Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional
Unidad Zacatenco

Departamento de Ingeniería Eléctrica
Sección de Computación

Implementación en hardware-software del algoritmo criptográfico DES [1]

Tesis que presenta:
Mizael Sánchez Santiago

Para obtener el grado de:
Maestro en Ciencias

En la especialidad de:
Ingeniería Eléctrica

Opción:
Computación

Director de tesis:
Dr. Arturo Díaz Pérez

Ciudad de México, DF. Julio de 2003.

[1] Este trabajo fue parcialmente financiado mediante el proyecto CONACyT 31892A: Algoritmos y arquitecturas de computadoras con dispositivos reconfigurables.

Agradecimientos

Quiero agradecerle a mi asesor de tesis, el Dr. Arturo Díaz Pérez, sus conocimientos invaluables que aportaron mucho a esta investigación y sobre todo su gran paciencia para esperar a que este trabajo pudiera llegar a su fin.

Agradezco a los miembros del Jurado, el Dr. Francisco Jose Rodríguez Henríquez y al Dr. Luis Gerardo De la Fraga, por las valiosas contribuciones que hicieron al trabajo final y por el tiempo que dedicaron para revisarlo, aún a pesar de tantas actividades que los ocupan.

Agradezco, también, al Dr. Jorge Buenabad Chávez y al Dr. Arturo Díaz Pérez por el apoyo que recibí de parte de ellos para no truncar mis estudios al inicio del programa. Tengo que agradecerles la confianza que depositaron en mí para continuar con este proyecto. Hoy, solo puedo decirles, "tal vez muy tarde, pero no tan tarde como nunca".

A aquellas grandes personas que hacen posible el conocimiento en las aulas de la sección, los excelentes profesores del programa de Maestria. A mis compañeros de la generación, por todos los buenos y malos momentos que viví con ellos. A todos los que alguna vez han compartido sus conocimientos para enriquecernos a todos.

Al CINVESTAV, ésta gran institución de enorme calidad, que me brindó todo el apoyo durante mi estancia.

Así también, quiero extender mi agradecimiento a los directivos de la Escuela Preparatoria Oficial No. 95, por todas las facilidades que me dieron durante todo este tiempo. Al Profr. Carlos Romero Hernández, al Profr. Sergio García Díaz, al Profr. Francisco López Arteaga y al Profr. José Luis Garcia.

Reconocimientos

A mi padre, por la fortaleza, dedicación y responsabilidad que tiene por el trabajo, que aunque no he terminado de aprenderlas, me entusiasma saber que él es así. Por sus sabios consejos sobre la vida, por su apoyo económico incondicional, sin importar mi promedio de licenciatura y sin importar que ya me haya tardado mucho tiempo en terminar y por ayudarme a hacer ésto, una realidad.

A mi madre, por enseñarme su valor y su decisión para enfrentar las cosas, por enseñarme que todo es posible aún a pesar de explicarle que era tan difícil como un problema computacional NP. Por no dejarme vencer en los malos momentos y por sus llamadas de atención necesarias, en el momento necesario, por su apoyo moral, sus facilidades de todo tipo y por ayudarme a hacer ésto, una realidad.

A Ivonne, por estar conmigo estos últimos cuatro años, por confiar en mi, por darme tranquilidad cuando la necesitaba, por alentarme a continuar, por inspirarme a emprender y terminar algo, por recargar mis baterias los fines de semana y por todo lo que significa en mi vida.

A mis hermanos Elit y Ofelia y mis otros hermanos Guadalupe y Jesús, por todo su apoyo incondicional. Así también, a mis queridos e inquietos sobrinos Eli y Aldo de 5.25 años y 3.75 años, respectivamente, por darme una buena excusa para dejar de escribir y jugar con ellos, sin olvidar, claro, al pequeño bebé de 2 meses y 11 dias.

A mi familiares más cercanos que contribuyeron a lograr este gran paso. A Homero, por sus grandes apreciaciones de sentido común sobre tópicos de computación.

Resumen

Un sistema hardware-software es aquel en el que coexisten por un lado un procesador programable convencional y por otro lado un circuito integrado de aplicación específica. Para una aplicación dada, el circuito en hardware específico normalmente se encarga de ejecutar la porción que consume el mayor tiempo de ejecución, mientras que el procesador convencional ejecuta la parte restante, de este modo se consigue mejorar el desempeño de tal aplicación. Algunos algoritmos criptográficos presentan características que pueden ser aprovechadas al ejecutarse en una implementación de este tipo. Dichas características son la realización de operaciones lógicas a nivel de bits y la aplicación repetitiva de dichas operaciones, como es el caso del algoritmo DES utilizado como medio para este estudio. Iniciando de una aplicación escrita puramente en software, el problema consiste en detectar la sección de código que irá implementada en hardware y la sección de código que se implementará en software. En este trabajo de investigación se propone una metodología para localizar la sección crítica, partiendo, como ejemplo, de un algoritmo de DES en C y generar una aplicación de tipo hardware-software. Se presentan los resultados obtenidos en cada etapa del desarrollo propuesto, hasta llegar a implementar los circuitos digitales correspondientes al hardware, usando VHDL y un dispositivo FPGA. Se muestran los resultados de recursos utilizados por el FPGA para cada una de las implantaciones de los circuitos. Se delinea una estrategia para comunicar el sistema en hardware con el software y finalmente se presenta un análisis de aceleración obtenido por la implantación en hardware sobre su similar en software.

Abstract

In a hardware-software system a conventional programmable processor and an ASIC (Application Specific Integrated Circuit) work together. For a specific application, the circuit usually executes the hardest part, in terms of execution time, while the conventional processor executes the rest. In this way, it is possible to improve the performance. For FPGA implementations, it is possible to take advantage of the inherent characteristics of cryptographic algorithms: the execution of logic operations at bit-level and the n iterations of these operations. DES algorithm is a good example of this, and we used it as a case-of-study in this project. Starting from a pure software application, the main problem addressed in this work is to detect the code section to be implemented in hardware and the section that will remain in the software. Through this research we propose a methodology to locate the critical section of C code in the DES algorithm and from there to derive a hardware-software implementation. The results obtained in every phase of the project are shown. In the last stage we implemented the digital circuits using VHDL and a FPGA. We show resources used by FPGA and the performance obtained for each implementation. We also outlined a strategy to communicate both hardware an software system. Finally we compare the acceleration of both implementations, in hardware vs. pure software.

Índice General

Índice de Tablas

Índice de Figuras

Introducción

En las computadoras las aplicaciones necesitan confidencialidad e integridad para proteger sus datos de accesos no autorizados. En una red de computadoras se requiere privacidad y autenticación para la transferencia de datos [Knu98]. Algunas soluciones a estos problemas son proporcionadas por la criptografía, que es considerada la mejor y la más importante herramienta de seguridad en el nivel de aplicaciones, además de ser un componente importante de muchas soluciones de seguridad en redes de computadoras [Knu98] [Mor88].

Los sistemas criptográficos se dividen básicamente en dos tipos: por una parte están los cifrados simétricos o de clave secreta, en los que tanto el cifrado como el descifrado requieren de la misma clave. Dentro de este tipo están los algoritmos DES, DESede, etc. Por otro lado están los cifrados asimétricos o de clave pública que siempre tienen dos claves diferentes una privada y una pública. De este tipo son los algoritmos RSA, DH, etc. [Luc02].

Los llamados algoritmos criptográficos fuertes fueron diseñados para ser ejecutados por dispositivos de hardware especializado, a pesar de esto, una gran cantidad de aplicaciones de computadoras realizan criptografía por software. Esto se debe a que los algoritmos criptográficos son sólo una parte de aplicaciones más completas dedicadas a aspectos de la seguridad en el intercambio y almacenamiento de información. Estas aplicaciones en software son ejecutadas en un procesador convencional de propósito general, por tanto, el desempeño que se obtiene al implantar un algoritmo criptográfico puramente en software en ocasiones no es el adecuado. Lo anterior se debe a la disparidad existente entre la arquitectura y el algoritmo, esto es, el tipo de operaciones que requiere el algoritmo y cómo éstas son mapeadas a la arquitectura del procesador. Entre tales disparidades se pueden mencionar el uso de longitudes de palabra no estándar, operaciones para manipulación de bits y manejo de permutaciones sobre secuencias de símbolos. Cuando se requiere superar las limitantes en rendimiento impuestas por la arquitectura de procesadores convencionales se pueden utilizar enfoques varios: optimizaciones sobre las operaciones para ajustar a longitudes de palabra estándar, explotar paralelismo, o desarrollar circuitos electrónicos de aplicación específica.

Algunos autores en [NOOS95] examinan tres diversos métodos para mejorar el funcionamiento del software criptográfico: diseño de nuevos algoritmos veloces, paralelización y soporte de hardware independiente del algoritmo. Eli Biham en [Bih96], planteó la idea de las implementaciones en software de rebanada de bits (bitslice) de los cifrados de bloque, aplicándolo en las máquinas seriales con grandes registros como los disponibles en computadoras modernas. El término "rebanada de bits" fue acuñado por Matthew Kwan y es práctica

común usarlo en máquinas vectoriales ya que permiten realizar operaciones paralelas [Sch96].

DES (Data Encryption Standard) [FIP77] es un algoritmo usado para cifrar texto en claro con datos de entrada en bloques de 64 bits que son procesados junto con una clave de 56 bits, para obtener a la salida un texto cifrado no comprensible de 64 bits. DES ha sido uno de los algoritmos criptográficos más usados en el pasado y aunque su robustez ha sido fuertemente cuestionada en los años recientes, la estructura seguida en su diseño es típica de los algoritmos criptográficos simétricos. Las operaciones básicas que se utilizan en DES son or-exclusivo bit a bit, permutaciones predeterminadas (cajas P) y tablas de sustituciones (Cajas S) [Sta99].

Michael Wiener en [Wie94] describe un diseño de DES a nivel de compuertas en que las iteraciones son desplegadas en pipeline y son ejecutadas por 16 instancias separadas, logrando un desempeño de ejecución de 3.2 Gigabit por segundo. Algunos autores en [SGRF97] describen una implementación de un flip-chip de DES que opera con un desempeño de ejecución de 9.6 Gigabits por segundo. Mientras que otros circuitos como el HiFn 7751 [HiF99] o el VMS115 VLSI [VLS99] se ejecutan a 80 MHz entregando un encriptado de aproximadamente 100Megabits y 200Megabits para SHA-1 y Triple DES.

Los sistemas hardware-software emplean una combinación de sus dos componentes para ejecutar una tarea específica, lográndose obtener las ventajas más apropiadas tanto de uno como de otro. Considerando que un algoritmo de cifrado puede formar parte de una aplicación más grande, el software se utiliza para proporcionar flexibilidad en la ejecución de tareas no susceptibles de mejorar su rendimiento en hardware. Por tanto, el hardware es usado para mejorar el desempeño de solamente algunas secciones del código, altamente repetitivas y con una complejidad aceptable para implantarse directamente en un circuito.

Los enfoques hardware-software han mostrado su factibilidad debido a la existencia de dispositivos programables para la implantación de circuitos. Entre los más utilizados actualmente se encuentran los FPGA's (Field Programmable Gate Arrays - Arreglo de Compuertas Lógicas Programables). Un FPGA es un dispositivo lógico programable consistente de bloques lógicos configurables (CLB's), y puertos de entrada/salida (IOB's) e interruptores programables. Los FPGAs se han usado para desarrollar de manera rápida circuitos de aplicación específica [Gra96]. La creciente capacidad de estos dispositivos y su facilidad para reprogramarse ha despertado el interés de una amplia parte de la comunidad de cómputo para implantar sobre hardware algunas tareas que usualmente se han realizado en software. La implantación de circuitos sobre FPGAs ha mostrado tener un ciclo de diseño más corto que la construcción de circuitos VLSI (Very Large Scale Integration). Implementar un cifrador parametrizado en un FPGA ofrece la oportunidad de modificar el diseño de una manera segura, simple y sin recurrir a los ajustes *ad-hoc* [MWMD97].

Los sistemas hardware-software son relativamente nuevos por lo que no existen metodologías bien definidas para construir sistemas de este tipo. Parte de la dificultad se debe a que, por un lado, las soluciones de cómputo basadas en FPGA's no son tan rápidas como sus competidores los procesadores convencionales en aplicaciones de propósito general. Además las arquitecturas actuales de FPGAs presentan limitaciones en densidad lógica [Gra96], lo

cual obliga a que los circuitos implantados en ellos no tengan una alta demanda de recursos. El factor velocidad puede ser contrarrestado al entender qué cualidades de una aplicación en software se pueden explotar para alcanzar una aceleración y cuánto cada cualidad puede contribuir a esa aceleración [Gra96]. Por otra parte, el diseño de un número importante de sistemas digitales se realiza a partir de descripciones funcionales o estructurales en un lenguaje de descripción de hardware como VHDL (Very high speed integrated circuit Hardware Description Language - Lenguaje de descripción hardware de circuito integrados de muy alta velocidad). Mediante VHDL se puede hacer el modelado de circuitos, la simulación lógica y aún la síntesis automática de circuitos [GVL99].

El objetivo de esta investigación es presentar una metodología para implementar un diseño en hardware-software a partir de una aplicación puramente en software de algoritmos que presentan características especiales, como es el caso de algunos algoritmos criptográficos. Se ha elegido específicamente el algoritmo DES en software como medio para demostrar cómo aplicar esta metodología y eventualmante mejorar el rendimiento mediante una implementación de este tipo. Aún cuando se trata de un algoritmo simétrico y por tanto de un algoritmo relativamente rápido, éste presenta características idóneas como son manejo de operaciones a nivel de bits y una gran cantidad de operaciones repetitivas.

Para llevar a cabo todo lo anterior se plantean los siguientes objetivos específicos:

- Se inicia con una implementación totalmente en software de DES. Después se realiza el análisis del algoritmo criptográfico, el cual consiste en detectar las partes más costosas del programa, debido al tiempo ocupado, mismas que son aprovechadas al ser ejecutadas por una máquina de cómputo basada en FPGA.

- Se hace un diseño para implementar un sistema de hardware específico, que cumple con el procesamiento correcto de datos y por consecuencia que mejora su rendimiento. Esta implementación en hardware es realizada en VHDL.

- La implementación llegó hasta la simulación funcional, la simulación lógica y el cálculo de retardos de los datos en su trayecto completo a través de un FPGA, en software.

- Se procede a realizar las pruebas de desempeño para dicho circuito.

- Con los resultados obtenidos se realiza un estudio comparativo del desempeño del algoritmo criptográfico DES utilizando: software únicamente (mediante un programa optimizado) y un sistema de hardware combinado con software.

El contenido del trabajo de tesis se describe a continuación.

En el primer capítulo de este trabajo se presenta una revisión de la criptografía en general, así como de los diferentes algoritmos existentes y una breve explicación del modo de operación de cada uno. Al final del capítulo se presenta un estudio detallado de operaciones que realizan tres algoritmos en particular: SHA-1, RC5 y AES. Este estudio tiene la finalidad de mostrar que estos algoritmos se componen de operaciones lógicas altamente repetitivas, y

que podrían ser buenos candidatos para su implementación en sistemas hardware-software.

El segundo capítulo se tratan los sistemas hardware-software. Aquí se revisan las ventajas y desventajas tanto del hardware y software. También se revisa la tecnología y las herramientas sobre las cuales se ha desarrollado este nuevo paradigma de diseño de sistemas. Se da también una estrategia general de particionamiento que determina cada uno de los componentes que deberá ir implementada en hardware y en software. Por último se habla de las computadoras reconfigurables como una aplicación de este tipo de sistemas.

Antecedentes y principios de operación que dieron origen al algoritmo DES, son presentados en el tercer capítulo. Se aborda el funcionamiento a detalle del algoritmo DES, explicando todas y cada una de las operaciones que realiza tal como se describe en la Norma FIPS-46-2 [FIP77] para su implementación original en hardware.

El cuarto capítulo presenta la parte sustancial del trabajo. Se inicia con los detalles de implementación en software del algoritmo DES. Enseguida se muestra el procedimiento a seguir para determinar qué parte del algoritmo se implementará en hardware y qué parte deberá ir en software, haciendo uso de un análisis de desempeño del algoritmo y un estudio para detectar cuál es la parte que consume más tiempo de ejecución.

En el capítulo cinco se presenta la implementación específica en hardware del segmento del algoritmo que consume la mayor cantidad de tiempo. Así tambien se muestran detalles del diseño y los resultados de desempeño obtenidos por los circuitos. También se muestra un análisis de aceleración entre software y hardware así como las comparaciones de ejecución. Al final de este capítulo se da un lineamiento de cómo establecer la coexistencia de ambos sistemas, por un lado el algoritmo en software y por otro lado el sistema en hardware (o máquina de cómputo basada en FPGA).

Por último se presentan las conclusiones y la bibliografía.

Capítulo 1

Criptografía

En este capítulo se presenta una pequeña revisión sobre criptografía, técnicas criptográficas y algunas aplicaciones de los algoritmos criptográficos. Después se presentan implementaciones actuales en software y hardware de algoritmos criptográficos. Por último se muestra un estudio de operaciones realizado a tres diferentes algoritmos, para mostrar el alto grado de repetición de sus operaciones básicas, lo que resultaría en un alto costosos para los procesadores de las computadoras.

1.1 Conceptos básicos.

Según el Diccionario de la Real Academia, la palabra criptografía proviene del griego $\kappa\rho\upsilon\pi\tau\acute{o}\varsigma$, que significa oculto, y, $\gamma\rho\acute{a}\varphi\epsilon\iota\nu$ que significa escritura, y su definición es: *"Arte de escribir con clave secreta o de un modo enigmático"*. La criptografía se ha convertido en un conjunto de técnicas que tratan sobre la protección de la información, frente a observadores no autorizados. Entre las disciplinas que engloba cabe destacar la teoría de la información, la teoría de números (Matemática discreta) y la Complejidad algorítmica [Luc02].

La criptografía es una parte de las matemáticas que se refiere al uso de códigos y consiste básicamente de un sistema, llamado criptosistema, que codifica los mensajes antes de enviarlos y que los decodifica al recibirlos. Por otro lado el criptoanálisis engloba a las técnicas que se usan para romper o analizar dichos códigos. El término criptología, se emplea habitualmente para agrupar las dos disciplinas anteriores, la criptografía y el criptoanálisis.

Un cifrado es un algoritmo matemático que transforma una cadena de datos fuente (texto en claro) en datos no entendibles (texto cifrado) y viceversa, de un modo que únicamente dependa del valor de una variable criptográfica o clave. Una clave es un valor secreto tal como una contraseña (password) o un código de tarjeta del banco y si se piensa como una secuencia de bytes, no es fácilmente entendible. Si no se tiene la clave no se puede llevar a cabo cualquier transformación. Al encriptar el mismo mensaje usando diferentes claves, se obtienen diferentes textos cifrados. Algunos cifrados tienen un tamaño de clave variable, mientras que otro tiene claves de longitud fija. Escoger el tamaño correcto de la clave presenta diferencias entre tener un nivel confortable de seguridad y tener una aplicación ejecutándose lentamente.

La criptografía proporciona características estándares de seguridad cuando es usada correctamente, tal como:

- Confidencialidad, asegura que los datos no puedan ser vistos por una persona no autorizada.
- Integridad, asegura que los datos no sean cambiados sin conocimiento previo.
- Autenticación, asegura que las personas con que se trata no sean impostoras [Knu98].

Los criptosistemas o sistemas criptográficos estan clasificados generalmente en tres grandes grupos independientes que se mencionan a continuación:

1. Número de llaves usadas. Si el transmisor y el receptor, ambos, usan la misma clave, el sistema es referido como un encriptamiento simétrico, de clave simple, clave secreta o convencional. Si el transmisor y receptor cada uno usa una clave diferente, el sistema es llamado encriptamiento asimétrico o de llave pública.
2. Tipo de operaciones. Todos los algoritmos de encriptamiento están basados en dos principios generales: sustitución, en el que cada elemento del texto plano (bit, letra, grupo de bits o letras) es mapeado en otro elemento, y transposición, en que cada elemento en el texto plano es reordenado. El requerimiento fundamental es que la información no se pierda (esto es, que todas las operaciones sean reversibles).
3. Modo de procesamiento. Un cifrado de bloque procesa a la entrada un bloque de elementos a la vez, produciendo un bloque de salida para cada bloque de entrada. Un cifrado de flujo procesa continuamente elementos de entrada, produciendo a la salida un elemento a la vez. Para comunicaciones es interesante, particularmente, el cifrado de flujo, de modo que sea capaz de transformar un mensaje a caracteres o bloques de datos, en un modo serie, entre una red de comunicación.

La combinación de estas tres dimensiones e incluso la combinación de los componentes de una dimensión, han dado lugar a las diferentes técnicas criptográficas, existentes actualmente. Por ejemplo, cualquiera de los dos cifrados de llave pública o llave secreta puede ser usado en el modo de flujo. Otros productos y sistemas involucran múltiples etapas de substitución y transposición. Mientras que otros como los sistemas híbridos combinan cifrados simétricos y asimétricos, etc.

1.1.1 Técnicas criptográficas.

Debido a las múltiples aplicaciones de computadoras sobre las cuáles se pueden aplicar herramientas criptográficas se han originado diversas técnicas de encriptado, siendo algunas más seguros, pero más costosas que otras. Estas técnicas se pueden agrupar en: encriptado convencional, encriptado de llave pública, funciones hash, firma digital y códigos de autenticación de mensajes (MAC, Message Authentication Code).

- Encriptado simétrico. Una cifrado simétrico o de llave secreta es aquel en el que ambos el cifrado y el descifrado requieren la misma clave; consecuentemente, el que envía y el que recibe pueden compartir información secreta; estrictamente hablando, las dos claves necesitan no ser las mismas, pero puede ser posible derivar cualquiera de las dos claves a partir de la otra. Si alguien encuentra la llave secreta, entonces debe generarse una nueva llave y transferirla cuidadosamente a la otra persona. Algunos de los algoritmos que utilizan esta técnica pueden encontrarse en [Sim92, Sch96, Sta99].

- Encriptado asimétrico. Un cifrado asimétrico o de llave pública siempre tiene dos claves diferentes una privada y una pública. En una aplicación típica, el que envía encripta con la llave pública del receptor, y únicamente el que recibe puede descifrar el mensaje, usando su llave privada. En algunos casos, el inverso del proceso también trabaja; los datos pueden ser encriptados con la llave privada y pueden ser decifrados con la llave pública, únicamente el receptor posee la información secreta. La llave pública es fácilmente calculada a partir de la llave privada vía una simple transformación matemática, pero, es computacionalmente imposible determinar la clave privada a partir de la clave pública. Los cifrados asimétricos son computacionalmente muy lentos, por ejemplo, en computadoras científicas de buen rendimiento tienen una razón de decenas de caracteres por segundo, por lo que su uso esta limitado a aplicaciones como administración de llaves, y encriptamiento de mensajes cortos. Algunos algoritmos pueden consultarse en [Sim92, Sch96, Sta99].

- Función Hash. El hash criptográfico es una función resumen que puede ser usada para verificar la integridad de los datos. Una función resumen genera un número especial obtenido de un conjunto de datos de entrada. El propósito de una función hash es producir una huella digital de un archivo, mensaje u otro bloque de datos. Una función hash puede ser aplicada a un bloque de datos de algún tamaño y produce una salida de longitud fija. Más información sobre funciones hash pueden ser encontradas en [Sim92, Sch96, Sta99].

- Firma digital. La firma digital extiende a la función resumen para resolver otros problemas. El método llamado firma digital, para autenticación, proviene de la combinación de un resumen de mensaje y un cifrado asimétrico. Para verificar la firma, primero se decifra la firma, usando la llave pública de la persona que encripto el mensaje. Esto deja el valor de la función resumen del mensaje. Entonces se calcula localmente la función resumen del mismo archivo. Si los dos valores hash, del mensaje o archivo, son iguales, la firma de la primera persona es verificada, y se puede estar seguro que el mensaje o el archivo es correcto. Algunos algoritmos que se encargan de proporcionar firmas digitales pueden ser encontrados en [Sim92, Sch96, Sta99].

- Código de autenticación de mensaje. Un Código de Autenticación de Mensaje (MAC - Message Authentication Code), es básicamente un mensaje asociado con una clave. También conocido como chequeo de suma (checksum) criptográfico, consiste en producir un valor corto que esta basado tanto en la entrada de datos como en la clave. En teoría, únicamente algunas personas con la misma clave pueden producir el mismo MAC de la misma entrada de datos. El MAC es agregado al mensaje fuente al mismo

tiempo que el mensaje se reconoce como correcto. El receptor autentica el mensaje recalculando el MAC. Los algoritmos que utilizan esta técnica pueden ser encontrados en [Sim92, Sch96, Sta99].

1.1.2 Aplicaciones de los algoritmos criptográficos.

Una vez revisadas las diferentes técnicas criptográficas, ahora se verá rápidamente los mecanismos desarrollados para la seguridad de aplicaciones específicas en diversos ambientes como: cliente/servidor, correo electrónico, comercio electrónico e incluso en las capas del protocolo OSI (Opens System Interconexion), etc.

- Servicios de autenticación. La criptografía proporciona métodos fuertes de autenticación llamados firmas y certificados. La autenticación es el proceso mediante el cual se proporciona una identidad. Muchos sistemas de computadoras usan una combinación de ID de usuario, para identificar a una persona, y un password, para autenticar la identidad de la persona. No obstante, se tuvieron que desarrollar otras funciones para soportar autenticación a nivel de aplicación y certificados digitales. Dos de los servicios más importantes son Kerberos y X.509 [Sta99].

- Seguridad en el correo-e. El correo electrónico es la aplicación mas ampliamente usada entre todas las arquitecturas y plataformas comerciales de ambientes distribuido. Con el crecimiento explosivo del correo electrónico, hay también un crecimiento en la demanda para autenticar y lograr confidencialidad en los servicios. Algunos de los sistemas de seguridad más usados para correo electrónico son: PGP y S/MIME [Sta99].

- Seguridad IP. El protocolo IPSec proporciona servicios de seguridad necesarios en la capa IP, del protocolo OSI, mediante un conjunto de capacidades que aseguran las comunicaciones entre una red local (LAN), entre redes publicas y privadas de área amplia (WANs) y entre el Internet. La seguridad en el nivel IP rodea tres áreas funcionales: autenticación, confidencialidad y administración de llaves. IPSec permite al sistema seleccionar protocolos requeridos de seguridad, determinar los algoritmos a usar para los servicios, y poner en algún lugar las claves criptográficas requeridas [Sta99].

- Seguridad en Internet. El número de individuos y compañías con acceso a Internet se ha expandido rápidamente y por consiguiente se ha incrementado la facilidad para proporcionar comercio electrónico en la web. Sin embargo, la web e Internet son extremadamente vulnerables, por lo que se ha trabajado en aplicaciones que permitan un intercambio de información de forma más seguro, cosiguiéndose algunos protocolos para la seguridad en Internet como SSL (Secure Socket Layer), TLS (Transport Layer Security) y SET (Secure Electronic Transactions) [Sta99].

1.2 Implementaciones en software de algoritmos criptográficos.

Los programas de encriptado de software son populares y están disponibles para la mayoría de los sistema operativos. Éstos pretenden proteger en su mayoría archivos individuales de

computadora, los usuarios generalmente tienen que encriptar y decifrar manualmente los archivos especificados. Es importante que el esquema de administración de llaves sea seguro. Las claves y los archivos no encriptados serían borrados después de la operación de encriptado. Algunas implementaciones en software de algoritmos criptográficos, tanto comerciales como académicos, se presentan a continuación.

- API's en Java. Son un conjunto de paquetes que se utilizan para escribir programas seguros en Java. El software de criptografía de Java llega de dos piezas. Una pieza es el JDK (Java Development Kitt - Kitt de desarrollo de Java), que incluye clases criptográficas para autenticación. La otra pieza es el JCE (Java Cryptography Extension - Extensión de Criptografía de Java), que incluye lo que ellos llaman "criptografía fuerte". El diseño total de las clases de criptografía está gobernadas por el JCA (Java Cyptography Architecture - Arquitectura Criptográfica de Java). El JCA especifica patrones de diseño y una arquitectura extensible para definir conceptos criptográficos y algoritmos [Knu98].

- DES en Software. Eli Biham describe una nueva implementación de DES [Bih96] que puede ser ejecutada eficientemente en software. Esta implementación usa una representación no estándar de bits para los bloques de DES y ve al procesador como una computadora SIMD (Single Instructions Múltiple Data - Instrucciones sencillas múltiples datos). Esta implementación no sufre de un alto overhead al hacer cómputo de permutaciones de bits. El procesador es visto, por ejemplo con palabras de 64 bits, como una computadora paralela que trabaja simultáneamente en 64 operaciones de un bit, mientras que en una implementación normal los 64 bits de cada bloque son asignados en 64 diferentes palabras. Realmente el cifrado completo se ve como un circuito de compuertas lógicas, incluso la representación de las cajas S, aplicadas en software. Este circuito es computado 64 veces en paralelo (como el tamaño de palabra del procesador), y entonces se puede conseguir una gran aceleración al usar operaciones muy simples. En este trabajo, se presenta primero una implementación con representación estándar optimizada que es cerca de dos veces más rápida que la implementación estándar más rápida de DES (libdes de Eric Young) en un procesador 8400 de una Alpha a 300 MHz. Mientras que la nueva implementación rápida de DES con representación no estándar es cinco veces más rápida que la implementación libdes de Eric Young, en el mismo procesador mencionado anteriormente. La misma idea puede ser aplicada a otro cifrados.

- Software criptográfico de alto desempeño [NOOS95]. Las actuales implementaciones de software de los actuales algoritmos criptográficos son de orden de magnitud más lentas que el requerido para asegurar, por ejemplo, una red Gigabit. Pero, además se requiere, una eficiente implementación de un conjunto estándar de algoritmos para una conexión segura; que permita flexibilidad en la selección de algoritmos para los estándares de seguridad de Internet; que pueda ser cambiado en corto tiempo si es que un algoritmo llega a ser quebrado, en término coloquial; que el algoritmo sea omnipresente, barato y de fácil exportación y además que no presente incrementos limitados en desempeño. Para mejorar el desempeño del software criptográfico se examinan tres diferentes métodos:

- Rediseño de los algoritmo más rápidos.
  Se pueden seleccionar de la literatura una variedad de algoritmos (un nuevo algoritmo es considerado inseguro). Entonces, en el caso de algoritmos con teoría de números, se pueden escoger los fundamentos para la implementación de las estructuras y/o los operadores especiales. Si implementamos cuidadosamente la representación de la estructura y los operadores especiales que se requieren, es posible mejorar el desempeño, sin que esto resulte en un algoritmo significativamente mas débil que el original. Un ejemplo es el DHKX [DH93, T.I95], que contiene un conjunto de modificaciones al algoritmo de intercambio de llaves de Diffie-Hellman y que ya se ha implementado .
- Paralelización.
  Algunos experimentos realizados en esta investigación, determinaron que en una paralelización a nivel de paquete, usando una conexión sencilla de TCP, se consigue una aceleración lineal para los algoritmos DES y MD5. Otros protocolos con cómputo intensivo, tal como triple DES y RSA, también presentarían una escala lineal de aceleración. Estos resultados sugieren que el protocolo de software de encriptado puede ser mejorado usando paralelismo.
- Soporte de hardware independiente del algoritmo.
  La idea básica es diseñar un procesador o coprocesador RISC clásico para un amplia gama de software criptográfico. Agregando instrucciones adicionales al conjunto de instrucciones estándar RISC, útiles para las operaciones de cifrado de los datos, se podría mejorar significativamente la ejecución total de la colección de algoritmos. Un cuidadoso examen de las actuales implementaciones de software criptográfico, sobre máquinas modernas RISC, tiene identificados tres problemas básicos: unidades de operaciones de subtamaños de palabras, unidades de operación de sobretamaños de palabras, y operaciones de otros grupos diferentes a los enteros. Si se puede demostrar que habilitando el corazón de un CPU estándar, con un numero mínimo de instrucciones tipo RISC, logramos el incrementar el desempeño de varios algoritmos seguros, entonces este sistema puede ser viable.

1.3 Implementaciones en hardware de algoritmos criptográficos.

El hecho de que el algoritmo no necesita ser guardado en secreto, quiere decir, que las manufactureras pueden desarrollar implementaciones de chips, de bajo costo, de los algoritmos de encriptamiento de datos. Con el uso del encriptamiento convencional, el principal problema de seguridad es mantener en secreto la llave. Aún cuando el encriptado por software esta llegando a ser predominante, hasta hace muy poco, todos los productos de encriptado tenían la forma de hardware especializado, y actualmente todavía es la opción seleccionada para aplicaciones militares y aplicaciones serias de comercio. La NSA (National Security Agency - Agencia Nacional de Seguridad), por ejemplo, únicamente autoriza el encriptado en hardware. Hay varias razones por las que esto es así: la velocidad, la seguridad y la facilidad de instalación.

Los dos algoritmos más comunes de encriptado, DES y RSA, se ejecutan de manera ineficiente en un procesador de propósito general. El procesador primario de las computadoras es poco efectivo para esto [Sch96]. Existen actualmente un gran número de productos criptográficos, tanto comerciales como académicos, desarrollados en hardware, pero solo tres ejemplos se mencionaran a continuación.

- Tarjetas inteligentes (T.I.). Una TI tiene el tamaño de una tarjeta de crédito convencional y tiene un CI en su interior, basado en un microprocesador. Además de la microelectrónica, utiliza otras dos técnicas como el procesamiento de datos y la criptografía. El CI guarda datos y programas que se protegen por mecanismos de seguridad: PIN (Personal Identification Number) y funciones criptográficas, que procesan los datos que se ingresan a la tarjeta, antes de guardarse en su memoria. La aplicación principal esta enfocada hacia los sistemas de pago en bancos, como tarjetas de crédito, tarjetas de débito y como monederos electrónicos, es por eso que las necesidades básicas de seguridad se consideraron una prioridad.

- Coprocesador criptográfico CMOS S/390. Un coprocesador criptográfico CMOS es simplemente el procesador secundario que se encarga de las funciones de seguridad y que con una programación adecuada, ayuda a impedir que personas no autorizadas accedan a información localizada en los servidores de una red empresarial. Este coprocesador es utilizado por los servidores G3, G4 y G5 S/390 Parallel Enterprise de IBM. Cada uno de estos servidores puede contener uno o dos chips de coprocesador criptográfico CMOS.

- Una implementación Flip-chip de DES. Esta implementación [SGRF97] describe un diseño del algoritmo DES usando chip flip-chip area-array I/O montado en un sustrato MCM-D (Multi-Chip Module - Modulo Multi-Chip). Entre sus características se incluyen las siguientes: uso de un area-array I/O denso para lograr un gran ancho de banda; arquitectura completamente en pipeline que soporta múltiples cifrados (por ejemplo, triple DES) sin pérdida de desempeño de ejecución; capacidad para multiplexar flujos de datos, cada una sobre el control potencial de una única llave; y el uso del sustrato MCM-D para distribuir las señales de potencia, tierra y reloj. El hecho de usar un sustrato MCM también salva el área dentro del chip, comparado con los métodos convencionales. Además, la baja *parasitics* de la conexión flip-chip resulta en un bajo consumo de potencia en el controlador y reduce los retrasos de comunicación chip a chip. Esta implementación desenvuelve el pipeline en 16 estados separados. Se realiza un pipeline en un modo bit, que direcciona cada estado para realizar cualquiera de las dos operaciones encriptado o decifrado. Como resultado el pipeline puede simultáneamente contener múltiples e independientes flujos de datos, cada una requiriendo ya sea encriptado o decriptado. Utilizando el gran ancho de banda producido por el arreglo de área de E/S, el chip acepta tanto el cifrado como el descifrado y genera un bloque de 64 bits cada ciclo de reloj. La simulación del circuito realizada sobre la netlist extraída del área de trabajo indica que en el proceso más lento el chip puede operar comódamente a 150 MHz. Esta frecuencia corresponde a un desempeño de ejecución de 9.6 Gb/s.

1.3.1 Criterios de diseño de algoritmos convencionales.

Las implementaciones de chips VLSI (Very Large Scale Integration - muy alta escala de integración), son diseñadas para lograr altas velocidades y junto con los principios de diseño pueden conseguir una mayor optimización. Algunos principios se mencionan a continuación:

- Similitud de cifrado y descifrado. Difieren únicamente en el modo de usar la llave para que el mismo dispositivo pueda ser usado para ambas operaciones.
- Estructura regular. El cifrado tendría una estructura modular regular para facilitar las implementaciones VLSI. Se construyen de módulos básicos repetidos varias veces, dependiendo del número de iteraciones.
- Los algoritmos convencionales utilizan operaciones primitivas, comúnmente encontradas en microprocesadores, algunas de estas operaciones son:
  - Adición.
  - OR exclusivo.
  - Rotación circular a la izquierda.
  - Substracción.
  - Complemento.
  - AND.
- Bajos requerimientos de memoria. Un bajo requerimiento de memoria hacen que sean apropiados para tarjetas pequeñas y otros dispositivos con memoria restringida.
- Operaciones a nivel de bits. Éstas son ejecutadas más rápido por los procesadores.

La Tabla 1.1 desarrollada por Schneier [Sch96], compara el numero de ciclos de reloj requeridos en una Pentium para varios algoritmos convencionales implementados en C.

Algoritmo	Ciclos de reloj por iteración	Número de iteraciones	Número de ciclos de reloj por byte encriptado
Blowfish	9	16	18
RC5	12	16	23
DES	18	16	45
IDEA	50	8	50
Triple-DES	18	48	108

Tabla 1.1: Comparación de desempeño de algoritmos convencionales

1.4 Estudio de tres algoritmos criptográficos

En esta sección se estudian tres algoritmos criptográficos, con la finalidad de mostrar que éstos consisten de operaciones complicadas, sumamente repetitivas y algunas en bits de texto plano, por lo que se convierten en tareas de computo intensivo.

1.4.1 SHA-1 (Secure Hash Algorithm)

El algoritmo SHA-1 toma un mensaje de entrada de una longitud menor que 2^{64} bits y produce un valor de resumen del mensaje que es de una longitud de 160 bits. El ciclo principal del algoritmo procesa el mensaje en 512 bits a la vez y continua para tantos bloques de 512 bits como estén en el mensaje. Primero el mensaje es completado para hacer que su longitud sea un múltiplo de 512 bits. Cinco variables de 32 bits son inicializadas (A, B, C, D, E) y copiadas en otras diferentes variables. El bloque de mensaje es transformado de 16 palabras de 32 bits (512 bits) a 80 palabras de 32 bits (2560), para formar los subbloques de mensaje expandido.

El bucle principal tiene cuatro rondas de 20 operaciones cada una. Cada operación ejecuta una función no lineal sobre tres de las variables (por lo general b, c y d), se hacen corrimientos de algunas variables (a y b) y algunos de estos resultados son sumados junto con una variable, un subbloque de mensaje expandido y una valor constante de acuerdo al número de iteración dada. Finalmente los resultados reemplazan los valores de a, b, c, d y e. En la Figura 1.1 se muestra una operación de SHA.

