UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

DESARROLLO DE ENCRIPTADO AES EN FPGA

Tesis presentada para obtener el grado de
Magíster en Redes de Datos

Autor: Mónica C. Liberatori

Director: Oscar N. Bría
Codirector: Horacio Villagarcía

Febrero 2006

Abstract

The Rijndael cipher, designed by Joan Daemen and Vincent Rijmen and recently selected as the official Advanced Encryption Standard (AES) is well suited for hardware use. This implementation can be carried out through several trade-offs between area and speed.

This thesis presents an 8-bit FPGA implementation of the 128-bit block and 128 bit-key AES cipher. Selected FPGA Family is Altera Flex 10K. The cipher operates at 25 MHz and consumes 470 clock cycles for algorithm encryption, resulting in a throughput of 6.8 Mbps. Synthesis results in the use of 460 logic cells and 4480 memory bits.

The VHDL code was simulated using the test vectors provided in the AES submission package. The results are functionally correct. The architecture needs fewer logic cells than other ciphers and uses as few memory blocks as possible. The design goals were area and cost optimisation.

TABLA DE CONTENIDOS

Capítulo 1

Introducción

La importancia de la criptografía aplicada a la seguridad de las transacciones electrónicas de datos ha adquirido fundamental relevancia en los últimos tiempos. Cada día millones de usuarios generan e intercambian grandes volúmenes de información en diversos campos, tales como archivos financieros y de índole legal, informes médicos, servicios bancarios a través de Internet, conversaciones telefónicas y transacciones de comercio electrónico. Estos y otros ejemplos de aplicaciones merecen un tratamiento especial desde el punto de vista de la seguridad no sólo en el transporte de dicha información, sino también en lo que respecta al almacenamiento de la misma. El uso de técnicas de criptografía es especialmente aplicable en este sentido.

Muchas han sido las propuestas para establecer estándares de implementación aplicables a las dos grandes ramas de la criptografía: clave simétrica y clave pública. Durante muchos años, uno de ellos, *Data Encryption Standard* (*DES*), dominó el área de la criptografía de clave simétrica. El avance tecnológico en lo que respecta a la velocidad de procesamiento de los datos puso a *DES* en una situación de vulnerabilidad debido al tamaño de su clave. Esto motivó su reemplazo como estándar.

A principios de este siglo, el *National Institute of Standards and Technology* (*NIST*) comenzó a trabajar en la elaboración de un nuevo estándar conocido como *Advanced Encryption Standard* (*AES*). Su objetivo final era el desarrollo del llamado *Federal Information Processing Standard* (*FIPS*) que especificara un algoritmo de cifrado capaz de proteger información sensible con el propósito de ser usado por el gobierno de los Estados Unidos. La competencia entre los finalistas fue muy importante y, luego de un proceso de evaluación complejo, en octubre de *2000*, el *NIST* seleccionó el *Rijndael* como el algoritmo propuesto para *AES*.

Una de las diferencias más impactantes de este algoritmo respecto de sus predecesores es la ausencia de una estructura tipo *Feistel*. En su lugar se adoptó una forma de ronda

compuesta de tres transformaciones uniformes e invertibles que garantizan difusión sobre el conjunto total de rondas fijadas y propiedades de alinealidad óptimas. Esta es la razón de su fortaleza frente a ataques conocidos.

Respecto de sus aspectos prácticos de realización, el algoritmo resulta especialmente atractivo para implementación eficiente sobre un amplio rango de procesadores y hardware dedicado. Por otra parte, debido al gran rango de aplicaciones que puede soportar, se torna importante la capacidad de diseñar productos que lo implementen y que ofrezcan diferentes características. Por ejemplo, aquellos capaces de ofrecer una rápida capacidad de procesamiento se podrían considerar para redes privadas virtuales (*VPN*) y los productos compactos ofrecerían una solución aplicable a aplicaciones inalámbricas.

Es importante destacar en este sentido las innovaciones introducidas en el diseño electrónico de la década de los ochenta. Las mismas se apoyaron en el desarrollo de nuevas tecnologías y alternativas de fabricación y diseño de circuitos integrados. A esto se sumó la evolución de metodologías y herramientas de diseño asistido por computadora (*CAD*, *Computer Aided Design*), que han ampliado las posibilidades de los ingenieros de aplicación, permitiéndoles diseñar chips con fines específicos (*ASICs*, *Application Specific Integrated Circuits*) para los productos que desarrollan.

Entre una nueva serie de alternativas se destaca la categoría de *Lógica Programable* (*FPGA*, *CPLD*) que consta de dispositivos totalmente fabricados y verificados que se pueden personalizar desde el exterior mediante diversas técnicas de programación, tales como *RAM* y fusibles. El diseño se basa en bibliotecas y mecanismos específicos de mapeo de funciones. La implementación requiere de una fase de programación del dispositivo que generalmente realiza el propio diseñador muy rápidamente.

En general, las metodologías de diseño son de tipo descendente *top-down* y se apoyan en lenguajes de descripción de hardware (*Verilog* y *VHDL*) que se usan en conjunto con herramientas de simulación y síntesis, concentrando el esfuerzo en la concepción funcional y

arquitectural y la evaluación de soluciones alternativas antes de abordar el diseño detallado y la implementación física.

Desde el punto de vista de la sintaxis, los *HDLs* (*Hardware Description Language*) adoptaron conceptos de la ingeniería de software para descripción y modelado de hardware. Son similares a los lenguajes de programación de alto nivel (*HLL*, *High Level Languaje*) para el desarrollo de software y muchos derivan de ellos, por ejemplo *Verilog* se parece mucho a *C* y *VHDL* procede de *ADA*. Nacen para modelar el comportamiento de un componente para su simulación y posiblemente posterior implementación. También permiten describir los circuitos a un nivel alto de abstracción independiente de la implementación tecnológica final. A partir de esas descripciones los procesos de diseño descendente aplican procedimientos progresivos de síntesis hasta alcanzar una descripción física concreta independiente de la tecnología seleccionada. La validación de las descripciones va de la mano de la simulación y las iteraciones de las correcciones resultantes.

Desde el punto de vista de simulación y síntesis los niveles de abstracción con *HDL* se reajustan al comportamiento del circuito o sistema como relación funcional entrada-salida, la partición en bloques funcionales con consideraciones de tiempos y, por último, la especificación de componentes como ecuaciones o elementos de biblioteca independiente o no de la tecnología. Los factores que caracterizan la precisión de las descripciones se concentran en la relación temporal, en el funcionamiento del circuito y los tipos de datos definidos por el usuario. A su vez el estilo de las descripciones puede adoptar varias formas que van desde el estilo algorítmico muy utilizado a nivel funcional, pasando por el de flujo de datos, más relacionado con el nivel arquitectural o dándole un sentido estructural a través de referencias y conexiones.

