión segura una vez negociada la clave de cifrado, ésta se mantiene durante toda la sesión. Entonces, en un entorno FPGA tendría sentido emplear un hardware con clave fija, el cual podría ser reconfigurado cada vez que la clave cambiase. Esta solución sería óptima y permitiría aprovechar la ventaja de un hardware basado en constantes, lo cual tiene más sentido si la reconfiguración se realiza lo suficientemente rápido para evitar parar el cifrado durante una cantidad de tiempo considerable. Por tanto, el hardware debe ser diseñado de modo que la clave pueda ser modificada alterando la configuración del dispositivo rápida y fácilmente.

3.4.1. Generación de subclaves

El algoritmo IDEA, al igual que otros algoritmos de cifrado, genera subclaves a partir de la clave inicial que se emplea en el proceso de cifrado. En el caso de IDEA, se deben generar 52 subclaves que son empleadas a lo largo del algoritmo. Este proceso de generación de subclaves solo es útil al inicio de cada sesión de cifrado, y por tanto, solo se emplea una pequeña cantidad de tiempo de procesamiento para esta tarea. Se trata de una operación que puede ser fácilmente trasladada a software. Con ello se consigue eliminar el hardware necesario para la entrada y captura de la clave, así como la generación de las subclaves. De este modo, desde el software es posible obtener la nueva clave de cifrado y generar las nuevas subclaves para ser posteriormente enviadas al hardware. Si no existiese la posibilidad de reconfigurar parcialmente el diseño hardware, se deberían introducir estas nuevas subclaves generando un nuevo diseño, y enviando toda la configuración completa, o bien, cargando estas subclaves en alguna memoria asociada a la FPGA. Sin embargo, esta última opción necesitaría de una lógica de control de acceso a memoria.

Empleando Reconfiguración Parcial es posible cargar estas nuevas subclaves directamente en las unidades que las necesitan como parte del propio diseño, ya sea reconfigurando valores internos de la unidad (LUTs, constantes, etc) o rediseñando la unidad parcialmente. Es importante recordar que la Reconfiguración Parcial requiere menos tiempo para reconfigurar el dispositivo que una

configuración completa. Al mismo tiempo, al reconfigurar el dispositivo con las nuevas subclaves también permite que, tras cambiar las subclaves de cifrado por las subclaves de descifrado, se pueda cambiar el modo de operación (cifrado/descifrado).

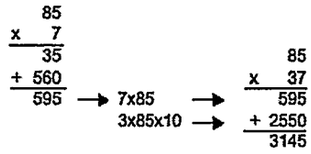
3.4.2. KCM, multiplicador de coeficiente constante

Como se ha comentado antes, al analizar el algoritmo IDEA se puede observar que algunas sumas y todas las multiplicaciones tiene asociada una subclave como uno de los operandos. Esto puede ser una ventaja si se piensa que durante el proceso de cifrado/descifrado esta subclave no cambia, al menos, durante la misma sesión. Si bien la implementación de sumadores por constante no reduce el número de recursos necesarios, en el caso de la multiplicación es diferente. Un multiplicador por constante, al estar "especializado", debería y resulta ser un diseño más sencillo, y por tanto de menor tamaño. Para tener una idea de la magnitudes que se manejan, un multiplicador paralelo de 8x8 bits ocupa 40 *slices* de una Virtex, comparado con los 21 *slices* de un multiplicador de coeficiente constante de 8 bits [164]. Sin embargo, este multiplicador presenta una seria desventaja: un cambio de constante requiere modificar el multiplicador. Aquí es donde la Reconfiguración Parcial entra en juego.

El multiplicador KCM

El multiplicador de coeficiente constante que se ha seleccionado es el KCM [163, 164], el cual simplifica la tarea de multiplicar una variable por una constante k. El multiplicador KCM mantiene un número de copias de tabla de multiplicar de la constante k, cada una con 16 entradas. Cuando multiplicamos una variable de n bits por una constante, se requieren n/4 copias de la tabla de multiplicar. Cada grupo de n/4 bits habilita una dirección de la tabla, y el producto parcial producido se suma con los demás. Para un cierto ancho de la constante el tamaño del multiplicador KCM es independiente de la constante codificada en él.

En las FPGAs de la serie Virtex existen tres métodos posibles para codificar la constante en el multiplicador KCM. La función generadora se puede configurar como LUTs (ROM), RAM o registros de desplazamiento (SRL16). En el primer método, basado en ROM, se modifica el valor almacenado mediante reconfiguración parcial o desarrollando un nuevo diseño y configurando de nuevo el dispositivo. En el segundo método, empleando RAM, el cambio de constante se realiza escribiendo en la memoria directamente empleando lógica adicional, para implementar el mecanismo de comunicación. En el tercer método, usando SRL16, se adaptan los valores a usar a la funcionalidad de los registros de desplazamiento, y las LUTs pueden ser conectadas juntas para proveer un mecanismo de carga. Al igual que con el uso de RAM, requiere de lógica adicional, pero emplea menos I/O que el acceso a RAM, que es paralelo.



55 x 2B 3A7 → B x 55 + 0AA0 → 2 x 55 0E47	
0 x 55 = 000	8 x 55 = 2A8
1 x 55 = 055	9 x 55 = 2FD
2 x 55 = 0AA	A x 55 = 352
3 x 55 = 0FF	B x 55 = 3A7
4 x 55 = 154	C x 55 = 3FC
5 x 55 = 1A9	D x 55 = 451
6 x 55 = 1FE	E x 55 = 4A6
7 x 55 = 253	F x 55 = 4FB

Figura 3.2: Metodología del KCM

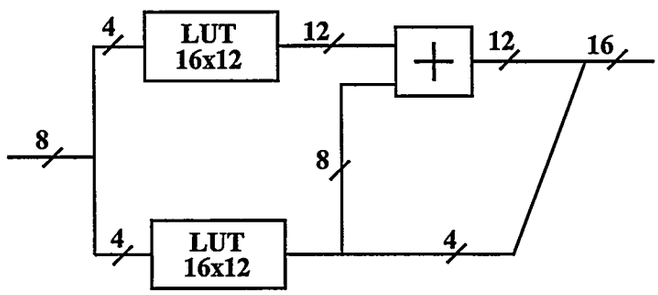


Figura 3.3: Multiplicador KCM de 8 bits

Con este multiplicador se consigue un tamaño de circuito entre 3,8 y 4 veces más pequeño que un multiplicador combinacional [163], manteniendo el mismo rendimiento. Un multiplicador KCM basado en ROM, de 8 bits, requiere 21 *slices* de una Virtex y puede operar a 180 MHz [164].

3.5. IDEA usando JBits

El objetivo de esta sección es realizar, partiendo de la metodología propuesta, una aplicación software enteramente en Java y empleando JBits, que sea capaz de generar un diseño hardware del algoritmo IDEA adaptado a las subclaves generadas a partir de una clave inicial. Posteriormente, un cambio de clave requiere que el diseño deba ser nuevamente adaptado, reconfigurando el dispositivo con los cambios necesarios.

Para tener conocimiento de la terminología empleada en el trascurso de este capítulo, la cual se encuentra relacionada con JBits y las diferentes herramientas que lo engloban, se encuentra disponible el apéndice B, que explica detalladamente todo lo que es necesario saber sobre JBits. Además, se dispone del código Java del multiplicador módulo $2^{16}+1$ desarrollado.

3.5.1. KCM en JBits

El multiplicador KCM requiere de recursos que particularmente se encuentran ya disponibles internamente dentro de la arquitectura de la familia Virtex. Es por tanto muy sencillo de mapear internamente, y en parte gracias a la posibilidad de trabajar a bajo nivel que ofrece JBits, el diseño resulta sencillo. Un multiplicador de 16x16 bits requiere de lo siguientes elementos:

- 4 LUTs de 16x20 bits para almacenar los productos parciales (memorias).
- 2 sumadores de 20 bits para sumar el primer nivel de productos parciales.
- 1 sumador de 24 bits para sumar el segundo nivel de productos parciales.

Cada *slice* de la familia Virtex dispone de dos LUTs de cuatro entradas, siendo fácilmente implementable una LUT de 16x20 bits empleando 10 *slices*. En total las memorias deben de ocupar 4x10 *slices* = 40 *slices* = 20 CLBs. Los sumadores de 20 bits se pueden implementar empleando el *RTP Core Adder* que viene con JBits. Este core implementa un sumador de tal forma que suma dos bits por *slice* empleando un único *slice* de cada CLB, lo que supone que cada sumador de 20 bits ocupa la mitad de 10 CLBs, y los dos juntos, 10 CLBs completos. Del mismo modo, el sumador de 24 bits ocupa la mitad de 12 CLBs.

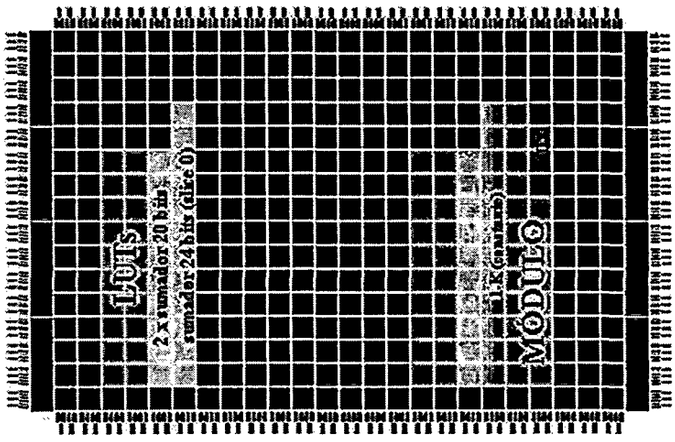


Figura 3.4: KCM 16x16 y KCM 16x16 módulo 2^{16}+1 en una Virtex XCV50

La parte izquierda de la figura 3.4 muestra el multiplicador KCM descrito anteriormente. Las dos primeras columnas de CLBs se corresponden con las LUTs que contienen las tablas. La siguiente columna contiene los dos sumadores de 20 bits. La última columna contiene el sumador de 24 bits, que solo ocupa la mitad de dicha columna (todos los *slices* 0 de cada CLB).

Es importante que las LUTs ocupen una misma columna de CLBs debido a que se trata de la parte del multiplicador que será necesario reconfigurar cada vez que cambie la constante.

3.5.2. Multiplicador módulo 2^{16}+1

Una vez presentada la alternativa al multiplicador combinacional clásico, es necesario añadir la parte que corresponde al ajuste para que el resultado se encuentre siempre dentro del rango de los 16 bits, el módulo. El módulo normalmente se encuentra asociado a una posterior división del resultado de la multiplicación entre 2^{16}+1, de modo que el resto vendría a ser el resultado esperado. No obstante, la división presenta un problema de tamaño muy similar a la multiplicación. Llegados a este punto es necesario sustituir la división por una alternativa que simplifique el cálculo del módulo.

Como se indicó anteriormente, el algoritmo Low-High resuelve el problema del módulo basándose en las particularidades matemáticas del módulo 2^{16}+1. En caso de que cualquiera de los dos operandos sea cero, entonces se devuelve el resultado de restar al valor '1' el otro operando. Esto tiene una interpretación bastante interesante si se tiene en cuenta que uno de los operandos es

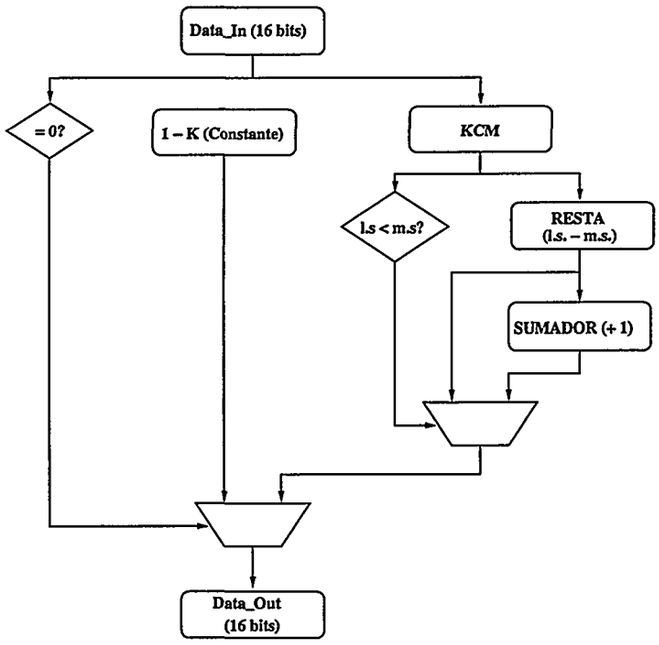


Figura 3.5: Cálculo del módulo 2^{16}+1 basado en el algoritmo Low-High

la constante del multiplicador KCM. En este sentido, si la clave es cero, el multiplicador completo puede ser sustituido por una resta entre el valor '1' y el operando. Esto simplifica al máximo el multiplicador ya que su tamaño sería reducido al de una simple resta. Por el contrario, si el operando es cero, el resultado corresponde a la resta entre el valor '1' y la constante del multiplicador, la subclave. Esto se traduce en la implementación de una constante cuyo valor sea 1 - k, siendo k la constante del multiplicador. Por ello, es posible implementar el multiplicador como una resta en caso de que la constante sea cero, y en el resto de los casos, por un lado implementar el multiplicador KCM, la resta de operandos, la comparación "menor que" y la suma (para sumar uno al resultado de la resta en caso de que la comparación sea positiva), y por el otro, una constante con el valor 1 - k para el caso de que el operando sea cero, con el correspondiente comparador de igualdad y los multiplexores necesarios para seleccionar el resultado. La figura 3.5 muestra el diagrama para el cálculo del módulo 2^{16}+1 basado en el algoritmo Low-High.

Todo este diseño al completo puede observarse en la figura 3.4 en su parte derecha. En este

caso, la constante no es cero. El resultado final de implementar el multiplicador módulo $2^{16}+1$ por constante es de 66 CLBs. Empleando Reconfiguración Parcial es posible modificar el diseño del multiplicador de modo que para un valor de constante cero, el multiplicador puede implementarse como una simple resta, y en el caso contrario, si cambia la constante a un valor distinto de cero, puede implementarse el multiplicador con el diseño de la figura 3.4. Si la constante cambia de nuevo, y no es cero, solo se reconfiguran los valores de las LUTs y la constante 1 - k, dado que solo es necesario cambiar los resultados relacionados con la constante.

3.5.3. Implementación

Siguiendo con la mejora mostrada para el multiplicador módulo $2^{16}+1$, los dos primeros sumadores de cada round y los dos sumadores de la transformación final, que al igual que los multiplicadores utilizan una subclave como operando, se han implementado como sumadores por constante, lo que significa que es necesario reconfigurarlos en caso de cambio de clave. El resto de unidades del algoritmo son fácilmente implementables empleando cores ya existentes en el API de JBits: sumador (Adder) y xor (XOR). Estos cores al ser configurables permiten adaptarlos a las necesidades del IDEA.

Para facilitar el posterior diseño del algoritmo, fue necesario crear un core para un round completo de IDEA. Este core debe contener los siguientes elementos:

- 4 multiplicadores, lo que suponen 4x66 CLBs.
- 4 sumadores, lo que suponen 4x8 CLBs (realmente solo se emplea un *slice* de cada CLB).
- 6 xor, lo que suponen 6x4 CLBs.

En total el core de round emplea 320 CLBs. A partir del core de un round se implementa fácilmente el algoritmo IDEA combinacional completo, el cual contiene 8 cores de round y una etapa final en la que se añaden dos multiplicadores módulo $2^{16}+1$ y dos sumadores por constante. Esto supone que los 8 rounds de IDEA deben ocupar 2560 CLBs que unidos a la etapa final formada por dos multiplicadores y dos sumadores hacen un total de 2708 CLBs.

3.5.4. Segmentación: Acelerando el diseño

De acuerdo con el esquema del algoritmo IDEA, además de registrar la entrada y salida del diseño combinacional, se pueden introducir distintos niveles de segmentación. Al ser todos los rounds de IDEA iguales, el primero de los niveles de segmentación podría ser registrar la salida de cada round. Esto supone añadir registros de 16 bits en cada una de las salidas del round. Otro nivel

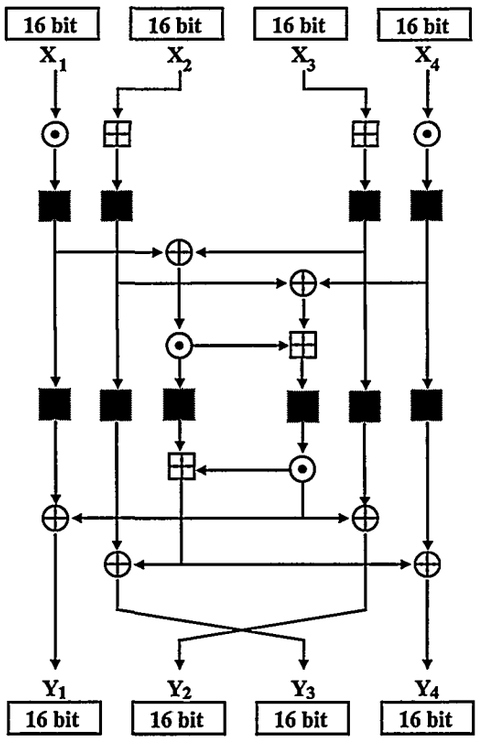


Figura 3.6: Round segmentado a nivel multiplicador

de segmentación posible es a nivel de multiplicador. Es fácilmente compresible que el componente con mayor retraso del algoritmo es el que marca la máxima velocidad del diseño, salvo que se segmentase dentro del propio multiplicador, por lo que la mejor optimización posible es registrar la salida de cada multiplicador. En la figura 3.6 se puede observar la segmentación realizada a este nivel.

El diseño en JBits para un round con esta segmentación se puede apreciar en la figura 3.7. Aquí puede observarse que el round abarca un array de 11x56 CLBs tras haber sido añadidos los 10 registros adicionales necesarios para segmentar a este nivel. Obviamente, no todo el array está ocupado dada la disposición de los componentes del round. Aunque esta segmentación no es precisamente la mejor que podría realizarse a nivel multiplicador, debido a que no se incluye

solo al multiplicador, es importante tener en cuenta que al ocupar un round 11 CLBs de ancho, la suma de los 8 rounds ocupa 88 CLBs. La XCV1000 está constituida por un array de 96x64, lo que significa que no podemos extendernos más porque todavía queda añadir la transformación final. Esta transformación final ocupa 6 CLB de ancho, el correspondiente al multiplicador. El resultado final del diseño IDEA segmentado ocupa un array de 94x56 CLBs.

Un detalle a tener en cuenta es que al emplear las BRAM de la Virtex, es necesario dejar al menos una columna de CLBs para el rutado de las conexiones con las memorias. Esto hace que se haya aprovechado la FPGA al 100% en cuanto al número de columnas de CLBs. Por otro lado, la FPGA no está completamente ocupada, lo que permite disponer de espacio para un correcto rutado.

En la figura 3.8 se puede apreciar el diseño del algoritmo IDEA segmentado y su ocupación en CLBs. Se distinguen claramente los 8 rounds y la etapa final. Esta simetría permite en principio que el rutado del diseño sea bastante simétrico también, ya que cada round debería presentar el mismo rutado interno y posteriormente la dificultad estaría en el rutado entre rounds.

En la figura 3.9 se muestra la densidad de rutado de cada CLB del diseño. Cuanto más oscuro está un CLB mayor es la cantidad de líneas de rutado que emplea.

Tras la segmentación el diseño contiene ahora 14 registros de 16 bits por round, lo que suponen 14x4 CLBs. Esto hace un total de 3156 CLBs para todo el diseño completo, lo que corresponde a un 51% (44% sin segmentación) de ocupación aproximadamente.

3.5.5. Validación y rendimiento

La verificación funcional se llevó a cabo inicialmente empleando el VirtexDS. Dadas las limitaciones de este simulador solo fue posible comprobar el correcto funcionamiento de los circuitos básicos como sumadores, comparadores y XOR, y del multiplicador módulo $2^{16}+1$ completo.

La validación del correcto funcionamiento de diseños posteriores se realizó empleando la plataforma RC1000PP de Celoxica [165] sobre un PII@333MHz, comparando los resultados obtenidos en hardware, con los resultados obtenidos de la ejecución de una versión de IDEA en Java que se ejecutaba simultáneamente. Para la entrada y salida de datos se emplearon las BRAM de la FPGA. La plataforma RC1000PP (Virtex XCV1000 BG560) fue seleccionada por estar perfectamente soportada por JBits. El manejo a nivel de carga del *bitstream*, reconfiguración, programación del reloj, *readback*, etc. es transparente al diseño empleando el API de JBits. Para la comprobación del correcto funcionamiento se cargaron las memorias BRAM de entrada con los datos a cifrar/descifrar, y posteriormente los resultados se obtenían de las memorias BRAM de salida. El correcto funcionamiento del algoritmo IDEA se validaba configurando inicialmente el diseño con las subclaves de cifrado, y posteriormente, con las subclaves de descifrado, verificando en todo

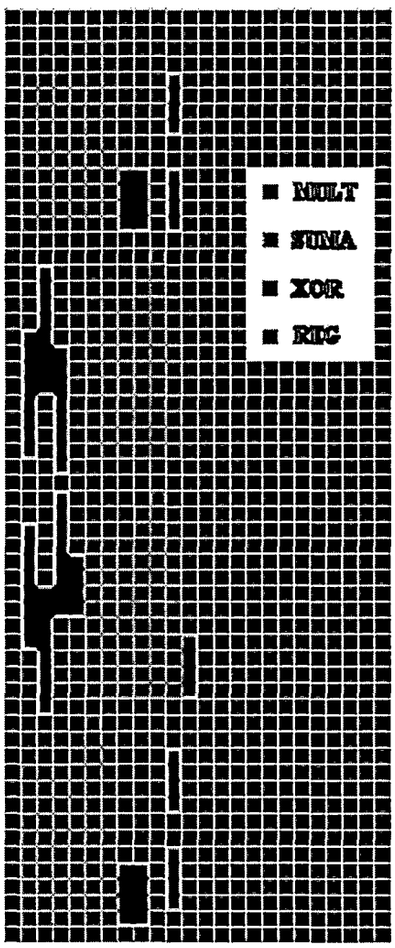


Figura 3.7: Round segmentado

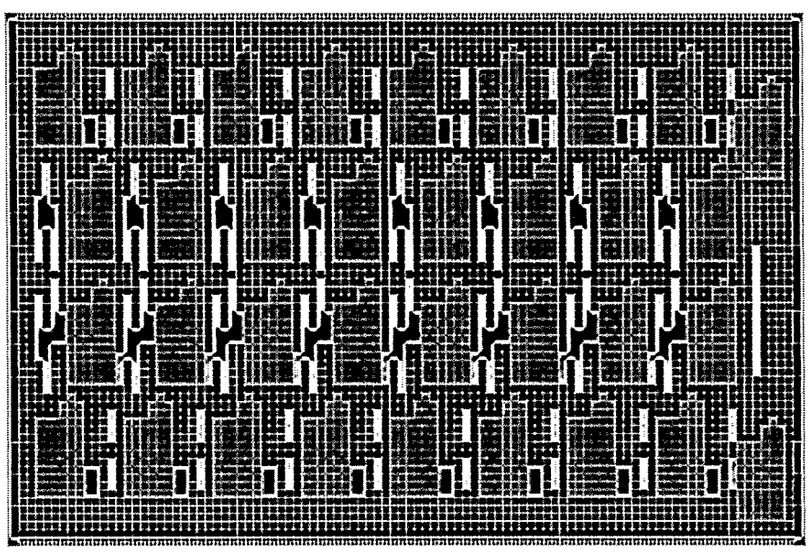


Figura 3.8: Implementación final de IDEA en una XCV1000 (BoardScope)

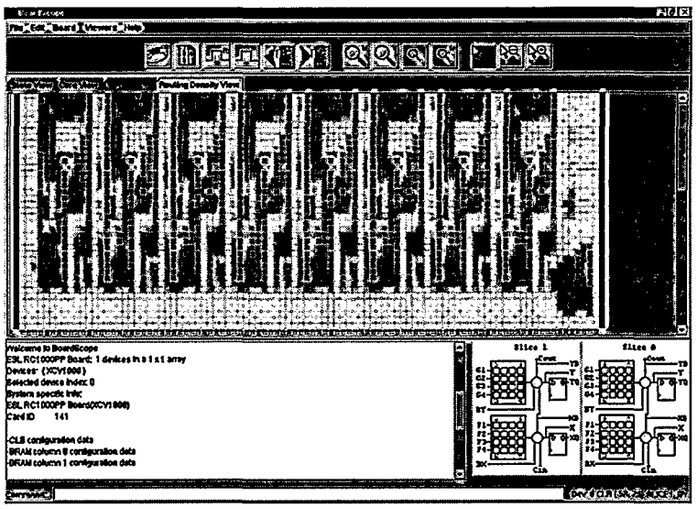


Figura 3.9: Densidad de rutado del diseño (BoardScope)

momento los resultados con una ejecución software paralela.

Una vez el algoritmo funciona correctamente, es necesario determinar la frecuencia de operación para conocer el rendimiento. Para ello hay que exportar el diseño desde JBits a formato XDL [166], posibilidad disponible en el API de JBits, con objeto de utilizar las herramientas estándar de Xilinx. Empleando la herramienta *xdl2ncd* se obtuvo el diseño en un fichero NCD, a partir del cual fue posible visualizar el rutado empleando el FPGA Editor (figura 3.10), y obtener el retardo máximo con el Timing Analyzer.

El primer diseño que se analizó fue el multiplicador, para el que se obtuvo una frecuencia de operación de 20 MHz. El resto de circuitos planificados, circuito combinacional y circuitos segmentados, no se han podido analizar debido a la falta de recursos en la plataforma de cálculo, dado el tamaño del fichero XDL resultante para cada uno de ellos. Sin embargo, a partir de la frecuencia del multiplicador es posible estimar la frecuencia de operación de la implementación combinacional de IDEA en 20/(3x8+1) MHz, o lo que es lo mismo 0,83 MHz. Las distintas implementaciones con segmentación planteadas también pueden ser estimadas. La frecuencia de operación segmentando a nivel de etapa es de aproximadamente 20/3 MHz, mientras que la segmentación a nivel de multiplicador presentaría una frecuencia aproximada a la del multiplicador, unos 20 MHz.

Para la verificación de las frecuencias estimadas, se ejecutaron los diferentes diseños variando

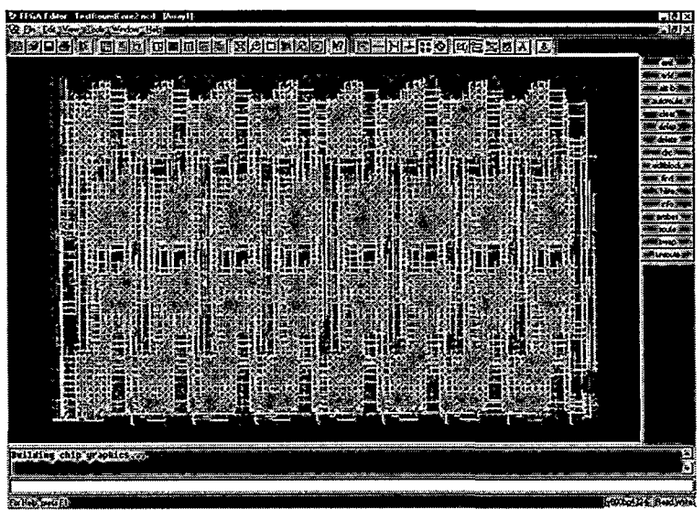


Figura 3.10: Rutado de los ocho rounds de IDEA en una XCV800

la frecuencia de operación en un rango aproximado al estimado, así como los diferentes valores de entrada, y comprobando en que momento los resultados ya no eran correctos, lo que puede suponer una frecuencia de operación demasiado alta. Las frecuencias de operación empleadas en las distintas pruebas y que resultaban en un incorrecto funcionamiento de los diseños se aproximan bastante a las estimadas, por tanto, y a falta de una comprobación más exhaustiva, se puede establecer que para los vectores de pruebas empleados, la implementación de IDEA con segmentación a nivel de multiplicador alcanza una frecuencia de operación de 20 MHz y por tanto, un rendimiento de 1,280 Gbits/s.

3.6. IDEA en VHDL + JBits

JBits se ha mostrado como una herramienta muy potente para el uso de la reconfiguración parcial. Sin embargo, el principal inconveniente de desarrollar con JBits es la necesidad de realizar un diseño a bajo nivel, que por un lado requiere conocer la arquitectura del dispositivo y al mismo tiempo, obliga al diseñador a emplazar manualmente el diseño, de modo que un mal emplazado puede afectar al rendimiento final, ya que el rutado puede resultar perjudicado por el posicionamiento de los distintos componentes. Y es este último, el mecanismo de rutado, uno de los puntos

débiles de la herramienta JBits.

Por otro lado, JBits presenta otro aspecto interesante como herramienta de diseño: permite unificar bajo un mismo lenguaje la descripción del circuito hardware y la aplicación software. En el caso del diseño de IDEA, la parte software se corresponde con la generación de las subclaves y el programa de test. Al ser el mismo programa el encargado de construir el circuito hardware, es posible optimizar el diseño en función de diferentes parámetros del programa durante su ejecución, y modificar el circuito en función de los mismos. Esto presenta unas enormes ventajas en temas de codiseño e interacción hardware/software.

Una vez conocido el potencial de JBits, en esta sección se propone realizar una nueva implementación combinando la sencillez de los lenguajes de descripción hardware (HDLs), y la capacidad de reconfiguración parcial de JBits para realizar cambios en el diseño. Esto supone que el diseño mediante HDLs debe estar preparado para que posteriormente partes del mismo puedan ser modificadas, de ahí que una parte fundamental del diseño es no permitir a las herramientas de síntesis e implementación, la aplicación de mecanismos de optimización o simplificación de los componentes involucrados en la reconfiguración. Además, y como veremos en más detalle, será necesario conocer la ubicación de esos componentes dentro del diseño para su posterior modificación.

3.6.1. Implementación

Este diseño presenta la misma filosofía que en el diseño con JBits: la implementación de IDEA se basa en reemplazar todas las unidades operacionales en las que interviene la clave por su equivalente basado en constante, esto es, el multiplicador módulo $2^{16}+1$ y el sumador módulo 2^{16}. Para simplificar el diseño se propone que todas las modificaciones del circuito, cuando cambia la clave, impliquen cambios en el contenido de LUTs, es decir, que los sumadores y multiplicadores módulo por constante deben ser diseñados de modo que puedan ser cambiados reconfigurando unas pocas LUTs.

La operación del circuito puede ser resumida como sigue. Primero, la FPGA es cargada con un diseño inicial del algoritmo IDEA susceptible de ser reconfigurada. Una vez determinada la clave, un microprocesador, en este caso el PC, calcula todas las subclaves y los nuevos valores de las LUTs, y envía los cambios en la configuración a la FPGA. Este proceso se realiza empleando JBits, ya que automáticamente infiere la diferencia entre el *bitstream* actual y el nuevo, reconfigurando solo los *frames* que hayan cambiado.

Algoritmo 2 Reconfiguración del sumador módulo 2^{16}

```
int[] f_zero = Expr.F_LUT("~F1");
int[] f_one = Expr.F_LUT("F1");

if(((constant >>bit)&0x1) == 0)
   jbits.set(row,column,LUT.SLICE0_F,f_zero);
else
   jbits.set(row,column,LUT.SLICE0_F,f_one);
```

Sumador módulo 2^{16}

Para esta unidad operacional, emplear un sumador por constante no supone una mejora significante con respecto a la versión sin operando constante, ya que solo se eliminan los registros necesarios para almacenar la subclave. El interés de la solución propuesta viene dada por que sirve de ejemplo para conocer, y evitar, las optimizaciones llevadas a cabo por la herramienta de síntesis.

El bloque básico para un sumador de n bits es un sumador de dos bits con acarreo. Si se realiza una descripción estructural en VHDL usando primitivas del *slice* de la familia Virtex, la operación XOR para sumar los dos bits de entrada es mapeada en una LUT de dos entradas. La versión con operando constante se obtiene transformando la anterior LUT de dos entradas en una LUT de una entrada. La operación de la LUT es invertir el bit de entrada o pasarlo directamente a la salida cuando el correspondiente bit de la constante es cero. Como el valor de la constante se debe modificar en tiempo de ejecución mediante reconfiguración parcial, esta LUT debe ser configurada por defecto como inversor, evitando una optimización no deseada ya que la herramienta puede rutar la entrada por otro componente que no sea la LUT. Posteriormente se puede cambiar su funcionalidad mediante JBits como se muestra en el algoritmo 2.

Multiplicador módulo $2^{16}+1$

Para esta unidad operacional se mantiene el uso del multiplicador por constante KCM propuesto en la implementación con JBits. El esquema para el multiplicador KCM de 16 bits desarrollado se muestra en la figura 3.11.

La solución óptima para su diseño es implementar las memorias como ROM. Sin embargo, existe un problema potencial con la herramienta de *place&route* ya que esta puede intercambiar las líneas para simplificar el rutado. Esto se resuelve empleando la primitiva SRL16 [164] en vez de una ROM. Esta primitiva se corresponde con un registro de desplazamiento, no con una memoria. Sin embargo, si la señal de *write enable* se mantiene baja y el contenido del registro es inicializado correctamente, puede trabajar del mismo modo que una memoria. Pero al no ser estrictamente una

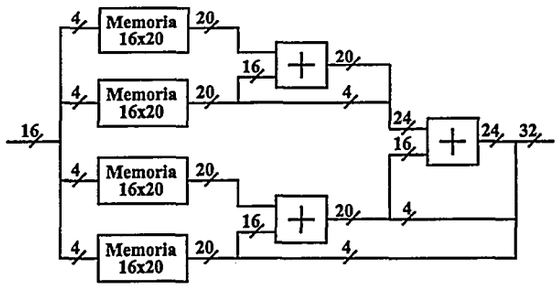


Figura 3.11: Multiplicador KCM de 16 bits

memoria, la herramienta de rutado no intercambia las líneas de entrada. Posteriormente durante la reconfiguración de las LUTs se convertirá en memoria ROM.

El cálculo del módulo $2^{16}+1$ se basa en el algoritmo Low-High presentando anteriormente. Sin embargo, en esta ocasión, se ha presentado una nueva adaptación, mucho más optimizada dada la experiencia adquirida durante el desarrollo del algoritmo IDEA en JBits. Esta nueva versión se muestra en la figura 3.12.

Como se aprecia en la figura 3.12A, el acarreo de salida de la resta "c - d" se emplea como comparación "c <d". Un selector de salida usando reconfiguración de LUTs, figura 3.12B, implementa la selección múltiple en el algoritmo Low-High. En tiempo de configuración hay dos posibilidades a considerar en función del valor constante. Cuando la constante es cero, la LUT debe funcionar como un buffer que deja pasar la entrada F4. En este caso, la entrada F4 está conectada al resultado de la resta $(1\text{-Operando})_i$. De otro modo, si la constante no es cero, el valor $(1\text{-K})_i$ es almacenado en la LUT y además la LUT es configurada como un multiplexor. Cuando el operando es cero la salida es $(1\text{-K})_i$, y en caso contrario la salida es el valor del bloque KCM_mod conectado a la entrada de la LUT (F2). Este diseño presenta el problema del intercambio en las entradas de la LUT, mencionado anteriormente para el multiplicador KCM, por lo que deben ser implementadas usando componentes SRL16. El código JBits para el selector de salida reconfigurable se muestra en el algoritmo 3.

Emplazamiento relativo

Para reconfigurar un componente desde JBits necesitamos conocer su ubicación dentro de la FPGA. Usando atributos de emplazamiento en el código VHDL, particularmente LOC y RLOC (figura 3.13), este problema se resuelve completa o parcialmente. LOC fija la posición de cada

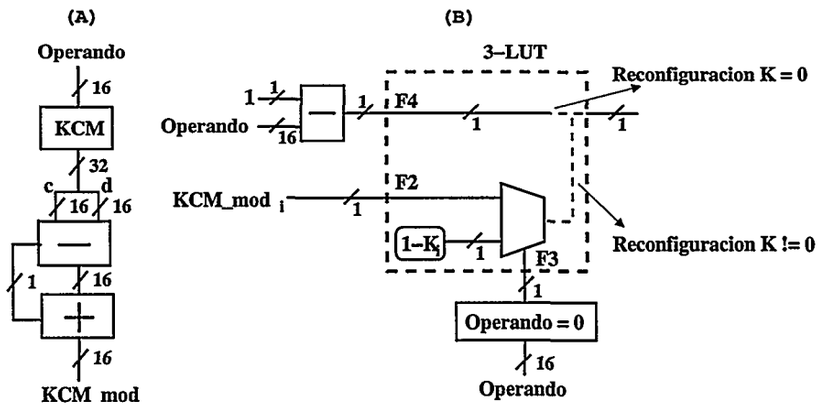


Figura 3.12: Multiplicador módulo $2^{16}+1$ (A)KCM módulo $2^{16}+1$ con operandos no cero (B)Selector de salida

Algoritmo 3 Reconfiguración del multiplicador $2^{16}+1$

```
int aux = (1-constant)&0xFFFF;
int[] f_operand = Expr.F_LUT("~F4");
int[] f_function1 = Expr.F_LUT("~(~F3 | (F3 & F2))");
int[] f_function0 = Expr.F_LUT("~(F3 & F2)");

if(constant == 0)
   jbits.set(row,column,LUT.SLICE0_F,f_operand);
else if(((aux >>bit)&0x1) == 0)
   jbits.set(row,column,LUT.SLICE0_F,f_function0);
else
   jbits.set(row,column,LUT.SLICE0_F,f_function1);
```

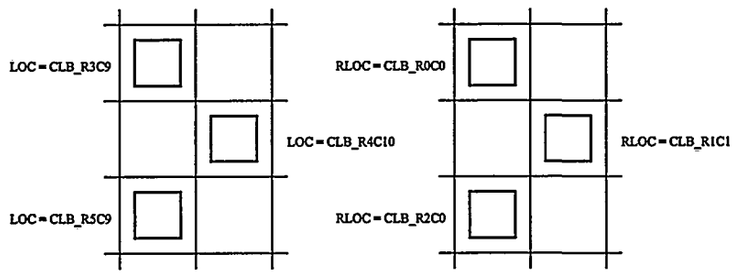


Figura 3.13: Restricciones LOC y RLOC

componente, por lo que es posible conocer inicialmente donde se encuentran, pero es una "restricción fuerte" para la herramienta de *place&route* ya que el emplazamiento de los componentes es fijo, y se hace manualmente. RLOC es una "restricción suave" porque permite al diseñador establecer la posición relativa de cada elemento básico dentro de cada componente. Es por ello que la herramienta puede elegir la posición del componente. Generalmente se obtienen mejores resultados porque el atributo RLOC es más flexible que el atributo LOC. La desventaja es que después de la implementación se deben examinar los resultados para encontrar la posición de cada una de los diferentes componentes a reconfigurar. Estos atributos de posicionamiento relativo son restricciones a nivel *slice*. Si además se emplean las dos LUTs de un *slice*, es necesario emplear el atributo BEL para determinar que función va en cada LUT.

Todos los componentes reconfigurables han sido relativamente posicionados en columnas usando RLOC. Esta restricción de posicionamiento decrementa el número de *frames* que es necesario reconfigurar durante el proceso, minimizando el tiempo de reconfiguración.

3.6.2. Segmentación del algoritmo

La segmentación es necesaria para incrementar el rendimiento. En el caso del algoritmo IDEA esto es sencillo porque su estructura es lineal y sin realimentaciones, al tratarse de una implementación en modo ECB. En una primera aproximación los registros de segmentación pueden ser añadidos detrás de cada operación básica, de este modo el periodo de reloj estará limitado por el retardo de la operación más compleja, el multiplicador módulo $2^{16}+1$. Por lo tanto se hace necesario segmentar el multiplicador.

La única desventaja de la segmentación del algoritmo IDEA es que causa un gran incremento de área, porque muchas de las líneas de datos cruzan los límites de la segmentación. En FPGAs, la segmentación después de una operación generalmente no causa un incremento del área, porque hay

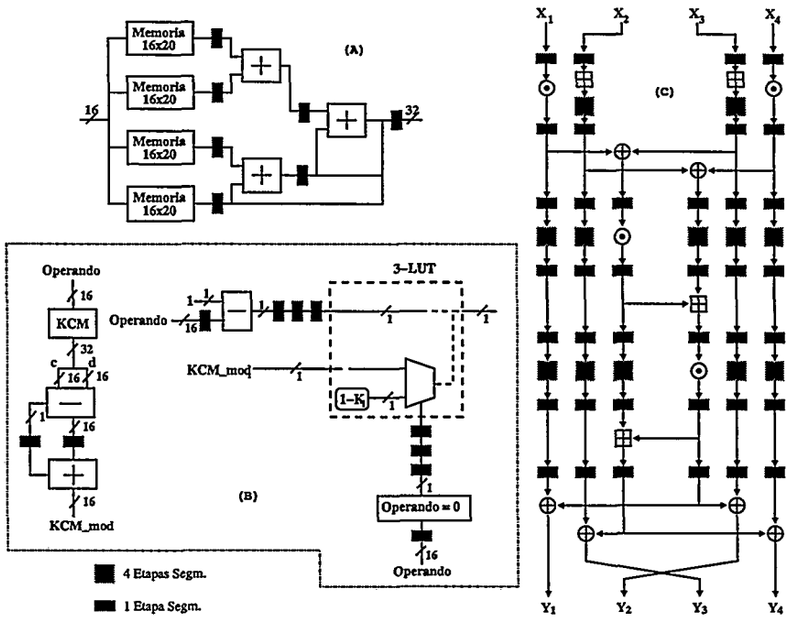


Figura 3.14: Segmentación (A)KCM (B)Multiplicador módulo (C)Round de IDEA

muchos CLBs que tienen las LUTs ocupadas y los *flip-flops* disponibles, pero si la segmentación cruza líneas, los *flip-flops* tienen que ser obtenidos de nuevos *slices* que probablemente tienen sus LUTs sin usar.

Segmentación del multiplicador módulo $2^{16}+1$

El multiplicador módulo es el componente más complejo del algoritmo IDEA, y el que presenta un retardo significativamente grande con respecto a otras operaciones. Para reducir este retardo el multiplicador ha sido segmentado en cuatro etapas. Las primeras tres etapas se corresponden con la segmentación del multiplicador KCM, añadiendo registros después de las memorias 16x20 y las dos etapas de sumadores (figura 3.14A). Esta segmentación es necesario propagarla a los otros bloques (figura 3.14B). Finalmente, otra etapa de segmentación se añade antes del multiplexor (figura 3.14C).

Diseño final segmentado de IDEA

El diseño final del algoritmo IDEA usa 19 etapas de segmentación por round, y 6 etapas de segmentación para la transformación de salida. Esto supone un total de (8x19)+6 = 158 etapas de segmentación, o lo que es lo mismo, hacen falta 158 ciclos de reloj para que los 64 bits de entrada lleguen a la salida. Esta latencia es muy alta para aplicaciones que operen con pequeñas cantidades de datos, pero normalmente los algoritmos del criptografía trabajan con enormes cantidades de datos, por lo que no representa un problema.

El diseño final del algoritmo IDEA completamente segmentado que se ha realizado puede ser implementado en una XCV600, empleando un 87 % de área y una frecuencia de operación de 131 MHz, lo que supone un destacado rendimiento de 8,3 Gbits/seg. Este resultado se compara satisfactoriamente con todas las implementaciones anteriores del algoritmo IDEA.

3.7. IDEA en Virtex-II: Multiplicadores embebidos

La propia evolución de la tecnológica de fabricación de circuitos integrados hace que cada nueva familia de FPGA que sale al mercado suponga un incremento de prestaciones. Estas mejoras intrínsecas al proceso de fabricación hacen que los dispositivos de alta gama de cada familia, dispongan de mayor capacidad de lógica y operen a frecuencias de funcionamiento cada vez más altas. De mayor interés resulta analizar las nuevas posibilidades que presentan y permiten desarrollar un sistema digital complejo en un solo circuito, además de las características particulares de las FPGAs en cuanto a productividad, tiempo de desarrollo y flexibilidad.

Si la característica fundamental de la familia Virtex de Xilinx ha sido la reconfiguración parcial y dinámica, y que se dispone de bloques de memoria distribuida en columnas a lo largo de la FPGA, en Virtex-II se presenta como novedad la existencia de multiplicadores de 18x18 bits embebidos junto con recursos de rutado dedicados para una fácil conexión con bloques de memoria. De este modo, se facilitan las operaciones tipo MAC, que suponen la multiplicación, suma y acumulación de los operandos. Con ello, la familia Virtex-II amplía el nicho de mercado de los circuitos FPGA y permite el desarrollo de aplicaciones en los campos de comunicaciones de datos, cifrado y procesamiento digital de señal.

En las secciones anteriores se ha podido comprobar que todo el esfuerzo a la hora de mejorar el rendimiento del algoritmo IDEA, se centra en la mejora y optimización del multiplicador módulo $2^{16}+1$. Esta optimización tiene sentido en dispositivos de la familia Virtex o anteriores, ya que la multiplicación requiere de una enorme cantidad de lógica. Sin embargo, los dispositivos de la familia Virtex-II incluyen multiplicadores embebidos dentro de la FPGA, lo que permite ahorrar parte de la lógica al no tener que implementar en ella la multiplicación, y al mismo tiempo,

obtener un alto rendimiento en esta operación debido a que se encuentra implementada directamente en hardware. El objetivo de esta sección es analizar la mejora que supone desarrollar una implementación del algoritmo IDEA que haga uso de los multiplicadores embebidos de la familia Virtex-II.

3.7.1. Consideraciones de implementación

Se pretende evaluar el efecto en el rendimiento de los nuevos recursos disponibles en la familia de FPGAs Virtex-II. Para ello, se comparan un conjunto de implementaciones experimentales simplificadas del algoritmo criptográfico IDEA, en las cuales se ha omitido el almacenamiento y generación de subclaves. Se asume que las subclaves se introducen directamente a los multiplicadores y no se han realizado optimizaciones como las propuestas en secciones anteriores. También se supone que existen conexiones de entrada/salida suficientes para una comunicación directa de los datos con el exterior del circuito.

Para la realización de estos experimentos se ha empleado Synplify 7.1 como herramientas de síntesis, e ISE 5.2 para el emplazamiento y rutado. Se han mantenido las opciones que traen por defecto ambas herramientas, siendo posible mejorar los resultados obtenidos en los experimentos sin más que incrementar el nivel de esfuerzo de las herramientas. El objetivo es comparar los resultados, y por esta razón con las opciones por defecto, los valores obtenidos son fácilmente reproducibles experimentalmente.

