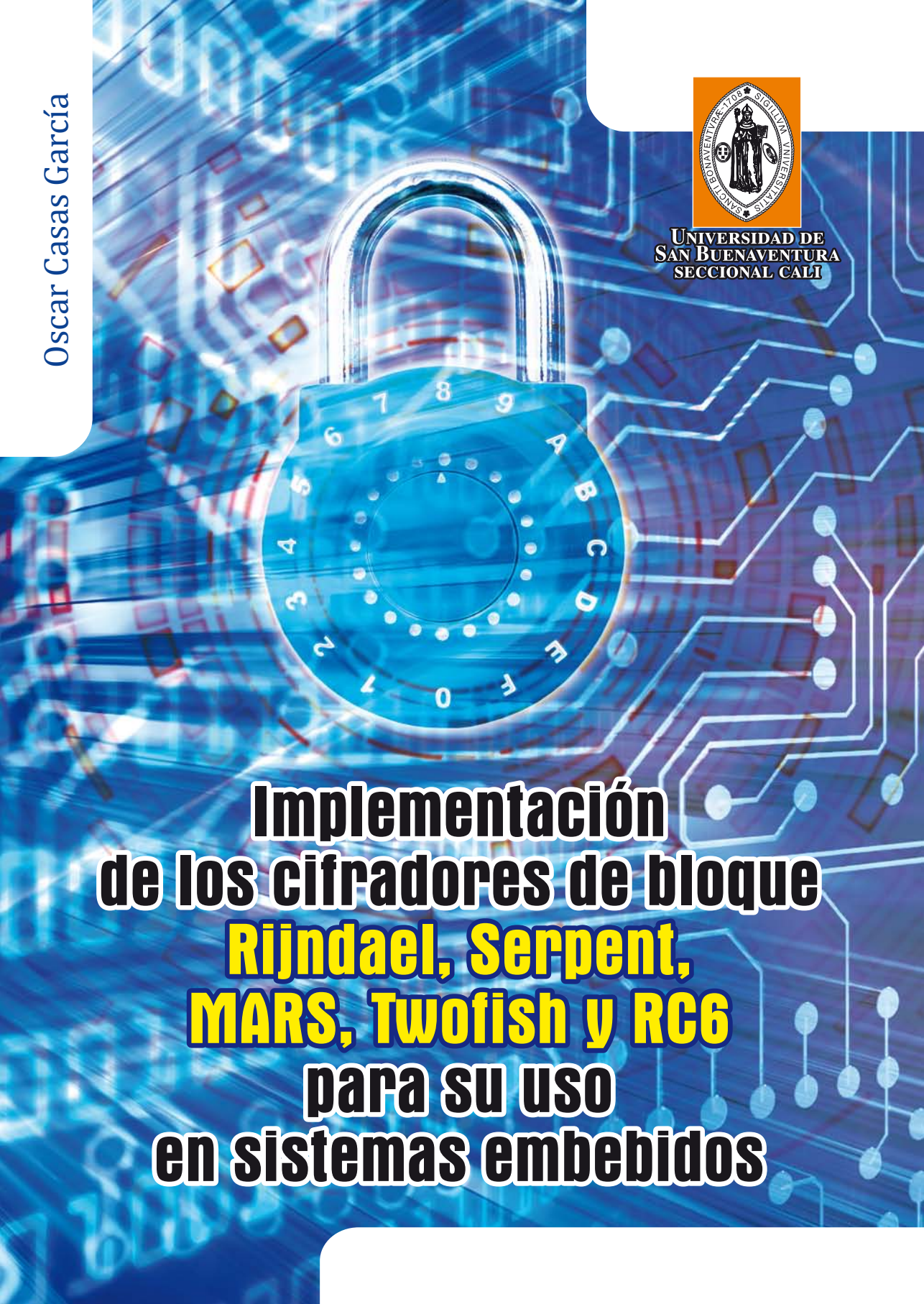
Oscar Casas García



Universidad de
San Buenaventura
seccional cali

Implementación de los cifradores de bloque Rijndael, Serpent, MARS, Twofish y RC6 para su uso en sistemas embebidos

Implementación de los cifradores de bloque *Rijndael*, *Serpent*, *MARS*, *Twofish* y *RC6* para su uso en sistemas embebidos

UNIVERSIDAD DE
SAN BUENAVENTURA
SECCIONAL CALI

Implementación de los cifradores de bloque *Rijndael*, *Serpent*, MARS, *Twofish* y RC6 para su uso en sistemas embebidos

Oscar Casas García

2010

Universidad de San Buenaventura, seccional Cali
Editorial Bonaventuriana

Título: ***Implementación de los cifradores de bloque Rijndael, Serpent, MARS, Twofish y RC6 para su uso en sistemas embebidos***

Autor: Oscar Casas García (ocgarcia@usbcali.edu.co)

ISBN: 978-958-8436-45-6

Rector
Fray Álvaro Cepeda van Houten, OFM

Secretario
Fray Hernando Arias Rodríguez, OFM

Vicerrector Académico
Juan Carlos Flórez Buriticá

Vicerrector Administrativo y Financiero
Félix Remigio Rodríguez Ballesteros

Directora Investigaciones
Angela Rocío Orozco Zárate
e-mail: arorozco@usbcali.edu.co

Director Proyección Social
Ricardo Antonio Bastidas

Coordinador Editorial Bonaventuriana
Claudio Valencia Estrada
e-mail: clave@usbcali.edu.co

Diseño y diagramación: Edward Carvajal A.

© Universidad de San Buenaventura, seccional Cali

Universidad de San Buenaventura, seccional Cali
La Umbría, carretera a Pance
A.A. 25162
PBX: (572)318 22 00 – (572)488 22 22
Fax: (572)488 22 31/92
www.usbcali.edu.co • e-mail: EditorialBonaventuriana@usbcali.edu.co
Cali - Colombia, Sur América

Este libro no puede ser reproducido total o parcialmente por ningún medio
sin autorización escrita de la Universidad de San Buenaventura, seccional Cali.

Cali, Colombia
Diciembre de 2010

Tabla de contenido

Presentación 7

Cifradores de bloque 9
Conceptos básicos sobre criptografía 9
Cifradores de bloque 12

Algoritmo de cifrado *Serpent* 19
Cifrador *Serpent* 19
Expansión de clave *Serpent* 26
Descifrador *Serpent* 27

Algoritmo de Cifrado MARS 33
Cifrador MARS 33
Expansión de Clave MARS 47
Descifrador MARS 49

Algoritmo de Cifrado *Twofish* 55
Cifrador *Twofish* 55
Expansión de clave *Twofish* 63
Descifrador *Twofish* 64

Algoritmo de cifrado *Rijndael* 67
Cifrador *Rijndael* 67

Expansión de clave *Rijndael* 71
Descifrador *Rijndael* 73

Algoritmo de cifrado RC6 77
Cifrador *RC6* 77
Expansión de clave *RC6* 82
Descifrador *RC6* 82

Diseño e implementación de los co-procesadores 87
Parámetros de desempeño relevantes en la implementación de los co-procesadores 88
Estimación de potencia 89
Diseño de los cifradores de bloque 91
Implementación de los cifradores de bloque 93
Diseño de los co-procesadores de cifrado de bloque 95
Implementación de los co-procesadores de cifrado de bloque 96

Bibliografía 107

Presentación

El rápido avance de la tecnología ha generado la necesidad de alcanzar una alta confidencialidad en el manejo de la información, para lo cual se han desarrollado varios algoritmos y sistemas criptográficos; sin embargo, algunos de estos ya han quedado obsoletos puesto que han podido ser descifrados exitosamente por los criptoanalistas. Por tanto, existe la necesidad de desarrollar nuevos algoritmos criptográficos que resistan los ataques basados en el uso de computadores de alto desempeño.

Durante el año 2001, el *National Institute of Standards and Technology*-NIST, llevó a cabo el proceso de coordinación para reemplazar el algoritmo *Data Encryption Standard*-DES, algoritmo nacido en 1974 que trabaja sobre bloques de 64 bits (Agudelo, 2007).

En este proceso de selección del nuevo algoritmo para reemplazar al DES y convertirse en el *Advanced Encryption Standard*-AES, sólo cinco algoritmos candidatos fueron seleccionados por NIST como finalistas: *MARS* (Burwick y otros, 1999), *RC6* (Rivest y otros, 1998), *Rijndael* (Daemen y Rijmen, 1997), *Serpent* (Anderson y otros, 1998) y *Twofish* (Schneier y otros, 1998). Estos candidatos fueron evaluados bajo los parámetros de diseño, codificación, simulación y síntesis. Finalmente, el algoritmo *Rijndael* fue seleccionado para ser el *Advanced Encryption Standard* debido a su seguridad, desempeño, eficiencia, flexibilidad y facilidad de implementación.

Es de notar que durante el proceso de selección del AES los algoritmos finalistas fueron implementados en software y algunos de estos algoritmos están diseñados para ser más eficientes en su implementación en software, que en hardware.

Son necesarias, entonces, las implementaciones en hardware de estos algoritmos para comparar diferentes características que estos presentan al momento de ser

implementados en hardware, como por ejemplo en un dispositivo lógico programable (PLD-*Programmable Logic Device*). Entre algunas de estas características encontramos: seguridad de la información, velocidad de cifrado, consumo de potencia y recursos usados en el PLD.

Por esta razón el grupo de investigación Laboratorio de Electrónica Aplicada – LEA, de la Universidad de San Buenaventura, seccional Cali, se puso a la tarea de implementar en hardware estos algoritmos a manera de co-procesadores dedicados a cifrar y descifrar datos. La implementación se realiza en un PLD usando captura esquemática y un lenguaje de descripción de hardware (HDL -*Hardware Description Language*).

El objetivo principal consistió en adquirir experimentalmente la *velocidad de cifrado*, *recursos usados* y *potencia consumida* de cada co-procesador. Estos parámetros son especialmente útiles en el momento de diseñar un sistema embebido que utilice criptografía, puesto que al tener un conjunto de 5 co-procesadores con sus respectivos parámetros de rendimiento, se podrá escoger el co-procesador que mejor se adapte a la aplicación específica del sistema embebido. Es decir, si se quiere que el sistema consuma la menos cantidad de potencia, sólo basta revisar los parámetros de los co-procesadores disponibles y utilizar el que presente mejor desempeño en esta característica.

Este proyecto se desarrolló, con el apoyo de la Universidad de San Buenaventura, seccional Cali, y participaron en la misma: Oscar Casas García como investigador, e Ivonne Agudelo Torres, Diana Fernanda Valencia Solarte, Ana Elvira Venté Mancilla, Jorge Alberto Medina Oliveros, David Guillermo Rivera Fierro y Manuel Alejandro Vélez Gordillo como auxiliares de investigación. Este libro presenta, entonces, una recopilación del trabajo realizado y los resultados definitivos alcanzados.

Cifradores de bloque

Conceptos básicos sobre criptografía

La definición de *criptografía* se puede dar como el estudio de técnicas matemáticas relacionado con los aspectos de seguridad de la información como la confidencialidad, integridad de los datos, autenticación de la entidad y autenticación de origen de datos. La criptografía no es el único medio para proporcionar seguridad a la información, pero sí es una de las técnicas más usadas.

A continuación se listan algunos de los objetivos más importantes de la criptografía (Menezes y otros, 1996):

– *Privacidad o confidencialidad.* Su objetivo es mantener en secreto la información, excepto para quienes están autorizados.

– *Integridad de los datos.* Asegurar que la información no ha sido alterada por entes desautorizados o medios desconocidos.

– *Autenticación de la entidad o identificación.* Corroborar la identidad de un ente (por ejemplo: una persona, una terminal de computador, una tarjeta de crédito, etc.).

– *Autenticación del mensaje.* Verificar la fuente de información; también conocido como la autenticación de origen de datos.

– *Firma.* Asignar la información a un ente en específico.

– *Autorización.* Aprobar a otro ente para acceder a la información.

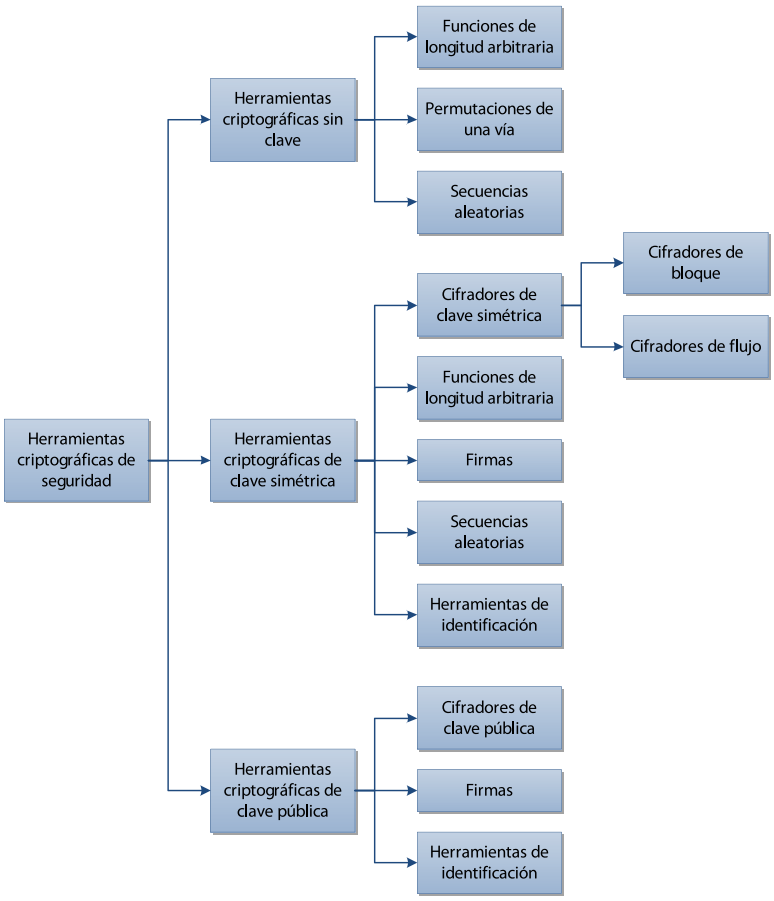
– *Validación.* Proveer un tiempo de autorización para utilizar o manipular la información.

– *Certificación.* Comprobar el origen de la información por medio de un ente confiable.

– *Confirmación.* Controlar de que se hayan proporcionado los servicios requeridos.

La criptografía en sí, se divide en dos tipos de sistemas: la criptografía simétrica y la asimétrica. Esencialmente, con la primera se resuelven los problemas de confidencialidad e integridad, mientras que con la segunda se resuelven los de autenticidad y no rechazo. La Figura 1.1 presenta un panorama más detallado sobre la división jerárquica de las herramientas criptográficas de seguridad (Schneier, 2000).

Figura 1.1. División de las herramientas criptográficas de seguridad



Criptografía simétrica

Los *sistemas simétricos* de cifrado utilizan una misma clave, tanto para el proceso de cifrado como el proceso inverso, o bien la clave para este último deriva fácilmente de la primera. Las dos partes que se comunican han de ponerse de acuerdo de antemano sobre la clave a usar. Una vez ambas tienen acceso a esta clave, el remitente cifra un mensaje usándola, lo envía al destinatario y este lo descifra con la misma.

El principal problema con los sistemas de cifrado simétrico no está ligado a su seguridad, sino al intercambio de claves. Una vez que el remitente y el destinatario han intercambiado las claves, ellos pueden usarlas para comunicarse con seguridad, pero ¿qué canal de comunicación, que sea seguro, han usado para transmitirse la clave entre sí? Sería mucho más fácil para un atacante intentar interceptar una clave, que probar las posibles combinaciones del espacio de claves.

Al comparar los sistemas criptográficos simétricos y asimétricos, se puede observar que los primeros son más sencillos, menos complejos matemáticamente y han sido empleados durante más tiempo. El nivel de seguridad de los sistemas simétricos depende, casi exclusivamente, de la clave compartida únicamente entre emisor y receptor.

Por otra parte los cifradores simétricos se pueden dividir en dos tipos: los cifradores de bloque y los cifradores de flujo. Los primeros operan partiendo la información en bloques de datos de tamaño fijo y realizando transformaciones individuales a cada bloque; mientras que los segundos generan un flujo de datos, donde cada nueva transformación depende de datos anteriores.

Criptografía asimétrica

La principal característica de la *criptografía asimétrica* (también llamada criptografía de clave pública), es que no se basa en una única clave sino en un par de ellas: una conocida (pública) y otra *privada*.

El nivel de seguridad de la criptografía de dos claves depende de problemas matemáticos difíciles de resolver, de una manera que no se aplica a la criptografía de clave simétrica; y recíprocamente relacionan el criptoanálisis con la investigación matemática en general de nuevas maneras; así que un punto obvio de ataque es desarrollar métodos para resolver el problema matemático en cuestión.

Es decir, los algoritmos asimétricos se diseñan en torno a la dificultad y necesidad de resolver ciertos problemas matemáticos. Por ejemplo, la seguridad del protocolo Diffie-Hellman depende de la dificultad de calcular un logaritmo discreto; aunque como se mencionó anteriormente, el ataque más prominente consiste en resolver los problemas matemáticos.

En 1983, Don Coppersmith encontró una manera más rápida de calcular logaritmos discretos (dentro de ciertos grupos), y por tanto obligó a los criptógrafos a utilizar grupos más grandes, o diferentes tipos de grupos. Actualmente existen muchos algoritmos de este tipo pero han demostrado ser poco utilizables en la práctica ya sea por la longitud de las claves, la longitud del texto encriptado generado o su velocidad de cifrado extremadamente prolongada.

Otra característica distintiva de los algoritmos asimétricos es que, a diferencia de los ataques sobre criptosistemas simétricos, cualquier criptoanálisis tiene la oportunidad de usar el conocimiento obtenido de la clave pública.

Cifradores de bloque

Un texto cualquiera de información, denominado texto plano, está compuesto por un determinado número de bits, estos bits se dividen en bloques del mismo tamaño. El método de cifrado en bloque transforma cada bloque de datos en otro de igual tamaño. Éste es el método de cifrado simétrico más empleado en la actualidad, básicamente por el apoyo recibido por el gobierno de los EE.UU.

Los cifradores de bloque de clave simétrica son los elementos más importantes en varios sistemas criptográficos. Individualmente proveen confidencialidad y como bloque fundamental, su versatilidad permite la construcción de generadores de números aleatorios, además de cifradores de flujo, entre otros.

Un cifrador de bloques no satisface idealmente todos los usos aunque este ofrezca un alto nivel de seguridad. Éste es un resultado de los compromisos inevitables requeridos en usos prácticos, incluyendo los que surgen de, por ejemplo, requerimientos de velocidad y limitaciones de memoria (tamaño de código, tamaño del dato, memoria caché); limitaciones impuestas por la plataforma de implementación (hardware, software, chipcards); y la diferencia de tolerancias de aplicaciones a las propiedades de varios modos de operación. Además, la eficacia debe ser compensada contra la seguridad. Entonces es conveniente tener un gran número de candidatos de cifrado para escoger.

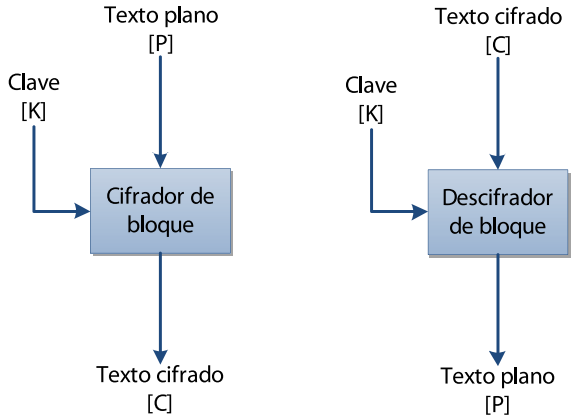
La diferencia esencial entre los cifradores de flujo y los cifradores de bloque radica en que en los primeros se cifra el mensaje 'dato a dato', mientras que en los cifradores de bloque se aplica el algoritmo de cifrado a 'bloques de datos' de un tamaño fijo y predeterminado; es decir, cifran un bloque de bits (habitualmente cada bloque es de 128 bits) como una única unidad.

Definición

Un *cifrador de bloque* es una función que mapea un bloque de texto plano de n-bits a un bloque de texto cifrado de n-bits. Donde n es la longitud del bloque. La función es parametrizada por una clave K de k-bits, tomando valores de un subconjunto del conjunto V_k de todos los vectores de k-bits. Asumiendo siempre que la clave es escogida al azar, los bloques de texto plano y texto cifrado de igual longitud evitan la expansión de los datos. Además, para permitir el cifrado y el descifrado, la función debe de ser uno a uno; es decir, esta función debe ser invertible.

Aunque los cifradores de bloque generalmente procesan texto plano en tamaños relativamente grandes, por ejemplo $n \geq 128$, y los cifradores de flujo típicamente procesan unidades más pequeñas, no hay una distinción exacta entre ambos, puesto que los cifradores de bloque se pueden utilizar para realizar cifrado de flujo.

Figura 1.2. Cifrador y descifrador de bloque



Modos de operación

Para los mensajes de texto plano que exceden la longitud del bloque, se usan varios modos de operación para los cifradores de bloque. Por ejemplo cuando un mensaje excede el tamaño del bloque, la forma más fácil de llevar a cabo el cifrado es dividiendo o particionando este mensaje en bloques de *n*-bits y así cifrar separadamente uno a uno.

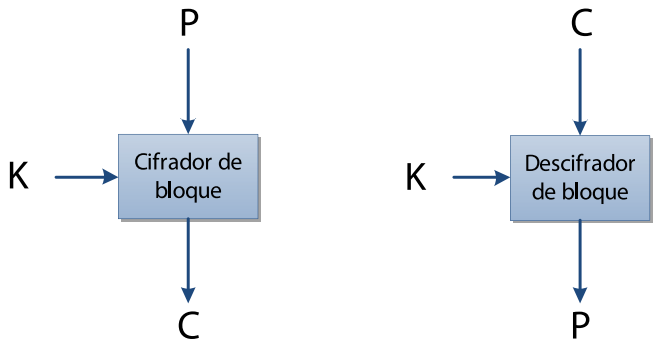
Este método empleado (denominado ECB) es uno de los modos de operación que existen para resolver problemas de texto excedido. Algunos de estos modos, y los más utilizados, son:

– *Electronics codebook* (ECB).
– *Cipher-block chaining* (CBC).
– *Cipher feedback* (CFB).
– *Output feedback* (OFB).

Electronics-codebook (ECB)

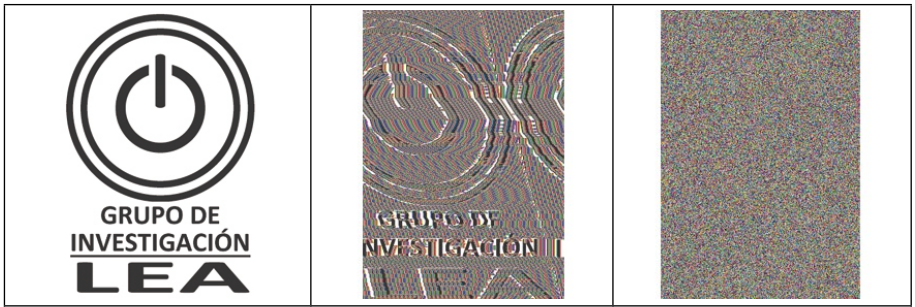
Es el más simple de los modos de cifrado, en ECB el mensaje es dividido en bloques, cada uno de los cuales es cifrado de manera separada. La desventaja de este método es que bloques idénticos de texto plano producirán bloques idénticos de texto cifrado. Por esto, no proporciona una auténtica confidencialidad y no es recomendado para protocolos criptográficos.

Figura 1.3. *Electronics codebook* (ECB)



La Figura 1.4 muestra un claro ejemplo de cómo el método ECB puede revelar patrones del texto. Una versión de mapa de bits de la imagen de la izquierda ha sido cifrada con ECB para generar la imagen del centro.

Figura 1.4. Patrones del modo ECB



a. Original	b. Cifrado usando modo ECB	c. Cifrado usando otro modo

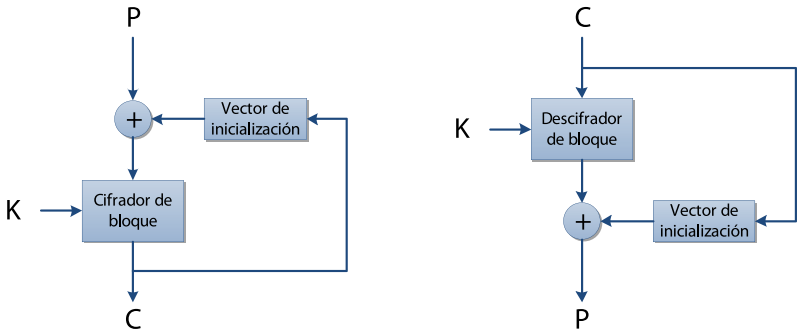
La imagen de la derecha muestra cómo podría aparecer la misma imagen cifrada con CBC, CTR o cualquiera de los otros métodos; esto es, indistinguible de ruido aleatorio. Aunque cabe notar que la apariencia aleatoria de la imagen de la derecha dice muy poco sobre si la imagen ha sido cifrada de manera segura. Muchos métodos no seguros de cifrado producen, a priori, un resultado aparentemente aleatorio.

El modo ECB puede hacer que protocolos sin protección de integridad sean aún más susceptibles a ataques de repetición, dado que cada bloque es descifrado de la misma manera.

Cipher-block chaining (CBC)

En este modo a cada bloque de texto plano se le aplica la operación XOR con el bloque cifrado anterior antes de ser cifrado. De esta forma, cada bloque de texto cifrado depende de todo el texto plano procesado hasta este punto. También, para hacer cada mensaje único se utiliza un vector de inicialización.

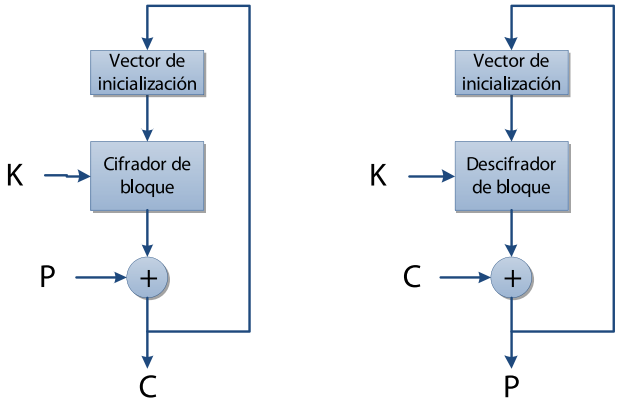
Figura 1.5. Cipher-block chaining (CBC)



Cipher feedback (CFB)

Mientras el modo CBC procesa texto plano de *n*-bits en un tiempo (usando un cifrador de bloques de *n*-bits) algunas aplicaciones requieren que *r*-bits del texto plano sean encriptados y transmitidos sin retraso, para algún $r<n$ (a menudo *r=1 o r=8*), es en este caso que se usa el cifrador de realimentación o *Cipher Feedback* (CFB).

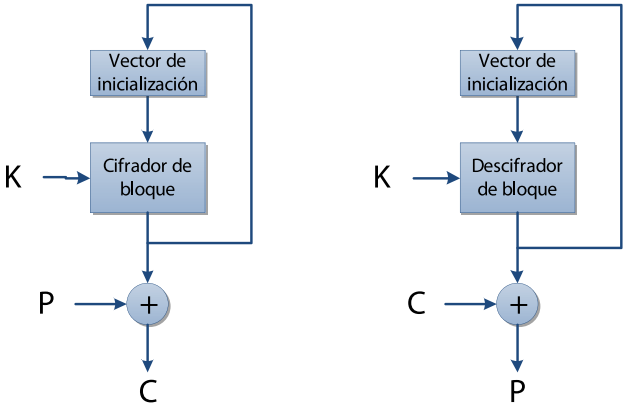
Figura 1.6. Cipher feedback (CFB)



Output feedback (OFB)

El modo de operación OFB puede usarse en aquellas aplicaciones que se quiera evitar el error de propagación. Es similar al CFB y permite la encriptación de varios tamaños del bloque pero difiere en que en este caso la salida del bloque de encriptación sirve como realimentación.

Figura 1.7. Output feedback (OFB)



Los ataques

El objetivo de un cifrador de bloque es proporcionar confidencialidad. El objetivo correspondiente de un adversario es recuperar el texto plano a partir del texto cifrado. Un cifrador de bloque está totalmente roto si la clave puede encontrarse, y parcialmente roto si un adversario puede recuperar parte del texto plano (pero no la clave) a partir del texto cifrado.

Para esto hay que anotar que siempre que se quiera evaluar la seguridad de un cifrador de bloque se debe asumir a un adversario y que este:

– Tenga acceso a todos los datos transmitidos sobre el canal del texto cifrado.
– Sepa todo los detalles de la función de encriptación excepto la clave secreta.

Lo ataques son clasificados en base a la información a la que tenga acceso el criptoanalista además del texto cifrado interceptado. Las clases de ataques más importantes para los cifradores de clave simétrica son (para clave fija):

– *Ataques de sólo texto cifrado*

  En esta situación el atacante no conoce nada sobre el contenido del mensaje, y debe trabajar sólo desde el texto cifrado. Es decir, ninguna información adicional está disponible. Los criptosistemas modernos no son débiles contra ataques de "texto cifrado", aunque algunas veces se considera que el mensaje contiene "tendencia" estática.

– *Ataques de texto plano conocidos*

  El atacante conoce o puede adivinar el texto de alguna parte del texto cifrado. La tarea es obtener la clave utilizando sólo esta información. Esto puede ser hecho utilizando el texto plano original o a través de algún atajo. Es decir, las parejas de texto plano y texto cifrado están disponibles.