Por todas las características mencionadas *FPGA* se presenta como la mejor alternativa de diseño para algoritmos como el *AES* debido fundamentalmente a su potencial para reprogramación rápida y de bajo costo. Esta propiedad se asocia por ejemplo a la experimentación de numerosas arquitecturas o variantes de una misma arquitectura.

Por estos motivos, el propósito del presente trabajo de investigación se centra en la fase

de cifrado, particularmente su desarrollo en un lenguaje de programación adecuado para *FPGA*, y con el objetivo de minimizar los recursos de hardware necesarios para su implementación. Una respuesta en este sentido parece encontrar su aplicación más próxima en la incorporación de este algoritmo como una alternativa de cifrado más poderosa en el estándar *802.11i*. Una implementación *FPGA* eficiente en términos de área parece adecuada a este problema. La elección de una arquitectura apropiada a la finalidad perseguida es fundamental en el sentido de la velocidad de procesamiento, ya que ambos objetivos, minimización de área y maximización de velocidad, se contraponen.

Cabe destacar que la plataforma de desarrollo a utilizar es básica, diseñada para aplicaciones de educación en el campo de diseño de circuitos digitales. El dispositivo disponible como objetivo de síntesis es un dispositivo de Altera de muy bajo volumen, característica que se traduce en un número muy bajo de celdas lógicas y memoria embebida en comparación con plataformas más nuevas. Esta propiedad representa un desafío si se compara con implementaciones conocidas del Algoritmo.

La tesis se desarrolla en dos secciones principales. La primera de ellas se extiende entre los Capítulos 2 a 4 y se ha dedicado a destacar los aspectos teóricos relevantes al cifrador Rijndael. La segunda sección ocupa los Capítulos 5 a 9 y en la misma se aborda el tema de la implementación en hardware.

En la primera sección el lector encontrará una introducción a la Criptografía (Capítulo 2), un resumen de los principios algebraicos básicos que sostienen las operaciones en campos finitos (Capítulo 3) y una descripción completa de la especificación del cifrador (Capítulo 4). Los primeros dos temas se debieron abordan desde una perspectiva resumida dada la extensión y complejidad de los mismos. Básicamente, en la introducción a la Criptografía se han resaltado aquellos aspectos de la Criptografía de Clave Simétrica más relacionados con el Algoritmo Rijndael. En cuanto a la presentación de los principios matemáticos se intenta avanzar desde los conceptos básicos hacia los más complejos sin presentar pruebas de teoremas pero facilitando abundantes referencias útiles para aquellos lectores que decidan adentrarse en el tema.

La sección dedicada a la implementación en hardware comienza en el Capítulo 5 donde se efectúa un análisis completo de las diferentes transformaciones asociadas al cifrador y la posibilidad de implementación de las mismas. Se explica en este Capítulo la estrategia y metodología de diseño utilizada, haciendo hincapié en las diferentes opciones de arquitectura conocidas y fundamentando la elección realizada.

En el Capítulo 6 se realiza una explicación detallada del circuito final, sus diversos componentes e interconexiones. Esta descripción se completa en el Capítulo 7, donde se ofrece un análisis pormenorizado del funcionamiento tomando como ejemplo los diversos resultados obtenidos de la simulaciones.

El Capítulo 8 presenta los resultados obtenidos, su aplicabilidad en términos de la implementación de un circuito descifrador y una comparación con otras implementaciones conocidas. Se destaca en este capítulo la concreción del objetivo buscado en la presente investigación: el diseño de un cifrador AES-128 de área mínima y, por ende, de bajo costo.

En el último capítulo, el Capítulo 9, se presenta el análisis final de este trabajo. De este modo, la tesis concluye con un resumen de los resultados y una serie de sugerencias para trabajos futuros.

Capítulo 2

Introducción a la Criptografía

Históricamente la idea común asociada a la palabra Criptografía se relaciona con la posibilidad de mantener una comunicación de manera secreta o privada entre dos partes a través de algún vínculo. De hecho, es en este objetivo que se ha puesto el mayor énfasis a lo largo de la historia de la protección de estrategias y secretos de índole militar, nacional o empresarial: la *confidencialidad*. El impactante crecimiento de la tecnología, ocurrido en los últimos cincuenta años, ha afectado enormemente a las comunicaciones en redes de datos y su efecto principal se ha reflejado en la aparición de nuevos requerimientos en aquellos aspectos relacionados con la protección de datos en formato digital y los servicios de seguridad en general. El concepto de información y las cuestiones relativas a la seguridad de la información han dado lugar a la aparición de diversas técnicas, algoritmos matemáticos y procedimientos que se aplican tanto al almacenamiento seguro de la información como al traslado protegido de la misma.

2.1 Definiciones Básicas

El nuevo conjunto de herramientas ofrecidas por la propia evolución tecnológica se mostró capaz de avanzar varios pasos más allá del propósito original relacionado con la *confidencialidad.* De este modo, comenzó a ser posible proporcionar además otros aspectos relacionados con la seguridad de la información: *autenticación*, *integridad y no repudio* [STA99]. Se instaló de esta manera un concepto más moderno de *Criptografía* y se redefine su alcance como el estudio de las técnicas matemáticas relacionadas a aspectos de la seguridad de la información, incluyendo así los nuevos requerimientos mencionados.

Los objetivos de confidencialidad, autenticación, integridad y no repudio, aplicados a un esquema general de comunicación, conforman un entorno de trabajo en el que se pretende que sea posible para el receptor de un mensaje cerciorarse de su origen y verificar que no ha sido modificado en tránsito. Desde el extremo transmisor la propiedad de no repudio evita que aquel

usuario del sistema que haya enviado el mensaje no pueda más tarde denegarlo falsamente. Desde el punto de vista del canal, el mensaje no debe ser comprendido por algún actor diferente del transmisor o receptor.

Con estos conceptos presentes se puede definir como *Técnica de Encriptado o Cifrado* a aquella transformación a la que se vea sujeto un mensaje de tal manera de convertirlo en algún otro, irreconocible para las partes no interesadas, y conocido generalmente como *criptograma*, *texto cifrado o texto encriptado*. El mensaje original, en contraposición, se conoce como *texto pleno*. Toda Técnica de Cifrado se asocia a otra inversa de *Desencriptado o Descifrado* que permite recuperar el mensaje original a partir del criptograma. La utilidad principal de estas transformaciones es la confidencialidad.

Una herramienta criptográfica fundamental en términos de autenticación y no repudio es la *firma digital* [MEN97]. Su propósito es proveer un medio a cualquier entidad para ligar su identidad a la información a transmitir. El proceso de firma consiste en transformar el mensaje y alguna información secreta que posea la entidad transmisora en un apéndice de características especiales y de longitud fija que se ha de adicionar al mensaje original en el momento de su transmisión. Dicho apéndice se conoce como *firma* y se somete a un proceso de verificación al momento de la recepción.