A lo largo de esta sección, se comparan los rendimientos de las diferentes implementaciones realizadas en una FPGA de la Familia Virtex, con sus equivalentes en Virtex-II, que incorporan los bloques de multiplicadores embebidos. Se eligen los dispositivos de cada familia con recursos similares, y se comparan los resultados del dispositivo de menor tamaño capaz de albergar el diseño, con la misma implementación en miembros de la familia con mayor capacidad.

3.7.2. Implementación del multiplicador módulo $2^{16}+1$

El primer experimento permite establecer la mejora que supone emplear un multiplicador embebido con respecto a la implementación en lógica. Su descripción corresponde a la sentencia VHDL: o <= a * b, y tanto los operandos como el resultado están directamente conectados a IOBs. Asignando al atributo *'syn_multstyle'* el valor *'LUT'* se indica que el multiplicador se implemente en lógica, y con el valor *'Block'*, que utilice un bloque multiplicador embebido.

Los resultados, resumidos en la tabla 3.3, muestran que en Virtex-II con el uso del multiplicador embebido se mejora la velocidad de operación del diseño. La multiplicación en lógica requiere el mismo número de *slices* que en Virtex, pero su conexionado es más eficiente, incluso en un dispositivo más ocupado.

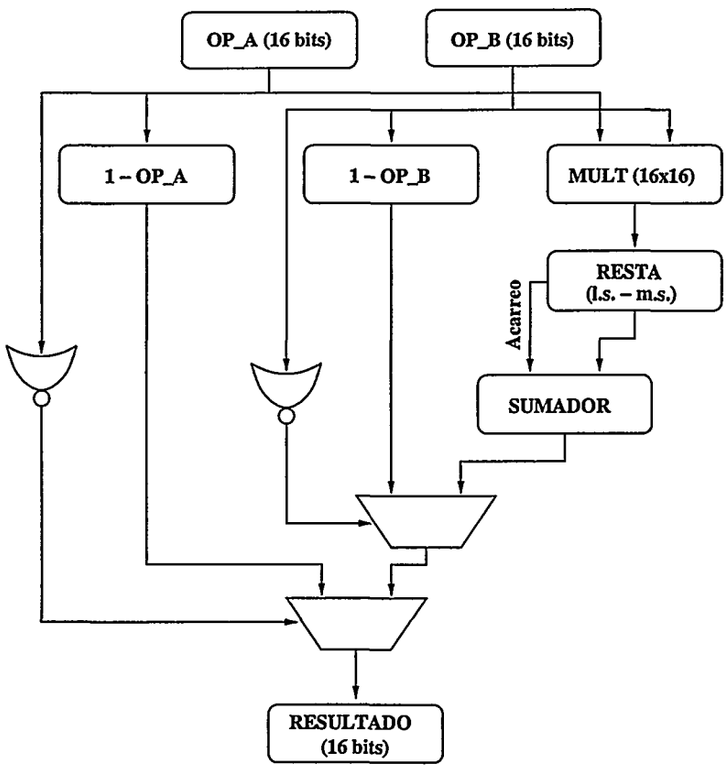


Figura 3.15: Algoritmo Low-High para la implementación de un multiplicador modulo $2^{16}+1$

Familia	Dispositivo	Multiplicador	Recursos	Ocup.	Retardo
Multiplicador 16x16 bits					
Virtex	XCV50-6	LUT	136 slices	17 %	25,1 ns
Virtex-E	XCV50E-6	LUT	136 slices	17 %	26,4 ns
Virtex-II	XC2V40-6	LUT	136 slices	53 %	18,8 ns
Virtex-II	XC2V40-6	Block	1 MULT18x18/0 slices	0 %	12,8 ns
Diseño Combinacional del multiplicador mod 2^{16}+1					
Virtex	XCV50-6	LUT	171 slices	22 %	33,7 ns
Virtex-E	XCV50E-6	LUT	171 slices	22 %	36,4 ns
Virtex-II	XC2V40-6	LUT	171 slices	66 %	25,3 ns
Virtex-II	XC2V40-6	Block	1 MULT18x18/34 slices	13 %	19,9 ns
Diseño Segmentado del multiplicador mod 2^{16}+1					
Virtex	XCV50-6	LUT	260 slices	33 %	12,8 ns
Virtex-E	XCV50E-6	LUT	260 slices	22 %	14,4 ns
Virtex-II	XC2V80-6	LUT	268 slices	52 %	9,3 ns
Virtex-II	XC2V40-6	Block	1 MULT18x18S/112 slices	43 %	5,8 ns

Tabla 3.3: Resultados de implementación del multiplicador módulo 2^{16}+1 en Virtex, Virtex-E y Virtex-II

Para la implementación del multiplicador módulo 2^{16}+1 se ha elegido el algoritmo Low-High (figura 3.15), presentado anteriormente. Este algoritmo requiere añadir al diseño anterior algunos elementos adicionales, lo que se traduce en un camino crítico mayor, o lo que es lo mismo, un menor rendimiento al considerar en su conjunto la operación multiplicador módulo 2^{16}+1. Para disminuir los retardos, se ha realizado adicionalmente un diseño segmentado en 5 etapas: entrada, resultado del multiplicador, resultado de la resta (y comparación), resultado de la suma y salida. La tabla 3.3 muestra los resultados para estos nuevos diseños.

La implementación combinacional presenta el mismo comportamiento obtenido con el multiplicador simple, lo que significa que la lógica que añade la operación del módulo afecta a todos por igual. En la implementación segmentada se obtienen muy buenos resultados para la familia Virtex-II, ya sea en la implementación mediante lógica o con el uso del multiplicador embebido. Como resultado significativo destaca la operación multiplicación módulo 2^{16}+1 segmentada empleando un multiplicador embebido, que alcanza una frecuencia de operación superior a 170 MHz.

3.7.3. Implementación desenrollando las 8 fases de IDEA

En el siguiente experimento se compara una implementación, para diferentes dispositivos de las dos familias, que desenrolla las 8 fases del algoritmo IDEA en su versión combinacional y en

Familia	Dispositivo	MULT18x18	Recursos	Ocup.	Retardo	Rendimiento Mbits/s
Diseño Combinacional IDEA						
Virtex	XCV1000-6	No Hay	6488 slices	52 %	754,6 ns	84,8
Virtex-E	XCV1000-6	No Hay	6472 slices	52 %	788,8 ns	81,1
Virtex-II	XC2V1500-6	No	6402 slices	83 %	515,9 ns	124,1
Virtex-II	XC2V1000-6	Si	34 MULT18x18 1745 slices	34 %	452,7 ns	141,4
Diseño Segmentado IDEA						
Virtex	XCV1000-6	No Hay	9624 slices	78 %	17,3 ns	3699,4
Virtex-E	XCV1000-6	No Hay	9624 slices	78 %	15,3 ns	4183,0
Virtex-II	XC2V3000-6	No	9897 slices	69 %	9,4 ns	6808,5
Virtex-II	XC2V1500-6	Si	34 MULT18x18S 4573 slices	59 %	7,9 ns	8101,3

Tabla 3.4: Resultados de implementación del algoritmo IDEA en Virtex, Virtex-E y Virtex-II

otra segmentada. Para esta última, además de la segmentación correspondiente a las 8 fases de IDEA, se mantienen 5 etapas de segmentación en cada multiplicación módulo $2^{16}+1$. Esto supone 15 niveles por fase en vez de 20, ya que dos de los cuatro multiplicadores operan en paralelo, y un nivel más debido a que también se ha registrado la suma de salida de cada fase. Todo ello involucra 16 niveles de segmentación por fase, más la transformación final que emplea 5 niveles adicionales dando como resultado un diseño de IDEA segmentado con una profundidad de 133 etapas.

La tabla 3.4 compara los resultados obtenidos. En todos los casos la familia Virtex-II es la mejor opción. La versión combinacional presenta un rendimiento 46 % mejor que en Virtex, y un 66 % mejor si se usan los multiplicadores embebidos. El mejor rendimiento se alcanza con el diseño segmentado sobre Virtex-II, con un 15 % de mejora si se comparan los valores del diseño que usa bloques embebidos de multiplicadores frente al que sólo usa lógica.

Con la idea de comprobar si el resultado del diseño segmentado con multiplicadores embebidos presenta alguna dependencia de la ocupación del dispositivo, de la cantidad de multiplicadores, o de su disposición en la FPGA, se repite el experimento para todos los dispositivos de la familia Virtex-II en los que es posible implementar el algoritmo IDEA completo. Para la realización de estos diseños se obliga a la herramienta a no incluir *flip-flops* en los IOBs, a fin de no depender del tamaño de la FPGA, y se deja total libertad de emplazamiento. Los resultados obtenidos se muestran en la figura 3.16, donde el tamaño del diseño es de 9913 *slices* si se usan multiplicadores en lógica, y 4589 *slices* si se usan los multiplicadores embebidos.

Diferenciando dos grupos de dispositivos dependiendo de si tienen 4 o 6 columnas de multiplicadores embebidos, la figura 3.16 muestra dos comportamientos. Por una parte, aumenta el

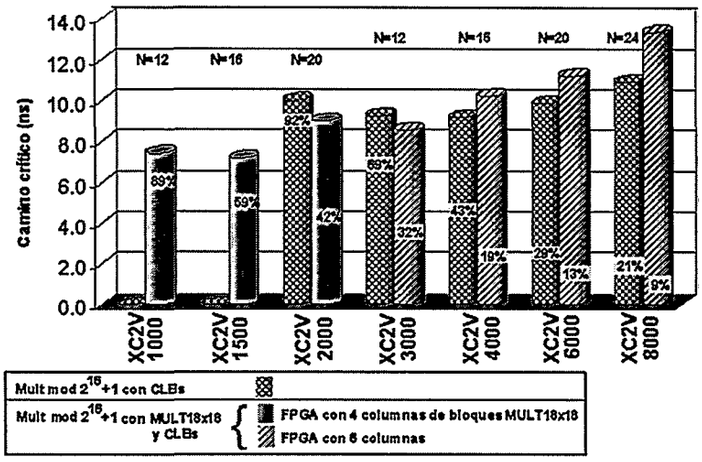


Figura 3.16: El algoritmo IDEA en diferentes dispositivos Virtex-II

retardo del diseño que usa multiplicadores embebidos para cada uno de los grupos a medida que se incrementa la distancia N, indicativa del número de columnas de *slices* entre columnas de multiplicadores. Para un mismo N los rendimientos son inferiores en el grupo de dispositivos con 6 columnas de multiplicadores. Por otra parte, para el diseño que implementa los multiplicadores en lógica, se observa un decremento del rendimiento al aumentar la ocupación y al aumentar la separación N. Se alcanza un compromiso en el dispositivo XC2V4000 con un 43 % de ocupación y una separación de 16 columnas de CLBs. Para un mismo N, en este experimento el único valor comparable entre los grupos es N=20, se obtiene el mismo rendimiento. Sorprende que en los dispositivos de gran capacidad los resultados sean mejores para la versión que implementa los multiplicadores en lógica.

3.7.4. Emplazamiento relativo de un core IDEA

En este experimento se considera la implementación del algoritmo IDEA como un core que ocupa 80x96 *slices*, un área equivalente al diseño de IDEA segmentado sobre una XC2V1500. El core supone una restricción sobre la lógica pero no sobre la posición de los multiplicadores, y es parte de un diseño mucho más grande. Se proponen dos ubicaciones para el core: en el lateral inferior y en el centro del dispositivo. El hecho de limitar el emplazamiento hace que mejoren notablemente los rendimientos para los dispositivos de mayor tamaño. A diferencia del experi-

Dispositivo	Ocupación	Core Lateral	Core Centrado
XC2V2000-6	44 %	8,3 Gbits/s	9,0 Gbits/s
XC2V3000-6	33 %	8,5 Gbits/s	8,0 Gbits/s
XC2V4000-6	20 %	8,6 Gbits/s	8,2 Gbits/s
XC2V6000-6	14 %	7,4 Gbits/s	8,2 Gbits/s
XC2V8000-6	10 %	5,8 Gbits/s	6,7 Gbits/s

Tabla 3.5: Resultados del emplazamiento relativo con un core IDEA en diferentes dispositivos Virtex-II

mento anterior, ahora en todos los casos los resultados son mejores si se utilizan multiplicadores embebidos en lugar de implementarlos en lógica.

La disposición en un lateral muestra la dependencia del rendimiento con respecto a la distancia N entre columnas de multiplicadores. Los resultados resumidos en la tabla 3.5, de nuevo siguen el comportamiento discutido para la figura 3.16.

El emplazamiento en el centro equivale a colocar un XC2V1500 en el interior de cada uno de los dispositivos. En este caso se utilizan principalmente las 4 columnas centrales de multiplicadores, lo que permite mejorar el rendimiento con respecto a la ubicación anterior. Esto se explica teniendo en cuenta que la dependencia de la distancia entre columnas es menor, al poder concentrar más multiplicadores en las columnas centrales. El caso del dispositivo XC2V3000 es especial, y emplea las 6 columnas disponibles porque la distancia entre ellas se corresponde con la menor posible (N=12). Una prueba adicional expandiendo el core al ancho del dispositivo, ha permitido mejorar los rendimientos para el dispositivo XC2V3000 a un valor de 8,9 Gbits/s y del XC2V8000 hasta 7,5 Gbits/s.

3.7.5. Conclusiones sobre Virtex-II

Las pruebas realizadas indican en todos los casos que la familia Virtex-II permite mejorar notablemente el rendimiento de la aplicación propuesta. El uso de multiplicadores embebidos mejora el rendimiento del algoritmo, aunque en dispositivos de gran capacidad el resultado se ve afectado por la disposición de los bloques de multiplicadores. Para un diseño segmentado del algoritmo IDEA, los parámetros a tener en cuenta son la ocupación del dispositivo y el número de columnas de *slices* que separan las columnas de bloques de multiplicadores. Para los dispositivos de mayor tamaño se obtienen los peores resultados, debido a que la herramienta de diseño utiliza por defecto columnas de multiplicadores separadas una mayor distancia. Forzando que la lógica del diseño quede delimitada en un área determinada, se obtiene una mejora importante. Esto solución obliga a la herramienta a emplear columnas de multiplicadores más próximas (o incluidas) al área defi-

Implementación	Slices	Dispositivo	Frecuencia	Throughput
Combinacional	6863	55 % XCV1000-6	1,4 MHz	0,08 Gbits/s
Segmentada	7410	60 % XCV1000-6	24,5 MHz	1,5 Gbits/s
JBits	6312	51 % XCV1000-6	20 MHz	1,28 Gbits/s
VHDL + JBits	5980	87 % XCV600-6	131,1 MHz	8,3 Gbits/s
Virtex-II	4589	44 % XC2V2000-6	140,6 MHz	9,0 Gbits/s

Tabla 3.6: Resultados de las diferentes implementaciones del algoritmo IDEA realizadas

nida. El mejor emplazamiento corresponde a un diseño centrado, y en el caso más desfavorable, experimenta un decremento de un 20 % en rendimiento al desplazarlo a un extremo.

3.8. Resultados

Los resultados de cada uno de los diseños del algoritmo IDEA implementados se muestran en la tabla 3.6. Todos ellos se comparan satisfactoriamente con los resultados de la bibliografía presentados en la sección 3.2. Destaca especialmente la implementación en VHDL para reconfiguración parcial (VHDL + JBits), que emplea un 87 % de una XCV600 y trabaja a 131 MHz, obteniendo un destacado rendimiento de 8,3 Gbits/seg. Este resultado es muy superior a la mejor implementación publicada sobre Virtex [157], y además, requiere un 34 % menos de área.

La última fila de la tabla 3.6 muestra el mejor resultado sobre la nueva familia Virtex-II. Este resultado también es el mejor de todos los presentados en la tabla, lo que demuestra que los multiplicadores de esta familia son adecuados para acelerar de forma importante el algoritmo IDEA. Las pruebas sobre Virtex-II vienen ha demostrar que con la llegada de nuevas familias de dispositivos reconfigurables, se pueden obtener mejores rendimientos debido a las nuevas características que muchas de ellas presentan. En particular, la mejor implementación en Virtex-II obtenida en nuestro estudio es muy superior a la mejor versión disponible en la bibliografía sobre Virtex-II [161], y muestra un rendimiento algo superior al diseño sobre Virtex empleando reconfiguración parcial. Sin embargo, una posterior adaptación de este último diseño reconfigurable para la familia Virtex-II, nos permite obtener un rendimiento de 9,7 Gbits/s (151,7 MHz) y un área de 6128 *slices* (79 % ocupación XC2V1500). Este nuevo diseño ha sido viable en la actualidad, al disponerse del API JBits para la familia Virtex-II. Del mismo modo y también para la familia Virtex-II, se ha desarrollado un nuevo diseño del algoritmo IDEA adaptado al procesador soft-core MicroBlaze, y empleando el dispositivo ICAP (periférico HWICAP) y el API Xilinx Partial Reconfiguration Toolkit (XPART) [16] ha sido posible realizar la reconfiguración del diseño, eliminando la necesidad de emplear un PC externo.

En definitiva, se puede concluir que se dispone de los diseños del algoritmo IDEA que ofrecen un mayor rendimiento y menor uso de recursos lógicos publicados hasta la fecha.

3.9. Conclusiones

La disponibilidad de plataformas reconfigurables con dispositivos de la familia Virtex abre nuevas posibilidades en la adaptación de algoritmos software a hardware, simplificando el diseño y permitiendo sacar un mejor partido del dispositivo reconfigurable. De este modo, es posible generar nuevos diseños en función de la evolución del algoritmo o del tipo de datos a procesar. Aunque el soporte para la reconfiguración parcial solo se encuentra disponible en un número reducido de herramientas, y en muchos casos el uso de estas herramientas en el diseño de implementaciones en hardware requiere de un esfuerzo extra, la reconfiguración parcial tiene un gran futuro por delante.

Particularmente, en el caso del algoritmo IDEA, el uso de reconfiguración parcial ha permitido adaptar las unidades operacionales que hacen uso de las subclaves, reduciendo el número de recursos necesarios, y al mismo tiempo obtener un alto rendimiento. Además, la adaptación de las unidades operacionales conlleva la eliminación de la lógica necesaria para la generación de las subclaves. El uso de multiplicadores por constante ha supuesto un ahorro del 10 % al comparar el diseño con JBits frente a al diseño segmentado en VHDL, y un 20 % al comparar el diseño en VHDL + JBits frente al diseño segmentado en VHDL. Además, al comparar estos últimos diseños, se puede apreciar que el rendimiento es casi seis veces mejor a favor de la solución reconfigurable.

Por otro lado, la irrupción de la familia Virtex-II parece no justificar todo el trabajo realizado sobre Virtex y el uso de reconfiguración parcial, ya que al comparar ambas implementaciones, Virtex + Reconfiguración frente a Virtex-II, los multiplicadores de Virtex-II convierten a esta familia en la candidata ideal para la implementación de algoritmos que hacen uso de la operación de multiplicación. Sin embargo, la ventaja obtenida no es muy superior, y no debemos olvidar que en los diseños sobre Virtex-II se ha eliminado la generación de subclaves, lo cual podría afectar al rendimiento final. Una posterior adaptación del diseño VHDL + JBits a Virtex-II ha permitido obtener el mejor rendimiento del algoritmo IDEA hasta la fecha: 9,7 Gbits/s.

Para finalizar este capítulo es necesario señalar que las implementaciones sobre procesadores de propósito general no se encuentran a la altura del rendimiento sobre hardware reconfigurable, convirtiendo a estos dispositivos en una solución adecuada para la mejora del rendimiento de sistemas mixtos basados en procesadores de propósito general y coprocesadores en hardware reconfigurable. Algo que tiene mucha importancia en sistemas embebidos para aplicaciones de seguridad, lo que constituye el objetivo de los sucesivos capítulos.

Capítulo 4

Sistemas Criptográficos basados en SCPs

4.1. Introducción

La mayor parte de los trabajos realizados en el ámbito de las aplicaciones criptográficas sobre FPGAs se basan principalmente, al igual que se ha hecho en el capítulo anterior de esta tesis, en la capacidad de acelerar el rendimiento empleando soluciones hardware puras sobre FPGA frente a procesadores de propósito general [167]. Sin embargo, los dispositivos reconfigurables actuales disponen de suficiente cantidad de lógica para incluir sistemas completos basados en uno o varios procesadores (hard-core o soft-core), los cuales pueden disponer de diferentes arquitecturas o incluso operar a diferentes velocidades de reloj, junto con una gran cantidad de lógica. Esta capacidad de incluir procesadores y lógica programable en un mismo dispositivo reconfigurable permite diseñar verdaderas soluciones on-chip que se conocen como CSoC [3, 4].

Los CSoC ofrecen un espacio de diseño en cuyos extremos se encuentra la solución software pura ejecutada en procesadores de propósito general, que no presenta un excesivo gasto en área, y la implementación hardware pura ejecutándose en lógica reconfigurable, que permite obtener el mejor rendimiento pero requiere de una gran cantidad de área. Como se muestra en la figura 4.1, una tarea puede ser ejecutada en un procesador embebido o implementada satisfactoriamente como un circuito hardware dedicado. Sin embargo, entre ambas soluciones existen varios parámetros de diseño que es necesario tener en cuenta: la velocidad del procesador, la velocidad de la lógica reconfigurable, la velocidad de las memorias o la velocidad de los buses de comunicaciones entre el procesador y la lógica desarrollada por el usuario.

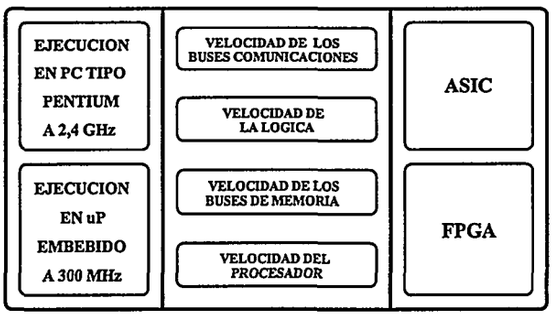


Figura 4.1: Factores que limitan los CSoC

En este capítulo se exploran las diferentes posibilidades que ofrecen los CSoC frente a los sistemas embebidos tradicionales. La mayor parte de los sistemas embebidos se basan en procesadores hard-core, sin embargo, este capítulo se centra en los CSoC que hacen uso de procesadores implementados en lógica reconfigurable o SCPs, los cuales al igual que el dispositivo reconfigurable en el que se implementan, disponen de una mayor flexibilidad porque permiten modificar su configuración para adaptarse de forma adecuada a los objetivos de la aplicación a ejecutar.

4.2. SCPs en aplicaciones criptográficas

Como viene siendo habitual en esta tesis, se propone estudiar el rendimiento de un conjunto de algoritmos de criptografía para conocer las posibilidades que ofrecen los SCPs en la ejecución de aplicaciones seguras. Para ello se han seleccionado los procesadores MicroBlaze de Xilinx y LEON2 de Gaisler Research, y se pretende evaluar su capacidad para mejorar su arquitectura interna mediante la adición de unidades hardware específicas, ya sea dentro de la propia arquitectura del procesador, o mediante las distintas interfaces que permiten una conexión a diferentes niveles dentro del esquema del procesador. Además de los SCPs, los diferentes CSoC están compuestos de un conjunto de periféricos estándar para el acceso a los recursos del sistema como memorias, puertos de E/S, etc. El resto de la lógica reconfigurable queda disponible para el diseño de hardware a medida.

Como complemento a los sistemas anteriores, se ha desarrollado un sistema basado en PowerPC sobre una FPGA de la familia Virtex-II Pro, el cual puede ser considerado como un sistema embebido tradicional, para su comparación con los sistemas basados en SCPs. Este sistema permite realizar una comparación más adecuada de los procesadores hard-core frente a los soft-core,

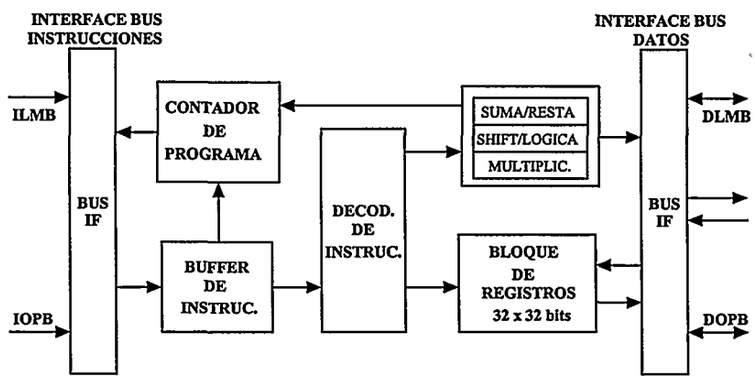


Figura 4.2: Arquitectura del procesador MicroBlaze

al ejecutarse ambos sistemas sobre el mismo tipo de dispositivos reconfigurables. Finalmente, se tendrán en cuenta los resultados sobre PC, la principal plataforma de desarrollo actual.

4.2.1. El procesador Xilinx MicroBlaze

MicroBlaze es un procesador RISC sintetizable de 32 bits tipo *big endian* con arquitectura Hardvard desarrollado y mantenido por Xilinx. MicroBlaze dispone de una unidad de enteros segmentada en 3 etapas, y un bloque de registros que incluye 32 registros de propósito general de 32 bits. Al estar especialmente desarrollado para FPGAs de Xilinx, presenta una alta optimización para estas FPGAs tanto en área como en frecuencia de operación. Un diagrama de la arquitectura del procesador se puede ver en la figura 4.2. MicroBlaze se distribuye con el Xilinx Embedded Development Kit (EDK) como una *netlist* parametrizable [168].

Modelo de programación: MicroBlaze tiene su propio set de instrucciones (ISA- Instruction Set Architecture) especialmente diseñado para la arquitectura de este procesador. Presenta dos formatos diferentes de instrucciones y dos modos de direccionamiento: inmediato y desplazamiento. El conjunto de instrucciones incluye instrucciones para multiplicación y división, cuyas unidades operacionales pueden ser implementadas en hardware opcionalmente. La multiplicación, que presenta una latencia de 3 ciclos de reloj cuando se implementa por hardware, puede ser solo implementada si la FPGA dispone de multiplicadores embebidos. La división tiene, si se implementa en hardware, una latencia de 34 ciclos de reloj. Las instrucciones de salto necesitan de 1 ciclo de retardo hasta que la condición se comprueba, y el mecanismo de predicción siempre considera

que los saltos son efectivos [168]. El ISA incluye además, instrucciones de lectura y escritura, bloqueantes o no bloqueantes, para el bus FSL (Fast Simplex Link) [169] que permite una rápida comunicación con una o varias unidades hardware a medida.

Sistema caché: La arquitectura caché es Hardvard, y tanto la caché de datos como la caché de instrucciones pueden variar de tamaño entre 2 y 64 Kbytes, con 4 bytes por línea de la caché. Las cachés son de correspondencia directa y soportan bloques individuales en cada línea. Al ser una caché de correspondencia directa, bloquear una línea puede dejar otras direcciones de memoria bloqueadas desde la caché, lo que puede afectar negativamente al rendimiento. La caché de datos opera como una cache de escritura directa (*write-through*) e implementa asignación en escritura (*allocate-on-write*). La caché de MicroBlaze está limitada para cachear únicamente un subespacio continuo de la memoria total.

Interfaz del sistema: Las interfaz del sistema MicroBlaze consiste de los buses denominados: Local Memory Bus (LMB), IBM CoreConnect On-Chip Peripheral Bus v2.0 (OPB) y Fast Simplex Link (FSL). La unidad de enteros se comunica con la memoria Block RAM mediante un acceso de un ciclo a través del bus LMB. El OPB, bus multi-master y multi-slave, provee un mecanismo de interfaz tanto con memorias como con periféricos internos o externos. El bus FSL es accesible a nivel de ISA, y permite una transferencia punto a punto de datos con una latencia de 2 ciclos. MicroBlaze se puede conectar a un gran número de tipos de memoria incluyendo Intel StrataFlash, SRAM, SDRAM y DDR SDRAM; además de a un amplio conjunto de periféricos: Xilinx Microprocessor Debug Module (MDM) para facilitar el debug, Ethernet MAC, Quixilica IEEE-754 single precisión FPU (MicroBlaze 4.00 la incluye por defecto), UARTs, Timers, etc.

Interfaz para coprocesador: Aunque no se presenta como una verdadera interfaz para coprocesador, el FSL [169] provee un mecanismo para el envío y recepción de datos desde/hacia los registros a alta velocidad (figura 4.3). MicroBlaze puede emplear hasta 16 FSLs, 8 de salida y 8 de entrada, cuyo ancho es parametrizable (8, 16 o 32 bits). Además dispone de un bit adicional de control por link que puede ser usado, por ejemplo, para diferenciar si los bits son comandos o datos. La conexión con los periféricos FSL se realiza mediante los buses FSL, los cuales emplean FIFOs como buffer de datos y hacen posible que el procesador y los periféricos operen en diferentes dominios de reloj. La profundidad de la FIFO también es parametrizable, desde 1 a 8192 palabras (hasta 128 si se emplean diferentes relojes para lectura y escritura). Están disponibles dos instrucciones para usar el FSL que son *'get'* y *'put'*: *'get'* para cargar los registros con datos de la interfaz FSL, *'put'* para escribir el contenido de los registros en un determinado canal FSL. Cada instrucción tiene cuatro variantes, dependiendo de si la operación es o no bloqueante y si el bit de

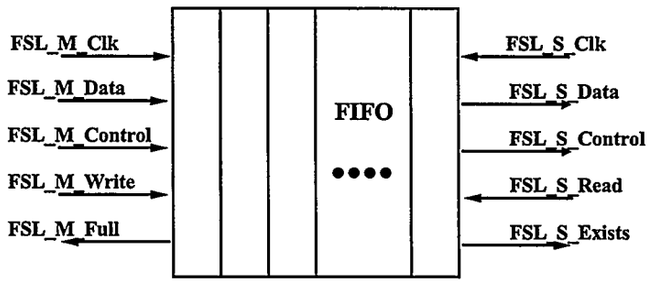


Figura 4.3: Interfaz FSL

control esta a cero o a uno. En todos los casos no bloqueantes la latencia es de dos ciclos, lo que supone una velocidad de transferencia de 200 MB/sec para un MicroBlaze a 100 MHz.

4.2.2. El procesador LEON

LEON es un procesador RISC de 32 bits que cumple con la arquitectura SPARC V8 [170], y usa ordenación de bytes tipo *big endian* como se especifica en el manual de referencia de SPARC V8. La figura 4.4 muestra la arquitectura del procesador LEON2, que es la versión empleada en esta tesis.

LEON2 es un procesador sintetizable desarrollado por la Agencia Espacial Europea (-ESA-European Space Agency) y mantenido por Gaisler Research. El procesador fue originalmente desarrollado como un procesador tolerante a fallos para aplicaciones espaciales. Esta tesis se basa en la versión no tolerante a fallos que se distribuye bajo licencia GNU GPL, de modo que se distribuye libremente como modelo VHDL desde la web de Gaisler Research [171]. LEON2 puede ser implementado en FPGAs o en ASICs.

Modelo de programación LEON2 utiliza el set de instrucciones de la arquitectura SPARC V8. Los objetivos de diseño del ISA SPARC V8 fueron permitir optimizaciones software fácilmente mediante un compilador, y facilitar las implementaciones hardware segmentadas. El ISA tiene tres formatos diferentes de instrucción y tres modos de direccionamiento: inmediato, desplazamiento e indexado. Las instrucciones de salto no necesitan un slot de retardo [170]. El ISA incluye instrucciones para operaciones de multiplicar y acumular (MAC), multiplicación y división. LEON2 implementa opcionalmente la unidad funcional MAC con entrada de 16x16 bits y un acumulador de 40 bits. La operación de multiplicación puede ser implementada de seis formas diferentes en

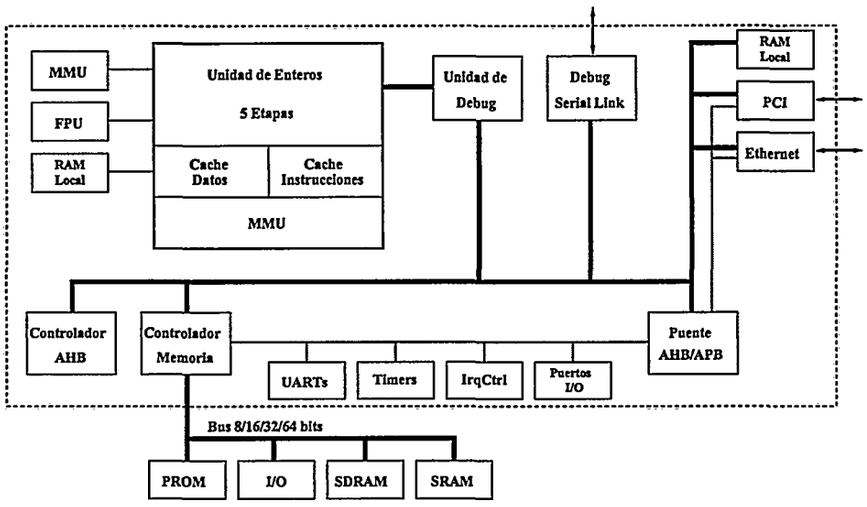


Figura 4.4: Arquitectura de un sistema basado en procesador LEON2

hardware, como se muestra en la tabla 4.1. La instrucción de división también se pude implementar en hardware como un divisor radix-2, cuyo resultado presenta una latencia de 35 ciclos de reloj [8]. El ISA incluye soporte para dos coprocesadores: un coprocesador en coma flotante (FPU) y un coprocesador de usuario a medida. La FPU se puede escoger entre las siguientes: GRFPU de Gaisler Research, Meiko FPU de Sun Microsystems y la incompleta de código libre LTH FPU desarrollada por Lunds Tekniska Högskola.

Unidad de enteros: La unidad de enteros implementa un conjunto de instrucciones ISA SPARC V8 con un pipeline de 5 etapas. El bloque de registros del procesador LEON2 está distribuido en ventanas que se pueden configurar en el rango de 2 a 32 ventanas de registros. Cada ventana de registros tiene 8 registros de entrada, 8 registros de salida y 8 registros locales. Los registros de entrada y salida son compartidos entre ventanas adyacentes. Además de los registros mencionados, existen 8 registros globales. Esto significa que el programador en cada instante puede acceder a 32 registros. El número total de registros necesarios puede ser calculado como 8+16*NWINDOWS, donde NWINDOWS es el número de ventanas. Esto da un total de 40-250 registros dependiendo de cantidad de ventanas de registros configuradas.

Configuración	Latencia (clk)	Área aprox. (K puertas)
iterativa	35	1000
M16x16 + pipeline reg	5	6500
m16x16	4	6000
m32x8	4	5000
m32x16	2	9000
m32x32	1	15000

Tabla 4.1: Configuraciones del multiplicador hardware en el procesador LEON2

Sistema caché: El sistema caché presenta una arquitectura Hardvard con 1 a 64 Kbytes por vía tanto para la caché de instrucciones como para la de datos. Cada línea de la caché puede contener entre 4 y 8 subbloques, cada uno de 4 bytes. Las caches pueden ser configuradas como correspondencia directa o completamente asociativa de 2 a 4 vías. En una configuración multivía se pueden emplear tres políticas de reemplazo: las menos recientemente usada (LRU - Least Recently Used), la menos recientemente reemplazada (LRR - Least Recently Replaced) y pseudo-aleatoria. Cuando se usa LRR, la caché solo se puede configurar como completamente asociativa de 2 vías. Para reducir la latencia de error en la caché de instrucciones, el dato se envía al procesador al mismo tiempo que se escribe en la caché. En el caso de fallo en la caché de datos, solo el subbloque solicitado se captura. La caché de datos usa una política de escritura directa (*write-trhough*). Las caches tienen soporte para un bloqueo individual de la línea de la caché en las configuraciones multivía. Para prevenir que una dirección específica de memoria se bloquee desde la caché, la última línea de cada vía no puede ser bloqueada. Para mantener la consistencia de la caché cuando hay varias unidades en el bus AHB (AMBA High Speed Bus) con capacidad para escribir en memoria, la caché puede realizar "espionaje" (*snooping*) en el bus AHB. Este "espionaje" solo está disponible cuando la MMU (Memory Management Unit) está deshabilitada, por lo que la caché es virtualmente direccionable. Para minimizar la parada del pipeline causada por las instrucciones de almacenamiento, se usa un buffer de doble palabra.

Unidad de gestión de memoria: La unidad de gestión de memoria o MMU puede ser habilitada para proveer mecanismos de protección de memoria, los cuales son requeridos por la mayoría de los sistemas operativos avanzados. La MMU puede ser configurada para hacer un uso compartido o separado del TLB (Translation Lookaside Buffer) para memorias de datos e instrucciones. El TLB es completamente asociativo, y el número de entradas puede ser configurado entre 2 y 32. La MMU soporta tamaños de página de 4 Kbytes, 256 Kbytes y 16 MBbytes.

Interfaz del sistema: La unidad de enteros se comunica con la memoria y otros periféricos mediante el AMBA Advanced High performance BUS (AMBA-2.0 AHB) y el AMBA Advanced Peripheral Bus (AMBA-2.0 APB). El AMBA-2.0 AHB bus conecta los periféricos de alta velocidad, controladores DMA, memorias on-chip e interfaces. El bus tiene operación segmentada y soporta transferencias en ráfaga (*burst*), múltiples maestros y transacciones partidas. El AMBA-2.0 APB bus está optimizado para un mínimo consumo y usa una interfaz de poca complejidad para soportar funciones de periférico. El protocolo está diseñado para periféricos de propósito general. Dispone de un controlador de memoria que soporta varios tipos de memoria incluyendo PROM, SRAM y SDRAM (hasta 2 bancos de PC100/PC133), además de un conjunto de unidades adicionales, algunas de ellas comentadas anteriormente:

- La Unidad de Soporte de Debug (DSU - Debug Support Unit), disponible para un fácil debug.
- Interfaz PCI.
- Ethernet MAC.
- Memoria RAM on-chip para un acceso rápido.
- GRFPU y Meiko FPU (ambas compatibles con el estándar IEEE-754) y la incompleta LTH FPU.

Consumo: LEON2 dispone de un modo de apagado. Cuando el procesador recibe una interrupción no enmascarable con una prioridad específica se activa. En el modo de apagado, la unidad de enteros se encuentra inactiva.

Interfaz genérica del coprocesador: El procesador LEON2 provee una interfaz genérica que permite conectar un coprocesador de usuario. Esta interfaz se encuentra definida en [8] y permite que el coprocesador ejecute sus instrucciones en paralelo con la unidad de enteros (IU). La figura 4.5 muestra el diagrama de las señales y tiempos para todas las entradas y salidas del coprocesador. Como se puede ver, el diagrama comienza con la activación en alto de la señal '*star*' en un flanco de reloj. Durante un ciclo de reloj se mantiene el código de la operación disponible en la entrada '*opcode*'. En el siguiente flanco de reloj, se activa la señal '*load*', indicando que los operandos se encuentran disponibles en las entradas '*op1*' y '*op2*'. Al mismo tiempo que la señal '*load*' se activa, es necesario activar la salida '*busy*', que debe mantenerse activa hasta que el resultado se encuentre disponible. Los códigos de condición y excepción se encuentran disponibles en las correspondientes salidas, *'cc'* y *'exc'*, respectivamente . Al presentar una arquitectura RISC, una de

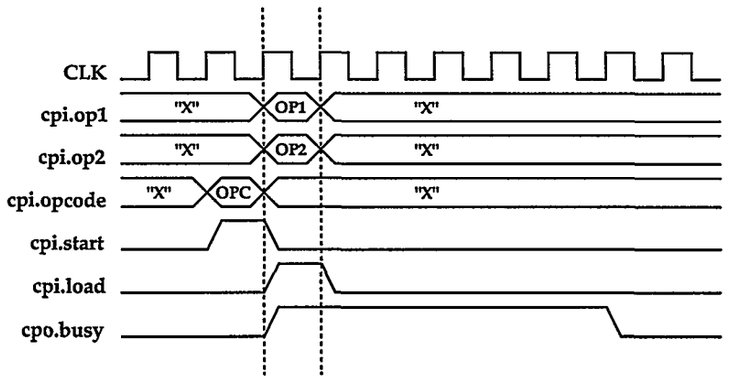


Figura 4.5: Interfaz genérica para coprocesador en LEON2

las características del estándar SPARC es que todos los tipos de operaciones se realizan exclusivamente de registro a registro. Para facilitar esto, se incluyen un conjunto de 32 registros de 32 bits en el diseño de la unidad coprocesador, como se muestra en la figura 4.6. Existen instrucciones específicas del coprocesador '*load*' y '*store*', que se emplean para mover datos entre los registros del coprocesador y la memoria. Cualquier operación básica sobre el coprocesador sigue los siguientes pasos:

1. Carga de los operandos en los registros del coprocesador.
2. Activación de la señal '*start*' conjuntamente con el código de operación para iniciar la ejecución.
3. Después de terminar la operación el coprocesador escribe el resultado en los registros del coprocesador.
4. Una operación de '*store*' escribe el resultado en memoria.

Códigos de operación del coprocesador: El modo de implementar los comandos del coprocesador se define en detalle en [170]. Se dispone de dos comandos de operación del coprocesador, '*cpop1*' y '*cpop2*', cada uno de los cuales dispone de un campo para el código de operación de 9 bits. El formato de las dos instrucciones se muestra en la tabla 4.2. En esta tabla '*rd*' representa el registro destino, '*rs1*' el registro operando 1 y '*rs2*' el registro operando 2. El código de operación de 10 bits pasado al coprocesador se construye añadiendo un 0 o 1, para '*cpop1*' o '*cpop2*'

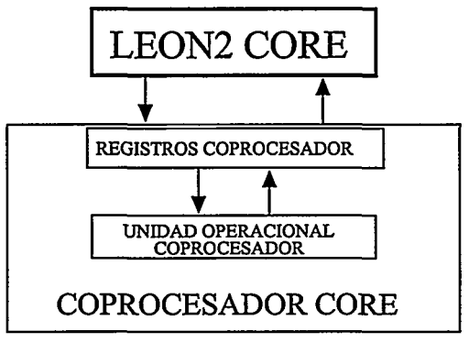


Figura 4.6: Esquema de registros de la interfaz genérica de coprocesador

Comando	31-30	29-25	24-19	18-14	13-5	4-0
cpop1	10	rd	110110	rs1	opc	rs2
cpop2	10	rd	110111	rs1	opc	rs2

Tabla 4.2: Formato de las instrucciones del coprocesador

respectivamente, seguido del código de operación de 9 bits suministrado por estos comandos en el campo '*opc*', bits 5 al 13.

4.2.3. El procesador PowerPC 405

El PowerPC 405 es una implementación de 32 bits de la arquitectura para sistemas embebidos PowerPC derivada de la arquitectura PowerPC. La arquitectura PowerPC provee un modelo software que asegura la compatibilidad entre toda la familia de microprocesadores, definiendo parámetros que garantizan la compatibilidad de los procesadores a nivel de programa, y permitiendo cierta flexibilidad en el desarrollo de variantes de la arquitectura PowerPC para cumplir ciertos requerimientos de mercado. Específicamente la versión incluida en las FPGAs Virtex-II Pro es el PowerPC 405D5, cuyo diagrama de organización se muestra en la figura 4.7.

Las principales características del PowerPC 405D5 son:

- Unidad de enteros completamente compatible con la arquitectura PowerPC. Arquitectura de 32 bits que incluye 32 registros de propósito general de 32 bits.
- Extensiones de la arquitectura para un completo soporte de aplicaciones embebidas: opera-

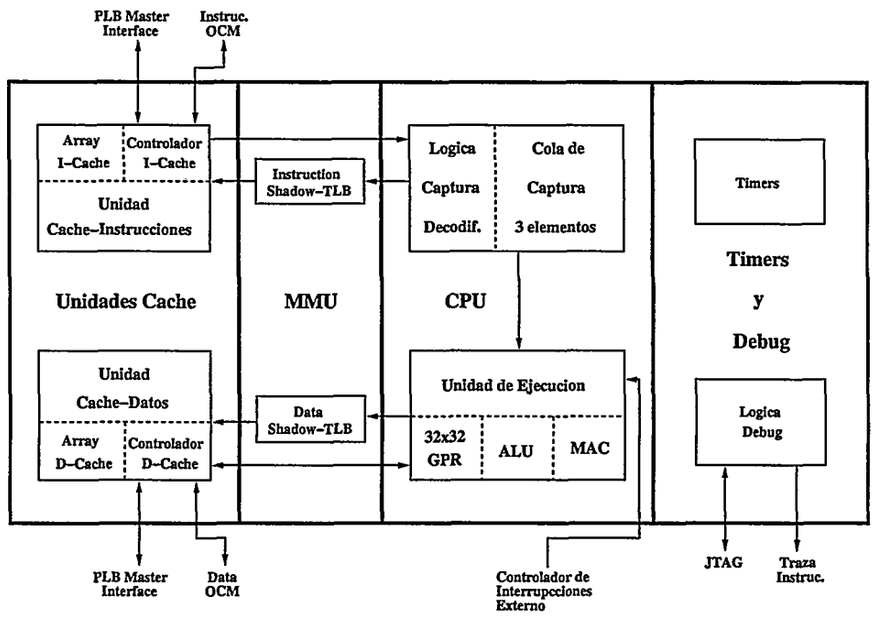


Figura 4.7: Arquitectura del procesador PowerPC disponible en Virtex-II Pro

ciones *little-endian* reales, manejo flexible de memoria, instrucciones MAC para computación intensiva, capacidad de debug.

- Característica para un alto rendimiento: predicción de salto estática, pipeline de 5 etapas con un solo ciclo de ejecución para la mayor parte de las instrucciones incluso *load* y *store*, instrucciones MAC, multiplicación y división por hardware (4 ciclos para la multiplicación, 35 para la división), mejora del manejo de cadenas (*strings*) y dobles palabras.
- Cache de datos e instrucciones de 16 KBytes asociativa de 2 vías y 8 bytes (32 bits) por línea. Política de reemplazo LRU. La cache de datos es configurable: post-escritura (*write back*) o escritura directa (*write through*).
- Soporte para On-Chip Memory (OCM) con rendimiento de acceso idéntico a la caché.
- Modos de direccionamiento: registro indirecto con índice inmediato, registro indirecto con índice y registro indirecto.
- Dispone de MMU para la gestión de un espacio direccionable de 4 Gb. Soporta tamaños de página de 1, 4, 16, 64 y 256 Kbytes y 1 y 4 MBytes.
- Soporte para la arquitectura de bus IBM CoreConnect.