– *Ataques de texto plano escogido*

  El atacante puede tener cualquier texto encriptado con una clave desconocida. La tarea es determinar la clave utilizada para cifrar. En otras palabras el texto cifrado sirve como opción de texto plano para el adversario.

En sí un ataque de texto cifrado escogido opera bajo el siguiente modelo: Teniendo en cuenta que se parte de la suposición de una clave desconocida, se permite que el adversario acceda a una pareja de texto plano y texto cifrado

para obtener información; después de esto, se intenta usar esta información para recuperar la clave (o texto plano que corresponde a un nuevo texto cifrado).

El ataque de texto plano escogido es el más usado puesto que es más resistente. Un cifrador resistente a los ataques de texto plano escogido, es también resistente contra los ataques de sólo texto cifrado y texto plano conocido.

Con pocas excepciones la mejor medida de seguridad contra los mejores ataques de la actualidad, disponible para los cifradores prácticos es la complejidad. Los aspectos que pueden distinguirse de tal complejidad son:

– *Complejidad de los datos:* hace referencia al número de unidades de datos de entrada requerido; por ejemplo, en un cifrador de bloque de 128 bits, la complejidad de los datos es de 128 bits.

– *Complejidad de almacenamiento:* hace referencia al número de unidades del almacenamiento requerido.

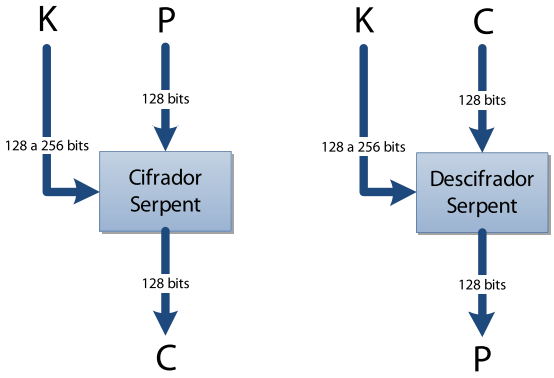
– *Complejidad de procesamiento:* hace referencia al número de operaciones exigido para procesar los datos de la entrada y/o almacenamiento de datos (por lo menos una unidad de tiempo por la unidad almacenamiento).

Algoritmo de cifrado Serpent

El algoritmo de cifrado *Serpent* fue diseñado por Ross Anderson, Eli Biham y Lars Knudsen (Anderson y otros, 1998).

Serpent encripta 128 bits de texto plano *P*, en 128 bits de texto cifrado C, a través de 32 rondas controladas por 33 subclaves de 128 bits, denominadas. $[\widehat{K_0}\ \widehat{K_1}\ \ldots\ \widehat{K_{32}}]$. La clave de usuario es variable, ya que uno de los propósitos es que se pueda manejar la clave con valores entre 128 y 256 bits. Las claves más pequeñas son introducidas como una clave completa de 256 bits y se manejan internamente como: [1000... *K*]. La Figura 2.1 presenta las convenciones seguidas en el diseño del cifrador *Serpent*.

Figura 2.1. Convenciones para el cifrado y descifrado *Serpent*



Cifrador *Serpent*

El cifrador de bloque *Serpent* en sí consta de:

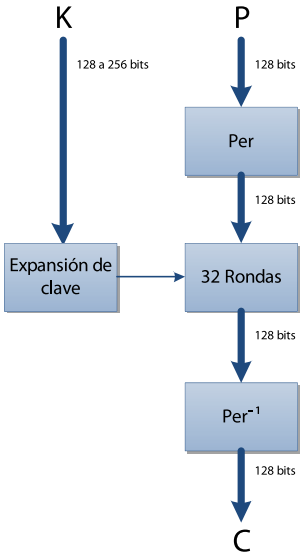
– Una *permutación inicial*, denominada *Per*.

– 32 rondas de mezcla, usando mezcla de clave de ronda, cajas-S y una transformación lineal. En la última ronda de reemplaza la transformación lineal por una mezcla adicional de la clave de ronda.

– Una *permutación final*, denominada Per^{-1}.

En la Figura 2.2 se presenta un esquema del diagrama de bloques del cifrado *Serpent*. El descifrado se puede apreciar en la página 27 (Descifrador *MARS*).

Figura 2.2. Esquema básico del cifrador *Serpent*



El algoritmo de cifrado se encuentra descrito por las siguientes ecuaciones:

$$B_0 = Per(P) \qquad (2.1)$$
$$B_{i+1} = R_i(B_i) \qquad (2.2)$$
$$C = Per^{-1}(B_{32}) \qquad (2.3)$$

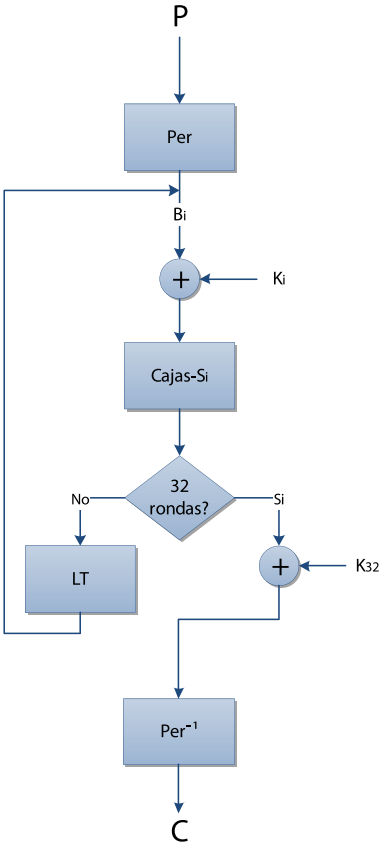
Donde,

$$R_i(X) = LT\left(S_i\left(X \oplus \widehat{K_i}\right)\right) \qquad i = 0,1,\ldots,30 \qquad (2.4)$$
$$R_i(X) = S_i\left(X \oplus \widehat{K_i}\right) \oplus \widehat{K_{32}} \qquad i = 31 \qquad (2.5)$$

S_i es la aplicación de las cajas-*S*, $S_{(i \bmod 8)}$, 32 veces en paralelo *LT*, es la transformación lineal, *Per* es la permutación inicial, es la permutación final R_i, es la función de ronda $\widehat{K_i}$ es la clave de ronda *P*, es el texto plano B_i, es la salida de

las rondas y C el texto cifrado. La Figura 2.3 muestra el algoritmo de cifrado *Serpent* en diagrama de flujo.

Figura 2.3. Algoritmo de cifrado *Serpent*



Permutación inicial

Consiste en trasladar cada uno de los bits del texto de entrada a posiciones establecidas a la salida, acorde con la Tabla 2.1. La Figura 2.4 muestra un esquema del diseño en hardware de esta permutación.

Figura 2.4. Permutación inicial

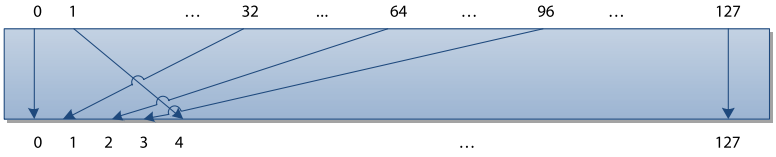


Tabla 2.1. Permutación inicial

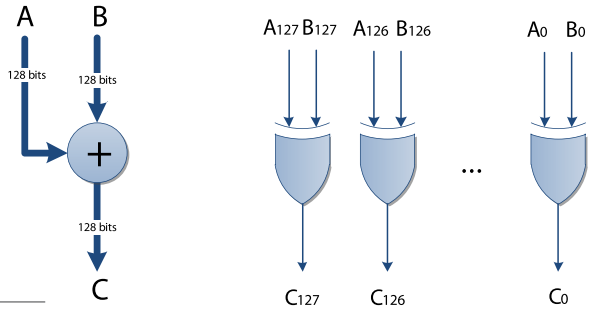
B.E[1]	B.S[2]	B.E	B.S	B.E	B.S	B.E	B.S	B.E	B.S	B.E	B.S	B.E	B.S	B.E	B.S
0	0	4	16	8	32	12	48	16	64	20	80	24	96	28	112
32	1	36	17	40	33	44	49	48	65	52	81	56	97	60	113
64	2	68	18	72	34	76	50	80	66	84	82	88	98	92	114
96	3	100	19	104	35	108	51	112	67	116	83	120	99	124	115
1	4	5	20	9	36	13	52	17	68	21	84	25	100	29	116
33	5	37	21	41	37	45	53	49	69	53	85	57	101	61	117
65	6	69	22	73	38	77	54	81	70	85	86	89	102	93	118
97	7	101	23	105	39	109	55	113	71	117	87	121	103	125	119
2	8	6	24	10	40	14	56	18	72	22	88	26	104	30	120
34	9	38	25	42	41	46	57	50	73	54	89	58	105	62	121
66	10	70	26	74	42	78	58	82	74	86	90	90	106	94	122
98	11	102	27	106	43	110	59	114	75	118	91	122	107	126	123
3	12	7	28	11	44	15	60	19	76	23	92	27	108	31	124
35	13	39	29	43	45	47	61	51	77	55	93	59	109	63	125
67	14	71	30	75	46	79	62	83	78	87	94	91	110	95	126
99	15	103	31	107	47	111	63	115	79	119	95	123	111	127	127

Ronda de cifrado

El cifrado en sí consta de 32 rondas en donde se utilizan siempre las mismas operaciones, excepto la última ronda (donde la transformación lineal se reemplaza por una mezcla adicional); entonces, se diseña un solo bloque de ronda que se utiliza iterativamente en 31 ocasiones. Las operaciones que utiliza el bloque de ronda son las siguientes:

– *Mezcla de claves de ronda:* Al inicio de la ronda, se utiliza una clave de ronda $\widehat{K}_i$ de 128 bits, y se realiza la operación XOR con los datos inmediatos que ingresan al bloque. En la Figura 2.5 se muestra un esquema del diseño en hardware de este bloque de mezcla.

Figura 2.5. Mezcla de claves de ronda



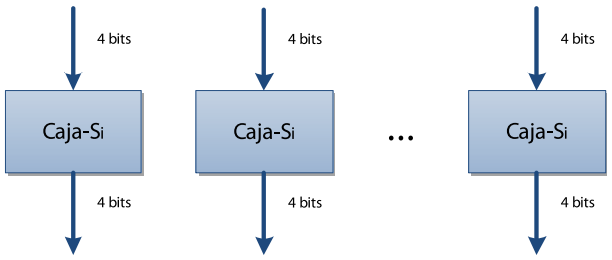
1. B.E: Bit de entrada.
2. B.S: Bit de salida.

– *Cajas-S_i:* Se usan 8 cajas de sustitución o cajas-S_i (S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7), cada una de las cuales sustituye 4 bits a su entrada, como lo muestra la Tabla 2.2. En este caso, en cada ronda se usa solamente una caja-S; pero cada caja-S es usada en 4 ocasiones durante toda la secuencia del algoritmo. Cada vez que se usa una caja es necesario 32 instancias de la misma, en paralelo, para poder sustituir un bloque de 128 bits (Ver Figura 2.6).

Tabla 2.2. Cajas-S_i

Entrada	Salida							
E	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
0x0	0x3	0xF	0x8	0x0	0x1	0x F	0x7	0x1
0x1	0x8	0xC	0x6	0xF	0xF	0x5	0x2	0xD
0x2	0xF	0x2	0x7	0xB	0x8	0x2	0xC	0xF
0x3	0x1	0x7	0x9	0x8	0x3	0xB	0x5	0x0
0x4	0xA	0x9	0x3	0xC	0xC	0x4	0x8	0xE
0x5	0x6	0x0	0xC	0x9	0x0	0xA	0x4	0x8
0x6	0x5	0x5	0xA	0x6	0xB	0x9	0x6	0x2
0x7	0xB	0xA	0xF	0x3	0x6	0xC	0xB	0xB
0x8	0xE	0x1	0xD	0xD	0x2	0x0	0xE	0x7
0x9	0xD	0xB	0x1	0x1	0x5	0x3	0x9	0x4
0xA	0x4	0xE	0xE	0x2	0x4	0xE	0x1	0xC
0xB	0x2	0x8	0x4	0x4	0xA	0x8	0xF	0xA
0xC	0x7	0x6	0x0	0xA	0x9	0xD	0xD	0x9
0xD	0x0	0xD	0xB	0x7	0xE	0x6	0x3	0x3
0xE	0x9	0x3	0x5	0x5	0x7	0x7	0xA	0x5
0xF	0xC	0x4	0x2	0xE	0xD	0x1	0x0	0x6

Figura 2.6. Cajas-S_i en paralelo

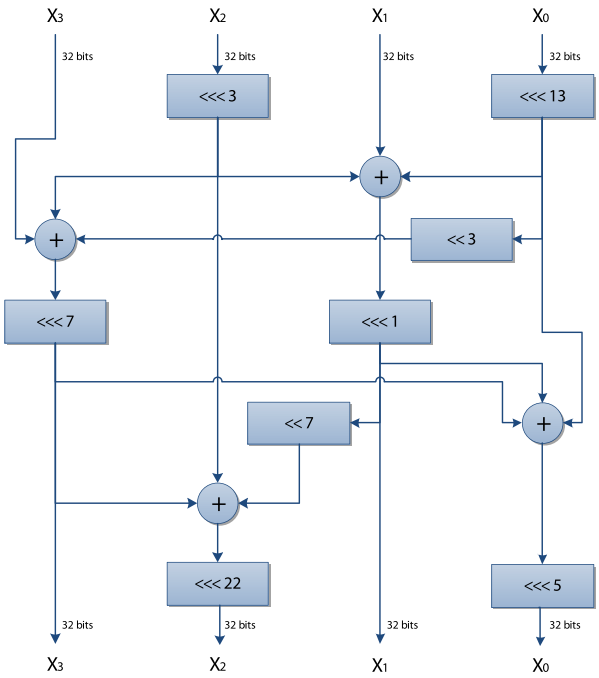


– *Transformación lineal:* Los 128 bits que salen de las cajas-S_i, tomados en 4 grupos de 32 bits $[X_0\, X_1\, X_2\, X_3]$, se combinan linealmente entre sí de la siguiente manera:

$$[X_0\, X_1\, X_2\, X_3] = S_i\left(B_i \oplus \hat{K}_i\right) \quad (2.6)$$
$$X_0 = X_0 <<< 13 \quad (2.7)$$
$$X_2 = X_2 <<< 3 \quad (2.8)$$
$$X_1 = X_0 \oplus X_1 \oplus X_2 \quad (2.9)$$
$$X_3 = X_3 \oplus X_2 \oplus (X_0 << 3) \quad (2.10)$$
$$X_1 = X_1 <<< 1 \quad (2.11)$$
$$X_3 = X_3 <<< 7 \quad (2.12)$$
$$X_0 = X_0 \oplus X_1 \oplus X_3 \quad (2.13)$$
$$X_2 = X_2 \oplus X_3 \oplus (X_1 << 7) \quad (2.14)$$
$$X_0 = X_0 <<< 5 \quad (2.15)$$
$$X_2 = X_2 <<< 22 \quad (2.16)$$
$$B_{i+1} = [X_0\, X_1\, X_2\, X_3] \quad (2.17)$$

Donde $<<<$ denota una rotación cíclica a la izquierda y $<<$ un desplazamiento o corrimiento lógico a la izquierda. La Figura 2.7 presenta el diagrama de bloques de la transformación lineal.

Figura 2.7. Transformación lineal



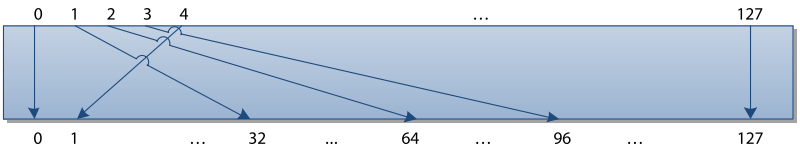
Permutación final

Al igual que la *permutación inicial*, la *permutación final* consiste en trasladar los bits del texto cifrado a posiciones establecidas a la salida, tal como se muestra en la Tabla 2.3. La *permutación final* es la permutación inversa de la inicial. La Figura 2.8 muestra un esquema del diseño en hardware de esta permutación.

Tabla 2.3. Permutación final

B.E[3]	B.S[4]	B.E	B.S	B.E	B.S	B.E	B.S	B.E	B.S	B.E	B.S	B.E	B.S	B.E	B.S
0	0	64	16	1	32	65	48	2	64	66	80	3	96	67	112
4	1	68	17	5	33	69	49	6	65	70	81	7	97	71	113
8	2	72	18	9	34	73	50	10	66	74	82	11	98	75	114
12	3	76	19	13	35	77	51	14	67	78	83	15	99	79	115
16	4	80	20	17	36	81	52	18	68	82	84	19	100	83	116
20	5	84	21	21	37	85	53	22	69	86	85	23	101	87	117
24	6	88	22	25	38	89	54	26	70	90	86	27	102	91	118
28	7	92	23	29	39	93	55	30	71	94	87	31	103	95	119
32	8	96	24	33	40	97	56	34	72	98	88	35	104	99	120
36	9	100	25	37	41	101	57	38	73	102	89	39	105	103	121
40	10	104	26	41	42	105	58	42	74	106	90	43	106	107	122
44	11	108	27	45	43	109	59	46	75	110	91	47	107	111	123
48	12	112	28	49	44	113	60	50	76	114	92	51	108	115	124
52	13	116	29	53	45	117	61	54	77	118	93	55	109	119	125
56	14	120	30	57	46	121	62	58	78	122	94	59	110	123	126
60	15	124	31	61	47	125	63	62	79	126	95	63	111	127	127

Figura 2.8. Permutación final



3. B.E: Bit de entrada.
4. B.S: Bit de salida.

Expansión de clave Serpent

El algoritmo de cifrado *Serpent* requiere de 33 palabras de 128 bits, denominadas claves de ronda, para el proceso de cifrado/descifrado. Estas claves de ronda se derivan de la clave *K*, a través de un proceso concurrente denominado *generador de claves de ronda*.

El primer paso, para generarlas, consiste en proporcionar la clave *K* de 256 bits, si es de menor tamaño se complementa la clave de 256 bits como [1000 ... *K*]. Luego la clave se amplía a 33 claves de ronda de 128 bits ($K_0 \ldots K_{32}$) por medio del método descrito a continuación:

– Se trabaja la clave *K* como palabras de 32 bits [$W_{-8} \ldots W_{-1}$] y se amplía esa clave a 132 preclaves intermedias de 32 bits cada [$W_0 \ldots W_{131}$] una usando la Ecuación 2.18, donde φ es *0x9E3779B9*.

$$W_i = W_{i-8} \oplus W_{i-5} \oplus W_{i-3} \oplus W_{i-1} \oplus \varphi \oplus i <<< 11 \qquad (2.18)$$

– Las subclaves de ronda K_i se calculan con las preclaves W_i. Se utiliza la caja de sustitución S_3 para transformar las preclaves W_i en las subclaves de ronda K_i.

$$k_i = S_3(W_i) \qquad i = 0,1,\ldots,132 \qquad (2.19)$$

– Entonces se renombra el valor de las claves K_i de 32 bits como subclaves K_i de 128 bits correspondientes a cada iteración de la siguiente forma.

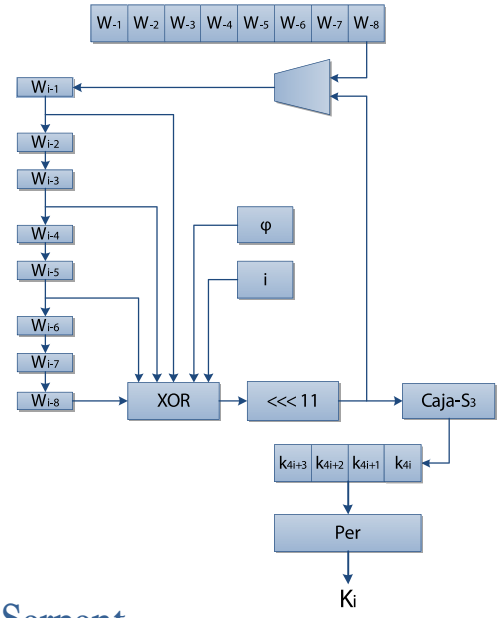
$$K_i = [k_{4i}\ k_{4i+1}\ k_{4i+2}\ k_{4i+3}] \qquad i = 0,1,\ldots,32 \qquad (2.20)$$

– Finalmente se aplica la permutación inicial *IP* a cada una de las claves K_i.

$$\widehat{K}_i = IP(K_i) \qquad (2.21)$$

En la Figura 2.9 se presenta el diagrama del bloque generador de claves de ronda acorde a las ecuaciones 2.18 – 2.21.

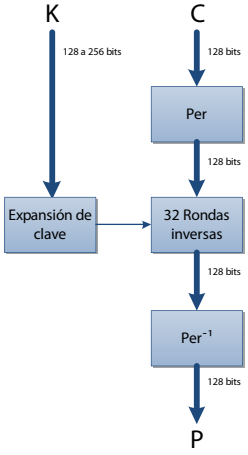
Figura 2.9. Generación de claves de ronda



Descifrador Serpent

El descifrador es similar al cifrador y contiene los mismos elementos básicos: las permutaciones inicial y final y 32 rondas de mezcla. Se diferencia del cifrador en que las cajas-S_i deben ser inversas, las claves de ronda $\widehat{K}_i$ se utilizan en orden inverso, y se debe utilizar la transformación lineal inversa. En la Figura 2.10 se presenta un esquema del diagrama de bloques del descifrado *Serpent*.

Figura 2.10. Esquema básico del descifrador *Serpent*



El algoritmo de descifrado es el siguiente:

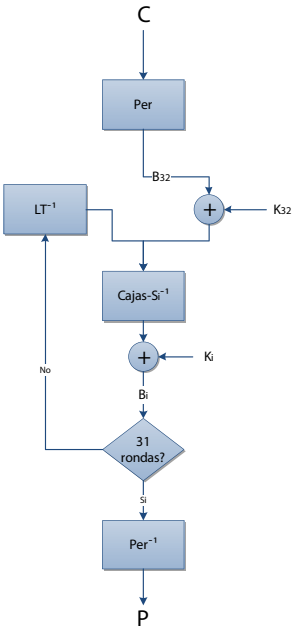
$$B_{32} = Per(C) \quad (2.22)$$
$$B_{i-1} = R_{i-1}(B_i) \quad (2.23)$$
$$P = Per^{-1}(B_0) \quad (2.24)$$

Donde

$$R_{i-1} = \left(\widehat{K_{i-1}} \oplus S_{i-1}{}^{-1}\left(B_i \oplus \widehat{K_i}\right)\right) \quad i = 32 \quad (2.25)$$
$$R_{i-1} = \left(\widehat{K_i} \oplus S_{(i \bmod 8)}{}^{-1}(LT^{-1}(B_i))\right) \quad i = 31,30,\ldots,1 \quad (2.26)$$

La Figura 2.11 muestra el algoritmo de descifrado *Serpent* en diagrama de flujo.

Figura 2.11. Algoritmo de descifrado *Serpent*



Permutación final inversa

Como su nombre lo indica, es la permutación inversa a la permutación final, y por tanto corresponde exactamente a la permutación inicial (ver Tabla 2.1 y Figura 2.4).

Ronda de descifrado

Al igual que el cifrado, el descifrado consta de 32 rondas en donde se utilizan siempre las mismas operaciones, excepto en la primera ronda (donde la transformación lineal inversa se reemplaza por una mezcla de clave de ronda);

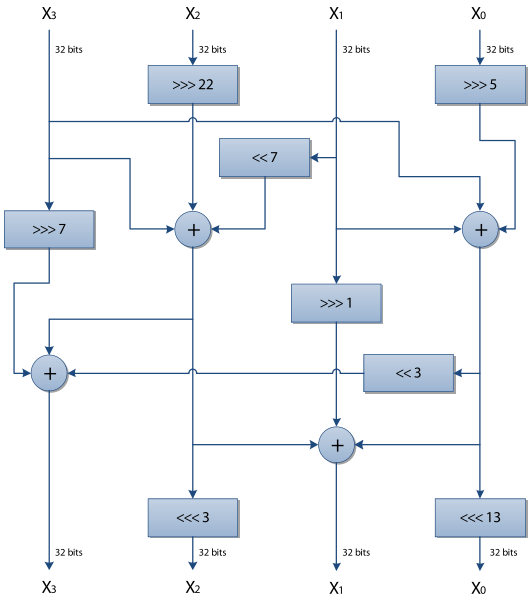
entonces, se diseña un solo bloque de ronda que se utiliza iterativamente en 31 ocasiones. Las operaciones que utiliza el bloque de ronda son las siguientes:

– *Transformación lineal inversa:* Los 128 bits que salen de la mezcla de clave de ronda, se toman en 4 grupos de 32 bits $[X_0\, X_1\, X_2\, X_3]$, y se combinan linealmente entre sí de la siguiente manera:

$$[X_0\, X_1\, X_2\, X_3] = B_i \quad (2.27)$$
$$X_2 = X_2 >>> 22 \quad (2.28)$$
$$X_0 = X_0 >>> 5 \quad (2.29)$$
$$X_2 = X_2 \oplus X_3 \oplus (X_1 << 7) \quad (2.30)$$
$$X_0 = X_0 \oplus X_1 \oplus X_3 \quad (2.31)$$
$$X_3 = X_3 >>> 7 \quad (2.32)$$
$$X_1 = X_1 >>> 1 \quad (2.33)$$
$$X_3 = X_3 \oplus X_2 \oplus (X_0 << 3) \quad (2.34)$$
$$X_1 = X_0 \oplus X_1 \oplus X_2 \quad (2.35)$$
$$X_2 = X_2 >>> 3 \quad (2.36)$$
$$X_0 = X_0 >>> 13 \quad (2.37)$$
$$I_i = [X_0\, X_1\, X_2\, X_3] \quad (2.38)$$

Donde $>>>$ denota una rotación cíclica a la derecha y un $<<$ desplazamiento o un corrimiento lógico a la izquierda. La Figura 2.12 presenta el diagrama de bloques de la transformación lineal inversa.

Figura 2.12. Transformación lineal inversa

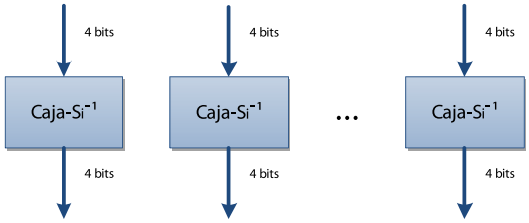


– *Cajas-S_i inversas:* Se usan 8 cajas de sustitución inversas o cajas-S_i^{-1} (S_0^{-1}, S_1^{-1}, S_2^{-1}, S_3^{-1}, S_4^{-1}, S_5^{-1}, S_6^{-1}, S_7^{-1}), cada una de las cuales sustituye 4 bits a su entrada, como s muestra en la Tabla 2.4. En este caso, en cada ronda se usa solamente una caja-S_i^{-1}; pero cada caja-S_i^{-1} es usada en 4 ocasiones durante toda la secuencia del algoritmo. Cada vez que se usa una caja es necesario 32 instancias de la misma en paralelo para poder sustituir un bloque de 128 bits (Ver Figura 2.13).

Tabla 2.4. Cajas-S_i inversas

Entrada	Salida							
E	S_0^{-1}	S_1^{-1}	S_2^{-1}	S_3^{-1}	S_4^{-1}	S_5^{-1}	S_6^{-1}	S_7^{-1}
0x0	0xD	0x5	0xC	0x0	0x5	0x8	0xF	0x3
0x1	0x3	0x8	0x9	0x9	0x0	0xF	0xA	0x0
0x2	0xB	0x2	0xF	0xA	0x8	0x2	0x1	0x6
0x3	0x0	0xE	0x4	0x7	0x3	0x9	0xD	0xD
0x4	0xA	0xF	0xB	0xB	0xA	0x4	0x5	0x9
0x5	0x6	0x6	0xE	0xE	0x9	0x1	0x3	0xE
0x6	0x5	0xC	0x1	0x6	0x7	0xD	0x6	0xF
0x7	0xC	0x3	0x2	0xD	0xE	0xE	0x0	0x8
0x8	0x1	0xB	0x0	0x3	0x2	0xB	0x4	0x5
0x9	0xE	0x4	0x3	0x5	0xC	0x6	0x9	0xC
0xA	0x4	0x7	0x6	0xC	0xB	0x5	0xE	0xB
0xB	0x7	0x9	0xD	0x2	0x6	0x3	0x7	0x7
0xC	0xF	0x1	0x5	0x4	0x4	0x7	0x2	0xA
0xD	0x9	0xD	0x8	0x8	0xF	0xC	0xC	0x1
0xE	0x8	0xA	0xA	0xF	0xD	0xA	0x8	0x4
0xF	0x2	0x0	0x7	0x1	0x1	0x0	0xB	0x2

Figura 2.13. Cajas-S_i inversas en paralelo



– *Mezcla de clave de ronda:* Es la misma operación que en el cifrador (Ver Figura 2.5).

Permutación inicial inversa

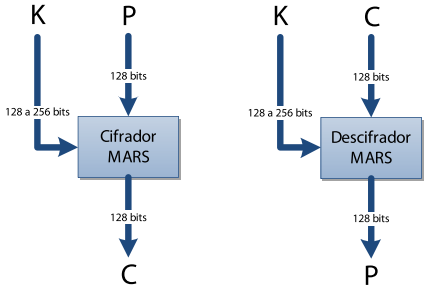
Como su nombre lo indica, es la permutación inversa a la *permutación final*, y por tanto corresponde exactamente a la *permutación inicial* (ver Tabla 2.3 y Figura 2.8).