La propiedad de integridad aplicada a datos es la capacidad adicional ofrecida para asegurar que los mismos no han sido alterados desde el momento de su creación, al ser transmitidos o mientras permanezcan almacenados, por ninguna fuente no autorizada. Su implementación práctica se realiza por medio de funciones *hash* [MEN97], generalmente conocidas como de una sola vía, en inglés *one-way hash functions*.

Los *Algoritmos Criptográficos* son funciones matemáticas usadas tanto por las técnicas para cifrado y descifrado de datos, como por aquellas herramientas relacionadas con la firma digital y las funciones de integridad. En la criptografía moderna estos algoritmos son públicamente conocidos y su seguridad se apoya en el conocimiento de algún elemento secreto llamado *llave* o *clave*. El conjunto todos los posibles textos plenos o mensajes a cifrar,

textos cifrados o criptogramas, algoritmos utilizados y claves almacenadas, con aplicación en un contexto particular, se conoce como *criptosistema*.

Existen dos tipos generales de algoritmos basados en claves: *Simétricos* o de *Clave Privada* y *Asimétricos* o de *Clave Pública* [STI95].

En los sistemas *Simétricos* los participantes en un protocolo de comunicación criptográfico comparten una única clave y la seguridad del sistema descansa en el secreto de dicha clave. Uno de los problemas más importantes que enfrentan estos sistemas es encontrar un método eficiente para intercambiar la clave de manera segura [MIT92]. A su vez, se distinguen dos tipos esquemas de criptografía de clave simétrica que dan lugar a dos clases de cifradores: *bloque* y *de flujo continuo* (en inglés *stream*) [SCH96].

Los *cifradores de bloque* dividen el mensaje de texto pleno a transmitir en bloques de tamaño fijo y cifran un bloque por vez. En este sentido poseen similitudes con códigos para corrección de errores [ADA91]. Existen dos clases de cifradores de bloque: *por sustitución* y *por transposición* [STA99].

Los *cifradores por sustitución* simplemente reemplazan símbolos por otros símbolos o grupos de símbolos. El problema de este tipo de cifradores es que en el texto cifrado se reflejan las propiedades de frecuencia de aparición de determinadas letras que caracteriza al idioma particular del texto pleno. Los *cifradores por transposición* realizan permutaciones de símbolos que pertenecen al mismo bloque. Obviamente, este tipo de cifradores no proveen un nivel de seguridad elevado cuando se utilizan de manera individual. Sin embargo, la combinación de ambos permite obtener cifradores más fuertes en términos de seguridad criptográfica. De esta manera surgen los *cifradores producto* que combinan operaciones de sustitución y de transposición y a dicha combinación la denominan ronda, en inglés *round*. Las operaciones de sustitución en una ronda de un cifrador producto adicionan *confusión*, en inglés *confusion*, al proceso de cifrado [STA99]. El propósito es que la relación entre clave y texto cifrado sea la más compleja posible. A su vez, las operaciones de transposición adicionan *difusión*, en inglés *diffusion*, al proceso de cifrado con el objetivo de reacomodar los bits del mensaje para que

cualquier redundancia en el texto pleno se desparrame en el texto cifrado [STA99]. De esta manera una ronda agrega ambas propiedades al esquema de cifrado. La mayoría de los cifradores de bloque modernos aplican una serie de rondas en sucesión al cifrado del texto original.

Por su parte, los *cifradores de flujo continuo* son una clase particular de cifradores bloque que utilizan un tamaño de bloque unitario. De este modo es factible cambiar la transformación de cifrado o la clave particular para cada símbolo de texto pleno a cifrar. Son muy aplicables en situaciones donde la probabilidad de error de la transmisión es muy elevada ya que no poseen errores de propagación [MEN97].

En general, una condición necesaria pero no suficiente para que el esquema de cifrado simétrico sea seguro, es que el espacio de la clave sea lo suficientemente amplio como para desalentar una búsqueda exhaustiva. Se denomina espacio de la clave justamente al conjunto de todas las claves posibles. Su número depende del tamaño asignado por el esquema a la clave. De este modo, una clave de *56* bits pertenece a un espacio de 2^{56} claves posibles.

Uno de los desarrollos más impactante en la historia de la Criptografía ocurrió a mediados de la década del setenta, cuando Diffie y Hellman introdujeron el revolucionario concepto de *Criptografía de Clave Pública* [SCH96]. En los sistemas de clave pública, cada usuario posee dos claves, una pública y otra privada. Ambas se encuentran relacionadas entre sí de tal manera que, conocida una de ellas, no es posible descubrir la otra. Los protocolos criptográficos se sirven de ambas para realizar la comunicación. El cifrado de los datos se realiza con la clave pública que podría ser enviada sin inconvenientes a través de un canal inseguro. Para descifrar los datos correctamente es necesario conocer la clave secreta o privada. La criptografía de clave pública también dio origen a varios esquemas de firma digital.

De esta manera pareciera ya no existir el problema de distribución de la clave [MIT92] que afecta a los sistemas simétricos, aunque en realidad las dificultades pasan a concentrarse alrededor de la autenticidad del poseedor de la clave pública pues existen posibles escenarios

de falsedad de una de las partes, en inglés *impersonation*. Una solución posible es la existencia de *centros de certificación* de claves.

A lo largo de los años ambos tipos de Criptografía, la de clave simétrica y la de clave pública, han evolucionado y su aplicación es habitual en muchas comunicaciones de uso cotidiano. A la vez, el crecimiento explosivo de Internet ha dado lugar a nuevos esquemas de *seguridad en redes* que se apoyan en alguno de estos algoritmos o en combinaciones de ambos tipos [SMI95].

2.2 Criptografía clásica

Se suele considerar criptografía clásica a la anterior a las técnicas de cifrado en bloque y a la criptografía de clave pública. Este tipo de criptografía se ocupa casi exclusivamente de la *privacidad*. Suele trabajar con el alfabeto, y las transformaciones de cifrado y descifrado involucran sustituciones y transposiciones.

En los esquemas de sustitución cada letra del mensaje se sustituye por otra según una determinada permutación del alfabeto. En las *sustituciones monoalfabéticas* se aplica la misma permutación a todas las letras del mensaje. En las *polialfabéticas* se usan diferentes permutaciones para diferentes letras del mensaje. Las primeras son sencillas de romper pues el método de cifrado no logra esconder la estadística subyacente en la aparición de diferentes letras del texto pleno. Las segundas tienen su ejemplo de aplicación en los famosos cifradores de *Vigenère* y de *Beaufort* [STI95] y en general el análisis criptográfico aplicado a los mismos es un poco más complejo que en el caso monoalfabético.