4.3. Trabajo experimental

Se propone un conjunto de experimentos para medir el rendimiento en el cifrado, y posterior descifrado, de bloques de datos hasta operar con un conjunto de datos de 4 MBytes, empleando diferentes algoritmos criptográficos. El programa de test y los datos a cifrar se almacenan en una memoria externa al dispositivo FPGA, representando la situación real de ejecución de este tipo de aplicaciones en sistemas embebidos. Aunque este procedimiento no busca obtener el mejor rendimiento global, al realizarse el cifrado y descifrado para cada bloque de datos (peor situación posible), se corresponde con el funcionamiento de la mayoría de aplicaciones interactivas de cifrado, como SSH [172].

4.3.1. Análisis de los algoritmos criptográficos seleccionados

Los algoritmos seleccionados para este estudio han sido presentados anteriormente en la sección 2.6, y todos ellos forman parte de la mayoría de los protocolos de seguridad actuales empleados en aplicaciones de seguridad. De igual modo, se trata de algoritmos ampliamente referenciados

Algoritmo	XOR AND OR	Suma Resta Mod.	Rotación Fija	Rotación Variable	MUL Mod.	GF Constant MUL	LUT	Tamaño Bloque Int.
IDEA	*	2^{16}			$2^{16}+1$			16
DES/3DES	*		*				6-to-4	32
Rijndael	*		*			GF(2^8)	8-to-8	32
RC4	*							32
MD5	*	2^{32}	*					32
SHA-1	*	2^{32}	*					32
SHA-256	*	2^{32}	*					32

Tabla 4.3: Operaciones básicas de los algoritmos criptográficos seleccionados

en la bibliografía tanto a nivel hardware como software, por lo que es posible comparar los resultados obtenidos con resultados de trabajos similares. Un análisis de cada uno de estos algoritmos permite conocer las operaciones básicas que los define, y que se muestran en la tabla 4.3:

- Operaciones XOR, AND y OR a nivel de bit.
- Suma o resta módulo el valor de entrada de una tabla.
- Desplazamiento o rotación de un número fijo de bits.
- Rotación dependiente de un valor de un número variable de bits.
- Multiplicación módulo el valor de entrada de una tabla.
- Multiplicación en los campos de Galois especificada por un valor de entrada de una tabla.
- Sustitución basada en *look-up-tables*.

Las operaciones booleanas a nivel de bit - XOR, AND y OR - son ampliamente empleadas por todos los algoritmos al igual que las operaciones de suma, resta, y desplazamiento o rotación. Todas estas operaciones son realizadas de forma rápida en procesadores de propósito general. Las operaciones más complejas como las multiplicaciones pueden resultar bastante pesadas para un procesador que no disponga de unidades específicas.

A nivel de implementación de los algoritmos en hardware, la operación más costosa es la multiplicación módulo $2^{16}+1$, presente en el algoritmo IDEA. La implementación en hardware requiere un gasto significativo de recursos, aunque las FPGAs más modernas como Virtex-II incluyen multiplicadores embebidos. Las S-boxes existentes en los algoritmos Rijndael y DES/3DES son fácilmente implementables en RAM o en lógica. Las operaciones basadas en manejo de bits

como las booleanas, rotación o desplazamiento, se realizan muy rápido en hardware y requieren muy pocos recursos. Adicionalmente, la operación de multiplicación de Galois, presente en Rijndael, se puede implementar eficientemente en hardware al ser multiplicaciones por constante.

4.3.2. Plataformas hardware de test

Para poder diseñar los sistemas propuestos ha sido necesario emplear dos plataformas reconfigurables. Por un lado, los diferentes sistemas basados en SCPs han sido implementados sobre la plataforma RC1000PP de Celoxica que incluye una FPGA XCV2000E y que dispone de memoria SRAM. La elección de esta plataforma se debe al tamaño de la FPGA, suficiente para la implementación de los diferentes SCPs y de los diferentes diseños hardware de los algoritmos sin problemas de espacio.

El sistema PowerPC no se puede implementar sobre la plataforma RC1000PP, por lo que la segunda plataforma empleada en estas pruebas es la placa ADM-XRCIIPro-Lite de Alpha Data, que dispone de un FPGA XC2VP20 con dos procesadores PowerPC y memoria externa SDRAM, además de memoria DDR. La memoria SDRAM es la que se ha seleccionado para las pruebas debido a que es la más parecida a la memoria SRAM disponible en la plataforma basada en Virtex-E. Aunque esta segunda plataforma se podría haber empleado para las pruebas de los SCPs, la cantidad de recursos lógicos disponibles y el hecho de que la mayor parte del dispositivo se encuentre ocupado por los dos PowerPC, no hace posible la implementación de los sistemas basados en SCPs cuando se emplean los diferentes cores hardware.

4.3.3. Experimentos

Se han definido dos conjuntos de experimentos:

- Un primer conjunto de experimentos propone la ejecución de una aplicación en lenguaje C de cada algoritmo criptográfico sobre el SCP, añadiendo progresivamente mejoras a la arquitectura para estudiar como afectan al rendimiento las diferentes unidades hardware disponibles: multiplicación, división, caché, etc.

- Un segundo conjunto de experimentos se centra en el uso de diseños hardware específicos para cada uno de los algoritmos seleccionados. Cada SCP presenta diferentes posibilidades de conexión de este hardware específico. El programa que se ejecuta para estos experimentos incluye las líneas de código necesarias para el envío y recepción de los datos al core hardware. El objetivo de este experimento es conocer como afecta al rendimiento de los procesadores el uso de las diferentes interfaces disponibles.

	IDEA	DES	3DES	AES	Blowfish	RC4	MD5	SHA-1	SHA-2
Clave	128	64	192	128	448	64	-	-	-
Bloque	64	64	64	128	64	128	512	512	512

Tabla 4.4: Configuración de los algoritmos criptográficos

Los resultados obtenidos en todos los experimentos se comparan con la ejecución de los programas del primer conjunto de experimentos sobre un sistema PowerPC basado en FPGA, y con el rendimiento obtenido en un PC con procesador Intel Pentium IV a 2.4 GHz. La comparación con el sistema PowerPC permite comprobar si los SCPs son candidatos adecuados para reemplazar a los tradicionales procesadores de propósito general empleados en los sistemas embebidos. La comparación con el PC, aunque no se corresponde con una comparación adecuada debido a las diferencias tecnológicas y frecuencia de operación, sirve para tener una idea de la situación real de los SCPs y de los sistema embebidos respecto al PC actual.

4.4. Funcionamiento de los sistemas criptográficos

Aunque la mayoría de los algoritmos de criptografía, y en particular los algoritmos de clave privada, presentan una determinada configuración en cuanto al tamaño de datos de entrada y tamaño de clave, algunos de los algoritmos seleccionados para estas pruebas pueden operar con diferentes configuraciones, por lo que se ha propuesto una configuración fija para todos los experimentos. La tabla 4.4 muestra la configuración seleccionada para cada algoritmo.

El caso de los algoritmos RC4, MD5, SHA-1 y SHA-2 es especial, ya que en ellos no se puede hablar de bloques de datos, por lo que se ha establecido como bloque de datos la cantidad de datos a cifrar por operación de cifrado/descifrado.

4.4.1. Ejecución de algoritmos completamente software

En este experimento los diferentes algoritmos criptográficos son ejecutados por el procesador del sistema correspondiente, siendo similar a la ejecución tradicional sobre PC. Para ello se han empleando codificaciones en C de los algoritmos, muchas de ellas disponibles de forma gratuita en varios enlaces web [173, 174]. El lenguaje C permite que los diferentes algoritmos sean fácilmente portables a los diferentes sistemas, o procesadores, ya que se dispone de compiladores para todos ellos. En todos los casos, los algoritmos han sido previamente ejecutados en PC para comprobar que todos funcionan adecuadamente, llevando a cabo la ejecución de pruebas que hacen uso de los vectores de test disponibles para verificar que las implementaciones son correctas. Posteriormente,

tras los mínimos cambios para adaptar el código a cada uno de los sistemas, se han repetido las mismas pruebas.

4.4.2. Incorporación de cores criptográficos

En este experimento toda la funcionalidad de los algoritmos se ha implementado en hardware, de modo que los diferentes algoritmos actúen como coprocesadores del procesador principal. De este modo, la funcionalidad del procesador del sistema se limita a las tareas de envío y recepción de datos, envío de clave y en el caso del algoritmo IDEA, a la generación de subclaves. Para ello se han realizados implementaciones hardware de los algoritmos siguiendo la configuración establecida (tabla 4.4). En todos los diseños se ha empleado el lenguaje VHDL y particularmente tres de los cores se han basado en implementaciones iniciales disponibles de forma pública: AES [175], Blowfish [176] y DES/3DES [177]. En todos los casos, los cores se han desarrollado sin dedicar demasiado esfuerzo de diseño, únicamente poniendo como objetivo que puedan operar a la misma frecuencia que el sistema procesador al que se conectan, y en lo posible, manteniendo una buena relación en área. En este caso, las frecuencia de operación requerida es de 50 MHz (frecuencia máxima de un sistema MicroBlaze sobre Virtex-E), y dependiendo de la dificultad para portar la funcionalidad de cada algoritmo al hardware, se han conseguido diferentes resultados.

Los algoritmos IDEA, Blowfish y RC4 siguen el esquema de la figura 4.8. En cada caso, el core consiste en un camino de datos con los componentes operacionales de un round, y una unidad de control secuencial implementada como maquina de estados finitos (FSM). La unidad de control maneja el algoritmo de cifrado de forma completa mediante la ejecución iterativa de los rounds. Adicionalmente, el camino de datos del round ha sido segmentado para alcanzar la frecuencia de reloj a la que opera el sistema. El modo de operación de estos cores es similar excepto en el proceso de generación de las subclaves. El coprocesador IDEA recibe los subclaves previamente calculadas por el procesador y las almacena en el bloque de memoria que se muestra en la figura 4.8. El algoritmo Blowfish genera las subclaves usando el mismo algoritmo de cifrado; consecuentemente el primer paso es usar el core de cifrado para generar las subclaves, y almacenarlas en el bloque de memoria de la clave. Además, al ser las S-boxes del algoritmo Blowfish dependientes de la clave, se hace necesaria una primera etapa de inicialización muy costosa. Aunque el algoritmo RC4 no tiene una unidad específica de round, el diseño cumple con el de la figura 4.8 porque la tarea computacional principal del algoritmo se ha diseñado como un componente round. Este algoritmo modifica la clave con los datos a cifrar por lo que los resultados intermedios se almacenan en el bloque de memoria de la clave.

La figura 4.9 describe la arquitectura de los algoritmos DES, 3DES y AES. Estos cores fueron implementados completamente desplegados y segmentados obteniendo una frecuencia de reloj

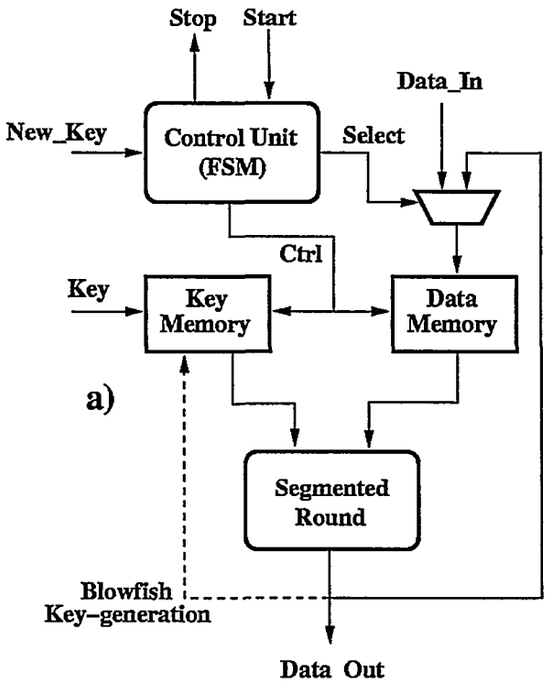


Figura 4.8: Arquitectura del coprocesador basada en FSM con etapa de manejo de la clave

Algoritmo	Etapas	
	Clave	Cifrado
IDEA	Software	175
DES	-	49
3DES	-	147
AES	-	12
BLOWFISH	21106	37
RC4	1537	625
MD5	-	384
SHA-1	-	198
SHA-256	-	166

Tabla 4.5: Características de los cores de cifrado desarrollados

muy superior a la frecuencia de operación del sistema basado en SCP. Una unidad de generación de subclaves fue diseñada para generar las diferentes subclaves empleadas en cada round.

Por último, los algoritmos de hash, MD5 y SHA, fueron diseñados usando una unidad de control secuencial (figura 4.10) similar a la empleada en la figura 4.8 pero sin etapa de gestión de clave, siendo en este caso el camino de datos una unidad funcional especial que implementa las operaciones necesarias empleadas en estos algoritmos.

La tabla 4.5 muestra los tiempos de operación en ciclos de reloj para cada uno de los cores y cada una de las etapas que lo involucran: generación de subclaves y cifrado.

4.5. Implementación de los sistemas

A continuación se describen los diferentes sistemas basados en SCPs implementados para las pruebas. También se detallan las diferentes mejoras arquitecturales que pueden ser añadidas a cada SCP, y los diferentes periféricos que acompañan al procesador. Para el diseño del sistema MicroBlaze se han empleado el EDK 6.2. En el caso de LEON2, se ha empleado la versión del core 1.0.24-xst especialmente desarrollada para una correcta síntesis con el sintetizador XST de Xilinx. Sin embargo, en este caso, la herramienta de síntesis ha sido Synplify 7.7.1. En ambos casos se ha empleado la herramienta ISE 6.2 de Xilinx para la generación del *bitstream*.

4.5.1. Sistema SCP MicroBlaze

El sistema MicroBlaze propuesto para estos experimentos se muestra en la figura 4.11. La memoria BRAM se emplea para almacenar un pequeño programa *bootloader* que permite la carga de los diferentes programas en la memoria externa. El controlador de memoria externa es el

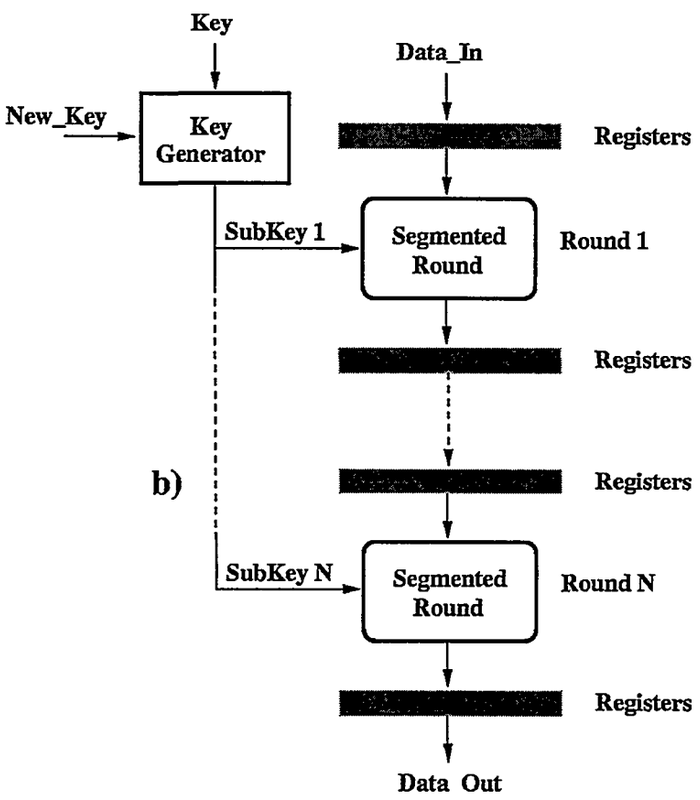


Figura 4.9: Arquitectura del coprocesador completamente desplegada y segmentada

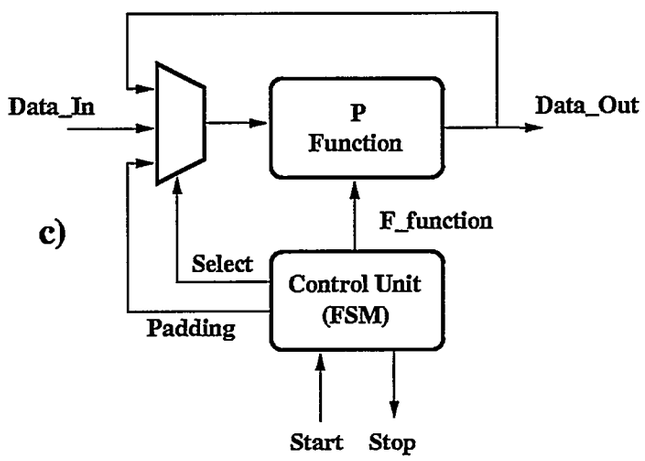


Figura 4.10: Arquitectura del coprocesador basada en FSM sin etapa de gestión de clave

OPB_EMC configurado para un acceso a la memoria SRAM de 32 bits de datos. Al sistema se le ha incorporado una periférico OPB_UARTLite para la comunicación externa con un terminal serie, y un OPB_Timer para la obtención del tiempo empleado en cada prueba.

La arquitectura interna del sistema MicroBlaze puede ser mejorada con las unidades hardware de multiplicación, división y rotación (*barrel shifter*). Además es posible añadir una caché de datos y otra de instrucciones parametrizable en tamaño. La tabla 4.6 resume las distintas arquitecturas que se han implementado para los diferentes experimentos. Ambas caches se han configurado de forma idéntica, esto es, con organización de correspondencia directa (no se puede variar) y con un tamaño de 8192 bytes y 256 entradas. Este tamaño de caché, en particular la caché de instrucciones, se considera suficiente para mejorar adecuadamente la ejecución de todos los algoritmos sin sacrificar demasiado recursos lógicos, al disponer de un tamaño razonable para contener las diferentes funciones de cifrado y descifrado de los algoritmos criptográficos seleccionados. Los resultados de implementación para cada una de las arquitecturas se muestran en la misma tabla, y permiten determinar el gasto de recursos que supone cada una de las mejoras en la arquitectura. La implementación de las caches se realiza en BRAM lo que supone un incremento de 32 a 74 bloques. El sistema MicroBlaze más complejo, MBlaze-E, alcanza una frecuencia de operación máxima sobre el dispositivo Virtex-E de 50 MHz, frecuencia que se ha elegido para realizar todas las pruebas.

Como se puede apreciar en la tabla 4.6, la unidad de multiplicación hardware no se encuentra

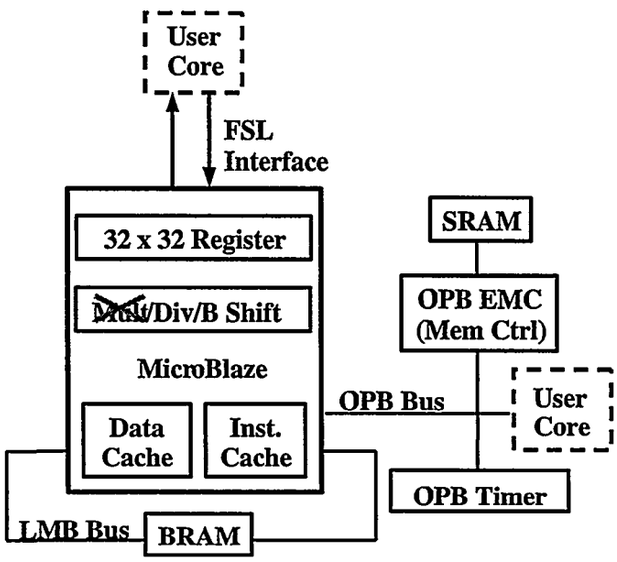


Figura 4.11: Sistema MicroBlaze desarrollado

Arquitectura	Barrel Shifter	Mult	Divisor	Caché (8 kbytes)		FPGA Slices	BRAM
				Datos	Instruc.		
MBlaze-A						1102	32
MBlaze-B	*					1213	32
MBlaze-C				*	*	1216	74
MBlaze-D	*			*	*	1321	74
MBlaze-E	*		*	*	*	1385	74

Tabla 4.6: (a) Arquitecturas MicroBlaze propuestas (b) Recursos para cada Arquitectura MicroBlaze

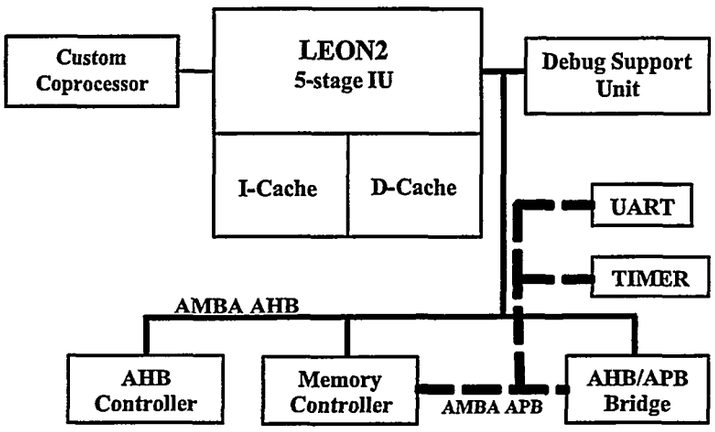


Figura 4.12: Sistema LEON2 desarrollado

disponible porque la FPGA Virtex-E no incluye multiplicadores embebidos. En cualquier caso, la ausencia de multiplicación como recurso hardware solo debería afectar al rendimiento del algoritmo IDEA, ya que es el único que depende en gran medida de esta operación. En el apartado 4.6.1 se analiza la influencia del multiplicador hardware en MicroBlaze para la ejecución del algoritmo IDEA.

4.5.2. Sistema SCP LEON2

El sistema propuesto para el procesador LEON2 es muy similar al sistema MicroBlaze con el fin de poder compararlos en igualdad de condiciones. Este sistema se basa en una selección de las unidades disponibles en la arquitectura inicial de LEON2: el controlador de memoria para acceder a la memoria SRAM externa, los *timers* para medir el tiempo de ejecución de los test y una UART como entrada/salida. Además, se ha incluido la unidad DSU de debug que nos permite cargar los diferentes programas de test desde el programa GRMON sin necesidad de desarrollar un *bootloader*. La figura 4.12 muestra el sistema LEON2 implementado.

La arquitectura del procesador LEON2 también es configurable, siendo posible complementarla con una unidad de multiplicación, una unidad MAC o una unidad de división. También se encuentra disponible una unidad en coma flotante. Sin embargo, se ha optado por una única arquitectura muy similar a la arquitectura MBlaze-D de MicroBlaze, sin multiplicador ni divisor, y con

	Slices	BRAM	Ocupación	Frecuencia
MBlaze-D	1321	74	6%	50 MHz
LEON	3806	40	19%	30 MHz

Tabla 4.7: Sistema LEON2 vs. Sistema MicroBlaze (MBlaze-D)

caché de instrucciones de 1 vía de 8 Kb y la caché de datos idéntica a la de instrucciones. El motivo de esta única configuración se debe a los resultados obtenidos previamente para MicroBlaze, discutidos en la sección 4.6.1.

Los resultados de implementación del sistema LEON2 sobre el dispositivo Virtex-E se muestran en la tabla 4.7, donde se comparan con la arquitectura MBlaze-D. Aunque el sistema LEON2 sobre el dispositivo seleccionado puede operar a una frecuencia de reloj de 30 MHz, se ha optado por emplear 25 MHz como frecuencia de reloj para todas las pruebas, con el objetivo de facilitar futuras comparaciones. De los resultados de implementación mostrados en la tabla 4.7 también se deduce que MicroBlaze está mejor desarrollado para FPGAs de Xilinx, tanto a nivel de área como de velocidad.

4.5.3. Sistema PowerPC

El sistema PowerPC propuesto para la comparación con los sistemas basados en SCPs se ha realizado mediante el EDK 6.2 de Xilinx. Al igual que en el caso de MicroBlaze, presenta una memoria BRAM para el *bootloader*, un OPB_UARTLite para la comunicación con un terminal serie y el OPB_Timer para la medición de los tiempo de ejecución. Del mismo modo, se ha empleado el controlador de memoria PLB_EMC para el acceso a la memoria SDRAM de la plataforma. Esto es posible ya que la memoria SDRAM de la plataforma Alpha-Data está configurada para un bus de datos de 64 bits, y es por ello que se puede emplear el controlador PLB_EMC en vez del OPB_EMC, lo que supone una clara ventaja con respecto a los sistemas SCPs de 32 bits de datos. El PowerPC no presenta ninguna característica de configurabilidad. Al contrario, dispone de un conjunto de unidades operacionales en hardware como multiplicador, divisor, etc. además de la caché de datos y la caché de instrucciones. En el caso de las caches es necesario habilitarlas para que el procesador haga uso de ellas, y han sido configuradas para cachear la región de memoria que comprende la memoria SDRAM externa.

4.6. Rendimiento de los sistemas

4.6.1. Sistema SCP MicroBlaze

El primer conjunto de experimentos incluye la ejecución de las implementaciones software de los diferentes algoritmos sobre las arquitecturas MicroBlaze mostradas en la tabla 4.6. Posteriormente, el segundo conjunto de experimentos consiste en emplear los diferentes cores hardware conectados a los buses OPB y FSL disponibles en MicroBlaze.

Arquitecturas configurables de MicroBlaze

Los resultados del primer experimento para las distintas arquitecturas MicroBlaze propuestas se muestra en la tabla 4.8. Estos resultados permiten determinar, como cabía suponer tras analizar las operaciones que intervienen en cada uno de los algoritmos (tabla 4.3), que la arquitectura MBlaze-E que incluye al divisor hardware no aporta ninguna ventaja sobre la arquitectura MBlaze-D. Por el contrario, la unidad *barrel shifter* que aparece en la arquitectura MBlaze-B se revela como el componente fundamental para la mejora del rendimiento en todos los algoritmos, con excepción del algoritmo IDEA, donde no se emplea este tipo de operación. El uso de memoria caché favorece sobretodo a los algoritmos IDEA, Blowfish y RC4, ya que para el resto la mejora de rendimiento es inferior que al incluir la unidad *barrel shifter*. El algoritmo MD5 presenta un buen rendimiento tanto empleando la unidad *barrel shifter* como caché, de ahí la enorme mejora que presenta al combinar ambos. El algoritmo SHA-2 no se ejecuta correctamente si no existe la unidad *barrel shifter*, lo que puede ser debido a que las operaciones de rotación que lo implementan no se realizan correctamente en MicroBlaze, posiblemente por problemas con el compilador.

La figura 4.13 nos permite comprobar adecuadamente estas conclusiones, al mostrar la mejora que supone cada uno de los componentes internos de MicroBlaze, o lo que es los mismo, comparar las diferentes arquitecturas propuestas respecto de la arquitectura MBlaze-A, para cada uno de los algoritmos. La gráfica permite determinar la importancia de la unidad *barrel shifter* y las caches, así como la combinación de ambas, para cada algoritmo. En esta figura no aparecen los resultados del algoritmo SHA-2 debido a que no se tiene el valor de referencia de la arquitectura MBlaze-A. De la gráfica se puede concluir que la combinación de las unidades *barrel shifter* y caché permite obtener mejoras considerables, que en algunos casos son muy destacadas como en el caso del algoritmo DES o MD5. También se puede apreciar que ciertos algoritmos no obtienen ningún beneficio de la unidad *barrel shifter* como el algoritmo IDEA o el algoritmo RC4. La caché, sin embargo, es positiva para todos ellos.

Como resultado general, podemos concluir que la combinación de caché y *barrel shifter*, denominada arquitectura MBlaze-D, presenta el mejor resultado, como se deduce del análisis de las

Algoritmo	MBlaze-A		MBlaze-B		MBlaze-C		MBlaze-D&E	
	# de ciclos	Tiempo Ejecución (s)	# de ciclos	SpeedUp	# de ciclos	SpeedUp	# de ciclos	SpeedUp
IDEA	48034	503,673	34252	1,40	5958	8,06	4209	11,41
DES	83676	877,410	12477	6,71	23036	3,63	1787	46,84
3DES	230493	2416,894	35749	6,45	97012	2,38	19085	12,02
AES	93539	490,416	14707	6,36	38072	2,46	7076	13,22
BLOWFISH	23695	248,456	8928	2,65	3361	7,05	1648	14,38
RC4	19447	101,960	18957	1,03	4387	4,43	4663	4,17
MD5	141647	185,660	30378	4,66	50233	2,82	5502	25,75
SHA-I	356611	467,417	64917	5,49	132375	2,69	21978	16,23
SHA-2	-	-	98534	-	-	-	86882	-

Tabla 4.8: Comparación de cada una de las arquitecturas MicroBlaze propuestas en relación a los resultados de MBlaze-A (Speedup = Tejec_MBlaze-A / Tejec)

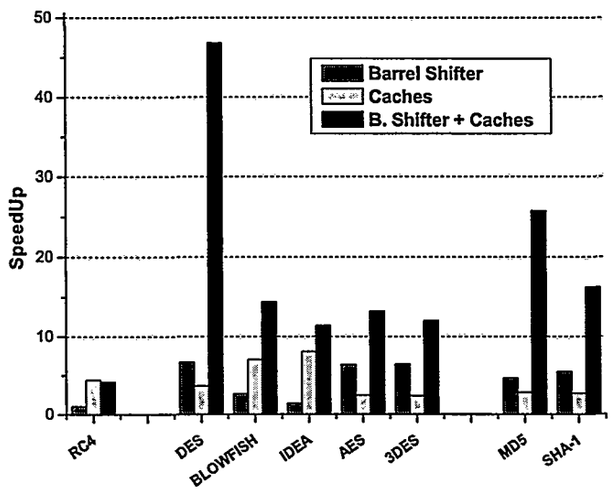


Figura 4.13: Mejora obtenida por cada una de las arquitecturas MicroBlaze propuestas en relación a la arquitectura MBlaze-A

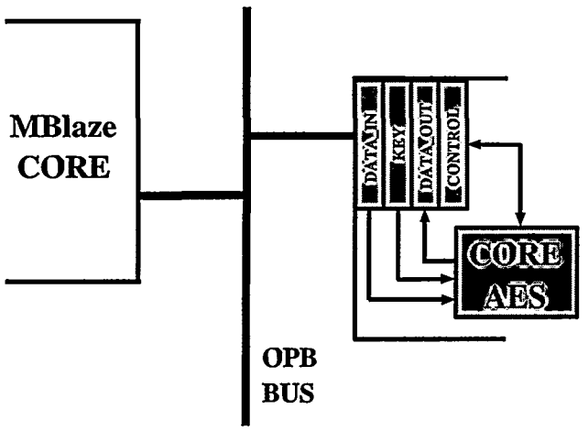


Figura 4.14: Interfaz de conexión de los cores OPB

diferentes arquitecturas, MBlaze-B y MBlaze-C, por separado.

Arquitectura mejorada mediante un módulo específico de cifrado

Con el objetivo de acelerar el rendimiento obtenido en el primer conjunto de experimentos, se procede a ampliar la funcionalidad de MicroBlaze conectado los cores específicos de cifrado utilizando los buses OPB y FSL. En el caso de la interfaz OPB, se han mapeado en memoria los diferentes registros necesarios para almacenar los datos de entrada, los datos de salida, la clave (o subclaves en el caso de IDEA, aunque estas se han almacenado en BRAM) y un registro de control para indicar el tipo de operación (cifrado/descifrado), iniciar el cifrado y detectar cuando éste ha terminado. De igual modo, la interfaz FSL con los cores consta de 5 links de 32 bits: uno de entrada para el envío de la clave, otros dos de entrada para el envío de los datos y los dos últimos de salida para leer el resultado. Para determinar el modo de operación se ha empleado la señal de control de los links de entrada de datos. Finalmente, en el caso de los algoritmos de hash, el canal de envío de la clave no se emplea por razones obvias. Las figuras 4.14 y 4.15 muestran las diferentes interfaces de conexión para los cores en MicroBlaze.

La tabla 4.9 muestra los recursos empleados en la implementación de los cores para cada una de las interfaces del procesador MicroBlaze. Los resultados permiten determinar el coste y complejidad de cada una de ellas. La adaptación de un core a la interfaz FSL es más simple que la

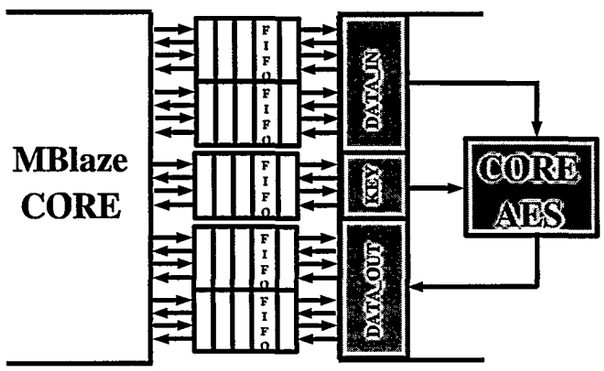


Figura 4.15: Interfaz de conexión de los cores FSL

interfaz OPB, y *a priori* supone un menor uso de recursos. En cuanto a las BRAM, ninguna de las interfaces incrementa su uso.

Al tiempo que se adaptan los diferentes cores a las diferentes interfaces es igualmente necesario adaptar los algoritmos en C, sustituyendo las funciones software de cada algoritmo por sus equivalentes de envío de clave, envío de datos y recogida de resultados. El código resultante es bastante reducido y sencillo, al estar toda la funcionalidad del algoritmo implementada en hardware. De igual modo que con los algoritmos software, los diferentes cores se han chequeado mediante los test de verificación para comprobar que las implementaciones hardware cumplen con las especificaciones del algoritmo. Es importante destacar que todos ellos se han ejecutado sobre las distintas arquitecturas del procesador MicroBlaze propuestas para el primer conjunto de experimentos, excepto la arquitectura MBlaze-E (con divisor hardware) que tampoco en este caso tiene nada que aportar.

Módulo periférico específico de cifrado (OPB) El bus OPB es un bus específico para la conexión de periféricos, por lo tanto, se espera una pequeña mejora de rendimiento al no estar especializado para la conexión de unidades hardware. La tabla 4.10 recoge los resultados del test sobre cada una de las arquitecturas MicroBlaze propuestas empleando los cores OPB. Estos resultados permiten determinar que el recurso fundamental de la arquitectura MicroBlaze es la caché, ya que lo fundamental es acelerar el software de manejo de los cores. Al mismo tiempo, se puede comprobar que los rendimientos para todos los algoritmos simétricos son similares, a pesar de las diferencias en los ciclos de cifrado, probablemente porque el rendimiento se ve afectado por el

Algoritmo	OPB Slices	FSL Slices	BRAM
IDEA	1979	1744	12
DES	4121	3920	-
3DES	12176	11786	-
AES	5143	4997	-
BLOWFISH	2877	2496	10
RC4	942	719	2
MD5	2449	2412	-
SHA-1	2160	2134	-
SHA-256	2608	2572	-

Tabla 4.9: Resultados de implementación de los cores de cifrado para cada una de las interfaces de conexión con MicroBlaze: bus OPB e interfaces FSL

hecho de compartir el mismo bus para el acceso a memoria, y los tamaños de bloque son similares para todos los algoritmos. La excepción es la implementación hardware del algoritmo AES, que emplea un bloque de datos mayor y un menor número de ciclos de operación para realizar un cifrado, permitiéndole obtener un rendimiento superior. En cuanto a los algoritmos de hash, las distintas implementaciones hardware son equivalentes y emplean un número de ciclos de operación muy similares, de ahí que los rendimientos obtenidos están muy próximos. La tabla 4.10 también permite conocer como afectan cada uno de los componentes hardware disponibles en MicroBlaze, *barrel shifter* y caché aunque también estaría la multiplicación y la división hardware, al rendimiento de los cores OPB. Como se podía esperar, el uso de caches permite acelerar el código de manejo de los cores, reduciendo el tiempo de acceso compartido al bus. Con respecto a la unidad *barrel shifter*, no aporta una mejora considerable, al reducirse el código de la aplicación únicamente a escribir y leer direcciones de memoria. Sin embargo, aporta una ligera mejora en el rendimiento al combinarse con la caché.

Modulo coprocesador específico de cifrado (FSL) Al contrario que el bus OPB, el bus FSL permite una comunicación más directa con el procesador al estar conectado directamente a los registros internos, lo que supone una mayor integración en la propia unidad de ejecución del procesador, y una mayor velocidad en el intercambio de datos. La tabla 4.11 muestra los resultados obtenidos para esta interfaz únicamente para la arquitecturas MBlaze-A y MBlaze-D. Se ha omitido el estudio de las unidades *barrel shifter* y caché por separado porque los resultados obtenidos para el bus OPB, tabla 4.10, pueden ser fácilmente asumidos para el bus FSL. De estos resultados se deduce que el uso de la caché permite diferenciar el rendimiento de cada uno de los algoritmos en función de su implementación hardware, ya que si no se emplea, los rendimientos son muy

Algoritmo	MBlaze-A			MBlaze-B		MBlaze-C		MBlaze-D	
	# de ciclos	Tiempo Ejecución (s)	SpeedUp	# de ciclos	SpeedUp	# de ciclos	SpeedUp	# de ciclos	SpeedUp
IDEA	2747	28,801	1,53	2619	1,61	763	5,52	749	5,62
DES	2634	27,619	0,68	2484	0,72	630	2,84	616	2,90
3DES	2759	28,930	6,92	2609	7,32	725	26,32	710	26,86
AES	3052	16,001	2,32	2752	2,57	646	10,95	619	11,44
BLOWFISH	3159	33,125	0,52	2529	0,65	674	2,44	597	2,76
RC4	3617	18,963	1,29	3317	1,41	1238	3,77	1211	3,85
MD5	7784	10,202	0,71	6503	0,85	1724	3,21	1609	3,42
SHA-1	7605	9,968	2,89	6325	3,47	1550	14,18	1446	15,20
SHA-2	7866	10,310	11,05	6586	13,19	1527	56,91	1462	59,44

Tabla 4.10: Comparación de cada una de las arquitecturas MicroBlaze propuestas con coprocesador OPB en relación a los resultados de MBlaze-D (Speedup = Tejec_MBlaze-D / Tejec)

similares para cada uno de los grupos de algoritmos (clave privada y hash). Cabe destacar una mejora considerable de rendimiento en aquellos algoritmos que presentan un número reducido de ciclos de cifrado.

Por último, la figura 4.16 trata de resumir como afecta al rendimiento de los cores, cada una de las mejoras disponibles en la arquitectura MicroBlaze.

Comparación de los resultados sobre MicroBlaze

La figura 4.17 muestra los mejores resultados obtenidos en los distintos experimentos para el sistema basado en MicroBlaze, empleando en todos los casos la arquitectura MBlaze-D. Los algoritmos se han ordenado de menor a mayor rendimiento final obtenido, y agrupados por tipo (clave privada y hash).

Para realizar una comparación más exhaustiva, la figura 4.18 presenta una visión diferente de los resultados mostrados en la figura 4.17. Se trata de la mejora obtenida para cada una de las soluciones basadas en cores hardware, incluyendo la comparativa FSL frente a OPB. Los resultados de esta figura permiten determinar que la mejora empleando cores hardware presenta un comportamiento casi idéntico en función del algoritmo, destacando sobremanera las mejoras obtenidas para los algoritmos 3DES, AES y SHA-2 respectivamente. También comparando las interfaces FSL y OPB se aprecian interesantes mejoras, que destacan a favor de FSL en los algoritmos DES y Blowfish, siendo ambos algoritmos de operaciones muy similares. En el caso de los algoritmos hash, la diferencia entre FSL y OPB es menor probablemente a que se pierde más tiempo accediendo a memoria para obtener los 512 bytes del bloque, que en la propia operación del algoritmo.

Algoritmo	MBlaze-A			MBlaze-D	
	# de ciclos	Tiempo Ejecución (s)	SpeedUp	# de ciclos	SpeedUp
IDEA	576	6,041	7,31	260	16,19
DES	935	9,804	1,91	135	13,24
3DES	1033	10,831	18,48	233	81,92
AES	1190	6,239	5,95	202	35,03
BLOWFISH	935	9,804	1,76	123	13,39
RC4	1686	8,839	2,77	815	5,72
MD5	4150	5,440	1,33	913	6,02
SHA-1	4099	5,373	5,36	754	29,16
SHA-2	4263	5,588	20,38	740	117,40

Tabla 4.11: Comparación de cada una de las arquitecturas MicroBlaze propuestas con coprocesador FSL en relación a los resultados de MBlaze-D (Speedup = Tejec_MBlaze-D / Tejec)

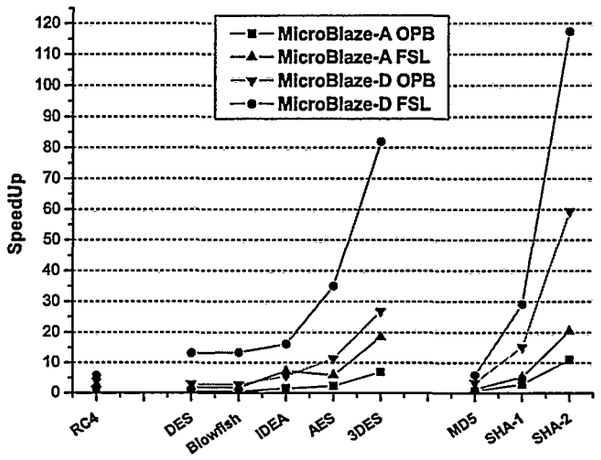


Figura 4.16: Mejora obtenida al emplear las diferentes arquitecturas MicroBlaze con coprocesador en relación a MBlaze-A sin coprocesador

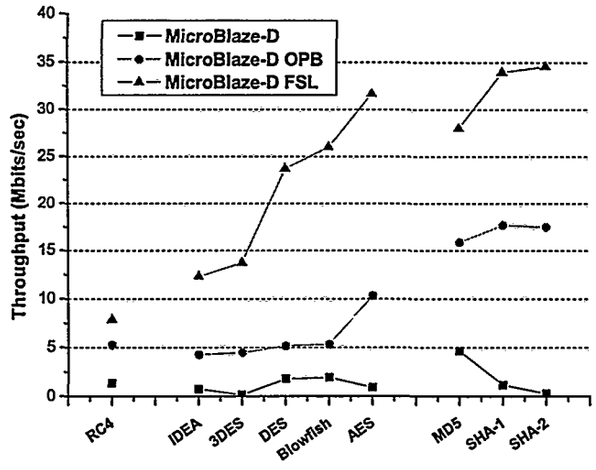


Figura 4.17: Mejores resultados obtenidos en MicroBlaze

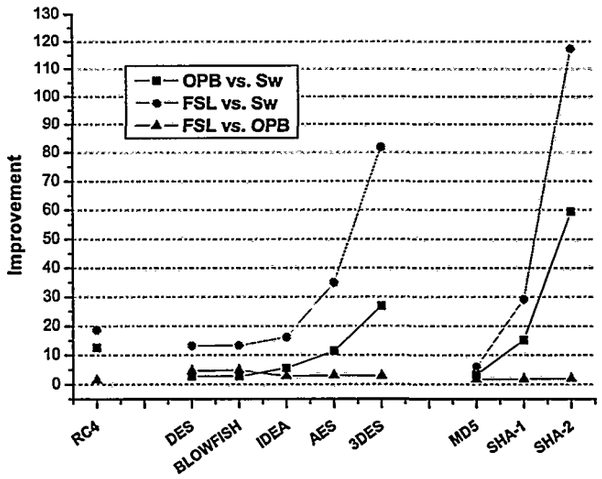


Figura 4.18: Mejoras obtenidas al emplear los coprocesadores hardware en MicroBlaze

Sistema MBlaze-D	Ejecución de IDEA # de ciclos	SpeedUp
Software	4209	1
MULT FSL	1320	3,19
IDEA OPB	749	5,62
IDEA FSL	260	16,19

Tabla 4.12: Resultados del algoritmo IDEA en MicroBlaze empleando diferentes soluciones para el multiplicador en Virtex-E

En resumen, como se presuponía, el bus FSL es el más indicado para la conexión de cores hardware cuando se quiere acelerar el rendimiento de las aplicaciones en MicroBlaze. Además, el uso de estos cores conectados mediante FSL permite a MicroBlaze obtener unos rendimientos razonables superándose perfectamente los 10 MBits/s, con excepción del algoritmo RC4.

IDEA en MicroBlaze

El estudio de los diferentes algoritmos sobre una dispositivo reconfigurable de la familia Virtex-E, no ha permitido establecer el rendimiento del algoritmo IDEA cuando el dispositivo reconfigurable dispone de multiplicadores embebidos. En este apartado se aborda el caso particular del algoritmo IDEA en sistemas MicroBlaze sobre dispositivos con multiplicadores embebidos.

Multiplicador como coprocesador Como primera posibilidad para disponer de unidad de multiplicación hardware en sistemas MicroBlaze sobre dispositivos reconfigurables que carecen de multiplicadores embebidos, se puede implementar un multiplicador conectado al bus FSL. Como se ha demostrado, se trata de la interfaz más indicada para conectar de una manera óptima hardware específico. De este modo es posible mejorar el rendimiento del algoritmo IDEA reemplazando la multiplicación software, por una multiplicación hardware basada en un core FSL. Los resultados de la tabla 4.12 muestran que la unidad de multiplicación es necesaria para mejorar el rendimiento del algoritmo IDEA, y los resultados mediante un multiplicador FSL parecen una buena alternativa a la falta de multiplicadores embebidos.

MicroBlaze con multiplicadores embebidos Al diseñar un sistema MicroBlaze sobre dispositivos Virtex-II y Virtex-II Pro, el sistema emplea 3 multiplicadores embebidos de forma automática para disponer de una unidad de multiplicación hardware de 32 bits. Opcionalmente, mediante un parámetro del compilador es posible hacer uso de las instrucciones que activan este multiplicador. Para estas pruebas hemos empleando la plataforma ADM-XRCIIPro-Lite de Alpha Data basada

Sistema MicroBlaze-D	Rendimiento (Mbit/s)			
	MBlaze-A	MBlaze-B	MBlaze-C	MBlaze-D
V2P Mult. Soft	0,065	0,092	0,528	0,748
V2P Mult Hard	0,141	0,367	1,098	2,813
Virtex-E	0,067	0,093	0,537	0,760

Tabla 4.13: Rendimiento del algoritmo IDEA sobre un dispositivo Virtex-II Pro (XC2V20P)

Sistema MicroBlaze-D	Rendimiento (MBits/seg)
Mult Hard (Virtex-II Pro)	2,81
FSL MULT (Virtex-E)	2,42
IDEA OPB (Virtex-E)	4,27
IDEA FSL (Virtex-E)	12,35

Tabla 4.14: Rendimiento del algoritmo IDEA en MicroBlaze

en el dispositivo XC2V20P. La tabla 4.13 muestra los resultados obtenidos para las distintas arquitecturas MicroBlaze sobre el dispositivo Virtex-II Pro resolviendo la multiplicación por software y por hardware. Estos resultados se comparan con los resultados obtenidos para el dispositivo Virtex-E. En ambas plataformas la frecuencia de operación es la misma, 50 MHz, y por ello los resultados sin multiplicación hardware son muy similares. Las pequeñas variaciones se puede deber a la dependencia que la multiplicación por software tiene de los operandos, dado que el contenido de las memorias externas no tiene porque ser idéntico en ambas plataformas (se generan aleatoriamente). En cuanto al uso de multiplicación hardware, el rendimiento es 2 veces superior para las arquitecturas sin caché y 4 veces superior para las arquitecturas con caché.