Algoritmo de cifrado MARS

El algoritmo de cifrado MARS fue diseñado por la empresa IBM, específicamente por: Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M. Matyas Jr., Luke O'Connor, Mohammad Peyravian, David Safford y Nevenko Zunic (Burwick y otros, 1999).

El algoritmo de cifrado MARS toma como entrada y produce como salida datos de cuatro palabras de 32 bits cada una, es decir un bloque de 128 bits. Maneja claves variables de 128 hasta 256 bits y presenta una estructura basada en redes Feistel tipo 3 (Menezes y otros, 1996). La Figura 3.1 presenta las convenciones seguidas en el diseño del cifrador MARS.

Figura 3.1. Convenciones para el cifrado y descifrado MARS



Cifrador MARS

El algoritmo de cifrado MARS consta de 32 rondas, las cuales se reparten de la siguiente manera:

- Adición de subclaves.
- 8 rondas en la fase denominada *mezcla anterior*.
- 16 rondas en la fase de *núcleo criptográfico*, que presenta dos modos: *transformación anterior* y *transformación posterior*.

– 8 rondas en la fase de *mezcla posterior*.
– Sustracción de subclaves.

En la Figura 3.2 se presenta un esquema del diagrama de bloques del cifrado MARS y en la Figura 3.3 se puede apreciar el diagrama de flujo del algoritmo de cifrado MARS. Se ha denotado con ①, ② y ③ las secciones correspondientes entre el diagrama de bloques y el algoritmo. El descifrado se puede apreciar en la sección 3.3.

Figura 3.2. Esquema básico del cifrador MARS

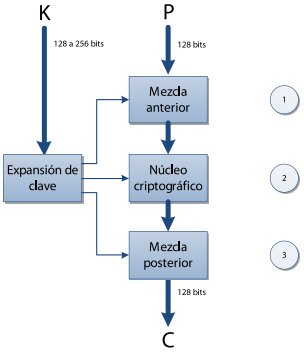
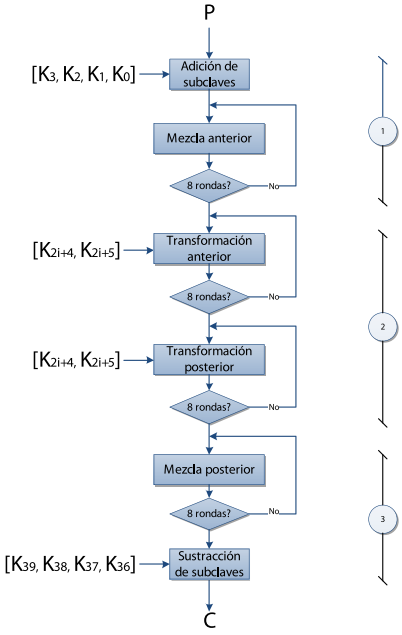


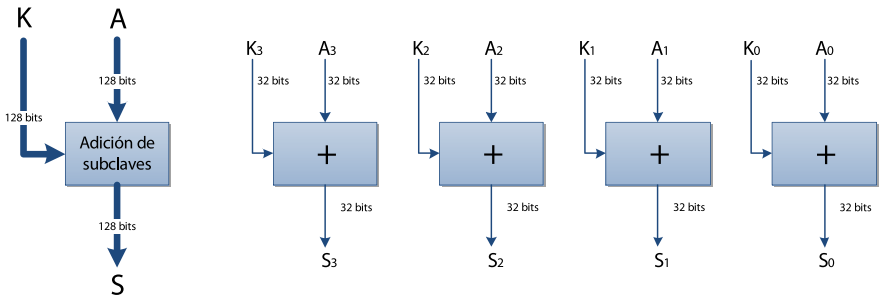
Figura 3.3. Algoritmo de cifrado MARS



Adición de subclaves

Se toma un bloque de texto plano de 128 bits agrupado en 4 palabras de 32 bits cada uno, y se realiza la adición módulo 2^{32} con las primeras 4 subclaves (de 32 bits cada una). Figura 3.4 muestra el esquema del diseño de este bloque de adición.

Figura 3.4. Adición de subclaves



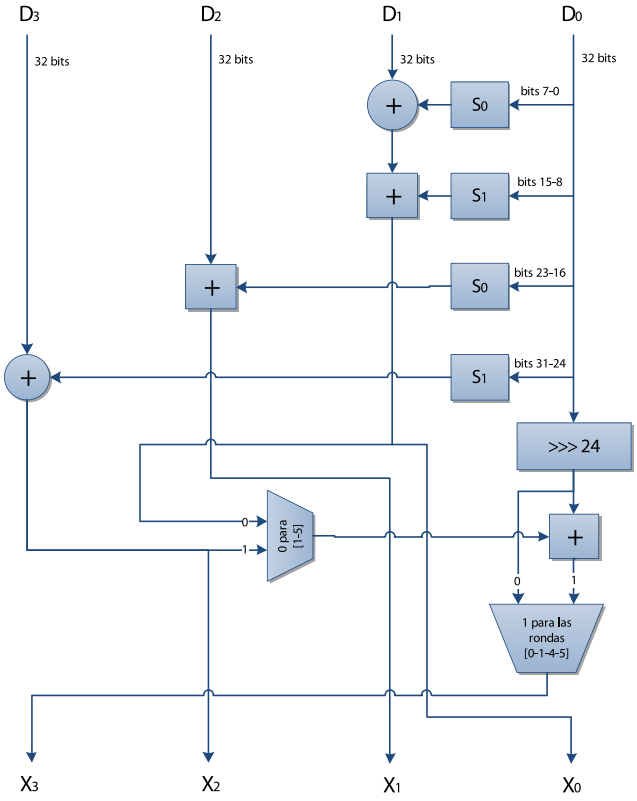
Mezcla anterior

La mezcla anterior consta de 8 rondas de un bloque mezclador Feistel tipo 3, combinado con algunas operaciones de mezcla adicionales.

En cada ronda se utiliza una palabra de 32 bits (palabra fuente) para modificar otras tres palabras de 32 bits (palabras objetivo). Para modificar las palabras objetivo se utilizan los bloques de suma modular de 2^{32} bits, *XOR*, Cajas-S_i, y rotación cíclica a la derecha de 24 bits. Las ecuaciones 3.1 a 3.8 describen el proceso de *mezcla anterior* para cada ronda y la Figura 3.5 muestra el esquema de la misma.

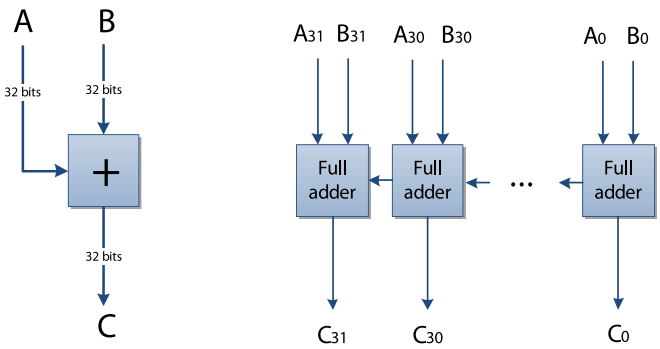
$$D_1 = D_1 \oplus S_0(byte\ 1\ de\ D_0) \quad (3.1)$$
$$D_1 = D_1 + S_1(byte\ 2\ de\ D_0) \quad (3.2)$$
$$D_2 = D_2 + S_0(byte\ 3\ de\ D_0) \quad (3.3)$$
$$D_3 = D_3 \oplus S_1(byte\ 4\ de\ D_0) \quad (3.4)$$
$$D_0 = D_0 >>> 24 \quad (3.5)$$
$$D_0 = D_0 + D_3\ (\text{Sólo para rondas 0 y 4}) \quad (3.6)$$
$$D_0 = D_0 + D_1\ (\text{Sólo para rondas 1 y 5}) \quad (3.7)$$
$$[D_3\ D_2\ D_1\ D_0] = [D_0\ D_3\ D_2\ D_1] \quad (3.8)$$

Figura 3.5. Mezcla anterior



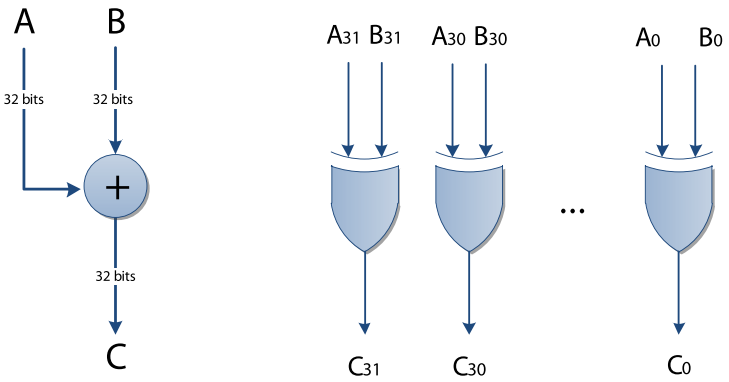
– *Suma módulo* 2^{32}: Es un bloque *full-adder* de 32 bits, sin *carry* de salida. La Figura 3.6 muestra un esquema de este bloque.

Figura 3.6. Suma módulo 2^{32}



– *XOR:* Es un bloque que realiza la operación lógica *XOR*, bit a bit, entre cada una de sus entradas. La Figura 3.7 muestra un esquema de este bloque.

Figura 3.7. XOR de 32 bits



– *Cajas-S_i:* Hay una Caja-S que se encuentra predeterminada y consta de 512 datos de 32 bits. Por simplicidad de trabajo y manejo en algunas fases del algoritmo, esta Caja-S se divide en dos: Caja-S_0 y Caja-S_1. Cada una de estas cajas tienen 256 datos de 32 bits cada una. La Caja-S_0 corresponde a los primeros 256 datos de la Caja-S, y la Caja-S_1, a los últimos 256 datos. Los valores de las Cajas-S_i son presentados en la Tabla 3.1 y 3.2, y la Figura 3.8 muestra un esquema de este bloque.

Figura 3.8. Caja-S y Cajas-S_i

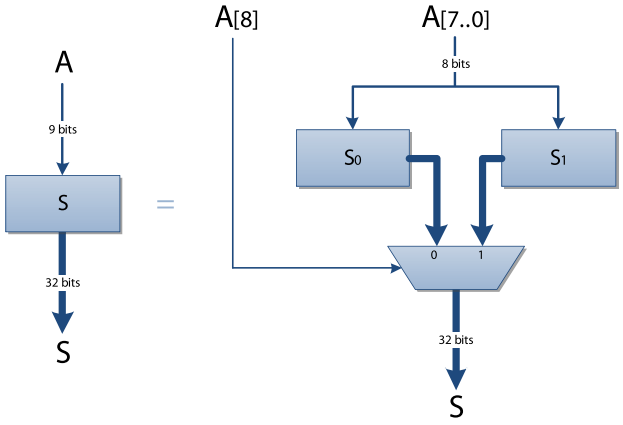


Tabla 3.1. Caja-S_0

Entrada	Salida	Entrada	Salida	Entrada	Salida	Entrada	Salida
0x00	0x09D0C479	**0x40**	0xA4CCAE59	**0x80**	0x3AFD7D3E	**0xC0**	0x5941792D
0x01	0x28C8FFE0	**0x41**	0x3798670D	**0x81**	0xD2F29E01	**0xC1**	0xFA90C1F8
0x02	0x84AA6C39	**0x42**	0xCBFA9493	**0x82**	0x29A9D1F6	**0xC2**	0x33F824B4
0x03	0x9DAD7287	**0x43**	0x4F481D45	**0x83**	0xEFB10C53	**0xC3**	0xC4965372
0x04	0x7DFF9BE3	**0x44**	0xEAFC8CA8	**0x84**	0xCF3B870F	**0xC4**	0x3FF6D550
0x05	0xD4268361	**0x45**	0xDB1129D6	**0x85**	0xB414935C	**0xC5**	0x4CA5FEC0
0x06	0xC96DA1D4	**0x46**	0xB0449E20	**0x86**	0x664465ED	**0xC6**	0x8630E964
0x07	0x7974CC93	**0x47**	0x0F5407FB	**0x87**	0x024ACAC7	**0xC7**	0x5B3FBBD6
0x08	0x85D0582E	**0x48**	0x6167D9A8	**0x88**	0x59A744C1	**0xC8**	0x7DA26A48
0x09	0x2A4B5705	**0x49**	0xD1F45763	**0x89**	0x1D2936A7	**0xC9**	0xB203231A
0x0A	0x1CA16A62	**0x4A**	0x4DAA96C3	**0x8A**	0xDC580AA6	**0xCA**	0x04297514
0x0B	0xC3BD279D	**0x4B**	0x3BEC5958	**0x8B**	0xCF574CA8	**0xCB**	0x2D639306
0x0C	0x0F1F25E5	**0x4C**	0xABABA014	**0x8C**	0x040A7A10	**0xCC**	0x2EB13149
0x0D	0x5160372F	**0x4D**	0xB6CCD201	**0x8D**	0x6CD81807	**0xCD**	0x16A45272
0x0E	0xC695C1FB	**0x4E**	0x38D6279F	**0x8E**	0x8A98BE4C	**0xCE**	0x532459A0
0x0F	0x4D7FF1E4	**0x4F**	0x02682215	**0x8F**	0xACCEA063	**0xCF**	0x8E5F4872
0x10	0xAE5F6BF4	**0x50**	0x8F376CD5	**0x90**	0xC33E92B5	**0xD0**	0xF966C7D9
0x11	0x0D72EE46	**0x51**	0x092C237E	**0x91**	0xD1E0E03D	**0xD1**	0x07128DC0
0x12	0xFF23DE8A	**0x52**	0xBFC56593	**0x92**	0xB322517E	**0xD2**	0x0D44DB62
0x13	0xB1CF8E83	**0x53**	0x32889D2C	**0x93**	0x2092BD13	**0xD3**	0xAFC8D52D
0x14	0xF14902E2	**0x54**	0x854B3E95	**0x94**	0x386B2C4A	**0xD4**	0x06316131
0x15	0x3E981E42	**0x55**	0x05BB9B43	**0x95**	0x52E8DD58	**0xD5**	0xD838E7CE
0x16	0x8BF53EB6	**0x56**	0x7DCD5DCD	**0x96**	0x58656DFB	**0xD6**	0x1BC41D00
0x17	0x7F4BF8AC	**0x57**	0xA02E926C	**0x97**	0x50820371	**0xD7**	0x3A2E8C0F
0x18	0x83631F83	**0x58**	0xFAE527E5	**0x98**	0x41811896	**0xD8**	0xEA83837E
0x19	0x25970205	**0x59**	0x36A1C330	**0x99**	0xE337EF7E	**0xD9**	0xB984737D
0x1A	0x76AFE784	**0x5A**	0x3412E1AE	**0x9A**	0xD39FB119	**0xDA**	0x13BA4891
0x1B	0x3A7931D4	**0x5B**	0xF257F462	**0x9B**	0xC97F0DF6	**0xDB**	0xC4F8B949
0x1C	0x4F846450	**0x5C**	0x3C4F1D71	**0x9C**	0x68FEA01B	**0xDC**	0xA6D6ACB3
0x1D	0x5C64C3F6	**0x5D**	0x30A2E809	**0x9D**	0xA150A6E5	**0xDD**	0xA215CDCE
0x1E	0x210A5F18	**0x5E**	0x68E5F551	**0x9E**	0x55258962	**0xDE**	0x8359838B

Entrada	Salida	Entrada	Salida	Entrada	Salida	Entrada	Salida
0x1F	0xC6986A26	**0x5F**	0x9C61BA44	**0x9F**	0xEB6FF41B	**0xDF**	0x6BD1AA31
0x20	0x28F4E826	**0x60**	0x5DED0AB8	**0xA0**	0xD7C9CD7A	**0xE0**	0xF579DD52
0x21	0x3A60A81C	**0x61**	0x75CE09C8	**0xA1**	0xA619CD9E	**0xE1**	0x21B93F93
0x22	0xD340A664	**0x62**	0x9654F93E	**0xA2**	0xBCF09576	**0xE2**	0xF5176781
0x23	0x7EA820C4	**0x63**	0x698C0CCA	**0xA3**	0x2672C073	**0xE3**	0x187DFDDE
0x24	0x526687C5	**0x64**	0x243CB3E4	**0xA4**	0xF003FB3C	**0xE4**	0xE94AEB76
0x25	0x7EDDD12B	**0x65**	0x2B062B97	**0xA5**	0x4AB7A50B	**0xE5**	0x2B38FD54
0x26	0x32A11D1D	**0x66**	0x0F3B8D9E	**0xA6**	0x1484126A	**0xE6**	0x431DE1DA
0x27	0x9C9EF086	**0x67**	0x00E050DF	**0xA7**	0x487BA9B1	**0xE7**	0xAB394825
0x28	0x80F6E831	**0x68**	0xFC5D6166	**0xA8**	0xA64FC9C6	**0xE8**	0x9AD3048F
0x29	0xAB6F04AD	**0x69**	0xE35F9288	**0xA9**	0xF6957D49	**0xE9**	0xDFEA32AA
0x2A	0x56FB9B53	**0x6A**	0xC079550D	**0xAA**	0x38B06A75	**0xEA**	0x659473E3
0x2B	0x8B2E095C	**0x6B**	0x0591AEE8	**0xAB**	0xDD805FCD	**0xEB**	0x623F7863
0x2C	0xB68556AE	**0x6C**	0x8E531E74	**0xAC**	0x63D094CF	**0xEC**	0xF3346C59
0x2D	0xD2250B0D	**0x6D**	0x75FE3578	**0xAD**	0xF51C999E	**0xED**	0xAB3AB685
0x2E	0x294A7721	**0x6E**	0x2F6D829A	**0xAE**	0x1AA4D343	**0xEE**	0x3346A90B
0x2F	0xE21FB253	**0x6F**	0xF60B21AE	**0xAF**	0xB8495294	**0xEF**	0x6B56443E
0x30	0xAE136749	**0x70**	0x95E8EB8D	**0xB0**	0xCE9F8E99	**0xF0**	0xC6DE01F8
0x31	0xE82AAE86	**0x71**	0x6699486B	**0xB1**	0xBFFCD770	**0xF1**	0x8D421FC0
0x32	0x93365104	**0x72**	0x901D7D9B	**0xB2**	0xC7C275CC	**0xF2**	0x9B0ED10C
0x33	0x99404A66	**0x73**	0xFD6D6E31	**0xB3**	0x378453A7	**0xF3**	0x88F1A1E9
0x34	0x78A784DC	**0x74**	0x1090ACEF	**0xB4**	0x7B21BE33	**0xF4**	0x54C1F029
0x35	0xB69BA84B	**0x75**	0xE0670DD8	**0xB5**	0x397F41BD	**0xF5**	0x7DEAD57B
0x36	0x04046793	**0x76**	0xDAB2E692	**0xB6**	0x4E94D131	**0xF6**	0x8D7BA426
0x37	0x23DB5C1E	**0x77**	0xCD6D4365	**0xB7**	0x92CC1F98	**0xF7**	0x4CF5178A
0x38	0x46CAE1D6	**0x78**	0xE5393514	**0xB8**	0x5915EA51	**0xF8**	0x551A7CCA
0x39	0x2FE28134	**0x79**	0x3AF345F0	**0xB9**	0x99F861B7	**0xF9**	0x1A9A5F08
0x3A	0x5A223942	**0x7A**	0x6241FC4D	**0xBA**	0xC9980A88	**0xFA**	0xFCD651B9
0x3B	0x1863CD5B	**0x7B**	0x460DA3A3	**0xBB**	0x1D74FD5F	**0xFB**	0x25605182
0x3C	0xC190C6E3	**0x7C**	0x7BCF3729	**0xBC**	0xB0A495F8	**0xFC**	0xE11FC6C3
0x3D	0x07DFB846	**0x7D**	0x8BF1D1E0	**0xBD**	0x614DEED0	**0xFD**	0xB6FD9676
0x3E	0x6EB88816	**0x7E**	0x14AAC070	**0xBE**	0xB5778EEA	**0xFE**	0x337B3027
0x3F	0x2D0DCC4A	**0x7F**	0x1587ED55	**0xBF**	0x5941792D	**0xFF**	0xB7C8EB14

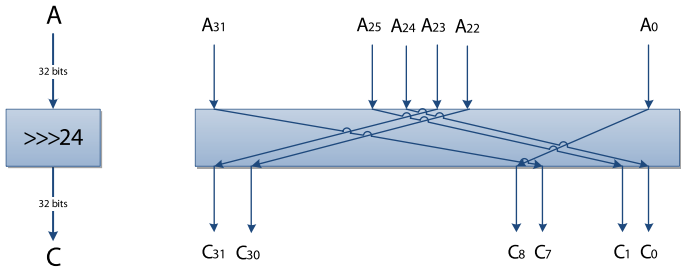
Tabla 3.2. Caja-S_1

Entrada	Salida	Entrada	Salida	Entrada	Salida	Entrada	Salida
0x00	0x9E5FD030	**0x40**	0x2F057D9F	**0x80**	0xCE092EE5	**0xC0**	0xA2C52D53
0x01	0x6B57E354	**0x41**	0x690624F8	**0x81**	0x01E87DA6	**0xC1**	0x42157ABE
0x02	0xAD913CF7	**0x42**	0x1CB0BAFD	**0x82**	0x6CE91E6A	**0xC2**	0xA2253E2E
0x03	0x7E16688D	**0x43**	0x7B0DBDC6	**0x83**	0xBB7BCC84	**0xC3**	0x7BF3F4AE
0x04	0x58872A69	**0x44**	0x810F23BB	**0x84**	0xC7922C20	**0xC4**	0x80F594F9
0x05	0x2C2FC7DF	**0x45**	0xFA929A1A	**0x85**	0x9D3B71FD	**0xC5**	0x953194E7
0x06	0xE389CCC6	**0x46**	0x6D969A17	**0x86**	0x060E41C6	**0xC6**	0x77EB92ED
0x07	0x30738DF1	**0x47**	0x6742979B	**0x87**	0xD7590F15	**0xC7**	0xB3816930
0x08	0x0824A734	**0x48**	0x74AC7D05	**0x88**	0x4E03BB47	**0xC8**	0xDA8D9336
0x09	0xE1797A8B	**0x49**	0x010E65C4	**0x89**	0x183C198E	**0xC9**	0xBF447469
0x0A	0xA4A8D57B	**0x4A**	0x86A3D963	**0x8A**	0x63EEB240	**0xCA**	0xF26D9483
0x0B	0x5B5D193B	**0x4B**	0xF907B5A0	**0x8B**	0x2DDBF49A	**0xCB**	0xEE6FAED5
0x0C	0xC8A8309B	**0x4C**	0xD0042BD3	**0x8C**	0x6D5CBA54	**0xCC**	0x71371235
0x0D	0x73F9A978	**0x4D**	0x158D7D03	**0x8D**	0x923750AF	**0xCD**	0xDE425F73
0x0E	0x73398D32	**0x4E**	0x287A8255	**0x8E**	0xF9E14236	**0xCE**	0xB4E59F43
0x0F	0x0F59573E	**0x4F**	0xBBA8366F	**0x8F**	0x7838162B	**0xCF**	0x7DBE2D4E
0x10	0xE9DF2B03	**0x50**	0x096EDC33	**0x90**	0x59726C72	**0xD0**	0x2D37B185
0x11	0xE8A5B6C8	**0x51**	0x21916A7B	**0x91**	0x81B66760	**0xD1**	0x49DC9A63
0x12	0x848D0704	**0x52**	0x77B56B86	**0x92**	0xBB2926C1	**0xD2**	0x98C39D98
0x13	0x98DF93C2	**0x53**	0x951622F9	**0x93**	0x48A0CE0D	**0xD3**	0x1301C9A2
0x14	0x720A1DC3	**0x54**	0xA6C5E650	**0x94**	0xA6C0496D	**0xD4**	0x389B1BBF
0x15	0x684F259A	**0x55**	0x8CEA17D1	**0x95**	0xAD43507B	**0xD5**	0x0C18588D
0x16	0x943BA848	**0x56**	0xCD8C62BC	**0x96**	0x718D496A	**0xD6**	0xA421C1BA
0x17	0xA6370152	**0x57**	0xA3D63433	**0x97**	0x9DF057AF	**0xD7**	0x7AA3865C
0x18	0x863B5EA3	**0x58**	0x358A68FD	**0x98**	0x44B1BDE6	**0xD8**	0x71E08558
0x19	0xD17B978B	**0x59**	0x0F9B9D3C	**0x99**	0x054356DC	**0xD9**	0x3C5CFCAA
0x1A	0x6D9B58EF	**0x5A**	0xD6AA295B	**0x9A**	0xDE7CED35	**0xDA**	0x7D239CA4
0x1B	0x0A700DD4	**0x5B**	0xFE33384A	**0x9B**	0xD51A138B	**0xDB**	0x0297D9DD
0x1C	0xA73D36BF	**0x5C**	0xC000738E	**0x9C**	0x62088CC9	**0xDC**	0xD7DC2830
0x1D	0x8E6A0829	**0x5D**	0xCD67EB2F	**0x9D**	0x35830311	**0xDD**	0x4B37802B
0x1E	0x8695BC14	**0x5E**	0xE2EB6DC2	**0x9E**	0xC96EFCA2	**0xDE**	0x7428AB54
0x1F	0xE35B3447	**0x5F**	0x97338B02	**0x9F**	0x686F86EC	**0xDF**	0xAEEE0347

Entrada	Salida	Entrada	Salida	Entrada	Salida	Entrada	Salida
0x20	0x933AC568	**0x60**	0x06C9F246	**0xA0**	0x8E77CB68	**0xE0**	0x4B3FBB85
0x21	0x8894B022	**0x61**	0x419CF1AD	**0xA1**	0x63E1D6B8	**0xE1**	0x692F2F08
0x22	0x2F511C27	**0x62**	0x2B83C045	**0xA2**	0xC80F9778	**0xE2**	0x134E578E
0x23	0xDDFBCC3C	**0x63**	0x3723F18A	**0xA3**	0x79C491FD	**0xE3**	0x36D9E0BF
0x24	0x006662B6	**0x64**	0xCB5B3089	**0xA4**	0x1B4C67F2	**0xE4**	0xAE8B5FCF
0x25	0x117C83FE	**0x65**	0x160BEAD7	**0xA5**	0x72698D7D	**0xE5**	0xEDB93ECF
0x26	0x4E12B414	**0x66**	0x5D494656	**0xA6**	0x5E368C31	**0xE6**	0x2B27248E
0x27	0xC2BCA766	**0x67**	0x35F8A74B	**0xA7**	0xF7D95E2E	**0xE7**	0x170EB1EF
0x28	0x3A2FEC10	**0x68**	0x1E4E6C9E	**0xA8**	0xA1D3493F	**0xE8**	0x7DC57FD6
0x29	0xF4562420	**0x69**	0x000399BD	**0xA9**	0xDCD9433E	**0xE9**	0x1E760F16
0x2A	0x55792E2A	**0x6A**	0x67466880	**0xAA**	0x896F1552	**0xEA**	0xB1136601
0x2B	0x46F5D857	**0x6B**	0xB4174831	**0xAB**	0x4BC4CA7A	**0xEB**	0x864E1B9B
0x2C	0xCEDA25CE	**0x6C**	0xACF423B2	**0xAC**	0xA6D1BAF4	**0xEC**	0xD7EA7319
0x2D	0xC3601D3B	**0x6D**	0xCA815AB3	**0xAD**	0xA5A96DCC	**0xED**	0x3AB871BD
0x2E	0x6C00AB46	**0x6E**	0x5A6395E7	**0xAE**	0x0BEF8B46	**0xEE**	0xCFA4D76F
0x2F	0xEFAC9C28	**0x6F**	0x302A67C5	**0xAF**	0xA169FDA7	**0xEF**	0xE31BD782
0x30	0xB3C35047	**0x70**	0x8BDB446B	**0xB0**	0x74DF40B7	**0xF0**	0x0DBEB469
0x31	0x611DFEE3	**0x71**	0x108F8FA4	**0xB1**	0x4E208804	**0xF1**	0xABB96061
0x32	0x257C3207	**0x72**	0x10223EDA	**0xB2**	0x9A756607	**0xF2**	0x5370F85D
0x33	0xFDD58482	**0x73**	0x92B8B48B	**0xB3**	0x038E87C8	**0xF3**	0xFFB07E37
0x34	0x3B14D84F	**0x74**	0x7F38D0EE	**0xB4**	0x20211E44	**0xF4**	0xDA30D0FB
0x35	0x23BECB64	**0x75**	0xAB2701D4	**0xB5**	0x8B7AD4BF	**0xF5**	0xEBC977B6
0x36	0xA075F3A3	**0x76**	0x0262D415	**0xB6**	0xC6403F35	**0xF6**	0x0B98B40F
0x37	0x088F8EAD	**0x77**	0xAF224A30	**0xB7**	0x1848E36D	**0xF7**	0x3A4D0FE6
0x38	0x07ADF158	**0x78**	0xB3D88ABA	**0xB8**	0x80BDB038	**0xF8**	0xDF4FC26B
0x39	0x7796943C	**0x79**	0xF8B2C3AF	**0xB9**	0x1E62891C	**0xF9**	0x159CF22A
0x3A	0xFACABF3D	**0x7A**	0xDAF7EF70	**0xBA**	0x643D2107	**0xFA**	0xC298D6E2
0x3B	0xC09730CD	**0x7B**	0xCC97D3B7	**0xBB**	0xBF04D6F8	**0xFB**	0x2B78EF6A
0x3C	0xF7679969	**0x7C**	0xE9614B6C	**0xBC**	0x21092C8C	**0xFC**	0x61A94AC0
0x3D	0xDA44E9ED	**0x7D**	0x2BAEBFF4	**0xBD**	0xF644F389	**0xFD**	0xAB561187
0x3E	0x2C854C12	**0x7E**	0x70F687CF	**0xBE**	0x0778404E	**0xFE**	0x14EEA0F0
0x3F	0x35935FA3	**0x7F**	0x386C9156	**0xBF**	0x7B78ADB8	**0xFF**	0xDF0D4164

– *Rotación cíclica a la derecha, 24 bits:* Este bloque realiza una rotación cíclica de 24 bits a la derecha de un dato de 32 bits. Es decir, el bit en la posición 24, quedará a la salida en el bit 0; el bit 25 en el bit 1 y así sucesivamente. Como se trata de una rotación cíclica, el bit 23 quedará a la salida en el bit 31; el bit 22 en el 30; y así. La Figura 3.9 muestra el esquema de esta rotación.