En los esquemas de transposición las letras del mensaje se mezclan entre sí según diferentes pautas. Por ejemplo podría tratarse de un arreglo geométrico donde se escribe el mensaje en un rectángulo por filas y se lee por columnas al momento de su transmisión. Casi todos los sistemas criptográficos basados en transposiciones solamente se pueden romper con un análisis de frecuencias de letras debido a regularidades estadísticas del mensaje [STA99].

Se dice que Julio Cesar cifraba sus mensajes por sustitución. Cada letra de un texto era reemplazada con una nueva letra obtenida por un desplazamiento de tres posiciones a la derecha en el alfabeto, considerado en forma circular. En la Fig. 2.1 se presenta un ejemplo de mensaje y criptograma asociado. Cada letra del mensaje se reemplaza por la que corresponde a tres posiciones posteriores en el alfabeto. Se trata de una sustitución monoalfabética. La clave del cifrado es simplemente el número de desplazamiento en el alfabeto.

Texto pleno:	A	H	O	R	A	E	S	E	L	M	O	M	E	N	T	O
Texto cifrado:	D	K	R	U	D	H	V	H	O	P	R	P	H	Q	W	R

Alfabeto:	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Alfabeto cifrado:	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

Figura 2.1 Alfabeto Cesar con desplazamiento de 3 lugares.

Otra forma de cifrado por sustitución interesante de analizar es el cifrador Vigenère (*1523-1596*) mencionado previamente. Para mejorar la técnica monoalfabética se pueden usar diferentes sustituciones monoalfabéticas a medida que se va avanzando en el texto a cifrar. El nombre genérico para este tipo de aproximación es cifrador polialfabético. Este tipo de cifradores utiliza una serie de reglas de sustitución monoalfabéticas y una clave que determina la regla particular a usar en la transformación del símbolo presente. Por ejemplo, el cifrador Vigenère utiliza 26 cifradores Cesar con corrimientos de *0* a *25*. Cada cifrador tiene como identificador una letra clave que es la letra que reemplaza al cifrado de la letra *a*. De este modo, el cifrador Cesar con un corrimiento de *3* se denota como *d*. Se puede armar una tabla como la que se presenta en la Fig. 2.2 que representa los *26* cifradores mencionados.

El proceso de cifrado es sencillo. Dada una letra de la clave *x* y una letra del texto pleno *y*, la letra del texto cifrado correspondiente es la que resulta de la intersección de la fila *x* con la columna *y* de la tabla. La clave generalmente es una palabra que se repite a lo largo del texto como se indica en la Fig. 2.3. Para el descifrado la letra de la clave especifica la fila y la posición de la letra del texto cifrado en esa fila determina la columna. La letra del texto descifrado se encuentra en la parte superior de dicha columna.

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
b	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
c	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
d	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
e	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
f	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
g	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
h	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
i	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
j	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
k	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
l	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
m	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
n	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
o	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
p	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
r	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
s	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
t	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
u	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
v	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
w	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
x	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Figura 2.2 Tabla de Vigenère

Clave:	T	I	P	O	T	I	P	O	T	I	P	O	T	I	P	O
Texto pleno:	A	H	O	R	A	E	S	E	L	M	O	M	E	N	T	O
Texto cifrado:	T	P	D	F	T	M	H	S	E	U	D	A	X	V	I	C

Figura 2.3 Método de Vigenère.

Este ejemplo cubre el caso de una sustitución polialfabética periódica, con período igual a la longitud de la palabra clave. Existen muchas variantes de este método, algunas emplean una letra inicial como clave y a partir de ahí las letras que resultan del texto cifrado, con lo que se genera una realimentación que otorga una mayor seguridad al cifrado.

Otro criptosistema derivado de los mencionados fue propuesto por *Gilbert Vernam*, ingeniero de AT&T, en el año 1917. Fue ideado para ser usado en sistemas de transmisión telegráfica que usaban el código de Baudot, alfabeto de 32 símbolos representados por cinco dígitos binarios, habitualmente escritos sobre cintas perforadas. La idea de este sistema consiste en utilizar claves de la misma longitud que el mensaje a transmitir. Los mensajes se cifran como en el modelo Vigenère, usando cada letra de la clave para cifrar cada una de las letras del mensaje [SCH96]. Para garantizar la máxima seguridad es imprescindible que las claves se generen en forma aleatoria y se utilicen una sola vez. Esta variante de clave fue introducida sobre el cifrador Vernam por un oficial de la Armada estadounidense, *Joseph Mauborgne*. El esquema se conoce también como *one time pad* y ha sido usado por espías en todo el mundo. Se guardaban en pequeñas libretas los caracteres aleatorios para cifrar los mensajes, arrancando cada página cuando había sido usada. La dificultad práctica de este método es que tanto transmisor como receptor del mensaje deben no sólo poseer la clave si no que además son depositarios de la seguridad de la misma.

Sin duda alguna el estudio y evolución de las técnicas más avanzadas de la criptografía tuvo lugar durante la *Segunda Guerra Mundial* a partir de la creación de las *máquinas de rotores*. Se trata de aparatos electromecánicos de cifrado. Los primeros modelos aparecieron en los años *20* como una contribución a la automatización del proceso de cifrado. Los rotores son ruedas que tienen contactos eléctricos en ambos lados, uno por cada letra del alfabeto. Un cableado conecta los de una cara con los de la otra, implementando así una permutación de las letras del alfabeto. Al girar el rotor sobre su eje cambia la permutación. Una máquina contiene varios rotores sobre un mismo eje. Durante el cifrado o descifrado, un mecanismo cambia la posición relativa de los rotores para cada letra, de manera que la permutación del alfabeto que afecta a cada letra es distinta. La más famosa es de estas máquinas el la conocida como *Enigma*, diseñada en el año *1923* por *Arthur Scheribus*. Fue utilizada tanto por

el ejército alemán como por el japonés durante el transcurso de la Segunda Guerra Mundial. El criptosistema fue roto por equipos de matemáticos polacos y británicos.

Desde la aparición de la criptografía de clave pública se ha producido un aumento espectacular de publicaciones y estudios en esta disciplina. Actualmente los beneficios de la criptografía se extienden al ámbito civil y comercial, dejando de ser de uso exclusivo de las agencias de inteligencia y organismos militares, como sucedía hasta hace pocos años. Actualmente la Criptografía presenta una dificultad normalmente no encontrada en otras disciplinas: la necesidad de una interacción apropiada entre ella misma y su contrapartida, el criptoanálisis. Esto significa que es fácil proponer un sistema criptográfico, más difícil es proponer uno que no pueda romperse fácilmente.