Resultados El resultado final obtenido para el algoritmo IDEA empleando multiplicadores embebidos permite mejorar su rendimiento casi 4 veces con respecto al sistema sin multiplicadores embebidos. Naturalmente se sabe que la operación de multiplicar es fundamental para el algoritmo IDEA, y por tanto la existencia de esta unidad es recomendable siempre que no se desee realizar un coprocesador para el algoritmo. Además, es importante comprobar que con la multiplicación hardware disponible, IDEA presenta el mejor rendimiento de los diferentes algoritmos simétricos analizados, y el segundo mejor por detrás del algoritmo MD5. Por último, la tabla 4.14 muestra un resumen de los rendimientos obtenidos para cada solución propuesta. Destaca la proximidad entre los rendimientos empleando multiplicadores embebidos y la solución basada en una unidad de multiplicar conectada al bus FSL, demostrando que esta interfaz es muy adecuada para ampliar el conjunto de instrucciones del procesador MicroBlaze.

Algoritmo	LEON2		LEON2 CP	
	# de ciclos	Tiempo Ejecución (s)	# de ciclos	SpeedUp
IDEA	2153	45,150	211	10,19
DES	974	20,418	85	11,39
3DES	5062	106,150	183	27,61
AES	3413	35,785	86	39,89
BLOWFISH	697	14,610	51	13,65
RC4	7551	79,173	698	10,82
MD5	2889	7,574	610	4,74
SHA-1	4595	12,045	416	11,04
SHA-2	23538	61,703	821	57,34

Tabla 4.15: Resultados sobre el sistema LEON2 propuesto con/sin coprocesador (Speedup = Tejec_LEON2 / Tejec_LEON2 CP)

4.6.2. Sistema SCP LEON2

A partir de los resultados obtenidos para el procesador MicroBlaze, se han simplificado los diferentes experimentos para el procesador LEON2:

- Para el primer conjunto de experimentos se plantea directamente el uso del core LEON2 con todas las posibles mejoras de que dispone, manteniendo las mismas capacidades que el procesador MicroBlaze como se describió en el apartado 4.5.2.
- Para el segundo conjunto de experimentos se emplea directamente la interfaz genérica de coprocesador de que dispone el LEON2, y se omiten los experimentos sobre el bus APB para la conexión de periféricos, dado que el rendimiento que se espera del bus APB es inferior con respecto a la interfaz para coprocesador [178].

Arquitectura LEON2

Al ejecutar el conjunto de programas del primer bloque de experimentos sobre el sistema LEON2 propuesto, se obtienen los resultados mostrados en la primera columna de la tabla 4.15. Los rendimientos obtenidos son bastante bajos, y en el mismo rango que MicroBlaze al emplear la arquitectura MBlaze-D. Sin embargo, LEON2 se ejecuta a la mitad de frecuencia que MicroBlaze.

Arquitectura basada en coprocesador específico

Una vez descartado el bus AMBA APB, muy similar al bus OPB, tras las pruebas sobre MicroBlaze, solo queda disponible el uso de la interfaz genérica para coprocesador. Cada uno de los

Algoritmo	LEON2 CP		
	Slices	BRAM	% Uso
IDEA	6048	48	32%
DES	8146	48	42%
3DES	16008	48	83%
AES	8278	48	43%
BLOWFISH	8762	48	46%
RC4	4977	50	26%
MD5	6628	48	35%
SHA-1	6250	48	33%
SHA-256	6672	48	35%

Tabla 4.16: Resultados de implementación del sistema LEON2 con los cores de cifrado como coprocesador

cores hardware se ha adaptado a dicha interfaz para complementar al procesador LEON2. Los resultados sobre LEON2 empleando los diferentes coprocesadores hardware se muestran en la tabla 4.16. Del mismo modo, el rendimiento de los coprocesadores empleando el segundo conjunto de experimentos para cada uno de los algoritmos se muestran en la segunda columna de la tabla 4.15. Los resultados son claramente superiores a la ejecución software, y en todos los casos se consigue una mejora considerable para todos los algoritmos, justificada en gran parte por tratarse de un interfaz específica para coprocesador.

Comparación de los resultados sobre LEON2

La tabla 4.15 muestra la mejora obtenida para cada algoritmo al usar los cores hardware como coprocesador. Destaca principalmente el aumento en el rendimiento de los algoritmos 3DES, AES y SHA-2, en parte debido al bajo rendimiento mostrado en su implementación software. Por el contrario, el algoritmo MD5, que presentaba un buen rendimiento al ejecutarlo completamente en software, presenta la mejora más reducida. En el resto de algoritmos el rendimiento de la arquitectura LEON2 era razonable y las mejoras también son importantes.

El uso de un coprocesador específico permite a LEON2 obtener unos resultados adecuados para las aplicaciones de cifrado en sistemas embebidos, aunque para tres de los algoritmos se ha obtenido un rendimiento por debajo de los 10 MBits/s a una frecuencia de operación de 25 MHz.

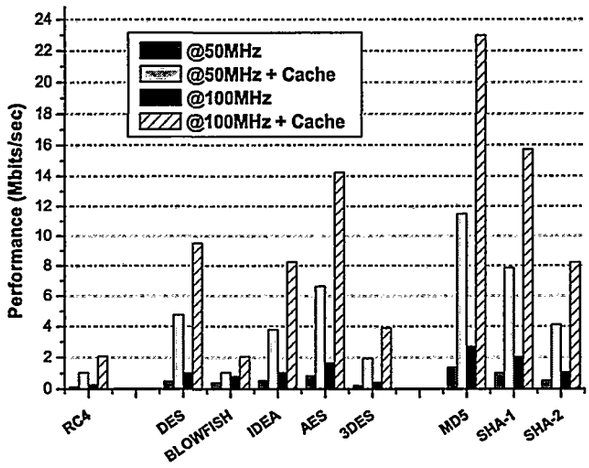


Figura 4.19: Comparativa del rendimiento de los diferentes algoritmos sobre el sistema PowerPC

4.6.3. Sistema PowerPC

Las pruebas realizadas sobre PowerPC solo cubren el primer conjunto de experimentos con el objetivo de utilizar el rendimiento obtenido para compararlo frente a los SCPs. Los resultados obtenidos de la ejecución de los diferentes algoritmos en C se muestran en la figura 4.19. Se han establecido cuatro configuraciones diferentes del mismo sistema variando la frecuencia de operación y el uso de caches, lo que nos permite elaborar un pequeño estudio sobre las características del procesador PowerPC. De los resultados obtenidos se puede determinar que al duplicar la frecuencia, se produce un incremento del doble en el rendimiento. Por tanto, es fácilmente estimable el rendimiento que el PowerPC puede ofrecer al modificar la frecuencia de operación. Del mismo modo, el uso de las caches es fundamental, y el rendimiento sufre una mejora considerable que en algunos casos permite obtener rendimientos superiores a 10 Mbits/s operando a 100 MHz. Además, la mejora que supone el uso de caché es independiente de la frecuencia de reloj, aunque presenta diferentes comportamiento dependiendo del algoritmo. Resulta interesante la poca efectividad de la caché en la ejecución del algoritmo Blowfish, al contrario que para el resto de los algoritmos, donde la mejora en el rendimiento es casi cuatro veces mayor. Sin embargo, Blowfish comparte con RC4 el peor rendimiento de todos los algoritmos.

La tabla 4.17 muestra los resultados obtenidos para el PowerPC, siendo de este modo más fácil

Algoritmo	PPC @50MHz		PPC @100MHz		
	# de ciclos	Tiempo Ejecución (s)	# de ciclos	Tiempo Ejecución (s)	SpeedUp
IDEA	843	8,842	775	4,065	2,18
DES	674	7,070	673	3,529	2,00
3DES	1647	17,267	1646	8,628	2,00
AES	967	5,070	900	2,358	2,15
BLOWFISH	3110	32,610	3110	16,305	2,00
RC4	6169	32,342	6168	16,168	2,00
MD5	2230	2,923	2228	1,460	2,00
SHA-1	3258	4,270	3256	2,134	2,00
SHA-2	6230	8,166	6229	4,082	2,00

Tabla 4.17: Comparación de los diferentes sistemas PowerPC propuestos (Speedup = Tejec_50MHz / Tejec_100MHz)

interpretar el rendimiento para cada una de las configuraciones propuestas (siempre empleando las cachés). Estos resultados se justifican debido a que el procesador PowerPC incorpora un conjunto de unidades operacionales, todas ellas implementadas por hardware, que le permiten acelerar el código software de los algoritmos, al estilo de las mejoras en la arquitectura de MicroBlaze.

Finalmente, aunque el segundo conjunto de experimentos se podría lleva a cabo sobre este sistema, la única alternativa que ofrece el PowerPC de la familia Virtex-II Pro es conectar los cores hardware al bus OPB o PLB, al no incluir ninguna interfaz específica para coprocesador. Esto supondría obtener unos rendimientos superiores, al igual que se ha obtenido para MicroBlaze, aunque quizás con una mejora inferior. En cualquier caso, el objetivo es comparar los SCPs frente los procesador hard-core tradicionales, en los cuales no es posible añadir este tipo de hardware específico.

4.7. Comparando los diferentes sistemas

A continuación se comparan los resultados obtenidos para los dos conjuntos de experimentos propuestos en los sistemas SCP, PowerPC y PC. Se propone la comparativa "PowerPC a 50 MHz frente a SCPs sin core", y "PowerPC a 100 MHz frente a SCPs con core". La comparación con el PC es muy diferente ya que los rendimientos, como se supone, no son comparables. Es importante destacar que en ningún caso el objetivo de esta comparativa es determinar que SCP es mejor, sino determinar a priori las posibilidades que ofrecen frente a los hard-cores (PowerPC), y las ventajas de usarlos en sistemas embebidos, como puede ser la flexibilidad o adaptación a la aplicación a

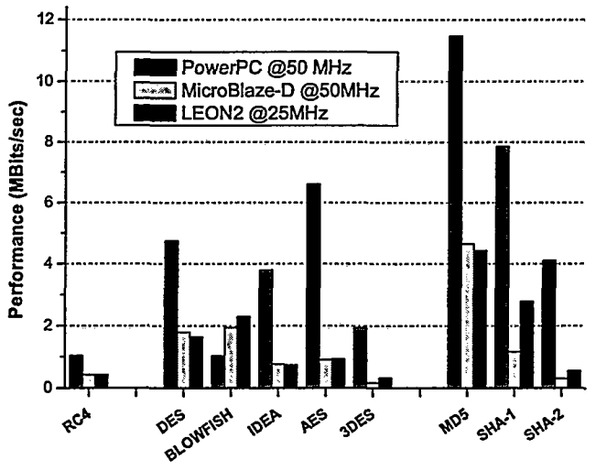


Figura 4.20: Comparación de los SCPs sin cores frente al PowerPC

ejecutar.

4.7.1. SCPs vs. PowerPC

En la figura 4.20 se muestra la comparativa de rendimiento "PowerPC a 50 MHz frente a SCPs sin core". En ellas se evalúa el potencial de cada uno de los SCPs por sí mismos, incluyendo las posibles mejoras disponibles en su arquitectura. La figura permite determinar que el rendimiento de ambos SCPs es bastante bajo comparado con el PowerPC, aunque es importante tener en cuenta que la caché del PowerPC es superior, y que éste incluye las operaciones de división y multiplicación por hardware, entre otras mejoras. Destaca, en el caso del algoritmo IDEA, la mejora en el rendimiento que supone que el PowerPC disponga de multiplicador hardware. Por el contrario, la excepción la cumple el algoritmo Blowfish, que en PowerPC presenta un rendimiento realmente bajo, permitiendo a ambos SCPs superar al hard-core con diferencia. Otro detalle interesante, que se analiza posteriormente, es el buen rendimiento mostrado por el procesador LEON2 frente a MicroBlaze en algunos algoritmos trabajando a la mitad de frecuencia de reloj.

La gráfica 4.21 muestra la comparativa de rendimiento "PowerPC a 100 MHz frente a SCPs con core". Sin lugar a dudas, en este caso los SCPs son muy superiores gracias a que todo el proceso de cifrado se resuelve por hardware, y solo es necesario ejecutar unas decenas de líneas

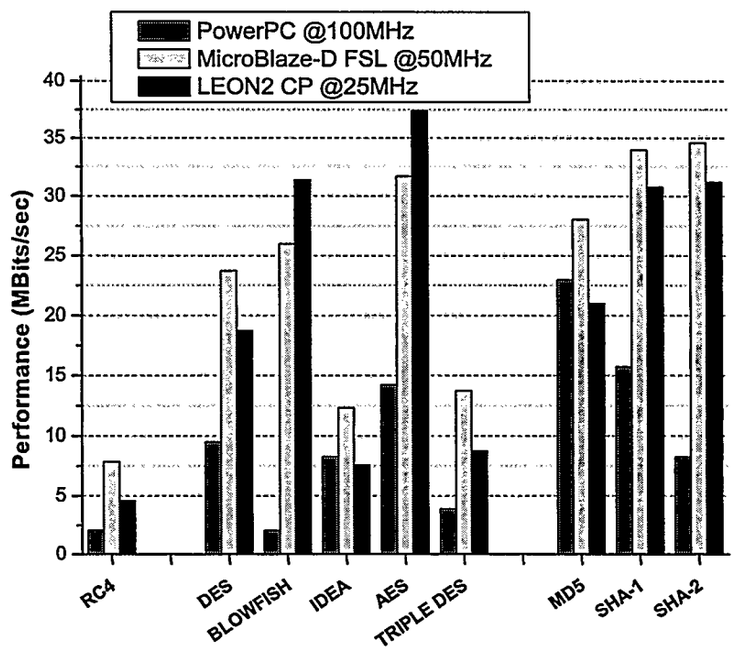


Figura 4.21: Comparación de los SCPs con cores frente al PowerPC

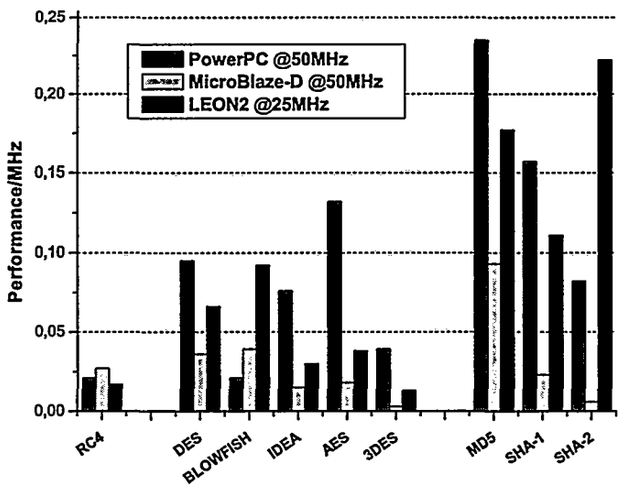


Figura 4.22: Comparación de los SCPs sin cores frente al PowerPC [rendimiento/MHz]

de código. En la mayoría de los algoritmos el PowerPC presenta un rendimiento muy inferior, excepto en el algoritmo MD5 donde el procesador PowerPC se muestra superior al procesador LEON2 CP, y muy próximo a MicroBlaze FSL. También en el algoritmo IDEA es superior por muy poco a LEON2 CP, gracias a la multiplicación hardware de que dispone.

Otra forma de medir el potencial de los procesadores es realizar un comparación en la que se mide el rendimiento por MHz, de modo que puedan compararse procesadores que trabajan a diferencias frecuencias de reloj. Si bien esta comparativa no es realista al comparar sistemas que emplean diferentes memorias (SRAM en SCPs, SDRAM en el caso del PowerPC), puede dar una idea de la efectividad de cada procesador. Las figuras 4.22 y 4.23 se corresponden con las comparaciones anteriores pero en su versión rendimiento por MHz. En la primera comparativa (figura 4.22), el PowerPC se muestra como el procesador más potente, seguido por el procesador LEON2, que se ha mostrado superior a MicroBlaze. Esta superioridad de LEON2 se podía deducir de los resultados obtenidos por este procesador, al ser en algunos casos similares a los obtenidos por MicroBlaze, pero operando a la mitad de frecuencia. En la segunda comparativa (figura 4.23), los SCPs con cores hardware, son muy superiores al PowerPC, mientras que LEON2 se sigue manteniendo como el más eficiente de los SCPs.

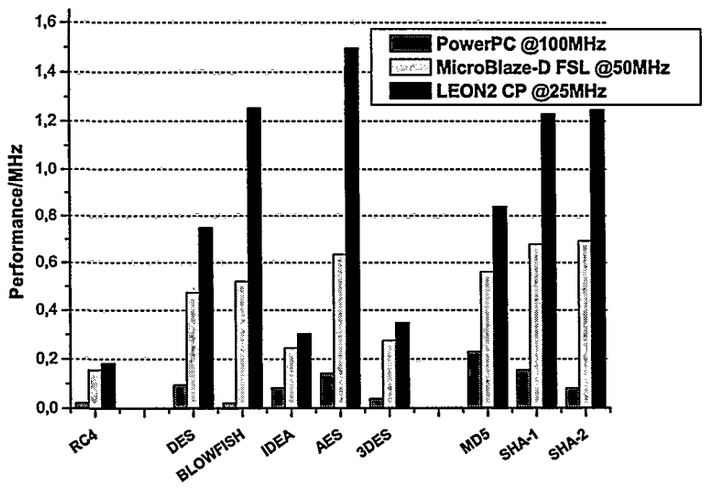


Figura 4.23: Comparación de los SCPs con cores frente al PowerPC [rendimiento/MHz]

4.7.2. Procesadores en FPGA vs. PC

En esta comparativa se muestran los mejores rendimientos obtenidos al realizar el primer conjunto de experimentos sobre los sistemas basados en FPGA, frente a la ejecución en PC. El sistema operativo que se ejecuta en el PC es Linux 2.4.18 y como era de esperar, a pesar de la "carga" que supone, los resultados obtenidos, y que se comparan en la figura 4.24, son claramente superiores tanto al hard-core PowerPC como a los SCPs con cores hardware. Lo que permite concluir que todos los sistemas implementados se encuentran muy alejados del rendimiento obtenido en un PC moderno. Si aceptamos la comparativa del rendimiento por MHz, entonces se obtienen resultados significativos como muestra la figura 4.25. En ellos el PC se presenta como el procesador menos efectivo por MHz. Y como viene siendo habitual, LEON2 sigue mostrándose como el procesador más efectivo.

Finalmente y como curiosidad, el rendimiento obtenido por el algoritmo RC4 en el PC es muy inferior al obtenido en el resto de los algoritmos, comportamiento que se ha mantenido prácticamente igual en todos los procesadores.

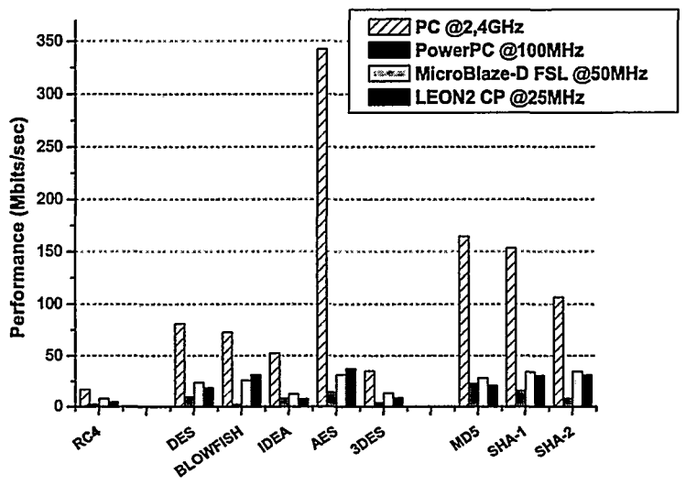


Figura 4.24: Comparación de los diferentes sistemas sobre FPGA en relación al PC

4.7.3. MicroBlaze vs. LEON2

En las comparaciones anteriores ya se ha puesto de manifiesto la superioridad de MicroBlaze sobre LEON2 en la mayoría de los algoritmos evaluados. En particular esta superioridad se debe en parte a que es posible ejecutar MicroBlaze a una frecuencia muy superior en FPGAs de Xilinx. Otra comparativa interesante es conocer que interfaz es más efectiva para mejorar el rendimiento del procesador, FSL de MicroBlaze o la Custom Coprocessor Interface de LEON2. La figura 4.26 muestra la mejora que supone emplear un coprocesadores específico para cada algoritmo con respecto a la implementación software.

Aunque las comparativas anteriores se basan siempre en el rendimiento, lo cual permite evaluar uno de los parámetros más determinantes a la hora de emplear SCPs en sistema embebidos, también se presenta como una característica fundamental en los diseños sobre FPGA, el número de recursos necesarios dentro del dispositivo reconfigurable para implementar el sistema SCP, como contraposición al rendimiento, a fin de poder determinar el coste en lógica de obtener el rendimiento correspondiente. Si bien es cierto que los dispositivos reconfigurables cada vez disponen de más lógica, también es fundamental que el sistema SCP debe dejar libre una mayor cantidad de lógica dentro del dispositivo, a fin de disponer de más espacio para por ejemplo, implementar los diferentes cores hardware que permitan acelerar la aplicación. La tabla 4.18 muestra los resul-

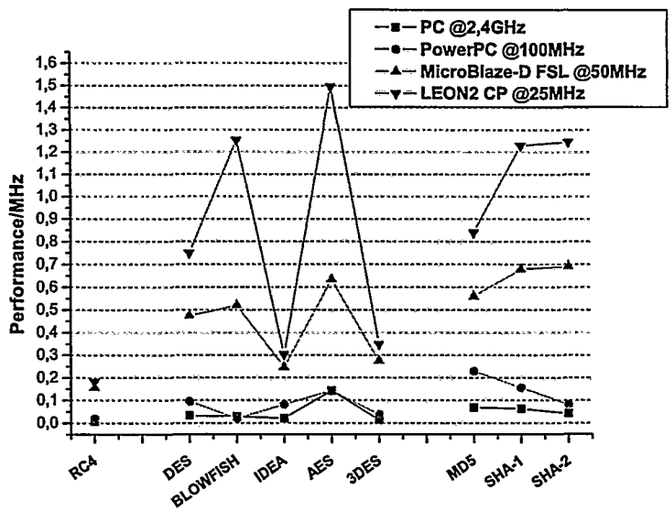


Figura 4.25: Comparación de los diferentes sistemas sobre FPGA en relación al PC [rendimiento/MHz]

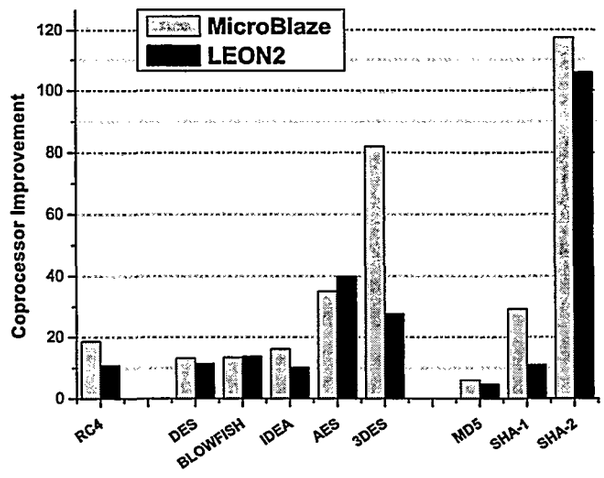


Figura 4.26: Mejora obtenida para cada uno de los SCPs al emplear coprocesador

	MicroBlaze-D OPB			MicroBlaze-D FSL			LEON2 CP		
Algoritmo	Slices	BRAM	% Uso	Slices	BRAM	% Uso	Slices	BRAM	% Uso
IDEA	3336	86	17%	3365	86	18%	6048	48	32%
DES	5462	74	28%	5547	74	29%	8146	48	42%
3DES	13388	74	70%	13396	74	70%	16008	48	83%
AES	6463	74	34%	6576	74	34%	8278	48	43%
BLOWFISH	4217	84	22%	4131	84	22%	8762	48	46%
RC4	2050	76	11%	2112	76	11%	4977	50	26%
MD5	3274	74	17%	3322	74	17%	6628	48	35%
SHA-1	3484	74	18%	3565	74	19%	6250	48	33%
SHA-256	3953	74	21%	4009	74	21%	6672	48	35%

Tabla 4.18: Resultados de implementación de los sistemas SCPs incluyendo los cores hardware

tados de implementación de los sistemas SCP incluyendo los cores hardware para los diferentes algoritmos.

Combinando los resultados del rendimiento de los SCPs y los recursos necesarios del dispositivo reconfigurable, es posible realizar una comparación que relacione ambos resultados. La figura 4.27 muestra una comparativa rendimiento frente a área para los sistemas MBlaze-D y LEON2 con coprocesadores. Los resultados permiten comprobar que MicroBlaze no solo es superior en rendimiento, sino que también emplea menos recursos del dispositivo. Esto confirma el hecho de que MicroBlaze se encuentra específicamente diseñado para FPGAs, y en particular para las FPGAs de Xilinx empleadas en estas pruebas, frente a LEON2, cuyo diseño no solo están pensado para FPGAs sino también para ASICs.

4.8. Análisis de los resultados

Los sistemas embebidos basados en FPGAs no pueden ser comparados favorablemente con las soluciones hardware especificas en FPGA [179, 180, 181, 124, 182, 183, 184], en las cuales los diseños son optimizados y complemente segmentados para obtener altos rendimientos en aplicaciones que requieren de altas velocidades. Sin embargo, cuando las especificaciones del sistema requieren del uso de un microprocesador de propósito general, los SCPs son candidatos adecuados. Como se puede comprobar en la tabla 4.19, los resultados son satisfactoriamente comparados con otros procesadores como el propio procesador LEON [178], el procesador a medida Xtensa [178], algunos DSPs [151, 185], el soft-core SCOB [186] o un procesador a medida para el algoritmo SHA-256 [187].

Otras comparativas con procesadores de propósito general se encuentra disponibles en la bi-

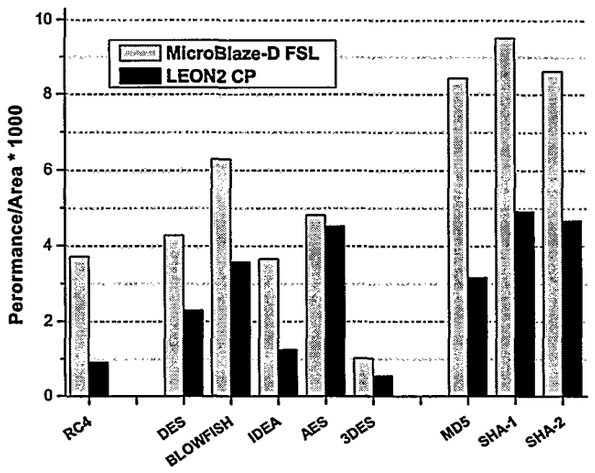


Figura 4.27: Comparación de los diferentes sistemas basados en SCPs con coprocesador [Rendimiento/Área]

Algoritmo	Procesador	Plataforma	Throughput (Mbits/seg)	MicroBlaze-D FSL @50MHz (Mbits/seg)
AES	LEON @50MHz [178]	FPGA	9,1	31.68
	LEON @50MHz [178]	ASIC	34,2	
	Xtensa @50MHz [178]	FPGA	4,57	
	Xtensa @50MHz [178]	ASIC	17,2	
IDEA	TI TMX320C6x @200MHz [151]	DSP	53,1	12,30
	DEC SA-110 @200MHz [151]		32	
DES	TMS320C6201 @200MHz [185]		53	23,71
	TMS320C6211 @150MHz [185]		39	
3DES	TMS320C6201 @200MHz [185]		22	13,73
	TMS320C6211 @150MHz [185]		18	
Blowfish	SCOB @10MHz [186]	FPGA	40	26,01
	SCOB @66MHz [186]	ASIC	266	
SHA-2	SHA-256 Custom-P @66MHz [187]	FPGA	53	34,60

Tabla 4.19: Comparación del rendimiento obtenido para MicroBlaze con otros resultados de la bibliografía en sistemas embebidos

bliografía [180, 188, 189]. En [188, 189] los algoritmos DES, 3DES, IDEA y AES han sido eficientemente ejecutados usando implementaciones software de bajo nivel. Los resultados de estos trabajos no se pueden comparar con la aproximación realizada en este capítulo, donde se han empleado implementaciones estándar de los algoritmos, sin estar optimizadas. En cualquier caso, en los sistemas basados en SCPs (e incluso en el hard-core PowerPC) y teniendo en cuenta que se ha minimizado el esfuerzo de diseño, es posible ejecutar aplicaciones de seguridad sobre conexiones de red de 10/100 Mbps.

Finalmente, aunque los resultados de la tabla 4.19 son adecuados al compararlos con sistemas embebidos equivalentes presentes en la bibliografía, es necesario hacer ciertas consideraciones sobre la comparación con las soluciones hardware sobre FPGAs, o con las implementaciones a bajo nivel. Por un lado, mientras los diseñadores pueden emplear las FPGAs para crear rápida y eficientemente diseños hardware, muchos sistemas requieren de la combinaciones tanto de hardware como de software. Los SCPs permiten a los diseñadores disponer de la flexibilidad para configurar los procesadores y facilitan a estos diseñadores construir sistemas FPGA que incorporen uno o varios procesadores y coprocesadores. Por otro lado, para los diferentes experimentos se han empleado algoritmos y cores hardware no optimizados, por lo que si se incrementa el esfuerzo de diseño es posible obtener rendimientos superiores a las obtenidos en este capítulo.

4.9. Sistemas multiprocesador basado en SCPs

En las secciones anteriores se ha abordado la creación de sistemas completos sobre FPGA basados en el uso de un procesador soft-core con un conjunto de periféricos. Del mismo modo, como forma de incrementar el rendimiento del procesador se incluye en el sistema un coprocesador hardware específico para la tarea a realizar. Sin embargo, una tercera posibilidad que no se ha presentado anteriormente, es el diseño de sistemas basados en múltiples unidades de procesamiento. En el caso del hardware reconfigurable, la cantidad de unidades de procesamiento que pueden emplearse se encuentra limitada tan solo por la cantidad de recursos lógicos disponibles.

Estas soluciones basadas en un conjunto de procesadores pueden se homogéneas, es decir, hacen uso de procesadores del mismo tipo, o heterogéneas, donde se emplean diferentes unidades de procesamiento. Un ejemplo de unidad homogénea sería un sistemas basados en uno o varios procesadores MicroBlaze [190, 191]. En cuanto a sistemas heterogéneos, se podría considerar aquel formado por un procesador PowerPC, disponible en un dispositivo Virtex-II Pro, junto con uno o varios procesadores soft-core como MicroBlaze, o bien emplear por ejemplo un sistema basado en diferentes soft-cores [192]. Del mismo modo, dentro de cada sistema multiprocesador es posible emplear diferentes técnicas para resolver el intercambio de datos, ya sea mediante buses

dedicados [190, 191, 193] o implementando arquitecturas multiprocesador [194].

En esta sección se quiere mostrar las posibilidades que ofrece una arquitectura multiprocesador basada en procesadores LEON3 [195]. LEON3 es la nueva versión del procesador LEON que presenta una variante con arquitectura multiprocesador, LEON3MP [194]. El procesador LEON3 se encuentra en el momento de estas pruebas en estado de desarrollo, pero ha sido posible implementar un sistema multiprocesador para su evaluación.

4.9.1. LEON3MP

El procesador LEON3 [195] es una versión mejorada del procesador LEON2, que al igual que éste, dispone del código fuente bajo licencia GPL. Entre las nuevas características del LEON3 se encuentran las siguientes:

- Set de instrucciones SPARC V8 con soporte para las extensiones V8e.
- Unidades de enteros con 7 etapas de segmentación mejoradas.
- Unidades hardware para la multiplicación, división y operaciones MAC.
- FPU IEEE-754 de alto rendimiento y completamente segmentada.
- Caches de instrucciones y datos separadas (Arquitectura Harvard) con capacidad de infiltrado.
- Caches configurables: 1 - 4 sets, 1 - 256 kbytes/set. Algoritmos de reemplazo: Random, LRR o LRU.
- SPARC Reference MMU (SRMMU) con TLB configurable
- Interfaz de bus AMBA-2.0 AHB
- Unidades de debug on-chip avanzada con buffer para trazas de instrucciones y datos.
- Soporte para Symmetric Multi-processor (SMP)
- Diseño robusto y completamente síncrono.
- Hasta 125 MHz en FPGA y 400 MHz en tecnología ASIC 0.13 um.
- Ampliamente configurable.
- Extenso conjunto de herramientas: compiladores, kernels, simuladores y monitores para debug.

El procesador LEON3 se distribuye como parte de la librería GRLIB IP [196], permitiendo una integración simple en diseños SoC complejos. GRLIB además incluye un diseño multi-procesador basado en LEON3 con hasta 4 procesadores y un amplio conjunto de periféricos on-chip. LEON3MP [194] es altamente reconfigurable, y consiste en los siguientes cores IP (figura 4.28):

- 1 - 4 procesadores LEON3 con soporte para arquitectura multiprocesador.
- Unidad de debug (DSU) con soporte para multiprocesador para LEON3.
- Controlador de memorias PROM/SRAM de 32 bits.
- Controlador de memorias PROM/SRAM/SDRAM de 8/16/32/64 bits.
- Interfaz PCI de 32 bits (target-only o initiator/target) con soporte para FIFO y DMA.
- Arbitro de bus AHB con algoritmo Round-robin y control con soporte plug&play.
- Puente AHB/APB con soporte plug&play.
- Controlador de interrupciones multiprocesador.
- Unidad de temporizaciones modular de 32 bits.
- 1 o 2 UARTs con FIFOs.
- Ethernet MAC 10/100.
- Interfaz CAN.
- Link de comunicación para debug serie, Etherner y JTAG.

Para esta tesis se ha empleado la versión GRLIB 0.16, la más moderna en ese momento. Actualmente la librería GRLIB presenta su primera versión estable.

4.9.2. eCos para procesadores LEON

La posibilidad de hacer uso de sistemas con arquitectura multiprocesador pasa por disponer del soporte software necesario. Inicialmente podría ser posible realizar una aplicación que habilitara el soporte para manejar todo el mecanismo de comunicación entre los procesadores, pero es cierto que en muchos casos el esfuerzo necesario para realizar esta aplicación no justifica en parte la pérdida de rendimiento que supone emplear un sistema operativo, el cual simplifica todo el proceso para el desarrollo de aplicaciones que hacen uso de varios procesadores. Actualmente,

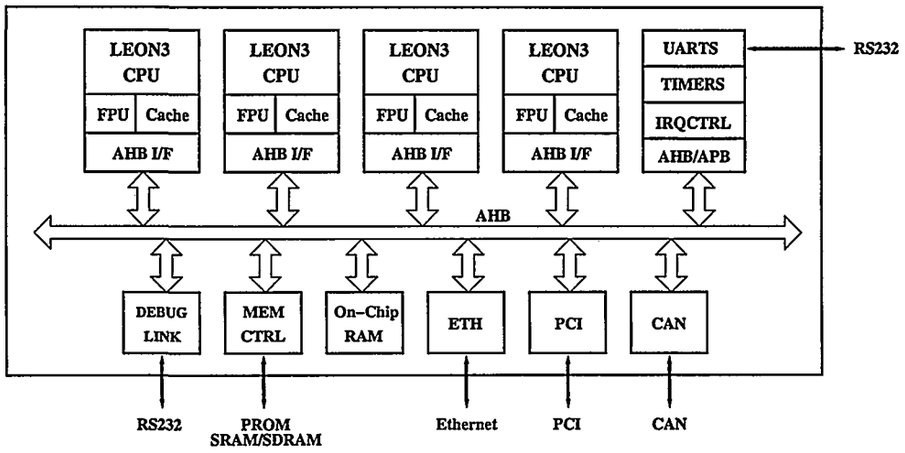


Figura 4.28: Arquitectura multiprocesador basada en LEON3

sólo el sistema operativo eCos permite manejar de forma sencilla un sistema basado en la arquitectura LEON3MP, siendo posible manejar hasta dieciséis procesadores LEON3. Sin embargo, la arquitectura LEON3MP actual soporta cuatro procesadores.

eCos es un sistema operativo de tiempo real *open source* y libre de *royalties* para aplicaciones embebidas. La naturaleza altamente configurable de eCos permite al sistema operativo ser ajustado para los requerimientos de la aplicación, permitiendo obtener el mejor rendimiento posible y optimizado los recursos hardware necesarios. La versión de eCos para los procesadores LEON2 y LEON3 incluye la siguiente funcionalidad:

* Soporte para Symmetric Multi-processing (SMP) en LEON3.
* Soporte para FPU en hardware.
* Drivers de red para el chip LAN91C111 y la MAC de OpenCores de 10/100 Mbits.
* Single-vector Trapping (SVT).

4.9.3. Experimentos

Siguiendo con la metodología de pruebas empleada, se pretende comprobar el potencial de un sistema multiprocesador basado en LEON3MP sobre la plataforma RC1000PP. Para ello se propone la ejecución del primer conjunto de experimentos propuesto anteriormente, es decir, se procede a la ejecución de una aplicación que realice el cifrado y sucesivo descifrado de hasta 4

Mbytes de datos alojados en memoria externa.

El primer paso para realizar este conjunto de experimentos es adaptar el programa de prueba a las exigencias del sistema operativo eCos. Ya que el sistema dispone de cuatro procesadores, y que todos deben trabajar simultáneamente, se decide dividir el proceso de cifrado en cuatro tareas iguales. Cada parte se declara como un *thread* dentro del sistema operativo, de modo que cada tarea acceda a un rango determinado de la memoria correspondiente a los datos que esa tarea debe cifrar/descifrar, por tanto, la memoria se encuentra dividida en cuatro partes. De este modo cada tarea operará sobre una parte de la memoria, pero todas las tareas ejecutarán el mismo programa. Para la medición de los tiempos se emplearon las funciones que eCos provee para ello.

Un análisis de la arquitectura del sistema LEON3MP a partir de la figura 4.28 nos permite determinar que todos los procesadores comparten el bus de acceso a la memoria externa, siendo el acceso a memoria el probable cuello de botella. Es por ello que el tamaño de la caché, en particular de la caché de instrucciones, se convierte en el componente más crítico del sistema. Esto último también se ha determinado en los resultados obtenidos para los sistemas con un solo procesador, donde las cachés fueron importantes para acelerar el rendimiento, al emplear memorias más rápidas y evitando salir al exterior de la FPGA. Para comprobar esta posibilidad, se establecen dos implementaciones con cuatro procesadores para cada una de las configuraciones de caché de instrucciones y cache de datos siguientes:

- 1 vía x 1 K palabras x 32 bits
- 2 vías x 2 K palabras x 32 bits

El objetivo de las diferentes configuraciones es comprobar que con un tamaño adecuado de la caché, es posible que todas las instrucciones a ejecutar se almacenan en la caché de instrucciones, reduciendo el acceso a memoria externa únicamente para coger los datos a manipular.

4.9.4. Resultados

Los resultados obtenidos de la ejecución del experimento propuesto se muestran en las tablas 4.20 y 4.21 para cada uno de los algoritmos seleccionados, y cada una de las configuraciones de caché propuestas. De estos resultados se puede concluir que efectivamente el tamaño de la caché es fundamental para obtener mejores resultados, justificando la idea inicial de que el acceso a la memoria externa es el cuello de botella en estos sistemas. Del mismo modo, a mayor número de procesadores mayor rendimiento, el cual puede incluso llegar a ser ideal (cuatro veces mayor para cuatro procesadores), siempre que el tamaño de la caché sea adecuado para almacenar el programa.

Profundizando más en los resultados obtenidos, la figura 4.29 muestra la evolución de los resultados cuando se emplea la configuración de caché más grande en función del número de

Algoritmo	Configuración Caché 1x1x32							
	1 cpu		2 cpu		3 cpu		4 cpu	
	# de ciclos	Tiempo Ejecución (s)	# de ciclos	SpeedUp	# de ciclos	SpeedUp	# de ciclos	SpeedUp
IDEA	4578	96	2575	1,78	1955	2,34	1764	2,59
DES	2003	42	1669	1,20	1621	1,24	1574	1,27
3DES	6151	129	5102	1,21	4864	1,26	4864	1,26
AES	3719	39	3052	1,22	2861	1,30	2861	1,30
BLOWFISH	2861	60	1621	1,76	1192	2,40	1001	2,86
RC4	7534	79	4387	1,72	3529	2,14	3338	2,26
MD5	4578	12	3815	1,20	3815	1,20	3815	1,20
SHA-1	8774	23	7248	1,21	7248	1,21	7248	1,21
SHA-256	19073	50	16022	1,19	15640	1,22	15640	1,22

Tabla 4.20: Resultados obtenidos por el sistema LEON3MP - Caché 1x1x32 (Speedup = Tejec_1cpu / Tejec_Ncpu)

Algoritmo	Configuración Caché 2x2x32							
	1 cpu		2 cpu		3 cpu		4 cpu	
	# de ciclos	Tiempo Ejecución (s)	# de ciclos	SpeedUp	# de ciclos	SpeedUp	# de ciclos	SpeedUp
IDEA	3004	63	1526	1,97	1001	3,00	763	3,94
DES	715	15	381	1,88	286	2,50	238	3,00
3DES	4482	94	3862	1,16	3862	1,16	3862	1,16
AES	3242	34	2670	1,21	2575	1,26	2575	1,26
BLOWFISH	1955	41	1001	1,95	668	2,93	525	3,73
RC4	5341	56	2670	2,00	1907	2,80	1717	3,11
MD5	2289	6	1526	1,50	1144	2,00	1144	2,00
SHA-1	6866	18	6104	1,13	6104	1,13	6104	1,13
SHA-256	14877	39	13351	1,11	13351	1,11	13351	1,11

Tabla 4.21: Resultados obtenidos por el sistema LEON3MP - Caché 2x2x32 (Speedup = Tejec_1cpu / Tejec_Ncpu)

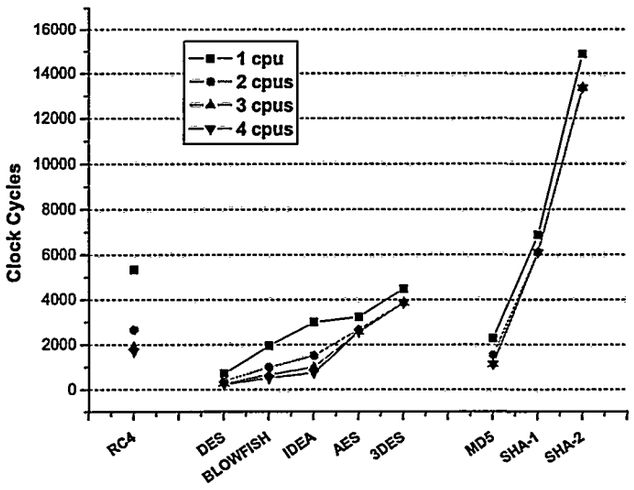


Figura 4.29: Evolución del rendimiento en función del número de procesadores en LEON3MP

procesadores empleados. Para algunos algoritmos se consigue mejorar proporcionalmente el rendimiento a medida que se añaden procesadores, y para otros algoritmos a partir de un número determinado de procesadores, no se incrementa el rendimiento.

La figura 4.30 muestra una comparación entre el rendimiento obtenido con los diferentes sistemas LEON3MP frente a LEON2. Se puede comprobar que para algunos algoritmos el rendimiento obtenido para el procesador LEON3 es superior que en LEON2, como se esperaba por las mejores características disponibles en LEON3. Sin embargo, existen algunos casos en los que el procesador LEON2 presenta un mejor rendimiento, lo que se debe en mayor medida a que LEON2 presenta una configuración de caché de 1x8x32, es decir, tiene una cache de correspondencia directa de 8 Kbytes, lo que puede beneficiar a algunos de los algoritmos. En cualquier caso, también es necesario tener en cuenta que en esta comparativa, LEON3 debe ejecutar el sistema operativo, lo cual supone una sobrecarga no despreciable que afecta particularmente al procesador principal de la arquitectura. De igual forma, resulta significativo que la caché sea un elemento tan importante que el resultado obtenido en LEON3MP con 1 CPU, para la configuración de caché 2x2x32, es mejor que el resultado obtenido con 4 CPUs para la configuración de caché 1x1x32 para casi todos los algoritmos. Lo mismo ocurre cuando se comparan los resultados empleando 4 CPUs, LEON3MP en su configuración 2x2x32 es superior a LEON2 en todos los algoritmos con la excepción de

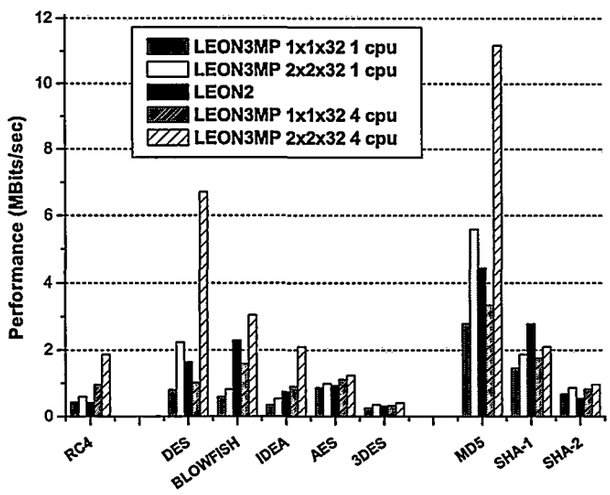


Figura 4.30: Comparativa de los diferentes resultados en LEON3MP incluidos los resultados de LEON2

SHA-1, y en algunos casos el incremento en el rendimiento es casi ideal, como se demuestra en los algoritmos RC4, DES y sobretodo MD5.