Figura 3.9. Rotación cíclica a la derecha de 24 bits



Núcleo criptográfico

El *núcleo criptográfico* del algoritmo de cifrado MARS es una red Feistel tipo 3, que consiste de 16 rondas de transformación: 8 en modo anterior y 8 en modo posterior. En cada vuelta se utiliza una función de expansión con clave, denominada E, bloques de suma modular de 2^{32} bits, XOR, y rotación cíclica a la derecha de 13 bits

Las Figuras 3.10 y 3.11 presentan los diagramas de las transformaciones con clave del núcleo criptográfico del algoritmo de cifrado MARS.

Figura 3.10. Transformación con clave en modo anterior

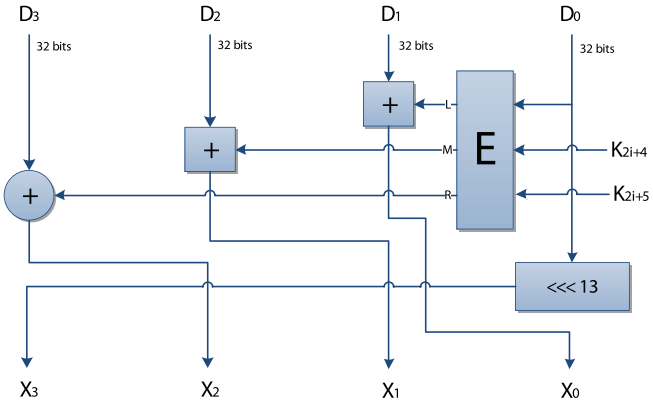
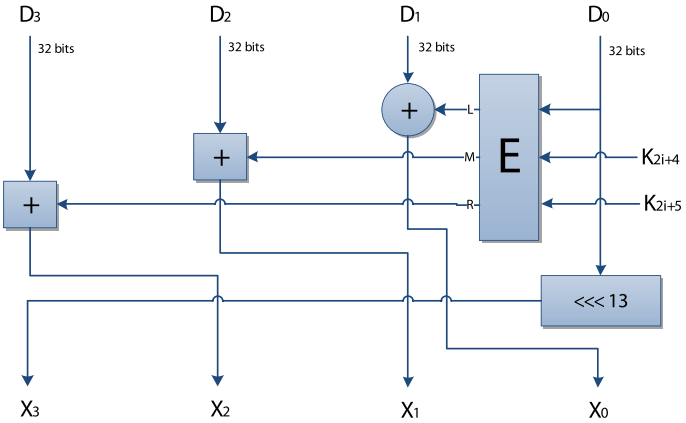


Figura 3.11. Transformación con clave en modo posterior



– *Suma módulo 2^{32}:* Es un bloque *full-adder* de 32 bits, sin *carry* de salida. Se presenta en la Figura 3.6.

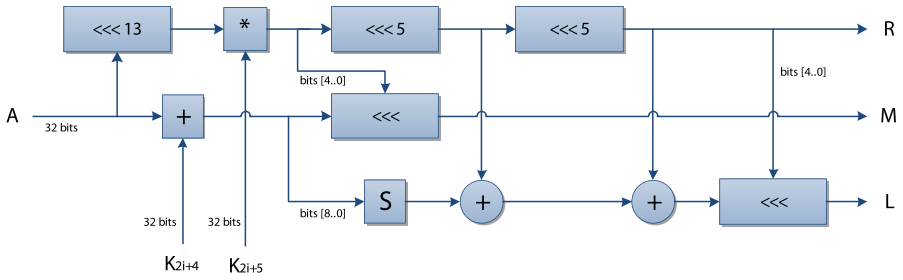
– *XOR:* Es un bloque que realiza la operación lógica *xor*, bit a bit, entre cada una de sus entradas. Se presenta en la Figura 3.7.

– *Rotación cíclica a la izquierda, 13 bits:* Este bloque realiza una rotación cíclica de 13 bits a la izquierda de un dato de 32 bits. Es decir, el bit en la posición 0, quedará a la salida en el bit 13; el bit 1 en el bit 14 y así sucesivamente. Como se trata de una rotación cíclica, el bit 31 quedará a la salida en el bit 12; el bit 30 en el 11; y así. La Figura 3.17 ilustra cómo se logra esta permutación.

– *Función E:* Esta función de expansión se basa en una combinación de multiplicaciones, rotaciones y reemplazo a través de la Caja-S. Se expanden 32 bits en 96 bits. En las ecuaciones 3.9 a 3.17 se describe esta función, y en la Figura 3.12 se puede apreciar el diagrama de la misma.

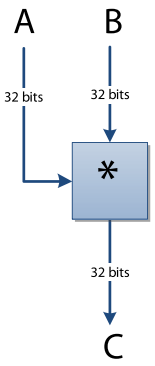
$$M = A + K_{2i+4} \quad (3.9)$$
$$R = (A <<< 13 \oplus K_{2i+5} \quad (3.10)$$
$$L = S(9\ bits\ bajos\ de\ M) \quad (3.11)$$
$$R = R <<< 5 \quad (3.12)$$
$$M = M <<< (5\ bits\ bajos\ de\ R) \quad (3.13)$$
$$L = L \oplus R \quad (3.14)$$
$$R = R <<< 5 \quad (3.15)$$
$$L = L \oplus R \quad (3.16)$$
$$L = L <<< (5\ bits\ bajos\ de\ R) \quad (3.17)$$

Figura 3.12. Función de expansión E



– *Multiplicación módulo 2^{32}:* Es un bloque de multiplicación de números enteros en formato *unsigned* que es utilizado al interior de la *Función E*. Usualmente un multiplicador de este tipo presenta el doble de bits a la salida que los operandos de entrada; es decir, si los operandos son de 32 bits, un multiplicador 'normal' deberá presentar 64 bits a la salida. Puesto que este es un multiplicador módulo, la salida sólo presentará los primeros 32 bits. En la Figura 3.13 se puede apreciar un esquema de este bloque.

Figura 3.13. Multiplicador módulo

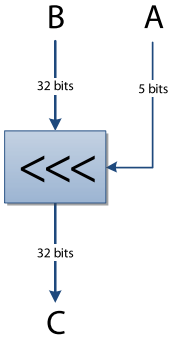


– *Rotación cíclica a la izquierda, 5 bits:* Este bloque realiza una rotación cíclica de 5 bits a la derecha de un dato de 32 bits y es utilizado al interior de la *función E*. Es decir, el bit en la posición 0, quedará a la salida en el bit 5; el bit 1 en el bit 6 y así sucesivamente. Como se trata de una rotación cíclica, el bit 31 quedará a la salida en el bit 4; el bit 30 en el 3; y así. La Figura 3.17 ilustra cómo se logra esta permutación.

– *Rotación cíclica a la izquierda:* Este bloque es utilizado al interior de la *función E* y a diferencia de las rotaciones cíclicas anteriores, cuya cantidad de

rotación es fija, la cantidad de rotación de éste está dado por una entrada adicional, se implementa a manera de *barrel shifter* (Menezes, 1996). En la Figura 3.14 se presenta el diagrama de este bloque.

Figura 3.14. Rotación cíclica a la izquierda



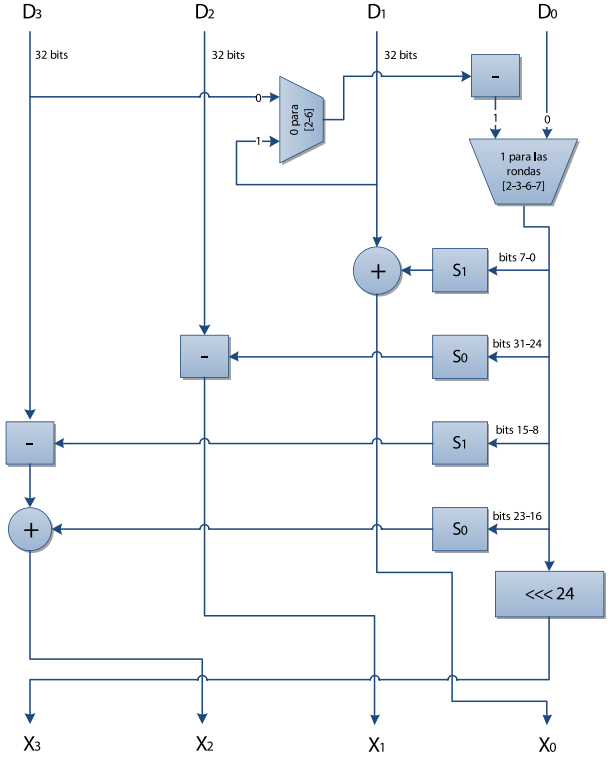
– *Caja-S:* Este bloque utilizado al interior de la *función E*, hace referencia a la Caja-S completa; es decir, a la unión de la Caja-S_0 y la Caja-S_1. Las Cajas-S_i presentan a su entrada 8 bits que se expanden a 32 bits acorde a las relaciones presentadas en las tablas 3.1 y 3.3. La Caja-S presenta un bit adicional en su entrada; y este bit, que es el más significativo, selecciona la Cajas-S_i que se utilizará. El diagrama de este bloque se presenta en la Figura 3.8.

Mezcla posterior

La *mezcla posterior* presenta una estructura similar a la *mezcla anterior*, donde en cada ronda se utiliza una palabra de 32 bits (palabra fuente) para modificar otras tres palabras de 32 bits (palabras objetivo). Las ecuaciones 3.18 a 3.25 describen el proceso de *mezcla posterior* para cada ronda, y la Figura 3.15 muestra el esquema de la misma.

$$D_0 = D_0 - D_3 \text{ (Sólo para rondas 2 y 6)} \quad (3.18)$$
$$D_0 = D_0 - D_1 \text{ (Sólo para rondas 3 y 7)} \quad (3.19)$$
$$D_1 = D_1 \oplus S_1(byte\ 1\ de\ D_0) \quad (3.20)$$
$$D_2 = D_2 - S_0(byte\ 4\ de\ D_0) \quad (3.21)$$
$$D_3 = D_3 - S_1(byte\ 3\ de\ D_0) \quad (3.22)$$
$$D_3 = D_3 \oplus S_0(byte\ 2\ de\ D_0) \quad (3.23)$$
$$D_0 = D_0 <<< 24 \quad (3.24)$$
$$[D_3\ D_2\ D_1\ D_0] = [D_0\ D_3\ D_2\ D_1] \quad (3.25)$$

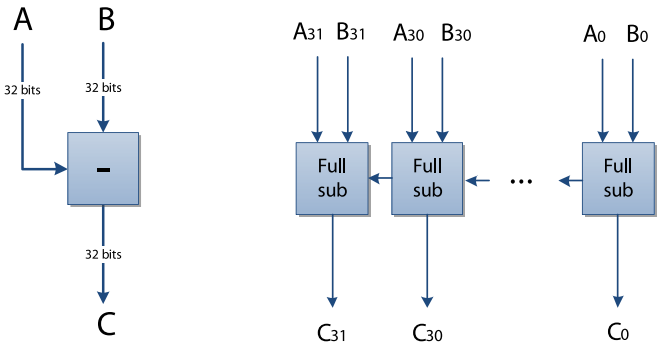
Figura 3.15. Mezcla posterior



Los bloques funcionales son similares a los de la *mezcla anterior*.

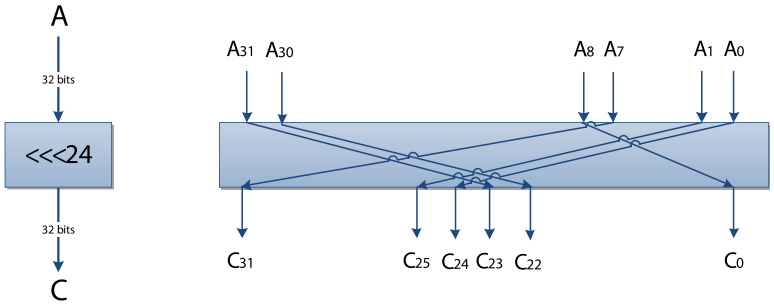
– *Restador módulo* 2^{32}: Es un bloque *full-subtractor* de 32 bits, sin *borrow* de salida. Se presenta en la Figura 3.16.

Figura 3.16. Resta módulo 2^{32}



– *Rotación cíclica a la izquierda, 24 bits:* Este bloque realiza una rotación cíclica de 24 bits a la izquierda de un dato de 32 bits. Es decir, el bit en la posición 0, quedará a la salida en el bit 24; el bit 1 en el bit 25 y así sucesivamente. Como se trata de una rotación cíclica, el bit 31 quedará a la salida en el bit 23; el bit 30 en el 22 y así. La Figura 3.17 muestra como se logra esta permutación.

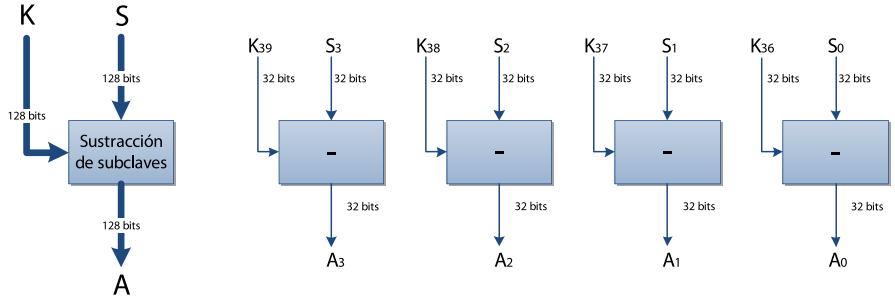
Figura 3.17. Rotación cíclica a la izquierda de 24 bits



Sustracción de subclaves

Se toma un bloque de texto plano de 128 bits agrupado en 4 palabras de 32 bits cada uno, y se realiza la resta módulo 2^{32} con las últimas 4 subclaves (de 32 bits cada una).

Figura 3.18. Sustracción de subclaves



Expansión de clave MARS

El algoritmo de cifrado MARS requiere de 40 palabras clave de 32 bits, denominadas subclaves K_i, para ser utilizadas en la *función E* del proceso de cifrado. Estas subclaves K_i se expanden de la clave K ingresada por el usuario, la cual consiste en palabras de 32 bits, siendo un número entre 4 y 14. El proceso de expansión garantiza que las subclave K_i presenten las siguientes propiedades:

– Los dos bits menos significativos de la palabra clave que se usa para la multiplicación, se ponen en 1.
– Ninguna subclave K_i presenta diez 1s ó 0s consecutivos.

La expansión de clave del algoritmo MARs presenta el siguiente pseudocódigo:

Algoritmo 3.1. Algoritmo de expansión de clave *MARS*

Entrada: $k[\,], n$
Salida: $K[\,]$

$//n$ es el número de palabras de 32 bits en la clave $k[\,]$, $4 \leq n \leq 14$
$//K[\,]$ es el arreglo con las 40 claves extendidas
$//T[\,]$ es un arreglo temporal de 15 palabras
$//B[\,]$ es una tabla fija de 4 palabras

$B[] = \{0xA4A8D57B, 0x5B5D193B, 0xC8A8309B, 0x73F9A978\}$
$T[0 \ldots n-1] = k[0 \ldots n-1]$
$T[n] = n$
$T[n+1 \ldots 14] = 0$

Para $j = 0$ hasta 3 hacer
    Para $i = 0$ hasta 14 hacer
        $T[i] = T[i] \oplus ((T[(i-7) \bmod 15] \oplus T[(i-2) \bmod 15]) <<< 3) \oplus (4i + j)$

    Repetir 4 veces
        Para $i = 0$ hasta 14 hacer
            $T[i] = (T[i] + S[low\ 9\ bits\ of\ T[i \oplus 1 \bmod 15]]) <<< 9$
    Fin repetir

    Para $i = 0$ hasta 9
        $K[10j + i] = T[4i \bmod 15]$
Fin para

Para $i = 5, 7, 9 \ldots 35$ hacer
    $j =$ dos bits menos significativos de $K[i]$
    $w = K[i]$ con los dos bits menos significativos en 1

    $M_l = 1$ $\quad iff\ w_l$ pertenece a una secuencia de 10 ceros o unos consecutivos en w
    y también si $2 \leq l \leq 30$ y $w_{l-1} = w_l = w_{l+1}$

    $r =$ cinco bits menos significativos de $K[i-1]$
    $p = B[j] <<< r$
    $K[i] = w \oplus (p \wedge M)$

Fin para

Descifrador MARS

El algoritmo de descifrado MARS es el proceso inverso al cifrado MARS; por tanto, ambos procesos son similares y comparten muchos de los mismos elementos básicos. El algoritmo de descifrado MARS consta de 32 rondas, las cuales se reparten de la siguiente manera:

- Adición de subclaves.
- 8 rondas en la fase denominada *mezcla posterior inversa*.
- 16 rondas en la fase de *núcleo criptográfico inverso*, que presenta dos modos: *transformación posterior inversa* y *transformación anterior inversa*.
- 8 rondas en la fase de *mezcla anterior inversa*.
- Sustracción de subclaves.

En la Figura 3.19 se presenta un esquema del diagrama de bloques del descifrado MARS, en la Figura 3.20 se puede apreciar el diagrama de flujo del algoritmo de descifrado MARS. Se ha denotado con ①, ② y ③ las secciones correspondientes entre el diagrama de bloques y el algoritmo.

Figura 3.19. Esquema básico del descifrador MARS

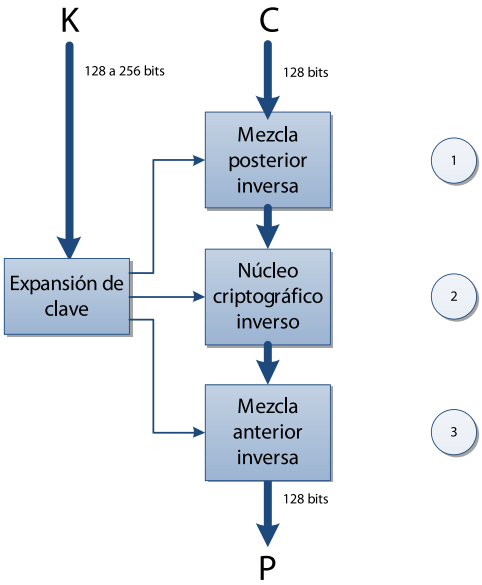
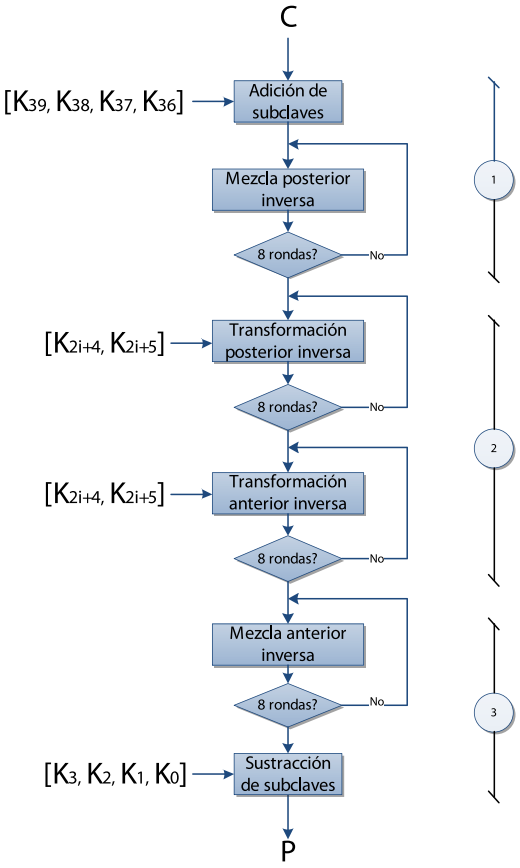


Figura 3.20. Algoritmo de descifrado *MARS*



Adición de subclave

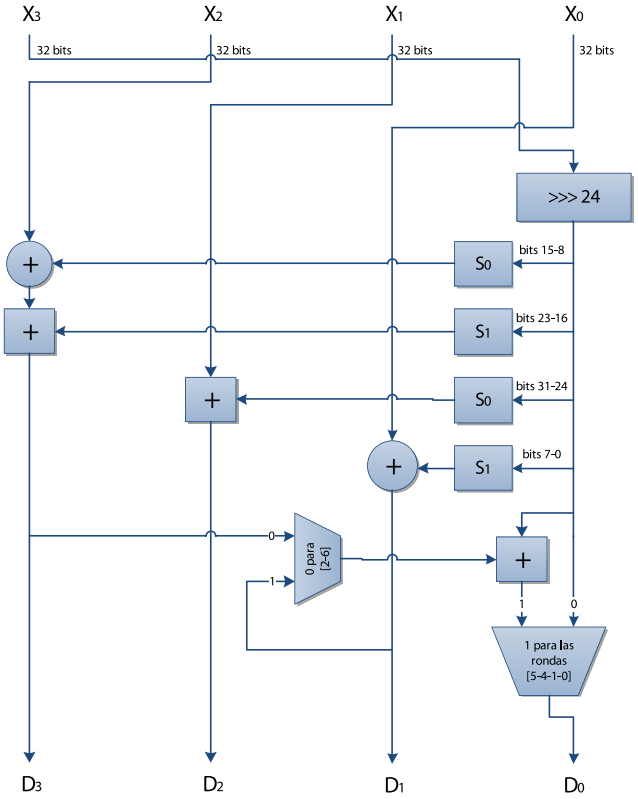
Es exactamente el mismo bloque utilizado en el cifrador (Ver Figura 3.4).

Mezcla posterior inversa

La *mezcla posterior inversa* corresponde a un bloque que realiza la función inversa a la mezcla posterior. La *mezcla posterior inversa* se compone de ocho rondas, en cada ronda se utiliza una palabra de 32 bits (palabra fuente) para modificar otras tres palabras de 32 bits (palabras objetivo). Para modificar las palabras objetivo se utilizan los bloques de suma modular de 2^{32} bits, XOR, Cajas-S_i, y rotación cíclica a la izquierda de 24 bits. Las ecuaciones 3.26 a 3.33 describen el proceso de *mezcla posterior inversa* para cada ronda y la Figura 3.21 muestra el esquema de la misma.

$$[D_3\ D_2\ D_1\ D_0] = [D_2\ D_1\ D_0\ D_3] \quad (3.26)$$
$$D_0 = D_0 <<< 24 \quad (3.27)$$
$$D_3 = D_3 \oplus S_0(byte\ 2\ de\ D_0) \quad (3.28)$$
$$D_3 = D_3 + S_1(byte\ 3\ de\ D_0) \quad (3.29)$$
$$D_2 = D_2 + S_0(byte\ 4\ de\ D_0) \quad (3.30)$$
$$D_1 = D_1 \oplus S_1(byte\ 1\ de\ D_0) \quad (3.31)$$
$$D_0 = D_0 + D_3\ (\text{Sólo para rondas 2 y 6}) \quad (3.32)$$
$$D_0 = D_0 + D_1\ (\text{Sólo para rondas 3 y 7}) \quad (3.33)$$

Figura 3.21. Mezcla posterior inversa



Núcleo criptográfico inverso

El *núcleo criptográfico* del algoritmo de descifrado MARS es una red Feistel tipo 3, que consta de 16 rondas de transformación: 8 en modo posterior y 8 en modo anterior. En cada vuelta se utiliza la misma función de expansión con clave E, bloques de resta modular de 2^{32} bits, XOR, y rotación cíclica a la derecha de 13 bits.

Las Figuras 3.22 y 3.23 presentan los diagramas de las transformaciones con clave del núcleo criptográfico del algoritmo de cifrado *MARS*.

Figura 3.22. Transformación con clave en modo posterior

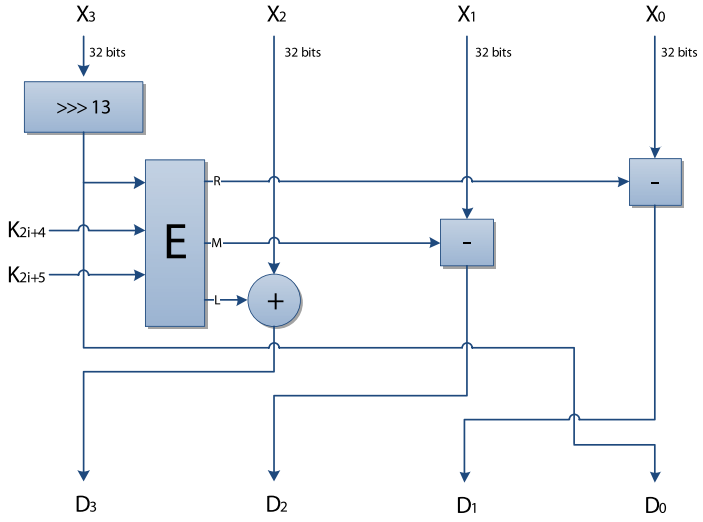
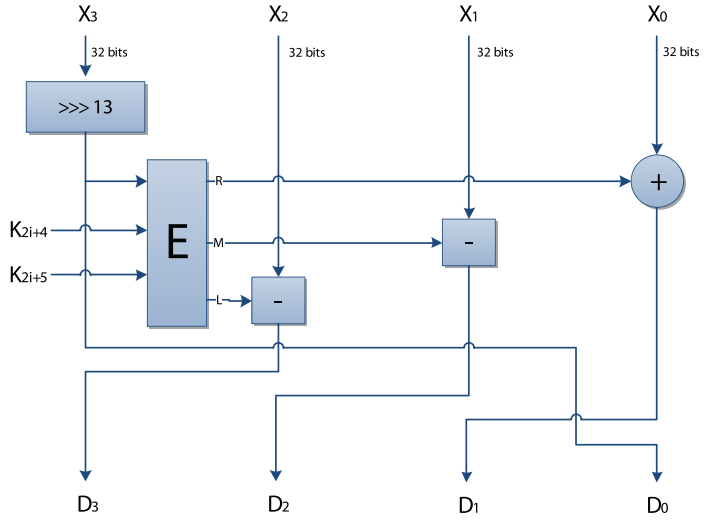


Figura 3.23. Transformación con clave en modo anterior



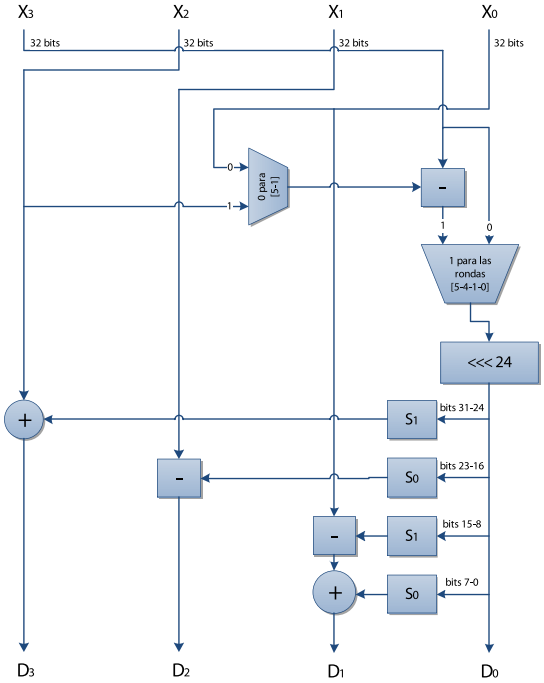
Mezcla anterior inversa

La *mezcla anterior inversa* corresponde a un bloque que realiza la función inversa a la mezcla anterior. La *mezcla anterior inversa* se compone de ocho rondas, en

cada ronda se utiliza una palabra de 32 bits (palabra fuente) para modificar otras tres palabras de 32 bits (palabras objetivo). Para modificar las palabras objetivo se utilizan los bloques de resta modular de 2^{32} bits, XOR, Cajas-S_i, y rotación cíclica a la izquierda de 24 bits. Las ecuaciones 3.34 a 3.41 describen el proceso de *mezcla posterior inversa* para cada ronda y la Figura 3.24 muestra el esquema de la misma.

$$[D_3\ D_2\ D_1\ D_0] = [D_2\ D_1\ D_0\ D_3] \quad (3.34)$$
$$D_0 = D_0 - D_3 \text{ (Sólo para rondas 0 y 4)} \quad (3.35)$$
$$D_0 = D_0 - D_1 \text{ (Sólo para rondas 1 y 5)} \quad (3.36)$$
$$D_0 = D_0 <<< 24 \quad (3.37)$$
$$D_3 = D_3 \oplus S_1(byte\ 4\ de\ D_0) \quad (3.38)$$
$$D_2 = D_2 - S_0(byte\ 3\ de\ D_0) \quad (3.39)$$
$$D_1 = D_1 - S_1(byte\ 2\ de\ D_0) \quad (3.40)$$
$$D_1 = D_1 \oplus S_0(byte\ 1\ de\ D_0) \quad (3.41)$$

Figura 3.24. Mezcla anterior inversa



Sustracción de subclaves

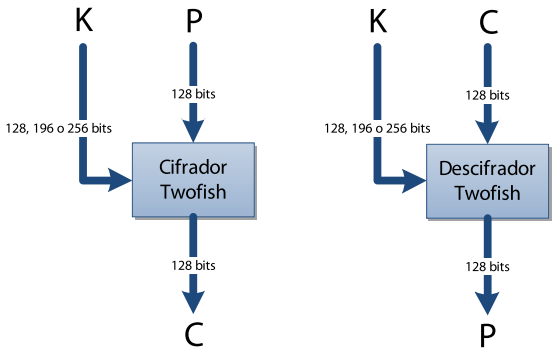
Es exactamente el mismo bloque utilizado en el cifrador (Ver Figura 3.16).