2.3 Criptoanálisis

Se denomina *criptoanálisis* [STA99] al análisis de los sistemas criptográficos con el fin de comprometer su seguridad. El objetivo del *criptoanalista* es descubrir la clave o bien encontrar la forma de descifrar o firmar mensajes sin necesidad de conocer la clave. En general dispone de criptogramas o mensajes firmados, e incluso de pares mensaje-criptograma para realizar su trabajo. Un posible ataque podría ser tratar de probar con todas las claves posibles, pero si el tamaño de la clave es grande este ataque de *fuerza bruta* se vuelve impracticable. De ahí la importancia del espacio de la clave.

Por otra parte un atacante u adversario que pretenda comprometer alguna de las características relacionadas con la seguridad de la información puede llevar adelante su acción de manera *pasiva* o *activa*. Un *ataque pasivo* es aquel en el que el atacante sólo monitorea el canal de comunicación, resultando una amenaza fundamentalmente en el sentido de la confidencialidad. En un *ataque activo* el adversario intenta borrar, agregar o alterar parte de la transmisión en el canal resultando de este modo una amenaza para la integridad, la autenticación y la confidencialidad de los datos.

Un ataque pasivo puede ser clasificado de manera más especializada según la información de la que se dispone. El objetivo es, en cualquier caso, la recuperación del texto pleno a partir del texto cifrado o, en un caso más dramático, la deducción de la clave de cifrado.

Si se establece una convención para identificar los factores analizables en un ataque, se puede representar el proceso de cifrado por la siguiente operación:

$$C = E_K(P) = Criptograma \tag{2.1}$$

Donde E representa el algoritmo de cifrado, K la clave y P el texto pleno a cifrar. Se pueden mencionar cuatro técnicas de ataque relacionadas con un ataque de tipo pasivo [STA99]. Se supone en lo que sigue que el criptoanalista lo único que conoce es el algoritmo de cifrado en cuestión.

2.3.1 Ataque basado en texto cifrado solamente. (*Ciphertext-only attack)*

Dados $C_1 = E_K(P_1)$, $C_2 = E_K(P_2)$,... $C_i = E_K(P_{i1})$, se trata de deducir P_1, P_2, P_i,... K o algún algoritmo para inferir P_{i+1} a partir de $C_{i+1} = E_K(P_{i+1})$. Es decir que, en este tipo de ataque, el criptoanalista trata de obtener la clave de cifrado o el texto original sólo por observación del texto cifrado. Cualquier esquema de cifrado vulnerable a este tipo de ataques es considerado completamente inseguro.

2.3.2 Ataque basado en texto pleno conocido. (*Known-plaintext attack)*

Dados P_1, $C_1 = E_K(P_1)$, P_2, $C_2 = E_K(P_2)$,... Pi, $C_i = E_K(P_i)$, deducir K o algún algoritmo para inferir P_{i+1} a partir de $C_{i+1} = E_K(P_{i+1})$. En este caso, el criptoanalista conoce los bloques de texto sin cifrar y sus respectivos textos cifrados, es decir que cuenta con cierta cantidad de pares de texto pleno - texto cifrado. Se trata de un ataque un poco más difícil de montar que el anterior.

2.3.3 Ataque basado en texto pleno elegido. (*Chosen-plaintext attack)*

Dados P_1, $C_1 = E_K(P_1)$, P_2, $C_2 = E_K(P_2)$,... P_i, $C_i = E_K(P_i)$, donde los P_i son seleccionables por el criptoanalista, deducir K o algún algoritmo para inferir P_{i+1} a partir de

$C_{i+1} = E_K(P_{i+1})$. En este caso, el atacante elige texto pleno conocido y de alguna manera obtiene el correspondiente texto cifrado. Luego utiliza cualquier información deducida para recobrar texto pleno correspondiente al texto cifrado no visto previamente.

2.3.4 Ataque basado en texto pleno elegido adaptativamente. (*Adaptive-chosen- plaintext attack)*

Es una variante del anterior pero no sólo el criptoanalista puede elegir el texto pleno que está cifrado sino que también puede modificar su elección basándose en el resultado de cifrados anteriores conocidos.

La mayoría de estos ataques pueden aplicarse además a esquemas de firma digital y códigos para autenticación de mensajes. También los protocolos que manejan estos algoritmos son susceptibles de ser atacados [SCH96].

La seguridad de un sistema criptográfico puede ser evaluada según diversos modelos. En un sistema *incondicionalmente seguro* la probabilidad de hallar el texto pleno, luego de haber observado el texto cifrado, se mantiene igual que la probabilidad que a priori existía. A estos sistemas en criptografía se los suele denominar *de secreto* perfecto, en inglés *perfect secrecy*. Un sistema de clave simétrica sería incondicionalmente seguro si la clave fuese al menos tan larga como el mensaje [MEN97].

Una evaluación más práctica se refiere a la seguridad computacional de un sistema. Cualquier técnica a evaluar se dice que es *computacionalmente segura* si el nivel de esfuerzo computacional necesario para tener éxito luego de haber aplicado alguna técnica de ataque conocida, excede la cantidad de recursos computacionales que se supone pueda poseer un potencial adversario. También se dice que un sistema es *computacionalmente seguro* si el costo de cualquier ataque conocido sobre el mismo es mayor que el valor de la información cifrada ó el tiempo que requiere el ataque excede el tiempo de vida útil de la información.

2.4 Codificación y Criptografía

Un sistema de cifrado opera sobre un flujo o *stream* de bits que puede representar ya sea texto, imagen, voz o cualquier otra información que se presente en forma binaria. Esta operación se realiza como parte del procesamiento sobre una señal en un sistema de comunicaciones digitales. La Fig. 2.4 muestra el flujo de señal a través de un sistema de comunicaciones típico.