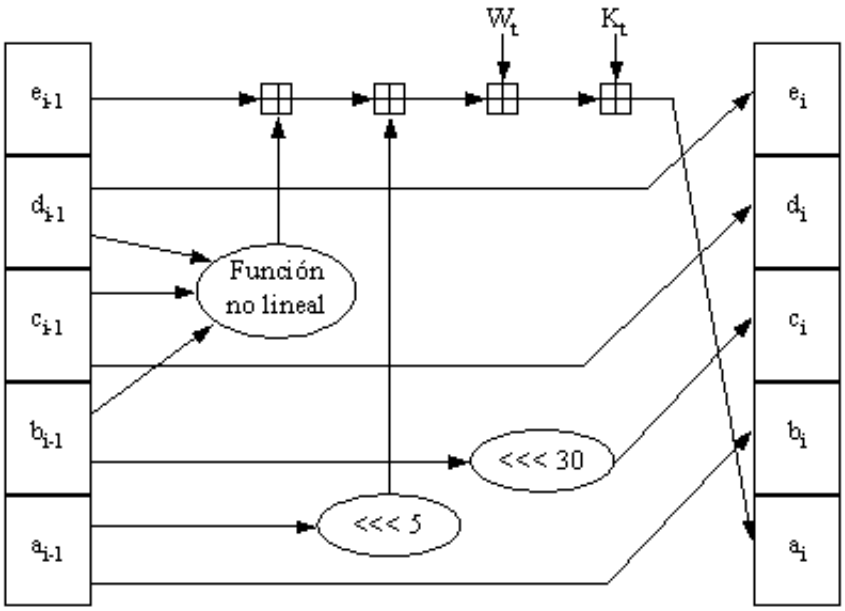

Figura 1.1: Una operación simple de SHA-1

El ciclo principal del algoritmo se vería como se muestra en el Algoritmo 1.1.
donde: t es el numero de operación, W_t representa el t-avo subbloque del mensaje, K_t un valor constante de acuerdo a t y f_t y que corresponde a una de las cuatro funciones siguientes:
$f_t(X, Y, Z) = (X \text{ and } Y) \text{ or } (\text{ (not } X) \text{ and } Z)$, para $t = 0$ a 19.
$f_t(X, Y, Z) = X \text{ xor } Y \text{ xor } Z$, para $t = 20$ a 39.
$f_t(X, Y, Z) = (X \text{ and } Y) \text{ or } (X \text{ and } Z) \text{ or } (Y \text{ and } Z)$, para $t = 40$ a 59.

Algoritmo 1.1 Ciclo principal de SHA-1.

Para $t = 0$ hasta 79 **hacer**
$TEMP = (a <<< 5) + f_t(bcd) + e + W_t + K_t$
$e = d$
$d = c$
$c = b <<< 30$
$b = a$
$a = TEMP$
fin de Para

$f_t(X, Y, Z) = X$ xor Y xor Z, para $t = 60$ a 79.

Después de todo esto, los valores de las variables son sumados a las variables de 32 bits inicializadas, y el algoritmo continúa con el siguiente bloque de datos. La salida final es la concatenación de A, B, C, D y E.

Para un bloque de mensaje de entrada de 512 bits, la cantidad total de operaciones realizadas por el algoritmo puede verse en la Tabla 1.2. Cabe destacar que todas estas operaciones se realizan a nivel de bits.

Nombre de la operación	Número de operaciones
Corrimientos a la izquierda	160
Sumas	320
NOT	20
AND	100
OR	60
XOR	80

Tabla 1.2: Cantidad de operaciones realizadas por SHA para un tamaño de mensaje de 512 bits

1.4.2 RC5

RC5 es una familia de algoritmos para encriptamiento simétrico de bloques de datos. Las implementaciones particulares de RC5 se designan como RC5-$w/r/b$, donde w es el tamaño de la palabra a utilizar; r es el número, no negativo, de rondas; y b es la longitud de la llave en bytes. Usa operaciones computacionalmente primitivas, encontradas en microprocesadores: XOR, suma y restas en módulo 2^{32}, así como rotaciones. Las rotaciones variables son una función no lineal, y además son operaciones de tiempo constante. En el algoritmo se incorporan rotaciones (cambios circulares de bits) cuya cantidad depende tanto de la llave como de los datos.

Para esta explicación se considera un bloque de 64 bits, sin embargo, el bloque puede tener una longitud variable. Para encriptar primero se divide el texto plano en dos palabras

de 32 bits: A y B (RC5 asume una convención *little-endian* para empaquetar los bytes en dos palabras: El primer byte va en la posición del bit de menor orden en el registro A, etc.). Al principio de las rondas r, se agrega un paso de preblanqueado que ayuda a ocultar parte de la entrada a la primera ronda. El ciclo principal del algoritmo tendría la forma del Algoritmo 1.2.

Algoritmo 1.2 Ciclo principal de RC5.

$A = A + S_0$
$B = B + S_1$
Para $i = 1$ hasta r **hacer**
  $A = ((A \text{ xor } B) <<< B) + S_{2i}$
  $B = ((B \text{ xor } A) <<< A) + S_{2i+1}$
fin de Para

donde: S_0, S_1, S_{2r+1} es el arreglo de 2r + 2 pa1abras de 32 bits, generadas a partir de la llave.

La salida final estará en los registros A y B. El proceso de descifrado se consigue invirtiendo los pasos del algoritmo anterior. La Figura 1.2 muestra una ronda del algoritmo.

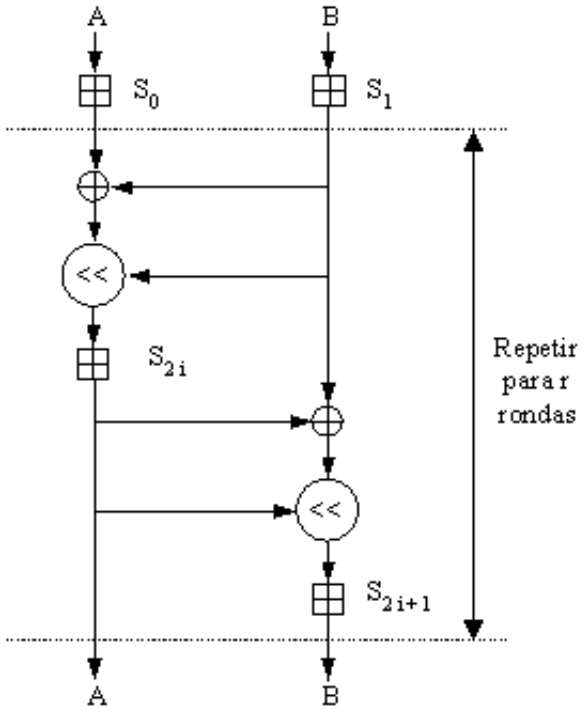

Figura 1.2: Una ronda de RC5.

Para crear el arreglo de llaves, primero, se copian los bytes de la clave en un arreglo, L, de c palabras de 32 bits, rellenando el final de la palabra con ceros si es necesario. Entonces,

se inicializa un arreglo, S, usando un generador de correspondencia lineal modulo 2^{32}, como se muestra en la estructura del Algoritmo 1.3.

Algoritmo 1.3 Inicialización del arreglo S en RC5.

$S_0 = \mathrm{P}$
Para $i = 1$ hasta $2(r+1) - 1$ **hacer**
  $S_i = (S_{i-1} + \mathrm{Q}) \bmod 2^{32}$
fin de Para

donde P y Q son constantes basadas en una representación binaria.

Finalmente se combina L en S como se muestra en Algoritmo 1.4.

Algoritmo 1.4 Combinar el arreglo L de la clave con el arreglo S inicializado para obtener el arreglo de llaves en RC5.

$A = B = i = j = 0$
Repetir
  $A = S_i = (S_i + A + B) <<< 3$
  $B = L_j = (S_j + A + B) <<< (A + B)$
  $i = (i + 1) \bmod 2(r + 1)$
  $j = (j + 1) \bmod c$
hasta 3n veces, donde n es el máximo de 2(r+1) y c

Para un cifrado RC5-32/20/64, en particular, con tamaños de palabra de 32 bits, 20 rondas, una clave de 64 bits, por consiguiente c=2 palabras de 32 bits. La cantidad de operaciones realizadas por el algoritmo se resumen en la Tabla 1.3. Es preciso recordar que se trata de un algoritmo orientado a palabras, por lo que las operaciones se realizan a nivel de bytes.

Nombre de la operación	Número de operaciones en encriptado	Número de operaciones en generación de subclaves
Corrimientos a la izquierda	40	252
Sumas modulo 32	42	923
Or-exclusivo bit a bit	40	0

Tabla 1.3: Cantidad de operaciones realizadas por RC5 para un bloque de 64 bits

1.4.3 AES (Rijndael)

El AES (Algorithm Encrypt Standard - Algoritmo de Encriptado Estándar) es un sistema simétrico de cifrado por bloques que encripta bloques de 128, 192 y 256 bits usando llaves de 128, 192 o 256 bits. Este algoritmo realiza varias de sus operaciones internas a nivel de

byte, interpretando estos como elementos de un campo de Galois (2^8), es decir, realiza una representación en forma de polinomios. El resto de operaciones se efectúan en términos de registros de 32 bits (palabras de 4 bytes). Sin embargo, en algunos casos, una secuencia de 32 bits se toma como un polinomio de grado inferior a 4, cuyos coeficentes son a su vez polinomios en GF(2^8).

AES posee un conjunto de rondas en el que se realizan iteraciones definidas de cuatro funciones invertibles diferentes, que operan con resultados intermedios llamados estados, formando así tres capas como se describe a continuación:

- Una capa de mezcla lineal. Formado por las funciones DesplazarFila y MezclarColumna. Permite obtener un alto nivel de difusión a lo largo de las diferentes rondas.
- Una capa no lineal. Formado por la función SustByte. Consiste en la aplicación paralela de cajas-S con propiedades optimas de no linealidad.
- Una capa de adición de claves. Un simple or-exclusivo entre el estado intermedio y la subclave correspondiente a cada ronda.

Los estados se representan mediante una matriz rectangular de bytes, que posee cuatro filas y Nb columnas (el numero de columnas es igual al tamaño del bloque utilizado dividido por 32). La matriz de estados se muestra en la Tabla 1.4.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$

Tabla 1.4: Ejemplo de una matriz de estado para un bloque de 160 bits y N_b=5.

La clave que se utilizará en el algorimo también se representa mediante una matriz rectangular de bytes, de cuatro filas y N_k columnas (el numero de columnas es igual al tamaño de la clave dividido entre 32), como se muestra en la Tabla 1.5

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Tabla 1.5: Ejemplo de una matriz de clave de 128 bits con $N_k = 4$

En algunos casos tanto el estado como la clave se consideran como vectores de 32 bits, estando cada registro constituído por los bytes de la columna correspondiente, ordenados de

arriba a abajo.

El algoritmo AES permite emplear diferentes longitudes tanto de bloque como de clave, el numero de rondas requerido en cada caso es variable El numero de rondas necesarias en función de N_b y N_k se muestran en la Tabla 1.6.

Tamaños	N_b=4 (128 bits)	N_b=6 (192 bits)	N_b=8 (256 bits)
N_k = 4 (128 bits)	10	12	14
N_k = 6 (128 bits)	12	12	14
N_k = 8 (128 bits)	14	14	14

Tabla 1.6: Numero de rondas para diferentes tamaños de bloques y de llaves.

Siendo M el bloque que queremos cifrar, S la matriz de estado, N_r el numero de rondas y K_i la subclave correspondiente a la ronda i-ésima, entonces se tendría la estructura del Algoritmo 1.5.

Algoritmo 1.5 Estructura del algoritmos AES

GenerarSubClaves(K)
$S = M$ xor K_0
Para $i = 1$ hasta N_r **hacer**
  $S \leftarrow$ SustByte(S)
  $S \leftarrow$ DesplazarFila(S)
  $S \leftarrow$ MezclarColumna(S)
  $S \leftarrow K_i$ xor S
fin de Para
$S \leftarrow$ SustByte(S)
$S \leftarrow$ DesplazarFila(S)
$S \leftarrow K_{Nr}$ xor S

El algoritmo de descifrado consiste en aplicar las inversas de cada una de las funciones en el orden contrario, y comenzar con el último K_i que se uso en el cifrado.

La función `SustByte ó ByteSub` es una sustitución no lineal que se aplica a cada byte de la matriz de estado, mediante una caja-S 8*8 de la forma xxxxyyyy. La función inversa de `SustByte` es la aplicación de la caja-S inversa correspondiente a cada byte de la matriz de estado.

La Función `DesplazarFila o ShiftRows` consiste en desplazar a la izquierda cíclicamente las filas de la matriz de estado. Cada fila fi se desplaza un numero ci de posiciones diferentes, cuyos valores están en función de Nb como se ve en la tabla 1.6. La función inversa es un desplazamiento a la derecha de las filas de la matriz de estado el mismo numero de posiciones que en la Tabla 1.7.

Valor de N_b	c_1	c_2	c_3
4 ó 6	1	2	3
8	1	3	4

Tabla 1.7: Valores de c_i según el tamaño de bloque N_b

La Función `MezclarColumnas o MixColumns` transforma los bytes de una misma columna de la matriz de estado. Cada columna del vector de estado se considera un polinomio cuyos coeficientes pertenecen a GF(2ˆ8) y se multiplican módulo $xˆ8 + xˆ4 + xˆ3 + xˆ2 + x + 1$ por $C(x) = 03xˆ4 + 01xˆ2 + 01x + 02$.

La inversa de `MezclarColumnas` se obtiene multiplicando cada columna de la matriz de estado por el polinomio $d(x) = 0Bxˆ4 + 0Dxˆ2 + 09x + 0E$.

La función `GenerarSublcaves o KeyScheduling` obtiene de la clave principal las subclaves K_i. Este numero de subclaves que se generan depende del número de rondas empleadas (N_r).

Una función de selección simplemente toma consecutivamente, de la secuencia obtenida por la función de expansión de clave, bloques del mismo de tamaño que la matriz de estado y los va asignando a cada subclave K_i.

Una función de expansión extiende y llena un arreglo lineal W con palabras de 4 bytes y de longitud $N_b \times (N_r + 1)$, donde N_b es el numero de columnas de la matriz. La función de expansión también depende del valor de N_k. Las primeras N_k palabras contienen la clave de cifrado, mientras que el resto de las palabras son las que se derivarán de la principal, como se puede ver en la estructura 1.6.

Algoritmo 1.6 Función de expansión de las claves en AES

$K_0 = W_0$ concatenación W_1 concatenacion W_2 concatenación W_3
Para ($i = N_k$ hasta $N_b \times (N_r + 1)$) **hacer**
  Si (i mod $N_k = 0$) **entonces**
    $W_i = W_{i-Nk}$ xor ByteSub[RotaciónCircIzq(W_{i-1}) xor $(2^{k-1}, 00, 00, 00)_h$]
  fin de Si
  Si ($N_k > 6$) y (i mod $N_k = 4$)) **entonces**
    $W_i = W_{i-Nk}$ xor SustByte(W_i)
  fin de Si
  Si (i mod $N_k \neq 0$) **entonces**
    $W_i = W_{i-Nk}$ xor W_{i-1}
  fin de Si
fin de Para

Dado un tamaño de bloque de 128 bits (16 bytes) y una tamaño de clave de 128 bits ($N_b = 4$), se tiene un numero de rondas $N_r = 10$. La longitud del arreglo lineal de subclaves es de

$4 \times (10 + 1) = 44$ palabras de 32 bits. Cada cuatro palabras tienen 128 bits y corresponde a la subclave K_r.En la Tabla 1.8 se pueden ver las cantidades de operaciones a realizar para encriptar los datos.

Nombre de la operación	Número de operaciones en encriptado	Número de operaciones Generación de subclaves
Corrimiento circular a la izquierda	30	10
Or-exclusivo	26	50
Multiplicación	9	–

Tabla 1.8: Cantidad de operaciones realizadas para un bloque de 128 bits.

1.4.4 Análisis de los algoritmos.

A partir del estudio de los tres algoritmos, realizado en la sección anterior, se pueden establecer los siguientes puntos de discusión:

- Aunque cada uno de los tres algoritmos realizan procesamientos diferentes sobre los datos, tienen operaciones lógicas y corrimientos que son comunes entre ellos. Estas operaciones son realizadas en su mayoría a nivel de bits y aunque en AES se trabaja con operaciones de enteros realmente son necesarios sólo el conjunto de bits que tiene el campo en proceso. Estas operaciones no son ejecutadas eficientemente en un procesador convencional, debido al nivel de abstracción existente en su conjunto de instrucciones, lo cual provocaría que tuvieran que realizarse más operaciones de las necesarias para completar un procesamiento.

- Las longitudes de palabras utilizadas en la mayoría de los algoritmos no son acordes con las longitudes de palabras de las actuales arquitecturas del procesador, esto agregría otra deficiencia en el desempeño.

- La robustez de un algoritmo criptográfico está basado en el número de iteraciones de una estructura básica y los algoritmos anteriores no son la excepeción. Éstos presentan operaciones que se repiten una gran cantidad de veces. Esto provoca que sean ejecutados todavía más ineficientemente.

Esta claro que una implantación en hardware, mejoraría el desempeño de tales algoritmos. Sin embargo, se quiere mantener cierta flexibilidad por lo cual se sugiere una implantación en hardware-software.

Capítulo 2

Los sistemas Hardware-Software

En los últimos diez años se ha visto un cambio dramático en el modo en que es realizado el diseño digital. La necesidad de reducir el tiempo en el mercado, los avances de tecnología y nuevas e innovadoras herramientas de software EDA (Electronic Design Automation), factores todos que han ayudado a abastecer este dramático cambio.

En este capítulo se hará una revisión breve sobre las tecnologías y herramientas utilizadas en el diseño de hardware actual, se hablará en especial de los FPGA's, el lenguaje VHDL y la síntesis de circuitos. Además, se revisarán los conceptos más importantes de los sistemas hardware-software, así como la separación de las partes que deberán ir tanto en hardware como en software. Se revisan también algunas implementaciones que se han realizado de estos sitemas. Por último se describirán algunas de las diferencias que hay entre desempeño y costo cuando se realizan diseños de aplicaciones en software y hardware,

2.1 Tecnologías para implantaciones de sistemas digitales.

Distinto a las generaciones previas de tecnología, en que el diseño a nivel de tarjeta incluía grandes números de chip's de pequeña escala de integración (SSI - small scale integration) conteniendo compuertas básicas, actualmente todos los diseños digitales producidos consisten en su mayor parte por dispositivos de alta densidad. Los avances logrados en la tecnología de muy alta escala de integración ó VLSI (Very Large Scale Integration), han permitido que los transistores puedan ser manufacturados con anchos de canales de submicras, resultando en tamaños reducidos (100 veces más pequeños que el espesor de un cabello humano) y logrando altas velocidades de conmutación. Esto ha conducido a que los circuitos integrados de silicio contengan comunmente millones de transistores (por ejemplo los microprocesadores contienen más de 7 millones de transistores) [GVL99] e implementados dentro de ellos grandes y complejos sistemas [Smi01].

El estándar de los circuitos integrados tiene una operación funcional fija definida por el fabricante del chip. Contrario a esto, tanto los circuitos integrados de aplicación específica (ASIC's - Application Specific Integrated Circuits) como los arreglos de compuertas de cam-

pos programables (FPGA's - Field Programable Gate Arrays), son tipos de circuitos integrados cuya función no esta fijada por el fabricante. La función es definida por el diseñador para una aplicación particular. Un ASIC requiere un proceso de manufactura final para personalizar su operación mientras que un FPGA no lo necesita.

2.1.1 ASIC's: Circuitos integrados de aplicación específica.

Un circuito integrado de aplicación específica es un dispositivo que es parcialmente manufacturado por un fabricante de ASIC's en forma genérica. Este proceso inicial de manufactura es el más complejo, consume más tiempo y una gran parte del proceso de fabricación total. El resultado es un chip de silicio con un arreglo de transistores desconectados.

El proceso de fabricación final, que consiste en conectar las juntas de los transistores, se terminará cuando los diseñadores de un chip tengan un diseño específico que deseen implementar en el ASIC. Un fabricante de ASIC puede normalmente hacer esto en un par de semanas y esto es conocido como el tiempo "de vuelta". Hay dos categorías de dispositivos ASIC: el arreglo de compuertas y las celdas estándar, que se describen a continuación:

- Arreglo de compuertas. Hay dos tipos de arreglos de compuertas: un arreglo de compuertas de canal y un arreglo de compuertas sin canal. Un arreglo de compuertas de canal es fabricado con renglones simples o dobles de celdas básicas entre el silicio. Una celda básica consiste de un número de transistores. Los canales entre los renglones de celdas son usados para interconectar las celdas básicas durante el proceso final de personalización. Un arreglo de compuertas sin canal es manufacturado con un "mar" de celdas básicas entre todo el silicio y no hay canales dedicados para interconexiones. El arreglo de compuertas contiene de unos cuantos miles de compuertas equivalentes a cientos de miles de compuertas equivalentes. Debido al espacio limitado por el enrutado en los arreglos de compuertas con canal, típicamente solo entre el 70% y 90% del número total de compuertas disponible pueden ser usadas.
- Celdas estándar. Los dispositivos de celda estándar no tienen el concepto de una celda básica y sus componentes no están prefabricados en el chip de silicio. El fabricante crea mascaras a la medida para todos los estados de los procesos del dispositivo y permite utilizar mucho mas eficientemente el silicio que en el arreglo de compuertas.

2.1.2 FPGA's: Arreglos de compuertas de lógica programable

El arreglo de compuertas de lógica programable es un circuito integrado que es completamente manufacturado, pero le falta un diseño independiente. Combina el concepto de dispositivos lógicos programables con el concepto de tecnología de arreglo de compuertas tradicional [Tri93]. Para configurar una operación funcional en particular dentro del dispositivo, se programan matrices de conmutación que rutean las señales entre los bloques de lógica individual.

Estos dispositivos tienen un gran aumento de densidad de transistores comparada con la de los procesadores de propósito general [GPI], como se puede ver en la Figura 2.1. Los

nuevos FPGAs pueden soportar circuitos con cerca de 500, 000 compuertas equivalentes en el mismo dispositivo y ahora existen mejores FPGA's con un millón de compuertas, tal como la serie Virtex(tm) de Xilinx [XDB99]. Esta tecnología hace posible el concepto de hardware ilimitado o hardware virtual.

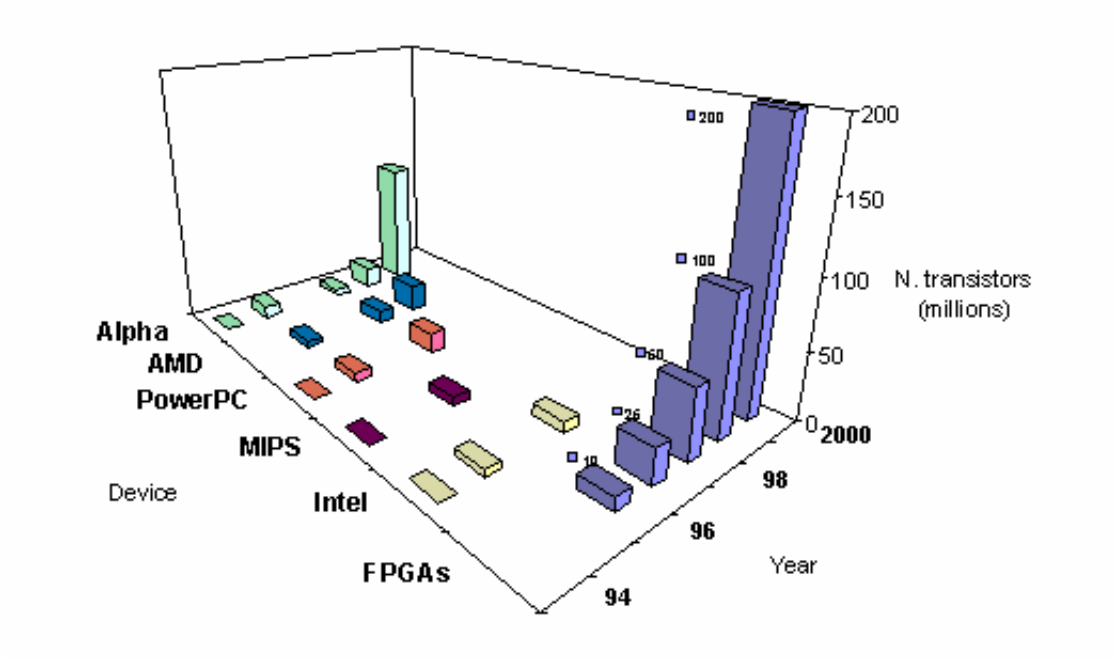

Figura 2.1: El aumento de la densidad de transistores en procesadores y dispositivos FPGA.

Los FPGA's normalmente consisten de grandes arreglos de dos dimensiones de bloques lógicos programables que pueden ser conectados a través de matrices de interconexiones también programables, a través de otros elementos lógicos usando algunos esquemas de interconexiones fijas, o una combinación de los dos. La figura 2.2 muestra una estructura básica de un FPGA. Cada fabricante de FPGAs fabrica dispositivos para una arquitectura propietaria, por lo que la funcionalidad de los elementos lógicos varía de una arquitectura de FPGA a otra; por ejemplo, algunos son capaces de ejecutar únicamente ciertas funciones de dos o tres variables boolenas [NS93] mientras que otros son capaces de ejecutar todas las funciones de seis variables boolenas [ACC+96]. Como con los esquemas de interconexión de los FPGAS, los elementos lógicos por sí mismos pueden ser programados para realizar funciones definidas por el usuario lo que significa que pueden ser usados para implementar casi cualquier circuito digital.

Las dos formas principales de tecnología de programación de FPGA son la tecnología antifusible y la tecnología SRAM. Los FPGA antifusible [GHB93] emplean lo que son llamados antifusibles que proporcionan la programabilidad de los dispositivos. Un antifusible es un dispositivo de dos terminales que esta normalmente desconectado o abierto, por ejemplo en un estado de alta impedancia. Cuando se quema, el dispositivo toma una baja resitencia y actúa realmente como un conductor entre sus dos terminales. Una vez que esta quemado, un antifusible queda permanentemente en estado conductor, así que estos FPGA's son esencialmente

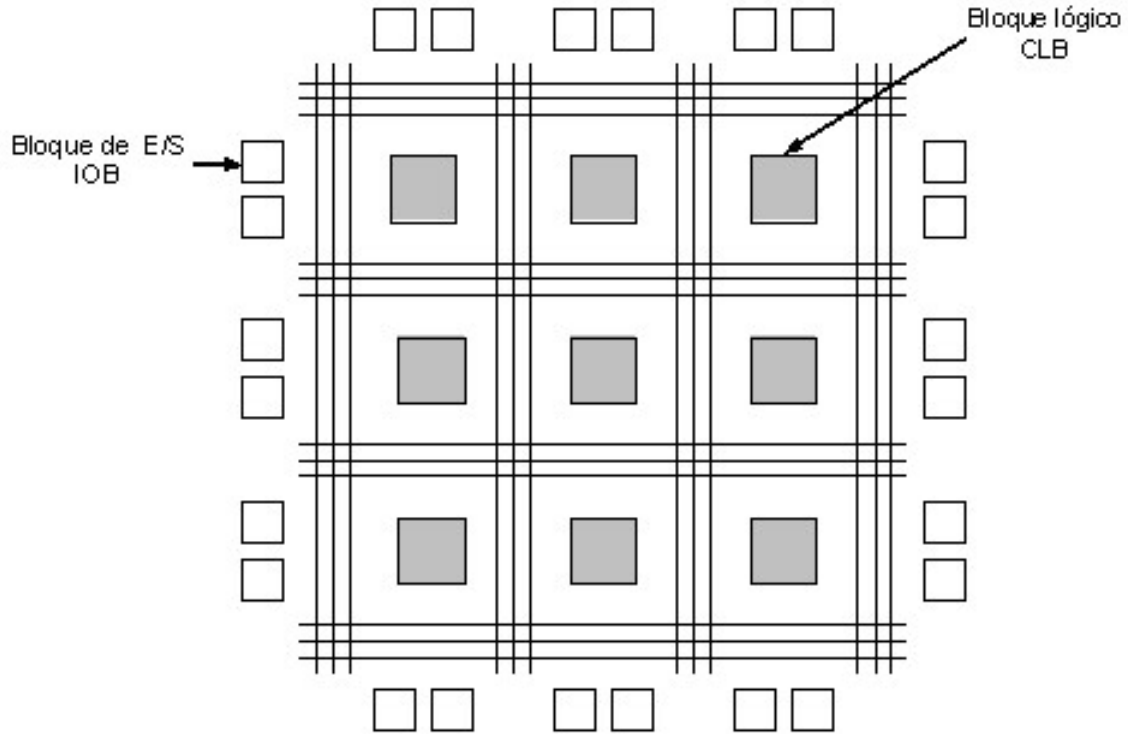

Figura 2.2: Estructura de un FGPA

dispositivos programables de una vez. En el mayor de los casos, estos antifusibles controlan los multiplexores del chip para cambiar las funciones del dispositivo. Los antifusibles son apropiados para FPGAs porque pueden ser construidos usando tecnología CMOS modificada.

En contraste, los FPGA's basados en SRAM [Tri93] [NS93] [Alt93] pueden ser programados múltiples veces, puesto que las características configurables, o programables, de los dispositivos son controladas cada una por bits SRAM que se distribuyen a través del FPGA.

Los bits SRAM son usados para controlar la activación o desactivación de transistores de paso y buffers que sean necesarios. Por medio del control de éstos las señales pueden ser ruteadas de elementos lógicos a elementos lógicos. Algunos FPGA's se caracterizan por la interconexión local únicamente entre los elementos lógicos mientras otros proporcionan otras rutas de comunicación de gran distancia entre elementos. Un ejemplo del uso de interruptores controlados por SRAM se ilustra en la Figura 2.3, mostrando dos aplicaciones de celdas SRAM: (1) para controlar los interruptores en los nodos de compuerta de los transistores de paso y (2) para controlar las líneas seleccionadas de multiplexores que manejan entradas de bloques lógicos. La Figura da un ejemplo de la conexión de un bloque lógico (representado por la compuerta AND en la esquina superior izquierda) a otro por medio de dos interruptores de transistor de paso, y después un multiplexor, todo controlado por celdas SRAM. Si un FPGA usa transistores de paso o multiplexores o ambos depende del producto en particular.

No sólo se utiliza el SRAM para controlar las estructuras de interconexión, los bits SRAM

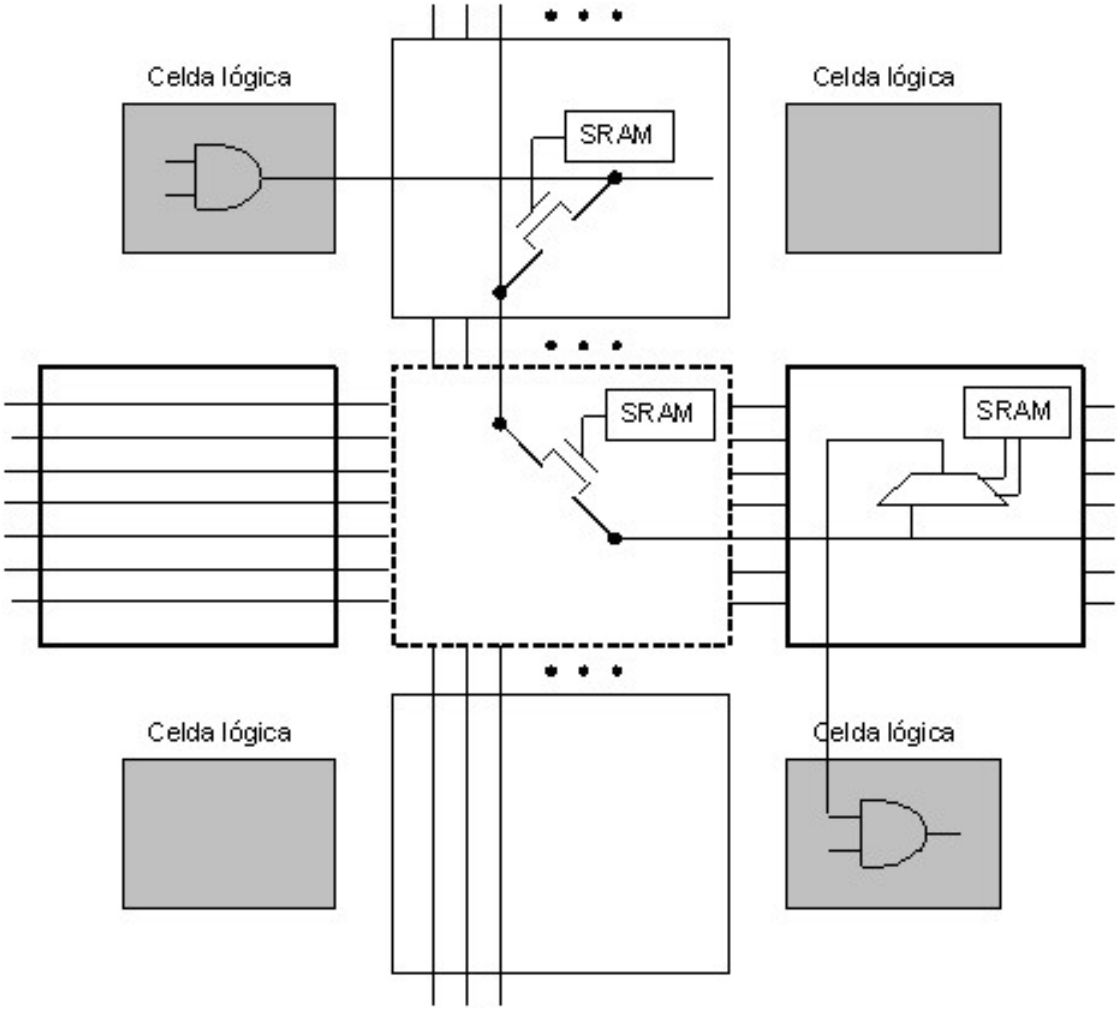

Figura 2.3: Interruptores programables controlados por SRAM.

son también usados para controlar las funciones de los elementos lógicos que se encuentran dentro de los bloques. Los elementos lógicos generalmente consisten de cualquiera de los dos tipos: tablas de consulta (Look-Up Tables - LUTs) creados con bits de SRAM o una cierta combinación de funciones de compuerta fija con varios mutliplexores para poder combinar las compuertas de diversas maneras. Para programar los LUTs, los bits de la SRAM se fijan al mismo tiempo que se programa el dispositivo. Para la otra forma de elementos lógicos, la configuración de bits SRAM son usados para controlar a los multiplexores dentro de los elementos lógicos y así establecer la función de los elementos. Los elementos lógicos de cualquiera de los tipos frecuentemente incluyen latches o flip-flops que pueden ser empleados en las aplicaciones de usuarios.

Para explicar la estructura de un bloque lógico se considera un producto de la familia XC4000 de Xilinx debido a su gran popularidad. La XC4000 se caracteriza por un bloque lógico configurable (llamado por Xilinx CLB o configurable logic block) que está basado en tablas look-up (LUT's). Una LUT es un pequeño arreglo de memorias de tamaño de un bit,

donde las líneas de dirección para la memoria son entradas del bloque lógico y la salida de un bit de la memoria es la salida de la LUT. Una LUT con K entradas entonces corresponde a una memoria de $2^k \times 1$ bit, y puede realizar alguna función lógica de sus K entradas por programación de la tabla de verdad de la función lógica directamente en la memoria. El CLB de la XC4000 contiene tres LUT's separadas, como se muestra en la configuración de la Figura 2.4. Hay dos LUTs de 4-entradas, denominadas funciones F y G, que son alimentadas por las entradas del CLB, y una tercera LUT de 3-entradas, denominada función H, que puede ser usada en combinación con las otras dos. Este arreglo permite al CLB implementar un amplio rango de funciones lógicas de hasta nueve entradas, dos funciones separadas de cuatro entradas u otras posibilidades. Cada CLB también contiene dos flip-flops.

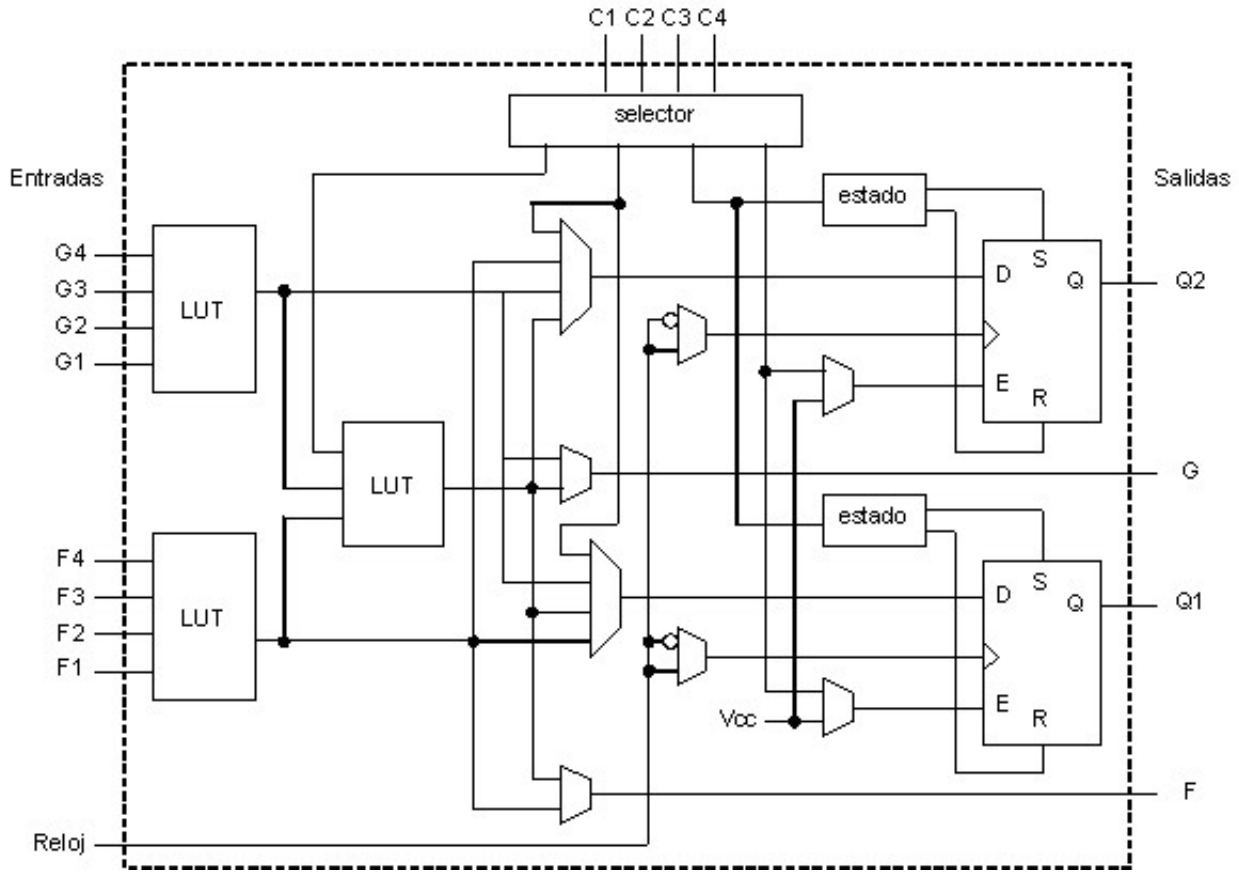

Figura 2.4: Bloque de lógica configurable de la familia XC4000 de Xilinx.