Algoritmo de cifrado Twofish

El algoritmo de cifrado *Twofish* fue diseñado por Bruce Schneier, John Kelseyy, Doug Whitingz, David Wagner, Chris Hall y Niels Ferguson (Schneier y otros, 1998).

Twofish encripta 128 bits de texto plano *P*, en 128 bits de texto cifrado C, a través de 16 rondas controladas por 40 subclaves de 32 bits. La clave de usuario *K* es variable, y puede ser de 128, 192 o 256 bits. La Figura 4.1 presenta las convenciones seguidas en el diseño del cifrador *Twofish*.

Figura 4.1. Convenciones para el cifrado y descifrado *Twofish*



Cifrador *Twofish*

El cifrador de bloque *Twofish* en sí consta de:

– Un *blanqueo de entrada*.
– 16 rondas de mezcla de una red Feistel (Menezes y otros, 1996).
– Un *blanqueo de salida*.

En la Figura 4.2 se presenta un esquema del diagrama de bloques del cifrado *Twofish* y en la Figura 4.3 se puede apreciar el diagrama de flujo del algoritmo de cifrado *Twofish*. El descifrado se puede apreciar en la sección 4.3.

Figura 4.2. Esquema básico del cifrador *Twofish*

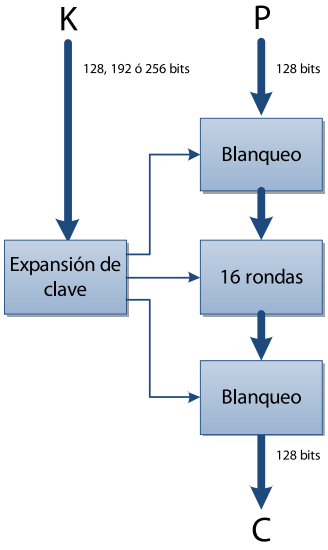
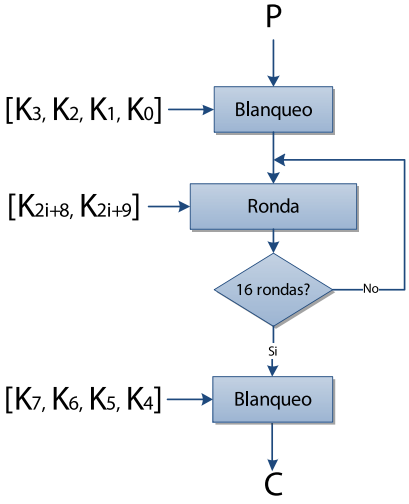


Figura 4.3. Algoritmo de cifrado *Twofish*

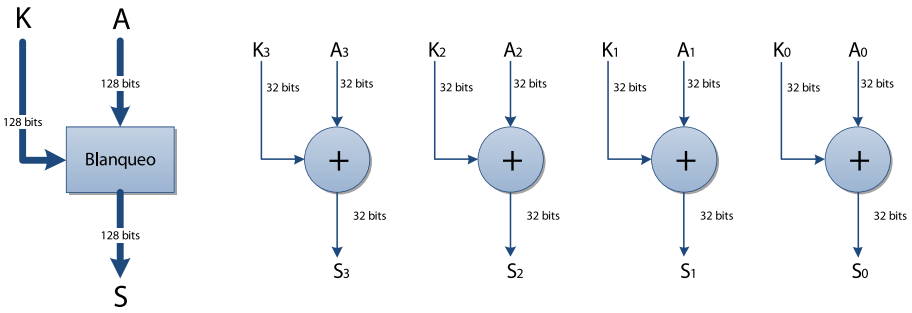


Blanqueo

Se toma un bloque de texto plano de 128 bits agrupado en 4 palabras de 32 bits cada uno, y se realiza la operación *XOR* bit a bit con 4 subclaves (de 32 bits cada

una). Para el caso del blanqueo de entrada las subclaves utilizadas son las K_0, K_1, K_2 y K_3; para el blanqueo de salida las subclaves corresponden a K_4, K_5, K_6 y K_7. La Figura 4.4 muestra un esquema del diseño en hardware de este bloque.

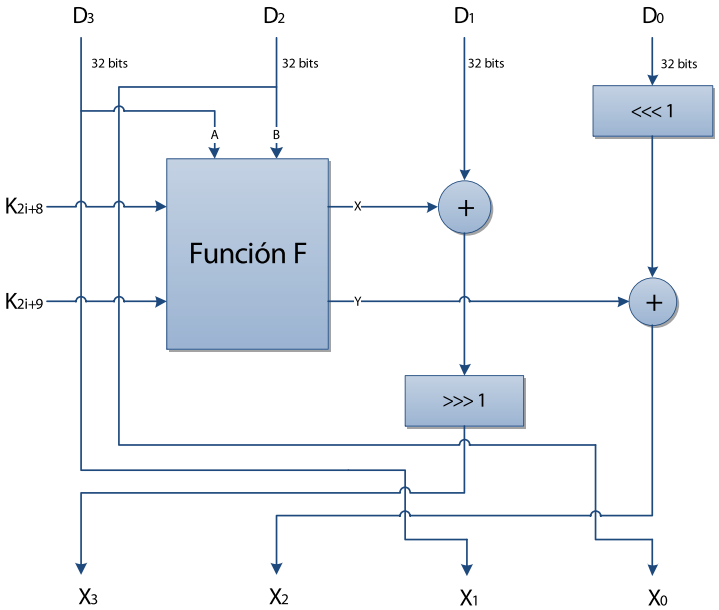
Figura 4.4. Blanqueo



Ronda

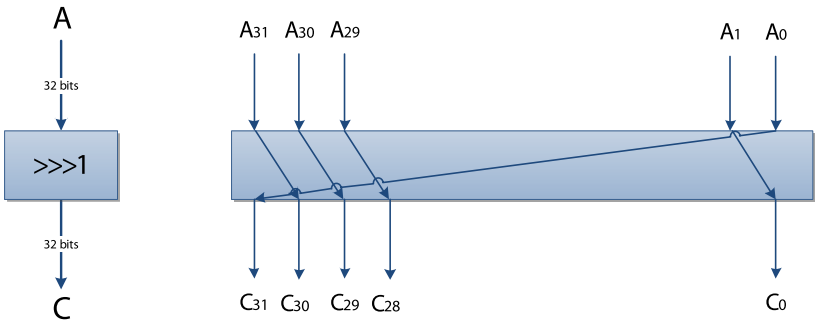
La ronda de cifrado *Twofish* consta de una red Feistel, o *función F* y unas operaciones adicionales de rotación cíclica de un bit y *XOR*, organizadas como se muestra en la Figura 4.5. Esta ronda se repite 16 veces.

Figura 4.5. Ronda *Twofish*



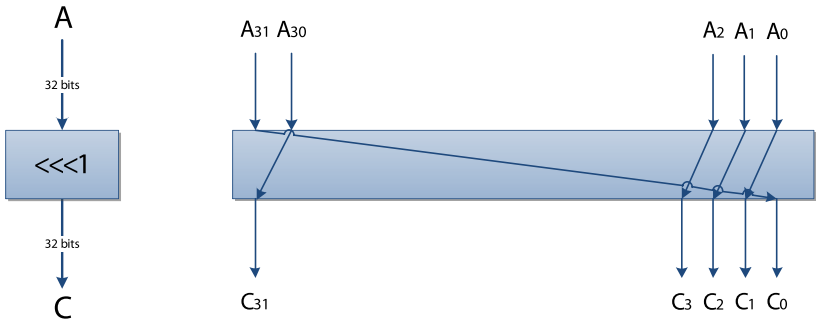
– *Rotación cíclica a la derecha, 1 bit:* Este bloque realiza una rotación cíclica de 1 bit a la derecha de un dato de 32 bits. Es decir, el bit en la posición 1, quedará a la salida en el bit 0; el bit 2 en el bit 1, y así sucesivamente. Como se trata de una rotación cíclica, el bit 0 quedará a la salida en el bit 31; el bit 31 en el 30; y así. La Figura 4.6 muestra el esquema de esta rotación.

Figura 4.6. Rotación cíclica a la derecha de 1 bit



– *Rotación cíclica a la izquierda, 1 unidad:* Este bloque realiza una rotación cíclica de 1 bit a la izquierda de un dato de 32 bits. Es decir, el bit en la posición 0, quedará a la salida en el bit 1; el bit 1 en el bit 2, y así sucesivamente. Como se trata de una rotación cíclica, el bit 31 quedará a la salida en el bit 0; el bit 30 en el 31; y así. La Figura 4.7 muestra el esquema de esta rotación.

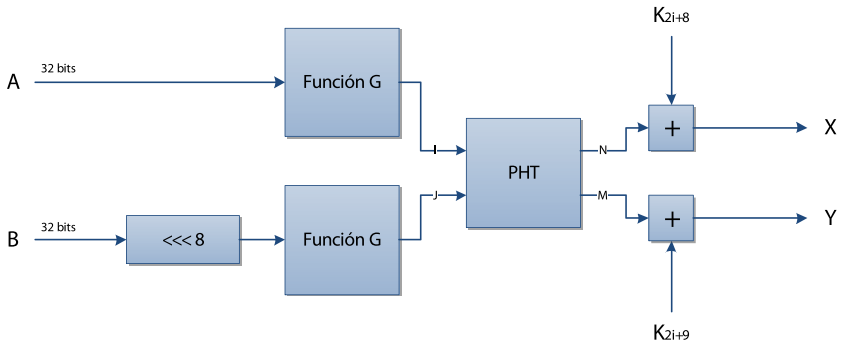
Figura 4.7. Rotación cíclica a la izquierda de 1 bit



– *Función F:* Es el bloque más importante del algoritmo, es una red Feistel y se compone de los siguientes bloques listados a continuación. En la Figura 4.8 se puede apreciar la estructura de este bloque.

- Rotación cíclica de 8 bits a la izquierda.
- Función G.
- Transformada pseudo-Hadamard (PHT).
- Suma módulo 2^{32} .

Figura 4.8. Función F



- *Rotación cíclica a la izquierda, 8 bits:* Este bloque realiza una rotación cíclica fija de 8 bits a la izquierda de un dato de 32 bits. Es decir, el bit en la posición 0 quedará a la salida en el bit 8; el bit 1 en el bit 9, y así sucesivamente. Como se trata de una rotación cíclica, el bit 31 quedará a la salida en el bit 7; el bit 30 en el 6; y así sucesivamente. La Figura 4.7 muestra como se puede implementar esta permutación de bits.

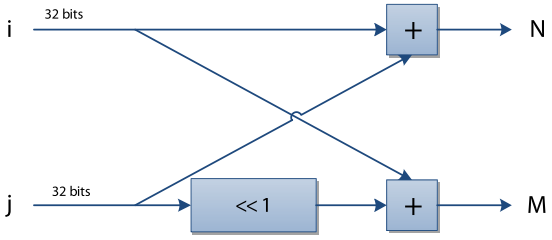
- *Suma módulo 2^{32}:* Es un bloque *full-adder* de 32 bits, sin *carry* de salida. La Figura 3.6 muestra un esquema de este bloque.

- *Transformada pseudo-Hadamard:* La transformada pseudo-Hadamard, ó PHT, corresponde a la implementación de las ecuaciones 4.1 y 4.2, siendo y dos datos de 32 bits cada uno.

$$N = (i + j) \ mod \ 2^{32} \qquad (4.1)$$
$$M = (i + 2 * j) \ mod \ 2^{32} \qquad (4.2)$$

La ronda *Twofish* utiliza una PHT para mezclar las salidas de las 2 funciones G paralelas. La Figura 4.9 presenta como se implementa la PHT.

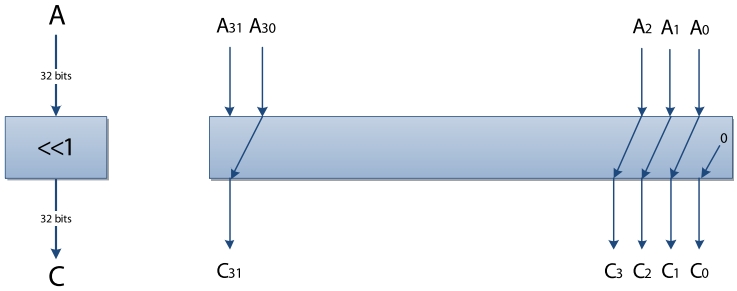
Figura 4.9. Transformada pseudo-Hadamard



La operación *2 * j* de la ecuación 4.2, se implementa con una rotación lógica de 1 bit, que es equivalente a multiplicar por dos.

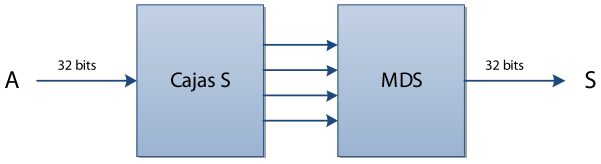
– *Rotación lógica a la izquierda, 1 bit:* Este bloque realiza una rotación lógica de 1 bit a la izquierda de un dato de 32 bits. Es decir, el bit en la posición 0, quedará a la salida en el bit 1; el bit 1 en el bit 2, y así sucesivamente. Como se trata de una rotación lógica, el bit 31 se pierde y el bit 0 queda con el dato 0. La Figura 4.10 muestra como se puede implementar esta permutación de bits.

Figura 4.10. Rotación lógica a la izquierda de 1 bit



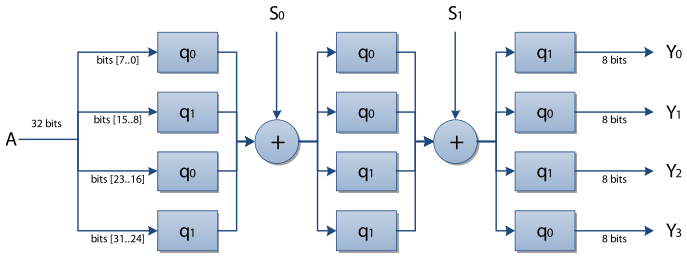
– *Función G:* Se compone de dos partes, primero se realiza una sustitución a los datos a través de Cajas-S, y posteriormente se realiza una operación matemática con los datos utilizando una matriz denominada MDS. La Figura 4.11 muestra la arquitectura de la función G.

Figura 4.11. Función G



– *Cajas-S:* Los datos de 32 bits se subdividen en cuatro palabras de 8 bits y estos se ingresan a los bloques de permutación q_i, posteriormente las salidas se unen y se blanquean con las subclaves S_i. La Figura 4.12 presenta la estructura del bloque de Cajas-S.

Figura 4.12. Cajas-S



– *Permutaciones q_i:* La entrada a la permutación es de 8 bits, y esta se separa en dos *nibbles* de 4 bits. Internamente las dos permutaciones son similares a excepción de las sustituciones que se realizan en los bloques t_0, t_1, t_2 y t_3. La Figura 4.13 muestra la estructura de las permutaciones. Y las Tablas 4.1 y 4.2 presentan los valores de las sustituciones para t_0, t_1, t_2 y t_3.

Figura 4.13. Permutaciones q_i

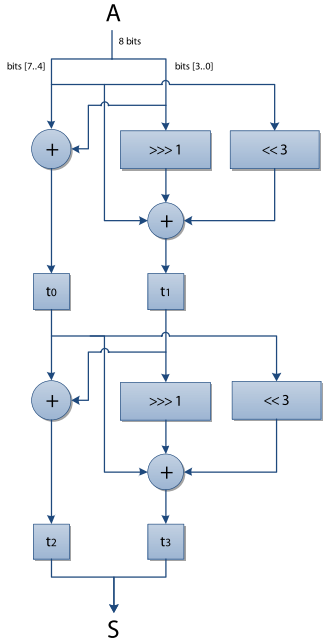


Tabla 4.1. Sustituciones t_i para q_0

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
t_0	0x8	0x1	0x7	0xD	0x6	0xF	0x3	0x2	0x0	0xB	0x5	0x9	0xE	0xC	0xA	0x4
t_1	0xE	0 C	0xB	0x8	0x1	0x2	0x3	0x5	0xF	0x4	0xA	0x6	0x7	0x0	0x9	0xD
t_2	0xB	0xA	0x5	0xE	0x6	0xD	0x9	0x0	0xC	0x8	0xF	0x3	0x2	0x4	0x7	0x1
t_3	0xD	0x7	0xF	0x4	0x1	0x2	0x6	0xE	0x9	0xB	0x3	0x0	0x8	0x5	0xC	0xA

Tabla 4.2. Sustituciones t_i para q_i

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
t_0	0x2	0x8	0xB	0xD	0xF	0x7	0x6	0xE	0x3	0x1	0x9	0x4	0x0	0xA	0xC	0x5
t_1	0x1	0xE	0x2	0xB	0x4	0xC	0x3	0x7	0x6	0xD	0xA	0x5	0xF	0x9	0x0	0x8
t_2	0x4	0xC	0x7	0x5	0x1	0x6	0x9	0xA	0x0	0xE	0xD	0x8	0x2	0xB	0x3	0xF
t_3	0xB	0x9	0x5	0x1	0xC	0x3	0xD	0xE	0x6	0x4	0x7	0xF	0x2	0x0	0x8	0xA

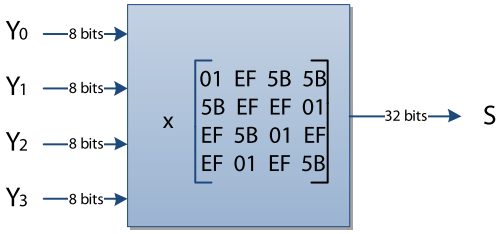
– *MDS:* Este bloque realiza una multiplicación sobre $GF(2^8)$ con la matriz de máxima distancia de separación. Este es el mecanismo principal de difusión de *Twofish*. La ecuación 4.3 describe la operación realizada en este bloque y la Figura 4.14 muestra la estructura del bloque MDS.

$$\begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} 01 & EF & 5B & 5B \\ 5B & EF & EF & 01 \\ EF & 5B & 01 & EF \\ EF & 01 & EF & 5B \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} \qquad (4.3)$$

El polinomio primitivo para realizar las operaciones en el campo $GF(2^8)$ esta dado en la ecuación 4.4

$$m_1(x) = x^8 + x^6 + x^5 + x^3 + 1 \qquad (4.4)$$

Figura 4.14. MDS



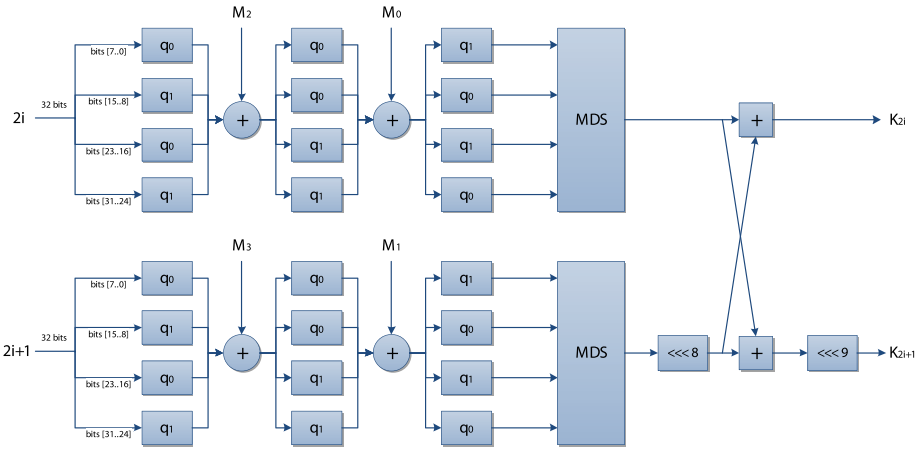
Expansión de clave *Twofish*

El algoritmo de encriptación *Twofish* utiliza tres tipos de subclaves diferentes en el proceso de cifrado, descifrado y expansión de clave: K_i, S_i y M_i.

Subclaves K_i

Las subclaves K_i son 40 palabras de 32 bits y se obtienen de una estructura parecida a una ronda del cifrador; las diferencias radican en que no hay una suma después de la transformada PHT, la rotación cíclica de 8 bits se realiza al final y no al inicio, la operación XOR se realiza con las subclaves M_i, en lugar de las subclaves S_i. La variable *i* varía entre los valores de 0 y 19 consecutivamente. La Figura 4.15 muestra la estructura del bloque de generación de las subclaves K_i.

Figura 4.15. Generador de subclaves K_i



Subclaves M_i

Las subclaves M_i son 4 palabras de 32 bits y se obtienen directamente de la clave *K* de 128 bits suministrada por el usuario. Siendo M_0 los 32 bits menos significativos y los M_3 32 bits más significativos.

Subclaves S_i

Las subclaves S_i son 2 palabras de 32 bits y se obtienen de multiplicar una parte de las subclaves por una matriz constante denominada RS. La multiplicación de esta matriz se realiza sobre un campo $GF(2^8)$ y se presenta en la ecuación 4.5, el polinomio primitivo esta dado por la ecuación 4.6.

$$\begin{bmatrix} S_{i,0} \\ S_{i,1} \\ S_{i,2} \\ S_{i,3} \end{bmatrix} = \begin{bmatrix} 01 & EF & 5B & 5B \\ 5B & EF & EF & 01 \\ EF & 5B & 01 & EF \\ EF & 01 & EF & 5B \end{bmatrix} \begin{bmatrix} m_{8i} \\ m_{8i+1} \\ m_{8i+2} \\ m_{8i+3} \\ m_{8i+4} \\ m_{8i+5} \\ m_{8i+6} \\ m_{8i+7} \end{bmatrix} \quad (4.5)$$

$$m_2(x) = x^8 + x^6 + x^3 + x^2 + 1 \quad (4.6)$$

Los datos m_j son 16 palabras de 8 bits, que resultan de dividir la clave *K* de 128 bits suministrada por el usuario en palabras de 8 bits, siendo m_0 los 8 bits menos significativos y m_{15} los más significativos.

Descifrador *Twofish*

El algoritmo de descifrado *Twofish* es el proceso inverso al cifrado *Twofish*; una de las características más importantes de *Twofish*, por no decir que la más importante, es que el descifrador presenta una estructura casi idéntica al cifrador, lo que lo hace muy atractivo para implementaciones en hardware. El algoritmo de descifrado *Twofish* consta de 32 rondas, las cuales se reparten de la siguiente manera:

- *Blanqueo de entrada*.
- 16 rondas inversas de mezcla.
- Un *blanqueo de salida*.

En la Figura 4.16 se presenta un esquema del diagrama de bloques del descifrado *Twofish* y en la Figura 4.17 se puede apreciar el diagrama de flujo del algoritmo de descifrado *Twofish*.

Figura 4.16. Esquema básico del descifrador *Twofish*

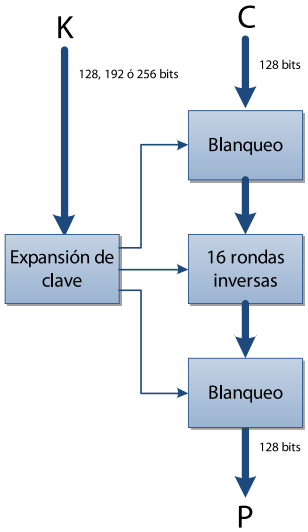
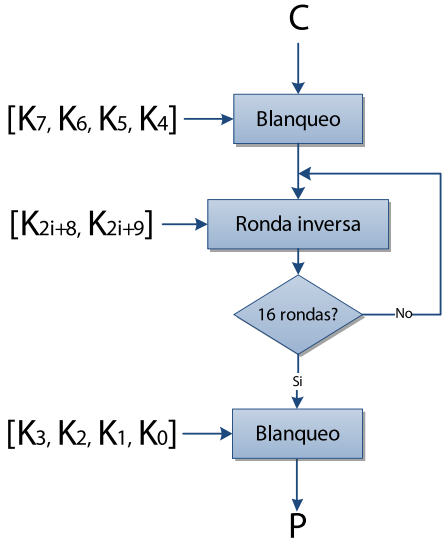


Figura 4.17. Algoritmo de descifrado *Twofish*



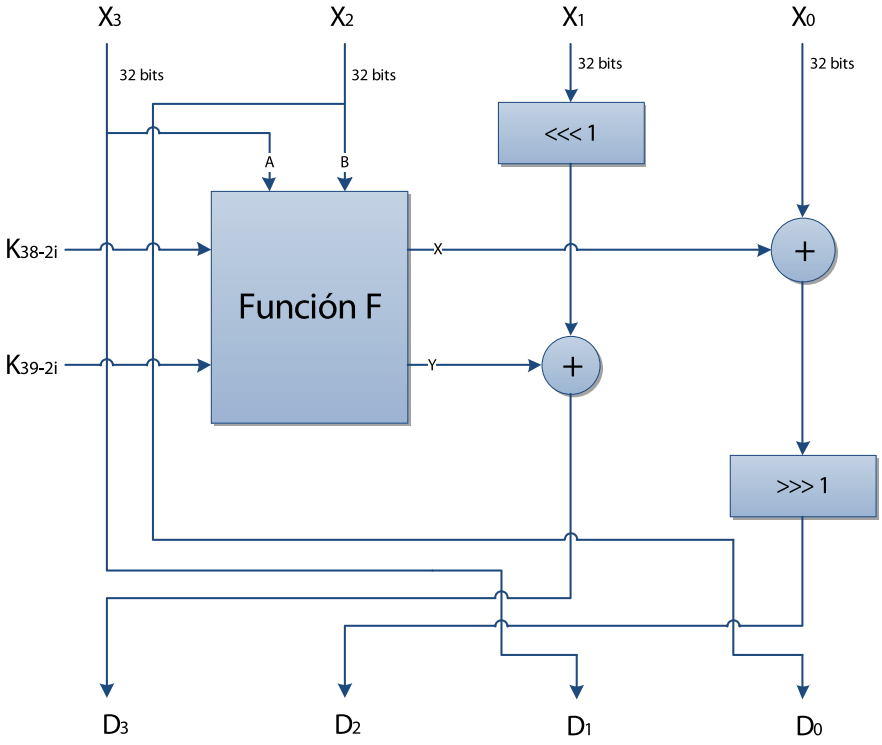
Blanqueo

Se toma un bloque de texto plano de 128 bits agrupado en 4 palabras de 32 bits cada uno, y se realiza la operación *XOR* bit a bit con 4 subclaves (de 32 bits cada una). Para el caso del blanqueo de entrada las subclaves utilizadas son las K_4, K_5, K_6 y K_7 ; para el blanqueo de salida las subclaves corresponden a K_0, K_1, K_2 y K_3. La Figura 4.4 presenta la estructura de este bloque.

Ronda inversa

La ronda de descifrado *Twofish* es muy similar a la ronda de cifrado, la única diferencia es el orden en que se realizan las rotaciones cíclicas de 1 bit, las operaciones *XOR*, y el orden de utilización de las subclaves K_i. De resto la ronda es idéntica, la función F no presenta alteraciones de ningún tipo.

Figura 4.18. Ronda inversa *Twofish*

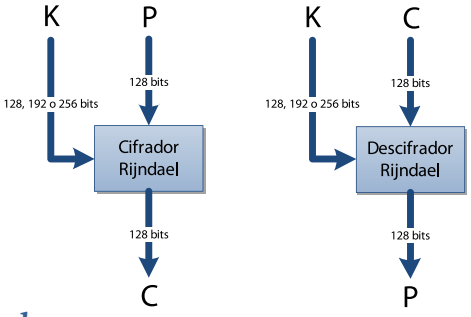


Algoritmo de cifrado Rijndael

El algoritmo de cifrado *Rijndael* fue diseñado por Joan Daemen y Vincent Rijmen (Daemen y Rijmen, 2000).

En general, *Rijndael* encripta texto plano *P*, de longitud variable, a texto cifrado C, a través de varias rondas controladas por 10, 12 o 14 subclaves. La clave de usuario *K*, el texto plano *P* y el texto cifrado C son variables con longitudes de 128, 192 o 256 bits. En este aparte se describirá el algoritmo *Rijndael* para texto plano *P* de 128 bits y una longitud de clave *K* de 128 bits. La Figura 5.1 presenta las convenciones seguidas en el diseño del cifrador *Rijndael*.

Figura 5.1. Convenciones para el cifrado y descifrado *Rijndael*



Cifrador *Rijndael*

El cifrador de bloque *Rijndael* en sí consta de:

– Una *ronda inicial*, que se compone del bloque AddRound.
– 10, 12 ó 14 rondas de cifrado (dependiendo del tamaño de la clave), cada ronda compuesta por los bloques SubByte, ShiftRow, MixColum y AddRound.
– Una *ronda final*, compuesta por los bloques SubByte, ShiftRow y AddRound.

En la Figura 5.2 se presenta un esquema del diagrama de bloques del cifrado *Rijndael* y en la Figura 5.3 se puede apreciar el diagrama de flujo del algoritmo de cifrado *Rijndael*. Se ha denotado con ①, ② y ③ las secciones correspondientes entre el diagrama de bloques y el algoritmo. El descifrado se puede apreciar en la sección 5.3.