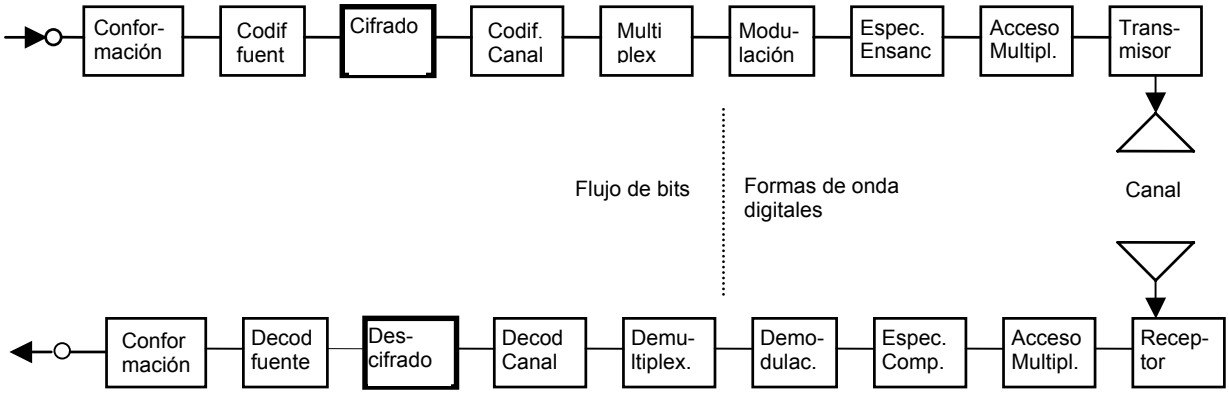

Figura 2.4 Diagrama en bloques de un esquema típico de comunicaciones

Los bloques superiores de la figura representan las diversas transformaciones que sufre una señal desde su origen o fuente hasta que se coloca en el canal. La *conformación* de los pulsos transforma la información de fuente en símbolos digitales, de tal manera que la información sea compatible con el procesamiento posterior que se hará sobre la señal. La *codificación de fuente* se refiere a la aplicación de alguna técnica para remoción de información redundante. El *cifrado* de los datos evita que usuarios no autorizados comprendan el mensaje y/o que inyecten falsos mensajes en el sistema. La *codificación de canal* se refiere a la adición de información redundante para el control de errores de transmisión [HON97]. La *modulación* se refiere al proceso en que cada símbolo se convierte en una forma de onda apropiada a la transmisión sobre el canal en cuestión [HEE82]. La aplicación de técnicas de *espectro esparcido* -spread spectrum, en inglés- puede reducir vulnerabilidades frente a interferencias. Las técnicas de *multiplexado* y *acceso múltiple* sirven para combinar señales con diferentes características o provenientes de diversas fuentes, de tal manera que puedan compartir los recursos de comunicación disponibles. De todos los pasos de procesamiento de la señal

solamente la conformación, modulación y demodulación son esenciales para los sistemas de comunicaciones digitales. Los otros pasos del proceso son opciones de diseño para cubrir necesidades de sistemas específicos [MCE02].

Como se ha comentado previamente, la evolución impactante de los sistemas de comunicaciones en las últimas décadas se mostró acompañada por una creciente exigencia de seguridad sobre los datos y sobre los recursos. En general, las propiedades exigidas sobre un esquema de cifrado son bastante diferentes de aquellas relacionadas con la codificación de canal. Por ejemplo en un esquema de cifrado, el texto pleno nunca aparece directamente en el texto cifrado. Por su parte la codificación de canal suele ser sistemática, en el sentido que se transmite no sólo el mensaje original, sino también los bits que comprenden su redundancia [PEE85].

De todas maneras, muchas de las técnicas de codificación utilizadas actualmente muestran mejoras importantes cuando se las combina con técnicas de entrelazamiento con algún tipo de manejo aleatorio [CAS96]. En este sentido, codificación y cifrado han evolucionado en los últimos tiempos con la utilización de herramientas comunes, tales como los campos de Galois [HIL78] o el uso de generadores pseudo-aleatorios [SCH96] [MEN97]. En la actualidad, la realización combinada de ambas técnicas es motivo de estudio de algunas líneas de investigación modernas.

2.5 Técnicas Modernas de Criptografía de Clave Simétrica

Virtualmente hasta la aparición del último estándar de cifrado de datos, *AES* (*Advanced Encryption Standard*), todos los algoritmos de clave simétrica se basaban en una estructura conocida como *Cifrador Bloque Feistel* [STA99]. Un cifrador de bloque opera sobre un conjunto de entrada de *n* bits para producir un texto cifrado de la misma longitud. Un cifrador *Feistel* utiliza el concepto de cifrador producto, en el sentido que coloca uno ó más cifradores básicos en secuencia de tal manera que el resultado final es criptográficamente más fuerte que el de los cifradores componentes.

Shannon propuso desarrollar tal cifrador alternando operaciones de *confusión* y *difusión* para frustrar técnicas de criptoanálisis basadas en estadísticas. Uno de los algoritmos de clave simétrica más conocidos y que utiliza estas técnicas es el *DES* (*Data Encryption Standard*), algoritmo de cifrado de datos estándar adoptado en 1977[MEN97]. *DES* ha perdurado como estándar hasta que fue reemplazado por *AES* en el año 2001.

Son muchas las diferencias entre ambos. *DES* utiliza una clave de *56 bits* y opera sobre bloques de *64 bits*. La estructura básica, o ronda, se repite *16* veces y utiliza una clave diferente o *sub-clave* para cada ronda. Cada una de las sub-claves se deriva de un procesamiento realizado sobre la clave original. El descifrado utiliza el mismo algoritmo pero el orden de aplicación de las sub-claves es inverso [STI95].

Por otra parte, para poder aplicar este algoritmo sobre una amplia gama de aplicaciones se definieron cuatro modos de operación: *ECB*, *CBC*, *CFB* y *OFB* [SCH96]. Estos modos se explican a continuación y pueden ser aplicados a cualquier algoritmo tipo bloque de clave simétrica.

En el modo *ECB* (*Electronic Codebook*) cada bloque de texto pleno es cifrado de manera independiente usando la misma clave. Este modo es apto para aplicaciones de pocas cantidades de datos, por ejemplo para cifrar claves. La velocidad de cifrado en este modo es la misma que la del algoritmo original y permite aplicar algún tipo de procesamiento en paralelo si el diseño del algoritmo lo permite. Su característica más peculiar es que el cifrado del mismo bloque de texto pleno produce el mismo bloque de texto cifrado. Esta propiedad lo hace vulnerable a ataques de tipo diccionario.

En el modo *CBC* (*Cipher Block Chaining*) cada bloque de entrada al algoritmo de cifrado se suma módulo-2 (*EXOR*) con la salida cifrada del bloque anterior, como se representa en la Fig. 2.5. El primer bloque de mensaje se suma con una semilla o vector de inicialización que debe ser conocido por ambos extremos de la transmisión. Con este modo se evita que un mismo texto pleno produzca el mismo texto cifrado. La desventaja es que no se pueden cifrar en este modo bloques de datos en paralelo. Su aplicación más usual es en algoritmos de

autenticación.