Para finalizar, destacar que no solo es importante fijarse en el rendimiento obtenido. Si tenemos en cuenta el área necesaria para implementar los sistemas LEON3MP con 4 CPUs, la relación rendimiento/área no resulta adecuada, siendo mucho más recomendable la solución basada en procesador y coprocesador. Sin embargo, a nivel de tareas software puede ser posible que la solución multiprocesador se muestre como una solución adecuada en aquellas aplicaciones que requieran ejecutar diferentes tareas. Por otro lado, queda pendiente la posibilidad de configurar a cada procesador con un coprocesador específico, algo que haría muy interesantes a estos sistemas.

4.10. Conclusiones

En este capítulo se han presentado diferentes implementaciones de algoritmos criptográficos sobre procesadores embebidos en FPGA. Los sistemas basados en FPGAs permiten combinar los microprocesadores de propósito general con cores hardware como soluciones SoC. Algunos factores limitan la mejora de rendimiento en los sistemas embebidos basados en FPGAs: la tecnología

de la lógica reconfigurable, la tasa de transferencia de los datos entre el procesador embebido y la lógica reconfigurable, la velocidad del procesador embebido, y además, el ancho de banda de la memoria. Los resultados prueban que los cuellos de botella más importantes son el ancho de banda y la latencia de interfaz que conecta al procesador embebido con la lógica configurable.

Los algoritmos de criptografía no son aplicaciones muy adecuadas para procesadores de propósito general [197], por lo que su rendimiento se ve claramente favorecidos por el uso de coprocesadores dedicados. Los dos SCPs evaluados se encuentran a la altura de lo esperado, siendo muy superiores al procesador PowerPC cuando se hace uso de coprocesadores específicos. Los diferentes recursos hardware disponibles para mejorar la arquitectura de los SCPs son determinantes para obtener un buen rendimiento en la ejecución de los diferentes algoritmos de criptografía en software, e incluso en el incremento de rendimiento cuando se emplean coprocesadores. Naturalmente, los rendimientos obtenidos al ejecutar los diferentes algoritmos en software no son suficientemente adecuados, por lo que el uso de coprocesadores específicos permite mejorar el rendimiento final de los algoritmos, posibilitando su uso de forma óptima en aplicaciones de cifrado de datos, todo ello, a pesar de que la implementación de los diferentes módulos hardware no haya sido la más eficiente posible. Al mismo tiempo, la elección de la interfaz de conexión entre procesador y coprocesador es determinante para los resultados finales. Si la mejor interfaz es seleccionada, el diseñador puede obtener una mejora de más de 2 ordenes de magnitud respecto a la solución software pura, sin un gran esfuerzo de diseño.

Finalmente, en este capítulo se ha presentado una solución multiprocesador como otra de las ventajas que ofrecen los SCPs. Los sistemas embebidos basados en FPGA pueden disponer de uno o más procesadores, de modo que se puede mejorar el rendimiento de las aplicaciones que presentan algún grado de paralelismo. En cualquier caso, el número de unidades procesador que se pueden emplear se encuentra únicamente limitado por la cantidad de recursos del dispositivo.

Capítulo 5

Auto-reconfiguración en Sistemas basados en SCPs

5.1. Introducción

Los sistemas embebidos basados en FPGA son una buena solución para el diseño y uso de coprocesadores hardware, al proveer la FPGA por sí misma los recursos lógicos necesarios para implementarlos, y porque estos aceleradores añaden la potencia computacional necesaria para hacer competitivos los modestos microprocesadores que usualmente se implementan en los diseños de lógica programable [93, 198, 197]. Sin embargo, el problema surge cuando el procesador debe ejecutar varios algoritmos en hardware, lo que supone desarrollar un coprocesador por tarea, ocupando una parte sustancial del área de la FPGA. Recursos que estarán desaprovechados parcialmente porque solo el coprocesador asociado a la tarea que se ejecuta en ese momento estará activo. Otra desventaja del desarrollo de aplicaciones con partes aceleradas en hardware es que se reduce mucho la versatilidad del sistema, al fijarse las aplicaciones en tiempo de diseño, y por tanto, también el coprocesador que deben emplear. Estos dos problemas pueden resolverse si el procesador dispone de la capacidad de cambiar en tiempo de ejecución, el coprocesador que necesita para ejecutar una cierta tarea. Con esta capacidad son posibles nuevas áreas de aplicación, en particular, actualizaciones de hardware y reconfiguración en tiempo de ejecución.

La reconfiguración parcial permite cargar diferentes diseños en la misma área del dispositivo, cambiar parte de un diseño sin hacer un *reset* o reconfigurar completamente el dispositivo. Esta capacidad hace mucho más atractivas a las FPGAs para su uso en sistemas embebidos porque permite habilitar el intercambio de tareas hardware. Actualmente ya existen diferentes desarrollos

académicos donde se intenta explotar esta novedosa característica [35, 118, 199, 200, 201]. Al mismo tiempo, esta capacidad puede permitir el intercambio de los diferentes módulos coprocesador de un sistema embebido basado en SCPs, resolviendo algunos de los problemas clásicos de los sistemas embebidos. En primer lugar, se resuelve el problema de la flexibilidad de los sistemas que emplean aceleradores hardware, ya que al disponer de un hardware fijo adecuado a las aplicaciones iniciales del sistema, no es posible ejecutar nuevas aplicaciones de forma eficiente. Pero incluso si esta capacidad de adaptación no fuera necesaria porque el código que se está ejecutando es fijo, como ocurre en muchos sistemas embebidos, la reconfiguración en tiempo de ejecución resulta interesante para reducir el área que se emplea en la FPGA. Por ejemplo, si un sistema tiene que dar soporte a varias aplicaciones aceleradas por hardware, la metodología clásica implementaría un coprocesador para cada algoritmo. Si usamos reconfiguración en tiempo de ejecución, solo el coprocesador para la tarea que actualmente se está ejecutando tendría que ser cargado en la FPGA. Posteriormente si ya no es necesario, sería eliminado del dispositivo dejando espacio para uno nuevo. Este último aspecto presenta otra interesante ventaja ya que hace posible reducir el consumo del sistema [202].

El mejor modo para explotar estas ventajas es usar sistemas auto-reconfigurables [56], es decir, aquellos donde la FPGA es capaz de modificar su propia configuración. De este modo es posible disponer de una verdadera solución SoC sin la necesidad de añadir un microcontrolador u otros componentes externos para realizar la reconfiguración. En este capítulo, se presenta un sistema auto-reconfigurable con la capacidad de reconfigurar en tiempo de ejecución su coprocesador con el objetivo de soportar el mayor número de coprocesadores, y en particular, de coprocesadores criptográficos. Se muestra un ejemplo de aplicación real que requiere de la necesidad de emplear reconfiguración parcial y auto-reconfiguración. Esta aplicación es SSH (Secure Shell), que permite llevar a cabo una comunicación segura entre hosts. Lo que hace particularmente interesante a esta aplicación es que los diferentes algoritmos de criptografía que se pueden emplear, son negociados entre las distintas partes y por tanto, solo en tiempo de ejecución se conoce el algoritmo a utilizar. Este desconocimiento del algoritmo requiere que todos los posibles algoritmos a emplear se encuentran disponibles como coprocesadores hardware, lo que supone que el dispositivo reconfigurable disponga de los recursos suficientes para todos ellos. Sin embargo, este problema se puede resolver mediante Reconfiguración Parcial, como se ha comentado anteriormente. Las aplicaciones de cifrado seguras no son las únicas que manejan una selección dinámica de algoritmos, por lo que es posible encontrar otras áreas donde un sistema auto-reconfigurable puede ofrecer muchas ventajas. La capacidad de reconfiguración parcial del coprocesador puede ser explotada de forma adecuada para mejorar el rendimiento final del sistema, eliminando la necesidad de implementar todos los algoritmos que requiere la aplicación en el dispositivo.

5.2. Sistemas embebidos basados en MicroBlaze

En el capítulo anterior se compararon los SCPs MicroBlaze y LEON2. Los resultados obtenidos mostraron que MicroBlaze presenta ciertas ventajas sobre LEON2, ya que alcanza un rendimiento superior con un coste de área menor, al ser MicroBlaze un procesador especialmente desarrollado para los dispositivos de Xilinx. Es por ello que a la hora de implementar un CSoC, disponer de una mayor cantidad de área libre es muy importante para la creación de sistemas auto-reconfigurables. De este modo, más recursos del dispositivo se pueden emplear para los diferentes coprocesadores que se necesiten, y se imponen menos restricciones para el desarrollo de sistemas parcialmente reconfigurables como se verá más adelante. Por todo ello, MicroBlaze se ha seleccionado para el diseño de nuestro sistema embebido auto-reconfigurable en FPGA, y la interfaz FSL se ha elegido para la conexión de los coprocesadores.

5.2.1. Requerimientos de los coprocesadores de cifrado

La solución clásica para diseñar un sistema que de soporte a un conjunto de algoritmos es implementarlos todos en el mismo dispositivo. Esto requiere, por un lado, disponer de un dispositivo de gran tamaño, y por otro, optimizar adecuadamente los diseños para conseguir que todos operen a las velocidades adecuadas. Si nos centramos en los coprocesadores desarrollados en el capítulo anterior para MicroBlaze a través de la interfaz FSL, los recursos necesarios se presentan en la tabla 5.1. A esa cantidad habría que sumarle los recursos necesarios para implementar un sistema MicroBlaze, por ejemplo, el sistema MBlaze-D.

La cantidad de recursos finales necesarios para nuestro sistema embebido basado en MicroBlaze están disponibles sobradamente en los dispositivos más grandes de algunas familias de FPGAs de Xilinx, como la familia Virtex-II. Sin embargo, al implementar todos los algoritmos en el mismo dispositivo es necesario realizar una interfaz para multiplexar los diferentes coprocesadores, dado que el número de canales FSL es limitado, lo puede afectar al rendimiento final de cada algoritmo. Además, el gasto de recursos del dispositivo reconfigurable no está justificado, ya que solo uno de los coprocesadores estará activo al tiempo. La solución a este problema es un sistema auto-reconfigurable, donde se configura únicamente el coprocesador necesario, y el procesador realiza las diferentes reconfiguraciones a lo largo de la ejecución de una aplicación.

5.2.2. Plataforma reconfigurable

La creación de un sistema auto-reconfigurable requiere de una plataforma que permita al dispositivo FPGA conectarse a la interfaz de configuración. En las FPGAs de Xilinx, solo las interfaces JTAG y SelectMap soportan reconfiguración parcial. Las familias más modernas, como Virtex-

Algoritmo	Slices	BRAM
IDEA	1744	12
DES	3920	-
3DES	11786	-
AES	4997	-
BLOWFISH	2496	10
RC4	719	2
MD5	2412	-
SHA-1	2134	-
SHA-256	2572	-
Sistema MBlaze-D	1321	74
Total	34101	98

Tabla 5.1: Recursos lógicos empleados por un sistema MBlaze-D incluyendo todos los cores FSL

II y Virtex-4, disponen del componente ICAP, lo que las convierte en la opción perfecta para la auto-reconfiguración.

De las plataformas disponibles en el laboratorio, solo la placa de Alpha-Data dispone de ICAP. Sin embargo, no dispone de recursos lógicos suficientes para implementar el sistema MBlaze-D junto con el coprocesador de mayor tamaño. Además, el pinout de esta plataforma no es adecuado para el desarrollo de sistemas parcialmente reconfigurables. Todos estos inconvenientes hacen que la plataforma RC1000PP haya sido seleccionada de nuevo. Esta plataforma incluye un pinout adecuado, pero no dispone de ICAP, lo que hace necesario buscar una solución para resolver el problema de la auto-reconfiguración. La solución más sencilla es realizar una conexión externa que permita a la FPGA manejar la interfaz SelectMap, pero al tratarse de una plataforma PCI, la interfaz SelectMap es compartida con el bus PCI para la reconfiguración desde el PC, complicando un poco el diseño. A pesar de este problema, se ha realizado un sistema parcialmente reconfigurable, detallado en la sección 5.5, donde la reconfiguración de los diferentes coprocesadores se ha realizado manualmente desde el PC.

Tras realizar las pruebas con éxito sobre la plataforma RC1000PP, ha sido necesario encontrar otra plataforma para diseñar un sistema auto-reconfigurable completo. La placa de desarrollo con Spartan-3 de Avnet junto con el módulo de comunicaciones, mostrada en la figura 5.1, fue la seleccionada. La distribución de pines de esta plataforma ha permitido realizar con éxito los diferentes sistemas parcialmente reconfigurables, y al mismo tiempo, se han realizado sin demasiados problemas las conexiones externas hacia la interfaz SelectMap, ya que Spartan-3 tampoco dispone de ICAP. Estas conexiones se corresponden con el cable plano que se aprecia sobre la imagen 5.1. El uso de un dispositivo Spartan-3 permite demostrar que se pueden crear sistemas

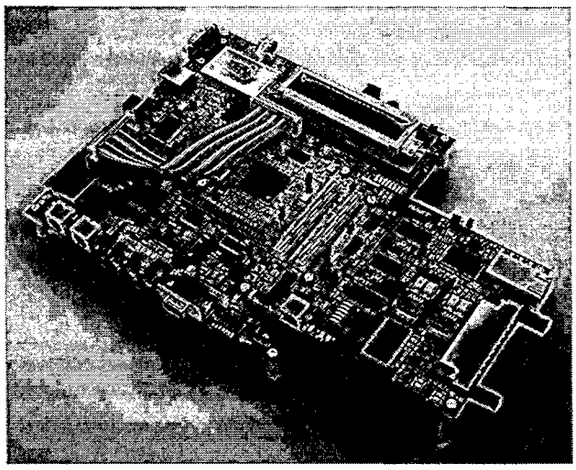


Figura 5.1: Plataforma Spartan-3 auto-reconfigurable

auto-reconfigurables en dispositivos de bajo coste, y además, ha supuesto el interesante reto de realizar pruebas de reconfiguración sobre un dispositivo que presenta ciertas restricciones para la realización de sistemas parcialmente reconfigurables.

5.3. Auto-reconfiguración en dispositivos Spartan-3

Desde 1998 con la familia Virtex, todas las FPGAs de Xilinx son parcialmente reconfigurables en tiempo de ejecución, esto es, parte del chip puede ser cambiado mientras el resto continua su operación normal. En Virtex, la unidad mínima que puede ser reconfigurada es un *frame*, que se corresponde con una columna vertical de slices del chip, y se garantiza que si algunos bits del nuevo *frame* no cambian con respecto a la configuración existente, no habrá *glitches* en los bits durante la reconfiguración. Dicho de otro modo, es posible cambiar un único bit de la configuración de la FPGA, mientras el resto de los bits del mismo *frame* no cambian. Desafortunadamente, esta característica fue eliminada en la familia de bajo coste Spartan-3, por lo que no se asegura que aparezcan *glitches* durante la reconfiguración. Además, la mínima unidad de configuración fue incrementada desde un *frame* a una columna de CLBs completa (19 *frames*). En todo caso, estas dos restricciones no suponen un problema ya que solo obligan a usar columnas completas de CLBs (incluyendo IOBs) para al definir el área ocupada por el coprocesador reconfigurable.

Otro inconveniente de los dispositivos de bajo coste como Spartan-3 es que no disponen de ICAP, por lo que se hace necesario disponer de un circuito externo conectado a la interfaz SelectMap, si se quiere desarrollar sistemas auto-reconfigurables. Este circuito puede estar formado por un I/O de propósito general (GPIO) del procesador embebido (por ejemplo, MicroBlaze), conectado externamente a los pines de la interfaz de configuración paralela. En total son necesarias 11 conexiones: 8 bits de datos, el reloj de configuración (CCLK), el selector de chip (CS_B) y la señal de lectura/escritura (RDWR_B). El modo de configuración debe ser establecido como esclavo paralelo para que la señal CCLK sea generada por el GPIO, y así la configuración pueda ser controlada por el software que se ejecuta en el procesador.

Si bien este procedimiento soluciona la reconfiguración parcial en tiempo de ejecución, se requiere de una primera configuración inicial al alimentar el chip. Este problema se resuelve empleando una memoria Flash de configuración que soporte el modo de configuración esclavo paralelo. Siguiendo el diagrama de conexiones recomendado por Xilinx [203], se evitan los conflictos entre los pines del GPIO y la memoria. En este diagrama, figura 5.2, la señal DONE de la FPGA está conectada a la señal de habilitación de la memoria (activa en bajo), de modo que cuando la FPGA finaliza su configuración, DONE se activa, los pines del GPIO abandonan el estado de alta impedancia (tri-estado) y la memoria se encuentra deshabilitada. La única consideración a tener en cuenta, es que el *bitstream* almacenado en la memoria debe generarse empleando la opción *"-g Persist:Yes"* de *bitgen*, permitiendo que la interfaz SelectMap se mantenga después de la primera configuración.

5.4. Metodología de diseño

La reconfiguración parcial requiere de herramientas automáticas de diseño que permitan modificar algunos bloques del circuito, manteniendo el resto fijos, y deben asegurar que el emplazamiento y el rutado de los bloques que están siendo modificados, no se solape con el resto. La solución de Xilinx a este problema es Modular Design, un flujo de diseño que permite construir el *layout* final de la FPGA a partir de módulos independientes, implementados en una sección rectangular del dispositivo. La creación de un diseño parcialmente reconfigurable, donde los módulos se comunican entre sí, requiere seguir un conjunto específico de reglas [204, 20].

5.4.1. Buses macro

Modular Design junto con el modo RECONFIG de la restricción AREA_GROUP [20] asegura que tanto el rutado como el emplazado de un módulo esté confinado en el área rectangular de la FPGA. Es importante mantener fijado el rutado en las áreas de la FPGA para evitar interferencias

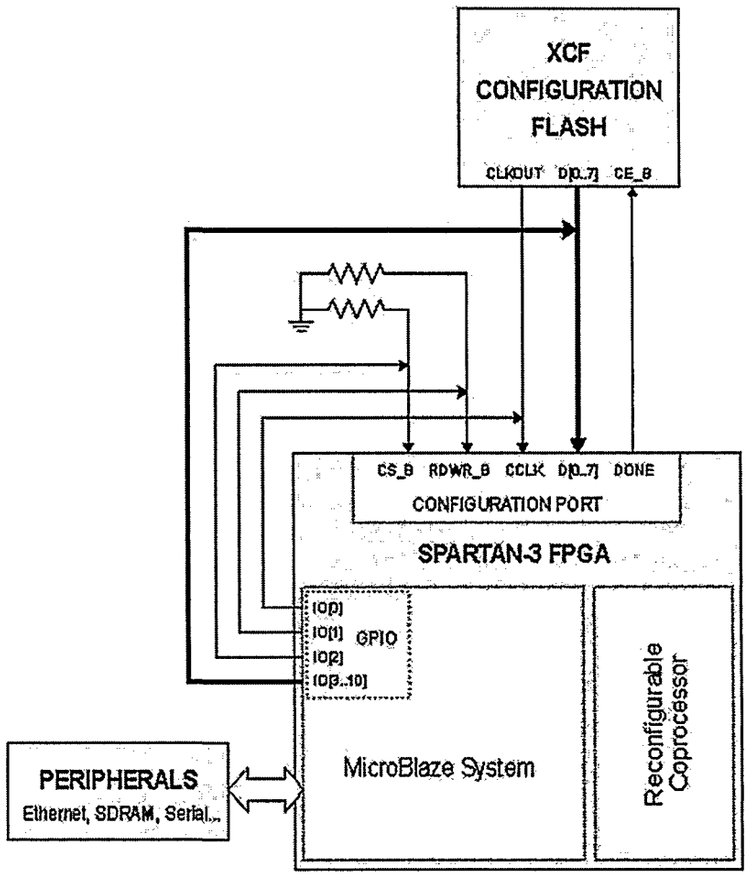


Figura 5.2: Esquema del sistema auto-reconfigurable basado en Spartan-3

con el resto del circuito durante el proceso de reconfiguración. El problema surge cuando se interconectan dos módulos, porque el modo RECONFIG evitará que cualquier conexión sea rutada fuera del módulo asignado. La solución es usar un componente para las interconexiones, que no pertenezca a ninguno de los módulos. Xilinx provee el bus macro [20], el cual implementa las conexiones usando un par de buffers tri-estado (TBUFs): un TBUF estará localizado en el área reservada para el primer módulo, y el otro en el espacio del segundo módulo. Dependiendo de que TBUF se encuentre en tri-estado, la comunicación va del primer módulo al segundo o viceversa. Este componentes se implementa como un *hard macro* para evitar que el rutado a través de los módulos cambie cuando se reimplementen los módulos reconfigurables. Las líneas de conexión tienen el atributo IS_BUS_MACRO para evitar que la herramienta de *place&route* muestre un error al comprobar que el rutado cruza de un módulo a otro.

5.4.2. Restricciones de emplazamiento

Las restricciones de emplazamiento se emplean exclusivamente para reservar un área del dispositivo para los diferentes módulos del diseño. Estas restricciones solo se pueden realizar en base al tamaño de los módulos, que pueden ser estimados después de la compilación del *top level* que contiene a los módulos como se define en la fase *Initial Budgeting*. La herramienta Xilinx PACE [20] puede ser empleada para visualizar gráficamente cada uno de los módulos.

Es muy importante destacar que todos los pines empleados por un módulo deben están fijados en el área (conjunto de columnas) ocupadas por ese módulo. De otro modo, se hace necesarios el uso de buses macro. El 80 % del desarrollo del diseño se emplea en resolver los problema de emplazado, lo cual no está relacionado con la reconfiguración parcial directamente. Desafortunadamente, la mayoría de las plataformas comerciales no están diseñadas para el propósito de la reconfiguración parcial. Es recomendable por tanto, conocer la asignación de pines de la plataforma antes de emplearla para el diseño de sistemas parcialmente reconfigurables.

5.4.3. Reconfiguración parcial de módulos

El proceso de generación de diseños para la reconfiguración parcial de módulos (*Module-Based*) está divido en 3 fases principales, mostradas en la figura 5.3, y que se detallan a continuación.

Planificación Inicial (*Initial Budgeting*)

Durante esta fase se describen todos los ficheros HDL que componen el sistema, así como el fichero UCF de restricciones siguiendo las siguientes reglas:

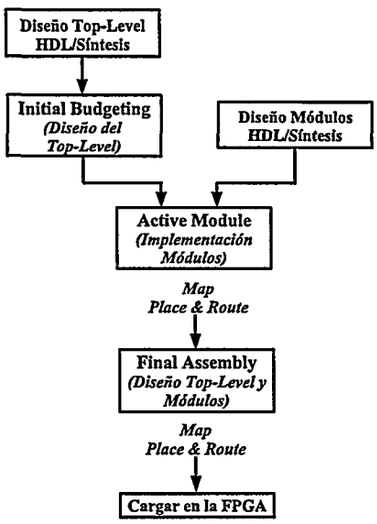


Figura 5.3: Flujo de diseño de Modular Design

- El área de los módulos debe ocupar un área mínima de 4 slices y tener siempre un ancho múltiplo de 4 slices (ej: 4, 8, 12, etc). Además deben ser igual de altos que el dispositivo. Estas especificaciones deben ser incluidas en el fichero UCF.
- Todos los IOBs deben estar contenidos en el espacio rectangular de su módulo asociado y fijados en esas posiciones. El resto de la lógica que no pertenece a los módulos debe tener restricciones para emplazarse en un sitio fijo. No debe haber por tanto, ninguna lógica en el *top level* sin restricciones.
- Las restricciones para cada bus macro deben ser insertadas manualmente en el fichero UCF, de modo que se localicen entre los módulos actuando de puentes de comunicación.

Los *buses macro* son elementos clave que permite realizar diseños parcialmente reconfigurables. Estos no son más que *hard macros* es decir, lógica ya mapeada sobre la FPGA y que se "pega" en la fase de *place&route* justo en la frontera entre los módulos. Esta lógica se conserva fija en el mismo sitio cuando se produce el proceso de reconfiguración parcial, tomándose como referencia a la hora de realizar conexiones que deben ser conservadas de un módulo reconfigurable a otro. Xilinx distribuye un *bus macro* [20] que puede ser utilizado para esta tarea.

La salida de esta fase es un fichero UCF que contiene todas las restricciones de emplazado y opcionalmente, de tiempos. Este fichero se empleará durante la implementación de cada uno de los módulos. La diferencia con el flujo de diseño estándar es que todas las entradas y salidas de los módulos reconfigurables deben estar conectadas a I/O, lógica global o buses macros, y ninguna señal pasa de un módulo a otro sin emplear un bus macro.

Generación de Módulos (*Active Module*)

En este momento el diseño ha sido sintetizado y se han definido las restricciones de emplazamiento. Se puede iniciar por tanto la implementación de todos los módulos. Cada módulo puede ser implementado separadamente pero siempre en el contexto del *top level* y de las restricciones especificadas. Posteriormente se pueden generar los *bitstreams* para cada uno de los módulos reconfigurables.

A continuación se explica como implementar cada módulo de forma independiente. De nuevo, este proceso cumple con el flujo de diseño modular:

- Copiar el fichero UCF generado en la fase anterior en el directorio donde se implementará cada módulo.
- Ejecutar *ngdbuild*, *map*, *par*, *bitgen* y *pimcreate* para cada módulo como se describe en el flujo de diseño de Modular Design. El resultado es un módulo emplazado y rutado, y un *bitstream*.
- El proceso *pimcreate* "publica" el diseño rutado para ser empleado en la fase final de ensamblado.
- Usar el FPGA Editor para visualizar el diseño del módulo y comprobar que se cumplen todas las condiciones del fichero de restricciones.

Repetir los pasos anteriores con cada uno de los módulos de diseño.

Ensamblado Final (*Final Assembly*)

En esta fase se realiza el proceso de combinación de los módulos individuales para obtener un diseño completo. El emplazado y rutado llevado a cabo durante la fase de implementación de cada módulo se preserva, manteniendo el rendimiento de cada módulo. El flujo de reconfiguración parcial requiere que los *bitstreams* inicialmente cargados en la FPGA se correspondan con un diseño completo para que toda la lógica global y no reconfigurable esté emplazada y fijada, de modo que solo las partes reconfigurables del diseño cambien durante la reconfiguración parcial.

Los pasos a seguir son los siguientes:

- Copiar el fichero UCF creado en la primera fase para cada uno de los diseños completos.
- Ejecutar *ngdbuild*, *map*, *par* y *bitgen* al igual que en la fase anterior. El resultado será un diseño completo emplazado y rutado en forma de *bitstream*.
- Usar el FPGA Editor para comprobar que el rutado local a cada módulo no se expande hacia el resto de módulos, excepto a través de los buses macro.

Repetir todos los pasos con cada combinación de los módulos fijos y módulos reconfigurables.

5.5. Sistema parcialmente reconfigurable

El sistema reconfigurable propuesto para poner en práctica la metodología de diseño de sistemas parcialmente reconfigurables presenta dos módulos. El primero está compuesto por MicroBlaze y sus periféricos, y está localizado en la mitad izquierda del circuito. El coprocesador es el segundo módulo, localizado a la derecha como se muestra en la figura 5.4. Para el cambio de coprocesador mediante reconfiguración parcial, solo este módulo debe ser reimplementado. Entonces, el nuevo módulo se une al módulo MicroBlaze creado en el paso anterior, para obtener un nuevo *layout* donde sólo el coprocesador ha cambiado.

Para la interconexión entre módulos no es posible emplear el bus macro propuesto por Xilinx. Desafortunadamente las nuevas familias Virtex-4 y Spartan-3 no disponen de buffers triestado (TBUFs). No obstante, el concepto de bus macros puede ser extendido a cualquier componente de la FPGA, siguiendo la idea sugerida en [21]. De este modo, se ha optado por desarrollar buses macro donde los buffers son implementados empleando LUTs. Igual que en el caso del bus macro de Xilinx, cada conexión está compuesta por dos de estos buffers, localizados en slices adyacentes de la FPGA. El *layout* de los nuevos buses macro está únicamente compuesto por dos columnas de slices vecinas que tienen que ser emplazadas entre los módulos: una columna en el área reservada para el módulo fijo, y la otra en el espacio del bloque reconfigurable. Mientras el bus macro de Xilinx es un componente genérico, se ha optado por desarrollar buses macro especialmente adaptados para los coprocesadores de MicroBlaze (figura 5.5): hay un bus macro para la interfaz FSL maestro (datos desde la CPU), otro para la interfaz FSL esclavo (datos desde el coprocesador) y un tercero que maneja el *reset* del módulo reconfigurable. Para crear estos buses macro se ha desarrollado primero un modelo VHDL estructural, que posteriormente fue sintetizado e implementado empleando la herramienta ISE de Xilinx. Es necesario añadir algunas restricciones de emplazado y rutado al diseño para obtener el *layout* esperado. Estos buffers fueron construidos a partir de LUT-1, por lo que es posible incluir dos por slice de la FPGA. Posteriormente a la fase

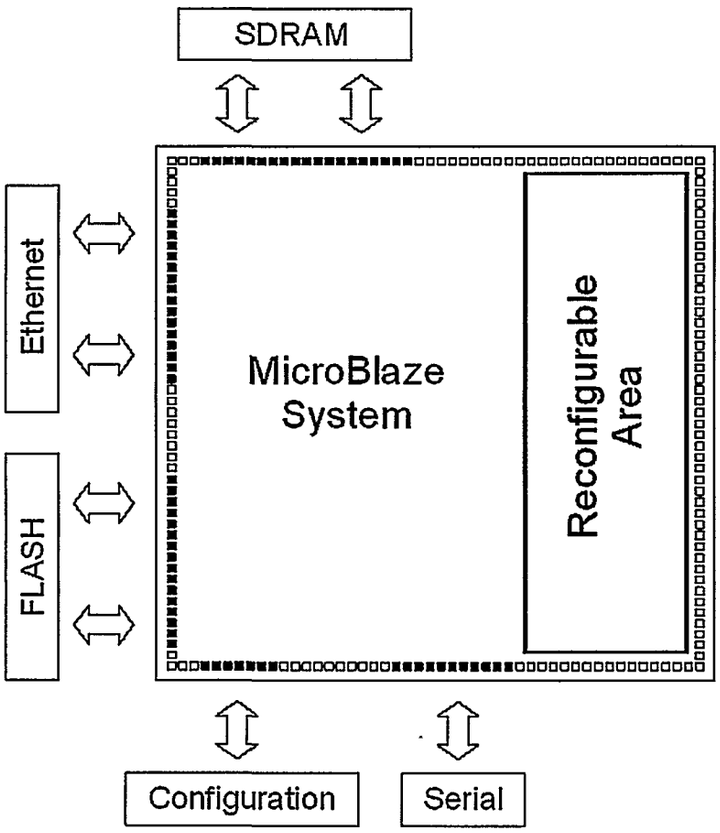


Figura 5.4: Sistema MicroBlaze parcialmente reconfigurable

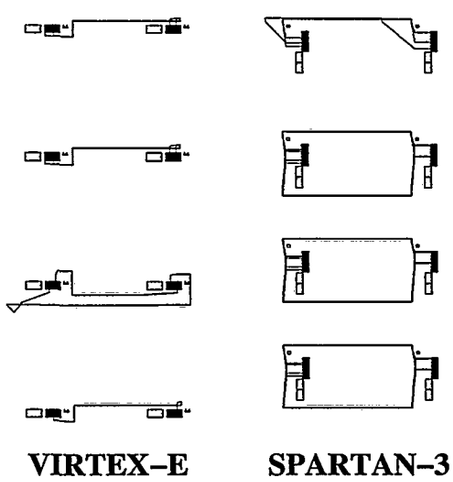


Figura 5.5: *Buses macro* específicas para interfaz FSL (Virtex-E y Spartan-3)

de emplazamiento y rutado, el circuito fue modificado mediante el FPGA Editor para convertirlo a *hard macro*, básicamente añadiendo pines externos.

Siguiendo la metodología presentada en la sección 5.4, se pueden generar los diferentes módulos parciales, como muestran las figuras 5.6 y 5.7[1]. Sin embargo, todos nuestros test con las versiones de ISE 6 fallaron debido a bugs en el flujo de Modular Design, los cuales evitaban que la fase final de ensamblado o la generación del *bitstream* se completaran. En la bibliografía se pueden encontrar trabajos recientes que muestran este mismo problema [205, 206], lo que ha obligado a muchos de ellos a desarrollar circuitos reconfigurables parcialmente empleando el flujo de diseño basado en diferencias (*Partial-Based*) [206], o complicadas metodologías [205].

Trabajando sobre este problema, y analizando los errores encontrados, se llegó a la conclusión de que los problemas se estaban relacionados con el procesador MicroBlaze o alguno de los componentes que forman parte del sistema MicroBlaze. Si además se tiene en cuenta que MicroBlaze es una *netlist*, es imposible realizar cambios sobre el procesador para intentar modificar las partes que dan problemas. Es por ello que se decidió crear una herramienta propia que permitiera combinar los módulos. El resultado fue un *script* de Perl que opera sobre los ficheros XDL de los

[1]El coprocesador 3DES para reconfiguración parcial emplea un core DES sobre el que se realizan 3 iteraciones cumpliendo las especificaciones del algoritmo 3DES. Esto se debe a que la implementación completa del algoritmo (3 cores DES) empleada en las pruebas del capítulo anterior necesitaba más área de la disponible en la parte que corresponde al módulo reconfigurable.

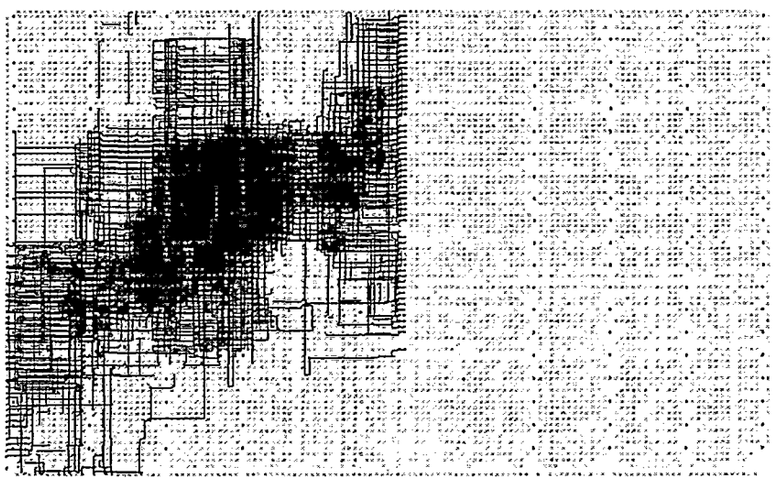


Figura 5.6: Módulo MicroBlaze sobre RC1000PP

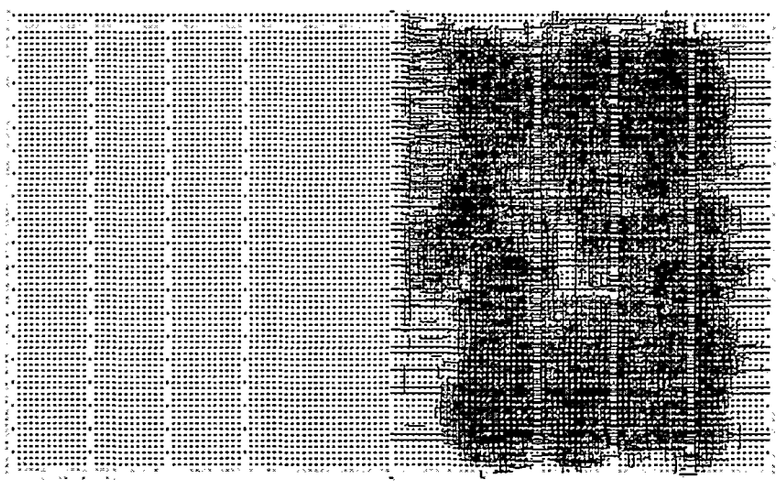


Figura 5.7: Módulo coprocesador 3DES sobre RC1000PP

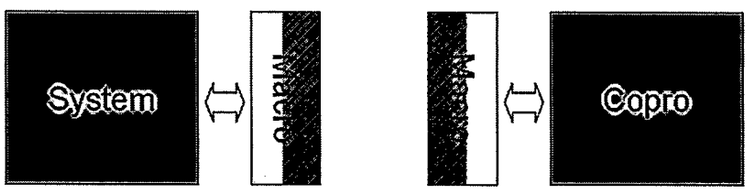


Figura 5.8: Combinación de módulos mediante el *script* de Perl

módulos. XDL [166] es un lenguaje de descripción del *layout* en modo texto, ofrecido por Xilinx como una alternativa abierta a su formato propietario NCD.

Para usar el *script* de Perl, el primer paso es traducir el *layout* de los módulos MicroBlaze y coprocesador desde el formato NCD a XDL. El contenido de cada *layout* se corresponde básicamente con el módulo más los componentes del *top level* (macros más buffers de I/O), los cuales se repiten en todos los ficheros. El algoritmo usado para combinarlos, mostrado en la figura 5.8, consiste en añadir todos los componentes del coprocesador al módulo MicroBlaze, excepto los buffers de I/O y la parte izquierda (parte MicroBlaze) de los macros de interconexión. Del módulo MicroBlaze solo se elimina el lado derecho (parte coprocesador) de la interconexión de los buses macro. Estas restricciones respecto de los buses macro se deben al hecho de que la herramienta de rutado puede intercambiar algunos pines de las LUTs, por lo que es importante mantener para cada módulo el lado del macro con el cual se encuentra directamente conectado. Finalmente cuando se mezclan los dos módulos, toda la información de rutado de reloj se debe eliminar. Este paso se hace necesario porque durante la implementación de los módulos parciales, Modular Design no usa recursos de reloj dedicados, por lo que es mejor re-rutar el reloj de nuevo. Tras la ejecución del *script* de Perl se obtiene un nuevo fichero XDL que incluye a MicroBlaze y al coprocesador. Este fichero debe ser convertido a formato NCD. Sin embargo, el fichero obtenido presenta formato ISE 5 y no puede ser usado directamente por *bitgen* [207], por lo que debe ser convertido al nuevo formato empleando la herramienta de emplazamiento y rutado, par. Durante esta conversión se aprovecha para realizar también el re-rutado del reloj.

Usando esta metodología se obtienen dos ficheros NCD que presentan el mismo *layout* en el lado de MicroBlaze, pero con diferente coprocesador en el área reconfigurable, por lo que es inmediato el obtener los *bitstream* parciales con los cambios de un coprocesador a otro (figura 5.9). El resultado se puede ver en las figuras 5.10 y 5.11. Es importante señalar que si tenemos n coprocesadores, necesitamos en principio *n*(n- 1) bitstream* parciales para cubrir todos los posibles cambios de un coprocesador por otro, ya que cada configuración parcial solo incluye los bits que

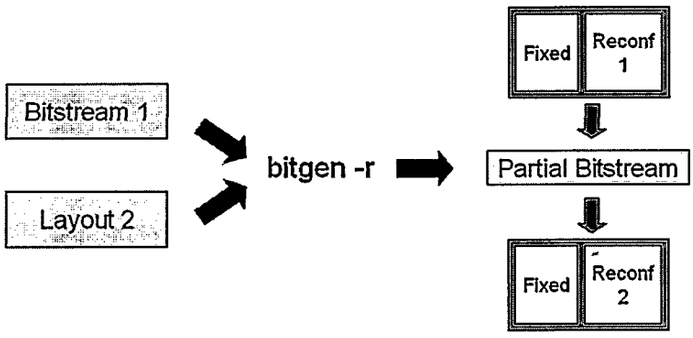


Figura 5.9: Generación de *bitstreams* parcial

hay es necesario modificar sobre el coprocesador actual para conseguir el nuevo. Sin embargo, se pueden emplear las opciones *"-g PartialMask..."* de *bitgen* para forzar que el *bitstream* parcial incluya todos los *frames* del área reconfigurable, no solo los que han cambiado de un diseño a otro. De este modo, solo se necesitan *n bitstreams*, aunque es necesario asumir el coste de manejar un fichero de configuración mayor. Esta solución presenta la ventaja de que al tener todos los *bitstreams* el mismo tamaño, el tiempo de reconfiguración es el mismo para todos los cambios de coprocesador.

Finalmente, las herramientas empleadas son Xilinx EDK 6.3i SP2 e ISE 6.3i SP3. El sistema MicroBlaze fue sintetizado con XST, mientras que Synplify 7.7.1 fue usado para los coprocesadores.

5.6. Auto-reconfiguración en tiempo de ejecución

Para comprobar el correcto funcionamiento de la metodología presentada en la sección anterior, se realizaron un conjunto de pruebas sobre la plataforma RC1000PP, en las cuales se reconfiguraba cada uno de los coprocesadores, y se comprobaba su correcto funcionamiento mediante la comparación del resultado con su respectiva versión en software. Una vez demostrado que la reconfiguración parcial funciona correctamente sobre esta plataforma, se repitieron las mismas pruebas sobre la plataforma auto-reconfigurable propuesta, el sistema construido sobre la placa de desarrollo con Spartan-3 de Avnet, pero en este caso la reconfiguración del coprocesador se realizó desde el procesador MicroBlaze.

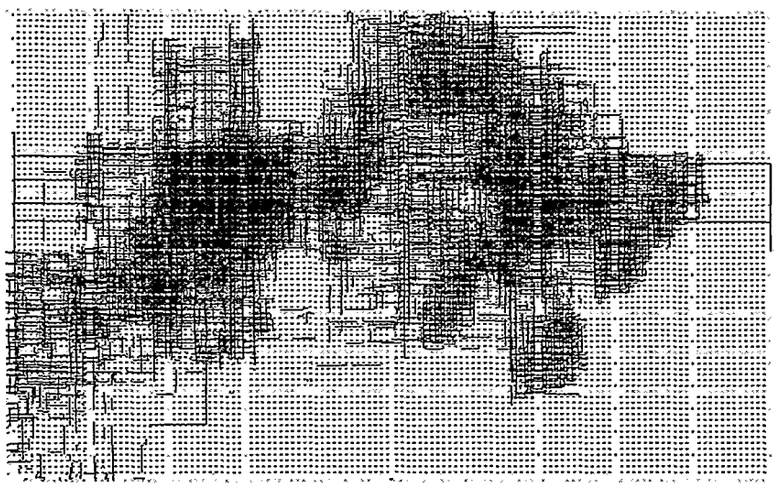


Figura 5.10: Sistema MicroBlaze con coprocesador IDEA sobre RC1000PP

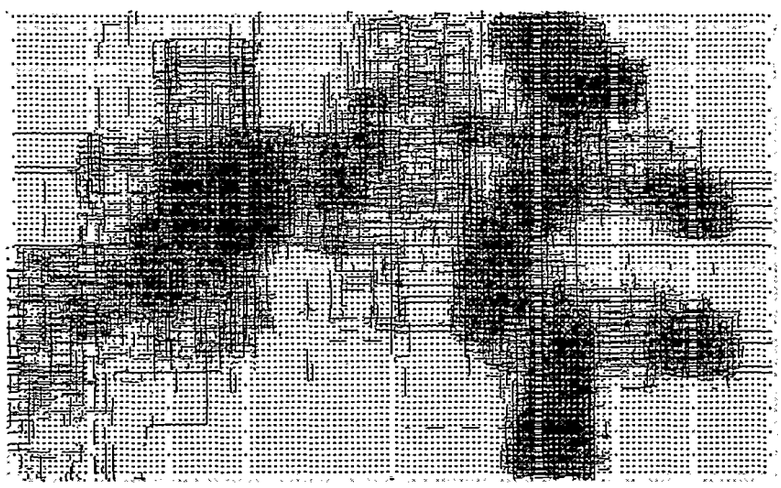


Figura 5.11: Sistema MicroBlaze con coprocesador MD5 sobre RC1000PP

La placa de desarrollo con Spartan-3 de Avnet dispone de una memoria Flash de configuración, por lo que se pueden seguir las indicaciones sobre el conexionado que se han comentado en la sección 5.3 (figura 5.2). El sistema MicroBlaze inicial se carga a través de la memoria de configuración al alimentar la plataforma, y posteriormente los cambios de coprocesador son realizados de forma autónoma por el sistema empleando las conexiones anteriores. Para las pruebas se han empleado tres de los algoritmos de cifrado: AES, RC4 e IDEA. Los diferentes *bitstream* parciales son generados en formato binario y almacenados en un servidor TFTP. A parte de la interfaz con los coprocesadores, el sistema MicroBlaze incluye la interfaz con la memoria SDRAM de 64 MB, un Ethernet MAC, tres GPIOs (uno para los LEDs y *switches*, otro para la auto-reconfiguración, y un tercero para el *reset* del coprocesador), un *timer* y una UART como consola. El módulo de debug no se ha añadido porque el acceso al JTAG se encuentra localizado en el área reservada al coprocesador. Es importante recordar que el área ocupada por el coprocesador reconfigurable está fijada por el pinout de la FPGA. La figura 5.12 muestra el resultado de conectar el sistema MicroBlaze al coprocesador AES.

Para el proceso de auto-reconfiguración de la FPGA, los *bitstream* parciales son enviados al GPIO que está conectado al puerto de configuración. La secuencia para la reprogramación es muy fácil, siendo solo necesario escribir un byte de datos y enviar un pulso de reloj. El código ejecutado en MicroBlaze permite bajar todos los *bitstream* del servidor TFTP, y almacenarlos en el sistema de ficheros creado en memoria SDRAM usando la librería de Xilinx MFS [208]. Como se explicó en la sección anterior, los *bitstream* parciales presentan el mismo tamaño por lo que el tiempo de reconfiguración es el mismo para los tres coprocesadores. Este tiempo se ha medido en *166 ms* con la caché habilitada. Este es un tiempo bastante alto, y se debe en parte a que el hardware empleado, un GPIO, en vez de un periférico dedicado. Sin embargo, el propósito de este trabajo es mostrar que la metodología es aplicable, sin optimizar el tiempo necesario para reconfigurar los coprocesadores. No obstante, este número puede considerarse como un máximo del tiempo de reconfiguración. En condiciones ideales, si se considera que los *bitstream* parciales tienen un tamaño de 335 KB y que la velocidad máxima de reconfiguración es de 66 MB/s, el cambio de coprocesador se podría realizar en solo 5,2 ms.

La FPGA empleada es una XC3S2000FG676-4C ejecutándose a 65 MHz, generados internamente a partir de un oscilador de 100 MHz. La tabla 5.2 muestra los tiempos de ejecución de los diferentes algoritmos acelerados por hardware frente a la versión software estándar[2]. Incluso considerando que la arquitectura del coprocesador no ha sido diseñada para el mejor rendimiento,

[2]Es importante destacar que los resultados obtenidos en este test no son comparables a los presentados en el capítulo anterior. Esto se debe por un lado a que solo se trata del proceso de cifrado, sin realizar el descifrado posteriormente, lo cual permite optimizar mejor el manejo del coprocesador e incluso la implementación en software de los algoritmos. Estas diferencias han permitido mejorar considerablemente el rendimiento como se indicaba en las conclusiones de ese mismo capítulo. Por otro lado, tanto la configuración del procesador como la memoria externa empleada son diferentes.