Figura 5.2. Esquema básico del cifrador *Rijndael*

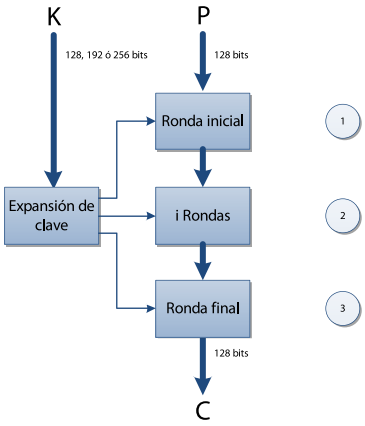
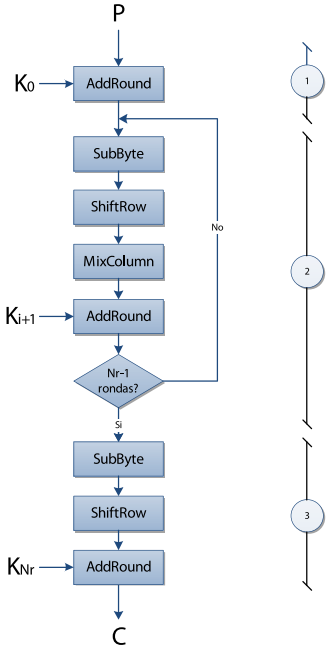


Figura 5.3. Algoritmo de cifrado *Rijndael*



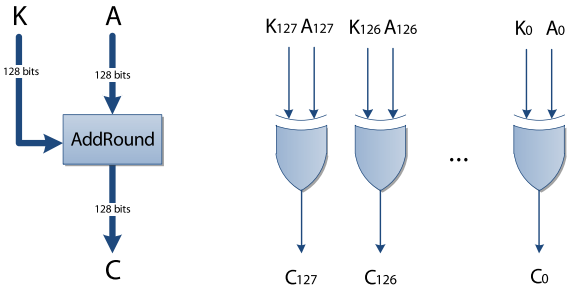
Los datos del texto plano *P*, se toman en palabras de 8 bits y se organizan en una matriz de denominada *estado* (igual pasa con la clave *K*). La Ecuación 5.1 muestra cómo se organizan los datos en esta matriz, siendo $p_{3,3}$ el byte más significativo, y $p_{0,0}$ el menos significativo.

$$[p_{3,3} \quad p_{3,2} \quad p_{3,1} \quad \cdots \quad p_{0,0}] = \begin{bmatrix} p_{0,0} & p_{0,1} & p_{0,2} & p_{0,3} \\ p_{1,0} & p_{1,1} & p_{1,2} & p_{1,3} \\ p_{2,0} & p_{2,1} & p_{2,2} & p_{2,3} \\ p_{3,0} & p_{3,1} & p_{3,2} & p_{3,3} \end{bmatrix} \qquad (5.1)$$

AddRound

Este bloque toma la matriz estado y hace una *XOR* bit a bit con la correspondiente matriz clave. La Figura 5.4 presenta el esquema de esta operación.

Figura 5.4. AddRound



SubByte

Este bloque lo que hace es sustituir cada uno de los bytes de la matriz de estado a través de la utilización de una Caja-S. La Figura 5.5 muestra el esquema de este bloque y la Tabla 5.1 presenta los valores de la Caja-S. Las filas corresponden al primer dígito y las columnas al segundo. Así, el dato 0x61 será sustituido por 0xEF.

Figura 5.5. SubByte

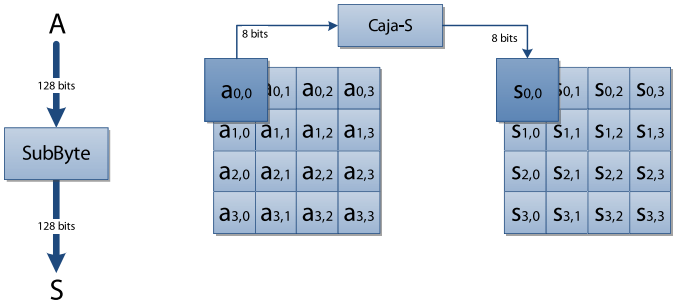


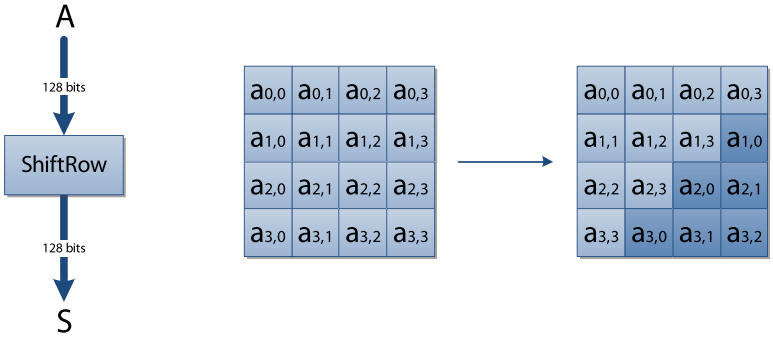
Tabla 5.1. Caja-S

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x0	0x63	0x7C	0x77	0x7B	0xF2	0x6B	0x6F	0xC5	0x30	0x01	0x67	0x2B	0xFE	0xD7	0xAB	0x76
0x1	0xCA	0x82	0xC9	0x7D	0xFA	0x59	0x47	0xF0	0xAD	0xD4	0xA2	0xAF	0x9C	0xA4	0x72	0xC0
0x2	0xB7	0xFD	0x93	0x26	0x36	0x3F	0xF7	0xCC	0x34	0xA5	0xE5	0xF1	0x71	0xD8	0x31	0x15
0x3	0x04	0xC7	0x23	0xC3	0x18	0x96	0x5	0x9A	0x07	0x12	0x80	0xE2	0xEB	0x27	0xB2	0x75
0x4	0x09	0x83	0x2C	0x1A	0x1B	0x6E	0x5A	0xA0	0x52	0x3B	0xD6	0xB3	0x29	0xE3	0x2F	0x84
0x5	0x53	0xD1	0x00	0xED	0x20	0xFC	0xB1	0x5B	0x6A	0xCB	0xBE	0x39	0x4A	0x4C	0x58	0xCF
0x6	0xD0	0xEF	0xAA	0xFB	0x43	0x4D	0x33	0x85	0x45	0xF9	0x02	0x7F	0x50	0x3C	0x9F	0xA8
0x7	0x51	0xA3	0x40	0x8F	0x92	0x9D	0x38	0xF5	0xBC	0xB6	0xDA	0x21	0x10	0xFF	0xF3	0xD2
0x8	0xCD	0x0C	0x13	0xEC	0x5F	0x97	0x44	0x17	0xC4	0xA7	0x7E	0x3D	0x64	0x5D	0x19	0x73
0x9	0x60	0x81	0x4F	0xDC	0x22	0x2A	0x90	0x88	0x46	0xEE	0xB8	0x14	0xDE	0x5E	0x0B	0xDB
0xA	0xE0	0x32	0x3A	0x0A	0x49	0x06	0x24	0x5C	0xC2	0xD3	0xAC	0x62	0x91	0x95	0xE4	0x79
0xB	0xE7	0xC8	0x37	0x6D	0x8D	0xD5	0x4E	0xA9	0x6C	0x56	0xF4	0xEA	0x65	0x7A	0xAE	0x08
0xC	0xBA	0x78	0x25	0x2E	0x1C	0xA6	0xB4	0xC6	0xE8	0xDD	0x74	0x1F	0x4B	0xBD	0x8B	0x8A
0xD	0x70	0x3E	0xB5	0x66	0x48	0x03	0xF6	0x0E	0x61	0x35	0x57	0xB9	0x86	0xC1	0x1D	0x9E
0xE	0xE1	0xF8	0x98	0x11	0x69	0xD9	0x8E	0x94	0x9B	0x1E	0x87	0xE9	0xCE	0x55	0x28	0xDF
0xF	0x8C	0xA1	0x89	0x0D	0xBF	0xE6	0x42	0x68	0x41	0x99	0x2D	0x0F	0xB0	0x54	0xBB	0x16

ShiftRow

Este bloque transforma la matriz de estado aplicando corrimientos cíclicos a sus filas hacia la izquierda. La fila 1 se rota 1 byte, la fila 2 se rota 2 bytes y la fila 3 se rota tres bytes. La Figura 5.6 presenta este proceso.

Figura 5.6. ShiftRow



MixColumn

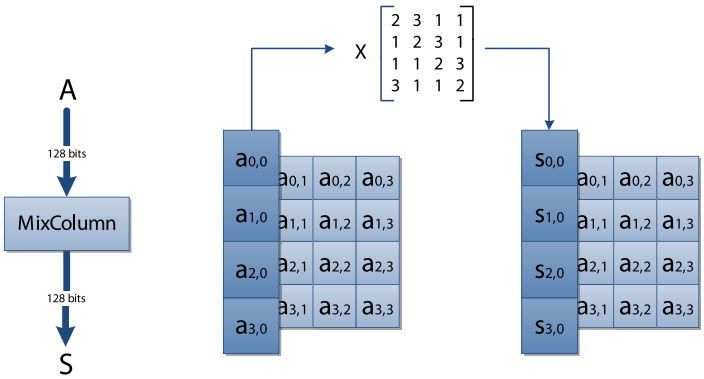
El bloque MixColum opera sobre cada columna de la matriz de estado. Se multiplica cada columna por una matriz constante sobre un $GF(2^8)$. La ecuación 5.2 presenta la operación que se realiza en el bloque MixColumn y la Figura 5.7 muestra la estructura del mismo.

$$\begin{bmatrix} S_{i,o} \\ S_{i,1} \\ S_{i,2} \\ S_{i,3} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} A_{i,o} \\ A_{i,1} \\ A_{i,2} \\ A_{i,3} \end{bmatrix} \qquad (5.2)$$

El polinomio irreducible utilizado en el se presenta en la ecuación 5.5.

$$m(x) = x^8 + x^4 + x^3 + x + 1 \qquad (5.3)$$

Figura 5.7. MixColumn



Expansión de clave *Rijndael*

Este proceso se encarga de generar las diferentes subclaves de ronda *Ki* que son utilizadas por el cifrador y el descifrador. El Algoritmo 5.1 presenta este proceso de expansión de clave siendo *Nb=4*, *Nr* el número de rondas (10 para clave de 128 bits, 12 para clave de 192 bits y 14 para clave de 256 bits) y *Nk* indica el tamaño de las claves de ronda (4 para clave de 128 bits, 6 para clave de 192 bits y 8 para clave de 256 bits).

Algoritmo 5.1. Algoritmo de expansión de clave *Rijndael*

Entrada: $\mathcal{K}, Nk, Nb, Nr$
Salida: $K[\,]$

//Nk Indica el tamaño de la clave de usuario (4 para 128, 6 para 192 y 8 para 256)
//Nb Indica el tamaño del texto plano (es constante e igual a 4)
//Nr Indica el número de rondas acorde a la clave (10 para 128 bits, 12 para 192 y 14 para 256)
//$W[\,]$ es un arreglo lineal de palabras de 32 bits en el rango $0 \leq i \leq Nb(Nr + 1)$
//$k[\,]$ es un arreglo lineal de palabras de 8 bits, con los datos de la clave de usuario.
//$K[\,]$ es el arreglo con las claves extendidas.

Para $i = 0$ hasta $i = Nk$ hacer
    $W[i] = \{ k[4i], k[4i + 1], k[4i + 2], k[4i + 3] \}$

Para $i = Nk$ hasta $i = Nb(Nr + 1)$ hacer
    temp = $W[i - 1]$

    Si $(i \bmod Nk) == 0$ hacer
        temp = SubByte(RotByte(temp)) xor $Rcon[i / Nk]$

    si $(Nk == 8)$ y $((i \bmod Nk) == 4))$
        temp = SubByte (temp)

    Fin si

    $W[i] = W[i - Nk]$ xor temp

Fin para

Para $i = 0$ hasta $i = Nr$ hacer
    $K[i] = \{ W[4i], W[4i + 1], W[4i + 2], W[4i + 3] \}$

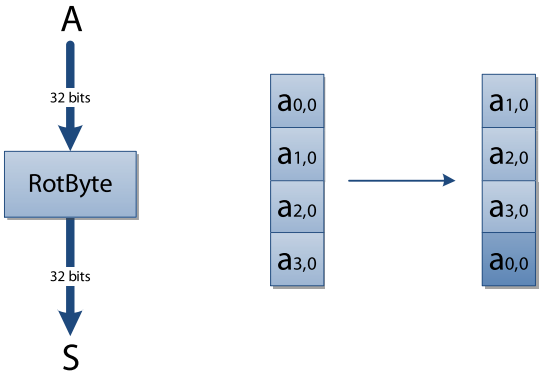
SubByte

Es el mismo bloque utilizado en el cifrador. Consiste en una Caja-S que sustituye el valor ingresado por otro. La Figura 5.5 muestra el esquema de este bloque y la Tabla 5.1 presenta los valores de la Caja-S.

RotByte

Devuelve una palabra en la cual los bytes son una rotación cíclica de su entrada, siendo la rotación de 1 byte. La Figura 5.8 presenta la estructura y el funcionamiento de este bloque.

Figura 5.8. MixColumn



RCon

Es un vector de datos constante y se encuentra definido por la Ecuación 5.5. Cada elemento de *RCon*, denominado *RCon[i]* es de 32 bits. Es un vector constante de datos de 8 bits definido por la ecuación 5.5.

$$RCon[i] = \{RC[i], 0,0,0\} \tag{5.4}$$

$$RC = \{0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x80, 0x1B, 0x36\} \tag{5.5}$$

Descifrador *Rijndael*

El descifrador es similar al cifrador y contiene elementos básicos similares, pero inversos. El descifrador de bloque *Rijndael* en sí consta de:

- Una *ronda inversa inicial*, que se compone de los bloque AddRound, InvShiftRow e InvSubByte.
- 10, 12 ó 14 *rondas inversas* de descifrado, cada ronda compuesta por los bloques AddRound, InvMixColum, InvShiftRow e InvSubByte.
- Un *ronda inversa final*, compuesta por el bloque AddRound.

En la Figura 5.9 se presenta un esquema del diagrama de bloques del descifrado *Rijndael*, en la Figura 5.10 se puede apreciar el diagrama de flujo del algoritmo de descifrado *Rijndael*. Se ha denotado con ①, ② y ③ las secciones correspondientes entre el diagrama de bloques y el algoritmo.

Figura 5.9. Esquema básico del descifrador *Rijndael*

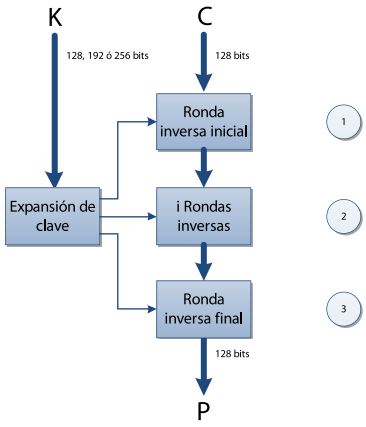
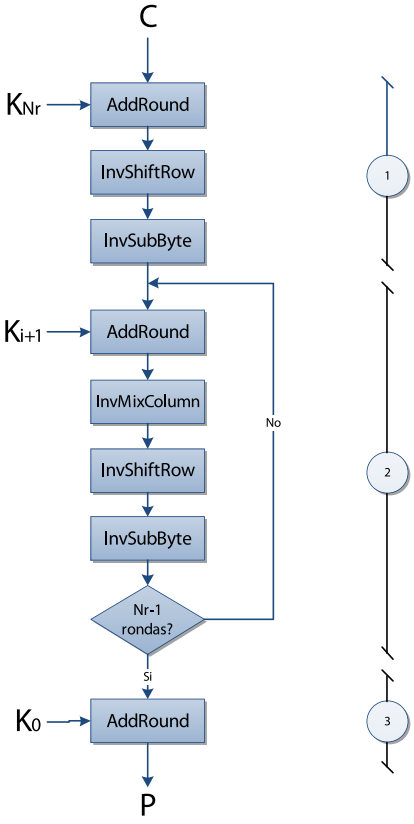


Figura 5.10. Algoritmo del cifrador *Rijndael*



InvSubByte

El bloque InvSubByte realiza la función inversa a SubByte, es decir utiliza la correspondiente Caja-S inversa para sustituir cada uno de los bytes que componen la matriz de estado. La Figura 5.11 muestra el esquema de este bloque y la Tabla 5.2 presenta los valores de la Caja-S.

Figura 5.11. InvSubByte

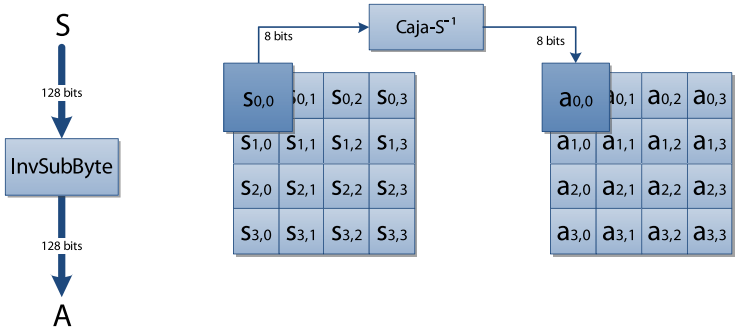


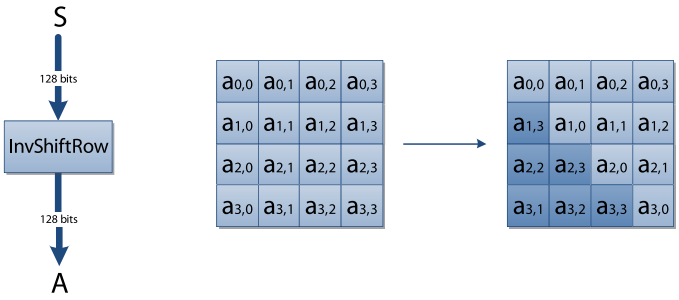
Tabla 5.2. Caja-S Inversa

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x0	0x52	0x09	0x6A	0xD5	0x30	0x36	0xA5	0x38	0xBF	0x40	0xA3	0x9E	0x81	0xF3	0xD7	0xFB
0x1	0x7C	0xE3	0x39	0x82	0x9B	0x2F	0xFF	0x87	0x34	0x8E	0x43	0x44	0xC4	0xDE	0xE9	0xCB
0x2	0x54	0x7B	0x94	0x32	0xA6	0xC2	0x23	0x3D	0xEE	0x4C	0x95	0x0B	0x42	0xFA	0xC3	0x4E
0x3	0x08	0x2E	0xA1	0x66	0x28	0xD9	0x24	0xB2	0x76	0x5B	0xA2	0x49	0x6D	0x8B	0xD1	0x25
0x4	0x72	0xF8	0xF6	0x64	0x86	0x68	0x98	0x16	0xD4	0xA4	0x5C	0xCC	0x5D	0x65	0xB6	0x92
0x5	0x6C	0x70	0x48	0x50	0xFD	0xED	0xB9	0xDA	0x5E	0x15	0x46	0x57	0xA7	0x8D	0x9D	0x84
0x6	0x90	0xD8	0xAB	0x00	0x8C	0xBC	0xD3	0x0A	0xF7	0xE4	0x58	0x05	0xB8	0xB3	0x45	0x06
0x7	0xD0	0x2C	0x1E	0x8F	0xCA	0x3F	0x0F	0x02	0xC1	0xAF	0xBD	0x03	0x01	0x13	0x8A	0x6B
0x8	0x3A	0x91	0x11	0x41	0x4F	0x67	0xDC	0xEA	0x97	0xF2	0xCF	0xCE	0xF0	0xB4	0xE6	0x73
0x9	0x96	0xAC	0x74	0x22	0xE7	0xAD	0x35	0x85	0xE2	0xF9	0x37	0xE8	0x1C	0x75	0xDF	0x6E
0xA	0x47	0xF1	0x1A	0x71	0x1D	0x29	0xC5	0x89	0x6F	0xB7	0x62	0x0E	0xAA	0x18	0xBE	0x1B
0xB	0xFC	0x56	0x3E	0x4B	0xC6	0xD2	0x79	0x20	0x9A	0xDB	0xC0	0xFE	0x78	0xCD	0x5A	0xF4
0xC	0x1F	0xDD	0xA8	0x33	0x88	0x07	0xC7	0x31	0xB1	0x12	0x10	0x59	0x27	0x80	0xEC	0x5F
0xD	0x60	0x51	0x7F	0xA9	0x19	0xB5	0x4A	0x0D	0x2D	0xE5	0x7A	0x9F	0x93	0xC9	0x9C	0xEF
0xE	0xA0	0xE0	0x3B	0x4D	0xAE	0x2A	0xF5	0xB0	0xC8	0xEB	0xBB	0x3C	0x83	0x53	0x99	0x61
0xF	0x17	0x2B	0x04	0x7E	0xBA	0x77	0xD6	0x26	0xE1	0x69	0x14	0x63	0x55	0x21	0x0C	0x7D

InvShiftRow

Este bloque realiza la función inversa a ShiftRow, transforma la matriz de estado, aplicando corrimientos cíclicos a sus filas hacia la derecha. La fila 1 se rota 1 byte, la fila 2 se rota 2 bytes y la fila 3 se rota tres bytes. La Figura 5.12 presenta este proceso.

Figura 5.12. InvShiftRow

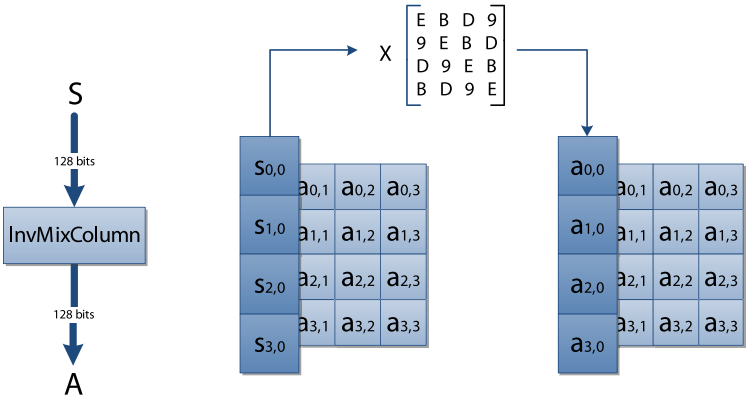


InvMixColumn

El bloque InvMixColum opera sobre cada columna de la matriz de estado. Se multiplica cada columna por una matriz constante, que es la inversa de la matriz constante utilizada en el bloque MixColumn. La ecuación 5.4 presenta la operación que se realiza en el bloque InvMixColumn y la Figura 5.13 muestra la estructura del mismo.

$$\begin{bmatrix} A_{i,o} \\ A_{i,1} \\ A_{i,2} \\ A_{i,3} \end{bmatrix} = \begin{bmatrix} E & B & D & 9 \\ 9 & E & B & D \\ D & 9 & E & B \\ B & D & 9 & E \end{bmatrix} \begin{bmatrix} S_{i,o} \\ S_{i,1} \\ S_{i,2} \\ S_{i,3} \end{bmatrix} \qquad (5.6)$$

Figura 5.13. InvMixColumn

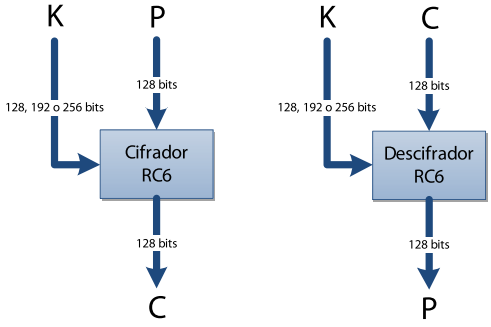


Algoritmo de cifrado RC6

El algoritmo de cifrado *RC6* fue diseñado por Ronald L. Rivest, M. J. B. Robshaw, R. Sidney y Y. L. Yin (Rivest y otros, 1998).

RC6 encripta 128 bits de texto plano en 128 bits de texto cifrado, a través de 20 rondas controladas por 44 subclaves de 128 bits. La clave de usuario es de 128 bits.

Figura 6.1. Convenciones para el cifrado y descifrado *RC6*



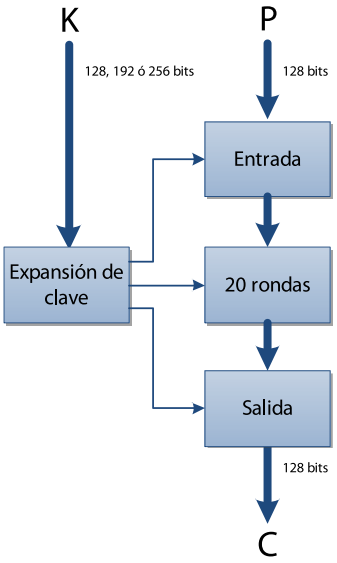
Cifrador RC6

El cifrador de bloque *RC6* en sí consta de:

– Una transformación inicial, denominada *entrada*.
– 20 rondas de mezcla
– Una transformación final, denominada *salida*.

En la Figura 6.2 se presenta un esquema del diagrama de bloques del cifrado *RC6*. El descifrado se puede apreciar en la página 82 (Descifrador *RC6*).

Figura 6.2. Esquema básico del cifrador *RC6*



El algoritmo de cifrado se encuentra descrito en el Algoritmo 6.1 que se presenta a continuación. Y la Figura 6.3 muestra el algoritmo de cifrado *RC6* en diagrama de flujo.

Algoritmo 6.1. Algoritmo de cifrado RC6

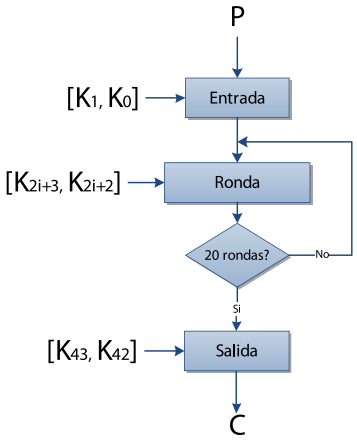
Entrada: Texto plano en $[A_3;\ A_2;\ A_1;\ A_0]$, 44 subclaves $K[\,]$.
Salida: texto cifrado en $[A_3;\ A_2;\ A_1;\ A_0]$

$A_1 = A_1 + K[0]$
$A_3 = A_3 + K[1]$

Para $i = 1$ hasta 19 hacer
$t = (A_1 \times (2A_1 + 1)) << 5$
$u = (A_3 \times (2A_3 + 1)) << 5$
$A_0 = ((A_0 \oplus t) << u) + K[2i + 2]$
$A_2 = ((A_2 \oplus u) << t) + K[2i + 3]$
$[A_3;\ A_2;\ A_1;\ A_0] = [A_0;\ A_3;\ A_2;\ A_1]$
Fin para

$A_0 = A_0 + K[42]$
$A_2 = A_2 + K[43]$

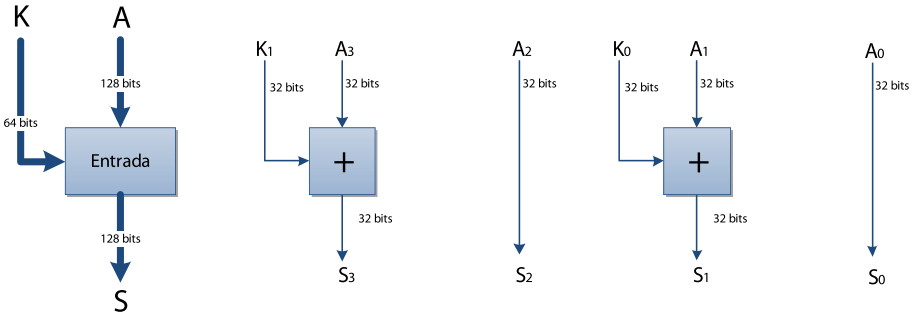
Figura 6.3. Algoritmo de cifrado *RC6*



Entrada

Consiste en realizar dos sumas módulo 2^{32} entre dos de las palabras que conforman el texto plano (las palabras 1 y 3) y dos de las subclaves K_i (las subclaves K_0 y K_1 respectivamente). En la Figura 6.4 se puede apreciar la estructura de este bloque.

Figura 6.4. Entrada para el cifrador *RC6*

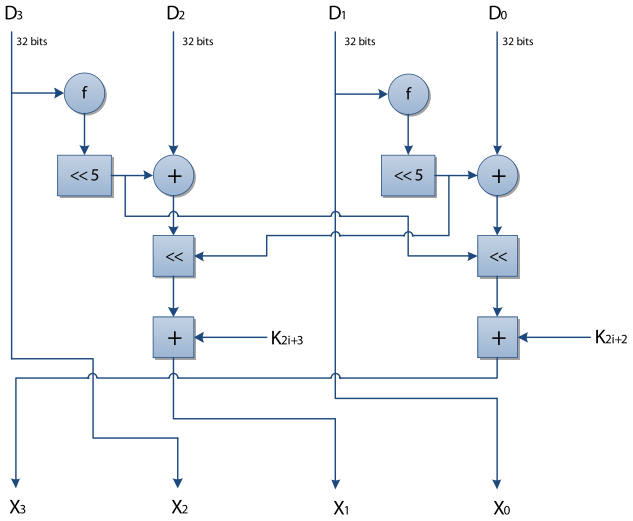


– *Suma módulo 2^{32}:* Es un bloque *full-adder* de 32 bits, sin *carry* de salida. La Figura 3.6 muestra un esquema de este bloque.