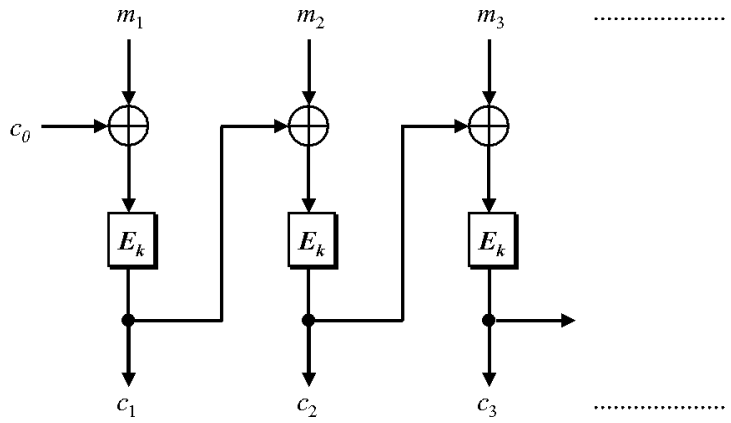

Fig. 2.5. Modo CBC

El modo *CFB* (C*ipher Feedback*) consiste en el procesamiento de *j bits* por vez, convirtiendo así un cifrador bloque en un cifrador de flujo. Este funcionamiento permite eliminar la necesidad de rellenar cualquier mensaje a un número entero de bits, múltiplo del tamaño de un bloque. También este modo es adaptable a aplicaciones en tiempo real. El texto cifrado resulta de sumar módulo-*2 j bits* de texto pleno con *j* bits que provienen, la primera vez de un bloque cifrado a través de un vector de inicialización, y el resto de las veces del bloque cifrado previo, como se muestra en la Fig. 2.6..

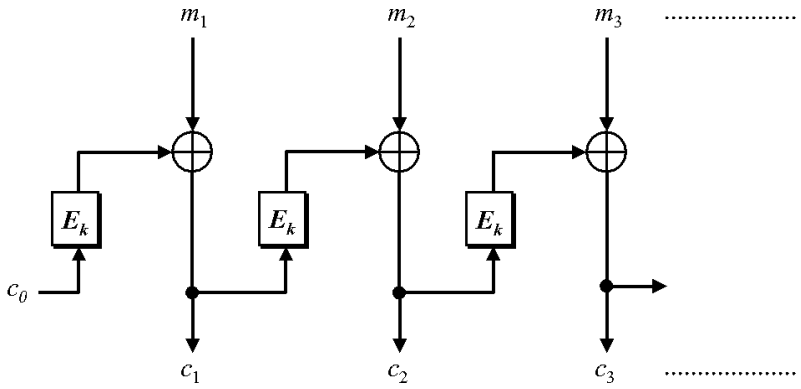

Fig. 2.6. Modo CFB

En el modo *OFB* (*Output Feedback*) el proceso es similar al modo *CFB*, pero lo que se realimenta a la etapa siguiente es la salida cifrada de la etapa previa. La ventaja de este modo

es que un error en cualquier bit transmitido sólo afecta al bloque al que dicho bit pertenece. La desventaja es que es más vulnerable a modificaciones de los mensajes en tránsito.

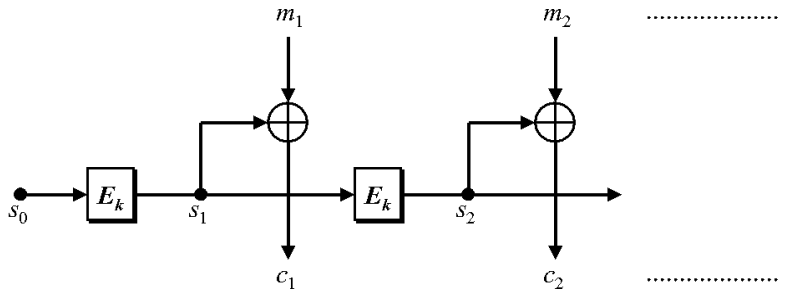

Fig. 2.7. Modo OFB

El nuevo estándar de cifradores bloque, *AES*, se basa en un algoritmo desarrollado por *Joan Daemen* y *Vincent Rijmen*, conocido como *Rijndael*. Sus autores tuvieron en cuenta varios criterios a la hora de diseñarlo: resistencia frente a ataques conocidos, velocidad, simplicidad y posibilidad de desarrollarlo sobre un código compacto para una gran variedad de plataformas [DAE99]. Este algoritmo se suele implementar para funcionamiento en modo ECB o CBC. También es muy conocida una variante, llamada modo *Cuenta* o *CTR* que en realidad cifra el valor de un contador y el resultado lo suma módulo-2 con el bloque de datos. El contador se incrementa con cada bloque, generando así un cifrador de flujo. La desventaja es que requiere el uso de alguna función hash para garantizar la integridad del mensaje [WEA02].

La transformación por ronda del algoritmo Rijndael no se ajusta a una estructura *Feistel*, sino que se compone de tres transformaciones uniformes distintas e invertibles, llamadas *capas*, *niveles* o *layers* [DAE99]. Son uniformes en el sentido que cada bit del bloque a cifrar se trata de manera similar al resto. Los autores basaron la elección específica para cada *capa* en un método de diseño que provee resistencia frente al criptoanálisis lineal o diferencial. Una capa de mezcla lineal garantiza gran difusión sobre múltiples rondas. Otra capa, no lineal, consiste en la aplicación de cajas *S* con características óptimas en cuanto a propiedades de alinealidad.

Antes de la primera ronda, se procede a sumar en módulo-*2* la clave inicial con el bloque a encriptar. Esta operación inicial es característica de muchos cifradores bloque. IDEA, SAFER y Blowfish la incluyen [SCH96].

Para que la estructura del cifrador y su inversa sea lo más parecida posible, la capa de mezclado lineal en la última ronda es diferente de las capas de mezclado de las demás.

Rijndael se diseñó como un cifrador de bloque iterativo con longitudes de bloque variable, al igual que la longitud de la clave. Ambas longitudes pueden independientemente fijarse en *128*, *192* o *256 bits*. Una ronda del algoritmo incluye operaciones en el campo $GF(2^8)$, rotaciones cíclicas sobre el bloque y operaciones de suma módulo-*2*. Por su parte la clave es expandida desde su longitud original para generar tantas sub-claves como rondas de iteración existan. Las operaciones que se utilizan en la expansión de la clave son similares a las que usa el propio algoritmo.

2.6 Aspectos de Implementación Relativos al AES

Uno de los aspectos más importantes que tuvieron en cuenta los autores del Algoritmo *Rijndael* en el diseño se refería a la posibilidad de la implementación del mismo en *hardware*. Existe una gran variedad de razones para preferir este tipo de implementación. La primera de ellas se refiere a la velocidad y está relacionada con la complejidad de las operaciones involucradas [DAE99]. Colocar el sistema criptográfico en un chip diferente al del procesador principal mejora la velocidad de cualquier sistema aumentando la eficiencia del mismo. Otra razón importante de destacar se relaciona con la seguridad física. Se pueden tomar recaudos para evitar el manoseo o *tampering* de un algoritmo más fácilmente si el mismo se encuentra desarrollado en un chip. La perspectiva de un medio de almacenamiento ligado a una implementación software siempre se presenta como menos segura [ELB01].