La otra característica clave que caracteriza un FPGA, además de la lógica, es su estructura de interconectado. El interconectado de la XC4000 es un arreglo de canales horizontales y verticales. Cada canal contiene un pequeño número de segmentos de alambres cortos que atraviesan un sencillo CLB (el número de segmentos en cada canal depende del número específico de componente), largos segmentos que atraviesan dos CLBs, y muy grandes segmentos que atraviesan la longitud entera del ancho del chip. Los interruptores programables están disponibles para conectar las entradas y salidas del CLB a los segmentos de alambre (Figura 2.3), o para conectar un segmento de alambre a otro. Una pequeña sección representativa de

canales de ruteo de un dispositivo aparece en la Figura 2.5. La figura muestra únicamente el segmento de alambre en un canal horizontal, y no muestra el canal de ruteo vertical, las entradas del CLB y las salidas, o los interruptores de ruteo. Un importante punto que vale mencionar, acerca de la interconexión de Xilinx, es que las señales pueden pasar a través de interruptores para comunicar un CLB con otro, y el número total de interruptores que se atraviesan depende del conjunto particular de segmentos de alambre usado. Entonces, el desempeño en velocidad de la implementación de un circuito depende en parte en cómo los segmentos de alambre son asignados a las señales individuales por la herramienta CAD.

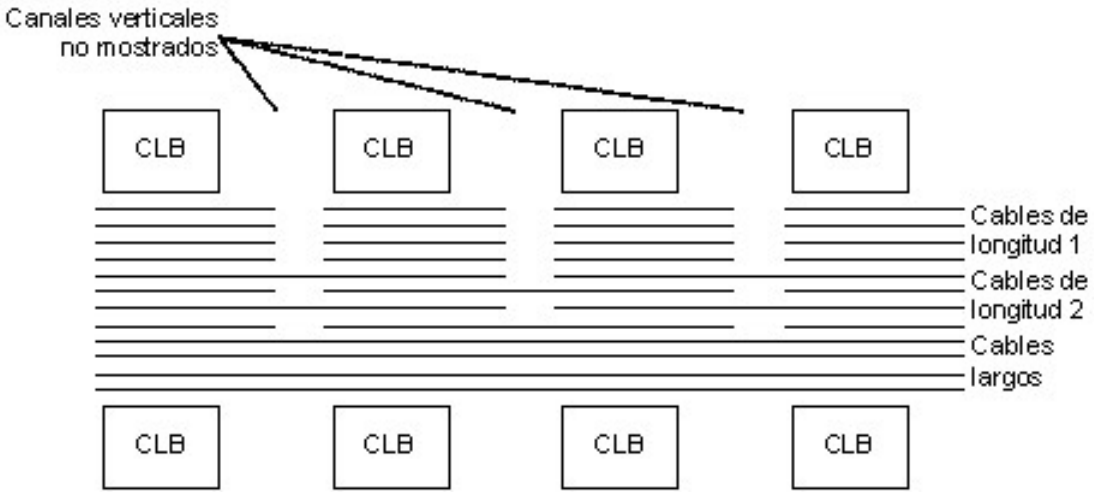

Figura 2.5: Segmentos de cable del Xilinx XC4000.

La tabla 2.1 muestra varios modelos de FPGA's con sus respectivas matrices de CLB's, IOB's y densidades en compuertas equivalentes.

Modelo (Fabricante)	Matriz de CLB's	IOB's	Compuertas equivalentes
XC4010E (Xilinx)	20x20	160	7,000 - 20,000
FLEX 8000 (Altera)	Menor de 84	-	4,000 - 15,000
ORCA 2 (AT&T)	-	-	40,000
XCV300 Virtex (Xilinx)	32x48	316	322,970
QL6500 (QuickLogic)	72x56	448	488,064

Tabla 2.1: Modelos de FPGA's con sus respectivas densidades

Una de las áreas propicias para aplicaciones de los FPGA, que esta comenzado a desarrollarse, es el uso de FPGAs como máquinas de cómputo a la medida. Esto involucra el uso de componentes programables para ejecutar software, en vez de compilar el software para ejecutarlo en un CPU normal. El lector interesado puede consultar el FPGA-based Custom Computing Machines Workshop, publicado por la IEEE en los últimos años.

Los diseños de FPGA's por sí mismos sufren en términos de velocidad de ejecución y uso del área de silicio usada, en comparación con los ASIC [ACC94]. El desempeño de un diseño FGPA es más lento que el de los C.I. a la medida por tres principales razones:

- Cada uno de los niveles de lógica requeridos agrega retrasos. Los elementos lógicos que contiene un FPGA comunmente tienen de dos a cuatro entradas, lo que significa que puedan requerir más niveles para implementar un circuito.

- La circuitería adicional de los propios elementos lógicos, tal como multiplexores y LUTS SRAM, provoca que sean más lentos.

- El enrutado de las señales dentro del chip. Las señales deben pasar a través de varios conmutadores programables y buffers mientras viajan entre elementos lógicos. La capacitancia y las resistencia parásita también se agregan a las estructuras de ruteo, retardando las señales incluso si la estructura de conmutación no es usada.

Los diseños mapeados en un FPGA son descompuestos en piezas lógicas de tamaños de bloque y distribuidos a través del área del FPGA. Entonces, el desempeño de los FPGA algunas veces depende más de los niveles más altos del diseño, es decir, de cómo las herramientas CAD mapean el circuito dentro del chip.

2.2 Herramientas de síntesis digital.

En el campo de la realización de sistemas digitales existen dos líneas que han seguido una evolución muy similar; por un lado la tecnología de realización de circuitos, y por otro, la metodología de diseño de estos circuitos. Se puede decir que la evolución de la tecnología se ha desarrollado hacia la alta escala de integración y los circuitos de lógica programable, como los FPGA's, mientras que la evolución de la metodología ha llevado a la necesidad de utilizar lenguajes de descripción de circuitos, como el VHDL (Very High Speed Integrated Circuit Hardware Description Language - lenguaje de descripción hardware de circuitos integrados de muy alta velocidad). Ambas líneas, metodología y tecnología, van necesariamente unidas; sin la aparición de la lógica programable, y el abaratamiento de fabricación de los circuitos integrados, no hubieran estado tan disponibles las herramientas que, a partir de una descripción mediante lenguaje, permiten la realización de un circuito. Por otro lado, sin la evolución de la metodología de diseño, no hubiera sido posible integrar y abordar diseños con la complejidad que los dispositivos actuales permiten [PB00].

Para lograr competir en la construcción de sistemas electrónicos, se está optando por usar metodologías de diseño top-down que incluye el uso de lenguajes de descripción de hardware y síntesis, además recientemente el proceso más tradicional de simulación[Smith01].

2.2.1 Metodología de diseño Top-Down

En años recientes, los diseñadores han adoptado cada vez más metodologías de diseño top-down incluso a pesar de que lo hacen en programación abstracta, lejos de los diseños lógicos y lejos del nivel de transistores. La introducción de estándares de lenguajes de descripción de hardware en la industria y herramientas de síntesis, comercialmente disponibles, han ayudado a establecer esta revolucionaria metodología de diseño. Las ventajas son claras y los métodos de ingeniería de diseño pueden cambiar. Algunas de estas ventajas son:

- Incrementa el campo de la productividad al reducir los ciclos de desarrollo, con mejores características de producto y reduciendo el tiempo de venta.
- Reduce los costos de ingeniería no recurrente (NRE - Non Recurring Engineering)
- El rehúso de diseños es posible
- Incrementa la flexibilidad para cambios en el diseño.
- Exploración rápida de arquitecturas alternativas
- Exploración rápida de librerías de tecnología alternativa.
- Habilita el uso de síntesis para dispersar rápidamente el diseño en el espacio de área y en la sincronización y generar automáticamente circuitos de prueba.
- Mejora y facilita la auditoria y verificación del diseño

Una metodología de diseño top-down toma el modelo de hardware en algún lenguaje de descripción de hardware (HDL), escrito en un alto nivel de abstracción de comportamiento (sistemas o algoritmos), descendiendo a través de los niveles intermedios, haciendo uso de herramientas automatizadas de particionamiento y síntesis, hasta el nivel más bajo (compuerta o transistor). La sintesis es el enlace clave entre cada etapa. Ver Figura 1.2.

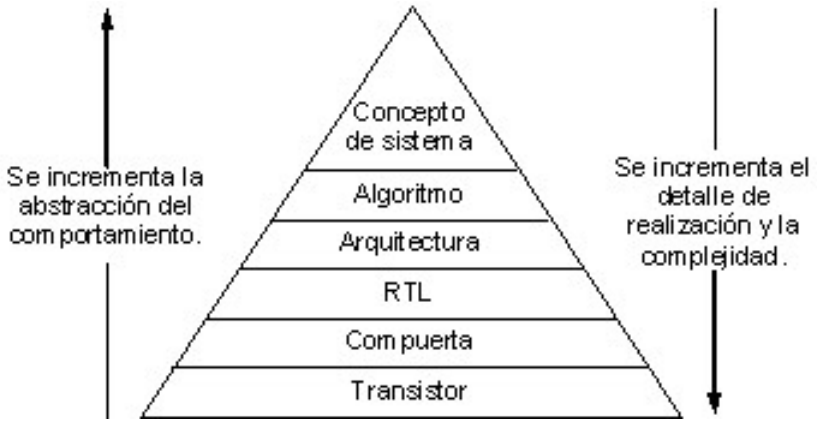

Figura 2.6: Piramide de abstracción del nivel de comportamiento.

Como los modelos en hardware son trasladados a niveles más bajos progresivamente estos llegaran a ser más complejos y contendrán mas detalles de estructura. El beneficio de modelar hardware en altos niveles de abstracción de comportamiento es que el diseñador no se preocupa de grandes cantidades de detalles innecesarios y la complejidad de la tarea de diseño se reduce. Actualmente se gasta mucho tiempo diseñando modelos HDL, considerando diferentes arquitecturas y considerando pruebas de sistema y problemas de prueba. Prácticamente no se gasta tiempo diseñando a nivel de compuertas.

Un ciclo tipíco de diseño de hardware se muestra con detalle en la Fig 2.7. Observe que se incluye una fase de simulación y comprobación de circuitos utilizando herramientas de diseño

asistidas por computadora (CAD - Computer Aided Design), de forma que no es necesario realizar físicamente un prototipo para comprobar el funcionamiento del circuito y si fuera necesario este se presentaría al final del proceso, evitando la repetición de varios prototipos y economizando así el ciclo de diseño [PB00].

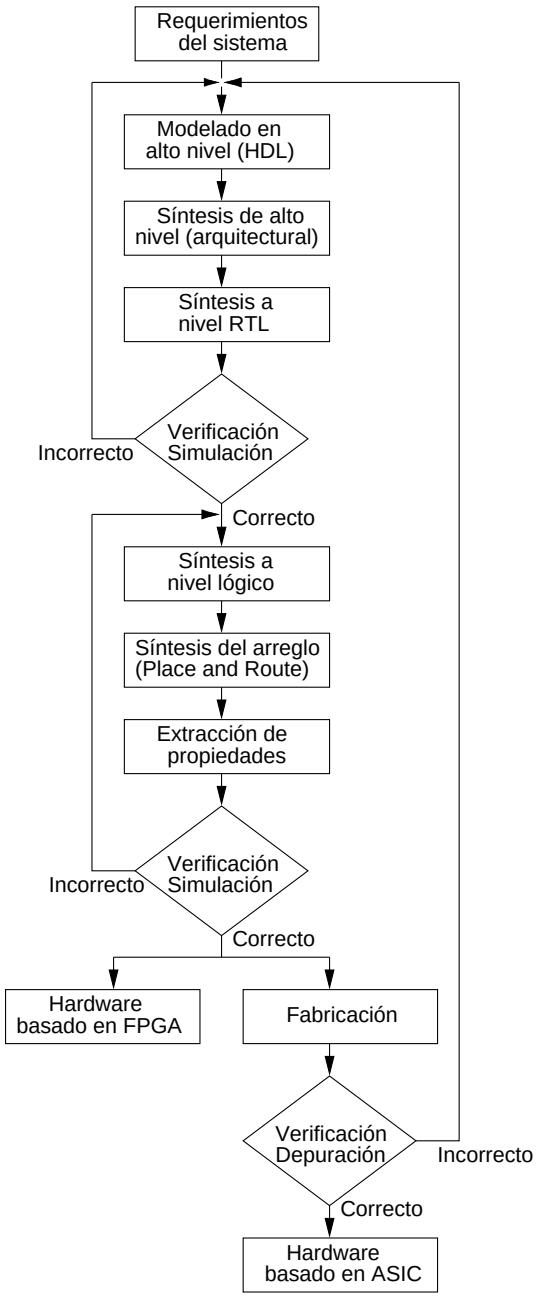

Figura 2.7: Flujo de diseño para sistemas digitales.

2.2.2 VHDL: Lenguaje de descripción hardware de circuitos integrados de muy alta velocidad

La pieza clave, en cualquier metodología actual de diseño de circuitos, es la especificación y descripción del diseño. La captura de esquemas es útil para circuitos sencillos, pero ya no sirve si se pretende abordar un sistema realmente complejo con miles de elementos. Se hace necesario, por tanto, la utilización de un lenguaje para la especificación del hardware [PB00].

Existen muchos lenguajes para la descripción de circuitos digitales, pero VHDL está alcanzando mayor popularidad debido a que nació a partir de los estándares de descripción de circuitos: 1076 de la IEEE (Institute of Electrical and Electrónics Engineers) y NIL-STD-454L por el departamento de Defensa de Estados Unidos [VHD99]. El resto de lenguajes de descripción suelen ser propios de una determinada herramienta o fabricante de chips, por ello resultan a veces más óptimos. VHDL, aparte de ser un estándar, tiene un amplio campo de aplicación, desde el modelado para simulación de circuitos, hasta la síntesis automática de circuitos [PB00].

VHDL divide las entidades (componentes, circuitos o sistemas) en una parte visible o externa (nombre de la entidad y conexiones) y una parte oculta o interna (algoritmo de la entidad e implementación). Después de definir la interfaz externa, otras entidades pueden usarla cuando ya haya sido completamente desarrollada. Este concepto de vista externa e interna es central para una perspectiva de diseño de sistemas en VHDL. Una entidad es definida, en relación a otras, por sus conexiones y comportamiento. Se pueden explorar implementaciones externas (arquitecturas) de una de ellas sin cambiar el resto del diseño. Después de definir una entidad para un diseño, se puede reusar este en otros tanto como sea necesario, incluso se pueden desarrollar bibliotecas para usarse hasta con una familia de diseños. Un modelo hardware VHDL es mostrado en la Figura 2.8.

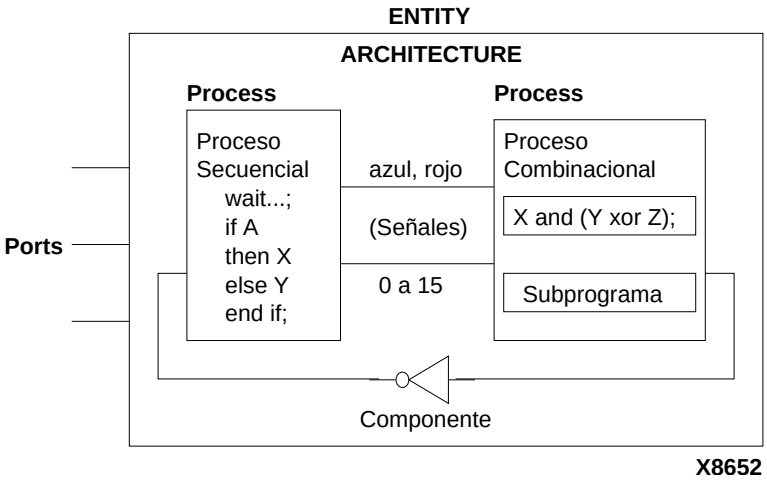

Figura 2.8: Un modelo VHDL de hardware

Una *entidad* (diseño) VHDL tiene uno o más puertos de entradas, salidas o entrada/salida

que son conectados (cableados) a sistemas vecinos. Ésta se forma de *procesos* y *componentes* interconectados, todos operando concurrentemente. Cada entidad es definida por una *arquitectura* particular, que se compone de construcciones VHDL tal como operaciones aritméticas, asignaciones de señal, o declaraciones de instanciación de componentes.

En VHDL, los circuitos de modelo secuencial en procesos independientes (sincrónos), usan flip-flops y latches, y los circuitos combinacionales (asincrónos) usan únicamente compuertas lógicas. Los procesos pueden definir y llamar (instanciar) *subprogramas* (subdiseños). Los procesos se comunican con cada uno de los otros procesos mediante *señales* (alambres).

Una señal tiene una fuente (manejador), uno o más destinos (receptores), y un *tipo* definido por el usuario, tal como "color" o "número entre 0 y 15".

VHDL proporciona un amplio conjunto de constructores. Con VHDL, se pueden describir sistemas electrónicos discretos de una complejidad variable (sistemas, tarjetas, chips o modelos) con diferentes niveles de abstracción.

Las construcciones del lenguaje VHDL están divididas en tres categorías por su nivel de abstracción: *comportamiento, flujo de datos,* y *estructural.* Estas categorías se describen enseguida.

- Comportamiento. Es posible describir un circuito electrónico, generalmente digital, simplemente describiendo el comportamiento funcional o algorítmico del diseño, expresado en un proceso secuencial VHDL.
- Flujo de datos. Se ven a los datos como un flujo a través del diseño, de la entrada a la salida. Una operación es definida en términos de una colección de transformación de datos, expresado como una sentencia concurrente.
- Estructural. La visión más cercana al hardware; un modelo donde los componentes de un diseño se enumeran y se interconectan. Esta expresada por instanciación de componentes.

2.2.3 Sintesís automática de circuitos.

El lenguaje VHDL, es también, un lenguaje de descripción para síntesis. La síntesis de un circuito, a partir de una descripción VHDL, consiste en reducir el nivel de abstracción de la descripción del circuito hasta convertirlo en una definición optimizada puramente estructural, a nivel de compuertas, cuyos componentes son elementos de librerías de una tecnología especifíca, ver Figura 2.9. Al final del proceso de síntesis se debe obtener un circuito que funcionalmente se comporte igual que la descripción que de él se ha hecho. Las posibilidades de realización física de la descripción VHDL, abarca una amplia variedad de dispositivos, entre ellos los FPGA's, las PAL's (Programmable And Logic - Lógica y programabilidad), PLD (Programmable Lógic Device - Dispositivos lógicos programables) y CPLD (Complex PLD - PLD Complejo) [PB00].

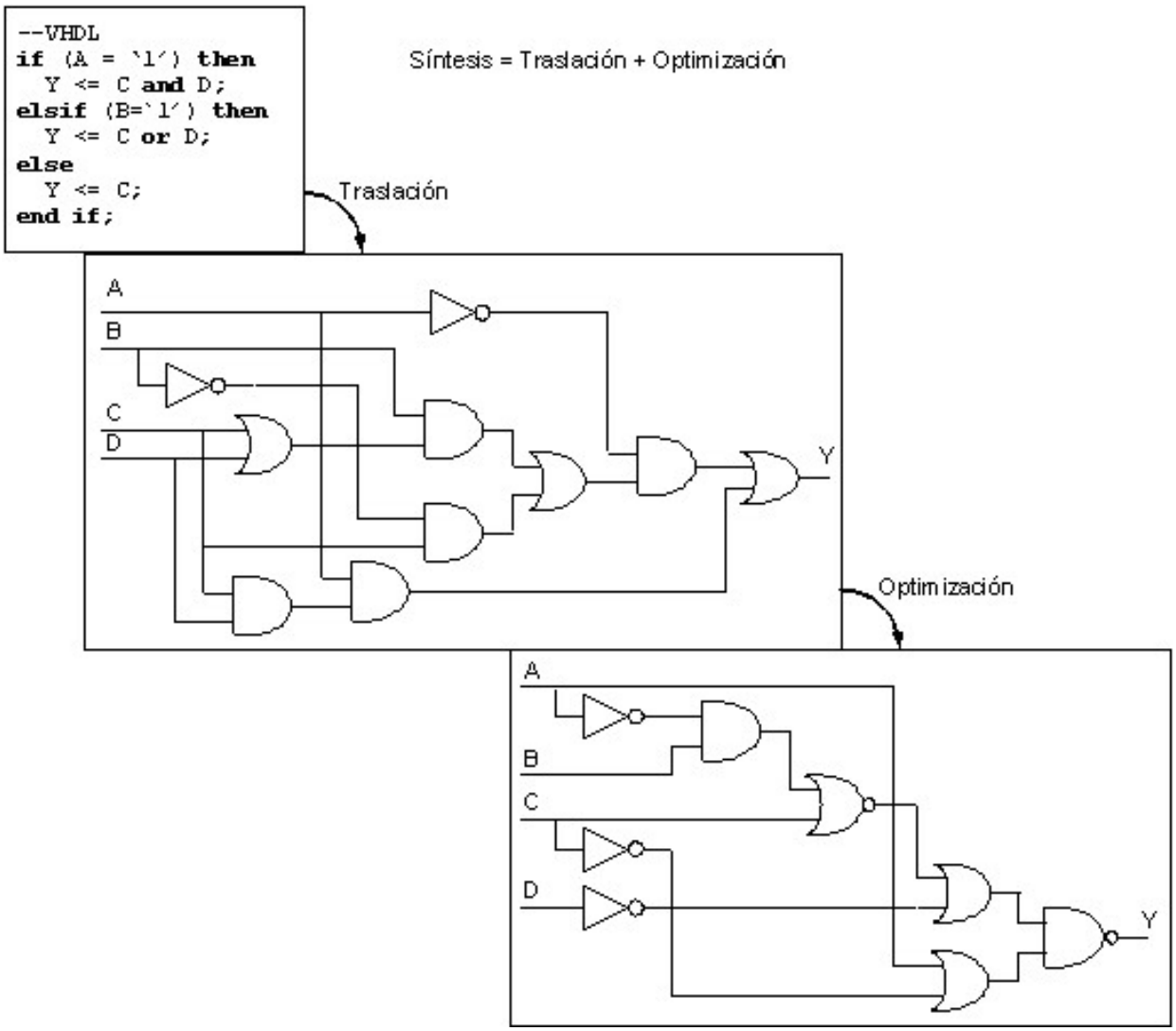

Figura 2.9: Síntesis igual a traslación y optimización.

Una herramienta de software de síntesis automatiza esta parte del proceso de diseño en los ASIC y FPGAs y forma el enlace central en una metodología de diseño top-down. La síntesis es por mucho el medio más fácil y rápido de diseñar y generar circuitos [Smi01], aunque no siempre es el modo más eficiente. Un flujo de proceso de síntesis típico usando una herramienta de síntesis es mostrada en la Figura 2.10. Se muestra una traslación inicial a una netlist sin optimización. En la práctica, una optimización fundamental de alto nivel es ejecutada, pero transparente al usuario. Este proporciona el punto de inicio en la curva área-tiempo para la optimización. La Figura 2.10 muestra un diseño optimizado tres veces con tres diferentes restricciones establecidas para producir tres diferentes puntos en la curva de área-tiempo. La metodología de optimización típica es optimizar primero para área, y entonces solamente optimizar para tiempo si alguna restricción de tiempo no fue reunida. Bloques jerárquicos en un gran diseño son normalmente optimizados iniciando del bloque de nivel más bajo en un proceso bottom-up.

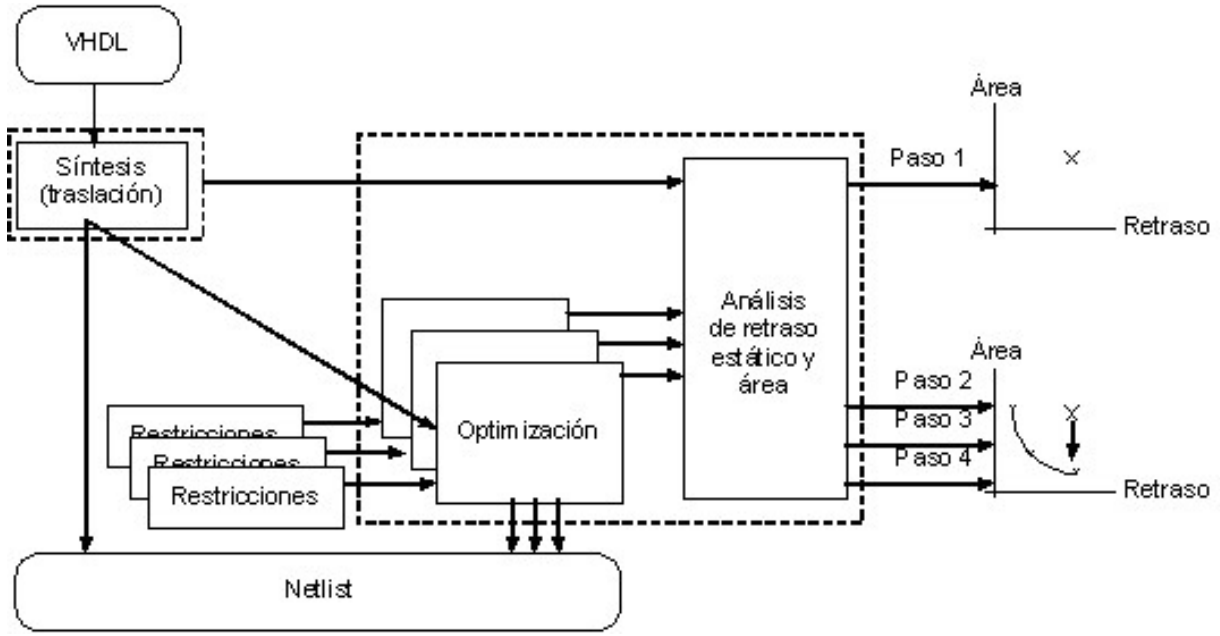

Figura 2.10: Flujo de proceso de traslación y optimización usando síntesis

La síntesis consiste de múltiples estados de transición y optimización. A un diseño le toma pasar a través de tres principales niveles internos de refinamiento interno (abstracción): el nivel RTL, el nivel lógico y el nivel de compuerta. La optimización automática ocurre en cada uno de estos niveles intermedios y es guiado por restricciones definidas por el usuario. Las restricciones proporcionan el objetivo que los procesos de traslación y optimización intentan reunir. Las herramientas actuales de síntesis permiten establecer típicamente restricciones para menor área y menor tiempo de retardo. La optimización de una herramienta de síntesis es un proceso heurístico que usa diferentes algoritmos basados en prueba y error para encontrar una implementación de circuito que mejor se ajuste a las restricciones. Un circuito optimizado para el área mínima tendría un área mínima basada en lo que el optimizador debe encontrar. Este puede no ser siempre el circuito mínimo absoluto que sería producido si el diseño fuera cuidadosamente diseñado a mano.

En este contexto, las herramientas de síntesis Synopsys son mucho más poderosas, un resultado interesante de cómo Synopsys puede mejorar el desempeño de ejecución de un circuito se puede ver en [JP98]. Otra ventaja de Synopsys es la capacidad de ejecutar archivos script. Los pasos necesarios para sintetizar y optimizar un diseño, preparar resúmenes y especificar los parámetros de configuración, pueden ser incluidos en un archivo script.

2.2.4 Simulación.

La simulación es la parte fundamental y esencial del proceso de diseño para algún producto basado en electrónica; no exactamente para dispositivos ASIC y FPGA. Para los dispositivos ASIC y FPGA, la simulación es el proceso de verificar las características funcionales de

modelos en algún nivel de comportamiento, esto es, desde altos niveles de abstracción bajando hasta niveles bajos. El arreglo básico para simulación es mostrado en la Figura 2.11.

Figura 2.11: Disposición de una simulación básica.

Un simulador es una herramienta de software de ingeniería asitida por computadora (CAE - Computer Aided Engineering) que simula el comportamiento de un modelo de hardware. Los simuladores usan la sincronización definida en un modelo HDL antes de la síntesis o la sincronización de las celdas de las bibliotecas de la tecnología destino, después de la síntesis. Un simulador puede ser un simulador funcional básico, un simulador de análisis de sincronización dinámica detallada o ambos. El simulador funcional básico es usado por la herramienta de síntesis durante la optimización por simple extracción de retardos de las celdas de la bilbioteca de la tecnología, mientras que el análisis de sincronización dinámica es usado en simulación para evaluar retrasos de sincronización a través del modelo más exacto.

Los sistemas digitales son verificados dos veces durante el proceso de diseño. Primero, se realiza una simulación, de los códigos fuente en VHDL, a Nivel de Transferencia de Registro (RTL - Register Transfer Level) llamada simulación funcional. Segundo, después del "place and route" llamada simulación lógica, ver Figura 2.7.

La simulación funcional, permite comprobar la funcionalidad de los circuitos en un caso ideal, es decir omitiendo detalles como retardos y sincronización de datos, producidos por la construcción inherente de un chip. Esta fase de diseño es el primer filtro que determina si el proceso debe continuar o se debe regresar al paso de especificación de entrada, para modificar los circuitos.

En cambio, la simulación lógica del circuito permite obtener el comportamiento del diseño más cercano a la realidad. Esta simulación permite detectar una posible falla o error lógico, como un conflicto de buses por ejemplo, antes de la programación final del dispositivo. Esta simulación hace uso de los retrasos de las señales y la sincronización de datos debido a los componentes utilizados en la costrucción física del chip.

Para cualquiera de las dos simulaciones anteriores, los mismos bancos de prueba pueden ser usados. El banco de prueba es un script que contiene el orden de los vectores de prueba y la sincronización de cómo estos son aplicados al diseño.

2.3 Implantaciones Hardware-Software

El microprocesador ha establecido los paradigmas comúnmente válidos para el diseño de hardware y software. El conjunto de instrucciones del procesador es la interfaz de los dos

mundos y a partir de éste los ingenieros en software tienen que expresar sus algoritmos. Debido a la estabilidad, similaridad y versatilidad de muchos conjuntos de instrucciones los programadores pueden usar lenguajes de alto nivel y compiladores así como interfaces convenientes. Los procesadores de propósito general, normalmente, son competentes al ejecutar operaciones relativamente complejas, dependiente de los datos y de tamaño variable [LWP94], debido a la flexibilidad de su conjunto de instrucciones permiten la implementación de casi todas las tareas computables, pero no están optimizados para aplicaciones específicas.

Los ASIC's son dispositivos de hardware especializado que son refinados para un muy pequeño número de aplicaciones o incluso para una tarea solamente, en donde logran un alto desempeño, requieren menos área de silicio, y consumen menos potencia que los procesadores programables a nivel de instrucción. Los ASIC son restringidos típicamente a sistemas, donde la demanda de alto desempeño computacional para una aplicación particular esta a la par con el costo de desarrollo para un volumen de producción razonablemente alto.

Hasta hace algunos años, cuando se implementaba un sistema de cómputo tradicional, se decidía entre una de las dos opciones, en hardware a la medida sobre ASIC's o en software sobre procesadores de propósito general. Con únicamente estas dos opciones, se conectaba el cómputo de hardware con procesamiento espacial y el cómputo de software con procesamiento temporal. La figura 2.12 representa la distinción entre cómputo espacial y temporal. En implementaciones espaciales, cada operador existe en un diferente punto en el espacio, permitiendo al cómputo explotar paralelismo para lograr alto desempeño de ejecución y bajo retardo de cómputo. En la implementación temporal, un pequeño número de recursos de cómputo más general son rehusados en el tiempo, permitiendo al cómputo ser implementado compactamente [DW99].

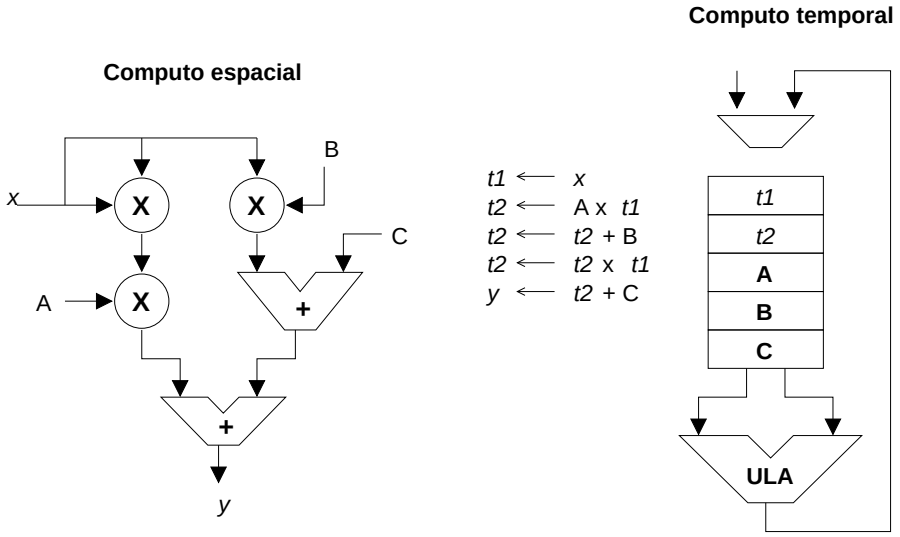

Figura 2.12: Cómputo espacial contra cómputo temporal para la expresión $y = \mathrm{A}x^2 + \mathrm{B}x + \mathrm{C}$.

El acoplamiento de hardware dedicado a una computadora host, como se muestra en la Figura 2.13, para acelerar algunas tareas de cómputo intensivo, no es un concepto nuevo. Un

ejemplo común en la industria de la PC es el uso de tarjetas para acelerar las transformaciones gráficas. Sin embargo, tal circuito acelerador en hardware de aplicación específica requiere gran esfuerzo de diseño y típicamente usa más área de silicio que el microprocesador por sí mismo.

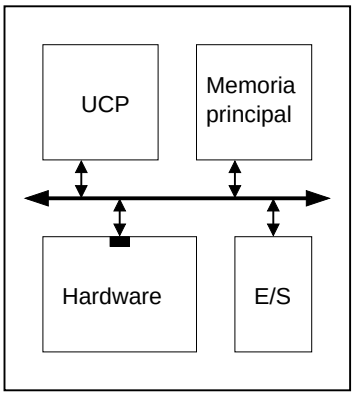

Figura 2.13: Un dispositivo hardware agregado a una arquitectura existente para propósitos de aceleración.

Un codiseño de sistema hardware-software contiene a ambos componentes. Avances recientes en lógica programable han permitido explotar la técnica de colocar algunas subtareas en hardware a la medida basados en FPGA, que son rápidos pero costosos, y colocar otras subtareas en software, con microprocesadores lentos pero baratos, permitiendo así, construir una buena implementación de aceleradores en hardware [LWP94].

Los microprocesadores son altamente optimizados para programas de propósito general, escritos en un lenguaje imperativo y compilado de manera eficiente. Por lo que no se puede esperar mejoras sustanciales de desempeño por el cambio de una implementación basada en microprocesador a una implementación basada en FPGA, para un programa dado. Sin embargo, muchos algoritmos de propósito especial y de cómputo intensivo, pueden ser implementados en un FPGA más eficientemente debido a la gran flexibilidad en asignar exactamente aquellos recursos necesarios en hardware.

Aplicaciones posibles de este paradigma incluyen visión por computadora y procesamiento de imágenes, donde la cantidad de datos es algunas veces grande como para manejarlo exclusivamente por el hardware a la medida.

2.3.1 Estrategia de particionamiento

Una regla general que muchas de las veces ocupa una pequeña cantidad de porción de código del software es implementar en hardware únicamente las regiones más críticas de una aplicación en software para mejorar su desempeño. Esta corresponde a una estrategia orientada a software para co-síntesis [KP98].

Un procesador de propósito general puede ser usado para "dividir" los datos de entrada en pequeños bloques de tamaño fijo y para "reunir" los resultados de los bloques procesa-

dos. El hardware de aplicación específica, normalmente de recursos limitados, es usado para "procesar" los datos particionados, algunas veces de un modo sistólico [Kun88].

La Figura 2.14 muestra el flujo total de diseño. Este inicia con una descripción de la aplicación en software de alto nivel. Como un primer paso a la optimización se identifican las secciones críticas en el código, mediante un análisis de perfil del código (profiling), usualmente en C. Una vez que el análisis ha sido hecho, el diseñador puede identificar los ciclos o las funciones que son más dominantes en el tiempo y por tanto las que se implantarían en hardware. El conjunto de regiones de código propuestos por los resultados del análisis son básicamente ciclos internos y funciones "leaf" altamente frecuentes que son particionados en hardware mientras el resto de la aplicación esta aún implementada en software, esta operación de particionado puede ser realizado ya sea por un compilador o manualmente.

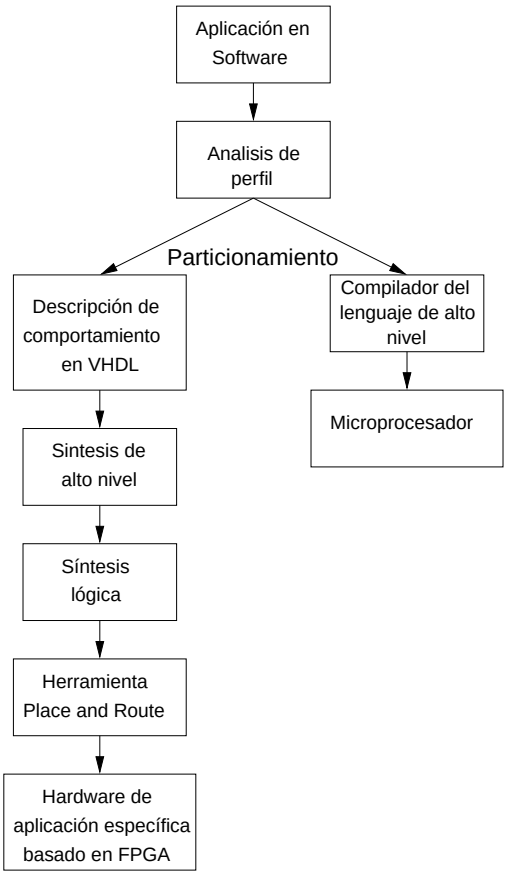

Figura 2.14: Flujo de diseño de un sistema hardware-software.

La fracción del tiempo de ejecución que un programa consume dentro de una región es llamado tamaño temporal del candidato y esta dado en un porcentaje del tiempo total de ejecución del programa [JEOH94].

El siguiente paso, es implementar una o más regiones de código fuente en algún lenguaje que pueda ser trasladado a hardware. Un modo de hacerlo es codificándolo como una descrip-

ción de comportamiento en VHDL. El comportamiento de un componente en hardware se describe como un algoritmo que es suficientemente similar a algún lenguaje de programación imperativo, tal como C, puesto que todos los datos de tipo simple y operadores pueden ser expresados usando los paquetes apropiados, como "Arithmetic", en VHDL. Esto permite un proceso de traslación directa. Si el software tiene un nivel muy alto de abstracción, o incluye operaciones de punto flotante o funciones de biblioteca que no son consideradas como regiones candidatas, entonces las partes en hardware tendrán que ser especificadas en una notación diferente al de las partes en software o en su defecto deben remplazarse por porciones de código con piezas de código equivalente.