Ronda

El cifrador *RC6* realiza 20 rondas donde 128 bits de datos se mezclan, como lo muestra la Figura 6.5, utilizando 2 subclaves K_i en cada iteración. La operaciones que se encuentran en la ronda de cifrado *RC6* son: *función f, XOR*, rotación lógica a la izquierda de 5 bits, rotación lógica a la izquierda variable y suma módulo 2^{32}.

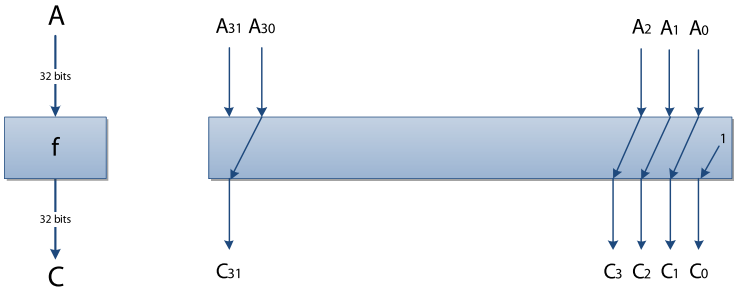
Figura 6.5. Ronda del cifrador RC6



– *Función f:* Consiste en multiplicar por 2 el dato de entrada y posteriormente sumarle 1. La ecuación 6.1 describe el proceso de la *función f*. La implementación de la función consiste en rotar lógicamente el dato un bit a la izquierda y colocar el bit 0 en 1. La Figura 6.5 presenta un esquema de la implementación de la función f.

$$C = f\{A\} = 2A + 1 \qquad (6.1)$$

Figura 6.6. Función f



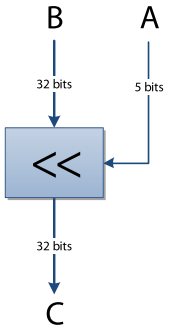
– *XOR:* Es un bloque que realiza la operación lógica *xor*, bit a bit, entre cada una de sus entradas. La Figura 3.7 muestra un esquema de este bloque.

– *Rotación lógica a la izquierda, 5 bits:* Este bloque realiza una rotación lógica de 5 bits a la izquierda de un dato de 32 bits. Es decir, el bit en la posición

0, quedará a la salida en el bit 5; el bit 1 en el bit 6 y así sucesivamente. Como se trata de una rotación lógica, los bits 27, 28, 29, 30 y 31 se pierden; y los bits 0, 1, 2, 3 y 4 quedan con el dato 0. La Figura 4.10 muestra como se puede implementar esta permutación de bits.

– *Rotación lógica a la izquierda:* A diferencia de las rotaciones lógicas anteriores, cuya cantidad de rotación es fija, la cantidad de rotación de esta está dada por una entrada adicional, se implementa a manera de *barrel shifter* (Menezes y otros, 1996). En la Figura 6.7 se presenta el diagrama de este bloque.

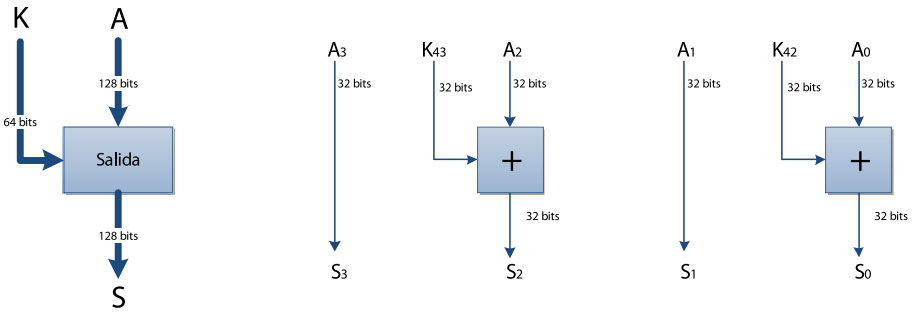
Figura 6.7. Rotación lógica a la izquierda



Salida

Consiste en realizar dos sumas módulo 2^{32} entre dos de las palabras que conforman el texto que va cifrado (las palabras 0 y 2) y dos de las subclaves K_i (las subclaves K_{42} y K_{43} respectivamente). En la Figura 6.8 se puede apreciar la estructura de este bloque.

Figura 6.8. Salida para el cifrador *RC6*



Expansión de clave RC6

El algoritmo de cifrado RC6 requiere de 44 palabras clave de 32 bits, denominadas subclaves K_i. Estas subclaves K_i se expanden de la clave K ingresada por el usuario, la cual consiste en palabras de 32 bits, siendo n un número entre 4 y 16.

Algoritmo 6.2. Algoritmo de expansión de clave RC6

Entrada: $L[\,]$
Salida: $K[\,]$

//$K[\,]$ es el arreglo con las 44 claves extendidas
//$L[\,]$ es un arreglo con la clave de usuario $\mathcal{K}$ dividida en palabras de 8 bits.

$K[0] = 0xB7E15163$

Para $i = 1$ hasta 43 hacer
    $K[i] = K[i-1] + 0x9E3779B9$
    $A = B = i = j = 0$
    $v = 3 \times max\{c, 44\}$

    Para $s = 1$ hasta v hacer
        $A = K[i] = (K[i] + A + B) <<< 3$
        $B = L[j] = (L[j] + A + B) <<< (A + B)$
        $i = (i + 1) \bmod 44$
        $j = (j + 1) \bmod 16$
    Fin Para

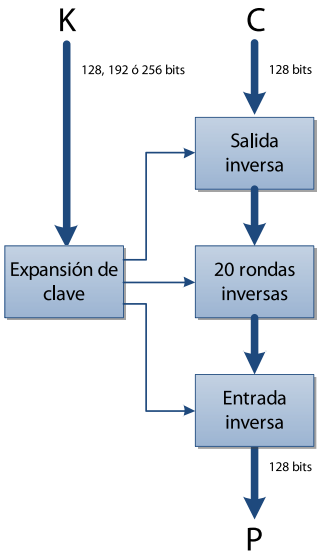
Fin Para

Descifrador RC6

El cifrador de bloque RC6, como se puede apreciar en la Figura 6.9, en sí consta de:

- Una transformación inicial, denominada *salida inversa*.
- 20 rondas inversas.
- Una transformación final, denominada *entrada inversa*.

Figura 6.9. Esquema básico del descifrador *RC6*



El algoritmo de descifrado se encuentra descrito en el Algoritmo 6.3 que se presenta a continuación. Y la Figura 6.10 muestra el algoritmo de cifrado *RC6* en diagrama de flujo.

Algoritmo 6.3. Algoritmo de descifrado *RC6*

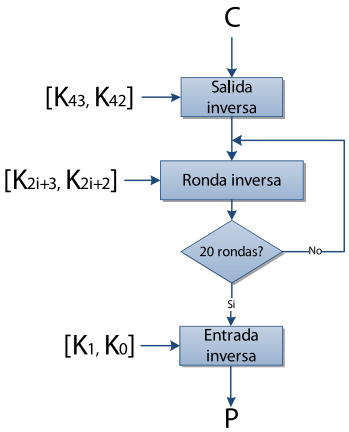
Entrada: Texto cifrado en $[A_3;\ A_2;\ A_1;\ A_0]$, 44 subclaves $K[\,]$.
Salida: Texto plano en $[A_3;\ A_2;\ A_1;\ A_0]$

$A_0 = A_0 - K[42]$
$A_2 = A_2 - K[43]$

Para $i = 19$ hasta 1 hacer
$[A_3;\ A_2;\ A_1;\ A_0] = [A_2;\ A_1;\ A_0;\ A_3]$
$t = (A_1 \times (2A_1 + 1)) << 5$
$u = (A_3 \times (2A_3 + 1)) << 5$
$A_0 = ((A_0 - K[2i]) >> u) \oplus t$
$A_2 = ((A_2 - K[2i+1]) >> t) \oplus u$
Fin para

$A_1 = A_1 - K[0]$
$A_3 = A_3 - K[1]$

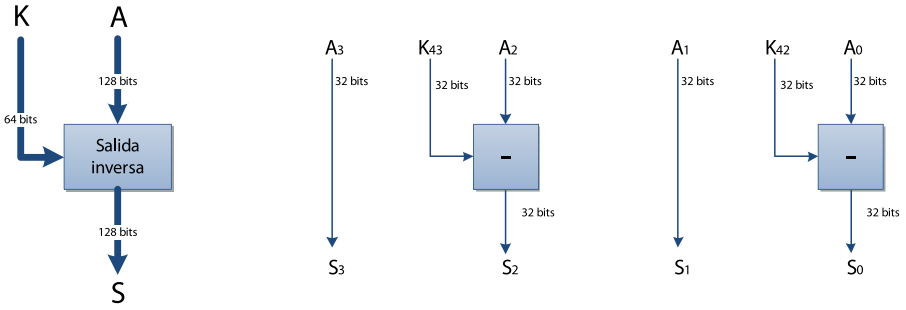
Figura 6.10. Algoritmo de cifrado *RC6*



Salida inversa

Consiste en realizar dos restas módulo 2^{32} entre dos de las palabras que conforman el texto plano (las palabras 0 y 2) y dos de las subclaves K_i (las subclaves K_{42} y K_{43}, respectivamente). En la Figura 6.11 se puede apreciar la estructura de este bloque.

Figura 6.11. Salida inversa para el cifrador *RC6*

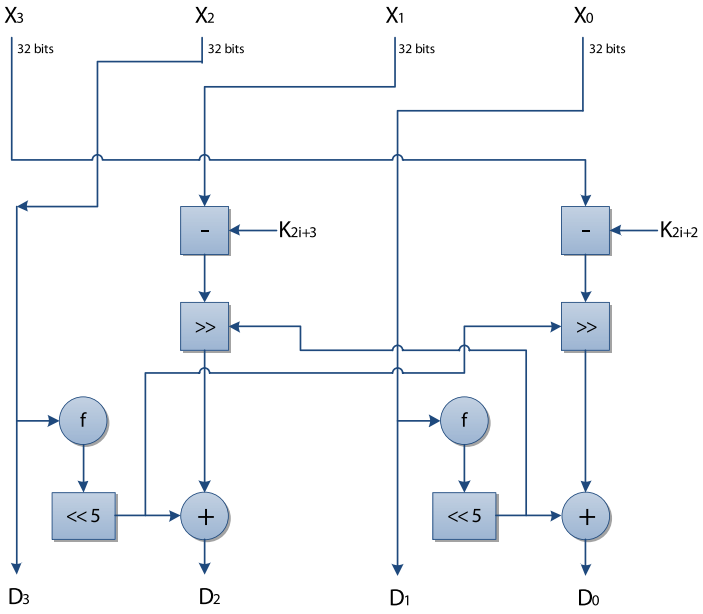


– *Resta módulo 2^{32}:* Es un bloque *full-subtractor* de 32 bits, sin *borrow* de salida. Se presenta en la Figura 3.16.

Ronda inversa

El cifrador RC6 realiza 20 rondas inversas donde 128 bits de datos se mezclan como lo muestra la Figura 6.12, utilizando 2 subclaves K_i en cada iteración. La operaciones que se encuentran en la ronda de cifrado RC6 son: *función f, XOR*, rotación lógica a la izquierda de 5 bits, rotación lógica a la derecha variable y suma módulo 2^{32}.

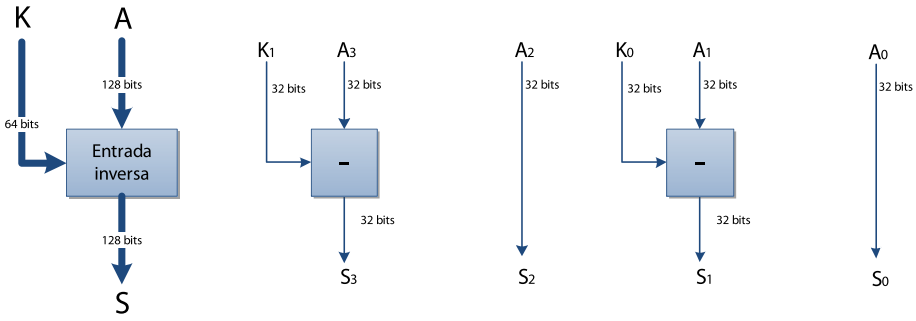
Figura 6.12. Ronda inversa del descifrador *RC6*



Entrada inversa

Consiste en realizar dos restas módulo 2^{32} entre dos de las palabras que conforman el texto que va cifrado (las palabras 1 y 3) y dos de las subclaves K_i (las subclaves K_0 y K_1 respectivamente). En la Figura 6.13 se puede apreciar la estructura de este bloque.

Figura 6.13. Entrada inversa para el cifrador *RC6*



Diseño e implementación de los co-procesadores

El proceso de diseño e implementación se los co-procesadores se realiza en dos partes, primero se diseña e implementa cada cifrador de bloque, de manera que presenten la misma entidad y la métrica sea efectivamente comparativa; es por esto que se deben homogeneizar los tamaños de texto plano y claves para cada uno de los cifradores. La Tabla 7.1 presenta un resumen de los tamaños de los datos de cada cifrador.

Tabla 7.1. Tamaño de datos para los cifradores

	Descripción del algoritmo en general		**Algoritmo homogeneizado**	
	Bits de texto plano/cifrado	**Bits de clave**	**Bits de texto plano/cifrado**	**Bits de clave**
Serpent	128	128 a 256	128	128
Mars	128	128 y 256	128	128
Twofish	128	128, 192 y 256	128	128
Rijndael	128	128, 192 y 256	128	128
RC6	128	128	128	128

La segunda parte consiste en diseñar una arquitectura de co-procesador que permita manejar datos para un sistema embebido con arquitectura escalable de 8, 16, 32 y 64 bits; y que fácilmente soporte el intercambio del cifrador sin necesidad de grandes modificaciones en el mismo.

Parámetros de desempeño relevantes en la implementación de los co-procesadores

Es usual en el diseño de procesadores y sistemas digitales que en los PLD sean medidos ciertos parámetros que indiquen cantidades que permitan realizar comparaciones con otras implementaciones. Entre los parámetros de desempeño usuales se encuentran: área usada, velocidad de procesamiento, potencia consumida y, en el caso de co-procesadores, el nivel de seguridad.

Recursos usados

Los recursos en un FPGA hacen referencia a la cantidad de recursos lógicos utilizados por el sistema diseñado al implementarse. Estos recursos lógicos pueden ser variados y dependen de la FPGA o CPLD en el cual se implementará el diseño. Para el caso de esta investigación, se utiliza el dispositivo Cyclone II EP2C35F672C6 de Altera, y entre los recursos lógicos disponibles se encuentran:

- LUTs (*Look Up Tables*).
- Registros.
- Multiplicadores embebidos de 9 bits.
- RAM.
- PLLs.

La arquitectura de las PLD de Altera presenta en su diseño unos elementos denominados LE (*Logic Element* – Elemento Lógico), el cual puede ser sintetizado como lógica combinatoria (en LUT) y/o como registro; por lo tanto, estos dos tipos de elementos lógicos se pueden medir en LE.

Como estrategia de diseño para permitir la implementación de los co-procesadores en diferentes referencias de PLDs y poder hacer comparaciones justas, se decide que las implementaciones sólo utilizarán LEs y ningún otro tipo de recurso lógico.

Entonces, la medición de recursos usados se determina conociendo el número de LEs que ocupa la sintetización del diseño, y usualmente se representa en el porcentaje de LEs que ocupa el sistema en relación a la cantidad total de LEs que posee un PLD. Así, si por ejemplo, un PLD posee 50.000 LEs, y la sintetización de un diseño presenta 10.000 LEs, al implementarse el diseño en este PLD se ocupará un área del 20% del total disponible.

Velocidad de cifrado

Se considera como "velocidad" en un sistema digital síncrono a la frecuencia mínima necesaria con la cual se puede realizar completamente el procesamiento.

La velocidad de cifrado depende principalmente de los ciclos de reloj que tarda el sistema en relación con las rondas que debe realizar para que se complete el proceso de cifrado. Cuanto más se realice el proceso, mayor es el tiempo de retardo.

$$V = \text{Tiempo de reloj} * \text{Ciclos de reloj} \qquad (7.1)$$

Por ejemplo, si el tiempo de reloj mínimo necesario para que un sistema funcione es de 128*ns* y el sistema utiliza 10 ciclos de reloj, entonces la velocidad total del sistema es de 1.28μs.

Cuando se habla de velocidad de cifrado, se debe tomar en cuenta el *throughput*; ya que este parámetro permite saber la cantidad de bits por segundo a la cual trabaja el sistema digital.

El *throughput* es un factor importante para determinar el rendimiento en cuanto a tiempo de un diseño. El *throughput* de un diseño es obtenido mediante la multiplicación de la frecuencia máxima de operación permitida para el reloj por el número de bits procesados por ciclo (Menezes y otros, 1996). Para los algoritmos criptográficos es definido como:

$$\text{Throughput} = \frac{\text{Número de bits}}{\text{Tiempo de reloj} * \text{Número de ciclos}} (\text{bits/s}) \qquad (7.2)$$

Así, por lo tanto, para el ejemplo anterior con un tiempo de reloj de 128, 10 ciclos de reloj y 128 bits, el *throughput* será igual a 100 Mbps.

Estimación de potencia

El consumo de potencia en una FPGA depende de los parámetros que pueden ser resumidos en la siguiente ecuación (Altera, 2006):

$$P_{FPGA} = V_{DD}F * tog * g(T°C, netlist, K) \qquad (7.3)$$

Donde:

– V_{DD} es el voltaje de alimentación.
– F es la frecuencia de la señal de reloj.

– *tog* es el porcentaje de actividad del nodo con respecto al reloj.
– *g* es una función que depende de la temperatura, la tecnología y los recursos internos utilizados (netlist) que es el equivalente a la capacitancia de carga.

$$g(T°C, \text{netlist}, K) = [\alpha_{LC} + B_{\text{interconnect}} + \delta_{CLK} + \varepsilon_{I/O} + \varphi_{Men}] \qquad (7.4)$$

Donde:

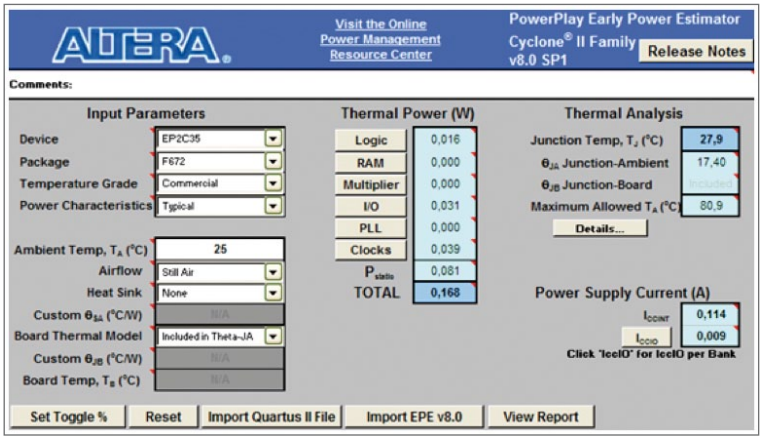
– α_{LC} es la corriente consumida por las celdas lógicas.
– $B_{\text{interconnect}}$ es la corriente consumida por los recursos de interconexión.
– δ_{CLK} el consumo de la distribución del reloj.
– $\varepsilon_{I/O}$ el consumo de las terminales de entrada y salida.
– φ_{Men} el consumo de las celdas de memoria interna.

Cálculo de potencia usando PowerPlay Early Power Estimator

Cuando se quiere calcular el consumo de potencia en un PLD, se debe tener en cuenta que sólo se puede hacer una estimación de esta, ya que con exactitud no se puede calcular la potencia que consume parte de un PLD.

Por esta razón además de la ecuación 7.3, existe una herramienta llamada *Cyclone II PowerPlay Early Power Estimator* que permite estimar la potencia estática consumida en FPGAs Cyclone II, que es la familia de FPGAs en la cual se implementaron los co-procesadores. Esta herramienta es una hoja de cálculo desarrollada en Excel y en la Figura 7.1 se puede ver su interfaz, la cual permite introducir los valores requeridos que se encuentran en los reportes que previamente se han obtenido al sintetizar el diseño.

Figura 7.1. PowerPlay Early Power Estimator

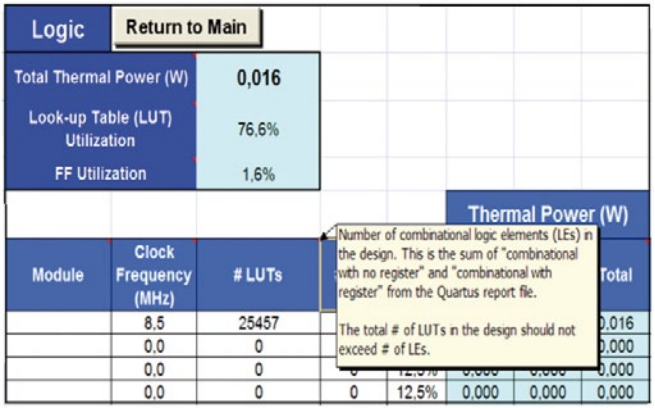


Para introducir los valores éxitosamente se debe habilitar los marcos en la ventana de advertencia de seguridad que aparece en la parte superior de la hoja.

Los parámetros requeridos incluyen: *device, package, temperature grade, power characteristics, ambient temperature, airflow, heat sink, custom ϑSA, board thermal model, custom ϑJB, board temp TB.*

Para facilidad al introducir los valores de *logic, RAM, multipliers, I/O, PLL* y *clocks*, los recuadros tienen comentarios los cuales explican detalladamente lo que se requiere, como lo muestra la Figura 7.2.

Figura 7.2. Requerimientos de elementos lógicos



Al final se puede encontrar un reporte con toda la información introducida por el usuario, cuando se regresa a la página principal en *Thermal power* (W) se encuentra el resumen de los datos que fueron modificados y al final se encuentra la potencia estática que se estima para el diseño. Ver Figura 7.1.

Diseño de los cifradores de bloque

Los cifradores de bloque *Mars*, *RC6*, *Rijndael*, *Serpent* y *Twofish* fueron diseñados en VHDL para que presentasen la misma entidad. Esta entidad consta de las entradas de *start, reset, enable, clock, tin* de 128 bits, *key* de 128 bits y EN_DEC, esta entidad también tiene un bus de salida Tout de 128 bits y Readyque es una salida que permite saber cuándo se completa el proceso de cifrado.

A esta entidad que cifra/descifra se le dio el nombre de *cipher core*, y se representa en la Figura 7.3 con sus respectivas entradas y salidas. En la Tabla 7.2 se describen las respectivas funciones entradas y salidas del *cipher core*.

Figura 7.3. Entidad del *cipher core*

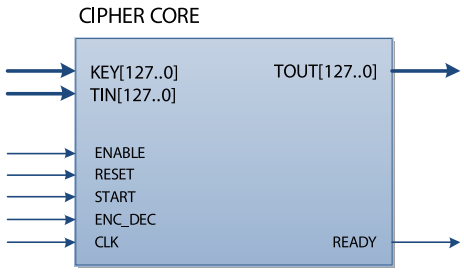


Tabla 7.2. Señales del *cipher core*

Nombre de la señal	Tamaño en bits	Dirección de la señal	Función
Enable	1	Entrada	Es la señal encargada de habilitar el sistema para su funcionamiento.
Reset	1	Entrada	Es el *reset* general del sistema. Es una salida activa baja.
Start	1	Entrada	Se encarga de iniciar el proceso de cifrado/descifrado. Es activa alta. Por lo tanto si Start=1 se inicia el proceso, si Start=0 el proceso no se realiza.
Clock	1	Entrada	Esta señal es el reloj externo. El valor mínimo depende de la velocidad de operación del procesador.
En_dec	1	Entrada	Indica que clase de función realiza el procesador. Si EN_DEC=1, realiza el proceso de cifrado; si EN_DEC=0, realiza el proceso de decifrado.
Tin	128	Entrada	Este bus corresponde a los datos de entrada.
Key	128	Entrada	Este bus corresponde a la clave con la cual se han de cifrar o descifrar los datos.
Ready	1	Salida	Esta señal de salida indica cuando se ha finalizado el proceso de cifrado ó descifrado. READY es igual 1, sólo cuando termina el proceso de cifrado (o descifrado).
Tout	128	Salida	Este bus corresponde a los datos de salida.

Como se puede apreciar, se ha diseñado una misma entidad para los diferentes cifradores de bloque; pero la arquitectura de éstos es diferente y acorde a las especificaciones de cada algoritmo (ver Capítulos 2 a 6).

Implementación de los cifradores de bloque

Los algoritmos de cifrado de bloque fueron diseñados e implementados en una misma plataforma: el dispositivo Cyclone II EP2C35F672C6 de Altera[1], donde se realizaron simulaciones en software (con el *cipher core* sintetizado) y pruebas directamente en la FPGA (con el *cipher core* implementado en la misma). En las Tablas 7.3 a 7.8 se presentan los resultados de la implementación para cada *cipher core*.

Cipher core Rijndael

Tabla 7.3. Resultados para el *cipher core Rijndael*

Parámetro	Valor
Tiempo de cifrado	180 ns
Throughput	711 Mbps
Recursos usados	30.997 LE
Estimación de potencia	718 mW

Cipher core Serpent

Tabla 7.4. Resultados para el *cipher core Serpent*

Parámetro	Valor
Tiempo de cifrado	480 ns
Throughput	266 Mbps
Recursos usados	25.740 LE
Estimación de potencia	701 mW

Cipher core MARS

El diseño del cifrador/descifrador no pudo ser sintetizado en el FPGA EP2C35F672C6 debido a su gran tamaño. Entonces, se sintetiza el diseño en el FPGA EPC70F896C6, pero estos resultados no se pueden comparar directamente con los resultados de las otras FPGA puesto que son integrados diferentes. Además, como el FPGA EPC70F896C6 no es de la familia Cyclone II, no se puede estimar su potencia de la misma manera que el resto. Se decide incluir los resultados de este *cipher core* sólo para comparar, más no para concluir.

1. El *cipher core MARS*, debido a su tamaño, se implementa en la FPGA EPC70F896C6 y se presentan como datos comparativos, más no para llevar a cabo conclusiones.

Tabla 7.5. Resultados para el *cipher core* MARS

Parámetro	Valor
Tiempo de cifrado	190 ns
Throughput	674 Mbps
Recursos usados	54.845 LE

Cipher core Twofish

Tabla 7.6. Resultados para el *cipher core* Twofish

Parámetro	Valor
Tiempo de cifrado	480 ns
Throughput	266 Mbps
Recursos usados	5.781 LE
Estimación de potencia	263 mW

Cipher core RC6

Tabla 7.7. Resultados para el *cipher core* RC6

Parámetro	Valor
Tiempo de cifrado	4.185 ns
Throughput	30,5 Mbps
Recursos usados	8.181 LE
Estimación de potencia	315 mW

En la Tabla 7.8 se presenta la información recopilada con los resultados de la implementación de estos 5 cifradores.

Tabla 7.8. Parámetros de la implementación de los *cipher core*

Cipher Core	Recursos usados		Velocidad de cifrado		Potencia (mW)
	LE	Porcentaje	Tiempo de cifrado (ns)	*Throughput* (Mbps)	
Rijndael	30.997	93%	180	711	718
Serpent	25.740	77%	480	266	701
Twofish	5.781	17%	480	266	263
Rc6	8.181	25%	4.185	30,5	315
MARS	54.845*	165%**	190*	642,19*	--

*Estos valores corresponden a una implementación hecha en FPGA EPC70F896C6 y se presentan como datos comparativos, mas no para llevar a cabo conclusiones.

** Al comparar con la FPGA EP2C35F672C6.

Diseño de los co-procesadores de cifrado de bloque

Como se mencionó anteriormente, a los cinco *cipher core*: *MARS, RC6, Rijndael, Serpent* y *Twofish*, ses le realizaron las respectivas pruebas de desempeño en cuanto a velocidad de cifrado, potencia consumida y área usada. Estas implementaciones de los algoritmos presentan buses de datos de 128 bits, lo cual es un problema al utilizarlos en un sistema embebido con un procesador central; por esta razón es necesario el diseño de una interfaz que permita integrar estos cifradores, a manera de co-procesadores, en un sistema embebido con procesador central, ya sea de 8, 16, 32 ó 64 bits.

La interfaz, ya integrada, cuenta con entradas de reloj, reset, entrada y salida de datos y un seleccionador de cifrado/descifrado. La Figura 7.4 muestra un diagrama de bloques de la interfaz diseñada y la Tabla 7.5 presenta una descripción de los pines de la interfaz.

Figura 7.4. Entidad del co-procesador

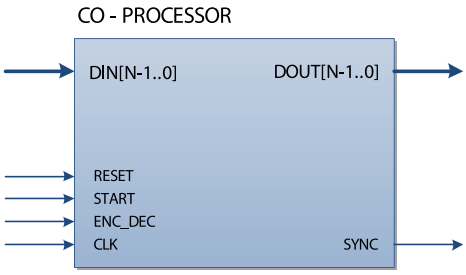
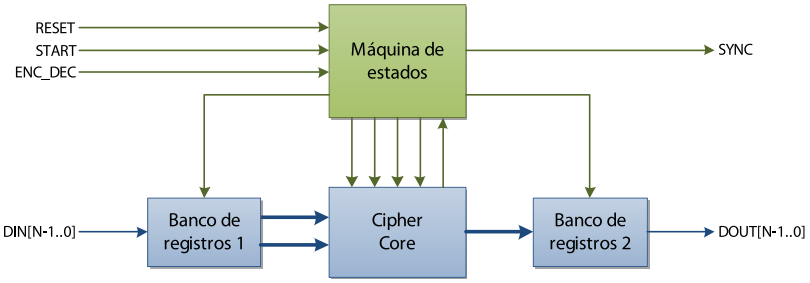


Figura 7.5. Arquitectura del co-procesador



Inicialmente se introducen los datos del menos al más significativo por la entrada de DIN, que puede ser de 8, 16, 32 ó 64 bits, dependiendo de la implementación deseada, y se registran estos datos en el *banco de registros 1*, primero el texto y

posteriormente la clave. Posteriormente los datos son recibidos en paralelo por el *cipher core*, y este resultado se almacena en el *banco de registros 2*, donde es transmitido serialmente por el puesto de salida DOUT. En la Tabla 7.9 se describen las respectivas funciones de las señales de entrada y salida del co-procesador.