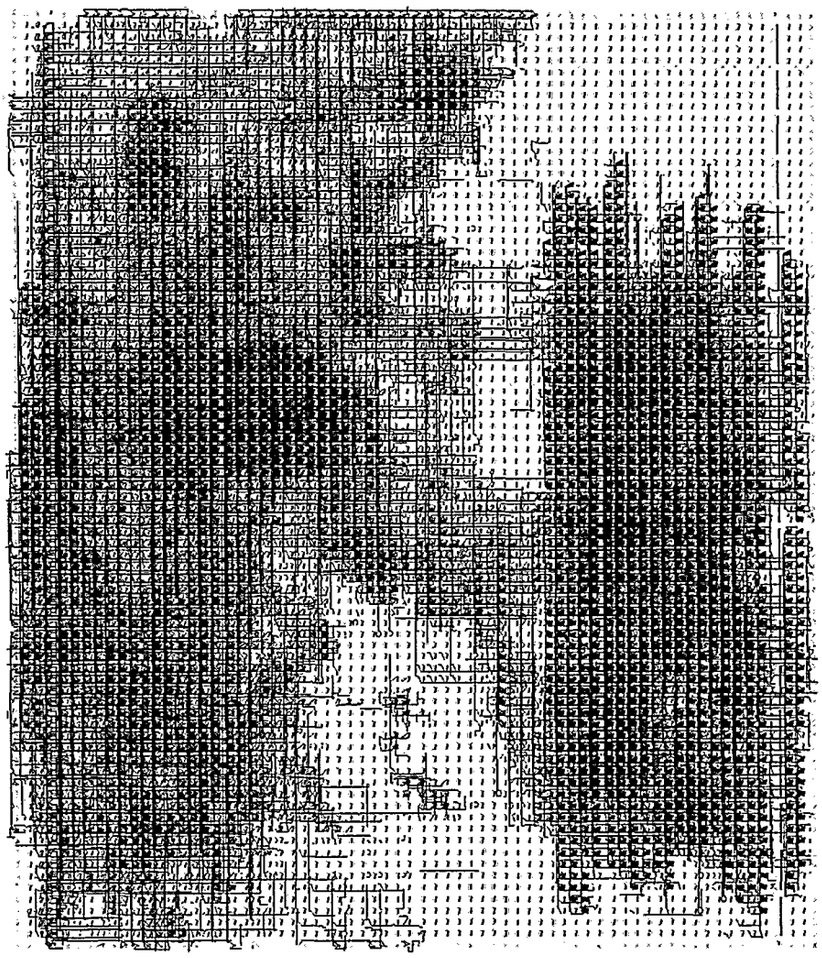


Figura 5.12: Sistema MicroBlaze auto-reconfigurable con coprocesador AES sobre Spartan-3

		Tiempos [s] de cifrado		
		IDEA	AES	RC4
Caches ON	SW	9,776	18,863	4,531
	Coprocesador	1,443	0,451	0,492
		Aceleración		
		6,8	41,8	9,2
Caches OFF	SW	104,739	86,014	33,858
	Coprocesador	1,519	1,519	1,519
		Aceleración		
		69,0	56,6	22,3

Tabla 5.2: Tiempos de cifrado para 4 MB de datos aleatorios junto con la aceleración HW para cada coprocesador

	Recursos de implementación				
	FPGA Slices	BRAM	Multiplicadores 18x18	LUTs	FFs / Latches
Sistema MicroBlaze	4198	20	3	4809	3618
Coprocesador AES	4260	0	0	4622	1674
Coprocesador RC4	575	1	0	426	533
Coprocesador IDEA	1497	0	6	725	1165

Tabla 5.3: Recursos FPGA empleados por el sistema MicroBlaze y los coprocesadores

los resultados son destacables. Con la caché habilitada, el rendimiento obtenido por los coprocesadores es al menos 6,8 veces superior a la solución software (IDEA), alcanzando una aceleración superior al 41 para el mejor caso (AES). Los números son mucho mejores cuando la caché no se emplea, llegando a una mejora hardware 69 veces más rápida que el software para IDEA. Sin caché, los tiempos en aceleración son los mismos para los tres algoritmos, al ser el cuello de botella el acceso a memoria y no la operación del coprocesador. Naturalmente, esta metodología también reduce el área empleada. El espacio ocupado por MicroBlaze y el coprocesador de mayor tamaño (AES) es de 8458 slices (tabla 5.3), pero si los tres coprocesadores fueran simultáneamente implementados, el área llegaría a los 10530 slices, un 25 % más. Este número podría ser algo mayor porque no se han incluido en la estimación los multiplexores necesarios para seleccionar el coprocesador a emplear.

5.7. Secure shell auto-reconfigurable

Con el objetivo de hacer un uso práctico del sistema auto-reconfigurable desarrollado, se propone el diseño de una versión acelerada por hardware de la aplicación Secure Shell (SSH) [172, 209, 210, 211, 212, 213]. SSH ha emergido como sustituta de las aplicaciones *telnet* y *ftp*, ambas ampliamente usadas en el mundo UNIX, incluyendo mecanismos de que garantizan la seguridad cuando se requiere acceder a máquinas remotas, empleando criptografía de clave pública para establecer una comunicación autentificada y cifrada sobre canales inseguros.

SSH es una aplicación que principalmente se ejecuta sobre sistemas operativos, principalmente del tipo UNIX y Linux, ya que requiere de un amplio soporte para el manejo de conexiones de red, ejecución de varios procesos (multitarea), temporización y otros mecanismos relacionados. Por este motivo ha sido necesario ejecutar sobre el sistema MicroBlaze auto-reconfigurable desarrollado, un sistema operativo para esta plataforma, uCLinux [48], recientemente portado a MicroBlaze [49],

5.7.1. El protocolo SSH

SSH permite la ejecución de sesiones remotas y transferencia de ficheros de forma segura sobre Internet y otras redes inseguras. Los algoritmos de criptografía se emplean para la autentificación de los dos puntos de la conexión, cifrar todos los datos transmitidos y proteger la integridad de los mismos. SSH además permite el redireccionamiento de conexiones X11 desde máquinas remotas de forma segura, y puede ser configurado para realizar el redireccionamiento de cualquier puerto TCP/IP. También puede ser empleado para la transferencia segura de ficheros.

Existen actualmente dos protocolos SSH: SSH1 y SSH2. El protocolo SSH2 se corresponde con una completa reimplementación del protocolo SSH1: cifra diferentes partes del paquete, usa diferentes algoritmos de cifrado, etc. Generalmente, el protocolo SSH2 se considera más seguro, evitando ciertas vulnerabilidades conocidas en la implementación de SSH1.

Hay tres partes diferenciadas dentro del protocolo SSH [172, 209]: la negociación del algoritmo, la autentificación y el cifrado de los datos. La negociación del algoritmo es la responsable principal de determinar los algoritmos de cifrado, algoritmos de compresión y los métodos de autentificación soportados que serán empleados entre el cliente y el servidor. La autentificación [213] se divide en dos procesos: intercambio de la clave (capa de transporte) y autentificación de usuario (capa de autentificación de usuario). El intercambio de claves se basa en Diffie-Hellman [127, 209] y tiene un doble propósito. En primer lugar, se encarga de autentificar el servidor al cliente. El cliente verifica la clave pública del servidor, verifica la firma del servidor recibida y entonces continua con la autentificación del usuario. Los métodos de autentificación de usuario

soportados y que forman parte del protocolo SSH2 incluyen *password*, clave pública, PAN y Kerberos [210, 211, 212, 213]. En segundo lugar, establece una clave compartida que se emplea como clave de sesión para cifrar todos los datos transferidos entre las dos máquinas, empleando un algoritmo de cifrado de clave simétrica. Estos algoritmos pueden emplear claves independientes para cada dirección de envío de datos. Todos los datos, incluida la longitud del paquete, son cifrados (excepto la MAC). Otros algoritmos negociados se emplean para la protección de la integridad de los datos.

5.7.2. Métodos de cifrado empleados en SSH

SSH intenta proveer un fuerte mecanismo de seguridad de un modo transparente al usuario. Esta seguridad se basa en métodos de criptografía. En particular, SSH2 es más seguro que SSH1 porque emplea diferentes algoritmos para el cifrado y autentificación. SSH1 permite cuatro algoritmos de cifrado [126, 127]: DES, 3DES, IDEA y Blowfish. SSH2 elimina el soporte para DES al tratarse de un algoritmo roto, e IDEA por problemas de patentes, pero añade tres nuevos algoritmos: AES (Rinjdael), Twofish y CAST. SSH1 emplea el algoritmo de autentificación RSA, mientras que SSH2 lo ha cambiado por el Digital Signature Algorithm (DSA). Estos cambios están relacionados con problemas de propiedad intelectual respecto del uso de IDEA y RSA, e incrementa el nivel de seguridad en SSH2 empleando algoritmos más fuertes. Además, los algoritmos hash MD5 (HMAC-MD5) o SHA-1 (HMAC-SHA) son empleados para la protección de la integridad de los datos, en vez del clásico CRC empleado en SSH1.

5.7.3. OpenSSH y OpenSSL

SSH1 esta disponible de forma gratuita para los usuarios a pesar de hacer uso de ciertas patentes. Con la llegada de SSH2, el creador restringió la licencia y desde ese momento ya no se volvió a ofrecer para su descarga gratuita. La comunidad *open source* rechazó este cambio en la licencia de SSH y desarrolló su propia alternativa, OpenSSH [214]. OpenSSH es una versión libre de la *suite* de herramientas de conexión de red que hace uso del protocolo SSH, y que ofrece la misma funcionalidad de SSH2 sin conflictos con ninguna restricción de propiedad intelectual. La versión 2 de OpenSSH y posteriores soportan ambos protocolos de SSH, el viejo protocolo SSH1 y el nuevo SSH2, lo que permite a OpenSSH crear conexiones hacia y desde clientes que soporten uno o ambos protocolos. OpenSSH emplea OpenSSL [215], un conjunto de herramientas criptográficas que implementan los protocolos de red Secure Sockets Layer (SSL v2/v3) y Transport Layer Security (TLS v1) [127], además de otros estándares de criptografía que son requeridos por éstos. OpenSSL incluye varias funciones de criptografía que pueden ser empleadas para:

- Creación de claves RSA, DH y DSA.
- Creación de certificados X.509, CSRs y CRLs.
- Cálculo de resúmenes de mensajes (Message Digests).
- Cifrado y descifrado con algoritmos de cifrado.
- Clientes SSL/TLS y test de servidores.
- Manejo de mails firmados o cifrados.

5.7.4. uCLinux en MicroBlaze

SSH es una aplicación estándar en la mayoría de los sistemas operativos del tipo Unix o Linux, aunque también se encuentra disponible para otros entornos. La necesidad de ejecutar un sistema operativo en un sistema embebido permite a éste disponer disponer de capacidades complejas como multitarea (procesos y threads), soporte para un amplio número de protocolos de red, etc. SSH requiere de estas capacidades, por lo que para llevar a cabo la aplicación propuesta, se hace necesario ejecutar un sistema operativo sobre el sistema auto-reconfigurable desarrollado. Con este objetivo se ha seleccionado el sistema operativo uCLinux [48], el cual ha sido portado recientemente para su uso en el procesador MicroBlaze [49].

uCLinux es una adaptación de Linux para microprocesadores que carecen de unidad de gestión de memoria (MMU). La principal implicación de la ausencia de MMU es que no existe protección de memoria, es decir, que los procesos pueden escribir en cualquier parte de la memoria, y que no existe memoria virtual (swapping, etc.). Sin embargo, para la mayoría de las aplicaciones la única limitación es que la llamada al sistema *fork()* no se encuentra disponible, y se debe emplear la llamada *vfork()*. Existe varios *ports* de uCLinux para procesadores sin MMU incluyendo el Motorola ColdFire o el NEC v850. La actual distribución se basa en el kernel de Linux 2.4.20, pero actualmente existe un trabajo activo para portar los kernels 2.5 y 2.6.

5.7.5. Implementación de SSH

La versión acelerada por hardware de SSH desarrollada, se basa en la adaptación de SSH de OpenBSD, OpenSSH, que se encuentra disponible en uCLinux. A este adaptación se le ha añadido la funcionalidad necesaria para el manejo del proceso de reconfiguración de los coprocesadores, y de los diferentes coprocesadores criptográficos soportados. Para simplificar la tarea se han seleccionado solo dos de los algoritmos de cifrado simétrico empleados por SSH2, y que son ejecutados

	Recursos de implementación				
	FPGA Slices	LUTs	FF / Latches	18x18 Mult	BRAM
Sistema MicroBlaze	4198	4809	3618	3	20
Coprocesador AES	4326	4632	1802	0	0
Coprocesador 3DES	5424	4493	5825	0	0

Tabla 5.4: Uso de lógica - MicroBlaze y coprocesadores

en hardware: el algoritmo por defecto AES128-CBC y el opcional 3DES-CBC. El resto de algoritmos se ejecutarán en software. En cualquier caso, para el resto de los algoritmos solo hay que repetir el mismo procedimiento de cambios, incluyendo los algoritmos de hash empleados en la integridad de datos y los algoritmos de clave asimétrica.

La plataforma de referencia

El sistema MicroBlaze ha sido implementado en la plataforma auto-reconfigurable sobre la placa de Avnet presentada en la sección anterior. Este sistema incluye, además de la interfaz con el coprocesador, el sistema MicroBlaze necesario para poder ejecutar el sistema operativo uCLinux eficientemente: una cache de datos de 8 KB, una cache de instrucciones de 8 KB, un controlador de memoria SDRAM, el Ethernet MAC, un *timer*, un controlador de interrupciones, una UART como consola de E/S y tres GPIOs: uno para manejar los leds y *switches* disponibles en la plataforma, otro para la auto-reconfiguración y el tercero como *reset* del coprocesador. El último GPIO es necesario para resetear la lógica que se inserta en cada reconfiguración, ya que la señal de global *set/reset* (GSR) no se puede emplear para evitar inicializar el procesador MicroBlaze que ejecuta la aplicación. Esta solución añade una señal de *reset* local para la lógica reconfigurable, la cual es controlada por el procesador. Esta señal se activa al iniciar el proceso de reconfiguración, y se mantiene activa hasta que finaliza, de modo que se asegura que la nueva lógica añadida se inicia en un estado conocido. El sistema final fue implementado en una FPGA XC3S2000FG676-4C ejecutándose a 65 MHz. La tabla 5.4 muestra los recursos empleados en los diferentes diseños: el sistema MicroBlaze y los diferentes coprocesadores de cifrado. El *layout* del sistema MicroBlaze se muestra en la figura 5.13.

Cada coprocesador ha sido implementado con la misma interfaz que el resto de coprocesadores desarrollados para MicroBlaze, pero ha sido necesario reimplementar los coprocesadores para permitir el uso de dos claves por algoritmo porque SSH2 emplea diferentes claves para cada operación: un bus FSL maestro para recibir la clave de cifrado y la clave de descifrado, y dos pares de buses FSL maestro/esclavo para el envío/recepción de datos. Para seleccionar el modo de ope-

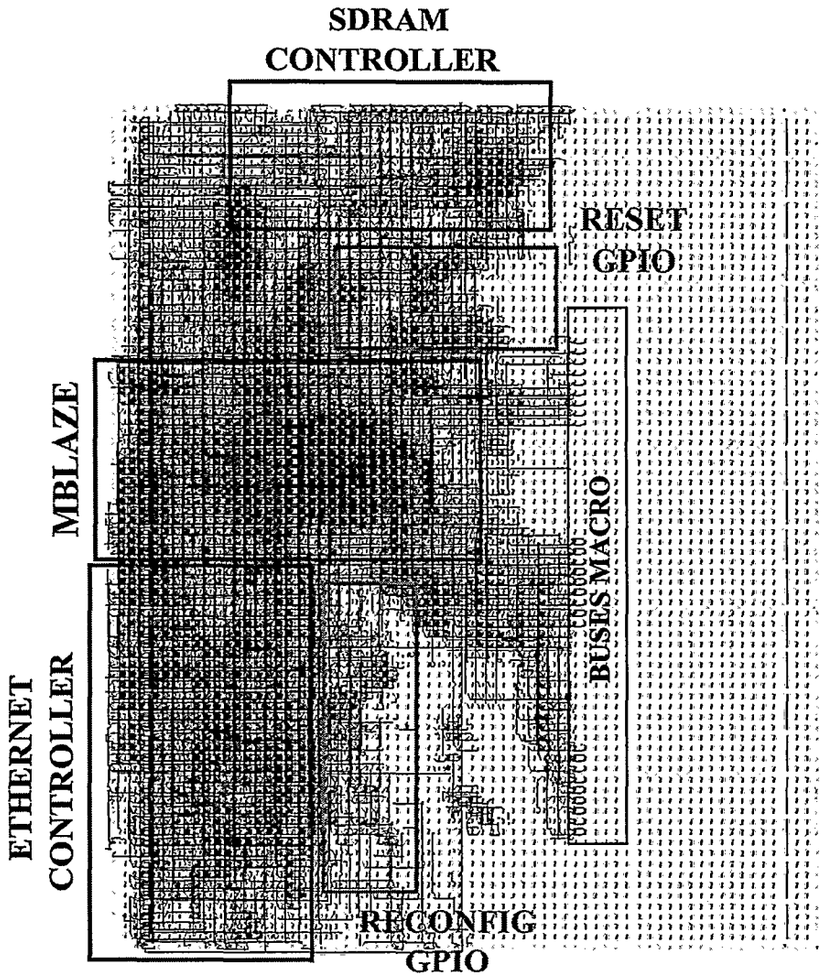


Figura 5.13: Sistema MicroBlaze-uCLinux auto-reconfigurable (sin coprocesador) sobre Spartan-3

ración (cifrado o descifrado) se han empleado el bit de control del bus FSL. Todos los buses FSL son de 32 bits de ancho y disponen de una FIFO de 16 palabras. Los diferentes algoritmos implementados en hardware emplean el tamaño de bloque/clave definido en el estándar. En particular, AES fue implementado como AES128, empleando una clave de 128 bits para cifrar un bloque de 128 bits, y 3DES cifra un bloque de 64 bits con una clave de 192 bits.

SSH optimizado con hardware específico

Para adaptar la implementación software de OpenSSH al uso de los coprocesadores de cifrado es necesario modificar la librería OpenSSL para cambiar la implementación software de los algoritmos seleccionados para su equivalente para el manejo de los coprocesadores hardware. Después de esto, la aplicación SSH ha sido cambiada para permitir la reconfiguración del coprocesador una vez que el algoritmo ha sido seleccionado.

La librería OpenSSL adaptada para uso de hardware específico Los cambios están relacionados con la librería "crypto". La implementación de AES y 3DES incluye ahora funciones ejecutándose en hardware para el cifrado, descifrado y el envío de claves empleando la interfaz FSL. Estas funciones son insertadas en las rutinas adecuadas para sustituir el proceso software de cifrado/descifrado. Por ejemplo, las funciones del coprocesador AES sustituyen a las funciones originales *AES_encrypt()* y *AES_decrypt()* en la función *AES_cbc_encrypt()* del fichero *aes_cbc.c*. Las funciones hardware de manejo de las claves son insertadas en la función *AES_set_encrypt_key()* dentro del fichero *aes_core.c*. Los algoritmos 4, 5, 6 muestran las funciones del coprocesador AES (las macros GETU32 y PUTU32 se emplean para la conversión de *char array* a *unsigned long* y viceversa). Idénticas funciones y cambios en la librería OpenSSL se han realizado para el algoritmo 3DES.

OpenSSH con reconfiguración dinámica La implementación de OpenSSH incluye un fichero llamado *cipher.c* que resuelve la interfaz con la librería OpenSSL. En este fichero se encuentra la función *cipher_init()*, donde se inicializan los algoritmos que serán empleados durante la comunicación segura. En esta función se ha añadido el código relacionado con la reconfiguración parcial. Este cambio permite leer el *bitstream* parcial del sistema de ficheros y enviarlo a través del GPIO que está conectado al puerto de configuración. La secuencia de pasos para reconfigurar la FPGA es simple, solo es necesario escribir un byte de datos y enviar un pulso de reloj.

El algoritmo 7 muestra la modificación realizada a la función *cipher_init()* para reconfigurar la FPGA con el coprocesador seleccionado. La primera condición comprueba que el algoritmo ha sido seleccionado para el cifrado, de modo que se evite reconfigurar el coprocesador dos veces.

Algoritmo 4 Funciones de envío de clave del coprocesador AES

```
void aes_copro_sendkey_encrypt(unsigned char *key)
{
   microblaze_bwrite_datafsl(GETU32(key),2);
   microblaze_bwrite_datafsl(GETU32(key+4),2);
   microblaze_bwrite_datafsl(GETU32(key+8),2);
   microblaze_bwrite_datafsl(GETU32(key+12),2);
}

void aes_copro_sendkey_decrypt(unsigned char *key)
{
   microblaze_bwrite_cntlfsl(GETU32(key),2);
   microblaze_bwrite_cntlfsl(GETU32(key+4),2);
   microblaze_bwrite_cntlfsl(GETU32(key+8),2);
   microblaze_bwrite_cntlfsl(GETU32(key+12),2);
}
```

Algoritmo 5 Función de cifrado del coprocesador AES

```
void aes_copro_crypt(const unsigned char *src, unsigned char *dst)
{
   u32 datain[4], dataout[4];

   datain[0] = GETU32(src); datain[1] = GETU32(src + 4);
   datain[2] = GETU32(src + 8); datain[3] = GETU32(src + 12);

   // Write data to encrypt
   microblaze_bwrite_datafsl(datain[0],0);
   microblaze_bwrite_datafsl(datain[1],1);
   microblaze_bwrite_datafsl(datain[2],0);
   microblaze_bwrite_datafsl(datain[3],1);

   // Read data encrypted
   microblaze_bread_datafsl(dataout[0],0);
   microblaze_bread_datafsl(dataout[1],1);
   microblaze_bread_datafsl(dataout[2],0);
   microblaze_bread_datafsl(dataout[3],1);

   PUTU32(dst, dataout[0]); PUTU32(dst + 4, dataout[1]);
   PUTU32(dst + 8, dataout[2]); PUTU32(dst + 12, dataout[3]);
}
```

Algoritmo 6 Función de descifrado del coprocesador AES

```
void aes_copro_decrypt(const unsigned char *src, unsigned char *dst)
{
   u32 datain[4], dataout[4];

   datain[0] = GETU32(src); datain[1] = GETU32(src + 4);
   datain[2] = GETU32(src + 8); datain[3] = GETU32(src + 12);

   // Write data to decrypt
   microblaze_bwrite_cntlfsl(datain[0],0);
   microblaze_bwrite_cntlfsl(datain[1],1);
   microblaze_bwrite_cntlfsl(datain[2],0);
   microblaze_bwrite_cntlfsl(datain[3],1);

   // Read data decrypted
   microblaze_bread_cntlfsl(dataout[0],0);
   microblaze_bread_cntlfsl(dataout[1],1);
   microblaze_bread_cntlfsl(dataout[2],0);
   microblaze_bread_cntlfsl(dataout[3],1);

   PUTU32(dst, dataout[0]); PUTU32(dst + 4, dataout[1]);
   PUTU32(dst + 8, dataout[2]); PUTU32(dst + 12, dataout[3]);
}
```

Algoritmo 7 Reconfiguración parcial en OpenSSH

```
if(encrypt && (cipher->number != 0)) {
   assert_copro_reset(); // assert coprocessor reset
   printf("Encryption Algorithm: %s\n",cipher->name);
   if(!strcmp(cipher->name,ciphers[4].name)) {
      printf("Reconfiguring 3DES coprocessor... ");
      send_cfg_file("partial_3des_2keys.bit");
   }
   else if(!strcmp(cipher->name,ciphers[8].name)) {
      printf("Reconfiguring AES coprocessor... ");
      send_cfg_file("partial_aes_2keys.bit");
   }
   release_copro_reset(); // release coprocessor reset
}
else if(!encrypt && (cipher->number != 0))
   printf("Decryption Algorithm: %s\n",cipher->name);
```

Usualmente los algoritmos de cifrado y descifrado en SSH2 son el mismo, pero es posible emplear diferentes algoritmos para cada operación. Después de esto, el código identifica el algoritmo y reconfigura la FPGA con el coprocesador adecuado.

Ventajas de la reconfiguración parcial en SSH Aparte de la posibilidad de acelerar los diferentes algoritmos, la reconfiguración parcial del coprocesador reduce el área necesaria para implementar todos los algoritmos empleados en los actuales protocolos de seguridad. Además, la habilidad de reconfiguración de las FPGAs contribuye a otras ventajas potenciales [44] como la agilidad del algoritmo, la actualización del algoritmo y su modificación. En este diseño, un PC comparte un directorio a través del protocolo NFS. En este directorio se encuentran disponibles los diferentes *bitstreams* parciales, por lo que el sistema MicroBlaze puede "montar" ese directorio y obtener los diferentes coprocesadores hardware desde esa ubicación, empleando una idea similar a [192]. Si se realiza una nueva implementación de un coprocesador o se realiza una modificación a cualquier de ellos, el sistema MicroBlaze puede emplearlo de forma transparente siempre que el *bitstream* de configuración con las nuevas mejoras esté disponible con el mismo nombre y el coprocesador mantenga la misma interfaz de comunicación con el procesador.

5.7.6. Test de transferencia de ficheros

Una vez implementado SSH para que soporte la reconfiguración parcial de los coprocesadores realizando las modificaciones indicadas, es posible conectar con un servidor SSH y abrir una shell remota. Sin embargo, este test solo permite comprobar que la funcionalidad de la aplicación se mantiene correcta a pesar de los cambios, lo que constituye el objetivo de este capítulo. No obstante, para evaluar la mejora debida al uso de coprocesadores de cifrado, se han medido las diferentes velocidades de transferencia al copiar un conjunto de ficheros a través de un canal seguro. La aplicación Secure Copy (scp), ha sido empleada para este experimento. Secure Copy permite transferir de forma segura ficheros entre una computadora local y un host remoto o entre dos host remotos empleando el protocolo SSH. Particularmente, esta aplicación es adecuada para este test porque durante la transferencia se muestra el tiempo empleado y la velocidad de transferencia para cada fichero enviado. Este test se ha realizado conectando el sistema MicroBlaze a una LAN con muy pocas máquinas, incluyendo el servidor SSH, para reducir el efecto del tráfico.

La tabla 5.5 muestra los resultados de las diferentes pruebas realizadas. Estos resultados determinan, por un lado, que el algoritmo AES es más eficiente que el algoritmo 3DES, y por otro, que cuando se emplean los coprocesadores hardware ambos algoritmos obtienen el mismo rendimiento y la mejora es dos veces superior en el mejor de los casos (3DES). Si se compara el rendimiento obtenido en la transferencia segura de los ficheros, se puede comprobar que está muy por debajo

Protocolo	Algoritmo	Fichero 1 335KB	Fichero 2 2410KB	Fichero 3 8266KB
SSH Software	AES	27,8	48,1	52,7
	3DES	16,7	26,6	30,3
SSH Hardware	AES	33,2	60,4	66,4
	3DES	32,0	60,3	67,7
FTP	Ninguno	877,3	696,1	683,2

Tabla 5.5: Rendimiento de la transferencia de ficheros en KBytes/seg.

del protocolo tradicional de envío de ficheros (FTP). La medida del rendimiento obtenido mediante el uso de FTP también nos permite determinar que el problema del bajo rendimiento en la transferencia de ficheros no se debe al canal de comunicación.

La principal razón de la diferencia entre FTP y SSH radica en la enorme complejidad del protocolo SSH con respecto al protocolo FTP y además, en el caso de la aceleración de los algoritmos en hardware, el poco rendimiento que se obtiene se debe a la ejecución software del algoritmo de hash empleado en la integridad de los datos, MD5 en este caso. Es por ello que hay que tener en cuenta que en esta implementación solo se ha acelerado el algoritmo de cifrado de datos, pero no el que comprueba la integridad de datos, por lo que para cada paquete de datos a procesar toda la mejora conseguida mediante el uso del coprocesador se pierde al ejecutar en software la integridad de ese paquete de datos. Aún así, se obtiene una interesante mejora.

Una posible solución para mejorar el rendimiento pasa por continuar con la aceleración de todas las tareas de cifrado que envuelven al protocolo SSH. Por ejemplo, emplear otro coprocesador reconfigurable para acelerar los algoritmos de integridad de datos, lo cual es adecuado ya que también son seleccionados de forma dinámica. Otra mejora interesante, aunque menos relevante, sería que los algoritmos de clave pública se resolvieran mediante otro coprocesador hardware, el cual se mantendría fijo e incluido en el sistema MicroBlaze, porque al realizarse la autentificación y negociación de la clave solo al principio de cada sesión, se trata de una parte no crítica del protocolo. En resumen, lo ideal es reemplazar todos los algoritmos de cifrado empleados en el protocolo SSH por coprocesadores hardware y optimizar el código de uso de estos coprocesadores, de modo que sea posible obtener un rendimiento similar al protocolo FTP. Finalmente, un paso más allá sería que el manejo de los coprocesadores fuera incluido dentro del propio sistema operativo [193]. En cualquier caso, se ha demostrado la viabilidad y funcionalidad de la reconfiguración parcial dinámica de coprocesadores en una aplicación real.

5.8. Conclusiones

La plataformas CSoC que incluyen procesadores y lógica reconfigurable son soluciones adecuadas para acelerar la ejecución de aplicaciones, donde las partes críticas son implementadas como coprocesadores hardware específicos. Las aplicaciones de criptografía son un buen ejemplo de aplicaciones que se benefician del uso de aceleradores hardware, pero el problema es el área necesaria para los coprocesadores cuando varios algoritmos de cifrado son soportados. En este capítulo se ha demostrado que la auto-reconfiguración puede ser empleada de manera útil para cambiar el coprocesador empleado en las aplicaciones de seguridad como SSH. Mientras que la aceleración hardware por sí misma provee la mejora del rendimiento, la habilidad de reconfigurar el hardware permite reducir la cantidad de área empleada y la posibilidad de actualizar los coprocesadores en tiempo de ejecución.

Del mismo modo, en este capítulo se ha comprobado que, a pesar de que solo las FPGAs más modernas presentan interfaces de configuración especialmente diseñadas para auto-reconfiguración, es posible emplear dispositivos de bajo coste. Con una simple adaptación hardware, añadiendo unas cuantas conexiones y ningún componente externo, es posible que FPGAs de bajo coste como la Spartan-3 de Xilinx se puedan emplear como plataformas auto-reconfigurables.

En resumen, los sistemas embebidos basados en FPGA que hacen uso de la Auto-reconfiguración Parcial Dinámica presentan una mayor flexibilidad porque permiten cambiar las aplicaciones que se están ejecutando en tiempo de ejecución, incluso si se emplea un coprocesador dedicado. Además, permiten emplear dispositivos más pequeños ya que los coprocesadores que no se utilizan pueden ser eliminados, dejando espacio para otros.

Capítulo 6

Conclusions and Future work

6.1. Conclusions

In the last years, reconfigurable hardware manufacturers have efficiently taken advantage of the exponential improvements in the integration level of integrated circuits. FPGAs have evolved from being mere support logic for other chips to being able to integrate true complete systems with capacities of several millions of logic gates. Present devices also contain memory blocks, arithmetic units and even processors physically embedded in the silicon. All these characteristics make possible to implement a system with one or more processors, boot-up memories and peripherals in a FPGA. Moreover, there will still be enough free area to include custom coprocessors to speedup the software tasks.

In the same way that FPGA capacity has increased, the cost to develop ASICs has soared too. Nowadays, the development of an ASIC supposes several million euros. Together with this high cost, it is necessary to take in account other important factors like the development time and the inexistent safety margin when it is necessary to update or redesign the chip. All these drawbacks have helped to expand FPGA market from industrial and communications applications to different consumer electronic products.

One of the more attractive features of FPGAs is their capability to change completely or partially the designs configured into them. Most systems load the FPGA configuration bitstream from a non-volatile external memory. Changing the contents of this memory makes possible to modify the software and hardware of the reconfigurable device. However, this usage is poor because the potential of reconfigurable hardware is not completely employed: Some FPGAs offer the possibility to make partial changes in the design without stopping the rest of the device. This reconfiguration can be done by means of an external component or using one inside of the reconfigurable

logic. For example, an embedded processor might modify the FPGA in which it is running to add a custom coprocessor or to execute some tasks in a new processor.

Being programmable, off-the-shelf and absent of NRE charges, FPGAs have proved their design worth. The question now is whether FPGAs have what it takes to make it in the SoC realm. The main goal of this thesis has been the study of the advantages offered by FPGA partial reconfiguration to increase the flexibility of current SoCs. In particular, ciphering algorithms and secure applications in FPGA-based embedded systems have been the selected application area. This kind of applications is not suitable for present embedded systems [46], but they are nowadays important for many applications as wireless phones and computing. The achieved results shown that partial reconfiguration can improve the lack of versatility and flexibility of present SoCs. Additionally, recent works [216] have demonstrated that partial reconfiguration is useful to increase the security in embedded systems. Although this is only currently applied in research works, partial reconfiguration will be available to the commercial world in the near future.

Below, a summary of the main conclusions of this thesis is presented. These results confirm the advantages of partial reconfiguration in the area of cryptography, but they can be extended to other application areas.

Dynamic partial reconfiguration of cores Partial reconfiguration offers the possibility to reduce the logic resources employed by the implementation of a ciphering algorithm. This optimisation can also be suitable to remove parts of the algorithms that are executed only for short periods of time during the operation, something interesting for hardware/software codesign approaches. These benefits have been successfully applied to the IDEA algorithm, selected as candidate because it is a high throughput computationally intense algorithm that has been suggested for use as a benchmark for reconfigurable computing [131]. Chapter 3 presents the reconfigurable designs of the IDEA algorithm implemented in commercial FPGAs. In these designs, the JBits tool generates the subkeys in software, freeing hardware of this task, and also, modifies the design according to the cipher key. The final result is a simplified design optimised due to the use of reconfiguration. However, JBits is a very complicated tool to develop big designs, because knowledge about the low level details of the device is required. Moreover, JBits is an experimental tool, and the performance results are not competitive. To solve this problem, a new methodology was proposed where different tools are combined: Traditional tools for developing the circuit and the JBits tool for reconfiguring the design. Using this new approach, better results were obtained, being possible to implement on Virtex devices an IDEA circuit that uses a less resources and better performance than any available design in the bibliography. This result shows the potential of partial reconfiguration to optimise hardware cores.

Dynamically reconfigurable systems FPGAs are currently widely used to develop CSoC solutions due to its capacity to implement soft-core processors (SCPs) and the availability of embedded processors in some devices. In this thesis, the possibilities offered by the SCPs to develop embedded systems are studied. The flexibility of SCPs to adapt their architecture or to include specific hardware cores has proved to be suitable to increase the performance of different applications. These qualities turn these systems into good alternatives to replace traditional general-purpose processors. Moreover, when the high flexibility offered by SCPs is combined with run-time partial reconfiguration capabilities, the final result is a platform where complex systems requiring hardware-software task switching can be implemented. As a test application, different ciphering algorithms have been implemented fully in software using SCPs, or as custom hardware cores. The improvement using specific coprocessors allow these modest processors to be suitable to implement secure applications. The next step was to develop a system with the ability to reconfigure the coprocessor unit depending on the task being executed in the processor. The result of this step was a self-reconfigurable platform based on a commercial low-cost FPGA. On this platform the well-known SSH application was executed as a practical example of the benefits of self-reconfiguration. This protocol negotiates, at run-time, the ciphering algorithm to be used from a set of standards, in order to ensure the secure communication between two hosts. A trivial solution would have been to implement all possible algorithms in the reconfigurable device, but the approach proposed in this thesis is to use partial reconfiguration to include in the FPGA only the negotiated ciphering algorithm. When the coprocessor is no longer necessary, or a new algorithm is selected, then it is changed by means of partial reconfiguration.

Final conclusion of this thesis Partial reconfiguration of FPGAs has been successfully evaluated in order to implement secure applications. The development of specific hardware implementations of ciphering algorithms and embedded systems, using partial reconfiguration, has been done. However, it is necessary to remember that the complexity of using partial reconfiguration is the main drawback to adopt this technology. In spite of this problem, the results obtained in this thesis allow the author to be optimistic, and most probably in a few years partial reconfiguration will be used in future electronic systems, as a solution to improve the lack of potential and flexibility of actual embedded systems.

6.1.1. Summary of the contributions

The main contributions of this thesis in the research area of partial reconfiguration and FPGA-based embedded systems are presented below:

1. Detailed analysis of the capabilities offered by the JBits tool for the development of recon-

figurable applications, particularly, for the optimisation of resources and the use of partial reconfiguration. JBits makes possible to design at run-time, being achievable to develop circuits that take into account different environmental circumstances: available device, free area, required speed, etc. The design is customised just before downloading it to the FPGA. It is important to remark that JBits is an interesting tool for HW/SW codesign because it is possible to unify, using the same language, hardware and software descriptions. The IDEA algorithm has been selected as a case-study. An implementation optimised in performance and resource usage based on constant multipliers has been developed: When a new key is given, the design is reconfigured to use the new constants.

2. A mixed methodology to develop partially reconfigurable designs based on the use of traditional HDL and tools for partial reconfiguration like JBits or XPART. Following this new methodology, an implementation of the IDEA algorithm that occupies an 87 % of a Virtex XCV600 and achieves a throughput of 8.3 Gbits/sec has been developed. To the best of my knowledge, this implementation uses less resources and obtains better performance than any available design in the bibliography. Using key replacement through partial reconfiguration can significantly improve the throughput and area of FPGA implementations of cryptographic algorithms.

3. Detailed study of three processors available for reconfigurable hardware: The soft-core processors MicroBlaze and LEON2/3, and the embedded hard-core processor PowerPC405. The advantages and improvements in the performance related to the different parameters of the architecture have been studied, including multiprocessor features, as well as the interfaces to add additional hardware cores as coprocessors. Without a great design effort, an improvement reaching 2 orders of magnitude over all-software solutions could be obtained.

4. Development of a complete set of cryptographic algorithms with the interface necessary to connect them to the soft-core processors MicroBlaze and LEON2. The IDEA, DES, 3DES, Blowfish, AES (Rijndael) and RC4 symmetric key algorithms and the MD5, SHA-1 and SHA-256 hash algorithms have been selected.

5. Use of the Xilinx Modular Design flow with partial reconfiguration to develop embedded systems where the coprocessor is dynamically reconfigured. This system is the first published solution that provides the capability to dynamically reconfigure hardware coprocessors in commercial low-cost devices. It was necessary to develop a custom tool to solve a bug in Xilinx tools when the MicroBlaze processor is used.

6. Development of a self-reconfigurable platform based on Spartan-3 devices. This contribution shows that it is possible to implement partial reconfigurable systems using this low-cost

device and also, allow self-reconfigurability although the device does not include ICAP interface. The use of partial reconfiguration featured in this platform not only improves the performance of the applications, but also increases the flexibility of the system.

7. Use of partial reconfiguration in a real application, the standard SSH, using a self-reconfigurable system implemented on a commercial FPGA running the uClinux operative system. The performance of this application has been improved and, in comparison to other alternatives not using reconfiguration, a reduction of the area requirements was also achieved.

6.1.2. Publications during the PhD. Thesis

In this paragraph the different papers published during the thesis are showed. They have been divided into international publications and conferences, national publications and conferences and related papers and conferences. The related papers and conferences section includes works which are not presented in this thesis, but make use of reconfigurable hardware.

International publications and conferences

- I. Gonzalez, F. Gomez, J. Martinez, *"A HW/SW Co-design Case Study: Implementing a Cryptographic Algorithm in a Reconfigurable Platform"*, XVI Conference on Design of Circuits and Integrated Systems, *DCIS 2001*, Porto, Portugal, November 2001.
- Ivan Gonzalez, Sergio Lopez-Buedo, Francisco J. Gomez and Javier Martinez, *"Using Partial Reconfiguration in Cryptographic Applications: An Implementation of the IDEA Algorithm"*, LNCS 2778, 13th International Conference *FPL 2003*, pp. 194-203, Lisbon, Portugal, September 2003.
- Gonzalez I., Gomez-Arribas, F.J., Lopez-Buedo S., *"Hardware-Accelerated SSH on Self-Reconfigurable Systems"*, Proceedings V IEEE International Conference on Field-Programmable Technology, *IEEE FPT 2005*, pp. 289-290, ISBN 0-7803-9407-0, Singapur, December 2005.
- I. Gonzalez and F. J. Gomez-Arribas, *"Ciphering algorithms in MicroBlaze-based embedded systems"*, IEE Proc.-Comput. Digit. Tech., 153 (2), pp. 87-92, March 2006 (in press).

Spanish publications and conferences

- I. González, F.J. Gómez, S. López-Buedo, J.L. Martínez, J-P. Deschamps, E. Boemo y J. Martínez, *"Implementación del Algoritmo Criptográfico IDEA en Virtex usando JBits"*, II Jornadas sobre Computación Reconfigurable y Aplicaciones, *JCRA 2002*, pp. 155-160, Almuñecar (Granada), Septiembre 2002.

- González I, Gómez-Arribas F, Martínez J, *"Estudio Comparativo de la Implementación del Algoritmo Criptográfico IDEA en Virtex y Virtex II"*, III Jornadas de Computación Reconfigurable y Aplicaciones, *JCRA 2003*, pp. 259-266, Madrid, Septiembre de 2003.

- Aguayo E, González I. y Boemo E., *"Tutorial Xilinx MicroBlaze-uCLinux"*, IV Jornadas de Computación Reconfigurable y Aplicaciones, *JCRA 2004*, pp. 135-145. FPGAS: Computación & Aplicaciones, ISBN 846887667-4, Barcelona, Septiembre 2004.

- González I, Gómez-Arribas F.J y Martínez J, *"MicroBlaze en Sistemas Embebidos para Aplicaciones Criptográficas"*, IV Jornadas de Computación Reconfigurable y Aplicaciones, *JCRA 2004*, pp. 389-396. FPGAS: Computación & Aplicaciones, ISBN 846887667-4, Barcelona, Septiembre 2004.

- Javier Castillo, Iván González, Pablo Huerta y J.I. Martínez, *"Auto-Reconfiguración en FPGAs"*, V Jornadas de Computación Reconfigurable y Aplicaciones, *JCRA 2005*, pp. 3-10. CEDI 2005: I Congreso Español de Informática, ISBN 84-9732-439-0, Granada, Septiembre 2005.

- Iván González, Estanislao Aguayo y Sergio Lopez-Buedo, *"Sistemas Embebidos Auto-Reconfigurables sobre Spartan-3"*, V Jornadas de Computación Reconfigurable y Aplicaciones, *JCRA 2005*, pp. 79-84, CEDI 2005: I Congreso Español de Informática, ISBN 84-9732-439-0, Granada, Septiembre 2005.

- J. Gonzalez-Gomez, I. Gonzalez, F. J. Gomez-Arribas y E. Boemo, *"Evaluación de un Algoritmo de Locomoción de Robots Ápodos en Diferentes Procesadores Embebidos en FPGA"*, V Jornadas de Computación Reconfigurable y Aplicaciones, *JCRA 2005*, pp. 109-116, CEDI 2005: I Congreso Español de Informática, ISBN 84-9732-439-0, Granada, Septiembre 2005.

- Ivan Gonzalez y F. J. Gomez-Arribas, *"Optimización de Algoritmos de Cifrado en Soft Core Processors"*, V Jornadas de Computación Reconfigurable y Aplicaciones, *JCRA 2005*, pp. 291-296, CEDI 2005: I Congreso Español de Informática, ISBN 84-9732-439-0, Granada, Septiembre 2005.

Other related publications and conferences

- González, F. Gómez, J. Martínez, *"LabomatWeb: Recursos Reconfigurables Remotos vía World Wide Web (R3W3)"*, I Jornadas sobre Computación Reconfigurable y Aplicaciones, *JCRA 2001*, Alicante, Octubre 2001.

- I. González, C. J. Venegas, S. López-Buedo, F.J. Gómez, J. Martínez y J. Garrido, *"LABOMICRO: Entorno de test para la verificación de Microprocesadores experimentales sobre circuitos FPGA"*, V Congreso de Tecnologías Aplicadas a la Enseñanza de la Electrónica, *TAEE 2002*, pp. 111-114, Las Palmas de Gran Canaria, Febrero 2002.

- Ivan Gonzalez, Ruben Cabello, Francisco Gomez-Arribas, Javier Martinez, *"Labomat-Web: A Web Laboratory Based on Reconfigurable Computing Technology"*, International Conference on Engineering Education, *ICEE 2003*, Valencia, Spain, July 2003.

- Gómez-Arribas F.J., González I., González J. y Martínez J., *"Laboratorio Web para Prototipado y Verificación de Sistemas Hardware/Software"*, III Jornadas de Computación Reconfigurable y Aplicaciones, *JCRA 2003*, pp. 493-500, Madrid, Septiembre 2003.

- González J., González I. y Boemo E., *"Alternativas Hardware para la Locomoción de un Robot Ápodo"*, III Jornadas de Computación Reconfigurable y Aplicaciones, *JCRA 2003*, pp. 327-334, Madrid, Septiembre 2003.

- Jorge Lopez, Javier S. Pastor, Ivan Gonzalez, Ruben Cabello, Francisco Gomez-A, Javier Martinez, *"A Distance Training Laboratory for Digital Circuit Design and Prototyping of Control Systems"*, Proceedings of *ED-MEDIA 2004*, World Conference on Educational Multimedia, Hypermedia and Telecommunications, Editorial AACE, pp. 678-685. Lugano, Switzerland, June 2004.

- Ivan Gonzalez, Javier Sanchez-Pastor, Jorge Lopez Hernandez-Ardieta, Francisco J. Gomez-Arribas, Javier Martínez, *"Using Reconfigurable Hardware through Web Services in Distributed Applications"*, LNCS 3203, 14th International Conference *FPL 2004*, pp. 1110-1112, ISBN 3-540-22989-2, Antwerp, Belgium, September 2004.

- J. Sanchez Pastor, I. Gonzalez, J. Lopez, F. Gomez-Arribas and J. Martinez, *"A Remote Laboratory for Debugging FPGA-Based Microprocessor Prototypes"*, 4th IEEE International Conference on Advanced Learning Technologies, *IEEE ICALT 2004*, pp. 86-90, ISBN 0-7695-2181-9, Joensuu, Finland, September 2004.