El código VHDL generado es trasladado por un sistema de síntesis arquitectural o síntesis de alto nivel (HLS - High Synthesis Level), por ejemplo SYNT [Hem92, HP90], a una descripción estructural VHDL a nivel de registro de transferencia (RTL). Esta síntesis generará una ruta de datos (datapath), asignará unidades de hardware, calendarizará operaciones en pasos de control. Realizando así, una representación intermedia que puede ser una gráfica de flujo de datos o de control. Una biblioteca de operadores genéricos es responsable de proporcionar estimaciones realistas sobre métricas dependientes de la tecnología tal como área del módulo, retrasos, tiempo de reconfiguración, etc. en varios estados durante el proceso de síntesis.

Después la herramienta de síntesis lógica, Synopsis por ejemplo, ejecuta la optimización de bajo nivel. La síntesis lógica produce una descripción a nivel de compuerta de cada una de las configuraciones y la máquina de estados finito (MEF) correspondiente.

Finalmente la herramienta específica de "place and route" del vendedor ASIC transforma el diseño en archivos de configuración para el dispositivo de lógica de campo programable generando los datos de distribución de la reconfiguración.

La parte en software restante, usualmente constituirá el fragmento más grande y es usado un compilador típico de un lenguaje de alto nivel y ejecutado en el microprocesador [VSS+02] de la estación de trabajo host.

Una vez implementado en hardware, la única indicación que una parte del programa esta asociada a un circuito y ejecutada realmente en un hardware de aplicación específica es la velocidad de ejecución [VSS+02]. El programa principal invocará a tales regiones, de manera similar a una llamada a función.

Los métodos para un particionamiento apropiado de cómputo para la ejecución coordinada en hardware y software son esenciales con el propósito de generar eficientes implementaciones con una variedad de intercambios en hardware y software. Por otro lado, verificar la correcta interacción entre estos es un problema clave en el diseño de un sistema combinado.

El particionado óptimo puede ser obtenido cuando el hardware y el software procesan la ejecución completa en una longitud de tiempo similar. Considerando que los programas más especializados por naturaleza gastan aproximadamente todo su tiempo de ejecución en pequeñas áreas logrando tamaños grandes de candidatos temporales, una mejora en velocidad

puede ser obtenida si los paso en los que se "dividen" los datos para las máquinas de cómputo a la medida y se "reúnen" de ellos, ejecutados por el microprocesador pueden ser realizados en paralelo con el paso de "procesamiento" del hardware basado en FPGA. Otra opción sería rescribir programas para explotar mejor el hardware.

El desempeño de la implementación en software es medido durante la ejecución del programa. El costo y el desempeño de la implementación en hardware es medido una vez que este ya esta configurado en el dispositivo reconfigurable. Detalles sobre el particionado en hardware-software pueden ser encontrados en [FBK98].

2.3.2 Trabajos relacionados con sistemas hardware-software.

Koch y Golze describen una tarjeta co-procesadora [KG93]. Se enfocan en el diseño de la tarjeta y no han presentado un método para particionar e implementar un algoritmo dado.

El trabajo en la arquitectura Mark-I por Lewis y otros [LvIW93] tiene el éxito de acelerar una clase de programas definiendo un conjunto de instrucciones de aplicación específica que es ejecutado en una arreglo de 16 FPGAs de Xilinx. La definición de un conjunto de instrucciones de propósito especial para un programa dado es similar a la identificación de regiones candidatas y su implementación en FPGAs. Estos autores no tienen automatizado aún el particionamiento, pero se enfocaron al desarrollo de la arquitectura de la tarjeta.

Axel Jantsch y otros [JEOH94] presentan el análisis de un método completamente automático para acelerar software estándar en C o C++ haciendo uso de FPGAs. Las regiones críticas en tiempo son identificadas por medio de perfiles y son implementados automáticamente en lógica programable desde un muy alto nivel, utilizando herramientas de diseño de síntesis lógica. La arquitectura de trabajo que ellos utilizan, consiste de un tarjeta con lógica programable por el usuario, conectada a una workstation basada en Sparc vía el bus del sistema. Utilizan ocho programas en C como pruebas de rendimiento, logrando aceleraciones de hasta 11.9 cuando los datos se mapean a memoria local o registros dentro de los FPGAs.

Otros autores en [LWP94] presentan un método para particionar programas mediante un lenguaje funcional simple. También han automatizado la producción de programas particionados en este lenguaje. Además han probado su estrategía en un sistema basado en FPGA usando algoritmos de visión por computadora. Con el algoritmo de detectores de limites de Sobel, han alcanzado una aceleración de 20 veces cuando el overhead consitente en la transferencia de datos, el tiempo gastado en dividir y combinar los datos y el tiempo de las rutinas de sincronización del programa, no se ha incluído. Mientras que una aceleración de un factor de 2 se logra si se incluye este overhead. Una versión del algoritmo Canny para detectar limites, asistida en hardware, es 39% más rápida que la versión en software. Ellos estiman que se puede logra una aceleración de cerca de 300 veces, si el overhead debido a la comunicación con la PC y si los cuellos de botella de las entradas y salidas pueden ser eliminadas, dejando como la única limitante de velocidad el retraso de la ruta crítica, no obstante aclaran que

la ruta crítica del hardware puede ser reducida mejorando la disposición de los elementos dentro del dispositivo FPGA.

Jason Villareal y otros [VSS+02] reportan mejoras en el desempeño y en la energía consumida por los sistemas para un juego de pruebas de desempeño de aplicaciones multimedia. Logrando un rango de acelaraciones de 1.5 a 2.1 en sistemas con relojes de 100 a 200 MHz, mientras que logran beneficios de ahorro en consumo de energía en un rango de 12 a 44%. Además, presentan un flujo de diseño que busca automáticamente ciclos críticos en software que se remapean manualmente en lógica reconfigurable. Para hacerlo utilizan una variación del lenguaje C, llamado SA-C (Single Assignment in C), que está especificamente diseñado para generar hardware a partir de una codificación en C.

Muchas de las teorías fundamentales que se utilizan actualmente fueron identificadas por el campo de investigación de co-diseño Hardware/software [DG97]. Otros detalles como el particionamiento de sistemas hardware-software pueden ser encontradas en [FBK98]. Más aún, existen varias herramientas de particionamiento que se han enfocado al particionamiento automático de un programa entre un microprocesador y un procesador a la medida [HL98, Hen99, EPKD97, GKM98, KL97]. Estos métodos leen una especificación ejecutable en una representación interna, posiblemente anotada con determinadas frecuencias a través de un perfil, y entonces aplican heurísticas de particionamiento automático de esa representación.

2.4 Costo de desarrollo y desempeño: Hardware contra Software

Cuando un diseñador encara un problema que puede ser resuelto a través de software, hardware de aplicación específica o una combinación de ambos, los factores para decidirse por alguno de ellos son: considerar si el desempeño del sistema compensará el tiempo, la facilidad y, por tanto, el costo de desarrollo.

En muchos casos, desarrollar una aplicación en software es una tarea mucho más simple que desarrollar su equivalente en hardware a la medida. Esta facilidad se da por tres factores: el nivel de abstracción, la flexibilidad y la depuración.

- El software es creado en un nivel de abstracción mucho más alto, comparado con el nivel en el cual el hardware es diseñado. Los sistemas en software tienen que ocuparse de la secuencia y sincronización de interrupciones, co-rutinas, sistemas distribuidos e hilos (threads) concurrentes, pero no está involucrado en cuestiones relacionadas a hardware como sistemas de relojes, retrasos de reloj, niveles lógicos, el inherente paralelismo y la cantidad de concurrencia, que dan como resultado una carga de diseño mucho más grande debido al esfuerzo de calendarizar y organizar eventos.

- El software es bastante flexible. Crear o cambiar software es solamente manipular de alguna manera código escrito en algún lenguaje y procesarlo con compiladores, ensambladores, ligadores, y/o otras herramientas, lo que también provoca que se consuma

poco tiempo para comprobar su operación apropiada. La entrada y prueba del diseño de hardware puede requerir mucho mas tiempo, incluso si los chips son reprogramados, el tiempo requerido para ir de una nueva modificación del diseño desde que el nuevo diseño se introduce, se convierten los elementos primitivos del dispositivo en circuitos, se organizan en el chip tal que estos puedan ser conectables entre sí con cables internos, hasta producir los bits de programación de un dispositivo listo a usar puede tomar mucho más tiempo.

- Depurar software puede ser mucho más fácil que hacer lo mismo con hardware. Las herramientas para depurar software, pueden ofrecer visibilidad tal y como el procesador ejecutaría el software. En cambio, el hardware es observado a un nivel tan bajo como compuertas individuales, chips, y/o niveles de señal.

Considerando que el ciclo de diseño y depuración es frecuentemente un proceso iterativo, estas ventajas de desarrollo de software sobre su similar en hardware llegan a ser incluso más grandes e importantes. Esta y otras razones ayudan a explicar porque el software es el método preferido para realizar muchas aplicaciones, especialmente, tomando en cuenta el corto potencial de vida del producto.

Con las ventajas en costos de desarrollo del software sobre el hardware viene la desventaja de baja velocidad de ejecución total. Esta diferencia de desempeño es el resultado de tres elementos: paralelismo, eficiencia y especialización.

- El hardware puede construirse para aprovechar más eficazmente el paralelismo y la concurrencia que pudiera existir dentro de una aplicación lo que le da una ventaja de ejecución. El software se ejecuta de manera secuencial en un procesador programable, lo que le agrega facilidad de desarrollo. Por ejemplo, el hardware podría procesar todos los pares de elementos concurrentemente, asumiendo que no es un problema el ancho de banda de la memoria, mientras que, un método en software podría procesar pares de elementos secuencialmente y guardar el resultado en un tercer elemento.

- Una aplicación perfectamente eficiente considera el uso óptimo de sus recursos para lograr el más corto tiempo de ejecución posible. Implementar algoritmos con la mínima redundancia y máximo uso de recursos es una parte clave de la velocidad de ejecución de la aplicación. Otra forma de eficiencia, es la medida del porcentaje de esfuerzo total del sistema (overhead) para producir los resultados finales, y tiene que ver con el trabajo extra que no es directamente útil para proporcionar los resultados esperados de la aplicación.

- Una implementación de hardware a la medida creado para una aplicación, puede ser específica sólo para esta, aprovechando el paralelismo inherente disponible en ella, así como llevando al cabo únicamente las funciones necesarias requeridas para completar la aplicación. El software que corre en una computadora programable puede ser por sí mismo una solución muy especializada a un problema, pero el hardware en el que es ejecutado está optimizado para traer (fectch), decodificar (decode), y ejecutar (execute) instrucciones, no para ejecutar la tarea específica de una aplicación dada.

Normalmente, a menos que el volumen del producto sea alto o el desempeño de la solución sea crítica, una solución en software se prefiere frecuentemente sobre una alternativa en hardware.

Capítulo 3

El algoritmo DES

En Agosto de 1974, la NBS (National Bureau of Standards) en la actualidad denominado NIST (National Institute of Standards and Technology) de E.U.A., invitó al sometimiento detallado de sistemas criptográficos disponibles. Se estipuló que el sistema debería de ser económico, viable y que sus detalles serían totalmente públicos. Lucifer fue el mejor algoritmo propuesto y provinó de IBM quien a finales de 1960 inició un proyecto de investigación dirigido por Horst Feistel, concluyendo en 1971 con el desarrollo de tal algoritmo [Sta99]. Las modificaciones, para conseguir el algoritmo DES consistieron básicamente en la reducción de la longitud de la clave y de los bloques.

En 1994, el NIST reafirmo a DES para su uso federal por otros cinco años más; recomendándolo para otras aplicaciones diferentes a la protección de información clasificada. William Stallings, piensa que, excepto en áreas de extrema sensibilidad, el uso de DES en aplicaciones comerciales no sería una causa inmediata de preocupación [Sta99].

En el verano de 1998 DES tuvo un ataque que logró tener éxito, este ataque se realizó usando fuerza bruta, análisis lineal y análisis diferencial, por lo que dicho algoritmo ya no es seguro. Sin embargo su diseño no tiene ninguna debilidad seria desde el punto de vista teórico, por lo que su estudio sigue siendo plenamente interesante; lo que sí quedó patente es que su longitud de clave es demasiado pequeña [Luc02].

En este capítulo se presentan algunas propiedades sobre las cuales se basa el diseño, así como, la especificación detallada para construir el algoritmo de acuerdo al estándar publicado [FIP77].

3.1 Cifrados de bloque.

Un cifrado de bloque es un sistema criptográfico que divide el texto plano dentro de bloques separados, usualmente todos del mismo tamaño, y puede operar independientemente en cada uno de ellos, como el caso del modo ECB, para producir una secuencia de bloques de texto cifrado [DH79]. Además cada bloque de símbolos se cifra siempre de igual manera, independientemente del lugar que ocupe en el mensaje, por lo que, para descifrar parte de un

mensaje no es preciso descifrarlo completamente desde el principio, basta con hacerlo desde el bloque que interese. Dos mensaje originales iguales, cifrados con la misma clave, producen siempre mensajes cifrados iguales.

Todos los cifrados en bloque se componen de cuatro elementos: una transformación inicial, una función criptográficamente débil iterada n veces o rondas, una transformación final y un algoritmo de expansión de clave [FGHM98]; éstos elementos se describen a continuación:

- La transformación inicial puede tener dos funciones: la primera consiste en aleatorizar simplemente los datos de entrada para ocultar bloques de datos de todos ceros o unos, etc., careciendo de significado criptográfico al no depender de la clave como sucede en DES. La segunda función, tiene significado criptográfico, como en RC5 e IDEA, la cual toma como argumento la clave utilizada lo que dificulta un ataque por análisis lineal o diferencial.

- Los paso intermedios consisten en una función no lineal que puede estar formada por una sola operación muy compleja o por la sucesión de varias transformaciones simples y que además puede ser unidireccional como en DES, o no como en IDEA y RC5. La aplicación repetida dos veces de la operación modular (pero eligiendo las claves de cifrado en orden inverso) conduce a los datos originales. Las transformaciones intermedias no han de formar estructuras de grupo, se dice que un cifrado no tiene estructura de grupo si cumple la propiedad 3.1.

$$\forall c_1,\, c_2 \nexists\, c_3 \text{ tal que } E_{c_1}(E_{c_2}(M)) = E_{c_3}(M) \tag{3.1}$$

  esto es, se cifra un mensaje primero con la clave c_1 y el resultado con la clave c_2 sucesivamente, no debe existir una clave c_3 que realize la transformación equivalente ya que no es recomendable para la robustez de un cifrador. Por el contrario, si el algoritmo criptográfico no presentará estructura de grupo, se hubiera incrementado la seguridad del sistema al emplear una clave de longitud doble.

- La transformación final sirve para que las operaciones de encriptado y decifrado sean simétricas. Cuando las vueltas de encriptado son de una sola operación, separadas por sumas módulo 2, bit a bit por ejemplo en DES y RC5, esta transformación se limita a realizar la operación inversa de la transformación inicial. Sin embargo, en los sistemas donde las vueltas de encriptado acaban con una operación que afecta a todos los bits del bloque, la transformación de salida debe realizar tanto la función inversa de esta operación como la inversa de la transformación inicial.

- El algoritmo de expansión de clave tiene por objeto convertir la clave de usuario, normalmente de longitud limitada entre 32 y 256 bits, en un conjunto de subclaves que pueden estar constituidas por varios cientos de bits en total. Conviene que sea unidireccional y que el conocimiento de una o varias subclaves intermedias no permita deducir las subclaves anteriores o siguientes. Además, se debe cuidar que las subclaves producidas no constituyan un pequeño subconjunto monótono de todas las posibles.

El cifrado de bloque más general posible es uno que pueda transformar cualquier bloque de texto en claro en algún bloque de texto cifrado tan grande como sea necesario para que pueda ocurrir la transformación reversible. Si los bloques son de longitud de n bits, hay 2^n distintos bloques y $2^n!$ modos en que éstos se pueden transformar. El número de bits de clave necesarios para seleccionar uno de estos modos es entonces $log_2(2^n!)$ o aproximadamente $n \times 2^n$. Este número es probablemente grande aún para valores claramente pequeños de n. Para n igual a 8, este requiere cerca de 2 000 bits de llave y para n igual a 16 este requiere aproximadamente un millón de bits. Un cifrado de este tipo es un cifrado de simple sustitución [DH79].

3.2 Cifrado de producto

La gran mayoría de los algoritmos de cifrado simétricos se apoyan en los conceptos de confusión y difusión propuestos por Claude Shannon en 1948, que se combinan para dar lugar a los denominados cifrados de producto. La confusión y difusión son dos técnicas básicas para ocultar la redundancia en un texto plano, y a pesar de su antigüedad, poseen una importancia clave en criptografía moderna.

La confusión consiste en tratar de ocultar la relación que existe entre el texto plano, el texto cifrado y la clave. Un buen mecanismo de confusión hará demasiado complicado extraer relaciones estadísticas entre las tres cosas. Por su parte la difusión trata de repartir la influencia de cada bit en el mensaje original lo más posible entre el mensaje cifrado.

La confusión por sí sola sería suficiente, ya que si se establece una tabla de sustitución completamente diferente para cada clave con todos los textos planos posibles se tendrá un sistema extremadamente seguro. Sin embargo, dichas tablas se ocuparían cantidades enormes de memoria, por lo que en la práctica serían inviables. Por ejemplo, un algoritmo que codificará bloques de 128 bits empleando una clave de 80 bits necesitaría una tabla de aproximadamente 10^{63} entradas.

Lo que en realidad se hace para conseguir algoritmos fuertes sin necesidad de almacenar tablas enormes es intercalar en capas alternadas la confusión (sustituciones simples, con tablas pequeñas) y la difusión (permutaciones). Esta combinación se conoce como cifrado de producto. La mayoría de los algoritmos criptográficos se basan en diferentes capas de sustituciones y permutaciones, estructura que denominaremos Red de Sustitución-Permutación (SP). En muchos casos el criptosistema no es más que un paso simple de sustitución-permutación repetido n veces, como ocurre con DES.

3.2.1 Redes de Feistel

Muchos de los cifrados de producto tienen en común que dividen un bloque de longitud n en dos mitades, L y R. Se define entonces un cifrado de producto iterativo en el que la salida de un proceso se usa como entrada para el siguiente, pero además, se trabaja alternadamente con una de las mitades en cada vuelta de encriptación. La mitad que es utilizada para realizar la actualización de la otra será intercambiada en la siguiente vuelta. Al proceso anterior se

le denomina ronda. El comportamiento de una ronda se establece en la ecuación 3.2. Sean L_{i-1} y R_{i-1}, las dos mitades del bloque en la iteración $i-1$ y f una función que toma como argumentos una mitad del bloque a encriptar y una llave,

$$\begin{aligned} L_i &= R_{i-1} \text{ si } i < n, \\ R_i &= L_{i-1} \oplus f(R_{i-1}, K_i) \text{ si } i < n, \\ L_n &= L_{n-1} \oplus fR_{n-1}, K_n, \\ R_n &= R_{n-1} \end{aligned} \tag{3.2}$$

Estos sistemas tienen la propiedad de ser reversibles, independientemente de cómo sea la función f, para ello basta con aplicar de nuevo el algoritmo al resultado, pero empleando las llaves K_i en orden inverso. Esto permite emplear el mismo mecanismo tanto para cifrar como para descifrar. Así también, es posible que dicha transformación sea una función de un solo sentido, empleando operaciones no lineales.

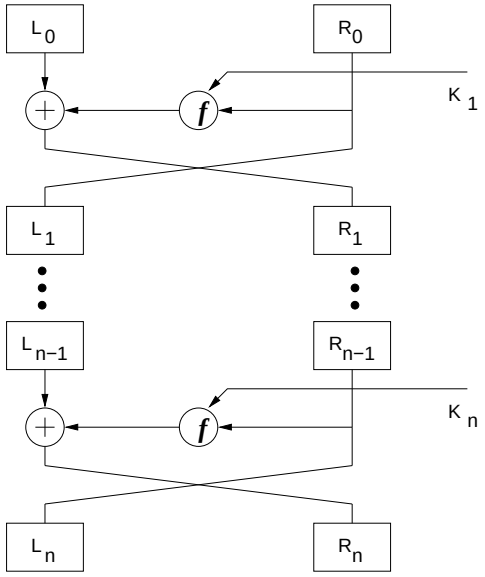

Figura 3.1: Estructura de una red de Feistel

3.3 El algoritmo DES.

El algoritmo ha sido diseñado para encriptar y descifrar bloque de datos de 64 bits con la combinación de una llave de 56 bits. El algoritmo transforma una entrada de 64 bits después de una serie de pasos, en una salida de 64 bits. El descifrado se realiza con la misma clave que se utilizá para encriptar, pero con el direccionamiento de los bits alterados de manera que se obtenga el proceso de descifrar como la manera inversa de el encriptado. Las operaciones básicas que se utilizan en el proceso son: xor (or-exclusivos), permutaciones predeterminadas (PI, PI^{-1}, P, E) y tablas de sustituciones (Cajas S).

DES tiene 19 etapas diferentes. El bloque a encriptar es sujeto a la permutación fija inicial IP (sin valor criptográfico) Se divide el bloque en dos mitades, la derecha y la izquierda. Después se realiza un cálculo complejo que depende de la llave. Se realiza una operación modular que se repite 16 veces; esta operación consiste en sumar módulo 2, la parte izquierda con una transformación (f) de la parte derecha, gobernada por una clave K_i. Es aquí en la estructura de transformación donde se lleva acabo la sustitución por valores contenidos dentro de las cajas-S, de las cuales depende fuertemente la fortaleza del criptosistema. Después se intercambian la parte derecha e izquierda para trabajar alternativamente sobre las dos mitades del bloque a cifrar. En la vuelta 16 se omite el intercambio, pero se finaliza aplicando una permutación que es inversa a la primera, o sea IP^{-1}. El cálculo que depende de la llave, puede ser definido en términos de la función f o función de cifrado, y una función KS de calendarización de llaves.

Para comprender el funcionamiento es necesario definir la siguiente notación: dados dos bloques L y R, LR es el bloque de los bits de L seguido de los bits de R. Puesto que la concatenación es asociativa $B_1B_2...B_n$ denota el bloque de los bits B_1 seguido de los bits de B_2...seguido de los bits B_n. La Figura 3.2 muestra la estructura general del algoritmo DES.

3.3.1 Permutación inicial y final

La primera etapa es una transposición, una permutación inicial (IP) del texto plano de 64 bits, independiente de la clave, que permite simplemente aleatorizar los datos de entrada (para ocultar bloques de datos de todos cero o unos, etc.) La Tabla 3.1 contiene la permutación inicial PI del algoritmo. Esta se lleva acabo justo al principio, antes de la primera ronda.

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Tabla 3.1: Permutacion IP para los 64 bits del bloque de entrada

Se observa que la permutación cambia a la primera posición al bit 58, a la segunda posición al bit 50 y así sucesivamente, dejando en última posición al bit 7.

La última etapa es otra transposición (IP^{-1}), que sirve para que las operaciones de encriptado y decifrado sean simétricas La Tabla 3.2 muestra la permutación final IP^{-1}, que se aplica justo al final de todas las demás operaciones del algoritmo. En las Tablas 3.1 y 3.2, se puede notar que cada una de estas permutaciones es la inversa de la otra.

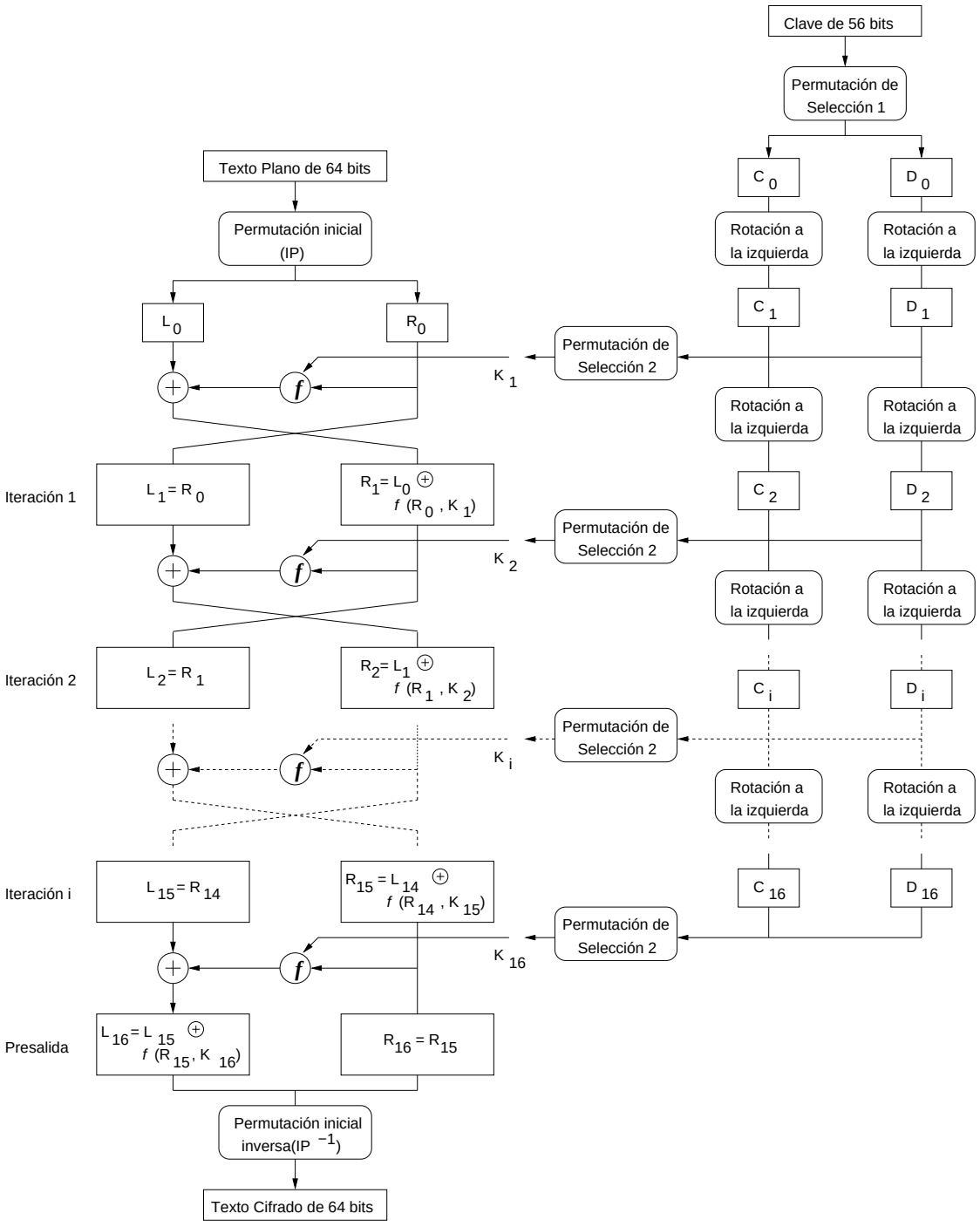

Figura 3.2: Estructura general del algoritmo DES

3.3.2 Estructura de transformación $f(R, K)$

El bloque permutado inicialmente será entonces la entrada al cálculo que se describe a continuación. La función no lineal esta formada por la sucesión de varias transformaciones simples. Los pasos intermedios se enlazan por suma módulo 2, bit a bit con los datos que vienen de la transformación inicial o de los pasos precedentes.

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Tabla 3.2: Permutacion IP inversa para los 64 bits del bloque de presalida

Este cálculo consiste de 16 iteraciones en términos de la función de cifrado f, la cual opera en dos bloques, uno de 32 bits y otro de 48 bits. De acuerdo a nuestra nomenclatura, L_0R_0 son los bloques de entrada a la primera iteración (dos bloques de 32 bits), que provienen del bloque de salida de IP, y a su vez K_0 como el bloque de 48 bits escogidos de entre los 64 bits de la llave. Entonces la salida para esta iteración con la entrada L_0R_0 quedará definida por la ecuación 3.3.

$$\begin{aligned} L_n &= R_{n-1} \\ R_n &= L_{n-1} \oplus (f(R_{n-1}, K_n)) \end{aligned} \tag{3.3}$$

donde $\oplus$ representa a la suma bit por bit modulo 2.

Por lo que $L_{16}R_{16}$ serán los bloques de presalida que entrarán a la permutación IP^{-1}, para ser permutada nuevamente.

Expansión (E)

La primera manipulación dentro de las estructura de transformación consiste en fabricar un vector de 48 bits a partir de los 32 iniciales mediante una expansión lineal, llamemos E a la función que realiza esta operación. En la Tabla 3.3 se presenta la expansión E. Los bits originales aparecen dentro de los recuadros resaltados; los bits añadidos aparecen en negrita (por conveniencia de la presentación, aparecen en cuatro filas, que han de considerarse correlativas).

Izquierda	**32**	1	2	3	4	**5**	**4**	5	6	7	8	**9**
Centro Izq.	**8**	9	10	11	12	**13**	**12**	13	14	15	16	**17**
Centro Der.	**16**	17	18	19	20	**21**	**20**	21	22	23	24	**25**
Derecha	**24**	25	26	27	28	**29**	**28**	29	30	31	32	**1**

Tabla 3.3: Selección de bits E

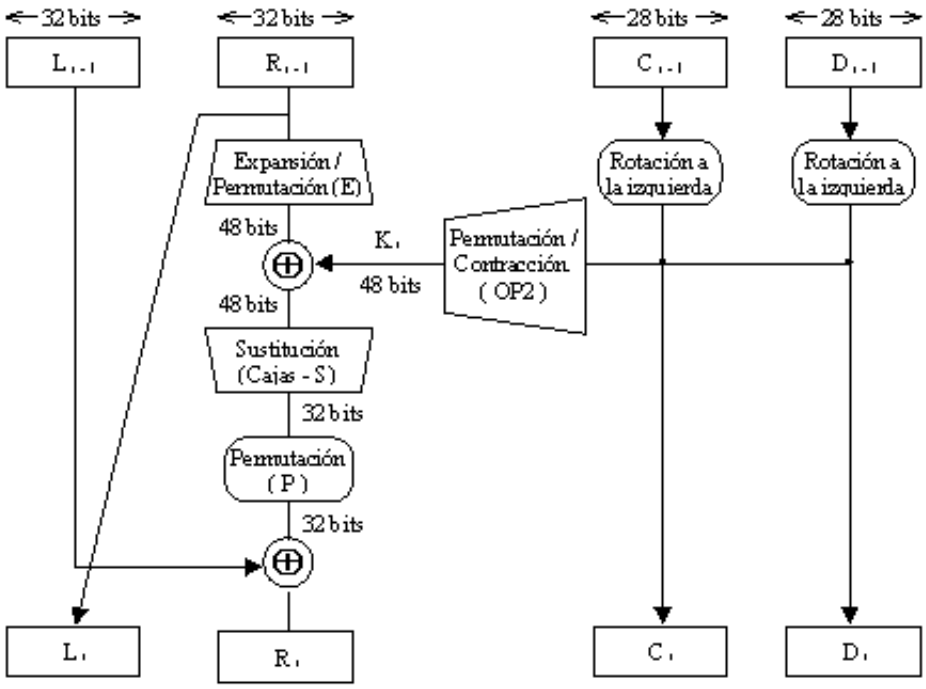

Figura 3.3: Estructura de transformación f

En cada iteración, un nuevo grupo de 48 bits son seleccionados de los 64 de la llave. Usando más notaciones podemos describir con mayor detalle las iteraciones del cálculo. En la ecuación 3.4, sea KS la función que toma como argumento un entero n en el rango de 1 a 16 y un bloque llave de 64 bits como entrada, y origina una salida K de 48 bits, la cual es la selección permutada de los bits de la llave.

$$K_n = KS(n, LLAVE) \tag{3.4}$$

La clave local de 48 bits y el vector proveniente de la expansión E se combinan mediante una suma módulo 2, bit a bit, obteniéndose K $\oplus$ E(R), que es otro vector de 48 bits. como se muestra en la Figura 3.4.

Cajas-S

Puesto que es muy difícil crear grandes cajas de sustituciones que sean altamente resistentes, DES cuenta con pequeñas cajas-S con las mejores propiedades posibles en su construcción [DT91].

El vector de 48 bits, obtenido anteriormente, se divide en ocho bloques, tal como se indica en la Figura 3.4. Se puede definir a $B_1, B_2, ..., B_8$ como los ocho bloques de 6 bits cada uno, de manera que se cumpla la ecuación 3.5.

$$B_1 B_2 ... B_8 = K \oplus E(R) \tag{3.5}$$

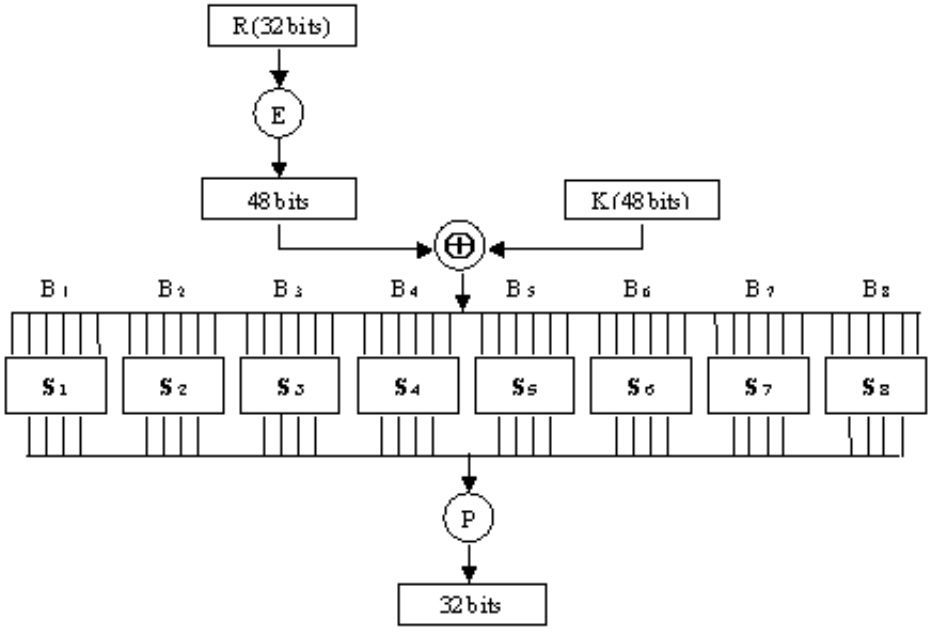

Figura 3.4: Cálculo de $f(R, K)$

Cada uno de estos grupos de 6 bits entra en una de las 8 funciones denominadas como Cajas-S, las cuales son las responsables de la no linealidad de DES, debido a que son elegidas de tal manera que la sustitución producida no sea afín ni función lineal de la entrada. Para poder construir buenos algoritmos de producto se intercalan tablas pequeñas de sustituciones sencillas (confusión) y permutaciones (difusión). Estas tablas pequeñas de sustitución se denominan de forma genérica Cajas-S y los principios para la elección de las Cajas-S, se supone deben ser secretos.

Cada una de las funciones de selección única S_1, S_2, ..., S_8 toma un bloque de 6 bits y entrega un bloque de 4 bits usando los valores que se muestran en la Tabla 3.4, de manera que, cada bloque B_1 es tomado como entrada a S_1, cada bloque B_2 es tomado como entrada a S_2, y así sucesivamente. La Figura 3.5 muestra el detalle de una caja que compone al sistema.

En el caso de que S sea la función definida y B sea un bloque de 6 bits, entonces S(B) se obtendrá así: El primer y último bit de B representarán en base 2 a un número en el rango de 0 a 3, dependiendo del valor de estos bits pueden haber cuatro sustituciones posibles, denotemos a ese número como i. Los 4 bits de en medio, sobre los que se hace la sustitución, representarán en base 2 a un número en rango de 0 a 15, denotemos a ese número como j. Al buscar ahora en la tabla la fila i, con la columna j, se encuentra un número en el rango de 0 a 15 el cual es representado únicamente por 4 bits; este número es el bloque de salida S(B). Por ejemplo, sea B(3)=42, en binario B(3)=101010. Busquemos el nuevo valor de B(3) en S(3)(fila i, columna j), según lo expuesto anteriormente i=10, j=0101, y en decimal i=2 y j=5. Por tanto, B(3) será S(3)(2,5)=15. Es importante decir que las tablas de sustituciones se han construido de manera que cuando se cambia un solo bit de la entrada, cambian por

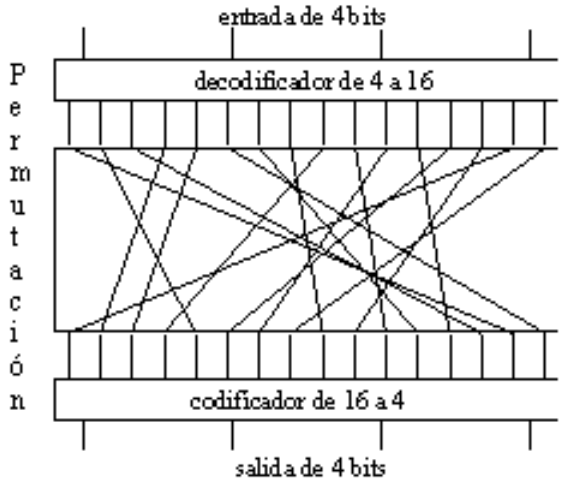

Figura 3.5: Detalle de la caja S1 (renglón 0)

lo menos dos bits de salida.

Los 8 bloques $S_k(B_k)$, $(1 \leqslant k \leqslant 8)$ de 4 bits cada uno son unidos para formar un bloque único de 32 bits los cuales forman la entrada a P, como se muestra en la Figura 3.4.

Permutacion (P)

La función de permutación P es una permutación lineal fija, elegida de tal forma que la difusión de bits sea máxima a lo largo del bloque de 32 bits. Tiene como entrada 32 bits y da por salida un bloque de 32 bits. Los bits son permutados de acuerdo a la Tabla 3.5.

En resumen, el bloque $f(R, K)$ queda definido por la ecuación 3.6.

$$P(S_1(B_1)S_2(B_2)...S_3(B_3)) \tag{3.6}$$

La salida de esta permutación es, entonces, la salida de la función f para las entradas R y K.

3.3.3 Generación de llaves

Por último falta definir la manera en que la llave se convierte en los bloques $K_1, K_2, ..., K_{16}$. En cada una de la 16 iteraciones se emplea un valor K_i, obtenido a partir de la clave de 56 bits y distinto en cada iteración.

Hay que recordar que K_i para $(1 \leqslant i \leqslant 16)$ es el bloque de 48 bits (ver Figura 3.4) de f. Entonces para describir KS (ver Ecuación 3.4) es suficiente describir cómo obtenemos K_i a partir de la clave. Basta con describir las dos opciones de permutación escogidas, así como el

	Columna																
Fila	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Caja-S
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7	
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8	S_1
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0	
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13	
0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10	
1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5	S_2
2	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15	
3	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9	
0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8	
1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1	S_3
2	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7	
3	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12	
0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15	
1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9	S_4
2	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4	
3	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14	
0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9	
1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6	S_5
2	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14	
3	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3	
0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11	
1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8	S_6
2	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6	
3	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13	
0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1	
1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6	S_7
2	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2	
3	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12	
0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7	
1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2	S_8
2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8	
3	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11	

Tabla 3.4: Especificación de las Cajas-S

16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	25

Tabla 3.5: Permutacion de una caja P.

orden de los corrimientos a la izquierda y entonces habremos definido KS.