Tabla 7.9. Señales del co-procesador

Nombre del puerto	Tamaño en bits	Dirección del puerto	Función
Reset	1	Entrada	Es el *reset* general del sistema. Es una salida activa baja.
Clock	1	Entrada	Esta señal es el reloj externo. El valor mínimo depende de la velocidad de operación del procesador.
En_Dec	1	Entrada	Esta señal indica que clase de función realiza el co-procesador. Si EN_DEC=1, realiza el proceso de cifrado; si EN_DEC=0, realiza el proceso de descifrado.
Entrada	8,16,32 ó 64	Entrada	Este bus corresponde a los datos de entrada. Los datos se reciben de manera serial en bloques de 8 bits, primero el texto plano y posteriormente la clave, iniciando con los datos menos significativos.
Salida	8,16,32 ó 64	Salida	Este bus corresponde a los datos de salida. Los datos salen de manera serial en bloques de 8 bits con los menos significativos de primero.
Sync	1	Salida	Esta señal de salida es una señal de reloj de sincronización que le permite al procesador central sincronizar los datos de salida.

Implementación de los co-procesadores de cifrado de bloque

La implementación de los co-procesadores (exceptuando *MARS*) con la interfaz de N bits, se realiza en la FPGA Cyclone II EP2C35F672C6 de Altera, que tiene un total de 33.216 elementos lógicos. En esta plataforma realizaron simulaciones en software (con el co-procesador sintetizado) y pruebas directamente en la FPGA (con el co-procesador implementado en la misma), utilizando el procesador Nios II. En las Tablas 7.10 a 7.14 se presentan los resultados de la implementación para cada co-procesador.

Co-procesador Rijndael

Tabla 7.10. Resultados para el co-procesador *Rijndael*

Parámetro	N=8	N=16	N=32	N=64
Tiempo de cifrado (ns)	3.968	1.934	996	542
Throughput (Mbps)	32,25	66,18	128,51	236,16
Recursos usados (LE)	31.169	31.143	31.118	31.025
Estimación de potencia (mW)	241	365	485	595

Co-procesador Serpent

Tabla 7.11. Resultados para el co-procesador *Serpent*

Parámetro	N=8	N=16	N=32	N=64
Tiempo de cifrado (ns)	10.248	5.632	3.134	1.665
Throughput (Mbps)	12,49	22,73	40,84	76,88
Recursos usados (LE)	25.735	25.741	24.733	25.749
Estimación de potencia (mW)	171	310	442	569

Co-procesador Twofish

Tabla 7.12. Resultados para el co-procesador Twofish

Parámetro	N=8	N=16	N=32	N=64
Tiempo de cifrado (ns)	2.772	1.208	1.002	731
Throughput (Mbps)	46,18	105,96	127,74	175,10
Recursos usados (LE)	5.943	5.903	5.857	5.809
Estimación de potencia (mW)	164	185	218	233

Co-procesador RC6

Tabla 7.13. Resultados para el co-procesador *RC6*

Parámetro	N=8	N=16	N=32	N=64
Tiempo de cifrado (ns)	8.815	7.022	5.837	4.985
Throughput (Mbps)	14,52	18,23	21,93	25,68
Recursos usados (LE)	8.355	8.291	8.275	8.221
Estimación de potencia (mW)	189	220	261	279

Resultados de co-procesadores de 8-bits

En las Figuras 7.6 a 7.9 se presentan unos histogramas comparativos de los diferentes parámetros de desempeño de los diferentes co-procesadores implementados de 16-bits; y en la Tabla 7.14 se presenta recopilada esta información.

Figura 7.6. Resultados de tiempo de cifrado para co-procesador de 8-bits

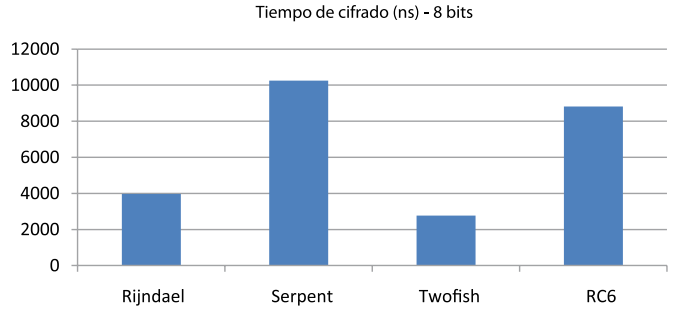


Figura 7.7. Resultados de *throughput* para co-procesador de 8-bits

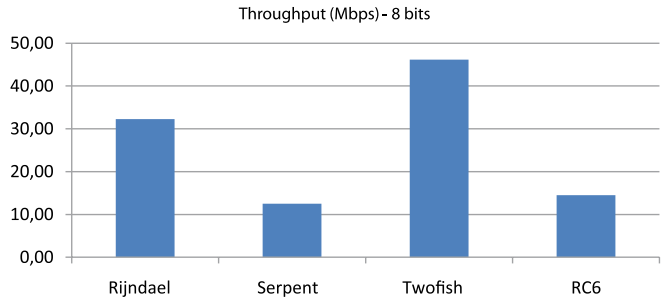


Figura 7.8. Resultados de área para co-procesador de 8-bits

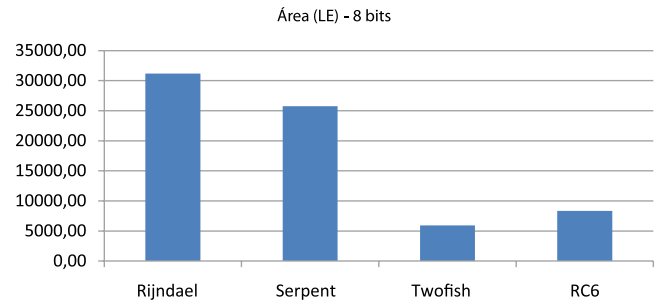


Figura 7.9. Resultados de potencia para co-procesador de 8-bits

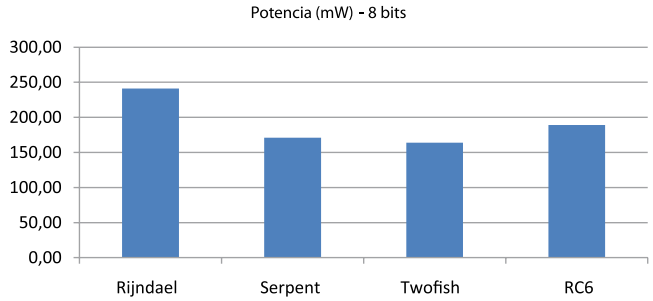


Tabla 7.14. Resultados de la implementación de los co-procesadores de 8-bits

Co-procesador de 8 Bits	Recursos usados		Velocidad de cifrado		Potencia (mW)
	LE	Porcentaje	Tiempo de cifrado (ns)	*Throughput* (Mbps)	
Rijndael	31.169	94%	3.968	32,26	241
Serpent	25.735	77%	10.248	12,49	171
Twofish	5.943	18%	2.772	46,18	164
RC6	8.355	25%	8.815	14,52	189
*MARS**	---	---	---	---	---

* El co-procesador *MARS* no se pudo implementar en la FPGA EP2C35F672C6 debido a su tamaño y por esto no se pueden presentar las métricas de desempeño con la interfaz de 8 bits.

Se puede apreciar que para el caso de un sistema embebido con un procesador central de 8-bits, el co-procesador *Twofish* es el que presenta las mejores métricas de desempeño. Y por lo tanto es el recomendado para este tipo de aplicaciones.

Resultados de co-procesadores de 16-bits

En las Figuras 7.10 a 7.13 se presentan unos histogramas comparativos de los diferentes parámetros de desempeño de los diferentes co-procesadores implementados de 16-bits. Y en la Tabla 7.15 se presenta recopilada esta información.

Figura 7.10. Resultados de tiempo de cifrado para co-procesador de 16-bits

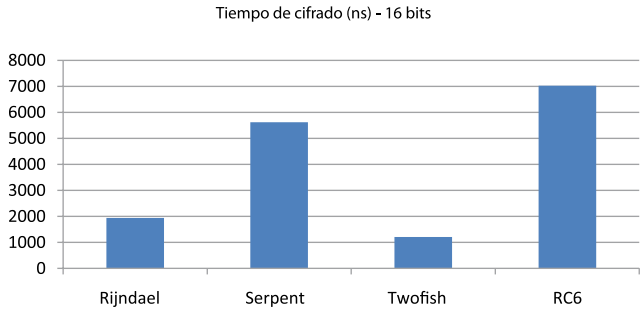


Figura 7.11. Resultados de *throughput* para co-procesador de 16-bits

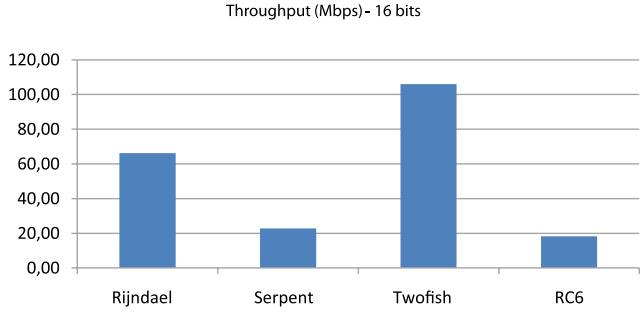


Figura 7.12. Resultados de área para co-procesador de 16-bits

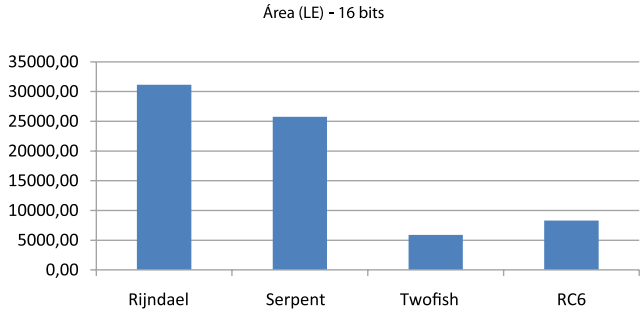


Figura 7.13. Resultados de potencia para co-procesador de 16-bits

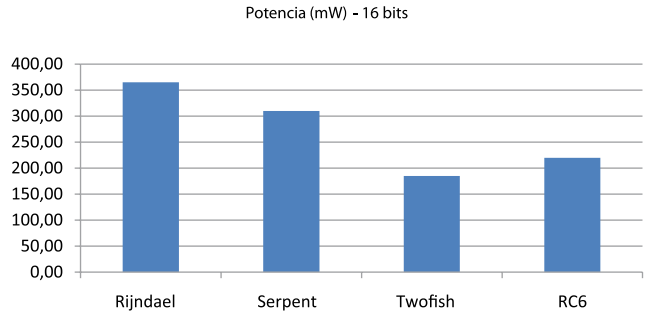


Tabla 7.15. Parámetros de la implementación de los co-procesadores

Co-procesador de 16 bits	Recursos usados		Velocidad de cifrado		Potencia (mW)
	LE	Porcentaje	Tiempo de cifrado (ns)	*Throughput* (Mbps)	
Rijndael	31.143	94%	1.934	66,18	365
Serpent	25.741	77%	5.632	22,73	310
Twofish	5.903	18%	1.208	105,96	185
RC6	8.291	25%	7.022	18,23	220
*MARS**	---	---	---	---	---

* El co-procesador *MARS* no se pudo implementar en la FPGA EP2C35F672C6 debido a su tamaño, y por esto no se pueden presentar las métricas de desempeño con la interfaz de 16 bits.

Se puede apreciar que para el caso de un sistema embebido con un procesador central de 16-bits, el co-procesador *Twofish* es el que presenta las mejores métricas de desempeño. Y por lo tanto es el recomendado para este tipo de aplicaciones.

Resultados de co-procesadores de 32-bits

En las Figuras 7.14 a 7.17 se presentan unos histogramas comparativos de los diferentes parámetros de desempeño de los diferentes co-procesadores implementados de 32-bits. Y en la Tabla 7.16 se presenta recopilada esta información.

Figura 7.14. Resultados de tiempo de cifrado para co-procesador de 32-bits

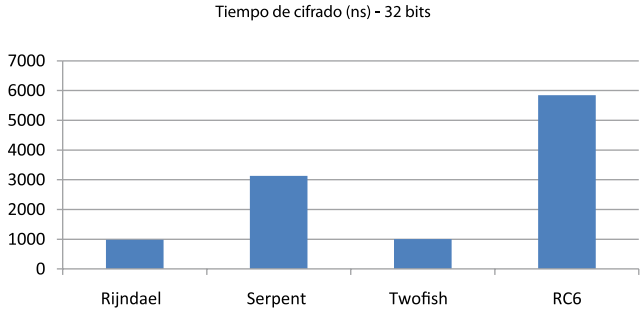


Figura 7.15. Resultados de *throughput* para co-procesador de 32-bits

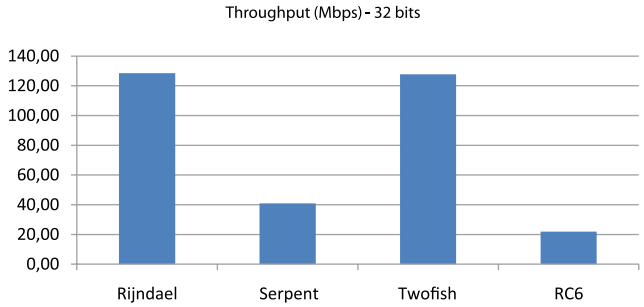


Figura 7.16. Resultados de área para co-procesador de 32-bits

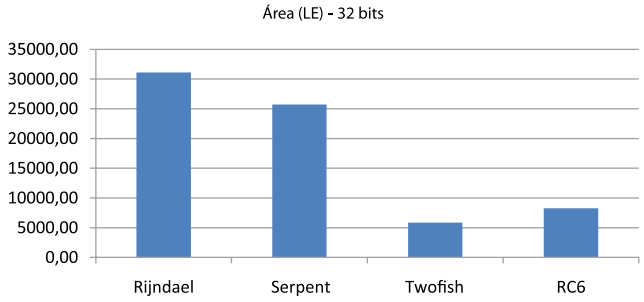


Figura 7.17. Resultados de potencia para co-procesador de 32-bits

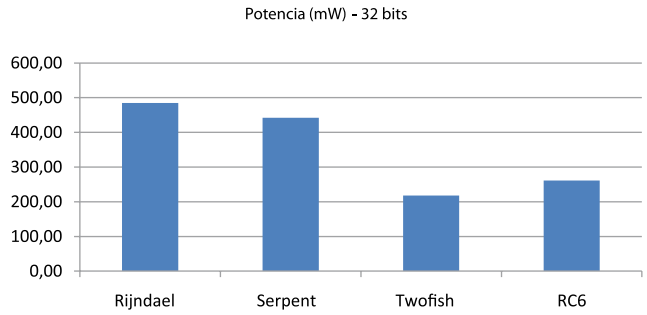


Tabla 7.16. Parámetros de la implementación de los co-procesadores

Co-procesador de 32 Bits	Recursos usados		Velocidad de cifrado		Potencia (mW)
	LE	Porcentaje	Tiempo de cifrado (ns)	*Throughput* (Mbps)	
Rijndael	31.118	94%	996	128,51	485
Serpent	25.733	77%	3.134	40,84	442
Twofish	5.857	18%	1.002	127,74	218
RC6	8.275	25%	5.837	21,93	261
MARS*	---	---	---	---	---

* El co-procesador MARS no se pudo implementar en la FPGA EP2C35F672C6 debido a su tamaño, y por esto no se pueden presentar las métricas de desempeño con la interfaz de 32 bits.

Se puede apreciar que para el caso de un sistema embebido con un procesador central de 8-bits, el co-procesador *Twofish* es el que presenta las mejores métricas de desempeño en cuanto a recursos usados y potencia. En cuanto a tiempo de cifrado y *throughput* tanto *Twofish* como *Rijndael* presentan las mejores métricas.

Resultados de co-procesadores de 64-bits

En las Figuras 7.18 a 7.21 se presentan unos histogramas comparativos de los diferentes parámetros de desempeño de los diferentes co-procesadores implementados de 64-bits. Y en la Tabla 7.17 se presenta recopilada esta información.

Figura 7.18. Resultados de tiempo de cifrado para co-procesador de 64-bits

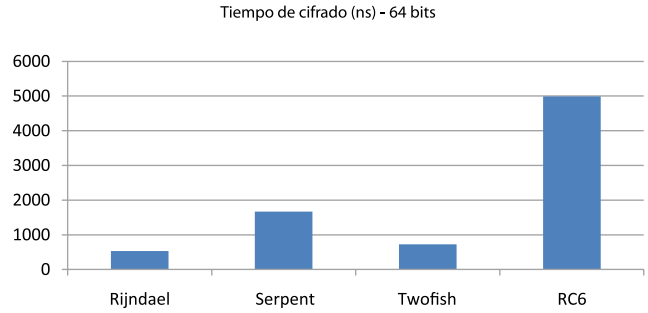


Figura 7.19. Resultados de *throughput* para co-procesador de 64-bits

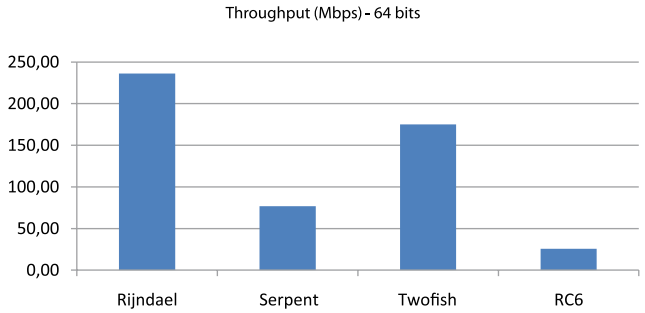


Figura 7.20. Resultados de área para co-procesador de 64-bits

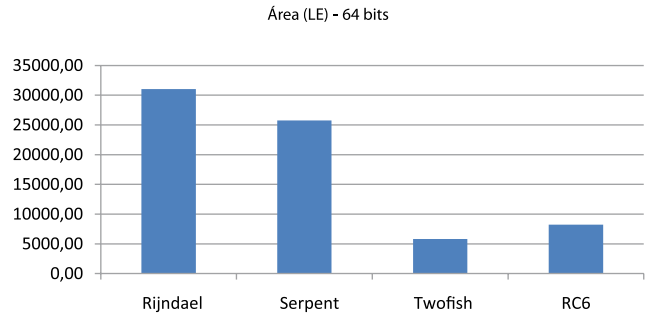


Figura 7.21. Resultados de potencia para co-procesador de 64-bits

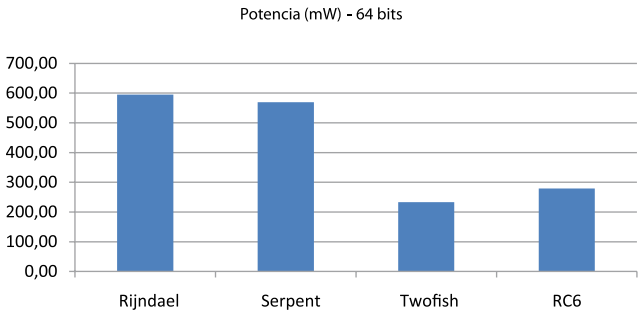


Tabla 7.17. Parámetros de la implementación de los co-procesadores

Co-procesador de 64 Bits	Recursos usados		Velocidad de cifrado		Potencia (mW)
	LE	%	Tiempo de cifrado (ns)	*Throughput* (Mbps)	
Rijndael	31.025	93%	542	236,16	595
Serpent	25.749	77%	1665	76,88	569
Twofish	5.809	17%	731	175,10	233
RC6	8.221	25%	4985	25,68	279
MARS*	---	---	---	---	---

* El co-procesador MARS no se pudo implementar en la FPGA EP2C35F672C6 debido a su tamaño, y por esto no se pueden presentar las métricas de desempeño con la interfaz de 64 bits.

Se puede apreciar que para el caso de un sistema embebido con un procesador central de 8-bits, el co-procesador *Twofish* es el que presenta las mejores métricas de desempeño en cuanto a recursos usados y potencia. En cuanto a tiempo de cifrado y *throughput* es *Rijndael* el que presentan las mejores métricas.

Co-procesadores recomendados

Se presenta entonces en la Tabla 7.18, un cuadro con los coprocesadores recomendados dependiendo de la métrica de interés y acorde al tamaño del bus de datos del procesador central de un sistema embebido.

Tabla 7.18. Co-procesadores recomendados

Parámetro	N=8	N=16	N=32	N=64
Tiempo de cifrado	*Twofish*	*Twofish*	*Twofish/ Rijndael*	*Rijndael*
Throughput	*Twofish*	*Twofish*	*Twofish/ Rijndael*	*Rijndael*
Recursos usados	*Twofish*	*Twofish*	*Twofish*	*Twofish*
Potencia	*Twofish*	*Twofish*	*Twofish*	*Twofish*

Bibliografía

- AGUDELO TORRES, Ivonne (2007). *Implementación algoritmo Rijndael*. Trabajo de grado, pregrado. Universidad de San Buenaventura Cali. Facultad de Ingeniería. Programa de Ingeniería Electrónica.
- ALTERA (2006). *Powerplay early power estimator. User guide* - for Cyclone II FPGAs. White paper Version 1.1.
- American National Standards Institute. American National Standard X3.106: *Data encryption algorithm, modes of operations*. 1983.
- ANDERSON, R.; BIHAM, E. & KNUDSEN, L. (1998). *Serpent: a proposal for the advanced encryption standard*. NIST AES Proposal.
- ÁNGEL, José de Jesús (2005). *AES – Advanced Encryption Standard. Version 2005, principiantes*. Technical report. México: Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional - CINVESTAV.
- ÁNGEL, José de Jesús (2000). *Criptografía simétrica y asimétrica*. Technical report. México: Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional – CINVESTAV.
- ASHENDEN P. (1990). *The VHDL cookbook*. Australia: Dept. Computer Science, University of Adelaide, South Australia. First edition.
- BIHAM, E. (1995). *Cryptanalysis of multiple modes of operation*. In Advances in Cryptology - Asiacrypt ‹94, Springer-Verlag.
- BROWN, S. & ROSE J. (2002). *FPGA and CPLD architecture: a tutorial*, Journal Design & Test of Computers, IEEE. Vol. 13. No. 2.

– BURTON, S.; KALISKI Jr & YIQUN L. Y. (1998). *On the security of the RC5 encryption algorithm*. Technical report TR-602, Version 1.0. RSA Laboratories.

– BURWICK, C.; COPPERSMITH D.; D'AVIGNON, E.; GENNARO, R.; HALEVI, Shai; JUTLA C.;, MATYAS, S. M. Jr.; O'CONNOR, L.; PEYRAVIAN, M; SAFFORD, D. & ZUNIC, N. (1999). *MARS a candidate cipher for AES*. IBM Corporation.

– CARPIO PARDO, F. (1997). *VHDL lenguaje para descripción y modelado de circuitos*. México: Editorial Alfaomega.

– CHODOWIEC, Pawel; GAJ, Kris (1999). *Implementation of the Twofish cipher using FPGA devices*. Technical Report. Electrical and Computer Engineering, George Mason University.

– CrypTool - Herramienta Educativa para Criptografía y Criptoanálisis (Version 1.4.21 Septiembre 2009). Disponible en <https://www.cryptool.org/>

– DAEMEN, Joan and RIJMEN Vincent (2000). *The Rijndael block cipher*. Smart Card - Research and Applications. Lecture Notes in Computer Science. Springer Berlin / Heidelberg.

– _____; KNUDSEN, L.R. & RIJMEN, V. (1997). *The block cipher Square*. Fast software encryption, LNCS 1267, E. Biham, Ed. Springer-Verlag, pp. 149-165.

– Federal Information Processing Standards Publication 197 (2001). *Advanced encryption standard (AES)*. Noviembre 26.

– GARCIA, A. (2000). *Power consumption and optimization in field programmable gate arrays*. PhD dissertation. Ecole Nationale Supérieure des Télécommunications. Paris.

– KAMMER, Raymond G. (1993). *Data encryption standard (DES)*. U.S. Department of Commerce, National Institute of Standards and Technology. FIPS PUB 462.

– KUZNETSOV, Grigori; KARRI, Ramesh & GOSSEL, Michael (2002). *Error detection by parity modification for the 128-bit Serpent encryption algorithm*. Polytechnical University Brooklyn, USA. University of Potsdam, Germany.

– MANTEENA, Rajender (2000). *A VHDL implementation of the advanced encryption standard-Rijndael algorithm*. Tésis de Maestría. University of South Florida. College of Engineering. Department of Electrical Engineering.

– MEDINA OLIVEROS, Jorge Alberto (2007). *Implementación algoritmo Serpent*. Trabajo de grado, pregrado. Universidad de San Buenaventura, Cali. Facultad de Ingeniería. Programa de Ingeniería Electrónica.

– MENEZES, A.; VAN OORSCHOT, P. & VANSTONE, S. (1996). *Handbook of applied cryptography*. Editorial CRC Press.

– RIVEST, Ronald L.; ROBSHAW, M.J.B.; SIDNEY, R. & YIN, Y.L. (1998). *The RC6 block cipher*. First Advanced Encryption Standard (AES) Conference.

– RIVERA FIERRO, David Guillermo (2008). *Diseño y simulación algoritmo Twofish*. Trabajo de grado, pregrado. Universidad de San Buenaventura, Cali. Facultad de Ingeniería. Programa de Ingeniería Electrónica.

– SCHNEIER, Bruce (2000). A *self-study course in block cipher cryptanalysis*. Cryptologia, Vol.24, No.1, pp. 18-34.

– _____; KELSEYY, John; WHITINGZ, Doug; WAGNERX, David; HALL, Chris & FERGUSON, Niels (1998). *Twofish: A 128-bit clock cipher*. NIST – AES.

– TORRES VALLE, F. J. (2008). *Síntesis de circuitos digitales utilizando VHDL (Vhsic hardware description language) y FPGAs (Field programmable gate arrays)*. Editorial Quito/EPN.

– VALENCIA SOLARTE, Diana Fernanda (2007). *Diseño y simulación del algoritmo Mars*. Trabajo de grado, pregrado. Universidad de San Buenaventura, Cali. Facultad de Ingeniería. Programa de Ingeniería Electrónica.

– VÉLEZ GORDILLO, Manuel Alejandro (2010). *Diseño y simulación algoritmo RC6*. Trabajo de grado, pregrado. Universidad de San Buenaventura, Cali. Facultad de Ingeniería. Programa de Ingeniería Electrónica. Preprint.

– VENTE MANCILLA, Ana Elvira (2009). *Optimización, implementación y pruebas de rendimiento de los esquemas de encriptación finalistas del AES*. Trabajo de grado, pregrado. Universidad de San Buenaventura, Cali. Facultad de Ingeniería. Programa de Ingeniería Electrónica.

Durante el año 2001, el *National Institute of Standards and Technology*-NIST, llevó a cabo el proceso de coordinación para reemplazar el algoritmo Data Encryption Standard-DES, algoritmo nacido en 1974 que trabaja sobre bloques de 64 bits .

En este proceso de selección del nuevo algoritmo para reemplazar al DES y convertirse en el *Advanced Encryption Standard*-AES, sólo cinco algoritmos candidatos fueron seleccionados por NIST como finalistas: *MARS*, *RC6*, *Rijndael*, *Serpent* y *Twofish*. Estos candidatos fueron evaluados bajo los parámetros de diseño, codificación, simulación y síntesis. Finalmente, el algoritmo *Rijndael* fue seleccionado para ser el *Advanced Encryption Standard* debido a su seguridad, desempeño, eficiencia, flexibilidad y facilidad de implementación.

Es de notar que durante el proceso de selección del AES los algoritmos finalistas fueron implementados en software y algunos de estos algoritmos están diseñados para ser más eficientes en su implementación en software, que en hardware.

Son necesarias, entonces, las implementaciones en hardware de estos algoritmos para comparar diferentes características que estos presentan al momento de ser implementados en hardware, como por ejemplo en un dispositivo lógico programable (*PLD-Programmable Logic Device*). Entre algunas de estas características encontramos: seguridad de la información, velocidad de cifrado, consumo de potencia y recursos usados en el PLD.

Por esta razón el grupo de investigación Laboratorio de Electrónica Aplicada – LEA, de la Universidad de San Buenaventura, seccional Cali, se puso a la tarea de implementar en hardware estos algoritmos a manera de co-procesadores dedicados a cifrar y descifrar datos. La implementación se realiza en un PLD usando captura esquemática y un lenguaje de descripción de hardware (*HDL -Hardware Description Language*).

GRUPO DE INVESTIGACIÓN
LEA

ISBN: 978-958-8436-45-6
9 789588 436456

Universidad de San Buenaventura, seccional Cali.
La Umbría, carretera a Pance
PBX: 318 22 00 – 448 22 22. Fax: 555 20 06
A.A. 7154 y 25162.
www.usbcali.edu.co