- Javier Sanchez-Pastor, Ivan Gonzalez, Jorge L. Hernandez-Ardieta, Francisco J. Gomez-Arribas y Javier Martínez, *"Uso de Hardware Reconfigurable a través de Servicios Web en Aplicaciones Distribuidas"*, IV Jornadas de Computación Reconfigurable y Aplicaciones, *JCRA 2004*, pp. 45-52. FPGAS: Computación & Aplicaciones, ISBN 846887667-4, Barcelona Septiembre 2004.

- J. Gonzalez-Gomez, I. Gonzalez, F. J. Gomez-Arribas and E. Boemo, *"Evaluation of a locomotion algorithm for work-like robots on FPGA-embedded processors"*, LNCS, International Workshop on Applied Reconfigurable Computing, *ARC 2006*, Delft, The Netherlands, March 2006 (in press).

6.2. Future work

Once the results of the thesis have been summarised, and the potential of the run-time reconfiguration has been presented, there are a lot of research areas and applications that can be proposed as future work:

Secure applications The goal is to continue implementing secure applications on self-reconfigurable CSoCs. Modern secure applications that make use of dynamic selection of algorithms, like SSH, are good candidates for partial reconfiguration. Also, there is a great interest in the security of embedded systems. This interest can be translated to FPGA-based embedded systems. For example, it is necessary to guarantee an environment appropriate for remote and secure downloading of IPs. There are some research works related with this feature [217, 206], where some solutions are provided to ensure the integrity of the bitstreams and the own system.

Operative systems for reconfigurable hardware The idea is to get support for partial reconfiguration in operative systems running in CSoCs. In the example of the SSH application, this application manages the partial reconfiguration of the coprocessors. However, if different tasks need to use dynamically coprocessors, it is not possible that each application could manage individually the partial reconfiguration. The solution is to bring the partial reconfiguration to the operative system level.

Multiprocessor systems on FPGA The great capacity of modern FPGAs allows designers to build multiprocessor systems based on networks of homogeneous or heterogeneous soft-core processors. These systems can be integrated with embedded hard-core processors when the reconfigurable device includes them. The different tools and methodologies for traditional processor-based systems have to be adapted to the development of these systems. An additional step on adaptability is the use of partial reconfiguration to replace/add more processors or custom hardware cores in order to increase the performance or flexibility of the system.

Pervasive computing There is another application area that is open up for the use of partial reconfiguration: Pervasive (or distributed) systems. In next future, most objects and devices used

commonly in everyday life are expected to be intimately integrated with computer systems. The concept of pervasive computing is creating an environment saturated with computing and communication capabilities. The hardware technology needs to provide resources to improve the coordination and cooperation of numerous devices and services, and execute diverse and multiple applications. The dynamic reconfiguration capability of reconfigurable hardware can be used to support a variety of applications, variable system requirements and standards, as well as different operating conditions.

Modular robotics Modular self-reconfigurable robots offer the promise of more versatility, more robustness and lower cost. They are composed of modules, capable of attach and detach one to each other, changing the shape of the robot. In this context, the word "reconfigurable" means the ability of the robot to change its form, not a hardware reconfigurable system. In the last years, the number of robots following this approach has grown substantially. An additional step on versatility is the use of FPGA technology instead of conventional microprocessors. It gives the designer the possibility of implementing new architectures, faster control algorithms, or dynamically modifying the hardware to adapt it to a new situation. In summary, modular reconfigurable robots controlled by FPGAs are not just able to change their shapes, but also their hardware, therefore, a complete versatility can be achieved.

Apéndice A

Codiseño en Sistemas Embebidos Pequeños

La evolución de la tecnología de los circuitos integrados está motivando nuevas aproximaciones al diseño de circuitos digitales [218, 151]. En la actualidad, un circuito FPGA [219] incluye en un mismo chip, memoria y recursos reconfigurables. Una plataforma basada en FPGA cumple con la solución a nivel de sistema. Complementando la FPGA con un microprocesador se incrementan las posibilidades de diseño, existiendo circuitos llamados System-on-Chip [2] que integran un core procesador y parte reconfigurable.

Actualmente se encuentran disponibles varias plataformas de investigación y comerciales. Por poner un ejemplo: RC1000-PP de Celoxica [165], CARMEN de Sidsa [220], Riley-2 [221] y Labomat 3 [47].

La mayoría de los diseños digitales se pueden ver como una colección de varios componentes hardware y software, cuya operación combinada provee un servicio. El codiseño Hardware/Software reune estos objetivos a nivel de sistema, explotando la sinergia del hardware y el software a través su diseño concurrente.

En este apéndice se estudian las posibilidades de las técnicas de codiseño en una pequeña plataforma, donde se proveerán soluciones a problemas que no sería posible resolver eficientemente sin la combinación de la sinergia HW y SW. El interés de este caso reside en la posibilidad de que la futura generación de circuitos electrónicos de bajo precio para aplicaciones de usuario forzará la integración dentro del mismo de una parte procesador y una parte reconfigurable.

Como caso de estudio, se presenta la implementación de IDEA [129] usando la plataforma Labomat3. Este algoritmo es usado como marco para diferentes arquitecturas desarrolladas usando

técnicas de codiseño.

Labomat3 es una plataforma para enseñanza e investigación en sistemas reconfigurables desarrollada por el Logic System Laboratory, Swiss Federal Institute of Technology en Lausanne (Swiss).

En la figura A.1 se muestra un diagrama de bloques [47] de la plataforma. La arquitectura se divide en dos partes principales:

1. La parte procesador se construye alrededor de un microcontrolador de 32 de Motorola, MC68EN360, con manejo de protocolo Ethernet integrado [222]. Está conectado a 512 KB de boot EPROM, 32 MB de DRAM y 4KB de SRAM de doble puerto

2. La parte reconfigurable se construye alrededor de las FPGAs XC4013E y XC6216 [223]. La XC6216 está conectada a una SRAM adicional de 128 KB. La XC4013E está conectada a la memoria SRAM de doble puerto, la cual está también conectada al microcontrolador.

El microcontrolador puede acceder directamente a la FPGAs como periféricos a través del los buses de datos y direcciones. La SRAM de doble puerto facilita el intercambio de datos entre el microcontrolador y la parte reconfigurable.

Un gran bus de 80 líneas interconecta las FPGAs y un conjunto de 52 I/Os está directamente disponible en conectores de 10 pines. El bus de extensión maneja los datos globales, el bus de direcciones y el bus de control. Las interrupciones y el control de bus se realiza mediante un chip de lógica programable (MAX7128).

Un generador de reloj programable disponible en la placa provee un rango de frecuencias de reloj. Cada FPGA recibe dos diferentes señales de reloj y cada una de estas señales puede ser asignada a diferentes fuentes. El reloj del sistema por defecto es de 25 MHz.

La comunicación con el mundo exterior se realiza a través de un puerto Ethernet 10Base-T y un puerto RS-232.

El microcontrolador MC68360 integra capacidades de comunicación. El código de bootstrap se ejecuta desde una EPROM. Este código simplemente carga un sistema operativo completo y aplicaciones vía Ethernet. Además, existen una gran cantidad de aplicaciones de software libre para este microcontrolador.

La plataforma Labomat3 ofrece un potente conjunto de interfaces de comunicación para configurar y controlar toda la placa. RTEMS [224], un sistema operativo de tiempo real, se ejecuta en la placa y provee los drivers para Ethernet y el stack TCP-IP, el cual permite usar protocolos TCP-IP estándar de alto nivel.

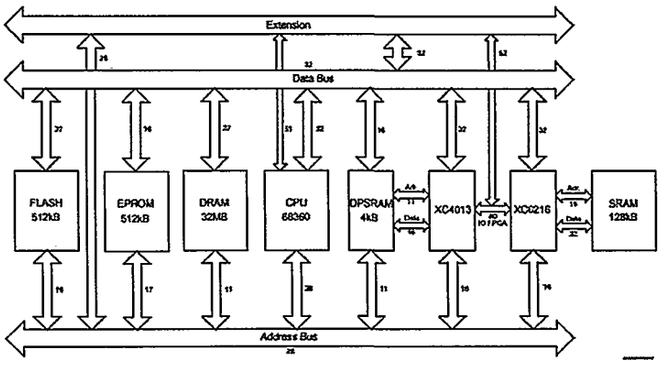


Figura A.1: Arquitectura de la plataforma Labomat3

A.1. Consideraciones del sistema de codiseño

Géneralmente, las técnicas de codiseño intenta implementar en hardware aquellos componentes que requieren de un mayor tiempo de procesamiento. Sin embargo, las limitaciones de los recursos pueden restringir el particionado HW/SW del problema. La presencia de realimentación (*feedback*) en un algoritmo reduce la cantidad de segmentación o paralelismo posible, decrementando el rendimiento de la solución hardware completa.

Finalmente, el rendimiento de una solución de codiseño dependerá de la correcta definición de las interfaces HW/SW [225]. A pesar de que el flujo de datos de IDEA puede ser paralelizado por el procesado de múltiples bloques de datos a través de un mismo pipeline, el uso de feedback necesariamente supone un límite superior al paralelismo disponible. El componente más crítico de una implementación IDEA es el multiplicador de 16 bits. La falta de un multiplicador hardware en la arquitectura del procesador penaliza la solución software. Un multiplicador puede ocupar un espacio prohibitivo dependiendo del tamaño del los operandos. El codiseño puede establecer un compromiso entre ambas soluciones.

El multiplicador módulo 2^{16+1} en IDEA se desarrolla para datos de 16 bits usando el siguiente algoritmo:

$xy \bmod (2n+1) = (xy \bmod 2n - xy \operatorname{div} 2n) \bmod (2n+1)$

Esta ecuación se mapea eficientemente tanto en hardware como en software [226].

Una implementación software estándar de IDEA en C alcanza un cifrado de datos de 16 Mbits/s en un Pentium Pro 180 MHz. Una implementación hardware del algoritmo aumenta el rendimien-

	Ciclos empleados	Tiempo (us)	Valor Normalizado
DRAM	2.5	0.1	1
4KB DPSRAM *	24	0.9	9
FPGA Acces	10	0.4	4

* Operaciones W/R 2 x 16 en DPSRAM

Tabla A.1: Tiempos de acceso a memoria

to.

En este caso, el estudio se centra en un rendimiento relativo obtenido con diferentes soluciones de particionamiento HW/SW, todas implementadas en la misma plataforma. Las limitaciones de recursos de esta plataforma no ofrecen resultados comparables a los mostrados previamente en esta tesis. Sin embargo, el principal interés es enfrentarse al problema global: captura de datos, procesamiento de datos y almacenamiento de datos, en la plataforma seleccionada.

La interface entre los diferentes módulos de la plataforma Labomat 3 juegan una papel importante en el throughput de los diseños. Las medidas de los tiempos de acceso a los diferentes elementos se muestra en la tabla A.1.

Para comunicar la FPGA con los buses del microcontrolador, se ha implementado un registro de 32 bits y la lógica de control asociada en la FPGA. El procesador puede leer o escribir en el registro siguiendo el protocolo de acceso a periférico que se muestra en la figura A.2. Un contador adicional incluido en la FPGA almacena el número de ciclos que han pasado. Estos valores coinciden razonablemente con los obtenidos por las funciones de medición de tiempo disponibles en el sistema operativo RTEMS.

Comparando el tiempo de acceso a la FPGA con el tiempo de acceso a la DRAM, se obtiene un factor 4 de penalización en la plataforma Labomat3 por acceso a los recursos reconfigurables. Esta diferencia puede asociarse al retardo del chip de control de bus (MAX7128) que maneja de forma oculta las especificaciones de tiempos de bus y los ciclos de configuración de la FPGA.

Se hace notar que todas las medidas se han desarrollado mientras el microcontrolador ejecutaba el sistema operativo RTEMS.

A.2. Implementación software

A pesar de sus cambios computacionales, el flujo de datos de IDEA es relativamente simple. Existe una gran variedad de referencias a paquetes que incluyen implementaciones de IDEA en código fuente, incluyendo el apéndice del Applied Cryptography [126], el paquete Pretty Good

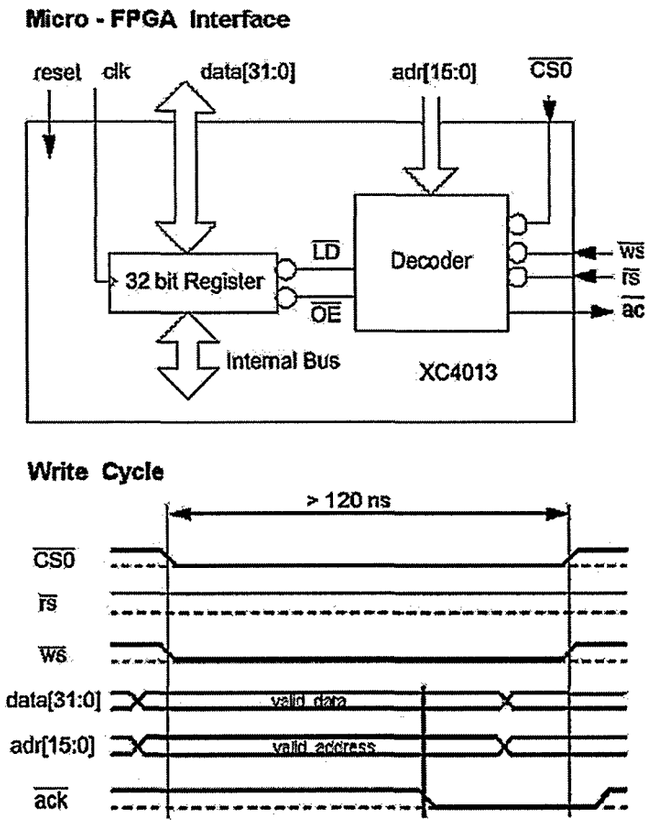


Figura A.2: FPGA: Acceso a periférico

Implementación	FPGA	CLBs*	Frecuencia (MHz)
Unidad Multiplicador	xc4013e-4-hq240	307 (53 %)	27
Unidad Round	xcv1000e-8-bg560	970 (7 %)	25
	xc4036ex-2-hq304	1277 (98 %)	3
	xcv150-6-bg352	970 (56 %)	14
	xc2s150-6-fg456	970 (56 %)	14
Unidad Idea	xc4013e-4-hq240	558 (96 %)	8

* En el caso de la familia Virtex y Spartan se trata del nº de Slices

Tabla A.2: Recursos hardware de las distintas implementaciones

Privacy, y la librería criptográfica RSAREF. El criterio final para un benchmark razonable es tener métricas útiles. Para IDEA hay dos: latencia de cifrado de un solo bloque, y el ancho de banda sostenido a través del dispositivo.

Para ambas medidas, el dispositivo deberá operar en modo ECB (Electronic Code Book), el cual asegura un tratamiento idéntico e independiente para cada uno de los bloques. Esto puede ser asumido si la expansión de subclaves ya ha ocurrido.

La primera métrica del benchmark es el rendimiento de los componentes computacionales midiendo el tiempo (en ms) para cifrar y descifrar un único bloque de datos de 500 Kbits.

La segunda medida es el máximo ancho de banda sostenido, típicamente se mide en Mbits/s. Esta medida está afectada mayormente por el tamaño del dispositivo y la disponibilidad de recursos computacionales paralelos.

Asumiendo un valor de 46 ciclos para una instrucción de multiplicación, para el microcontrolador *MC68360 a 25 MHz se estima un throughput para IDEA de 0.96 Mbits/s*. Los resultados experimentales para el procesamiento de 1 Mbit de datos fue de 18711 milisegundos, que corresponden a un *throughput real de 0.54 Mbits/s*. Se debe recordar que la medida fue desarrollada sobre un sistema operativo multitarea.

A.3. Alternativas basadas en codiseño HW/SW

A continuación se estudian tres implementaciones con diferentes codependencias de codiseño. Usando la plataforma Labomat3, la principal restricción son los recursos hardware reconfigurables. En el mejor de los casos, los recursos hardware permiten una implementación secuencial de IDEA basada en la implementación completa de un round.

El test consiste en cifrar y descifrar 500 Kbits (1 Mbits en total) de datos aleatorios. La tabla A.2 muestra los recursos hardware necesarios para cada implementación.

Con diferente particionamiento HW/SW las implementaciones propuestas son:

- Unidad Multiplicador Combinacional
- Unidad Round Combinacional
- Unidad Round Secuencial
- Unidad IDEA Secuencial

A.3.1. Unidad multiplicador combinacional

El análisis del algoritmo IDEA evidencia una cierta dependencia del multiplicador. Como primera aproximación se decidió implementar este multiplicador como unidad hardware, por lo que fue necesario incluir la interfaz de registros explicada en la sección A.1.

El objetivo era desarrollar un multiplicador combinacional (figura A.3), cuyo tiempo de operación es de un ciclo de reloj. El resultado obtenido en la FPGA XC4013E fue de una frecuencia de reloj de 27 MHz con un 53 % de ocupación. Estos rangos de velocidad de operación permiten que el microcontrolador pueda operar a 25 MHz. Sin embargo, la ocupación es demasiado alta para nuestros propósitos de implementar un round hardware.

Los resultados experimentales obtenidos para el bloque de test fueron de 13861 milisegundos, que se corresponde con un *throughput de 0.77 Mbits/s*. Este desarrollo muestra una mejora de un 25 % con respecto a la solución software.

Solo el multiplicador como unidad hardware no es un buen ejemplo de codiseño. Sin embargo, permite demostrar que la solución software presenta una gran dependencia de la operación de multiplicación.

A.3.2. Unidad round combinacional

En esta solución, la implementación en la FPGA incluye los elementos necesarios para un round IDEA. La figura A.4 muestra un esquema con 4 multiplicadores, 4 sumadores y 6 xor. En este caso, es necesaria una interface de 226 pines I/O pines.

La FPGA XC4013E de la plataforma Labomat3 no tiene un número suficiente de CLBs. El diseño además, es pad-limited. Por esta razón se eligieron otras FPGAs. Los resultados se muestran en la tabla de la sección A.3, dando la información sobre la relación entre la velocidad y los recursos empleados.

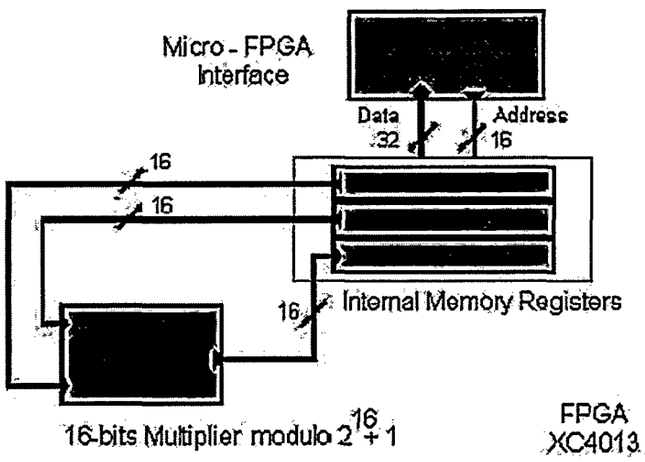


Figura A.3: Unidad Multiplicador Combinacional

A.3.3. Unidad round secuencial

Debido a la imposibilidad de introducir un round combinacional completo en la FPGA XC4013E, se propuso un nuevo diseño de un multiplicador secuencial. Algunas implementaciones en pequeñas FPGAs ya han sido llevadas a cabo [131], donde el multiplicador fue sustituido por un simple multiplicador de suma y rotación. El tamaño de este multiplicador es menor, sobre 53 CLBs (9 % sin incluir la interface de registros), pero necesita 17 ciclos de reloj para realizar una operación completa. Este nuevo diseño es claramente peor que el combinacional anterior.

Las unidades necesarias para desarrollar un round son mostradas en la figura A.5. Además del multiplicador, hay un sumador, una unidad xor y el circuito lógico de control (Finite-State-Machine) para habilitar los registros en los ciclos de reloj apropiados.

Una memoria de doble puerto implementada dentro de la FPGA almacena las seis subclaves de 16 bits y los cuatro datos de 16 bits.

El resultado ha sido satisfactorio, disponiendo de un round completo con un 96 % de ocupación y una frecuencia de reloj de 10 MHz. Con este diseño son necesarios 87 ciclos de reloj para completar el round.

Los resultados experimentales obtenidos para el bloque de test fue de 10241 milisegundos, que

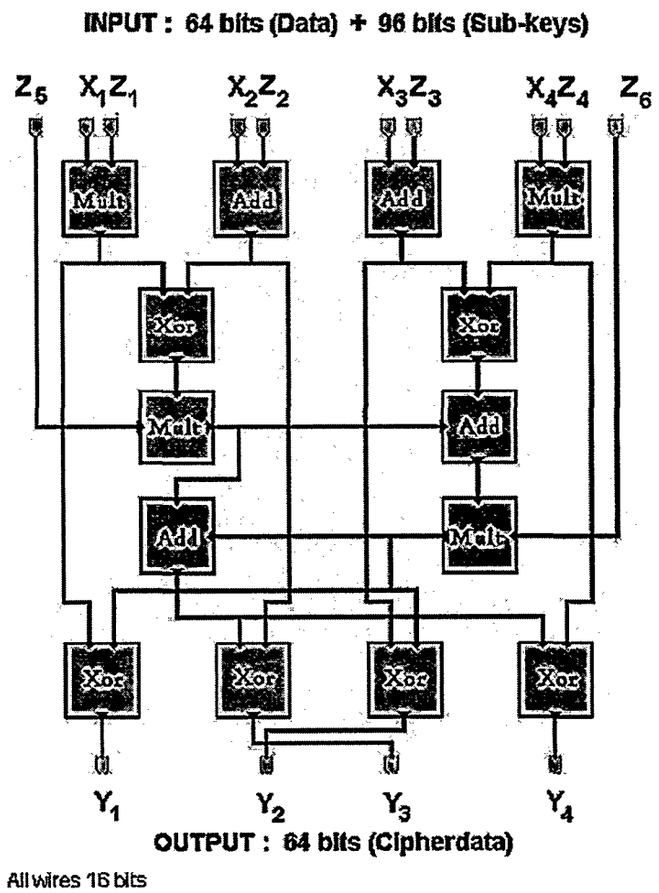


Figura A.4: Unidad Round Combinacional

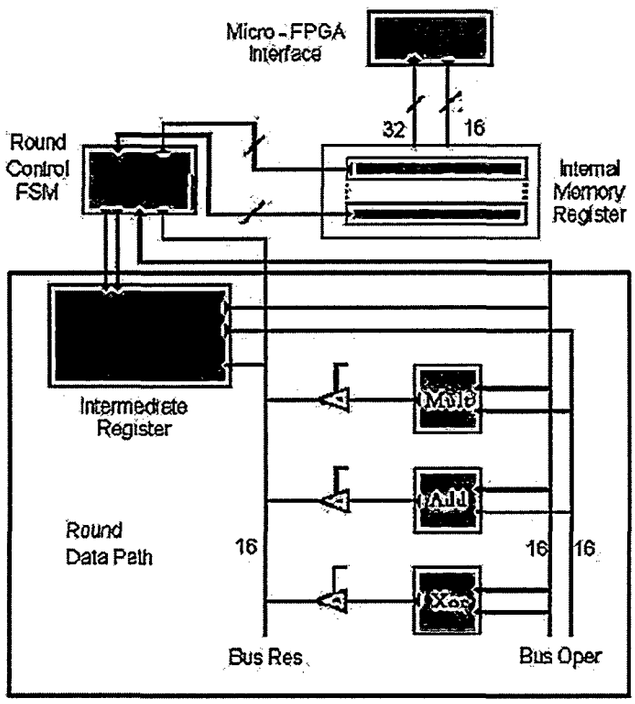


Figura A.5: Unidad Round Secuencial

se corresponden con un *throughput de 0,98 Mbits/s*. La mejora obtenida es un 50 % mejor que la solución software.

A.3.4. Unidad IDEA secuencial

En esta solución, se realiza un implementación completa de IDEA como operación secuencial de la unidad round anterior. El nuevo diseño necesita el control de una FSM con muchos más estados y registros intermedios. Además, todas las subclaves empleadas en el algoritmo deben estar almacenadas en registros. Como no existe suficiente espacio para implementar la memoria de doble puerto en la FPGA con capacidad suficiente para alojar el número de registros necesarios, se empleó como alternativa la DPSRAM de 4 Kbyte de la plataforma Labomat3 para almacenar todas las subclaves. El resultado obtenido para la unidad IDEA completa tiene ahora una ocupación del 59 % y opera a una frecuencia de 15 MHz. El número de ciclos de reloj necesarios para completar un round es de 104 ciclos.

Los resultados experimentales obtenidos para el bloque de test fue de 8312 milisegundos, lo que corresponde con un *throughput de 1.0 MBits/s*. El rendimiento obtenido en este caso es un 125 % mejor que la solución software.

A.4. Resultados

La plataforma Labomat 3 nos ha permitido experimentar con distintas técnicas de codiseño debido a su particular arquitectura. En ella se dispone de recursos reconfigurables directamente comunicados con una parte procesador de propósito general. Los recursos reconfigurables disponibles son limitados para el objetivo propuesto. Esto nos ha permitido comprobar que un mayor aprovechamiento de estos recursos hardware viene acompañado de un incremento en el rendimiento, por tanto, un particionamiento donde la mayor parte del problema se trata en hardware presenta un mejor resultado. Esta conclusión, que basa su fundamento en que el hardware específico funciona siempre mejor que el software, plantea el análisis de otra característica importante a tener en cuenta en codiseño, la interface hardware/software. En el caso de la plataforma Labomat 3 los distintos mecanismos de comunicación entre procesador y FPGA tienen distinta penalización y complejidad de diseño, lo cual ha obligado a un estudio de estos mecanismos con el fin de elegir la mejor interface.

La solución con mejor rendimiento que se presenta ha sido aquella que a resuelto la mayor parte del algoritmo IDEA en hardware, operando a una frecuencia razonable, y que ha empleado la mejor interface de comunicación al tiempo que ha minimizado el intercambio de datos.

Implementación	Tiempo (ms)	Throughput (Mbits/s)	Mejora
Software	18711	0,54	0 %
Multiplicador Combinacional	13861	0,77	25 %
Round Secuencial	10241	0,98	50 %
Idea Secuencial	8312	1.0	125 %

Tabla A.3: Resumen de resultados

Los resultados obtenidos para las tres implementaciones presentadas, tabla A.3, han obtenido un mejor rendimiento que la solución software, presentado para la mejor solución de codiseño una mejora del 125 % sobre la solución software.

Apéndice B

El API JBits

JBits es un API de Java que provee un acceso muy eficiente a todos los recursos reconfigurables de una FPGA. Intenta proveer una solución para el soporte de RTR (Run-Time Reconfiguration) en las FPGAs basadas en SRAM, en las cuales es posible modificar el circuito durante varias veces a lo largo de la ejecución de la aplicación. Por sus capacidades de diseño a bajo nivel, JBits provee las herramientas necesarias para modificar o crear de forma eficiente un diseño, soportando RTR y diseños estáticos.

Un programa de JBits típico crea la lógica en un nuevo bitstream, modifica la lógica y la interconecta en un bitstream existente, o realiza un análisis del bitstream. El API de JBits está construida a partir de cuatro métodos principales. Los dos primeros permiten que un bitstream se pueda leer o escribir desde/a un fichero. El otro método permite manipular los recursos, por ejemplo, poner un punto de interconexión programable (PIP), a un cierto valor. El último de los métodos permite leer el estado de los recursos desde el bitstream. El resto del API es un conjunto de constantes que definen los recursos reconfigurables y sus características.

La comunicación se puede establecer con una FPGA que pertenece a una placa mediante el API XHWIF [227]. La Xilinx HardWare InterFace es un API que provee métodos para comunicarse con las placas basadas en FPGAs. Esta incluye un método para leer el bitstream de las FPGAs. Hay métodos para escribir en las FPGAs, dar pulsos de reloj y escribir o leer de las memorias de la placa. Esencialmente, XHWIF provee una interface universal para comunicarse con diferentes placas.

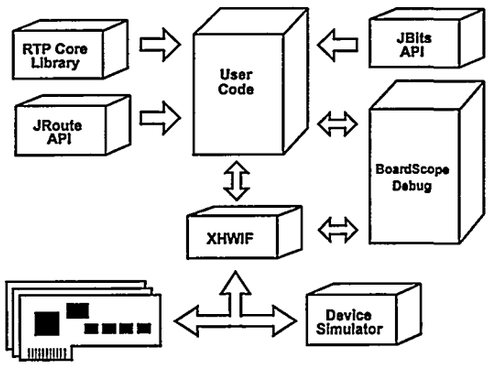


Figura B.1: Diagrama de Bloques de JBits

B.1. Entorno

A pesar de que JBits es una herramienta de diseño muy poderosa, diseñar a este nivel puede ser muy difícil incluso para los diseños más triviales. Sin embargo, debido a las técnicas de programación orientada a objetos que añade Java, se pueden construir herramientas por encima de JBits que proveen niveles de abstracción bastante altos (figura B.1). Herramientas como JRoute, RTPCores y JRTR permiten a los usuarios un control del diseño bastante aceptable. JRoute[110] es un rutador que tiene métodos con distintos niveles de abstracción. Un usuario puede definir completamente cada recursos en una conexión, definir un plantilla a seguir, o permitir que el auto-router lo haga todo. La especificación de los Cores Parametrizables en Tiempo de Ejecución (RTP) [109] proveen un modo de abstraerse de las llamadas de configuración a bajo nivel, creando un entorno similar a los lenguajes de descripción de hardware tradicionales (HDLs) mientras concurrentemente fuerzan la habilidad para realizar modificaciones a nivel bitstream. El API JRTR [111] es una extensión del API JBits para aprovechar la ventaja de la reconfiguración parcial soportada por la familia Virtex. Esta interface provee de un modelo cacheado que automáticamente controla los cambios en la configuración de los datos y solo los datos modificados son escrito o leídos del dispositivo.

B.2. Virtex Device Simulator

Otra herramienta construida a partir de JBits es el Virtex Device Simulator (VirtexDS) [228]. El VirtexDS trabaja directamente a partir del bitstream de la FPGA usando JBits para analizar

la configuración de los recursos del dispositivo. La simulación a nivel de dispositivo tiene varias ventajas sobre los simuladores tradicionales como los simuladores de VHDL. La primera ventaja es que soportan RTR mientras que el resto de los simuladores reconducen el soporte de RTR. Únicamente JBits con el VirtexDS permite que los circuitos sean sintetizados e insertados en la simulación a nivel de bitstream en tiempo de ejecución. Usando el entorno JBits, el comportamiento del circuito durante la reconfiguración es simulado de forma precisa como resultado de la aproximación a la emulación del hardware de la FPGA usado en VirtexDS. La simulación a nivel de dispositivo tiene la ventaja de que es independiente de la herramienta de implementación. El bitstream es el formato común entre todas las herramientas, de forma similar al EDIF pero a más bajo nivel. Otra ventaja sobre los simuladores tradicionales viene del hecho de que el VirtexDS implementa el API XHWIF, lo cual permite al usuario cambiar de forma transparente entre el simulador y la FPGA actual.

Otra herramienta de debug es el BoardScope [229], el cual permite al diseñador ver y controlar las operaciones en la FPGA de forma gráfica.

B.3. Reconfiguración parcial con JBits

JBits soporta RTR a través de los siguientes mecanismos:

```
JBits jbits = new JBits(deviceType);                    /* Stage 1 */
jbits.readPartial(inputfile);
board.setConfiguration(device, jbits.getPartial());     /* Stage 2 */
jbits.set(...);                                         /* Stage 3 */
board.setConfiguration(device, jbits.getPartial());     /* Stage 4 */
```

1. El bitstream original es creado usando las herramientas de desarrollo de Xilinx o el API JBits, y leído dentro de una instancia de la clase JBits.

2. Esta configuración es enviada a la placa - jbits.getPartial() - devolviendo TODA la configuración de la placa.

3. Usando el API JBits se cambia el circuito original, y la librería determina que parte del circuito ha cambiado.

4. Se envía la nueva configuración a la placa, pero el API solo envía la parte del bitstream que corresponde a la parte del circuito que ha cambiado (jbits.getPartial()).

B.4. CoreTemplates y RTP Cores

Muchos de los diseños realizados en FPGAs se han construido a partir de librerías de hardware disponibles, cuyos diseños se han implementado enteramente a muy bajo nivel empleando APIs como JBits. Como resultado, es posible definir algunos diseños comúnmente usados, o cores, construyendo con ellos diseños mucho más complejos. Los RTP Cores son componentes de circuitos a alto nivel que pueden configurarse en tiempo de ejecución, lo cual permite al diseñador decidir los atributos de estos cores en tiempo de uso, por ejemplo, el ancho de las entradas y el número de etapas que se deben usar. Con estos parámetros es posible proveer velocidad con restricciones de área, permitiendo menor tamaño de circuito y un mayor rendimiento.

Naturalmente Java provee una muy buena encapsulación para estos diseños, por ejemplo, el uso de constructores, clases abstractas, abstracción de datos y ocultación de información pueden simplificar en gran medida el uso y diseño de estos RTP Cores. Para facilitar el desarrollo de RTP Cores, JBits provee de un paquete llamado CoreTemplate el cual establece una interface estándar para la declaración de cores y permite al usuario instanciar estos cores fácilmente. Los diseñadores pueden implementar diseños reusables usando este marco de trabajo (framework) y el usuario puede proveer los parámetros e implementar el circuito usando implement().

Existen dos metodologías para los cores:

1. *Low Level Sub Cores*: Establecen las especificaciones físicas y lógicas del diseño del circuito y configuran los recursos CLB directamente usando la librería JBits.

2. *Top Level RTP Cores*: Estos cores están implementados a partir de SubCores.

El paquete JBits incluye en sí mismo algunos de estos cores como ejemplos para mostrar como se pueden implementar.

B.5. Java como lenguaje de descripción HW/SW

Con el uso del API JBits es posible definir, mediante una aplicación Java, tanto el diseño hardware como la parte que corresponde al software (figura B.2). Esto rompe con el flujo de diseño tradicional que se sigue utilizando en sistemas reconfigurables, el cual se muestra en la figura B.3. Inicialmente un programa que emplee JBits permite una enorme vinculación entre el hardware y el software, siendo posible que determinados eventos que se producen en el software afecten al diseño hardware.

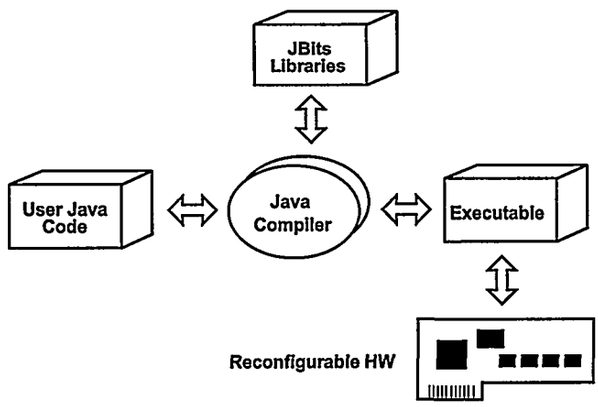


Figura B.2: Flujo de diseño con JBits

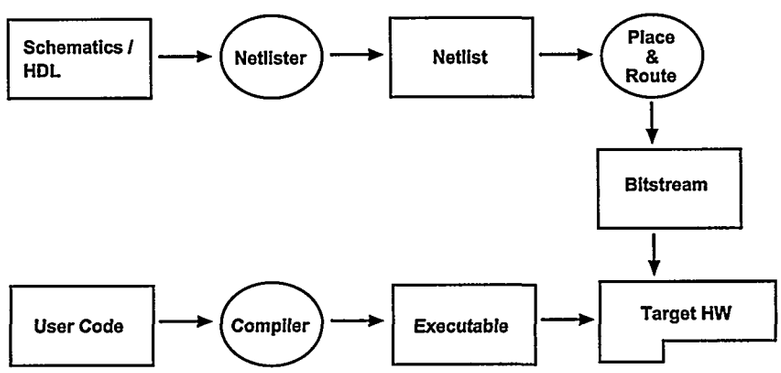


Figura B.3: Flujo de diseño tradicional en Sist. Reconfigurables

B.6. Ejemplo de código JBits: Multiplicador mod. $2^{16}+1$

```
import com.xilinx.JBits.CoreTemplate.Bus;
import com.xilinx.JBits.CoreTemplate.Gran;
import com.xilinx.JBits.CoreTemplate.Net;
import com.xilinx.JBits.Virtex.RTPCore.ULPrimitives.LUT;
import com.xilinx.JBits.CoreTemplate.Bitstream;
import com.xilinx.JBits.CoreTemplate.CoreParameterException;
import com.xilinx.JBits.Virtex.ConfigurationException;
import com.xilinx.JBits.CoreTemplate.Offset;
import com.xilinx.JBits.CoreTemplate.Place;
import com.xilinx.JBits.CoreTemplate.Port;
import com.xilinx.JBits.CoreTemplate.RTPCore;
import com.xilinx.JBits.CoreTemplate.CoreException;
import com.xilinx.JBits.Virtex.RTPCore.Arithmetic.Adder;
import com.xilinx.JBits.Virtex.RTPCore.Basic.Constant;
import com.xilinx.JBits.Virtex.RTPCore.Arithmetic.ConstantComparator;
import com.xilinx.JBits.Virtex.RTPCore.Arithmetic.ConstantComparatorParameters;
import com.xilinx.JBits.Virtex.RTPCore.Basic.Multiplexer.MUX2_1;
import com.xilinx.JBits.Virtex.RTPCore.Basic.Gnd;
import com.xilinx.JBits.Virtex.RTPCore.Arithmetic.Subtracter;


public class Mult_mod extends RTPCore {

   public Mult_mod(String instanceName, Bus din, Bus dout) throws CoreException {

      super(instanceName);
      this.busWidth = din.getWidth();
      try {
         checkParameters(din.getWidth(), dout.getWidth());
      } catch (CoreParameterException cpe) {
         throw new CoreException(cpe);
      }

      setHeightGran(calcHeightGran());
      setWidthGran(calcWidthGran());
      setHeight(calcHeight());
      setWidth(calcWidth());

      dinPort = newInputPort("DIN", din);
      doutPort = newOutputPort("DOUT", dout);
   }

   public final void implement(int constante) throws CoreException {

      if(this.first == true) {
         if(constante == 0) {
            System.out.println("Multiplicador optimizado - Clave cero");
            implement_resto(constante);
         }
         else if(constante != 0) {
            System.out.println("Multiplicador");
```

```
            implement_mult(constante);
         }
         this.first = false;
      }
      else {
         if(constante == 0 && this.last_constant != 0) {
            System.out.println("Multiplicador optimizado - Clave cero");
            implement_resto(constante);
         }
         else if(constante != 0 && this.last_constant == 0) {
            System.out.println("Multiplicador - Constante "+constante);
            implement_mult(constante);
         }
         else if(constante != 0 && this.last_constant != constante) {
            cambiar_memoria(constante);
            System.out.println("Modificar memoria - Constante "+constante);
         }
      }
      this.last_constant = constante;
   }

   public final void implement_mult(int constante) throws CoreException {

      int i, k, l;
      Offset offset = calcAbsoluteOffset();

      Bus addr = newBus("ADDR",16);
      dinPort.setIntSig(addr);

      Bus []addrs = new Bus[4];
      for(i = 0; i <4; i++)
         addrs[i] = new Bus("ADDR"+i,null,4);

      Bus []data = new Bus[4];
      for(i = 0; i <4; i++)
         data[i] = new Bus("DATA"+i,null,20);

      Bus data1_1 = new Bus("DATA1_1",null,20);
      Bus data1_2 = new Bus("DATA1_2",null,24);
      Bus data1_3 = new Bus("DATA1_3",null,24);

      Bus data2_1 = new Bus("DATA2_1",null,16);

      Bus data3_1 = new Bus("DATA3_1",null,20);
      Bus data3_2 = new Bus("DATA3_2",null,16);

      Bus data4_1 = new Bus("DATA4_1",null,16);

      Bus mod_out = new Bus("mod_out",null,16);

      Bus constante_bus = new Bus("constante_bus",null,16);
      Net select = new Net("select",null);
```

```
Bus dina = new Bus("dina",null,16);
Bus dinb = new Bus("dinb",null,16);
Bus dout = newBus("dout",16);
doutPort.setIntSig(dout);

/* Interconexion de buses */
// Establecemos direcciones
for(i = 0; i <4; i++) {
   addrs[0].setNet(i,addr.getNet(i+12));
   addrs[1].setNet(i,addr.getNet(i+8));
   addrs[2].setNet(i,addr.getNet(i+4));
   addrs[3].setNet(i,addr.getNet(i));
}

// Establecemos lineas de datos DATA2
for(i = 0; i <16; i++) {
   data2_1.setNet(i,data[1].getNet(i+4));
}

// Establecemos lineas de datos DATA4
for(i = 0; i <16; i++) {
   data4_1.setNet(i,data[3].getNet(i+4));
}

// Establecemos lineas de datos DATA3
for(i = 0; i <16; i++) {
   data3_2.setNet(i,data3_1.getNet(i+4));
}

// Establecemos lineas de datos DATA1
for(i = 0; i <24; i++) {
   if(i <4)
      data1_2.setNet(i,data[1].getNet(i));
   else
      data1_2.setNet(i,data1_1.getNet(i-4));
}

for(i = 0; i <32; i++) {
   if(i <4)
      dina.setNet(i,data[3].getNet(i));
   else if(i <8)
      dina.setNet(i,data3_1.getNet(i-4));
   else if(i <16)
      dina.setNet(i,data1_3.getNet(i-8));
   else
      dinb.setNet(i-16,data1_3.getNet(i-8));
}

/* Memoria_UL */
this.memoria = new MemoriaUL("memoria",addrs,data);
this.addChild(this.memoria);
Offset memoriaOffset = this.memoria.getRelativeOffset();
memoriaOffset.setHorOffset(Gran.CLB, 0);
```

```
memoriaOffset.setVerOffset(Gran.CLB, 0);
this.memoria.implement(constante);

/* Adder Core */
Adder sumador1_1 = new Adder("SUMADOR1_1",data[0],data2_1,data1_1);
this.addChild(sumador1_1);
Offset sumador1_1Offset = sumador1_1.getRelativeOffset();
sumador1_1Offset.setHorOffset(Gran.CLB, 2);
sumador1_1Offset.setVerOffset(Gran.CLB, 0);
sumador1_1Offset.setHorOffset(Gran.SLICE, 0);
sumador1_1.implement();

Adder sumador1_2 = new Adder("SUMADOR1_2",data[2],data4_1,data3_1);
this.addChild(sumador1_2);
Offset sumador1_2Offset = sumador1_2.getRelativeOffset();
sumador1_2Offset.setHorOffset(Gran.CLB, 2);
sumador1_2Offset.setVerOffset(Gran.CLB, 0);
sumador1_2Offset.setHorOffset(Gran.SLICE, 1);
sumador1_2.implement();

Adder sumador2_1 = new Adder("SUMADOR2_1",data1_2,data3_2,data1_3);
this.addChild(sumador2_1);
Offset sumador2_1Offset = sumador2_1.getRelativeOffset();
sumador2_1Offset.setHorOffset(Gran.CLB, 3);
sumador2_1Offset.setVerOffset(Gran.CLB, 0);
sumador2_1Offset.setHorOffset(Gran.SLICE, 0);
sumador2_1.implement();

/* Module Core */
ModuleUL modulo = new ModuleUL("modulo",dina,dinb,mod_out);
this.addChild(modulo);
Offset moduloOffset = modulo.getRelativeOffset();
moduloOffset.setHorOffset(Gran.CLB, 4);
moduloOffset.setVerOffset(Gran.CLB, 0);
modulo.implement();

/* Constant Core (1-K) */
this.Kconstante = new Constant("Kconstante",constante_bus);
this.addChild(this.Kconstante);
Offset KconstanteOffset = this.Kconstante.getRelativeOffset();
KconstanteOffset.setHorOffset(Gran.CLB, 3);
KconstanteOffset.setVerOffset(Gran.CLB, 0);
KconstanteOffset.setHorOffset(Gran.SLICE, 1);
this.Kconstante.implement(1-constante);

/* ConstantComparator Core */
ConstantComparatorParameters parametros = new ConstantComparatorParameters();
parametros.setIn_din(addr);
parametros.setConstant(0);
parametros.setMode(ConstantComparatorParameters.EQUALS);
parametros.setOut_dout(select);
ConstantComparator comparador = new ConstantComparator("comparador",parametros);
this.addChild(comparador);
```

```
    Offset comparadorOffset = comparador.getRelativeOffset();
    comparadorOffset.setHorOffset(Gran.CLB, 3);
    comparadorOffset.setVerOffset(Gran.CLB, 8);
    comparadorOffset.setHorOffset(Gran.SLICE, 1);
    comparador.implement();

    /* MUX2_1 Core */
    MUX2_1[] multiplexor = new MUX2_1[16];
    for(i = 0; i <16; i++) {
       multiplexor[i] = new MUX2_1("multiplexor"+i,
                    mod_out.getNet(i),constante_bus.getNet(i),select,dout.getNet(i));
       this.addChild(multiplexor[i]);
    }

    Offset[] multiplexorOffset = new Offset[16];
    int slice = 0;
    int le = 0;
    int col = 5;
    int row = 8;

    for(i = 0; i <16; i++) {
       if(le == 2) { slice++; le = 0; }
       if(slice == 2) { slice = 0; row++; }

       multiplexorOffset[i] = multiplexor[i].getRelativeOffset();
       multiplexorOffset[i].setHorOffset(Gran.CLB, col);
       multiplexorOffset[i].setVerOffset(Gran.CLB, row);
       multiplexorOffset[i].setHorOffset(Gran.SLICE, slice);
       multiplexorOffset[i].setVerOffset(Gran.LE, le++);
       multiplexor[i].implement();
    }

    /* Connect the Nets/Buses */
    Bitstream.connect(data[0]);
    Bitstream.connect(data2_1);
    Bitstream.connect(data1_2);
    Bitstream.connect(data[2]);
    Bitstream.connect(data3_2);
    Bitstream.connect(data4_1);
    Bitstream.connect(constante_bus);
    Bitstream.connect(select);
    Bitstream.connect(mod_out);
    Bitstream.connect(dina);
    Bitstream.connect(dinb);

    tagCore_mult(offset, 0x0800);
}

public final void implement_resto(int constante) throws CoreException {

    int i, k, l;
    Offset offset = calcAbsoluteOffset();
```

```
    Bus dina = newBus("dina",16);
    dinPort.setIntSig(dina);

    Bus dinb = new Bus("dinb",null,16);
    Net cero = new Net("cero",null);

    Bus dout = newBus("dout",16);
    doutPort.setIntSig(dout);

    /* Interconexion de buses */
    for(i = 1; i <16; i++)
       dinb.setNet(i,cero);

    /* Constant Core */
    Constant constant = new Constant("constant",dinb.getNet(0));
    this.addChild(constant);
    Offset constantOffset = constant.getRelativeOffset();
    constantOffset.setHorOffset(Gran.CLB, 0);
    constantOffset.setVerOffset(Gran.CLB, 0);
    constantOffset.setHorOffset(Gran.SLICE, 0);
    constantOffset.setVerOffset(Gran.LE, 0);
    constant.implement(constante);

    /* Gnd Core */
    Gnd tierra = new Gnd("tierra",cero);
    this.addChild(tierra);
    Offset tierraOffset = tierra.getRelativeOffset();
    tierraOffset.setHorOffset(Gran.CLB, 0);
    tierraOffset.setVerOffset(Gran.CLB, 0);
    tierra.implement();

    /* Subtracter Core */
    Subtracter resta = new Subtracter("resta", dinb, dina, dout);
    this.addChild(resta);
    Offset restaOffset = resta.getRelativeOffset();
    restaOffset.setHorOffset(Gran.CLB, 0);
    restaOffset.setVerOffset(Gran.CLB, 0);
    restaOffset.setHorOffset(Gran.SLICE, 1);
    resta.implement();

    /* Connect the Nets/Buses */
    Bitstream.connect(dinb);

    tagCore_resto(offset, 0x0800);
}

private void tagCore_mult(Offset offset, int tag) throws CoreException {

    /* get row, column, and slice offsets */
    int row = offset.getVerOffset(Gran.CLB);
    int col = offset.getHorOffset(Gran.CLB);

    try {
```

```
      for(int i = col; i <6+col; i++) {
         for(int j = row; j <12+row; j++) {
            if(i <3+col && j >9+row) {}
            else
               Bitstream.getVirtex().setTag(j, i, tag);
         }
      }
      Bitstream.getVirtex().setTag(row+12, col+4, tag);
   } catch (ConfigurationException ce) {
       throw new CoreException(ce);
   }
}

private void tagCore_resto(Offset offset, int tag) throws CoreException {

   /* get row, column, and slice offsets */
   int row = offset.getVerOffset(Gran.CLB);
   int col = offset.getHorOffset(Gran.CLB);

   try {
      for(int j = row; j <8+row; j++) {
         Bitstream.getVirtex().setTag(j, col, tag);
      }
   } catch (ConfigurationException ce) {
      throw new CoreException(ce);
   }
}

public static int calcHeightGran() {
   return Gran.CLB;
}

public static int calcWidthGran() {
   return Gran.CLB;
}

public static int calcHeight() {
   return 13;
}

public static int calcWidth() {
   return 6;
}

private void
checkParameters(int busawidth, int busbwidth) throws CoreParameterException {
   if ((busawidth != busbwidth) || ((busawidth% 4) != 0))
      throw new CoreParameterException(this, "Buses width error");
}

private void cambiar_memoria(int constante) throws CoreException {
   if(this.memoria == null)
      System.out.println("La memoria no esta asociada");
```

```
        else {
            this.memoria.implement(constante);
            this.Kconstante.update(1-constante);
        }
    }

    /**
     * Input port.
     */
    private Port dinPort;

    /**
     * Output port.
     */
    private Port doutPort;

    /**
     * Width.
     */
    private int busWidth;

    private MemoriaUL memoria;
    private Constant Kconstante;

    private int last_constant = 0xFF;
    private boolean first = true;
}
```

Bibliografía

[1] H. Chang, L. Cooke, M. Hunt, G. Martin, A. J. McNelly, and L. Todd, *Surviving the SOC revolution: a guide to platform-based design.* Kluwer Academic Publishers, 1999. ISBN:0-7923-8679-5.