Un bit de cada 8 es escogido para asegurarnos de que cada byte tenga una paridad impar, y estos bits son los bits 8, 16, 24, ..., 64. De este modo se realiza una reducción a 56 bits, eliminando un bit de cada ocho. La permutación de selcción 1 (PC-1) reordena los bits, operación que carece de significado criptográfico y que se determina en la Tabla 3.6.

57	49	41	33	25	17	9	1	58	50	42	34	26	18
10	2	59	51	43	35	27	19	11	3	60	52	44	36
63	55	47	39	31	23	15	7	62	54	46	38	30	22
14	6	61	53	45	37	29	21	13	5	28	20	12	4

Tabla 3.6: Permutacion de selección 1 (PC-1).

Esta tabla ha sido dividida en dos partes, cada una de 28 bits. La primera determina como se escogerán los bits para formar el segmento de la izquierda del bloque de claves llamado C_0 y la segunda, determina como se escogerán los bits para formar el segmento de la derecha del bloque de claves llamado D_0. Los bits de la Clave se numeran de 1 a 64; los bits de C_0 serán los bits 57, 49, 41, ..., y 36. Los bits de D_0 serán los bits 63, 55, ..., 12 y 4.

Una vez definidos C_0 y D_0, los bloques C_i y D_i, $(1 \leqslant i \leqslant 16)$, se obtienen a partir de los bloques $C_{i-1}D_{i-1}$ más un corrimiento individual a la izquierda de cada uno de ellos (rotación o permutación circular), como se ve en la Tabla 3.7.

Iteración número	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Número de corrimientos	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Tabla 3.7: Corrimiento de la clave de acuerdo al número de iteración.

Después de las rotaciones se vuelven a unir las mitades, teniendo de nuevo 16 grupos de 56 bits. A continuación se procede a seleccionar 48 bits de cada grupo para formar finalmente las 16 subclaves, en lo que se denomina permutación con comprensión. Los bits elegidos son iguales para todas las subclaves, y se rigen por la permutación de selección 2 (PC-2) que se determina en la Tabla 3.8.

Por lo tanto de acuerdo a la Tabla 3.8, el primer bit de K_i es el 14avo bit de C_iD_i, el segundo será el 17avo y así sucesivamente hasta llegar al último bit el 48 que será el 32, (ver Figura 3.3).

14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

Tabla 3.8: Permutación de selección 2 (PC-2).

Capítulo 4

Análisis de la versión en software

El primer paso para implantar el algoritmo criptográfico DES en un sistema hardware-software, consiste en hacer un ánalisis del algoritmo en software que nos pueda proporcionar información sobre el costo en tiempo, la cantidad de operaciones y el trabajo realizado por cada una de las partes que componen dicho algoritmo. Este análisis tiene como objetivo detectar las secciones críticas del programa, es decir las que consumen la mayor cantidad de tiempo de ejecución. Los factores que se han considerado para identificar secciones críticas o secciones de código susceptibles de ser implantadas en hardware son: las operaciones más repetitivas del código, las operaciones que operan a nivel de bits y las operaciones no adecuadas para un procesador como son permutaciones, corrimientos y rotaciones.

En este capítulo se presenta el análisis que se le hace a la implementación en software del algoritmo DES. Primero se muestran los detalles de la implementación del algoritmo en software. Después se realiza una prueba para encontrar el tiempo que ocupan las operaciones de cifrado/descifrado de datos. En la tercera sección se realiza un análisis de perfiles del algoritmo para localizar sus secciones críticas, se utilizan diferentes arquitecturas para comparar los desempeños de cada una de ellas. En la siguiente sección se hace un análisis a nivel de operaciones primitivas para mostrar los grados de repetición de las operaciones lógicas que intervienen en la sección critica del algoritmo, puesto que éste es un parámetro importante que debe considerarse a la hora de decidirse por una implementación en hardware. Por último se presenta un análisis a nivel de secuencias de operaciones que componen la sección crítica del algoritmo y que nos conduce a detectar de manera más definida los fragmentos de la función que pueden ser aprovechadas al ser ejecutadas por una máquina de cómputo basada en FPGA.

4.1 Implementación del algoritmo.

El código fuente usado para este análisis proviene de *Applied Cryptography. Protocolos, algorithms and source code in c*, publicado por Bruce Schneier [Sch96]. El programa tiene la característica de no ser una emulación paso a paso del FIPS-46 [FIP77] (norma descrita para implementar DES en hardware). En vez de eso, el algoritmo realiza el procesamiento con un método más sencillo y eficiente para software. Esto es apropiado debido a que permite realizar

una comparación de ejecución más real con el hardware. Cabe mencionar que el orden de complejidad del algoritmo implementado es líneal: $O(n)$, donde n corresponde a la catidad de datos a procesar. Las diferencias entre la implementación en software del algoritmo para este estudio y la norma FIPS-46 son los siguientes:

- La implementación convierte los 64 bits de entrada (ocho datos tipo char) en un arreglo de dos elementos de datos de tipo long (32 bits), y realiza todas las operaciones con estos dos bloques. Mientras que en la norma de DES se trabaja con datos de 56, 48 y 32 bits.
- La implementación convierte los 64 bits de la clave (ocho datos tipo char) en dos arreglos de treinta y dos elementos de datos de tipo long (32 bits), y utiliza estos bloques de 32 bits en la estructura de transformación $f(R, K)$. Mientras que la norma de DES, utiliza una calendarización de 16 subclaves de 48 bits.
- Las cajas S están declaradas como un arreglo de 64 elementos de 32 bits que no tienen un parecido inmediato a las cajas S de 6×4 bits establecidas en la norma.
- No existe propiamente la caja E (Permutación/Expansión), ni la caja P (Permutación), tal como se describe en la norma de DES, pero en cambio estas operaciones estan implícitas en la función que se encarga de las operaciones de encriptado/desencriptado, consiguiendo el mismo resultado.

Ésta versión en software esta escrita en Lenguaje C y fué ejecutada usando una plataforma con sistema operativo Red Hat Linux v7.0. Los pseudocódigos 4.1 y 4.2 presentan la estructura más general del algoritmo completo.

En el Pseudocódigo 4.1, se muestra la calendarización de la clave. En esta parte se obtienen los dos arreglos de 32 elementos de 32 bits que se utilizarán en las operaciones de cifrado o descifrado. Inicialmente clavesdes() realiza un preprocesamiento de los datos originales de la clave y estos datos son pasados a procesar_clave(), quien termina de realizar el procesamiento. Enseguida los datos pasan a la función claves_a_usar() quien los guarda temporalmente en un arreglo. Finalmente la función copiar_claves se encarga de copiar desde el contenido temporal hasta el arreglo de claves destinado para ello.

Algoritmo 4.1 Pseudocódigo para la calendarización de la clave

```
clavesdes(clave a usar, seleccion de operación: codificar o decodificar)
   procesar_clave(clave semiprocesada)
      claves_a_usar(clave procesada)
copiar_claves(areglo de claves para codificar o decodificar)
```

En el Pseudocódigo 4.2, puede verse el procesamiento que sufren los datos en las operaciones de cifrado o descifrado. El algoritmo trabaja sobre los bloques de datos tantas veces como bloques existan en el mensaje de entrada. La función juntar() se encarga de reunir ocho elementos de 8 bits en un arreglo de dos elementos de 32 bits. Enseguida se aplica la función funcdes() que realiza el procesamiento de cifrado o descifrado sobre el arreglo obtenido

previamente. Cuando el bloque de datos ha terminado de ser procesado, la función separar() fragmenta el arreglo de dos elementos de 32 bits, utilizado en el procesamiento, en 8 unidades de 8 bits cada uno. Por último, se avanza al siguiente bloque y se repite el proceso anterior.

Algoritmo 4.2 Pseudocódigo para codificar y decodificar datos

```
Para bloques = 0 a n hacer
  juntar(datos de 8 bits, arreglo de datos de 32 bits)
  funcdes(arreglo de datos, claves de cifrado o descifrado)
  separar(arreglo de datos de 32 bits, datos de 8 bits)
  avanzar al siguiente bloque de datos
fin de Para
```

4.2 Tiempo de ejecución del algoritmo.

La primera prueba de rendimiento (benchmark) de la versión en software del algoritmo sirvió para medir el tiempo de ejecución y se realizó en un cluster de tecnología abierta, con microprocesadores Pentium III de Intel a 800 MHz, memoria RAM de 128 MBytes y con un disco duro de 29 GB. El ejecutable fue compilado usando el compilador de proyectos de GNU para C y C++ (gcc-2.96).

En esta prueba se utilizan sockets de comunicación y se comparan los tiempos de ejecución de dos versiones diferentes del programa. En la primera versión se envían mensajes codificados utilizando operaciones de DES más operaciones de comunicaciones inherentes al socket. En la segunda versión se envían mensajes íntegros, utilizando únicamente operaciones de comunicaciones inherentes al socket.

La primera versión del programa realiza el siguiente procedimiento: en la computadora local se codifican los mensajes de longitud L que posteriormente son enviados a través de un socket, donde (512 bytes $\leqq$ L $\leqq 1 \times 10^6$ bytes). En la computadora remota se reciben y se decodifican los datos primero, para después volverlos a codificar y volverlos a envíar a través del socket. Se reciben nuevamente en la computadora local en donde son finalmente decodificados, terminando así el procedimiento. Este esquema de operación se puede apreciar en la Figura 4.1. En esta Figura, el proceso 1 se ejecuta en la computadora local y el proceso 2 se ejecuta en la computadora remota, estos procesos son también llamados proceso cliente y proceso servidor respectivamente.

El tiempo de ejecución obtenido en la primera versión del programa, es el resultado de efectuar las siguientes operaciones: generación de llaves, cifrado, comunicación de datos (envío y recepción) y descifrado. Estas operaciones se hacen dos veces de manera simétrica, como puede verse con la línea intermitente que aparece dentro de la Figura 4.1. Considerando que el proceso se puede repetir n veces, el tiempo de ejecución se determina mediante la ecuación 4.1:

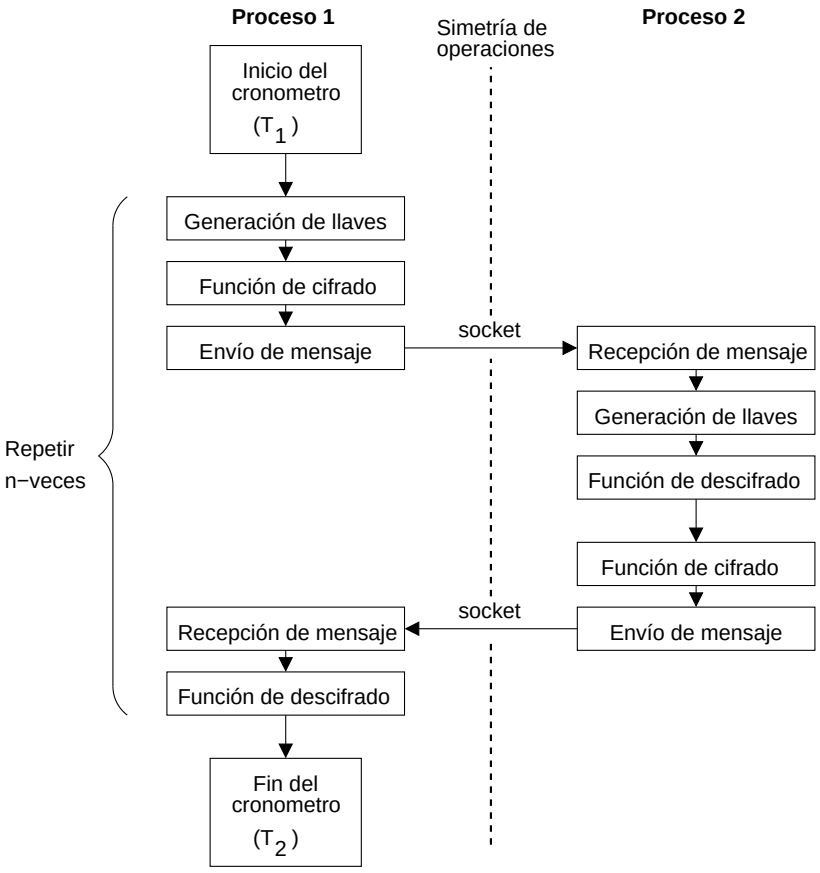

Figura 4.1: Esquema de operación del primer experimento

$$\text{Tiempo de ejecución} = \frac{T_2 - T_1}{2n} \tag{4.1}$$

donde: T_1 es el tiempo inicial en el proceso 1, T_2 es el tiempo final en el proceso 1 y n es el número de repeticiones.

En la segunda versión del programa el procedimiento fue el siguiente: En la computadora local se envían mensajes de longitud L a través de un socket, donde (512 bytes $\leqq$ L $\leqq 1 \times 10^6$ bytes). El mensaje se recibe en la computadora remota y se reenvía nuevamente a través del socket. Finalmente la computadora local, recibe el mensaje y termina el procedimiento. El esquema de esta operación es casi el mismo que el de la Figura 4.1 solo que omitiendo las operaciones de generación de llaves, función de cifrado y función de descifrado.

El tiempo de ejecución obtenido en la segunda versión del programa, es el resultado de efectuar únicamente operaciones de comunicación de datos (envío y recepción). Aquí también las operaciones se realizan dos veces de manera simétrica y además se pueden repetir n veces, por lo que el tiempo de ejecución también se determina usando la ecuación 4.1.

En la Figura 4.2, se muestra la gráfica en la que se pueden comparar los tiempos de

ejecución obtenidos al realizar los dos experimentos anteriores. En esta gráfica, en donde se utiliza una escala logarítmica, podemos observar que el tiempo en ejecutar las operaciones de encriptado, desencripatado para DES más el tiempo de comunicación tiene un valor considerable, con respecto al tiempo que consumen únicamente las comunicaciones de los mensajes íntegros.

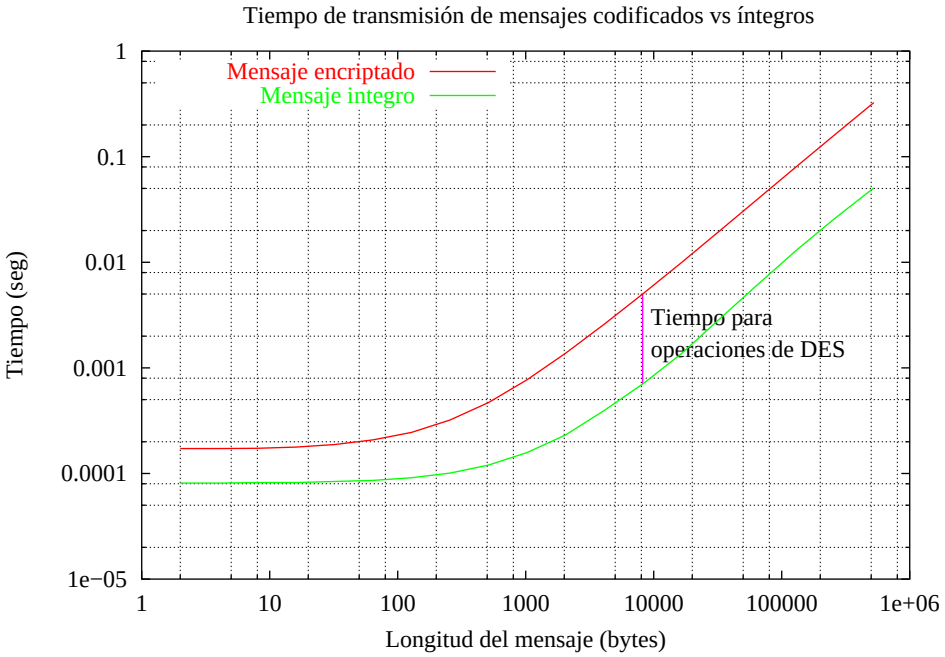

Figura 4.2: Gráfica comparativa del consumo de tiempo entre mensajes íntegros y mensajes codificados. La linea vertical entre las dos gráficas señala el tiempo extra invertido por las operaciones de DES.

4.3 Perfil del tiempo de ejecución del algoritmo.

Un análisis de perfiles del algoritmo en software permite ver cómo se distribuye el tiempo entre cada una de las funciones que integran el algoritmo y esta basado en los tiempos de ejecución medidos dentro de cada una de la funciones. Los tiempos obtenidos nos dan información detallada sobre los costos de tiempo de ejecución de cada una de ellas. En esta parte del análisis se realizaron experimentos para obtener los perfiles de tiempo, con la finalidad de detectar las funciones de mayor tiempo de ejecución del algoritmo.

Con el propósito de mostrar que los aspectos arquitecturales no jugaran un papel importante, en los resultados relativos de las operaciones a realizar, se ejecutó el programa de prueba sobre tres computadoras con procesadores diferentes. La primera se realizó en una PC de escritorio con microprocesador Pentium MMX a 120 MHz, con 32 MB en RAM y un disco duro de 2 GB. Sobre una plataforma Linux con sistema operativo Red Hat 7.1. Se usó la herramienta gcc(v2.96) para compilar el programa. El segundo de los experimentos de esta prueba se realizó en un cluster de tecnología abierta, con microprocesadores Pentium II a 266

MHz, con 64 MB en RAM y disco duro de 4.32 GB. El sistema operativo usado fue Red Hat Linux 6.2 y se uso el compilador gcc(v2.96). El último experimento de esta prueba se realizó en un cluster de tecnología abierta, con microprocesadores Pentium III de Intel a 800 MHz, memoria RAM de 128 MBytes y con un disco duro de 29 GB. El ejecutable fue compilado usando el compilador de proyectos de GNU para C y C++ (gcc-2.96).

La Figura 4.3 muestra la gráfica de perfiles obtenida de los tres experimentos anteriores. En esta gráfica se presentan los porcentajes que consumen cada una de las funciones mas representativas que componen el algoritmo, con respecto al tiempo total de ejecución, utilizando diferentes tamaños de bloques a encriptar que van desde 1KB hasta 64KB.

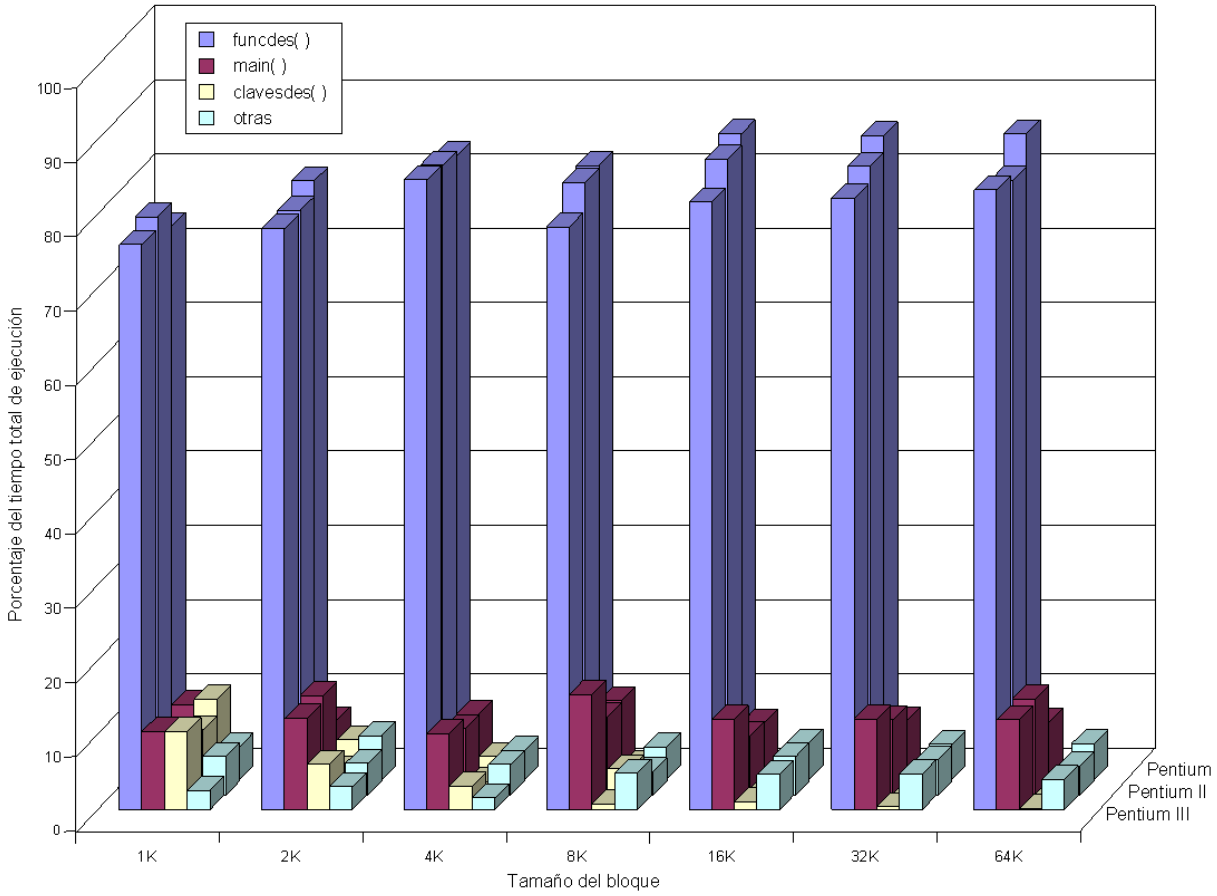

Figura 4.3: Gráfica de perfiles para diferentes procesadores y diferentes tamaños de bloque.

En la Tabla 4.1 se presenta un resumen de los resultados obtenidos en la prueba de perfiles del algoritmo. Estos resultados corresponden a los promedios de los porcentajes de los tiempos totales de ejecución, obtenidos en cada uno de los diferentes procesadores, con los siete diferentes tamaños de bloques utilizados. Cabe decir que estos resultados fueron obtenidos a partir de un promedio de numerosas ejecuciones del algoritmo con la finalidad de hacer más representativo el fenómeno.

Función	Porcentaje de tiempo de ejecución		
	Pentium MMX	Pentium II	Pentium II
funcdes()	83.4%	82.7%	81.0%
main()	8.7%	10.3%	12.2%
clavesdes()	3.2%	2.7%	3.2%
Otras	4.7%	4.3%	3.7%

Tabla 4.1: Promedios obtenidos en los perfiles de tiempo del algoritmo

La parte más costosa, de acuerdo a la Figura 4.3 y la Tabla 4.1, corresponde a la función `funcdes()` que esta consumiendo en promedio cerca del 80% del tiempo total de ejecución del algoritmo, en cualquiera de los tres procesadores utilizados, incluso a pesar del incremento en las velocidades de operación y en el aumento de los recursos utilizados. Puede decirse, entonces, que en el diseño actual de las arquitecturas de los microprocesadores se han incrementado capacidades como acceso a memoria cache, velocidad de operación, ancho del bus, etc., dejando a un lado la funcionalidad de operaciones básicas tales como corrimientos y operaciones lógicas a nivel de bits, por lo que aún cuando la tecnología en los procesadores se incrementa, el costo para realizar estas operaciones se sigue manteniendo alto.

Se puede determinar entonces que la función `funcdes()` es la sección de código candidata para implantarse en hardware. Esta función corresponde al núcleo del algoritmo ya que, es aquí, donde se lleva a cabo todo el procesamiento de los datos para encriptado y/o desencriptado, realizando operaciones lógicas AND, OR, OR exclusivo, y corrimientos, todo a nivel de bits. El estudio detallado de esta función se encuntrará en las secciones siguientes.

4.4 Análisis a nivel de operaciones primitivas.

Una vez encontrada la parte más costosa del algoritmo implementado en software o el candidato temporal (`funcdes()`), se procedió a realizar un estudio de lo que ocurre dentro de esta función para entender el porqué del alto costo de operación.

La función `funcdes()` del algoritmo, se compone de tres partes fundamentales y claramente divisibles. Una de ellas hace una emulación de la Permutación Inicial descrita en la norma FIPS/46 para implementar DES en hardware. La segunda parte importante emula las cajas S y la permutación P. Por último la tercera parte emula lo que en la norma se describe como Permutación Final. Todas ellas compuestas de operaciones lógicas primitivas a nivel de bits que se encargan de realizar el procesamiento requerido para encriptar los datos.

El Pseudocódigo 4.3, presenta las operaciones que se realizan dentro de la función `funcdes()`. En este pseudocódigo se han omitido los operandos y se presentan únicamente las operaciones que se realizan,de forma abreviada, respetando la correspondencia de una línea del pseudocódigo a una línea del código fuente. Las abreviaciones usadas para representar las operaciones, corresponden a las contracciones de los nombres como se muestra enseguida:

CD Corrimiento a la Derecha, **CI** Corrimmiento a la Izquierda, **X** operación or-eXclusivo, **C** Corrimiento a la derecha y Corrimiento a la izquierda, **A** operación And, **O** operación Or y **AM** Acceso a Memoria.

Algoritmo 4.3 Pseudocódigo de `funcdes()`

Repetir
  CD, X, A
  X
  CI, X
hasta 4 veces
C, A, O, A
X, A
X
X
C, A, O, A

(a) Permutación Inicial

Repetir
  C, O
  X
  Repetir
    CD, A, AM, O
  hasta 4 veces
  X
  Repetir
    CD, A, AM, O
  hasta 4 veces
  X
hasta 2× 8 veces

(b) Permutación de cajas S y P

C, O
X, A
X
X
C, O
CD, X, A
Repetir
  X
  CI, X
  CD, X, A
hasta 3 veces
X
CI, X

(c) Permutación Final

4.4.1 Conteo de operaciones.

La primera parte de este análisis consistió en contabilizar las operaciones primitivas lógicas a nivel de bits que componen a `funcdes()`. Mediante revisión de cada línea de código que compone a la función, se determinó cuántas operaciones lógicas se realizan sobre los datos, encontrándose así el grado de repetitividad que presenta cada operación lógica. En la Tabla 4.2 se pueden observar las cantidades de operaciones totales contenidas dentro de la función.

Operación	No. de repeticiones			Total	Porcentaje
	Permutación Inicial	Permutación de Cajas S y P (8 ciclos)	Permutación Final		
AND	9	128	5	142	19.35%
OR	2	128	2	132	17.98%
Corrimiento a la derecha	6	112	6	124	16.89%
OR Exclusivo	15	48	15	78	10.63%
Corrimiento a la izquierda	6	16	6	28	3.81%
Asignaciones	19	192	19	230	31.34%

Tabla 4.2: Número de operaciones lógicas realizadas en la función `funcdes()`

4.5 Análisis a nivel de secuencia de operaciones.

Una inspección visual al pseudocódigo 4.3 del algoritmo, muestra que por lo general se realizan más de una operación lógica primitiva en cada una de sus líneas. Al juntar las operaciones lógicas primitivas que aparecen en la misma línea, podemos formar lo que se ha denominado como secuencias de operaciones. Encontrando un total de siete patrones de secuencias de operaciones que pueden realizar todo el procesamiento requerido por la función original y que equivaldría a las aproximadamente 58 líneas de código de `funcdes()`.

A continuación, se describen con más detalle cada una de las secuencias de operaciones encontradas, recuerde que el nombre propuesto para cada secuencia proviene, por lo general, de las iniciales de las operaciones lógicas que se realizan. Se mencionan los números de repetición y los porcentajes, considerando que una ejecución completa de `funcdes()` equivale a la suma de todas y cada una de las repeticiones de las secuencias de operaciones. La descripción se inicia con un pseudocódigo muy parecido a la línea de código fuente en C, a la cual sustituyen para posteriormente detallar las tareas que se realizan.

1. **CDXA.** Operación: Corrimiento a la Derecha, or-eXclusivo, And.
   (8 repeticiones en el código equivalente al 3.5%)
   destino `=` `(` (*fuente*1 `>>` *corrimiento*) `^` *fuente*2 `)` `&` *mascara*;
   La secuencia realiza un CORRIMIENTO A LA DERECHA al operando *fuente*1 de

tantos bits como valor tenga *corrimiento*. Inmediatamente despúes realiza un OR-EXCLUSIVO entre el resultado conseguido anteriormente y *fuente*2. Este resultado junto con un valor de *mascara* realizan una operación AND. Finalmente el resultado es asignado a la variable *destino*. Los valores que puede tomar *corrimiento* son "8, 2, 16, 4" y los valores que toma *mascara* son "0x00ff00ffL, 0x33333333L, 0x0000ffffL, 0x0f0f0f0fL", ambos *corrimiento* y *mascara* tienen una correspondencia uno a uno entre sus valores siguiendo el orden en que aparecen.

2. **X.** Operación: or-eXclusivo.
(60 repeticiones en el código equivalente al 26.5%)
destino = *fuente*2 ^ *fuente*1;
La operación realiza un OR-EXCLUSIVA entre dos operandos *fuente*2 y *fuente*1. El resultado se asigna a *destino*.

3. **CIX.** Operación: Corrimiento a la Izquierda, or-eXclusivo.
(8 repeticiones en el código equivalente al 3.5%)
destino = *fuente*2 ^ (*fuente*1 << *corrimiento*);
La operación realiza un CORRIMIENTO A LA IZQUIERDA de tantos bits como valor sea *corrimiento* al operando *fuente*1 y enseguida se hace una operación OR-EXCLUSIVA junto con el operando *fuente*2. El resultado es asignado a *destino*. Los valores que puede tomar *corrimiento* son "2, 4, 8, 16".

4. **CAOA.** Operación: Corrimiento a la Derecha, Corrimiento a la Izquierda, And, Or, And.
(2 repeticiones en el código equivalente al 0.8%)
destino = ((*fuente*1 << 1) | ((*fuente*1 >> 31) & 1L)) & 0xffffffffL;
La operación debe realizar un CORRIMIENTO A LA DERECHA al operando *fuente*1 de 31 bits, después este valor se enmascara mediante una operación AND usando el valor "1L". El resultado de la operación anterior es procesado en una operación OR, junto con un CORRIMIENTO de 1 bit A LA IZQUIERDA del operando *fuente*1. Enseguida, el valor obtenido se vuelve a enmascarar usando "0xffffffffL" mediante una operación AND. Por último el valor es asignado a *destino*. Básicamente correponde a un registro de desplazamiento circular a la izquierda.

5. **XA.** Operación: or-eXclusivo, And.
(2 repeticiones en el código equivalente al 0.8%)
destino = (*fuente*1 ^ *fuente*2) & 0xaaaaaaaaL;
La operación realiza un OR EXCLUSIVO entre los operandos *fuente*1 y *fuente*2. Después se realiza un enmascaramiento mediante una operación AND y el valor "0xaaaaaaaaL". El resultado final se asigna a *destino*.

6. **CO.** Operación: Corrimiento a la Derecha, Corrimiento a la Izquierda, Or.
(18 repeticiones en el código equivalente al 8.0%)
destino = (*fuente*1 << *corrimI*) | (*fuente*1 >> *corrimD*);
La operación debe realizar tanto un CORRIMIENTO A LA IZQUIERDA al operando *fuente*1 del numero de bits como valor tenga *corrimI*, como un CORRIMIENTO A LA

DERECHA del numero de bits como sea *corrimD*, a este mismo operando $fuente1$. Los valores obtenidos son procesados posteriormente en una operación OR y el resultado se asigna a *destino*. Los valores posibles para *corrimI* son "28, 31" y sus correspondientes valores para *corrimD* son "4, 1", lo cual asegura que seconvierta en un registro de desplazamiento circular a la derecha tomando como base los valores de *corrimD* o un registro de desplazamiento circular a la izquierda tomando como base los valores de *corrimI*.

7. **CDAAMO.** Operación Corrimiento a la Derecha, And, Acceso a Memoria, Or.
(128 repeticiones en el código equivalente al 56.6%)
$destino$ `=` $fuente2$ `|` $fuente1$ `[(`a_operar `>>` $corrimiento$ `) & 0x3fL];`
La operación debe realizar un CORRIMIENTO A LA DERECHA de tantos bits como valor tenga *corrimiento*, al operando *a_operar*. El valor obtenido se procesa posteriormente en una operación AND junto con la mascara "0x3fL" cuya finalidad es la de habilitar los 6 primeros bits de un vector de 32 bits. Este resultado proporciona el ACCESO A MEMORIA de uno de los ocho arreglos de 64 elementos disponibles, que queda determinado por $fuente1$. Tomado el valor de la matriz se introduce en una operación OR junto con el valor de $fuente2$. Por último el valor se asigna a *destino*. Los accesos a memoria son basicamente los accesos a las cajas S del algoritmo. Los valores que puede tomar *corrimiento* son "0, 8, 16, 24", estos valores tienen una correspondencia de uno a dos con las cajas SP7 y SP8, SP5 y SP6, SP3 y SP4, SP1 y SP2. Esto es, si se realiza un corrimiento de 0 bits entonces se pude accesar a una de las dos cajas disponibles, la SP7 o la SP8 y así sucesivamente.

Los grados de repetición que presentan cada una de las secuencias de operaciones, descritas anteriormente, nos dan una primera aproximación del aporte que cada una de ellas realiza al tiempo de ejecución del algoritmo. En la segunda columna de la Tabla 4.3 se muestran los grados de repetición de cada una de las secuencias de operacion, obtenidos a partir de un conteo directo dentro de `funcdes()`. Observe que las secuencias de operaciones con un alto grado de repetición como CDAAMO, XOR, CO, son las que aportan el mayor consumo de tiempo de ejecución del algoritmo. Un dato importante es que estas tres secuencias se localizan en su mayoría dentro de un ciclo que itera 2×8 veces, lo que provocaría su alto número de reincidencia.

4.5.1 Perfil del tiempo de las secuencias de operaciones.

Se realizó un perfil de la distribución del tiempo de ejecución, como en la sección 4.3, para obtener los consumos más reales de tiempo de cada uno de los siete patrones de secuencias que componen al algoritmo. El comportamiento del algoritmo, se muestran en la columna tres de la Tabla 4.3. Observe que cada patrón presenta un porcentaje diferente de consumo de tiempo con respecto al tiempo total de ejecución y es precisamente este porcentaje el que marca el nivel de jerarquía a la hora de decidirse por cuál fragmento de la función implantar en hardware. Recordando que en una implementación en hardware-software resulta apropiado definir qué sección del algoritmo debe de ser implementada en software o en hardware, para conseguir un mejor desempeño.

En la Tabla 4.3 se reporta solamente el 57.43% del tiempo total de ejecución del algoritmo completo, mientras que el 42.56% de tiempo restante se ha distribuido entre $Op - funcdes$ y $Otras - funciones$. La operación $Op - funcdes$ consume el 35.58% del tiempo total de ejecución y corresponde a las operaciones inherentes a llamadas a funciones, como cambios de contexto, guardado de las variables, recuperación de las variables, etc. $Otras - funciones$ se refiere a funciones insignificantes para el estudio pero que aportan el 6.98% restante del tiempo total de ejecución, mismo que se reparte de la siguiente manera: 4.34% en la función `main()`, 1.51% en la función `clavesdes()` y 1.03% en otras.

Con los valores de la columna dos y tres de la Tabla 4.3, se puede calcular el costo unitario de operación, a partir de la ecuación 4.2, que equivaldría al costo de una sola repetición para una secuencia dada. La columna cuatro de la Tabla 4.3, presenta los resultados obtenidos.

$$\text{Costo de operación} = \frac{\text{Consumo de tiempo}}{\text{grado de repetición}} \tag{4.2}$$

Nombre del circuito	Número de repeticiones	Consumo de tiempo de ejecución (%)	Costo unitario de operación (%)
CDAAMO	128	42.11	.328
X	60	6.89	.114
CSO	18	4.30	.238
CIX	8	1.57	.196
CDXA	8	1.24	.155
CAOA	2	0.82	.410
XA	2	.50	.250
Total	226	57.43	–

Tabla 4.3: Costo de operación para una repeticion en cada una de las secuencias de encontradas en `funcdes()`

La tabla 4.4 presenta la distribución del tiempo de ejecución entre los tres componentes principales de `funcdes()`: Permutacion Inicial (P. I.), Permutacion de Cajas SP (P. SP) y Permutación Final (P. F.). Esta estimación se obtiene de multiplicar el costo unitario de operación de cada secuencia por el número de repeticiones que hay en cada una de las tres permutaciones que componen a la función.

Con los resultados obtenidos de la prueba de perfiles y presentados en la Tabla 4.3, se puede establecer el tamaño temporal del candidato o la fracción del código susceptible de implantarse en hardware. El valor de este tamaño temporal es de 57.4 y se obtuvó del total de consumo de tiempo de ejecución de las siete secuencias a implementar, ver Tabla 4.3.

Con el valor de 57.4, obtenido anteriormente, se puede calcular la máxima aceleración teórica a partir de la ley de Amdahl, mostrada en la ecuación 4.3 [DG97].

Nombre del circuito	Costo unitario de operación	P. I. (Repeticiones)	P. SP (Repeticiones)	P. F. (Repeticiones)
CDAAMO	.328	0	128	0
X	.114	6	48	6
CO	.238	0	16	2
CIX	.196	4	0	4
CDXA	.155	4	0	4
CAOA	.82	2	0	0
XA	.25	1	0	1
Consumo de tiempo de ejecución	–	3.358%	51.264%	2.814%

Tabla 4.4: Distribución del tiempo de ejecución entre los tres componentes principales de `funcdes()`

$$\text{max acel} = \frac{1}{1 - \left(\frac{\text{tamaño temporal}}{100}\right)} \tag{4.3}$$

El resultado que se obtiene, después de sustituir los valores, es igual a una aceleración maxima de 2.34, lo cual significa que una implementación ideal en hardware sería 2.34 veces más rápida compararada con la implementación puramente en software ejecutada en una computadora Pentium III a 880 MHz.

Capítulo 5

Diseño e implantación del circuito de cifrado/descifrado

En este capítulo se presenta el diseño e implementación en hardware del algoritmo específico de DES. De acuerdo al estudio realizado al algoritmo en el capítulo anterior, se determinó que existen siete patrones de secuencias de operaciones que pueden realizar todo el procesamiento requerido por la función candidata `funcdes()`. Tomando como base a estos siete patrones, se proyectaron los circuitos digitales usando únicamente compuertas lógicas y dispositivos de hardware. Se presentan las generalidades del diseño, las unidades funcionales que los componen, detalles de la implantación y los resultados del desempeño obtenido. Al final se presenta el diseño final en hardware/software del sistema.

5.1 Generalidades del diseño.

En esta sección se presentan algunas consideraciones que preceden al diseño de los circuitos como es el dispositivo empleado, el ciclo de diseño en el FPGA, los recursos lógicos utilizados y la relación de señales ocupadas en cada uno de los circuitos.

5.1.1 Dispositivo y Herramientas de diseño.

Para poder iniciar la implantación fué necesario seleccionar un dispositivo específico. Se decidió por el dispositivo XC4013XLBG256 de la familia X4000XL de Xilinx, un fabricante de FPGAs cuyos datos técnicos se presentan en la Tabla 5.1. Esta decisión estuvo basada principalmente en la capacidad del FPGA para poder implementar elementos críticos, como las ocho cajas S.

Para el desarrollo de los proyectos se usó el grupo de herramientas de diseño de Xilinx Foundation F4.1i (Build 3.1.195), que permitió implementar el diseño en un FPGA de Xilinx. La tabla 5.2 presenta un resumen de las herramientas utilizadas.

Dispositivo XC4013XLBG256			
Celdas lógicas	1,368	Número de flip-flops	1,536
Máximo número de compuertas lógicas (No RAM)	13,000	Máximo tamaño en bits de la RAM (No lógicos)	18,432
Rango de compuertas típico (lógicos y RAM)	10,000 - - 30,000	Máximo número de E/S de usuario	192
Matriz de CLB's	24×24	Total de CLB's	576

Tabla 5.1: Datos técnicos para el dispositivo XC4013 de Xilinx.

Etapa de desarrollo	Herramienta de software (versión)
Administrador de proyectos	Project Manager (Build 6.00.18c).
Entrada de los diseños	
VHDL	HDL Editor (Build 0.42)
Diagramas esquemáticos	Schematic Editor (Build 3.3.36)
Compilador de alto nivel	DPMCOMP (versión 3.1.1.0z)
Simulación funcional	Logic Simulator (Build 3.165)
Sintesis de bajo nivel	FPGA Express Xilinx Edition 3.6.1 de Synopsys Ngdbuild E.33 (versión liberada 4.1.03i) Edif2ngd E.33 (versión liberada 4.1.03i)
Mapear	Map E.33 (versión liberada 4.1.03i)
Enrutar	Par E.33 (versión liberada 4.1.03i.)
Análisis de retrasos y sincronización	Trc E.30 (versión liberada 4.1.03i) ngdanno E.33 (versión liberada 4.1.03i) ngd2edif E.33 (versión liberada 4.1.03i)
Bitstream	Bitgen E.33 (versión liberada 4.1.03i).

Tabla 5.2: Herramientas de software utilizadas en el proceso de diseño.

5.1.2 Ciclo de diseño de circuitos en FPGA's.

El ciclo general del diseño en hardware para este trabajo consistió de los siguientes pasos:

1. Proyección de los patrones de secuencia a implementar.
   Básicamente en esta parte se hizo la migración de software a hardware de las operaciones lógicas necesarias.

2. Optimización de la arquitectura.
   Aquí se buscaron posibles mejoras a los circuitos para obtener un mejor desempeño de ellos.

3. Implementación en bloques básicos de VHDL (Entrada del diseño).
   Se utilizó el método de diseño Top-Down, de modo que se capturó la idea usando un alto nivel de abstracción y después se fue incrementando el nivel de detalle de acuerdo al circuito propuesto. Se usaron diagramas esquemáticos y en el nivel más bajo descripciones mediante VHDL.

4. Simulación de cada uno de los circuitos a nivel de registro de transferencia (RTL).
   Una vez capturado el diseño, éste fue compilado para pasar de un modelo en VHDL, una síntesis de alto nivel, hasta llevar al circuito a una descripción estructural a nivel de transferencia de registros. Inmediatamente después, se realizó la simulación funcional a nivel RTL, para probar los diseños.

5. Implementación en un dispositivo FPGA específico.
   Después de obtener las simulaciones funcionales requeridas, se procedió a implementarlos en el dispositivo. Este proceso de implementar se compone de las etapas que se describen a continuación:

   (a) Síntesis y optimización lógica (Translated and optimization).
   Se realiza una traducción del diseño a una netlist. La netlist consiste de una lista de componentes sin jerarquía, un lista de conexiones plana (flaten), las entradas y las salidas. Esta síntesis, crea una descripción del diseño a nivel de compuertas con el formato de Xilinx.

   (b) Mapeo (Map or Placement).
   El netlist creado anteriormente, es usado para colocar o distribuir el circuito entre los recursos disponibles en el dispositivo utilizado.

   (c) Enrutar (Routing).
   Se genera un enrutado de las señales a través de los recursos de conexión que provee el dispositivo utilizado, tomando como entrada tres archivos: un archivo netlist de diseño, un archivo de restricciones (constraint) generado por Synopsys y un archivo de restricciones de usuario en donde se especifica el máximo periodo de reloj deseado y las asignaciones de pines.

   (d) Retro-anotación del diseño (Back-annotated).
   Se hace una retro-anotación (back-anotation) de los datos de retardo y sincronización de cada módulo y de los recursos de interconexión, en la netlist del circuito. El mínimo periodo de reloj, para el diseño dado, esta garantizado por Xilinx y sin embargo puede ser un tanto pesimista. Estos resultados de retrasos son usados para nuestros cálculos de velocidades.

6. Verificación de cada uno de los circuitos.
   Se verifica la retroanotación del diseño y se comprueba el grado de aceptabilidad del modelo de simulación, generado por la herramienta de Xilinx en el paso de implementación. Este modelo de simulación contiene retardos producidos por la red física actual, los CLBs, y los pines "pad"introducidos por la configuración del dispositivo.

7. Configurar (Bit-stream).
   Los resultados finales de los pasos anteriores generan un muy buen modelo de simulación, datos exactos de sincronización y una cadena de bits que puede ser usado para programar el chip FPGA.

La Figura 5.1 presenta un diagrama de flujo del diseño de los circuitos con las herramientas de Xilinx Foundation 4.1i.

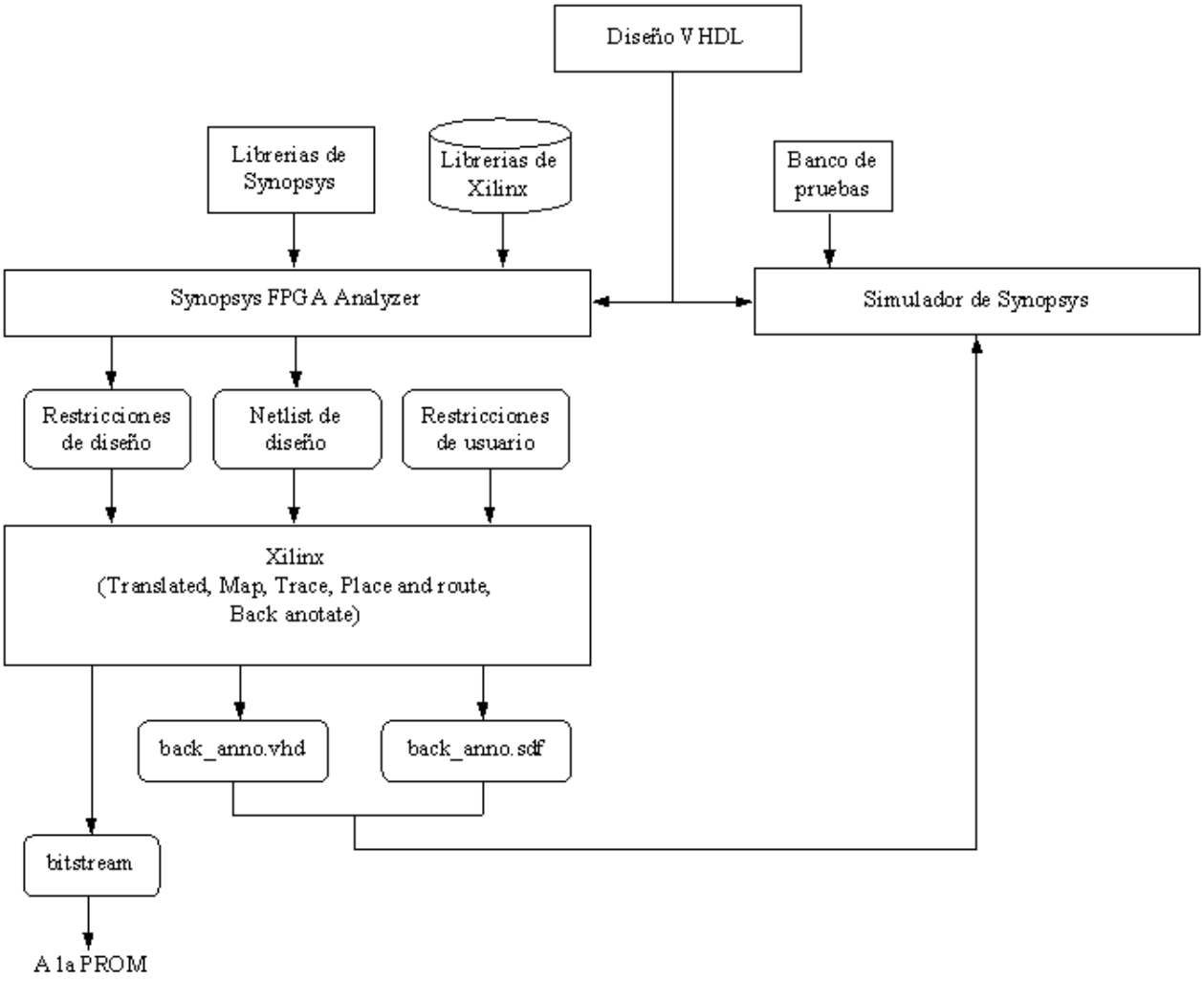

Figura 5.1: Diagrama de flujo para el diseño de los circuitos usando las herramientas de Xilinx

5.1.3 Recursos lógicos.

La implementación inició estableciendo la arquitectura de hardware del diseño y en el que se proponene datos de entrada/salida parametrizados de 32 bits. Después cada arquitectura se fragmentó en pequeñas unidades computacionales llamadas bloques funcionales, que podían usarse en diseños más avanzados, lo que permitiría una reutilización de componentes. En esta

descomposición, se analizaron ciertas optimizaciones que podían ser hechas para implementarse eficientemente en hardware.

Los bloques funcionales o símbolos describen el comportamiento de un circuito mediante código VHDL (ver Apéndice A). Los tiempos de compilación y síntesis de estos símbolos utilizados fueron en general muy cortos, lo que indica que son circuitos sencillos y fácilmente sintetizables. Los bloques funcionales que conforman los circuitos de los patrones de secuencias encontrados pertenecen a dispositivos de lógica combinacional y se describen a continuación:

- Seleccionador de bits de 32 a 6 (SB). Es un dispositivo que selecciona los 6 bits menos significativos de cada uno de los cuatro octetos que forman una cadena de entrada. Es un elemento síncrono y se describió en 45 líneas de código en VHDL.

- Memoria ROM para Cajas SP (SPs). Son tablas look-up que contienen 64 valores de 32 bits. Corresponde a un símbolo asíncrono, que utilizó 177 líneas de código para su descripción. Un estudio de Greg Haskins [Has97] muestra que usar elementos ROM es el modo más eficiente para implementar Cajas-S.

- Compuertas compuestas OR, AND y XOR (CompOR, CompAND, CompXOR). Son compuertas lógicas estándar que incluyen un latch y una compuerta de tercer estado. Permiten mantener el valor de la señal durante todo un ciclo de reloj y poner alta impedancia a la salida cuando no están habilitadas. Son elementos síncronos y su descripción requirió de 29 líneas de código en promedio.

- Registro de corrimiento a la derecha (RCD), Registro de corrimiento a la izquierda (RCI) y Registro de corrimiento circular. Un RCD y un RCI son circuitos que desplazan bits de una cadena de entrada, un número fijo de posiciones a la derecha o a la izquierda. Los espacios que quedan vacíos al desplazarse se rellenan con valores de cero. Mientras tanto un registro de corrimiento circular corre una cadena de bits, los bits que salen del registro aparecen del otro lado de la cadena de bits a desplazar, creando un efecto de enlace circular. Se pueden obtener a partir de un RCD y un RCI cuyos bits de corrimiento para cada registro son complementarios en un complemento a 32. La salida de ambos registros se procesan finalmente en una compuerta OR. El registro de corrimiento a la izquierda ocupó en promedio 64 líneas de código. Mientras que para el registro de corrimiento a la derecha se utilizaron 56 líneas de código. Estos registros de corrimiento son esencialmente permutaciones que reordenan los bits de una secuencia y pueden ser implementados mediante cableado únicamente.

- Multiplexor. Son circuitos de lógica combinacional que seleccionan información binaria de 2^n líneas de entrada y dirige la información a una sola línea de salida. La selección de una línea de entrada particular es controlada por n entradas de selección, cuyos bits de combinación determinan que entrada es seleccionada [MK00]. Para su descripción se requirió de 8 líneas de código.

5.1.4 Señales utilizadas.

Las señales son las líneas que transportan información dentro del circuito y pueden tener diferentes anchos dependiendo de para qué se utilicen. A continuación se presenta una lista de las señales comunes que son utilizadas en la etapa de simulación de los circuitos. Después del nombre de la señal se da una descripción general de su uso.

- RLJ es la señal de reloj requerida para sincroniza a los elementos síncronos del circuito.
- ENT_UNO corresponde al primer dato de entrada que se procesa en el circuito.
- HAB es la señal que habilita la operación del circuito.
- ENT_DOS corresponde al segundo dato de entrada que se procesa en el circuito.
- DAT_SAL es la señal contiene los datos de salida, después de que el circuito ha realizado todo el procesamiento.
- SET es una señal que se activa con el flanco de bajada del reloj y después de que HAB ha entrado en operación. Esta señal permite habilitar específicamente a la compuerta de tercer estado a la salida de la última función lógica OR, XOR o AND, según sea el caso.
- BITCORRD indica el numero de bits que se van desplazar a la derecha, del dato original.
- BITCORRI indica el número de bits que se tiene que desplazar a la izquierda al dato de entrada original.
- SALRCD esta señal contiene los datos de salida que se obtienen del registro de corrimiento a la derecha.
- SALRCI esta señal contiene los datos de salida que se obtienen del registro de corrimiento a la izquierda.

5.2 Diseño e implementación de los circuitos.

La migración a hardware, de los patrones de secuencias de operaciones que componen a la función funcdes() en software, fue un paso casi directo, permitiendo la conversión de cada una de ellas en un circuito digital de manera fácil y gradual. En las siguientes subsecciones se muestran los diseños de los circuitos digitales, siguiendo el orden determinado por el tiempo de consumo de cada uno de los patrones encontrados. Estos constan de una breve descripción del patrón, un diagrama esquemático, un diagrama de forma de onda conseguido con el simulador lógico y la descripción de las señales que se involucran en ese diagrama. Hasta el final, se presenta un resumen de los recursos utilizados así como un resumen del rendimiento de procesamiento obtenido en cada uno de los diseños.

5.2.1 Circuito CDAAMO.

El patrón CDAAMO (Corrimiento a la derecha, And, Acceso a Memoria, Or), presentó el más alto tiempo de ejecución en software, con el 42.11%, debido al alto grado de repetición encontrado en el algoritmo, 128 repeticiones. La Figura 5.2 muestra el diseño esquemático de este circuito.

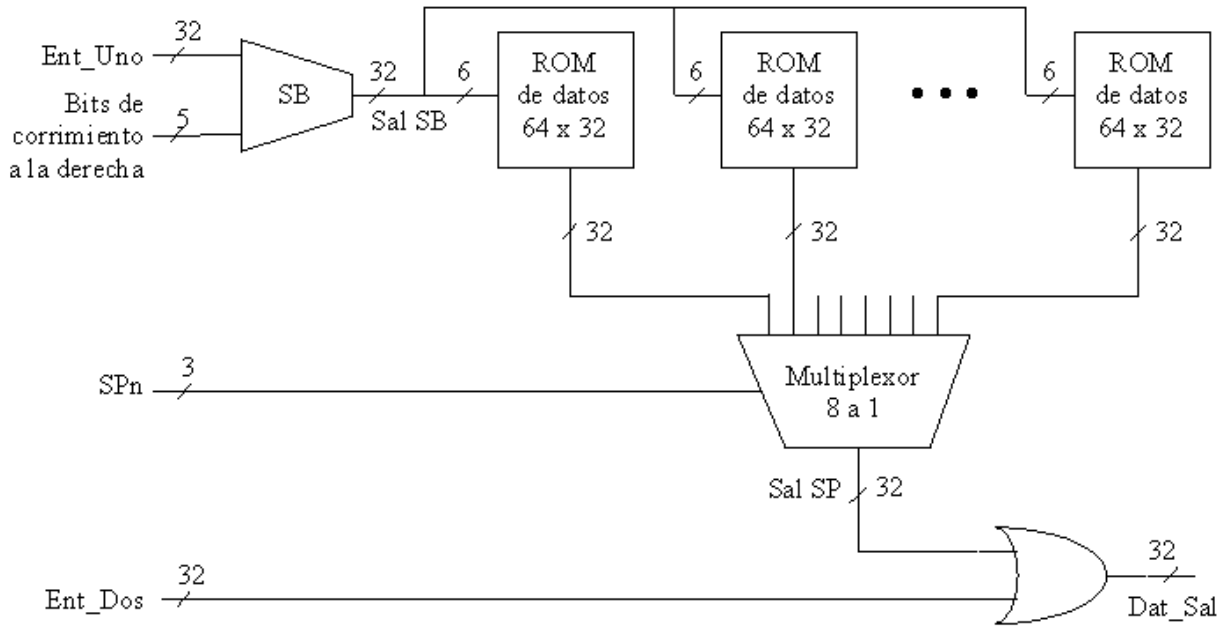

Figura 5.2: Secuencia de operación de CDAAMO.

La descripción del funcionamiento de este circuito es el siguiente: Un símbolo llamado SB (Seleccionador de bits) se encarga de seleccionar 6 bits de una cadena de 32 bits proveniente de "*Ent_Uno*", según el valor que tenga "*Bits de corrimiento a la derecha*" y que puede ser de 0, 8, 16 o 24. Este resultado de 6 bits proporciona la dirección de una localidad de una de las ocho ROM SPs, habilitada esta última por el valor de "*SPn*", que puede tomar los valores de 7, 5, 3, 1, 8, 6, 4, 2 . Una vez que esta listo el valor de la memoria correspondiente, se introduce a un símbolo OR junto con otro valor proveniente de "*Ent_Dos*". El valor final se recupera en "*Dat_Sal*".

La operación completa tomaría un retardo l debido al seleccionador de bits, un retardo m por el acceso y la resolución del dato en la ROM, más un retardo n debido a la operación de la compuerta OR. Por lo tanto, el circuito sufriría de un retraso de al menos tres ciclos de reloj. No obstante, utilizando tanto el flanco de subida del reloj, como el flanco de bajada e intercalando elementos síncronos, como SB y la compuerta OR, con elementos asíncronos, como la ROM SPS, podemos obtener resultados en un solo ciclo de reloj. Esto ayuda porque se aprovecha todo el ciclo de reloj para realizar las operaciones necesarias. La Figura 5.3 muestra el diagrama de forma de onda obtenido por el simulador lógico para el circuito CDAAMO. En esta figura se pueden ver los datos de entrada y la trayectoria que siguen

dentro del circuito a través del tiempo hasta que llegan a la salida del circuito. Algunas de las señales, no descritas anteriormente, son las siguientes:

- HAB permite activar el registro de corrimiento a la derecha y activar a la ROM SPs.
- BITCORRD especifica los 6 bits a seleccionar de la cadena de 32 bits.
- SALSB permite observar la salida del seleccionador de bits.
- SPN corresponde a una señal que habilita una y solo una de las ocho ROM SPs.
- SALSP permite ver el valor de salida de la caja SP habilitada.
- ENT_DOS se utiliza en la operación OR junto con el dato de salida de la respectiva caja SP habilitada.
- GSR es una señal auxiliar requerida por el simulador.

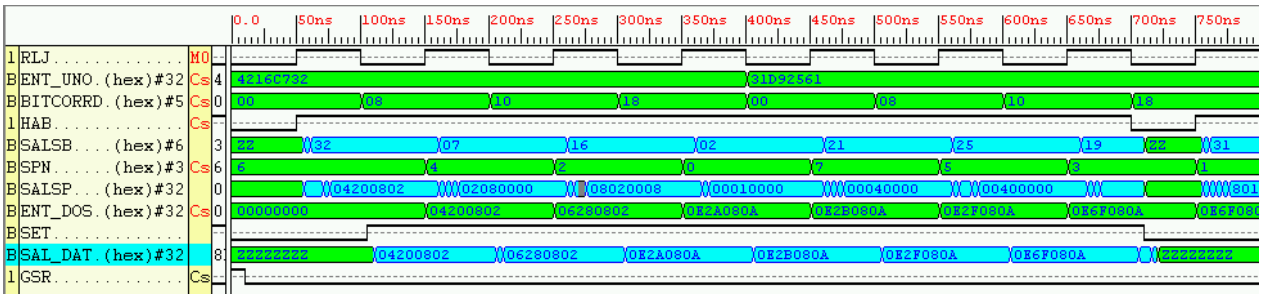

Figura 5.3: Diagrama de forma de onda obtenida para el circuito CDAAMO, para una escala de 5ns/division y un periodo de reloj de 100ns.

Observe que a la salida se tiene un retraso de medio ciclo con respecto al flanco de subida del reloj, esto se debe a que se utiliza el flanco de bajada para activar el elemento de salida del circuito. Pero además del retraso de medio ciclo existe un retardo promedio de 6ns más, desde que se origina el flanco de bajada del reloj, hasta que el dato en esta señal es estable.

5.2.2 Circuito X.

El patrón X (or-eXclusivo), fue el segundo patrón en orden de importancia de acuerdo a su consumo del 6.89% del tiempo de ejecución en software y con 60 repeticiones en el algoritmo. La Figura 5.4 muestra el diseño esquemático de este circuito.

La descripción del funcionamiento de este circuito es el siguiente: Una compuerta XOR, realiza una operación XOR entre los operandos "*Ent_Uno*" y "*Ent_Dos*". El resultado final es obtenido en "*Sal_Dat*".

Esta operación tomaría un retardo l debido a la compuerta XOR, lo que indica que es necesario un ciclo de reloj, como puede verse en la Figura 5.5. En esta Figura se muestra el diagrama de forma de onda correspondiente al circuito X. Obsérvese en la figura el retraso sufrido por la señal desde que llega el flanco de subida del reloj hasta que este dato aparece estable a la salida, dicho retardo es de 4.5ns.

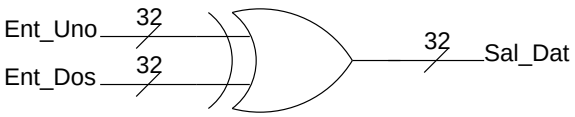

Figura 5.4: Secuencia de operación X.

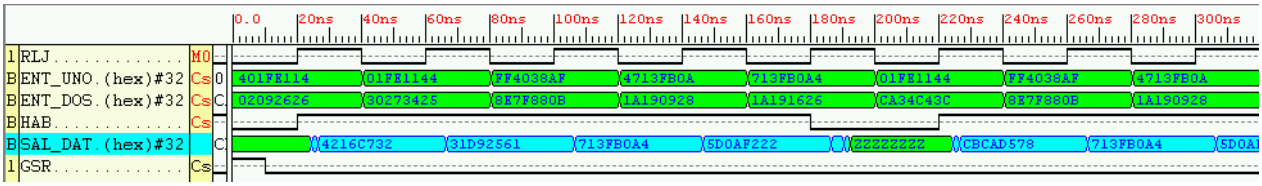

Figura 5.5: Diagrama de forma de onda obtenida para el circuito X, usando una escala de 2ns. Por división y un periodo de reloj de 40ns.

5.2.3 Circuito CO.

El patrón CO (Corrimientos a la derecha e izquierda, OR) fue el tercero en importancia, con un consumo de tiempo de ejecución de 4.30% en software, debido a sus 18 repeticiones dentro del algoritmo. La Figura 5.6 muestra el diseño esquemático del circuito.

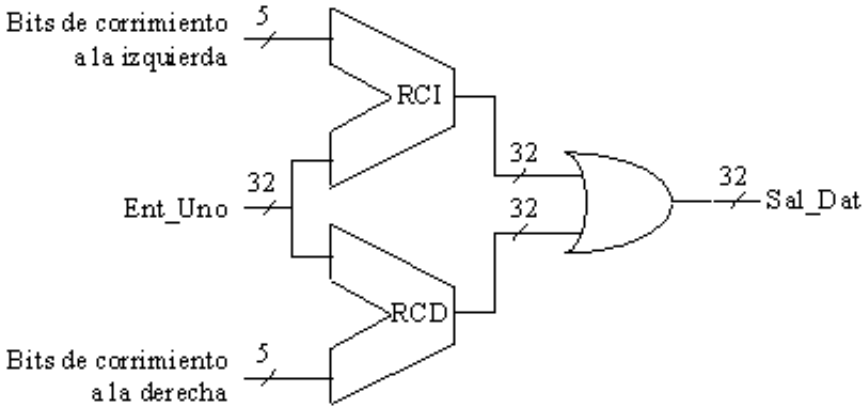

Figura 5.6: Secuencia de operación CO.

La descripción del funcionamiento de este circuito es el siguiente: CO realiza un corrimiento a la izquierda al operando "*Ent_Uno*" de 28 o 31 bits según sea el valor de "*Bits de corrimiento a la izquierda*". También hace un corrimiento a la derecha al mismo operando "*Ent_Uno*" de 4 o 1 bits, según sea el caso del valor de "*Bits de corrimiento a la derecha*". El resultado de estos dos corrimientos es procesado enseguida en una compuerta OR para finalmente enviar los datos de salida a "*Sal_Dat*".

La operación completa tomaría un retardo l debido al corrimiento a la derecha y al corrimiento a la izquierda que se aplica al mismo dato de entrada, solo es un retardo porque ambos se ejecutarían en paralelo. Otro retardo m es necesario debido a la compuerta OR. Por

lo tanto el retardo total tendría que ser dos ciclos de reloj. En cambio, se consigue realizar toda la operación en un solo ciclo utilizando tanto el flanco de subida del reloj como el flanco de bajada, como puede verse en la Figura 5.7.

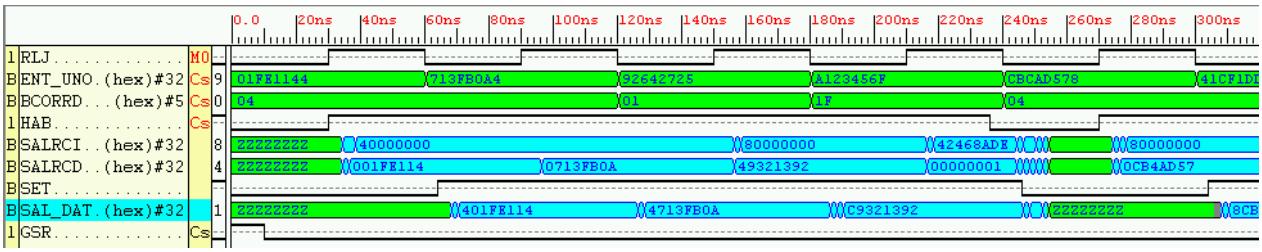

Figura 5.7: Diagrama de forma de onda obtenida para el circuito CO, con una escala de 2 ns. Por división y un periodo de reloj de 60 ns.

En la señal de salida, del diagrama de forma de onda de CO, se puede apreciar como el dato final aparece unos nanosegundos después de que llega el flanco de bajada del reloj, por lo que existe un retardo de 8.7ns desde que parece el flanco de bajada del reloj hasta que el dato esta disponible a la salida.

5.2.4 Circuito CIX.

El patrón CIX (Corrimiento a la Izquierda, Xor), presentó un tiempo de ejecución de 1.57% como consecuencia de sus 8 repeticiones en el código. La Figura 5.8 presenta el diagrama esquemático de este diseño.

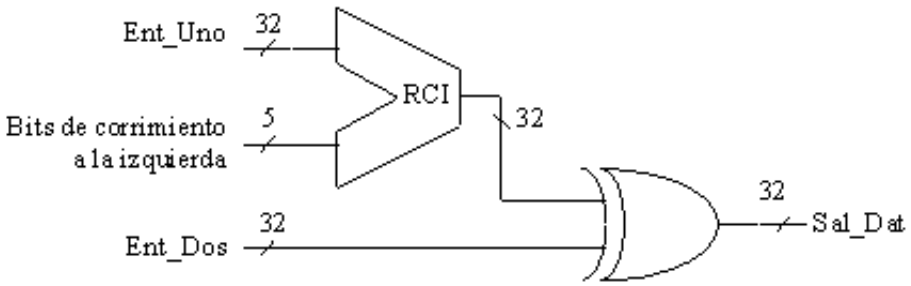

Figura 5.8: Secuencia de operación de CIX.

A continuación se describe el funcionamiento de este circuito. CIX realiza un corrimiento a la izquierda al operando "*Ent_Uno*", dicho corrimiento puede ser de 4, 16, 2 u 8 bits, según sea el caso para el valor de "*Bits de corrimiento a la izquierda*". El resultado de esta operación se introduce junto con el valor de "*Ent_Dos*" a una operación XOR. Por último el valor final es manejado en "*Sal_Dat*".

La operación completa tomaría un retardo l debido al corrimiento a la derecha y un retardo m debido a la compuerta XOR. Por lo que se requerirían de dos ciclos de reloj. Pero,

siguiendo la estrategia aplicada anteriormente se pudo mejorar el tiempo ocupado, haciendo uso del flanco de subida y del flanco de bajada del reloj. Por lo que se consiguió realizar la operación en un solo ciclo de reloj.

El diagrama de forma de onda que corresponde a este circuito se muestra en la Figura 5.9. En esta figura se puede observar que existe un intervalo de tiempo de aproximadamente 5.9ns., desde que aparece el flanco de bajada del reloj, hasta que el valor aparece finalmente estable a la salida del circuito.

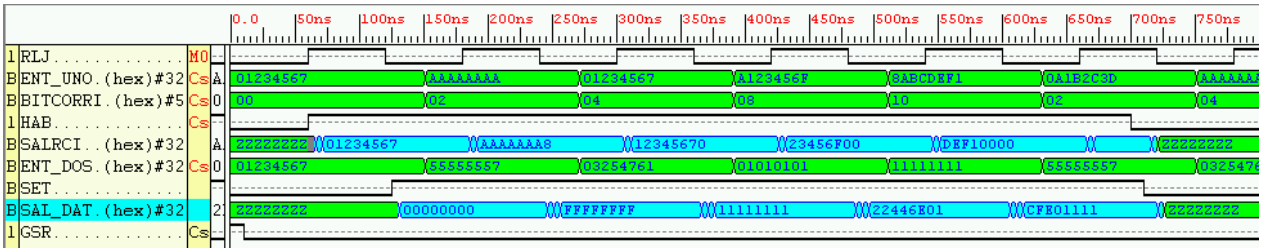

Figura 5.9: Diagrama de forma de onda obtenida para el circuito CIX, para una escala de 5ns por división y un periodo de reloj de 120ns.

5.2.5 Circuito CDXA.

El patrón CDXA (Corrimiento a la derecha, Xor, And) presenta un consumo del tiempo de ejecución de 1.24% y un número de 8 repeticiones en el código. La Figura 5.10 muestra el diagrama esquemático del circuito.

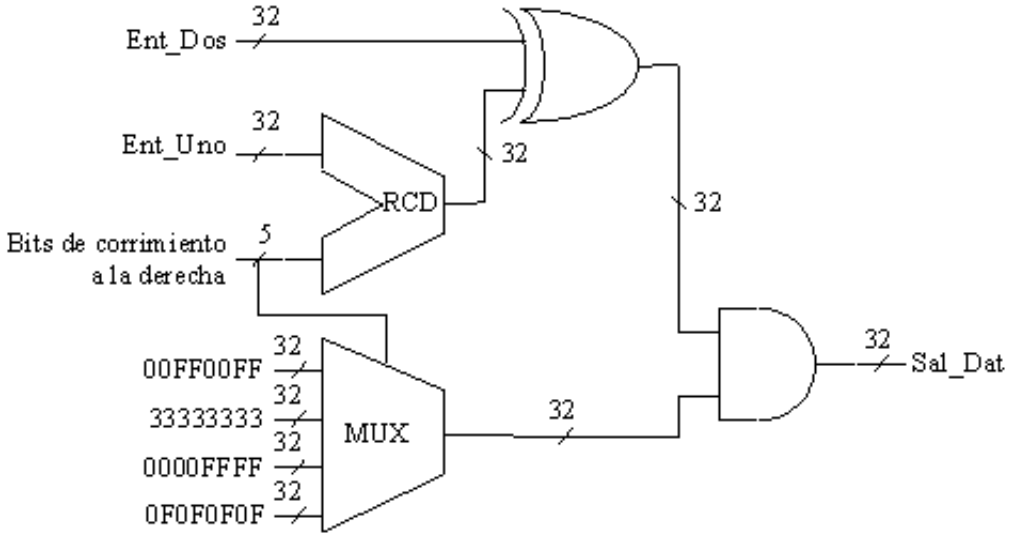

Figura 5.10: Secuencia de operación CDXA.

A continuación se describe el funcionamiento de este circuito. CDXA primero trabaja sobre el operando "*Ent_Uno*" realizando un corrimiento a la derecha con valores de 4, 16, 2, u 8, según sea el caso para el valor de "*Bits de corrimiento a la derecha*". El resultado de la operación anterior se introduce junto con el valor de "*Ent_Dos*" dentro de una operación XOR. El valor de los bits de corrimiento al mismo tiempo funcionan como una señal de control para un multiplexor que determina un valor de "Máscara" a utilizar, dichos valores pueden ser 0F0F0F0F, 0000FFFF, 33333333, 00FF00FF00FF, guardando una correspondencia, en orden con los valores de bits de corrimiento. Uno de estos valores junto con el valor obtenido en la operación XOR, son procesados mediante una operación AND. Por último el valor se obtiene en "*Sal_Dat*".

La operación completa tomaría un retardo l debido al tiempo que toma el corrimiento a la derecha, para desplazar los datos, al mismo tiempo el multiplexor selecciona el valor de máscara a utilizar. Se requiere de otro retardo m debido a la operación compuesta XOR y otro retardo n para realizar la operación AND. Sin embargo, siguiendo el mismo principio utilizado en los diseños anteriores se consigue utilizar todo el ciclo de reloj haciendo uso del flanco de subida y del flanco de bajada, permitiendo realizar toda la operación en un solo ciclo de reloj. Esto puede observarse en la Figura 5.11.

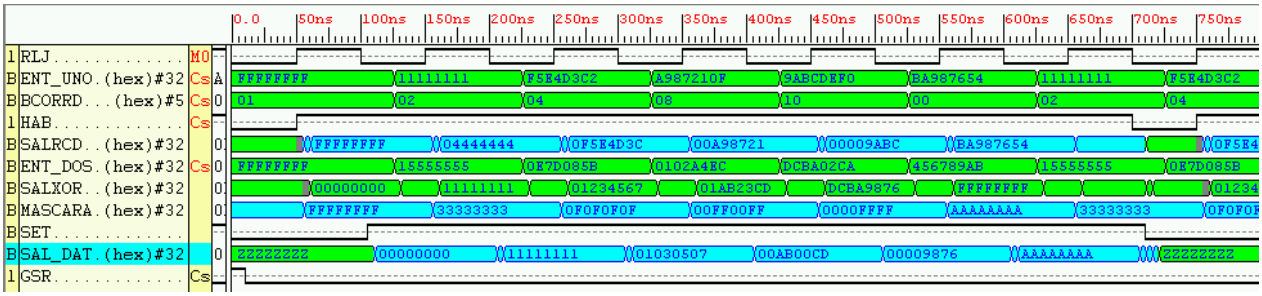

Figura 5.11: Diagrama de forma de onda obtenida para el circuito CDXA, con una escala de 5ns por división y un periodo de reloj de 100ns.

En el diagrama de forma de onda para este circuito, que aparece en la Figura 5.11, se puede apreciar que el retraso para que la señal aparezca estable a la salida tiene un valor de 6ns aproximadamente. Además en esta Figura se ven algunas señales que no han sido definidas y que se presentan a continuación:

- SALXOR esta señal contiene el valor obtenido a la salida el circuito Xor.
- MASCARA es el valor de máscara requerido en la operación AND final y seleccionado por el multiplexor.

5.2.6 Circuito CAOA.

Uno de los últimos patrones el CAOA (Corrimiento a la derecha e izquierda, And, Or, And) consumió el 0.82% del tiempo de ejecución y tuvo 2 repeticiones en el código.

CAOA realiza un corrimiento a la derecha de 31 bits al operando "*Ent_Uno*", a este corrimiento le aplica un máscara con valor de 0x1L procesados ambos en una operación AND. También aplica un corrimiento a la izquierda de 1 bit al operando "*Ent_Uno*". Las salidas de la operación AND y la salida del último corrimiento son procesadas en una operación OR. Finalmente la salida de la operación OR, junto con otro valor de mascara 0xFFFFFFFFL son procesadas en una operación AND. En esencia CAOA realiza una rotación circular a la izquierda de 1 bit haciendo que el bit 1 sea el bit 32, el bit 2 sea el bit 1, y así sucesivamente.

Los dos valores de mascara utilizados, 0x1L y 0xFFFFFFFFL son un mecanismo usado en software para preservar el uso de los mismos tipos de datos (tipo long), pero en hardware, al utilizarse elementos parametrizados, estos mecanismos no son necesarios. De este modo las operaciones AND pueden ser eliminadas, quedando únicamente los corrimientos y la operación OR. Dado lo anterior se trata de un caso particular del circuito CO, pudiéndose aprovechar el diseño de este último y con algunas modificaciones conseguir la inclusión de CAOA dentro de CO. La figura 5.6 muestra el diagrama esquemático del circuito CO, utilizado para realizar esta secuencia de operación.

5.2.7 Circuito XA.

El ultimo de los patrones, el XA (or-eXclusivo, And), presentó un consumo del tiempo de ejecución de 0.50% con un numero de 2 repeticiones.

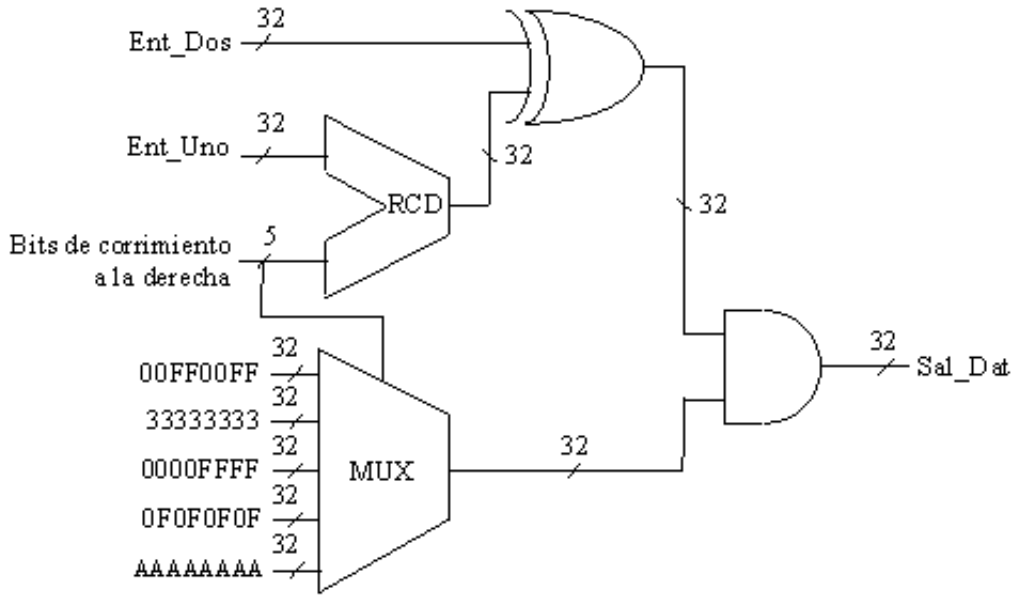

Figura 5.12: Secuencia de operación CDXA, modificada para incluir a XA.