[2] R. Rajsuman, "System-on-a-Chip. Design and Test," *Artech House Publishers*, 2000.

[3] J. Becker, "Configurable Systems-on-Chip (CSoC)," *15th Symposium on Integrated Circuits and System Design (SBCCI2002)*, pp. 379–384, Sept. 2002.

[4] S. Knapp and D. Tavana, "Field Configurable System-on-Chip Device Architecture," *Custom Integrated Circuits Conf.*, pp. 155–158, 2000.

[5] Grant Martin and Henry Chang, ed., *Winning the SoC Revolution : Experiences in Real Design.* Massachusetts, USA: Kluwer Academic Publishers, 2003.

[6] Altera Corporation, "Nios Embedded Processor System Development." Disponible Online: http://www.altera.com/products/ip/processors/nios/nio-index.html.

[7] Xilinx Inc., *MicroBlaze Processor Reference Guide.*

[8] Jiri Gaisler, *LEON2 Processor User's Manual, XST edition, version 1.0.24.* Gaisler Research, Disponible Online en: http://www.gaisler.com/doc/leon2-1.0.24-xst.pdf, 2004.

[9] Damjam Lampret, *OpenRISC 1200 IP Core Specification*, Sept. 2001.

[10] "Triscend A7S 32-bit Customizable System-on-Chip Platform," datasheet, Triscend Corporation, Disponible Online: http://www.triscend.com/products/Documentation/dsa7csoc.pdf.

[11] "Triscend E5 Customizable Microcontroller Family - Full Version," datasheet, Triscend Corporation, Disponible Online: http://www.triscend.com/products/Documentation/e5_datasheet_mar2003.pdf.

[12] *Virtex-II Complete Data Sheet.* Disponible Online: http://direct.xilinx.com/bvdocs/publications/ds031.pdf, 20 Nov. 2004.

[13] Altera Corporation, Disponible Online: http://www.altera.com/literature/ds/ds_arm.pdf, *Excalibur Device Overview*, 20 Nov. 2004.

[14] J. Tuley, "Soft Computing Reconfigures Designer Options," *Embedded Systems*, p. 76, Apr. 1997.

[15] P. Lysaght and J. Dunlop, "Dynamic Reconfiguration of FPGAs," *International Workshop on Field Programmable Logic and Applications*, pp. 82–94, 1993.

[16] B. Blodget, P. James-Roxby, and E. Keller, "Self-reconfiguring Platform," *Proceedings of the 13th International Workshop on Field-Programmable Logic and Applications (FPL'03)*, vol. LNCS 2778, pp. 565–574, Sept. 2003.

[17] "XC6200 Field Programmable Gate Arrays," tech. rep., Xilinx Inc., 2002.

[18] A. Donlin, "Self Modifying Circuitry - A Platform for Tractable Virtual Circuitry," *Field-Programmable Logic: From FPGAs to Computing Paradigm*, pp. 200–208, 1998.

[19] "Virtex 2.5V Field Programmable Gate Arrays Product Specification," tech. rep., Xilinx Inc., 2001.

[20] "Two Flows for Partial Reconfiguration: Module Based or Difference Based," Application Note 290, Xilinx Inc., http://www.xilinx.com.

[21] M. Dyer, C. Plessl, and M. Platzner, "Partially Reconfigurable Cores for Xilinx Virtex," *Lecture Notes in Computer Science*, vol. 2438, pp. 292–301, 2002.

[22] R. J. Fong, S. J. Harper, and P. M. Athanas, "A Versatile Framework for FPGA Field Updates: An Application of Partial Self-Reconfiguration," *Proceedings of the 14th IEEE International Workshop on Rapid Systems Prototyping (RSP'03)*, pp. 117–123, 2003.

[23] K. Danne, C. Bobda, and H. Kalte, "Run-Time Exchange of Mechatronic Controllers Using Partial Hardware Reconfiguration," *Field-Programmable Logic and Applications, FPL 2003*, vol. LNCS 2778, pp. 272–281, 2003.

[24] M. Dyer and M. Wirz, "Reconfigurable System on FPGA," Master's thesis, Swiss Federal Institute of Technology Zurich, 2002.

[25] F. Braun, J. W. Lockwood, and M. Waldvogel, "Layered Protocol Wrappers for Internet Packet Processing in Reconfigurable Hardware," Tech. Rep. WUCS-01-10, Applied Research Lab. Department of Computer Science. Washignton University, July 2001.

[26] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor, "Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX)," *ACM International Symposium in Field Programmable Gate Arrays (FPGA'2001)*, pp. 87–93, Feb. 2001.

[27] F. Braun, M. Waldvogel, and J. Lockwood, "OBIWAN - an Internet Protocol Router in Reconfigurable Hardware," Tech. Rep. WUCS-01-11, Applied Research Lab. Department of Computer Science. Washignton University, July 2001.

[28] J. W. Lockwood, C. Neely, and C. Zuver, "An Extensible, System-On-Programmable-Chip, Content-Aware Internet Firewall," *Field-Programmable Logic and Applications, FPL 2003*, vol. LNCS 2778, pp. 859–868, Sept. 2003.

[29] R. Sidhu and V. Prasanna, "Fast Regular Expression Matching using FPGAs," *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2001)*, pp. 698–709, 2001.

[30] R. Sidhu and V. Prasanna, "Efficient Metacomputation using Self-Reconfiguration," *Field-Programmable Logic and Applications, FPL 2002*, vol. LNCS 2438, pp. 698–709, Sept. 2002.

[31] M. Ullmann, M. HÄubner, B. Grimm, and J. Becker, "On-Demand FPGA Run-Time System for Dynamical Reconfiguration with Adaptive Priorities," *Field-Programmable Logic and Applications, FPL 2004*, vol. LNCS 3203, pp. 454–463, Aug. 2004.

[32] M. Ullmann, M. HÄubner, B. Grimm, and J. Becker, "An FPGA Run-Time System for Dynamical On-Demand Reconfiguration," *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW/IPDPS)*, Apr. 2004.

[33] L. Möller, N. Calazans, F. G. Moraes, E. Brião, E. Carvalho, and D. Camozzato, "FiPRe: An Implementation Model to Enable Self-Reconfigurable Applications," *Field-Programmable Logic and Applications, FPL 2004*, vol. LNCS 3203, pp. 1042–1046, Aug. 2004.

[34] J. Palma, A. V. de Mello, F. G. Moraes, and N. Calazans, "Core Communication Interface for FPGAs," June 2002.

[35] B. Blodget, S. McMillan, and P. Lysaght, "A Lightweight Approach for Embedded Reconfiguration of FPGAs," *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, pp. 399–400, 2003.

[36] F. G. Moraes, D. Mesquita, J. C. Palma, L. Möller, and N. Calazans, "Development of a Tool-Set for Remote and Partial Reconguration of FPGAs," *Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, vol. IEEE Computer Society Press, pp. 11122–11123, Feb. 2003.

[37] H. Walder and M. Platzner, "Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations," *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, vol. CSREA Press, pp. 284–287, June 2003.

[38] H. Walder and M. Platzner, "A Runtime Environment for Reconfigurable Hardware Operating Systems," *Field-Programmable Logic and Applications, FPL 2004*, vol. LNCS 3203, pp. 831–835, Aug. 2004.

[39] H. Walder, S. Nobs, and M. Platzner, "XF-Board: A Prototyping Platform for Reconfigurable Hardware Operating Systems," *3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, vol. CSREA Press, p. 306, June 2004.

[40] C. Steiger, H. Walder, and M. Platzner, "Heuristics for Online Scheduling Real-Time Tasks to Partially Reconfigurable Devices," *Field-Programmable Logic and Applications, FPL 2003*, vol. LNCS 2778, pp. 575–584, Sept. 2003.

[41] A. Segard and F. Verdier, "SOC and RTOS: Managing IPs and Tasks Communications," *Field-Programmable Logic and Applications, FPL 2004*, vol. LNCS 3203, pp. 710–718, Aug. 2004.

[42] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol," RFC 2401, Corporation for National Research Initiatives, Internet Engineering Task Force, Network Working Group, Reston, Virginia, USA, 1998.

[43] J. Wollinger, J. Guajardo, and C. Paar, "Cryptography in Embedded Systems: An Overview," *Proc. of the Embedded World 2003 Exhibition and Conference*, vol. Design & Elektronik, pp. 735–744, 2003.

[44] T. Wollinger, J. Guajardo, and C. Paar, "Security on FPGAs: State-of-the-Art Implementations and Attacks," *ACM Transactions in Embedded Computing Systems*, vol. 3, pp. 534–574, Aug. 2004.

[45] A. J. Elbirt, W. Yip, and B. Chetwynd, "An FPGA Based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, pp. 545–557, Aug. 2001.

[46] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in Embedded Systems: Design Challenges," *ACM Transactions on Embedded Computing Systems*, vol. 3, pp. 461–491, Aug. 2004.

[47] C. Teuscher, J. Haenni, F. J. Gomez, and E. Sanchez, "Labomat 3: A Reconfigurable Platform for Academic Purposes," *Proceedings of the IEEE Symposium Field-Programmable Custom Computing Machines, FCCM'99*, Apr. 1999.

[48] "uCLinux - Embedded Linux/Microcontroller Project." Home page: http:// www.uclinux.org.

[49] "Microblaze uClinux Project." Home page: http://www.itee.uq.edu.au/ jwilliams/mblaze-uclinux/.

[50] A. DeHon, "DPGA Coupled Microprocessors: Commodity ICs for the Early 21st Century," *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'94)*, pp. 31–39, Apr. 1994.

[51] A. DeHon and J. Wawrzynek, "The Case for Reconfigurable Processors," preliminary copy for berkeley reconfigurable architectures, systems, and software group, University of California at Berkeley, Computer Science Division, 31 Jan. 1997.

[52] T. J. Moeller and D. R. Martinez, "Field Programmable Gate Array Based Radar Front-End Digital Signal Processing," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*, pp. 178–187, Apr. 1999.

[53] D. R. Martinez, T. J. Moeller, and K. Teitelbaum, "Application of Reconfigurable Computing to a High Performance Front-End Radar Signal Processor," *The Journal of VLSI Signal Processing*, vol. 28, pp. 63–83, May 2001.

[54] K. Bondalapati and V. K. Prasanna, "Reconfigurable Computing Systems," *Proceedings of the IEEE*, vol. 90, pp. 1201–1217, July 2002.

[55] S. Casselman, "Virtual Computing and The Virtual Computer," *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'93)*, pp. 43–48, Apr. 1993.

[56] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys*, vol. 34, pp. 171–210, June 2002.

[57] S. M. Scalera and J. R. Vazquez, "The Design and Implementation of a Context Switching FPGA," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'98)*, pp. 78–85., Apr. 1998.

[58] "Virtex FPGA Series Configuration and Readback," tech. rep., Xilinx Inc., 2002.

[59] "Virtex Series Conguration Architecture User Guide," tech. rep., Xilinx Inc., 2003.

[60] B. L. Hutchings and M. J. Wirthlin, "Implementation Approaches for Reconfigurable Logic Applications," *Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications (FPL'95)*, vol. 419-428, Aug. 1995.

[61] F. Barat and R. Lauwereins, "Reconfigurable Instruction Set Processors: A Survey," *Proceedings of the 11th International Workshop on Rapid System Prototyping (RSP'00)*, pp. 168–173, June 2000.

[62] R. Hartenstein, M. Herz, and T. Hoffmann, "On Reconfigurable Co-Processing Units," *Proceedings of the 1998 Fifth Reconfigurable Architectures Workshop (RAW'98)*, pp. 67–72, 30 Mar. 1998.

[63] S. Hauck, T. W. Fry, and M. M. Hosler, "The Chimaera Reconfigurable Functional Unit," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, pp. 206–217, Feb. 2004.

[64] D. A. Buel, J. Arnold, and W. Kleinfelder, "Splash 2: FPGAs in a Custom Computing Machine," 1996.

[65] R. Keaney, L. C. Lee, and D. J. Skellern, "Implementation of Long Constraint Length Viterbi Decoders using Programmable Active Memories," *11th Australian Microelectronics*, Oct. 1993.

[66] P. M. Athanas and H. F. Silverman, "Processor Reconfiguration Through Instruction-Set Metamorphosis," *IEEE Computer*, vol. 26, pp. 11–18, 18 Mar. 1993.

[67] N. Dutt and K. Choi, "Configurable Processors for Embedded Computing," *IEEE Computer*, vol. 36, pp. 120–123, Jan. 2003.

[68] D. Andrews, D. Niehaus, and P. Ashenden, "Programming Models for Hybrid CPU/FPGA Chips," *IEEE Computer*, vol. 37, pp. 118–120, Jan. 2004.

[69] R. Enzler, M. Platzner, and C. Plessl, "Reconfigurable Processors for Handhelds and Wearables: Application Analysis," *Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communications III*, vol. 4525 of Proceedings of SPIE, pp. 135–146, Aug. 2001.

[70] A. Lawrence, A. Kay, and W. Luk, "Using Reconfigurable Hardware to Speed up Product Development and Performance," *Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications (FPL'95)*, pp. 111–118, Sept. 2005.

[71] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, pp. 12–21, Apr. 1997.

[72] J. R. Hauser, *Augmenting a Microprocessor with Reconfigurable Hardware*. PhD thesis, University of California, Berkeley, 2000.

[73] E. Sanchez, J.-O. Haenni, J.-L. Beuchat, A. Stauffer, A. Perez-Uribe, and M. Sipper, "Static and Dynamic Configurable Systems," *IEEE Transactions on Computers*, vol. 48, no. 6, pp. 556–564, 1999.

[74] M. J. Wirthlin, B. L. Hutchings, and K. L. Gilson, "The Nano Processor: a Low Resource Reconfigurable Processor," *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'94)*, pp. 23–30, Apr. 1994.

[75] M. J. Wirthlin and B. L. Hutchings, "DISC: A Dynamic Instruction Set Computer," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'95)*, pp. 99–107, Apr. 1995.

[76] M. J. Wirthlin and B. L. Hutchings, "Sequencing Run-Time Reconfigured Hardware with Software," *ACM/SIGMA Internaltinal Symposium on FPGA*, pp. 122–128, 1996.

[77] M.-H. Lee, H. Singh, G. Lu, N. Bagherzadeh, F. J. Kurdahi, E. M. C. Filho, and V. C. Alves, "Design and Implementation of the MorphoSys Reconfigurable Computing Processor," *The Journal of VLSI Signal Processing*, vol. 24, pp. 147–164, Mar. 2000.

[78] R. D. Wittig, *OneChip: An FPGA Processor With Reconfigurable Logic*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, 1995.

[79] R. D. Wittig and P. Chow, "OneChip: An FPGA Processor With Reconfigurable Logic," *IEEE Workshop on FPGAs for Custom Computing Machines, IEEE Computer Society*, Apr. 1996.

[80] J. E. Carrillo and P. Chow, "The Effect of Reconfigurable Units in Superscalar Processors," *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field-Programmable Gate Arrays (FPGA'01)*, pp. 141–150, Feb. 2001.

[81] M. Dales, "Initial Analysis of the Proteus Architecture," *Proceedings of the 11th International Workshop on Field-Programmable Logic and Applications (FPL'01)*, pp. 623–627, Aug. 2001.

[82] S. Vassiliadis, S. Wong, and G. Gaydadjiev, "The MOLEN Polymorphic Processor," *IEEE Transactions on Computers*, vol. 53, pp. 1363–1375, Nov. 2004.

[83] S. Seng, W. Luk, and P. Cheung, "Flexible Instruction Processors," *CASES, ACM*, pp. 193–200, 2000.

[84] S. Seng, W. Luk, and P. Y. Cheung, "Run-Time Adaptive Flexible Instruction Processors," *Proceedings of the 12th International Workshop on Field-Programmable Logic and Applications (FPL'02)*, pp. 545–555, Sept. 2002.

[85] J. Faura, M. Aguirre, and J. Moreno, "FIPSOC: A Field Programmable System On a Chip," *Proceedings of the Design of Circuits and Integrated Systems, DCIS'97*, 1997.

[86] Pleaides Group, "Pleaides," tech. rep., U. Berkeley, http://bwrc.eecs.berkeley.edu/Research/Configurable_Architectures.

[87] H. Zhang, V. Prabhu, and V. George, "A 1V Heterogeneous Reconfigurable Processor IC for Baseband Wireless Applications," *Proceedings of ISSCC2000*, 2000.

[88] J. Becker, T. Pionteck, and M. Glesner, "An Application-tailored Dynamically Reconfigurable Hardware Architecture for Digital Baseband Processing," *Proceedings of XIII Brazilian Symposium on Integrated Circuit Design (XIII SBCCI)*, Sept. 2000.

[89] J. Becker, N. Liebau, T. Pionteck, and M. Glesner, "Eficient Mapping of pre-synthesized IP-Cores onto Dynamically Reconfigurable Array Architectures," *Proceedings 11th Int. Conference on Field Programmable Logic and Applications, FPL'01*, 2001.

[90] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP - A Self-Reconfigurable Data Processing Architecture," *The Journal of Supercomputing*, vol. 26, pp. 167–184.

[91] "The XPP Communication System," Tech. Rep. 15, PACT Corp, http://www.pactcorp.com, 2000.

[92] T. Halfhill, "MIPS embraces configurable technology," *Microprocessor Report*, pp. 7–15, Mar. 2003.

[93] R. Lysecky and F. Vahid, "A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning," *Proc. Design, Automation and Test in Europe (DATE'05)*, pp. 18–23, Mar. 2005.

[94] G. McGregor and P. Lysaght, "Self Controlling Dynamic Reconfiguration: A Case Study," *Lecture Notes in Computer Science*, pp. 144–154, 1999.

[95] N. Shirazi, W. Luk, and P. Y. K. Cheung, "Automating Production of Run-Time Reconfigurable Designs," *IEEE Symposium on FPGAs for Custom Computing Machines*, vol. IEEE Computer Society Press, pp. 147–156, 1998.

[96] W. Luk, N. Shirazi, and P. Cheung, "Compilation tools for run-time reconfigurable designs," *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 1997)*, vol. IEEE Computer Society Press, pp. 56–65, 1997.

[97] W. Luk, N. Shirazi, and P. Cheung, "Modelling and Optimizing Run-Time Reconfigurable Systems," *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 1996)*, vol. IEEE Computer Society Press, pp. 167–176, 1996.

[98] J. D. Hadley and B. L. Hutchings, "Design Methodologies for Partially Reconfigured Systems," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'95)*, pp. 78–84, Apr. 1995.

[99] M. Vasilko, "Design Visualisation for Dynamically Reconfigurable Systems," *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications (FPL'00)*, pp. 131–140, Aug. 2000.

[100] P. Bellows and B. Hutchings, "JHDL - an HDL for Reconfigurable Systems," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'98)*, pp. 175–181, Apr. 1998.

[101] S. A. Guccione and D. Levi, "JBits: A Java-Based Interface to FPGA Hardware," tech. rep., Xilinx Corporation, San Jose, CA, USA. Disponible Online en http://www.io.com/guccione/Papers/Papers.html.

[102] S. Guccione, D. Levi, and P. Sundararajan, "JBits: Java-Based Interface for Reconfigurable Computing," *Proceedings of Second Annual Military and Aerospace Applications of Programmable Devices and Technologies (MAPLD'99)*, Sept. 1999.

[103] E. L. Horta, J. W. Lockwood, and S. T. Kofuji, "Using PARBIT to Implement Partial Run-Time Reconfigurable Systems," *Proceedings of the 12th International Workshop on Field-Programmable Logic and Applications (FPL'02)*, pp. 182–191, Sept. 2002.

[104] G. Brebner and A. Donlin, "Runtime Reconfigurable Routing," *Parallel and Distributed Processing (IPPS/SPDP'98)*, vol. LNCS 1388, pp. 25–30, 1998.

[105] Z. Li, K. Compton, and S. Hauck, "Configuration Caching Techniques for FPGA," *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, 2000.

[106] K. Compton, Z. Li, and J. Cooey, "Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing," *IEEE Transactions on VLSI Systems*, vol. 10, no. 3, pp. 209–220, 2002.

[107] E. Lechner and S. Guccione, "The Java environment for recongurable computing," *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, FPL 1997*, vol. LNCS 1304, pp. 284–293, 1997.

[108] S. A. Guccione and D. Levi, "XBI: A Java-based interface to FPGA Hardware," *Proceedings SPIE Photonics East*, pp. 97–102, 1998.

[109] S. A. Guccione and D. Levi, "Run-Time Parameterizable Cores," *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications (FPL'99)*, vol. LNCS 1673, pp. 215–222, Aug. 1999.

[110] E. Keller, "JRoute: A Run-Time Routing API for FPGA Hardware," *7th Reconfigurable Architectures Workshop*, vol. Lecture Notes in Computer Science 1800, pp. 874–881, May 2000.

[111] S. McMillan and S. A. Guccione, "Partial Run-Time Reconfiguration Using JRTR," *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications (FPL'00)*, pp. 352–360, Aug. 2000.

[112] P. James-Roxby and S. A. Guccione, "Automated Extraction of Run-Time Parameterisable Cores from Programmable Device Configurations," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, pp. 153–161, Apr. 2000.

[113] A. Rincon, C. Cherichetti, and J. Monzel, "Core Design and System-on-a-Chip Integration," *IEEE Design & Test of Computers*, pp. 26–35, Oct. 1997.

[114] A. Astarloa, U. Bidarte, A. Zuloaga, and J. Jimenez, "Run-time Reconfigurable Hybrid Multiprocessor Cores," *Proceedings of the 2004 IEEE Internacional Conference on Industrial Technology*, vol. IEEE Society Press, Dec. 2004.

[115] K. Vissers, "Parallel Processing Architectures for Reconfigurable Systems," *DATE*, vol. IEEE Computer Society Press, pp. 10396–10397, 2003.

[116] D. Taylor, J. Turner, J. W. Lookwood, and E. L. Horta, "Dynamic Hardware Plugins (DHP): Exploiting recongurable hardware for high-performance programmable routers," *Computer Networks*, vol. 38, pp. 295–310, Feb. 2002.

[117] E. L. Horta and J. W. Lookwood, "PARBIT: A Tool to Transform BitFiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs)," tech. rep., Applied Research Lab. Department of Computer Science. Washignton University, July 2001.

[118] E. L. Horta, J. W. Lockwood, and D. E. Taylor, "Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration," *Design Automation Conference (DAC)*, June 2002.

[119] D. W. Jones, "The Ultimate RISC," *ACM Computer Architecture News*, vol. 16, pp. 48–55, June 1988.

[120] "FPSLIC on-chip Partial Reconguration of the Embedded AT40K FPGA," atmel application note, Atmel Inc., 2002.

[121] S. Young, P. Alfke, and C. Fewer, "A High I/O Reconfigurable Crossbar Switch," *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2003)*, pp. 3–11, 2003.

[122] M. HÄubner, M. Ullmann, and L. Braun, "Scalable Application-Dependent Network on Chip Adaptivity for Dynamical Reconfigurable Real-Time Systems," *Field-Programmable Logic and Applications, FPL 2004*, vol. LNCS 3203, pp. 1037–1041, Aug. 2004.

[123] F. G. Morares and N. L. V. Calazans, "R8 Processor Architecture and Organization Specification and Design Guidelines," tech. rep., http://www.inf.pucrs.br/ graph/Projects/R8/pulbic/R8_arq_apec_eng.pdf, Sept. 1996.

[124] F.-X. Standaert, G. Rouvroy, and J.-J. Quisquater, "Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs," *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, pp. 334–350, Sept. 2003.

[125] A. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, Florida, USA: CRC Press, 1997.

[126] B. Schneier, *Applied Cryptography*. 1996.

[127] William Stallings, *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 3rd edition ed., Aug. 2002.

[128] NIST, "FIPS 46-3, Data Encryption Standard (DES)," tech. rep., U.S. Department of Commerce/National Institute of Standards and Technology, Disponible Online: http://csrc.nist.gov/CryptoToolkit/tkencryption.html, Mar. 1999.

[129] X. Lai and J. L. Massey, "A Proposal for a New Block Encryption Standard," *Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology*, pp. 389–404, May 1990.

[130] NIST, "FIPS 197, Advanced Encryption Standard (AES)," tech. rep., U.S. Department of Commerce/National Institute of Standards and Technology, Disponible Online: http://csrc.nist.gov/CryptoToolkit/tkencryption.html, Nov. 2001.

[131] N. Weaver and E. Caspi, "IDEA as a benchmark for reconfigurable computing," technical report, University of California at Berkeley, BRASS Research Group, Berkeley, CA, Dec. 1996.

[132] E. Biham and A. Shamir, "Differential Cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer," *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '91)*, vol. LNCS 576, pp. 156–171, 1991.

[133] NIST, "SP 800-67, Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher," tech. rep., U.S. Department of Commerce/National Institute of Standards and Technology, Disponible Online: http://csrc.nist.gov/CryptoToolkit/tkencryption.html, May 2004.

[134] B. Schneier, "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)," *Fast Software Encryption, Cambridge Security Workshop*, vol. LNCS 809, pp. 191–204, Dec. 1999. Disponible Online en: http://www.schneier.com/paper-blowfish-fse.html.

[135] J. Daemen and V. Rijmen, "AES Proposal: Rijndael," tech. rep., NIST AES Proposal, June 1998.

[136] J. Daemen and V. Rijmen, "The Design of Rijndael: AES-The Advanced Encryption Standard," *Springer-Verlag*, 2002. ISBN 3-540-42580-2.

[137] R. L. Rivest, "The RC4 Encryption Algorithm," tech. rep., RSA Data Security, Mar. 1992.

[138] R. L. Rivest, "The MD5 Message Digest Algorithm," RFC 1321, MIT LCS & RSA Data Security, Apr. 1992.

[139] NIST, "FIPS 180-2, Secure Hash Standard (SHS)," tech. rep., U.S. Department of Commerce/National Institute of Standards and Technology, Disponible Online en: http://csrc.nist.gov/CryptoToolkit/tkhash.html, Aug. 2002.

[140] A. O. Freier, P. Karlton, and P. C. Kocher, "The SSL Protocol Version 3.0," internet-draft, Transport Layer Security Working Group, 1996.

[141] R. R. Taylor and S. C. Goldstein, "A High-Performance Flexible Architecture for Cryptography," *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems (CHES'99)*, pp. 231–245, Aug. 1999.

[142] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," RFC 2246, Corporation for National Research Initiatives, Internet Engineering Task Force, Network Working Group, Virginia, USA, 1999.

[143] L. Batina, S. B. Örs, B. Preneel, and J. Vandewalle, "Hardware Architectures for Public Key Cryptography," *Integration, the VLSI Journal*, vol. 34, pp. 1–64, 2003.

[144] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, Feb. 1978.

[145] V. Miller, "Uses of elliptic curves in cryptography," *Advances in Cryptology (CRYPTO '85)*, vol. LNCS 218, pp. 417–426, 1986.

[146] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203–209, 1987.

[147] C. Patterson, "High Performance DES Encryption in Virtex FPGAs using JBits," *IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM'00*, pp. 113–121, Apr. 2000.

[148] S. L. C. Salomao, J. M. S. de Alcantara, V. C. Alves, and F. M. G. Franca, "Improved IDEA," *Proceedings 13th Symposium on Integrated Circuits and Systems Design*, pp. 47–52, Sept. 2000.

[149] A. V. Curiger, H. BonnenBerg, and R. Zimmermann, "VINCI: VLSI Implementation of the New Block Cipher IDEA," *Proceedings of the IEEE CICC'93*, pp. 15.5.1–15.5.4, May 1993.

[150] "IDEACrypt," tech. rep., Ascom, http://www.ascom.com/.

[151] O. Mencer, M. Morf, and M. J. Flynn, "Hardware Software Tri-Design of Encryption for Mobile Communication Units," *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, pp. 3045–3048, 1998.

[152] S. L. C. Salomao, V. C. Alves, and E. M. C. Filho, "iPCrypto: A High-Performance VLSI Cryptographic Chip," *Proceedings of the Eleventh Annual IEEE ASIC Conference*, pp. 7–11, 1998.

[153] M. Leong, O. Cheung, and K. Tsoi, "A Bit-Serial Implementation of the International Data Encryption Algorithm IDEA," *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, pp. 122–131, Apr. 2000.

[154] E. Mosanya, C. Teuscher, H. F. Restrepo, P. Galley, and E. Sanchez, "CryptoBooster: A Reconfigurable and Modular Cryptographic Coprocessor," *Proc. of the First International Workshop on Cryptographic Hardware and Embedded System, CHES'99*, vol. LNCS 1717, pp. 246–256, Aug. 1999.

[155] J. L. Beuchat, J. O. Haenni, C. Teuscher, F. J. Gomez, H. F. Restrepo, and E. Sanchez, "Approches matérielles et logicielles de l'algorithme IDEA," *Technique et Science Informatiques*, vol. 21, no. 2, pp. 203–204, 2002.

[156] O. Cheung, K. Tsoi, and P. H. W. Leong, "Tradeoffs in Parallel and Serial Implementations of the International Data Encryption Algorithm IDEA," *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, pp. 333–347, May 2001.

[157] A. Hämäläin, M. Tommiska, and J. Skyttä, "6.78 Gigabits per Second Implementation of the IDEA Cryptographic Algorithm," *12th International Conference, FPL 2002*, pp. 760–769, Sept. 2002.

[158] J. O. Haenni, *Architecture EPIC et jeux d'instructions multimédias pour application cryptographiques*. PhD thesis, Laussane EPFL, 2002. Tesis 2540(2002) pp. 77-130.

[159] J. O. Haenni, "IDEA implementation on IPF," tech. rep., EPFL, http://www.haenni.info/thesis/tutorials/idea_ia64/performance.html, 25 Mar. 2001.

[160] X. Lai, "On the Design and Security of Block Ciphers," *Dissertation ETH Zürich No. 9752, ETH Series in Information Processing*, vol. 1, 1992. ISBN 3-89191-573-X.

[161] Z. Pan, S. Venkateshwaran, and S. T. Gurumani, "Exploiting Fine-Grain Parallelism of IDEA Using Xilinx FPGA," *Proceedings of the 16th International Conference on Parallel and Distributed Computing Systems (PDCS-2003)*, Aug. 2003.

[162] J. Leonard and W. Magione-Smith, "A Case Study of Partially Evaluated Hardware Circuits: Keyspecific DES," *Seventh International Workshop on Field-Programmable Logic and Applications, FPL '97*, Sept. 1997.

[163] K. Chapman, "Constant Coefficient Multipliers for the XC4000E," Application Note 054, Xilinx Inc., http://www.xilinx.com.

[164] P. James-Roxby and B. J. Blodget, "A study of high-performance reconfigurable constant coefficient multiplier implementations," tech. rep., Xilinx Inc., Disponible Online en http://www.chipcenter.com/pld/pldf085.htm.

[165] "RC1000PP," data sheet, Celoxica (Embedded Solution), http://www.celoxica.com, 1999.

[166] Xilinx Inc., *Xilinx Design Language*, July 2000. HTML document provided with ISE tools.

[167] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A Quantitative Analisys of the Speedup Factors of FPGAs over Processors," *ACM Int'l Symposium on FPGAs*, pp. 162–170, 2004.

[168] Xilinx Inc., Disponible Online en http://www.xilinx.com/ise/embedded/edk6_2docs/mb_ref_guide.pdf, *Embedded System Tools Guide v3.0, Embedded Development Kit, Version 6.2*, June 2004.

[169] "Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL)," Application Note 529, Xilinx Inc., http://www.xilinx.com.

[170] SPARC International Inc., Disponible Online en: http://www.sparc.org/standards/V8.pdf, *The SPARC Architecture Manual, Version 8, Rev. SAV080SI9308*, 1992.

[171] "Gaisler Research, "www.gaisler.com", [accessed: Sept 2004]."

[172] T. Ylonen, "SSH – Secure Login Connections over the Internet," *Proc. 6th USENIX Security Symposium*, pp. 37–42, 1996.

[173] C. Devine, "Crypto :: Source Code." Disponible Online en http://www.cr0.net:8040/code/crypto/.

[174] R. de Moliner, "Implementation of IDEA." Disponible Online en http://www.demoliner.ch/richard/downloads/idea.V1.2.tar.Z.

[175] Hemanth Satyanarayana, "AES128 (OpenCores)." Disponible Online: http://www.opencores.org/projects.cgi/web/aes_crypto_core/overview, 2004.

[176] Wesley J. Landaker, "Free Blowfish VHDL Core." Disponible Online: http://sourceforge.net/projects/blowfishvhdl, 2004.

[177] "High-Speed DES and Triple DES Encryptor/Decryptor," Application Note 270, Xilinx Inc., http://www.xilinx.com.

[178] A. Hodjat and I. Verbauwhede, "Interfacing a High Speed Crypto Accelerator to an Embedded CPU," *IEEE Proc. 38th Asilomar Conference on Signals, Systems, and Computers*, vol. 1, pp. 488–492, Nov. 2004.

[179] F. Rodriguez-Henriquez, N. Saqib, and A. Diaz-Perez, "4.2 Gbits/s single-chip FPGA implementation of AES algorithm," *Electronics Letters*, vol. 39, no. 15, pp. 1115–1116, 2003.

[180] A. Dandalis, V. K. Prasanna, and J. D. P. Rolim, "An Adaptive Cryptographic Engine for IPSec Architectures," *IEEE Symposium on Field-Programming Custom Computing Machines*, 2000.

[181] G. Rouvroy, F.-X. Standaert, and J.-J. Quisquater, "Design Strategies and Modified Descriptions to Optimize Cipher FPGA Implementations: Fast and Compact Results for DES and Triple-DES," *Proceedings of FPL'03*, vol. LNCS 2778, pp. 181–193, 2003.

[182] P. Kitsos, G. Kostopoulos, N. Sklavos, and O. Koufopavlou, "Hardware Implementation of the RC4 Stream Cipher," *46th IEEE Midwest Symposium on Circuits & Systems '03*, 2003.

[183] J. Deepakumara, H. M. Heys, and R. Venkatesan, "FPGA implementation of MD5 hash algorithm," *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, vol. 2, pp. 919–924, 2001.

[184] R. Lien, T. Grembowski, and K. Gaj, "A 1 Gbit/s Partially Unrolled Architecture of Hash Functions SHA-1 and SHA-512," *3th annual RSA Conference*, pp. 324–338, 2004.

[185] R. S. Preissig, "Data Encryption Standard (DES) Implementation on the TMS320C6000," tech. rep., Texas Intruments, 2000. Online: http://www.eetc.globalsources.com/ARTICLES/2001MAY/PDF1/2001MAY23_DSP_CT_AN1365.PDF.

[186] S. L. C. Salomao, M. S. de Alcantara, V. C. Alves, and A. C. C. Vieira, "SCOB, a softcore for the Blowfish cryptographic algorithm," *Proceedings XII Symposium on Integrated Circuits and Systems Design*, pp. 220–223, 1999.

[187] K. K. Ting, S. C. L. Yuen, K. H. Lee, and P. H. W. Leong, "An FPGA Based SHA-256 Processor," *Proc. 12th International Conference FPL 2002*, p. 577, 2002.

[188] J. Burke, J. McDonald, and T. Austin, "Architectural Support for Fast Symmetric-Key Cryptography," *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 178–189, 2002.

[189] K. Nadehara, M. Ikekawa, and I. Kuroda, "Extended instructions for the AES cryptography and their efficient implementation," *IEEE Workshop on Signal Processing Systems (SIPS)*, pp. 152–157, 2004.

[190] P. Huerta, J. Castillo, J. I. Martinez, and V. Lopez, "MicroBlaze based MultiProcessor Soc," *WSEAS TRANSACTIONS on CIRCUITS and SYSTEMS*, vol. 4, May 2005.

[191] P. Huerta, J. Castillo, J. Martinez, and V. Lopez, "Multi-Microblaze System for Parallel Computing," *9th WSEAS CSCC Multiconference*, July 2005.

[192] J. A. Williams and N. W. Bergmann, "Embedded Linux as a platform for dynamically self-reconfiguring systems-onchip," *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '04)*, 2004.

[193] J. Williams and N. Bergmann, "Programmable Parallel Coprocessor Architectures for Reconfigurable System-on-Chip," *International Conference on Field-Programmable Technology*, vol. 1, pp. 193–200, Dec. 2004.

[194] "LEON3MP - a reference LEON3 design with MP support," tech. rep., Gaisler Research, http://www.gaisler.com/products/grlib/docs/designs/leon3mp/index.html.

[195] Jiri Gaisler and Edvin Catovic, "Gaisler Research IP Core's Manual Version 0.16," tech. rep., Gaisler Research.

[196] Jiri Gaisler, Sandi Habinc, and Edvin Catovic, "GRLIB IP Library User's Manual Version 0.16," tech. rep., Gaisler Research.

[197] A. Hodjat and I. Verbauwhede, "High-throughput programmable cryptocoprocessor," *IEEE Micro*, vol. 24, pp. 34–45, May 2004.

[198] M. Oullette and D. Connors, "Analysis of Hardware Acceleration in Reconfigurable Embedded Systems," *Proc. 12th Reconfigurable Architectures Workshop (RAW 2005)*, 2005.

[199] T. Marescaux, A. Bartic, and D. Verkest, "Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs," *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL 2002)*, pp. 795–805, Sept. 2002.

[200] J.-Y. Mignolet, V. Nollet, and D. Verkest, "Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip," *Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, pp. 986–991, Mar. 2003.

[201] G. Haug and W. Rosenstiel, "Reconfigurable Hardware as Shared Resource for Parallel Threads," *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 320–321, Apr. 1998.

[202] J. Becker, M. Huebner, and M. Ullmann, "Power Estimation and Power Measurement of Xilinx Virtex FPGAs: Trade-offs and Limitations," *16th Symposium on Integrated Circuits and System Design (SBCCI 2003)*, Sept. 2003.

[203] "Platform Flash In-System Programmable Configuration PROMs," Data Sheet 123, Xilinx Inc., http://www.xilinx.com, 2005.

[204] Xilinx Inc., http://www.xilinx.com, *Modular Design, ISE 6.3 Development System Reference Guide, pp. 81-118*, 2004.

[205] A. Upegui, R. Moeckel, and E. Dittrich, "An FPGA Dynamically Reconfigurable Framework for Modular Robotics," *Proceedings of the 18th International Conference on Architecture of Computing Systems 2005 (ARCS 2005)*, pp. 83–89.

[206] A. S. Zeineddini and A. S. K. Gaj, "Secure Partial Reconfiguration of FPGAs," *IEEE 2005 Conference on Field-Programmable Technology (FPT' 05)*, Dec. 2005.

[207] "What are the file conversion issues between XDL and NCD databases?," Xilinx Answer Record 17204, Xilinx Inc., http://www.xilinx.com.

[208] Xilinx Inc., *Embedded System Tools Guide.*

[209] D. J. Barrett and R. E. Silverman, *SSH: The Secure Shell The Definitive Guide*. 2001.

[210] T. Ylonen, T. Kivinen, and M. Saarinen, "SSH Protocol Architecture," internet draft, Network Working group, 31 Jan. 2002.

[211] T. Ylonen, T. Kivinen, and M. Saarinen, "SSH Connection Protocol," internet draft, Network Working group, 31 Jan. 2002.

[212] T. Ylonen, T. Kivinen, and M. Saarinen, "SSH Transport Layer Protocol," internet draft, Network Working group, 31 Jan. 2002.

[213] T. Ylonen, T. Kivinen, and M. Saarinen, "Authentication Protocol," *Network Working group*, vol. Internet Draft, 28 Feb. 2002.

[214] "OpenSSH Project." Home page: http://www.openssh.com/.

[215] "OpenSSL Project." Home page: http://www.openssl.org/.

[216] G. Gogniat, T. Wolf, and W. Burleson, "Reconfigurable Security Primitive for Embedded Systems," *Proceedings of International Symposium on System-on-Chip (SOC)*, Nov. 2005.

[217] J. Castillo, P. Huerta, and J. Martínez, "A Secure Self-Reconfiguring Architecture based on Open Source Hardware," *International Conference on Reconfigurable Computing and FPGAs, ReConFig'05*, Sept. 2005.

[218] G. de Micheli and R. K. Gupta, "Hardware/Software Co-Design," *Proceedings of the IEEE*, vol. 85, Mar. 1997.

[219] S. Trimberger, *Kluwer Academic Publishers*. 1994.

[220] "CARMEN Core ARM EmulatioN for Embedded SOC," data sheet, SIDSA, 1999. http://www.sidsa.es.

[221] P. I. Mackinlay, P. Y. Cheung, W. Luck, and R. Sandiford, "Riley-2: A Flexible Platform for Codesign and Dynamic Reconfigurable Computing Research," *Proceedings of Field Programmable Logic and Programmable Logic and Applications, FPL '97*, vol. LNCS 1304, pp. 91–100, 1997.

[222] "Motorola MC68360. "Quad Integrated Communication Controller"," user's manual, Motorola Inc., 1993.

[223] "The Programmable Logic Data Book," tech. rep., Xilinx Inc., 1997. http://www.xilinx.com.

[224] "RTEMS: Real-Time Executive for Multiprocessor Systems," tech. rep. http://www.rtems.com/.

[225] C. Jensen, J. Madsen, and S. Pedersen, "The importance of Interfaces: A HW/SW Codesign Case Study," *Hardware/Software Codesign, 1997. (CODES/CASHE '97), Proceedings of the Fifth International Workshop on*, pp. 87–91, Mar. 1997.

[226] R. Zimmermann, "Efficient VLSI implementation of modulo (2n+1) addition and multiplication," *Proceedings of the 14th IEEE Symposium on Computer Architecture*, pp. 158–167, Apr. 1999.

[227] P. Sundararajan and S. A. Guccione, "XVPI: A Portable Hardware / Software Interface for Virtex," *SPIE - The International Society for Optical Engineering*, vol. Reconfigurable Technology: FPGAs for Computing and Applications II, Proc. SPIE 4212, pp. 90–95, Nov. 2000.

[228] S. P. McMillan, B. J. Blodget, and S. A. Guccione, "VirtexDS: A Device Simulator for Virtex," *SPIE - The International Society for Optical Engineering*, vol. Reconfigurable Technology: FPGAs for Computing and Applications II, Proc. SPIE 4212, pp. 50–56, Nov. 2000.

[229] D. Levi and S. A. Guccione, "BoardScope: A Debug Tool for Reconfigurable Systems," *SPIE - The International Society for Optical Engineering*, vol. Configurable Computing: Technology and Applications, Proc. SPIE 3526, pp. 239–246, Nov. 1998.

Reunido el tribunal que suscribe en el día
de la fecha acordó calificar la presente Tesis
doctoral con SOBRESALIENTE CUM LAUDE
Madrid, 12/05/2006

Firmas

Presidente:
Juan Manuel Sánchez Pérez

Secretario:
José Ignacio Martínez Torre

Vocal 1
Jean Pierre Deschamps

Vocal 2
Eduardo Sánchez Mujica

Vocal 3
Eduardo Bueno Scalvinoni