XA realiza una operación XOR entre los operandos "*Ent_Uno*" y "*Ent_Dos*". La salida de esta operación es procesada junto con el valor de la mascara 0xAAAAAAAAL en una

compuerta AND. El resultado final es enviado a "*Sal_Dat*".

Un análisis más detenido, permitió ubicarlo en un circuito más general, el CDXA. Obsérvese, que XA realiza las dos últimas operaciones que realiza el circuito CDXA, una operación XOR y una operación AND. Si al circuito CDXA aplicamos un corrimiento a la derecha de 0 posiciones al RCD y agregamos otra entrada al multiplexor con el dato 0xAAAAAAAAL que corresponde al valor de la máscara usado por el circuito XA, tendríamos entonces el circuito equivalente a XA, pero realizado por CDXA. Además, al igual que con el circuito anterior, es poco significativo su número de repeticiones y su consumo de tiempo de ejecución, lo que resultaría en un costo de diseño no recuperable. La Figura 5.12, muestra el diseño del circuito CDXA utilizado para realizar esta secuencia de operación.

5.2.8 Recursos utilizados y resultados obtenidos.

Un resumen de los porcentajes de recursos utilizados en el dispositivo FPGA XC4013XLBG256 y un resumen de los resultados experimentales obtenidos, para cada uno de los circuitos implementados anteriormente, se presentan en la Tabla 5.3. Recuerde que los datos de resultados, son obtenidos a partir del reporte que proporciona la herramienta Trace de Xilinx después de que los componentes del diseño hayan sido colocados y ruteados (place and route) en el FPGA.

El renglon de "Número de operaciones por segundo" para cada circuito, se obtiene haciendo la consideración de que en un período minimo de reloj se realiza una operación de este tipo, pero en un segundo se realizarían $\frac{1}{\text{Período mínimo de reloj}}$ operaciones de este tipo.

El desempeño de ejecución total o "Throughput total" de cada circuito, se obtiene con el valor calculado en "Número de operaciones por segundo" multiplicado por 32, debido a que se trabajan con bloques de 32 bits a la entrada y se obtiene bloques de 32 bits a la salida, en todos los circuitos.

5.3 Diseño de la interfaz de comunicación del sistema Hardware-Software.

Es necesario establecer un modo de coexistencia entre ambos sistemas, por un lado el algoritmo en software ejecutándose en la PC y por otro lado el sistema en hardware (o máquina de cómputo basada en FPGA). Esta coexistencia permitirá establecer, entre otras cosas, el lugar de donde se tomaran los datos a procesar. Es necesario recordar que el principio de operación de un sistema hardware-software se basa en que el algoritmo en software realiza las tareas menos costosas y entonces la máquina de cómputo ejecuta a nivel de bits las operaciones restantes más complicadas, para cumplir con la especificación del algoritmo.

La arquitectura propuesta para la interfaz de comunicación entre ambos sistemas, se basa principalmente en una memoria RAM, en la que el algoritmo en software podría escribir los datos en las localidades de memoria, haciendo uso de un puerto de salida de la PC, como el

Número contenido en el FPGA – Componentes del FPGA	Nombre del circuito				
	CDAAMO	X	CO	CIX	CDXA
192 – IOB's externos	98	98	71	103	103
8 – IOBs controladores de buffers globales	1	1	1	1	1
576 – CLB's	127	17	89	66	98
1152 – Flops CLB	39	32	97	65	97
1152 – LUTs de 4 entradas	245	33	175	114	182
576 – LUTs de 3 entradas	89	89	5	28	22
8 – Buffers globales de bajo retardo (BUFGLSs-global low-skew buffers)	1	1	1	1	1
1248 – Buffers de Tercer Estado (TBUFs - 3-State buffers)	70	32	96	64	64
Periodo mínimo del reloj (ns)	55.564	7.467	17.914	15.426	12.944
Frecuencia máxima del reloj (MHz)	18.001	133.923	55.892	64.826	77.256
Número de operaciones por segundo (1×10^3)	18 001	133 923	55 892	64 826	77 256
Throughput total (Gbits/seg)	0.576	4.285	1.786	2.074	2.472

Tabla 5.3: Resumen de recursos utilizados y resultados obtenidos para cada circuito.

puerto paralelo. Mientras que el hardware podría leer esos datos en las localidades de memoria, procesarlos y volverlos a escribir en la misma o en alguna otra localidad de memoria. Una vez guardado el dato por el hardware, nuevamente el software podría leer esté último dato y utilizarlo en el procesamiento final del algoritmo.

La Figura 5.13 muestra el diagrama a bloques del circuito propuesto que realizaría las operaciones de encriptado y en donde puede verse la interfaz de comunicación del sistema. En esta figura se puede ver que la señales de control necesarias, para la memoria RAM, como son HabLE, HabCh, ReIni, están proporcionadas por el circuito FUNCDES. Los buses de entrada y salida de datos de 64 bits cada uno, comunican los datos de entrada de la memoria RAM al circuito FUNCDES, y los datos de salida del circuito FUNCDES a la memoria RAM. El direccionamiento de datos, DirDat, es una señal externa, que puede ser controlada por alguna otra entidad, que bien podría ser el algoritmo en software ejecutándose en la PC. Actualmente esta memoria tiene 8 localidades de 64 bits.

Otras señales importantes son las señales externa INICIO1 e INICIO2. Estas permiten configurar uno de los cuatro estados posibles del circuito, la Tabla 5.4 describe cada uno de estos estados.

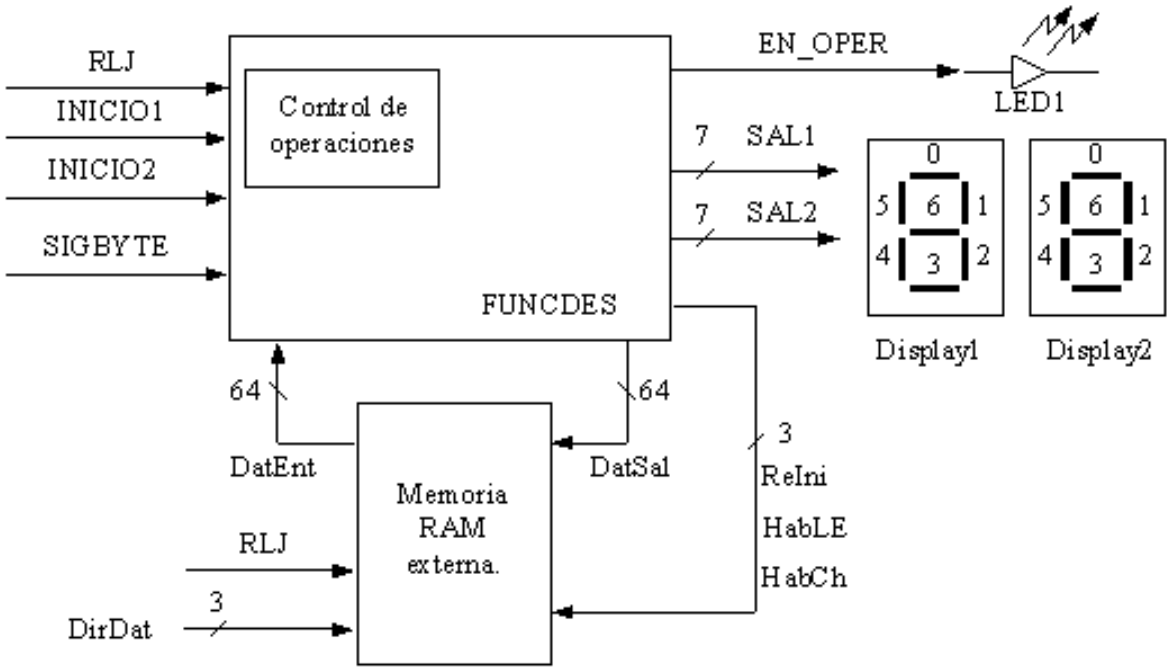

Figura 5.13: Propuesta del circuito que realizaría las operaciones de encriptado. Aquí se puede ver a la RAM como una interfaz de comunicación de datos.

Valor de Inicio1	Valor de Inicio2	Operación	Descripción
1	1	Direccionar	Direcciona la localidad de la memoria RAM de donde se leerán los 64 bits de texto plano.
1	0	Cargar	Copia los datos de la memoria RAM a un registro de propósito general, de aquí serán tomados al iniciar el procesamiento
0	1	Ver	Permite visualizar los datos contenidos en alguna localidad de la memoria RAM, mediante un display de 7 segmentos
0	0	Activar	Activa el funcionamiento del circuito que encripta los datos.

Tabla 5.4: Descripción de los estados del circuito.

5.4 Implementación de `funcdes()` en hardware.

Es claro que al agrupar todas las secuencias de la sección 5.2 de este capítulo, se puede recurrir a procesamientos paralelos que incrementen su desempeño. Para lograr agrupar todos

los circuitos, se tuvo que utilizar un dispositivo más grande que el que se había utilizado con anterioridad. El dispositivo mencionado corresponde a un Virtex XCV300 de Xilinx que es un FPGA con un sistema de 322, 970 compuertas, un arreglo de CLBs de 32×48, 6 912 Celdas lógicas, 316 E/S máximas disponibles como máximo, 65 536 bits en Bloques de RAM y 98 304 bits de RAM seleccionable.

La Figura 5.14 muestra el diagrama esquemático general de la implementación del circuito FUNCDES. En esta figura aparecen todos los elementos que componen al circuito y que ayudan a completar el procesamiento de datos. Entre estos elementos se encuentran la unidad de control de estado, la ROM de claves, el registro, un contador de operaciones, la unidad de control de encriptado, la permutación inicial, la permutación final y el más importante, el circuito CO_X_CDAAMO_X, donde se realiza la permutación de cajas S y permutación P.

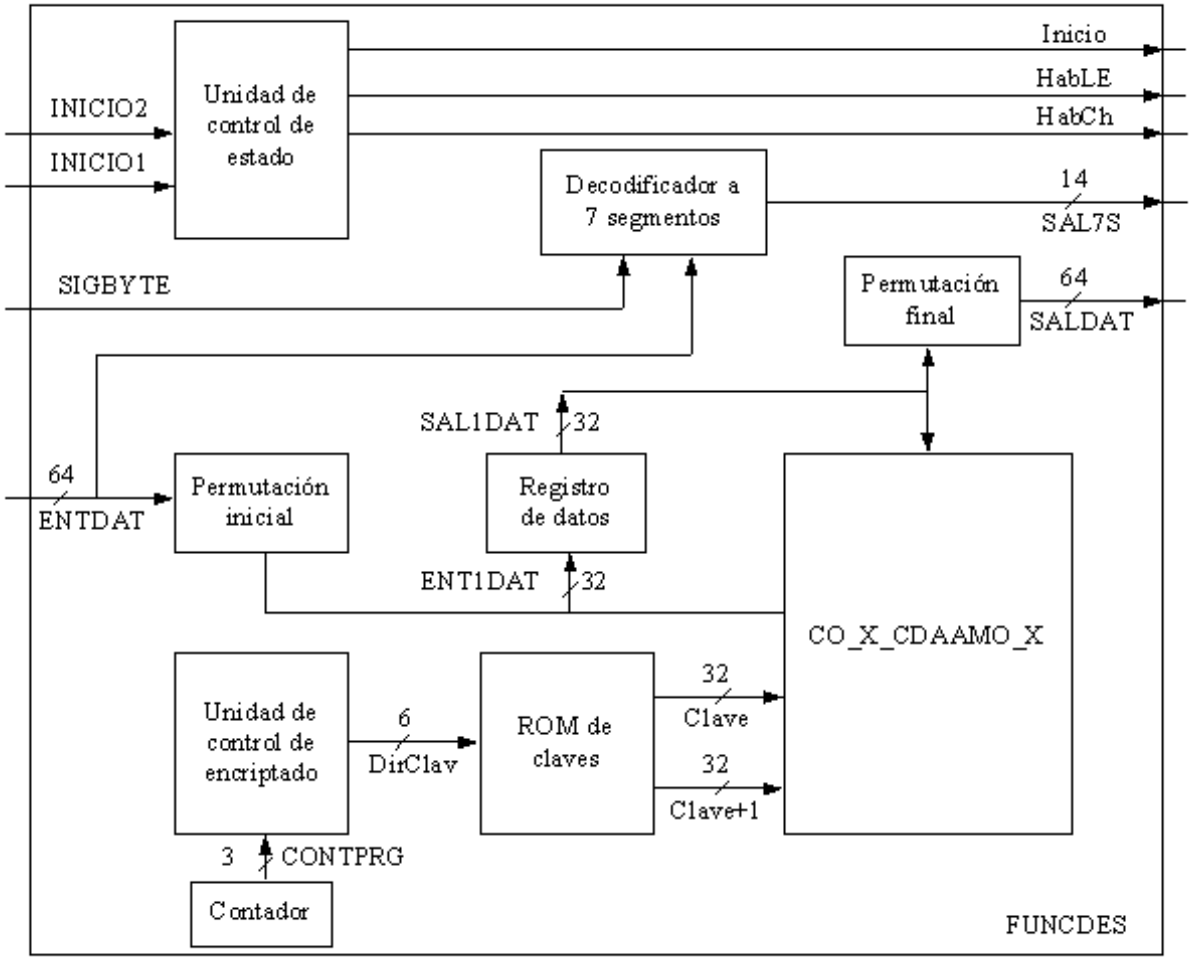

Figura 5.14: Implementación del circuito FUNCDES

Una vez propuesta la arquitectura del circuito se puede detallar cada uno de los cuatro estados en que éste puede encontrarse, ver Tabla 5.4. En el estado Direccionar, que es el más simple de todos, el circuito se encuentra inactivo y prácticamente no se hace nada, por

lo que es un buen momento para indicar la dirección de la memoria de donde se tomarán datos. En el estado Cargar se realizan básicamente tres funciones: se habilita la lectura de la memoria RAM, se habilita la permutación inicial y se habilita la escritura en el registro. Toma aproximadamente dos ciclos de reloj a partir del primer flanco de subida, un ciclo es para leer de la memoria RAM y un ciclo para escribir el dato en el registro. El estado Activar, es el más importante y aquí se realizan varias funciones como son: se habilita el registro de propósito general para lectura y escritura, se habilitan los circuitos para realizar las operaciones de encriptado, también se realizan funciones para habilitar la permutación final y para habilitar la escritura del dato de salida a la memoria RAM. En el estado Ver se habilita la lectura de la memoria RAM y se habilita el dispositivo para decodificar a los datos de entrada provenientes de la memoria a un código de 7 segmentos.

La Figura 5.15 muestra el diagrama de estados correspondiente al sistema. En éste diagrama se pueden observar las transiciones, entre estados, permitidas para el correcto funcionamiento del circuito.

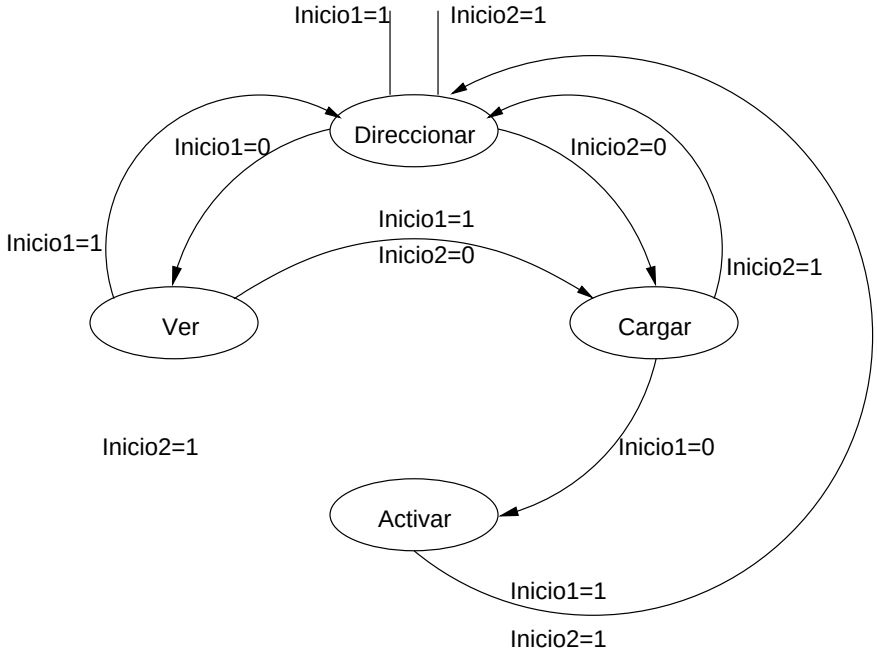

Figura 5.15: Diagrama de estados del circuito.

5.4.1 Circuitos auxiliares.

La unidad de control de estado es una parte importante del diseño ya que se encarga de las operaciones más generales que se realizan en el circuito y que corresponden a los estados del circuito ya explicados anteriormente. Además, en cada uno de estos estados realiza funciones diferentes. Así, en el estado cargar habilita la lectura de la memoria RAM, habilita el circuito de permutación inicial y habilita al registro para escribir el dato proveniente de la memoria. En el estado Activar, habilita el funcionamiento de la unidad de control y del contador. En el estado Ver, habilita la lectura de la memoria RAM y habilita al circuito decodificador.

La unidad de control de estado, también se encarga de las señales de control de la memoria RAM externa.

En cambio, la unidad de control de encriptado funciona únicamente cuando el circuito se encuentra en el estado Activar y se encarga del control de todos los dispositivos que tienen que ver con el encriptado. Algunas de sus funciones son: habilitar la lectura/escritura del registro para leer y guardar datos intermedios; proporciona la dirección de las localidades de las claves que corresponden en cada iteración; a si mismo, habilita la lectura de datos de la ROM de claves; y habilita las operaciones del circuito CO_X_CDAAMO_X.

El registro de datos sirve para guardar los datos intermedios del procesamiento entre cada una de las 16 iteraciones realizadas por el circuito. Se compone de dos registros de 32 bits.

La ROM de claves tiene los valores almacenados en un arreglo de 32 localidades de 32 bits, además cuenta con dos buses de salida para proporcionar dos valores simultáneamente.

El contador utilizado en el circuito es un contador que cuenta permanentemente en módulo 7, mientras el circuito esta activo. Algunos estado del contador son estados claves, ya que le indican a la unidad de control de encriptado, el momento en el que hay que leer y escribir en el registro, así como el momento en el que hay que administrar a la ROM una nueva dirección de claves.

El decodificador a 7 segmentos, solo opera en el estado Ver y se encarga de decodificar datos de hexadecimal a siete segmentos, previa descomposición del vector de 64 bits en pares de 4 bits. La fragmentación de los datos se realiza desde la posición de los bits más significativos a los menos significativos, esto es de izquierda a derecha. La señal externa llamada SIGBYTE permite al usuario cambiar al siguiente byte o siguiente par de 4 bits, para nuevamente decodificarlos y ponerlos en el bus de salida del decodificador.

5.4.2 Permutación inicial y permutación final.

Como ya se había mencionado anteriormente la porción del algoritmo llamada `funcdes()` se compone de tres partes principales, la primera parte es la permutación inicial, la segunda parte la forman la permutación de cajas S y permutación P, mientras que la tercera parte esta formada por la permutación final.

La arquitectura adoptada para el diseño de la permutación inicial (PI) y permutación final (PF) corresponde a una sencilla caja de cableado que se muestra en la Figura 5.16. Puede decirse que este circuito es la versión más optimizada para un circuito de permutación, en donde la principal operación es el cambio de posición de los bits. Es necesario recordar que esta operaciones no tienen un valor criptográfico, por lo que su implementación puede ser de este modo. El consumo de tiempo de ejecución no es tan significativo de acuerdo a la Tabla 4.4 del capítulo anterior tan solo ocupan el 3.358% y 2.814% para la permutación inicial y la final respectivamente, del tiempo total del algoritmo en software.

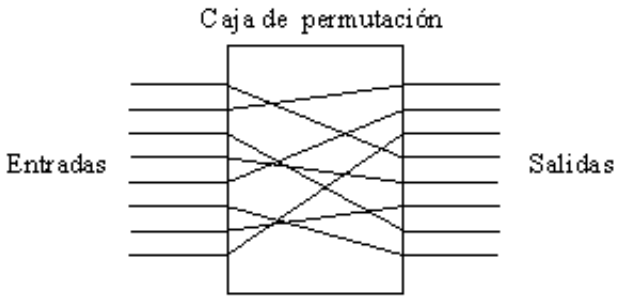

Figura 5.16: Caja de permutación para realizar la operación de PI y PF.

5.4.3 CO_X_CDAAMO_X.

De acuerdo con la Tabla 4.4 del capítulo anterior, la permutación de cajas S y P ocupa el 51.26% del tiempo total del algoritmo en software. En esta misma tabla se puede ver que esta permutación, se compone básicamente de tres secuencias de operación CO, X, CDAAAMO. Las dos primeras secuencias se localizan fácilmente en el diagrama esquemático de la Figura 5.17, en donde se muestra el diseño del circuito propuesto para esta permutación y que se ha denominado CO_X_CDAAMO_X. Observe que en la parte izquierda del diagrama se encuentra la operación CO, después dos operaciones XOR, ejecutándose en paralelo, enseguida dos módulos seleccionadores de bits, que proporcionarían al mismo tiempo ocho direcciones de localidades de la memoria ROM SP, en cuyo contenido están los valores de las cajas S. A la salida de la memoria ROM esta una compuerta OR de ocho entradas. Estas entradas de la compuerta OR corresponden a la salida de las ocho cajas S. Al final se encuentra una compuerta XOR.

El hecho de realizar dos operaciones XOR al mismo tiempo, direccionar simultáneamente las ocho localidades de la memoria ROM, recuperar simultáneamente los valores respectivos de las ochos cajas SP, así como procesarlas simultánemente en una compuerta OR, nos permite afirmar que la arquitectura desarrollada para el algoritmo adoptado en este estudio, es apropiada para utilizar un cómputo masivamente paralelo.

Un diseño de circuito para resolver este problema, pudo haber utilizado unidades de registro de propósito general y señales de control que permitieran utilizar cada uno de los elementos a la vez, ahorrando así recursos en el FPGA. Pero, el objetivo era diseñar un circuito específico para la aplicación, no teniendo que ser general y ocuparse de un elemento un tiempo a la vez, por lo que la solución propuesta fue la más apropiada, ya que explotó la mayor cantidad de paralelismo en la aplicación. Sin embargo hay que recordar que la permutación de cajas S y permutación P tiene un orden de repetición de 2×8, es decir, se encuentran repetido dos veces dentro de un ciclo que itera 8 veces por lo que se realizan 16 operaciones de este tipo.

Si se tomará al circuito FUNCDES descrito anteriormente como una instancia de las 16 necesarias para el procesamiento total, entonces, una operación completa de los 16 estados tomaría m ciclos de reloj, en una configuración en la que las instancias se conectarían de manera serial y los recursos ocupados del FPGA sería 16 veces más. Mientras que en una configuración con reutilización de una instancia, se tomarían m ciclos de reloj para el procesamiento más n ciclos de reloj para guardar los datos en un registro intermedio, pero los recursos utilizados en el FPGA sería el de una sola instancia. Lo anterior motivó a usar una estrategia, en la que se implementará solo una de las 16 instancias y se reutilizará, mediante una unidad de control y un registro de propósito general que guarda los valores obtenidos en cada una de ellas.

De acuerdo a la Figura 5.17, supongamos que las operaciones que se encuentran encerradas en la línea entrecortada, tomarían un retardo $c1$, el acceso a la memoria tomaría un retardo $c2$, la operación OR tomaría un retardo $c3$ y la operación XOR tomaría un retardo $c4$. Pensando que al menos pudieran ocupar un ciclo de reloj cada una, entonces, obligarían a tomar cuatro ciclos de reloj para completar una operación de permutación S y P. En cambio, usando elementos asíncronos como son: CO, XOR21, XOR11, SelBits, ROM SP, OR, conectados uno detrás de otro y únicamente la última compuerta XOR3 como elemento síncrono se aprovecha toda la mitad de un periodo para casi todas las operaciones necesaria y con la otra mitad del ciclo se fija el valor final a la salida de XOR3. No obstante, el registro, consume un ciclo de reloj para escribir el dato, por lo que son necesarios dos ciclos de reloj para completar uno de los 16 estados de la permutación de cajas S y permutación P.

5.4.4 Diagramas de forma de onda.

En la Figura 5.18, 5.19, 5.20 y 5.21 se muestran los diagramas de forma de onda obtenidos para el circuito FUNCDES, en cada uno de sus cuatro estados. La calendarización de la ejecución fue optimizada para proporcionar el tiempo de cómputo más corto. El código VHDL entero ha sido simulado extensivamente. La descripción general de las señales que se presenta a continuación y siguen el orden en el que van apareciendo en los diagramas.

- RLJ. Es una señal de entrada y corresponde al reloj utilizado para sincronizar a los elementos sincronos del circuito.
- INICIO1, INICIO2. Son las señales externas de entrada que permiten controlar el estado del circuito mediante un código de operación formado por las cuatro combinaciones descritas en la Tabla 5.10.
- DIR. Es un dato de entrada que proporciona la dirección de la localidad de la memoria en donde se encuentra el dato de 64 bits a encriptar. Esta misma dirección junto con un offset de +3 establece la localidad en donde se almacenará el dato encriptado.
- HCH_ME es una señal interna, es decir que funciona dentro del circuito, y que permite habilitar al chip de la memoria externa.
- RI_ME. Es una señal interna que reinicia los valores de la memoria externa.

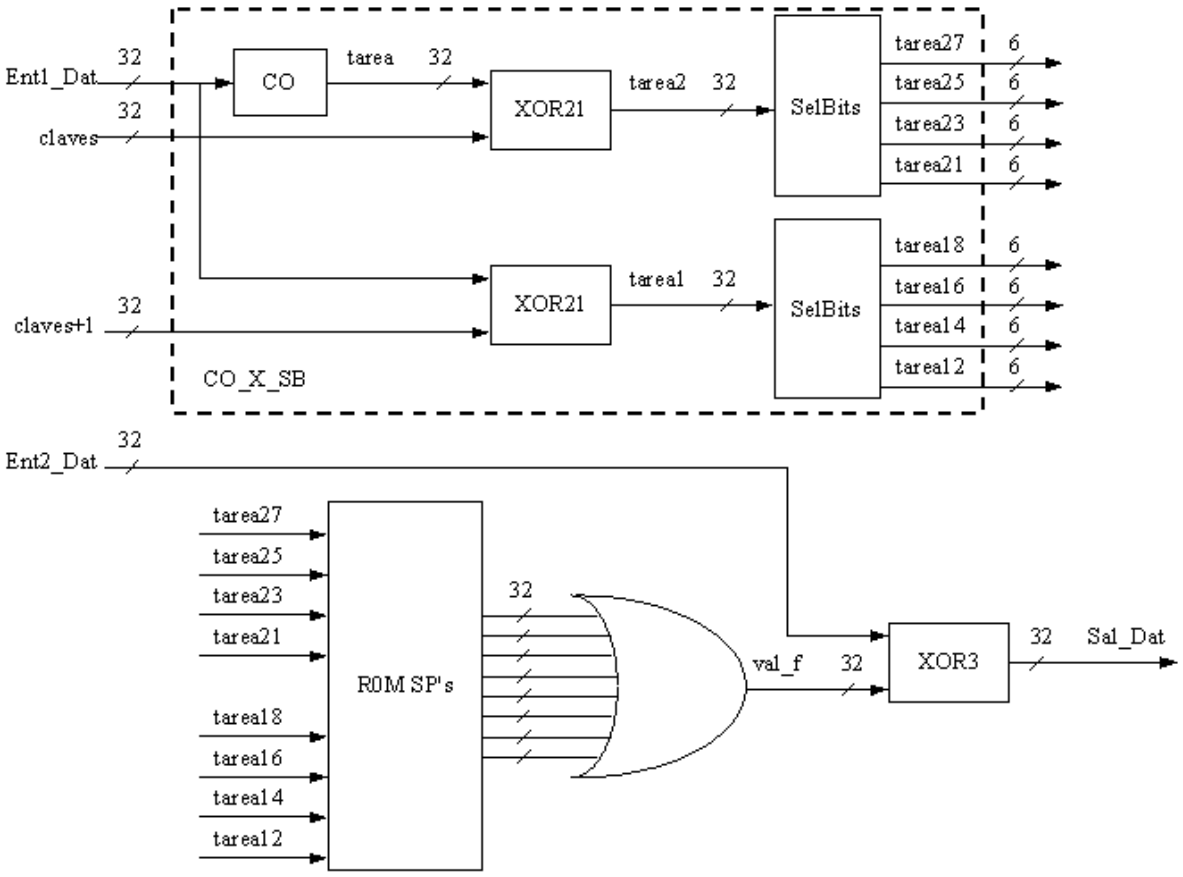

Figura 5.17: Circuito CO_X_CDAAMO_X que corresponde a la permutación de cajas S y Permutación P.

- HLE_ME. Es una señal interna que permite activar la Lectura/Escritura de la memoria externa.
- DSAL_ME. Es un bus interno que proporciona el valor de 64 bits de la localidad de memoria direccionada.
- DENT_ME. Es un bus interno por medio del cual se envían los datos finales después de la operación de encriptado, para que estos sean almacenados en una localidad de la memoria externa.
- HABEREG. Es una señal interna que habilita la operación de escritura del registro. Funciona únicamente en el estado de direccionar y cargar, después permanece en alta impedancia. Esta señal es producida por la unidad de control de operaciones.
- HPI. Es una señal interna que se encarga de habilitar al circuito de permutación inicial.
- ENT1DAT y ENT2DAT. Es un bus interno que maneja los datos de entrada del registro de propósito general.

- HabFD. Es una señal interna que permite habilitar el funcionamiento del circuito decodificador de hexadecimal a siete segmentos.
- SAL1R y SAL2R. Corresponde a una señal interna que proporciona los valores de las salidas del registro, esto es, después de que el valor se ha almacenado en el registro.
- INICIO. Es una señal de salida que indica que el circuito de encriptado esta en operación.
- HabLEReg. Es una señal interna que convive con la señal HABEREG, solo que esta es generada por la unidad de control de encriptado y actúa en dos estados del sistema: Activar y Ver. Cuando esta señal se encuentra en alta impedancia la anterior esta activa y viceversa. Habilita la lectura y escritura del registro de propósito general.
- DIRCLAVE. Es una señal interna que proporciona la dirección de una localidad de la ROM de claves.
- Cont7M. Es una señal interna que contienen los valores de un contador modulo 7 y que básicamente cuenta el numero de flancos, tanto de subida como de bajada.
- CLVUNO y CLVDOS. Son dos buses que proporciona los datos de dos localidades contiguas de la ROM de claves direccionada mediante DIRCLAVE. Se proporcionan dos valores de claves a la vez para incrementar el paralelismo.
- FinIter. Esta señal interna se encarga de llevar el conteo de las iteraciones y es la que detiene el funcionamiento del circuito de encriptado una vez que se han completado las 16 iteraciones.
- HPF. Es una señal interna que se encarga de habilitar al circuito de permutación final.
- S17S y S27S. Son buses de salida que llevan la información de salida ya decodificada a 7 segmentos hacia los display del mismo nombre. . Para visualizar los contenidos en alguna localidad de la memoria RAM.
- SIGBYTE. Es una señal de entrada que permite pasar al siguiente byte que componen al dato de entrada de 64 bits proveniente de la memoria RAM externa. Con un byte se pueden ver dos datos distintos a la vez.

En la Figura 5.18 se presenta un diagrama de forma de onda en el que se pueden ver los dos primeros estado por los que transita el circuito, el estado Direccionar y el estado Cargar. Cuando se esta en el estado Direccionar, las salidas de los circuitos se encuentran en alta impedancia. Un ciclo después de pasar al estado cargar aparecen a la salida de la memoria RAM los datos, enseguida se activa la permutación inicial y los datos obtenidos por esta se reflejan en la entrada del registro. Medio ciclo después, los datos aparecen a la salida del registro, indicando que estos ya han sido almacenados, manteniéndose ahí, incluso, a pesar de que el dato de entrada del registro desaparece.

La Figura 5.19 muestra el diagrama de forma de onda del circuito cuando se encuentra al principio del estado Activar. Se puede notar que los circuito comienzan a trabajar cada

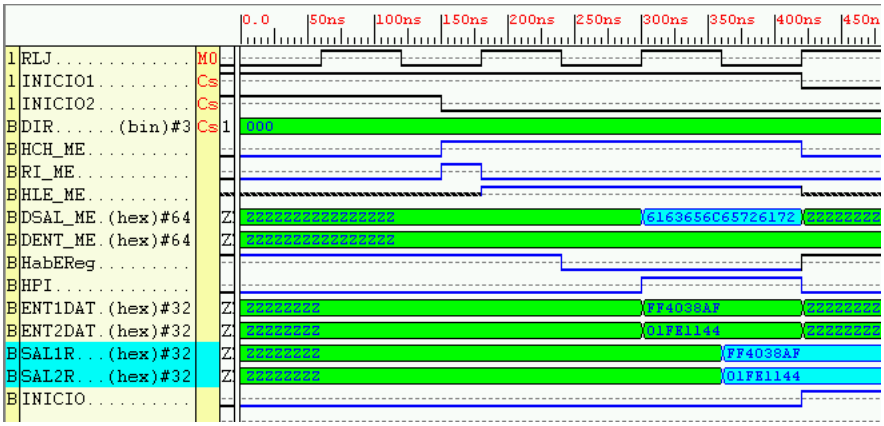

Figura 5.18: Diagrama de forma de onda en el que se muestran los estados direccionar (Inicio1=1 e Inicio2=1) y el estado cargar (Inicio1=1 e Inicio2=0).

uno haciendo su parte para completar el encriptado de los datos. Los datos a la salida del registro son procesados junto con los valores de las claves, en el estado 0 y en el estado 4 del contador. Medio ciclo después aparecen a la entrada del registro los nuevos datos. El valor se mantiene fijo medio ciclo y un ciclo completo es requerido para guardar estos nuevos datos en el registro. Lo anterior se repite durante 16 veces. Por su parte la figura 5.20 presenta el diagrama de forma de onda al final del estado activar. Aquí se puede ver como la señal FinIter se activa, deteniendo a todos los circuitos. El último valor guardado en el registro se conserva a la salida de este, inmediatamente después se activa el circuito de permutación final y el resultado de esta permutación se refleja a la entrada de la memoria externa para ser guardada en la localidad actual +3.

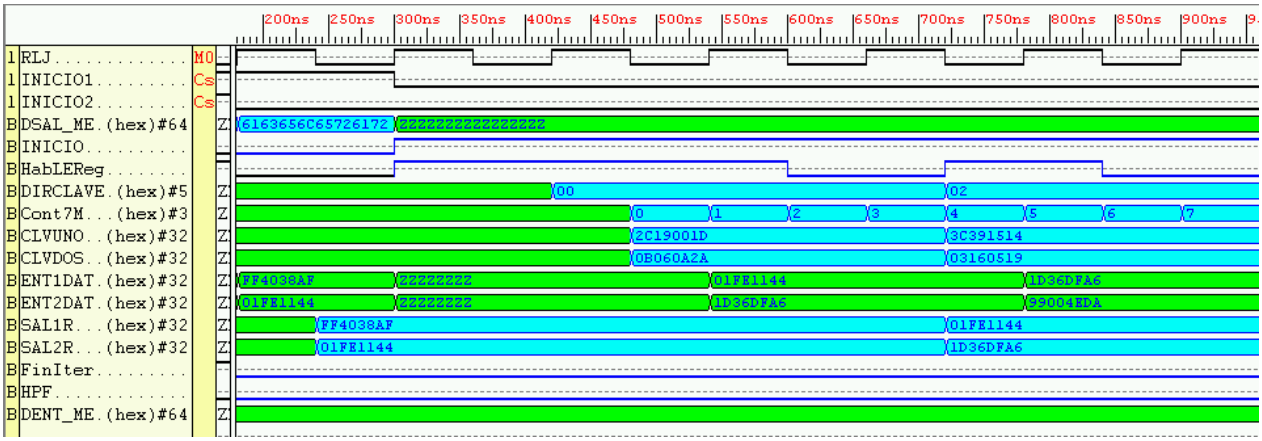

Figura 5.19: Diagrama de forma de onda que muestra las primeras operaciones del estado activar (Inicio1=0 e Inicio2=0).

Por último la Figura 5.21 muestra el diagrama de forma de onda para el estado Ver. Para poder ver otra localidad de memoria diferente a la actual, entonces será necesario poner al circuito en el estado Direccionar y establecer la dirección de memoria que se quiere visualizar, después hay que poner al circuito en el estado Ver. Recuerde que el dato encriptado se guarda

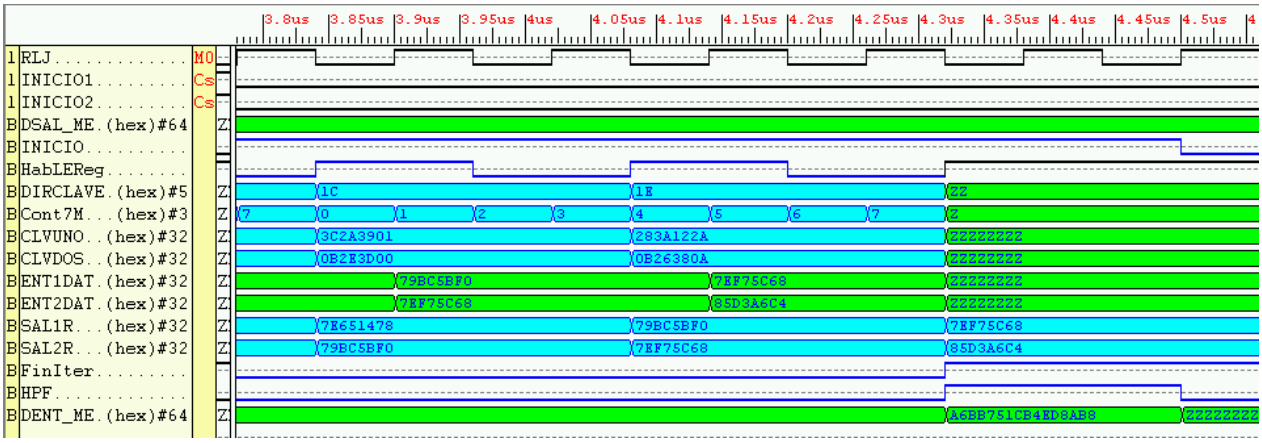

Figura 5.20: Diagrama de forma de onda que muestra las últimas operaciones del estado activar (Inicio1=0 e Inicio2=0).

en la dirección de la localidad actual mas un offset de 3. En las señales S17S y S27S los datos de salida ya aparecen decodificados a 7 segmentos. La señal SigByte, permite pasar al siguiente par de 4 bits del dato de 64 bits.

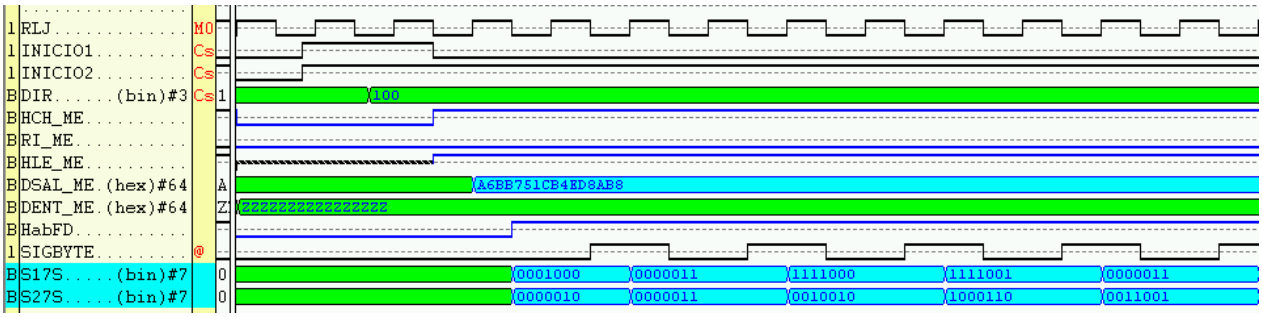

Figura 5.21: Diagrama de forma de onda que muestra el estado direccionar (inicio1=1 e Inicio2=1) y el estado Ver (Inicio1=0 e Inici2=1) del circuito FUNCDES.

5.4.5 Recursos utilizados.

La Tabla 5.5 presenta un reporte de los recursos utilizados por la implementanción del circuito en el dispositivo Virtex300.

El número de de recursos utilizados demuestran que un FPGA VIRTEX300 es ciertamente suficiente para implementar el circuito, ya que el más alto porcentaje de utilización fue en promedio del 25%, ésto reafirma que la complejidad de la partición desarrollada en hardware no es alta.

Componentes del FPGA	Número usado	Contenido en el FPGA
Slices	807	3,072
Slices sin contener relaciones lógicas	0	807
Slice Flip Flops	863	6,144
LUTs de 4 entradas	1,363	6,144
IOBs asegurados	21	162
Buffers de Tercer Estado (Tbufs)	696	3,200
Relojes globales (GCLKs)	1	4
IOBs de Relojes globales (GCLKIOBs)	1	4
Equivalentes a compuertas totales contadas para el diseño:	18,181	–
JTAG adicionales a la cuenta de compuertas para IOBs	1,056	–

Tabla 5.5: Resumen de recursos utilizados en la implementación del circuito `funcdes()`

5.4.6 Desempeño de ejecución y rendimiento de procesamiento.

El período mínimo reportado por la herramienta Trace de Xilinx fue de 104.130ns lo que significa que el circuito opera a una frecuencia máxima de 9.603MHz. Es importante recordar que el circuito ocupa aproximadamente 32 ciclos de reloj en el estado Activar, para completar sus 16 iteraciones, por lo que el tiempo total, necesario para encriptar un bloque de 64 bits, sería de 3.332microseg. El desempeño obtenido sería por lo tanto de 300,105 operaciones FUNCDES por segundo. Considerando que los bloques de entrada y salida son de 64 bits entonces se tendría un rendimiento de procesamiento total, "throughput", de 19.2 Mbits/seg, equivalente a 2.4 Mbyte/seg.

Para poder establecer una referencia de desempeño del hardware contra su similar en software, la Tabla 5.6 presenta el desempeño de ejecución del FPGA de la familia Virtex300 con un reloj a razón de 9.6 MHz, comparada con la ejecución del software en tres procesadores de la misma familia pero con velocidades diferentes. La versión del compilador utilizado fue gcc-2.96. En esta tabla se comparan los tiempos de ejecución para una operación de cifrado con un bloque de 64 bits. Los resultados obtenidos varían debido a que los procesadores presentan tecnologías y recursos diferentes.

Es posible mejorar aún más el desempeño de la implantación en hardware: utilizando un diseño pipeline o una implementación de unidades de cifrado en hardware trabajando paralelamente. Un diseño pipeline de 16 estados podría procesar una de las 16 etapas que

Hardware	Software		
	Pentium I (sw/hw)	Pentium II (sw/hw)	Pentium III (sw/hw)
3.332×10^{-6}seg.	17.45×10^{-6}seg. (5.23)	6.23×10^{-6}seg. (1.86)	2.07×10^{-6}seg. (0.62)

Tabla 5.6: Comparaciones de tiempos de ejecución para un bloque de 64 bits. El valor entre paréntesis (sw/hw) establece la relación existente entre el software y el hardware.

contiene una operación FUNCDES en un ciclo de reloj de un período determinado. Cada estado ocuparía 1 ciclo de reloj y se podría eliminar el ciclo de reloj necesario para escribir, el dato obtenido, al registro de propósito general, enviándose directamente al siguiente estado del pipeline. De este modo, se alcanzaría un rendimiento de procesamiento de 32 veces más que el actual, con la desventaja de la sincronización de secciones criticas de cada etapa y la cantidad de área ocupada en el dispotivo sería mayor. Si tomamos como base la mayor cantidad de elementos utilizados en el dispositivo (Slices) ver Tabla 5.5 y lo multiplicamos por 16 que serían los estados del pipeline y además pensando en que cada estado contiene todas y cada una de las unidades necesarias para realizar el trabajo de una sola etapa, la cantidad de recursos equivalente para este diseño sería de aproximadamente el de cuatro dispositivos Virtex300. Mientras que, un diseño de n unidades FUNCDES en hardware, procesando paralelamente un bloque diferente cada una, podría alcanzar un rendimiento de procesamiento de n veces más que el actual y con la ventaja de evitar una unidad de control para sincronizar los datos intermedios, pero con la desventaja, en este caso, de ocupar un área de n veces la cantidad de recursos utilizados hasta ahora en el dispositivo FPGA para su implementación.

Una arquitectura que cifra un solo bloque de 8 bytes a la vez, como la presentada hasta ahora, no es propiamente una implementación en hardware-software en donde se pretende sustituir en hardware la parte más complicada de un algoritmo en software y utilizarse ésta un gran número de veces debido al hecho de cifrar considerables cantidades de datos. Sin embargo, si se piensa como una unidad básica conectada al microprocesador, en el que éste le envía los datos a procesar y el hardware los retorna ya procesados, entonces podremos llegar a un sistema hardware-software. El diseño del cifrador (el hardware), se podría conectar con el microprocesador (el software) a través de diferentes esquemas: (1) usando el bus de accesos a memoria, en el que el cifrador estaría dentro del módulo de memoria, (2) a través de un bus PCI, en el que el cifrador estaría en una tarjeta de expansión o (3) en el mejor caso, alojada en una pequeña sección del microprocesador como una unidad funcional más.

Tomando el esquema basado en una memoria RAM, en el que se intercambian datos entre harwdare y software a través del bus de acceso a memoria, el troughput de este sistema se vería influenciado por el tiempo de acceso a la memoria, es decir los tiempos de lectura y escritura, como puede verse en la Figura 5.22. Considerando una memoria RAM estática (SRAM) con un tiempo de acceso aproximado de 10ns para una operación de lectura y 10ns para otra operación de escritura [NEC03], así como 3.332 μs de tiempo de cifrado, el throughput

del sistema sería de 19.093 Mbits/seg equivalente a 2.386 Mbytes/seg. Mientras que utilizar una memoria RAM dinámica (DRAM) con tiempos de acceso de 50ns para escritura y 50ns para lectura [ELP01], más 3.32 μs de tiempo de cifrado, se tendría un throughput de 18.648 Mbits/seg equivalente a 2.331 Mbytes/seg.

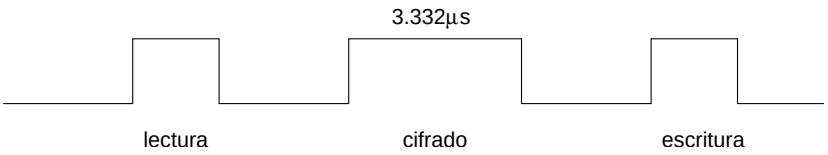

Figura 5.22: Ciclos de reloj necesarios para una operacion de cifrado en un esquema basado en memoria RAM.

Esta claro que hay más trabajo por hacer y que se pueden seguir varias líneas de investigación como la optimización de los circuitos VHDL, la interacción hardware-software, la automatización de la partición en hardware y en software, y otras más. Sin embargo, el primer acercamiento a una metodología formal para dividir un algoritmo criptográfico en componentes hardware y software se ha dado en este trabajo.

Conclusiones

En este trabajo de investigación se presentó una metodología para implementar el algoritmo DES en hardware-software. Partiendo de una versión puramente en software y mediante un análisis desde un nivel global hasta un nivel de instrucción del algoritmo, esta metodología permitió detectar la sección más costosa del algoritmo. Una vez encontrada dicha sección, se propusieron los circuitos digitales apropiados para realizar las tareas respectivas. La parte restante, es decir aquella que no es costosa, no tuvo cambio alguno en software, por lo que se forma un sistema hardware-software.

Esta propuesta contempla la realización de varios experimentos. Una primera prueba de rendimiento, permitió comparar el costo de encriptamiento con el costo de una operación de comunicaciones, observándose que es considerable el costo de encriptar los datos comparada con el costo de la comunicación, por lo que resultaba necesario mejorar el rendimiento en el procesamiento del algoritmo de encriptado.

Utilizando un análisis de perfiles de rendimiento, se pudo establecer la sección en particular del algoritmo que consumía la mayor cantidad de tiempo de ejecución. Después de varios experimentos realizados, se estableció que la función que realiza el encriptamiento era la sección crítica del algoritmo.

Un análisis más detallado a nivel de secuencias de instrucciones, permitió localizar y jerarquizar con más exactitud los componentes del algoritmo que consumen el total del tiempo de ejecución de esta función. Esta clasificación es importante, porque permite determinar cuál de los componentes tiene mayor orden de importancia para implementarse en hardware. De este modo, la sección que consume el mayor tiempo de ejecución tendrá prioridad para implantarse en hardware, mientras que la sección que consume el menor tiempo de ejecución tal vez pueda no implantarse.

Continuando con el procedimiento, se encontraron siete patrones con diferentes grados de repetición dentro de la función de encriptado. Este procedimiento a la vez que permitió localizar las unidades básicas de operación, también permitió la migración de cada secuencia de operaciones en un circuitos digitales de manera gradual y fácil.

Los diseños propuestos son circuitos relativamente simples y presentan un grado de complejidad muy bajo, lo cual proporciona una ventaja al momento de implementarlo en un FPGA ya que el factor área o densidad no se ve comprometida con diseños como éstos. Por

si fuera poco, un circuito simple ejecuta su tarea más rápidamente. Las implementaciones de este tipo son factibles, ya que se puede acelerar un algoritmo en software mediante la utilización de circuitos sencillos en hardware.

En el diseño de los circuitos digitales se tomaron consideraciones, algunas generales y otras para mejorar el rendimiento de operación de los mismos, dichas consideraciones se enlistan a continuación:

- La relación tiempo-espacio, es un parámetro fundamental que debe estar siempre presente en un diseño de un sistema hardware-software. Algunas veces, cuando se intenta reducir el tiempo de ejecución de un circuito se puede provocar el aumento del área de diseño y por el contrario intentar reducir el área del diseño puede desencadenar un retardo excesivo en el tiempo de operación del circuito.

- Es importante el uso de elementos disjuntos y complementarios en un diseño, porque evita una avalancha de cambios en la arquitectura original cuando se realiza un pequeño cambio en algún componente.

- Especializar los diseño para condiciones estrictamente particulares mejora la rapidez con que se ejecuta el algoritmo.

- El uso de los dos flancos de reloj, tanto el de subida como el de bajada, pudo reducir el número de ciclos necesarios para completar la operación de las funciones del circuito.

- Tener siempre en mente crear diseños que exploten el paralelismo inherente en hardware, lo que permite optimizar desempeños de operación en aquellos lugares donde sea posible, como por ejemplo, donde no existan dependencias de datos.

- Buscar la menor cantidad de símbolos a utilizar en cada diseño, considerando que una gran base de símbolos proporciona incrementos en el uso total de las unidades de procesamiento (throughput of proccessing).

En este reporte, se abordó la funcionalidad de las herramientas de especificación, verificación y síntesis de alto nivel, utilizadas en el diseño actual de los circuitos digitales. Se puntualizó sobre todo en el uso del lenguaje VHDL para modelar hardware, como medio para conseguir circuitos lógicos en muy poco tiempo y de manera casi automática a partir de un lenguaje imperativo como C, además de la ventaja de ser portable a diferencia de los esquemáticos que son dependientes de la tecnología. A pesar de que este lenguaje permite describir el funcionamiento del circuito, debe cuidarse en no excederse el alto nivel con que se hace, porque se puede llegar a no describir lo suficiente al circuito, provocando ambigüedades, operaciones incorrectas, o que se generen una mayor cantidad de elementos lógicos al momento de sintetizar el diseño.

El uso de los FPGA's llegará a ser la forma dominante de diseño e implementación de lógica digital, esto conforme mejore la arquitectura y las herramientas CAD. Su facilidad de acceso, principalmente a través del bajo costo de los dispositivos, sus crecientes prestaciones en términos de capacidades y de velocidad, la inherente flexibilidad que ofrecen y la rápida

"fabricación de vuelta" que proporcionan son elementos esenciales de éxito en el mercado.

Los resultados parciales mostraron que el 57.4% del código del algoritmo en software de DES fue susceptible de implementarse en hardware y en un momento dado mejorar su rendimiento. Se realizaron dos implementaciones en hardware. En la primera se utilizó un FPGA XC4013 de Xilinx y se implementaron por separado cada uno de los circuitos que componen la función de cifrado, después se hizó un estudio del aporte de desempeño que produce cada uno de ellos. En la segunda se utilizó un FPGA Virtex XCV300 de Xilinx y se implementó toda la sección crítica, la cual incluye a cada uno de los circuitos anteriores. El desempeño, "throughput", conseguido es de 2.4 Mbyte/seg. El comportamiento del hardware con respecto al software, no es completamente sorprendente. Este proyecto ha sido solo el primer acercamiento de diseño y síntesis en VHDL para una aplicación de una máquina de cómputo a la medida, basada en un FPGA; pero el software fue creado basándose en numerosos años de experiencia en programación. Esto permite que el software este mucho más eficientemente construido que la realización en hardware. Dado que se trata de un diseño en VHDL, las herramientas de síntesis no generan circuitos optimizados por tanto, la frecuencia de operación del circuito no es muy alta. Haciéndo optimizaciones en el circuito sintetizado y utilizando algunos otros mecanismos propiso de la familia Virtex es posible incrementar la frecuencia de operación. Así también, es importante considerar que al comparar con procesadores, éstos están construidos en circuitos VLSI y sus relojes de operación por lo general son varias veces más grande de loq ue se puede obtener con un FPGA.

Una metodología como la que se propone en esta investigación permite tomar ventaja al explotar las características inherentes del algoritmo para potencialmente incrementar su eficiencia. Mediante esta metodología se puede establecer que un sistema hardware-software es una alternativa que podría conseguir mejorar el desempeño de algoritmos que presentan entre otras cosas: manejo de datos y operaciones lógicas a nivel de bits y un alto número de repeticiones, tal es el caso de algunos algoritmos criptográficos, como el que fué utilizado en este estudio.

Apéndice A

Códigos fuente.

A.1 Seleccionador de Bits

```
-----------------------------------------------------
-- Nombre:      sb.vhd
-- Descripcion: Seleccionador de bits de 32 a 6.
-- Notas:       Toma los 6 bits menos significativos, despues de haber
--              recorrido 0, 8, 16 y 24 posiciones a la derecha
--              a un vector de entrada de 32 bits.
--              Utiliza asignaciones de bitslice.
--              Se obtiene a la salida un vector de
--              solo 6 bits.
-----------------------------------------------------
-- Version: 1.0
-- Creado: 18.10.2001
-- Modificado: 17.10.2002
-----------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity sb is
   port(
      EntSb: in STD_LOGIC_VECTOR (31 downto 0);
      BCORR: in STD_LOGIC_VECTOR (4 downto 0);
      RLJ:   in STD_LOGIC;
      HabSb: in STD_LOGIC;
      SalSb: out STD_LOGIC_VECTOR (5 downto 0)
   );
end sb;

architecture sb_arq of sb is
signal SalTemp: STD_LOGIC_VECTOR (5 downto 0);
begin

   SEL_BITS: PROCESS(rlj)
   begin
      IF (rlj'event AND rlj = '1') THEN
         case BCORR is
            when "01000" =>        -- 8 posiciones
               SalTemp <= EntSb(13 downto 8);

            when "10000" =>        -- 16 posiciones
               SalTemp <= EntSb(21 downto 16);

            when "11000" =>        -- 24 posiciones
               SalTemp <= EntSb(29 downto 24);
```

```
            when others =>          -- Otra posicion
               SalTemp <= EntSb(5 downto 0);
         end case;
      END IF;
   END PROCESS;


   HABILITAR: PROCESS (HabSb, SalTemp)
   begin
      if (HabSb='0') then
         SalSb <= (others => 'Z');
      else
         SalSb <= SalTemp;
      end if;
   END PROCESS;

end sb_arq;
```

A.2 Compuerta OR compuesta

```
-------------------------------------------------------------
-- Nombre:      CompOr.vhd
-- Descripcion: Compuerta OR compuesta.
-- Notas:       Cuando llega el flanco de subida del reloj se realiza
--              una operacion OR entre los dos vectores.
--              Este valor es registrado y almacenado.
--              La senhal HabOR habilita la salida del dispositivo
-------------------------------------------------------------
-- Version: 1.0
-- Creado: 28.01.2002
-- Modificado: 28.01.2002
-------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity CompOR is
    port (
        Ent1OR: in STD_LOGIC_VECTOR (31 downto 0);
        Ent2OR: in STD_LOGIC_VECTOR (31 downto 0);
        RLJ:    in STD_LOGIC;
        HabOR:  in STD_LOGIC;
        SalOR:  out STD_LOGIC_VECTOR (31 downto 0)
    );
end CompOR;

architecture CompOR_arch of CompOR is
signal SalTemp: std_logic_vector(31 downto 0);
begin

  OPERACION: PROCESS(RLJ)
  begin
    IF (RLJ ='0' AND RLJ'event) THEN
       SalTemp <= Ent1OR OR Ent2OR;
    END IF;
  END PROCESS;

  HABILITAR: PROCESS(HabOR, SalTemp)
  begin
    IF (HabOR = '0') then
       SalOR <= (others => 'Z');
    ELSE
       SalOR <= SalTemp;
  END IF;
  END PROCESS;

end CompOR_arch;
```

A.3 Compuerta AND compuesta

```
-------------------------------------------------------------
-- Nombre:      Compand.vhd
-- Descripcion: Compuerta AND compuesta.
-- Notas:       Cuando llega el flanco de subida del se realiza
--              una operacion AND entre los dos vectores.
--              Este valor es registrado y almacenado.
--              La senhal HabOR habilita la salida del dispositivo
-------------------------------------------------------------
-- Version: 1.0
-- Creado: 28.01.2002
-- Modificado: 06.03.2002
-------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity CompAND is
    port (
        Ent1AND: in STD_LOGIC_VECTOR (31 downto 0);
        Ent2AND: in STD_LOGIC_VECTOR (31 downto 0);
        RLJ:     in STD_LOGIC;
        HabAND:  in STD_LOGIC;
        SalAND:  out STD_LOGIC_VECTOR (31 downto 0)
    );
end CompAND;

architecture CompAND_arch of CompAND is
signal SalTemp: std_logic_vector(31 downto 0);
begin

   OPERACION: PROCESS(RLJ)
   begin
      IF (RLJ ='0' AND RLJ'event) THEN
         SalTemp <= Ent1AND AND Ent2AND;
      END IF;
   END PROCESS;

   HABILITAR: PROCESS(HabAND, SalTemp)
   begin
      IF (HabAND = '0') then
         SalAND <= (others => 'Z');
      ELSE
         SalAND <= SalTemp;
      END IF;
   END PROCESS;

end CompAND_arch;
```

A.4 Compuerta XOR compuesta

```
-------------------------------------------------------------
-- Nombre:      CompXor.vhd
-- Descripcion: Compuerta XOR compuesta.
-- Notas:       Cuando llega el flanco de subida del reloj se realiza
--              una operacion OR eXclusiva entre los dos vectores.
--              Este valor es registrado y almacenado.
--              La senhal HabOR habilita la salida real del dispositivo
-------------------------------------------------------------
```

```
-- Version: 1.0
-- Creado: 28.01.2002
-- Modificado: 20.03.2002
----------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity CompXOR is
    port (
        Ent1XOR: in STD_LOGIC_VECTOR (31 downto 0);
        Ent2XOR: in STD_LOGIC_VECTOR (31 downto 0);
        RLJ:    in STD_LOGIC;
        HabXOR:  in STD_LOGIC;
        SalXOR:  out STD_LOGIC_VECTOR (31 downto 0)
    );
end CompXOR;

architecture CompXOR_arch of CompXOR is
signal SalTemp: std_logic_vector(31 downto 0);
begin

  OPERACION: PROCESS(RLJ)
  begin
    IF (RLJ ='1' AND RLJ'event) THEN
       SalTemp <= Ent1XOR XOR Ent2XOR;
    END IF;
  END PROCESS;

  HABILITAR: PROCESS(HabXOR)
  begin
    IF (HabXOR = '0') then
       SalXOR <= (others => 'Z');
    ELSE
       SalXOR <= SalTemp;
    END IF;
  END PROCESS;

end CompXOR_arch;
```

A.5 Registro de corrimiento a la derecha

```
-------------------------------------------------------
-- Nombre:      rcdl.vhd
-- Descripcion: Registro de corrimiento a la derecha.
-- Notas:       Recorre 31 pos. a la der. cuando BCORRD es 31,
--              4 pos. a la der, cuando BCORRD es 4
--              o 1 pos. a la der cuando BCORRD es 1,
--              Vector de entrada/salida de 32 bits.
-------------------------------------------------------
-- Version: 1.0
-- Creado: 04.03.2002
-- Modificado: 07.03.2002
-------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity rcdl is
    port (
        EntRCD: in STD_LOGIC_VECTOR (31 downto 0);
        BCORRD: in STD_LOGIC_VECTOR (4 downto 0);
        RLJ:    in STD_LOGIC;
        HabRCD: in STD_LOGIC;
        SalRCD: out STD_LOGIC_VECTOR (31 downto 0)
    );
end rcdl;
```

```
architecture rcdl_arch of rcdl is
signal CorrimDer: STD_LOGIC_VECTOR (31 downto 0);
begin
   CORRIMIENTO: PROCESS(rlj)
   begin
      IF (rlj'event AND rlj = '1') THEN
         case BCORRD is

            when "00001" =>   --Corrim a la der 1 pos
               --Efecto de corrimiento a la derecha (1 pos.)
               for I in 0 to 31 loop
                  if( I < 31 ) then
                     CorrimDer(I) <= EntRCD(I+1);
                  else
                     CorrimDer(I) <='0';
                  end if;
               end loop;

            when "00100" =>   --Corrim a la der 4 pos
               --Efecto de corrimiento a la derecha (4 pos.)
               for I in 0 to 31 loop
                  if( (I+3) < 31 ) then
                     CorrimDer(I) <= EntRCD(I+4);
                  else
                     CorrimDer(I) <='0';
                  end if;
               end loop;

            when "11111" =>   --Corrim a la der 31 pos
               --Efecto de corrimiento a la derecha (1 pos.)
               for I in 0 to 31 loop
                  if( (I+30) < 31 ) then
                     CorrimDer(I) <= EntRCD(I+31);
                  else
                     CorrimDer(I) <='0';
                  end if;
               end loop;

            when others =>          -- Otra posicion
               CorrimDer <= EntRCD;

         end case;
      END IF;
   END PROCESS;

   HABILITAR: PROCESS (HabRCD, CorrimDer)
   begin
      if (HabRCD='0') then
         SalRCD <= (others => 'Z');
      else
         SalRCD <= CorrimDer;
      end if;
   END PROCESS;

end rcdl_arch;
```

A.6 Registro de corrimiento a la izquierda

```
-----------------------------------------------------
-- Nombre:      rcil.vhd
-- Descripcion: Registro de corrimiento a la izquierda (Usa un loop)
-- Notas:       Recorre 31 pos. a la izq cuando BCORRD es 1,
--              28 pos. a la izq, cuando BCORRD es 4,
--              o 1 pos. a la izq cuando BCORRD es 31
--              Vector de entrada/salida: 32 bits.
```

```
----------------------------------------------------
-- Version: 1.0
-- Creado: 04.03.2002
-- Modificado: 07.03.2002
----------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity rcil is
    port (
        EntRCI: in STD_LOGIC_VECTOR (31 downto 0);
        BCORRD: in STD_LOGIC_VECTOR (4 downto 0);
        RLJ:    in STD_LOGIC;
        HabRCI: in STD_LOGIC;
        SalRCI: out STD_LOGIC_VECTOR (31 downto 0)
    );
end rcil;

architecture rcil_arch of rcil is
signal SalTemp: STD_LOGIC_VECTOR (31 downto 0);
begin

   CORRIMIENTO: PROCESS(rlj)
   begin
      IF (rlj'event AND rlj = '1') THEN
         case BCORRD is

            when "00001" =>    -- Corrim a la der 1 pos
               --Efecto de corrimiento a la izquierda (31 pos.)
               for I in 0 to 31 loop
                  if (I < 31 ) then
                     SalTemp(I) <= '0';
                  else
                     SalTemp(I) <= EntRCI(I-31);
                  end if;
               end loop;

            when "00100" =>    --Corrim a la der 4 pos
               --Efecto de corrimiento a la izquierda (28 pos.)
               for I in 0 to 31 loop
                  if (I < 28 ) then
                     SalTemp(I) <= '0';
                  else
                     SalTemp(I) <= EntRCI(I-28);
                  end if;
               end loop;

            when "11111" =>    -- Corrim a la der 31 pos
               --Efecto de corrimiento a la izquierda (1 pos.)
               for I in 0 to 31 loop
                  if (I < 1 ) then
                     SalTemp(I) <= '0';
                  else
                     SalTemp(I) <= EntRCI(I-1);
                  end if;
               end loop;

            when others =>     -- Otra posicion
               SalTemp <= EntRCI;
         end case;
      END IF;
   END PROCESS;

   HABILITAR: PROCESS (HabRCI, SalTemp)
   begin
      if (HabRCI='0') then
         SalRCI <= (others => 'Z');
      else
```

```
        SalRCI <= SalTemp;
      end if;
   END PROCESS;

end rcil_arch;
```

A.7 Registro de corrimiento a la derecha con multiplexor

```
-----------------------------------------------------
-- Nombre:      rcdlm.vhd
-- Descripcion: Registro de corrimiento a la derecha.
-- Notas:       Recorre 2, 4, 8 y 16 posiciones a la derecha
--              a un vector de entrada de 32 bits. (usa un loop)
--              Vector de salida de 32 bits.
--              Incluye multiplexor del segundo operando de la
--              siguiente compuerta.
-----------------------------------------------------
-- Version: 1.0
-- Creado: 18.10.2001
-- Modificado: 06.03.2002
-----------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity rcdlM is
    port (
        EntRCD: in STD_LOGIC_VECTOR (31 downto 0);
        BCORR:  in STD_LOGIC_VECTOR (4 downto 0);
        RLJ:    in STD_LOGIC;
        HabRCD: in STD_LOGIC;
        SalRCD: out STD_LOGIC_VECTOR (31 downto 0);
        Operan: out STD_LOGIC_VECTOR (31 downto 0)
    );
end rcdlM;

architecture rcdlM_arch of rcdlM is
signal CorrimDer: STD_LOGIC_VECTOR (31 downto 0);
begin

   CORRIMIENTO: PROCESS(rlj)
   begin
      IF (rlj'event AND rlj = '1') THEN
         case BCORR is
            when "00010" =>        -- 2 posiciones
               for I in 0 to 31 loop
                  if( (I+1) < 31 ) then
                     CorrimDer(I) <= EntRCD(I+2);
                  else
                     CorrimDer(I) <='0';
                  end if;
               end loop;
               Operan <= X"33333333";

            when "00100" =>        -- 4 posiciones
               for I in 0 to 31 loop
                  if( (I+3) < 31 ) then
                     CorrimDer(I) <= EntRCD(I+4);
                  else
                     CorrimDer(I) <='0';
                  end if;
               end loop;
               Operan <= X"0F0F0F0F";
```

```
            when "01000" =>        -- 8 posiciones
               for I in 0 to 31 loop
                  if( (I+7) < 31 ) then
                     CorrimDer(I) <= EntRCD(I+8);
                  else
                     CorrimDer(I) <='0';
                  end if;
               end loop;
               Operan <= X"00FF00FF";

            when "10000" =>        -- 16 posiciones
               for I in 0 to 31 loop
                  if( (I+15) < 31 ) then
                     CorrimDer(I) <= EntRCD(I+16);
                  else
                     CorrimDer(I) <='0';
                  end if;
               end loop;
               Operan <= X"0000FFFF";

            when "00000" =>        -- 0 posiciones
               CorrimDer <= EntRCD;
               Operan <= X"AAAAAAAA";

            when others =>          -- Otra posicion
               CorrimDer <= EntRCD;
               Operan <= X"FFFFFFFF";
            end case;
      END IF;
   END PROCESS;


   HABILITAR: PROCESS (HabRCD, CorrimDer)
   begin
      if (HabRCD='0') then
         SalRCD <= (others => 'Z');
      else
         SalRCD <= CorrimDer;
      end if;
   END PROCESS;

end rcdlM_arch;
```

Bibliografía

[ACC94] O.T. Albaharna, P.Y.K. Cheung, and T.J. Clarke. Area & time limitations of fpga-based virtual hardware. In *Proceeding of the IEEE International Conference on Computer Design*, pages 184–189. IEEE, October 1994.

[ACC+96] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, G. Snider, and L. Albertson. An fpga for million gate systems. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterrey, CA, February 1996.

[Alt93] Altera Corporation. *Data Book*, August 1993.

[Bih96] Eli Biham. A fast new des implementation in software. In Springer Verlag, editor, *Lecture Notes in Computer Science*, volume 1267, pages 260–, Berlín, 1996.

[DG97] G. De Michelli and R. Gupta. Hardware/software co-design. In *Proceedings of the IEEE*, volume 85, pages 349–365, March 1997.

[DH79] W. Diffie and M.E. Hellman. Privacy and authentication: An introduction to criptography. In Proceeding of the IEEE, editor, *Tutorial: The security of data in networks, U.S.A. 1981*, volume 67, pages 397–427. IEEE, March 1979.

[DH93] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, pages 22(6):644–654, February 1993.

[DT91] Michael H. Dawson and Stafford E. Tavares. An expanded set of design criteria for substitution boxes and their use in strengthening DEs-like cryptosytems. *IEEE Pacific Rim Conference on Communications, Computers ans Signal Procesing*, pages 191–195, May 1991.

[DW99] André DeHon and John Wawrzynek. Reconfigurable computing: What, why, and design automation. In *Proceedings of the 1999 Design Automation Conference*, pages 610–615, June 1999.

[ELP01] ELPIDA Memory Inc., U.S.A. *Synchronous DRAM, User's Manual*, May 2001. Document No. E0124N10 (ver1.0).

[EPKD97] P. Eles, Z. Peng, K. Kuchinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Journal on Design Automation Embedded Systems, Kluwer Academic Publishers*, 2(1):5–32, January 1997.

[FBK98] J. Fleischmann, K. Buchenrieder, and R. Kress. A hardware/software prototyping environment for dynamically reconfigurable embedded systems. *6th. International Workshop on Hardware/Software Codesign, CODES/CASHE'98*, March 1998.

[FGHM98] Amparo Fuster, Dolores De La Guía, Luis Hernández, and Fausto Montoya. *Técnicas criptográficas de protección de datos.* Computer ra-ma, Alfaomega, España, 1st edition, 1998.

[FIP77] National Bureau of Standards, U.S.A. *Data Encription Standard*, January 1977.

[GHB93] J. Greene, E. Hamdy, and S. Beal. Antifuse field programmable gate arrays. *Proceeding of the IEEE*, 81(7):1042–1056, July 1993.

[GKM98] J. Grode, P. Knudsen, and J. Madsen. Hardware resource allocation for hardware/software partitioning in lycos system. In *Proceedings of the 1998 Design Automation and Test in Europe*, 1998.

[GPI] See General Processor Information, http://infopad.eecs.berkeley.edu/CIC/summary/local.

[Gra96] Paul S. Graham. A description, analysis and comparision of a hardware and a software implementation of the splash genetic algorithm for optimizing symmetric tsp. Master's thesis, Departament of Electrical and Computer Engineering. Briham Young University, October 1996.

[GVL99] Manuel E. Guzmán, Arturo Veloz, and José Leyva. Diseño de circuitos integrados de aplicación específica. *Avance y perspectiva*, 18:21–28, January 1999.

[Has97] Greg M. Haskins. Securing asynchronous transfer mode networks. Master's thesis, WPI, Worcester, Massachusetts, USA,, May 1997.

[Hem92] Ahmed Hemani. *High Level Synthesis of Synchronous Digital System using Self-Organization Algorithm for Scheduling and Allocation.* PhD thesis, Departament of Applied Electronics, Royal Institute of Technology, Stockholm, Sweden, 1992.

[Hen99] J. Henkel. A low power hardware/software partitioning approach for core-based embedded systems. In *Proceedings of the 36th. ACM/IEEE conference on Design automation conference*, pages 122–127, 1999.

[HiF99] Hi/fn, Inc. *HiFn 7751, Encryption Processor Data Sheet*, 1999.

[HL98] J. Henkel and Y. Li. Energy-conscious hw/sw-partitioning of embedded systems: A case study on an mpeg-2 encoder. In *Proceedings of the 6th. International on Hardware/Software Codesign*, pages 23–27, March 1998.

[HP90] Ahmed Hemani and Adam Postula. A neural net based self organizing scheduling algorithm. *European Design Automation Conference*, pages 136–140, 1990.

[JEOH94] Axel Jantsch, Peeter Ellervee, Johnny Öberg, and Ahmed Hemani. A case study on hardware/software partitioning. In *Proceedings IEEE Workshop on FPGA's for Custom Computing Machines*, pages 111–118, April 1994.

[JP98] Kaps Jens-Peter. *High Speed FPGA Architectures for the Data Encryption Standard.* PhD thesis, Worcester Polytechnic Institute., May 1998.

[KG93] Andreas Koch and Ulrich Golze. An fpga based coprocessor for sbus workstations. In Springer, editor, *Proceedings Lecture in Notes in Computer Science*, April 1993.

[KL97] A. Kalavade and E. A. Lee. The extended partitioning problem: Hardware/software mapping, scheduling and implementation-bin selection. *Journal on Design Automation Embedded Systems, Kluwer Academic Publishers*, 2(2):125–163, March 1997.

[Knu98] Jonathan Knudsen. *Java Cryptography.* O'Reilly & Associates, U.S.A., 1998.

[KP98] Rainer Kress and Andreas Pyttel. High-level synthesis for dynamically reconfigurable hardware/software systems. In Springer, editor, *Proceedings Lecture in Notes in Computer Science*, volume 1482, pages 288–297, August 1998.

[Kun88] S. Y. Kung. *VLSI Array Proccessors.* Prentice Hall, 1988.

[Luc02] Manuel J. Lucena, López. *Criptografía y seguridad en computadores.* Universidad de Jaen, España, 3rd edition, 2002.

[LvIW93] David M. Lewis, Marcus H. van Ierssel, and Daniel H. Wong. A field programmable accelerator for compiled code applications. In Springer, editor, *Proceedings Lecture in Notes in Computer Science*, April 1993.

[LWP94] Wayne Luk, Teddy Wu, and Ian Page. Hardware-software codesign of multidimensional programs. In *Proceeding of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 82–90, Napa VAlley California, April 1994. IEEE.

[MK00] M. Morris Mano and Charles R. Kime. *Logic and computer design fundamentals.* Prentice Hall, USA, 2nd edition, 2000.

[Mor88] Gasser Morrie. *Criptography*, pages 246–255. Van Nostrand Reinhold, New York, U.S.A., 1988.

[MWMD97] William Millan, Kim Wong, Wark Melanie, and Ed Dawson. A single-chip fpga implementation of a self-synchronous cipher. In *IEEE TENCON. Speech and image technologies for computing and telecomunications*, pages 223–226, 1997.

[NEC03] NEC Electronics, U.S.A. *Device DataSheet: Standard Synchronous SRAM.*, 2003. Ver también: http://www.nec.com.

[NOOS95] E.Ñahum, S. O'Malley, H. Orman, and R. Schroeppel. Towards high performance cryptographic software. In *Proceeding of the Third IEEE Workshop on the architectures and implementation of high performance communications subsystem*, HPC, Mystic, Conn, August 1995.

[NS93] National Semiconductor, U.S.A. *Configurable Logic Array (CLAy)*, 1993.

[PB00] Fernando Pardo and José A. Boluda. *VHDL. Lenguaje para síntesis y modelado de circuitos.* Alfaomega, ra-ma, México, 1ra edition, 2000.

[Sch96] Bruce Schneier. *Applied Cryptography. Protocols, algorithms and source code in c.* John Wiley & Son, U.S.A., 2nd edition, 1996.

[SGRF97] T. Schaffer, A. Glaser, S. Rao, and Paul Franzon. A flip-chip implementation of the des. In *Proceeding of the 1997 IEEE Multi-chip Module Conference MCM'97.* IEEE, 1997.

[Sim92] Gustavus J. Simmons. *Contemporary cryptology.* IEEE Press, Sandia National Laboratories, U.S.A., 1992.

[Smi01] Douglas J. Smith. *HDL, Chip Design.* Doone Publications, MAdison, AL, U.S.A., 1st. edition, July 2001.

[Sta99] William Stallings. *Cryptography and network security. Principles and practice.* Prentice Hall, U.S.A., 1999.

[T.I95] T.I.E.T.F.W.G. on security for ipv4. Technical report, IPsec draft, 1995.

[Tri93] S. Trimberger. A reprogrammable gate array and applications. In *Proceeding of the IEEE*, pages 1030–1041. IEEE, July 1993.

[VHD99] Xilinx, Inc, San Jose California 95124. *VHDL Reference Guide*, 1999.

[VLS99] VLSI Technology. *VLSI 115, Datasheet version 2.0*, January 1999.

[VSS+02] Jason Villarreal, Dinesh Suresh, Greg Stitt, Frank Vahid, and Walid Najjar. Improving software performance with configurable logic. *Journal on Design Automation Embedded Systems, Kluwer Academic Publishers*, 2002.

[Wie94] Michael Wiener. Efficient des key search. Technical Report TR-244, School of Computer Science, Carleton University, May 1994.

[XDB99] Xilinx, Inc, San Jose California 95124. *The Programmable Logic Data Book*, 1999. See "The Virtex Series of FPGAs": http://www.xilinx.com/products/virtex.