Los autores del algoritmo *Rijndael* pensaron en operaciones que no ofreciesen dificultades de implementación sobre procesadores de *8* o *32* bits y en *hardware* dedicado. También incluyeron algún grado de paralelismo entre las operaciones relativas a una ronda. En

su propuesta original mencionan la principal relación de compromiso existente en el caso de una implementación en *hardware*. Se trata de la relación entre área y velocidad.

Otra cuestión relacionada con aspectos de su implementación es que el cifrador y su inverso usan distintas transformaciones, es decir que un circuito que implemente *Rijndael* no soportará automáticamente la transformación inversa. De todas maneras, un detalle a rescatar es el hecho que existirán partes del circuito de cifrado que podrán ser usadas en el descifrado. Esto se relaciona con la característica de re-uso de los componentes de un circuito.

El surgimiento de las nuevas tecnologías de *FPGA* (*Field Programmable Gate Arrays*) significó un importante avance en el diseño de circuitos para criptografía ya que ofrecen una gran flexibilidad debido a su capacidad de modificación del hardware. Deja de pensarse en el diseño de módulos autocontenidos o dedicados para pasar a imaginar circuitos reconfigurables [BRO96], capaces de adaptarse a las más diversas aplicaciones.

Este nuevo sentido de diseño en *hardware* obliga al desarrollador a pensar primero en la organización básica de cualquier circuito de aplicación práctica. De esta manera sería posible adaptar el mismo circuito rápidamente sobre una gama completa de utilidades.

En este sentido es conveniente destacar las unidades básicas que han de conformar un cifrador de bloque simétrico [GAJ01], comunes por tanto a la implementación del AES:

- Una *Unidad de Encriptado/Desencriptado* que recibe los bloques de datos y los procesa. Deberá poseer la mayor cantidad de componentes posibles de re-usar en ambos sentidos de procesamiento.
- Una *Unidad de Expansión de la Clave* que origina las diversas sub-claves necesarias a partir de la clave original.
- *Memoria* para almacenar las sub-claves generadas por la *Unidad de Expansión de la Clave* o ingresadas externamente a través de alguna *Interfaz*.
- Una *Interfaz de Entrada/Salida* para ingresar los bloques de datos o las claves al circuito de encriptado.

- Una *Unidad de Control* que genere las señales de control apropiadas para manejar los diversos bloques componentes del circuito.

Esta manera de plantear el diseño, unida a la posibilidad de reconfiguración del circuito, abre un panorama más que alentador a la implementación de diversas técnicas criptográficas modernas.

Capítulo 3

Principios Algebraicos Básicos

En este capítulo se resumen los aspectos más sobresalientes de la Teoría de Campos Finitos. Se presentan primero conceptos fundamentales del álgebra de Grupos, Anillos y Campos. A continuación se examina la estructura de Campos Finitos. Por último se analizan los principios básicos de Campos Compuestos por hallarse este tema íntimamente relacionado con la implementación del Algoritmo Rijndael.

3.1 Grupos

Definición 3.1.1: Un conjunto de *G* elementos sobre los cuales se ha definido una operación binaria, se denomina *grupo* cuando satisface condiciones de asociatividad, contiene un elemento identidad único y, para cada elemento de *G*, existe un único elemento inverso [ALL83].

Si la operación binaria satisface condiciones de conmutatividad se dice que *G* es un *grupo conmutativo o abeliano.* Entre los grupos conmutativos más conocidos se encuentran el de los números enteros y el de los números racionales. El conjunto de enteros es un grupo conmutativo bajo la operación de suma (*+*), siendo el *0* el elemento identidad y *–i* la inversa de un número entero *i*. El conjunto de números racionales excluyendo el cero es un grupo conmutativo bajo la operación de multiplicación (*x*), siendo *1* el elemento neutro y el número *b/a* el inverso multiplicativo de *a/b.*

El conjunto *G = {0, 1}* bajo la operación suma *módulo-2* (⊕) es un grupo conmutativo. La operación es equivalente en el Algebra Booleana a una *EXOR*. En dicho grupo, *0* es el elemento identidad y la inversa de cada elemento es el mismo elemento.

Definición 3.1.2: El número de elementos de un grupo *G* se denomina *orden del grupo* y se denota *ordG*. De esta manera, según la cantidad de elementos, se puede hablar de dos grandes clases de grupos: *infinitos* y *finitos*. Para los fines de esta tesis, se consideran sólo los grupos finitos.

Definición 3.1.3: El orden de un elemento *g* en un grupo finito *G*, es el número más pequeño $s > 0$, tal que g^s tenga como resultado el elemento identidad y se denota $ordg$. Se deduce que existirán *s* potencias distintas de g [ALL83]. Se deduce también que el orden máximo de un elemento no puede superar al orden del grupo al que pertenece, pues entonces el conjunto de potencias de dicho elemento sería mayor que el número de elementos del grupo. Dado que todos los elementos de *G* deben ser distintos entre sí, esta última situación no sería posible.

Definición 3.1.4: Siendo *m* un entero positivo, es posible definir un grupo finito conmutativo de orden *m*: *G = { 0, 1, 2, ... ,m-1}*, bajo la operación *suma módulo-m* [HIL78], siendo *0* el elemento identidad y *(m-i)* el elemento inverso de *i*. Se lo conoce como *grupo aditivo*.

Definición 3.1.5: El grupo de enteros *G = {1, 2, 3,...., p-1}*, con *p* primo, bajo la operación *multiplicación módulo-p* es un grupo conmutativo cuyo elemento identidad es *1* [ALL83]. Siendo *i* un elemento de *G* tal que $i < p$, *i* y *p* deben ser primos relativos por lo que se verifica que existen dos enteros *a* y *b* que cumplen el *Algoritmo de Euclides*. Por el *Teorema de Euclides a* y *p* son primos relativos y se relacionan por medio de la Ec.(3.1) [HIL78].

$$a.i + b.p = 1 \tag{3.1}$$

Se verifica que, si el producto $(a.i)$ se divide por *p*, el resto de la división da *1*. Si $0 < a < p$, a pertenece a *G* y resulta ser inverso multiplicativo de *i* [LIN83].

Sin embargo, si a no pertenece a *G*, al dividir a por *p*, el resto $r < p$ es distinto de *p* y de *0* pues a y *p* son primos relativos. De este modo *r* pertenece a *G* y es tal que $1 < r < (p-1)$ y

resulta ser el inverso multiplicativo de *i*. Las Ec.(3.2) a Ec.(3.5) ilustran lo explicado.

$$a = qp + r \tag{3.2}$$

$$a.i = qp.i + r.i \tag{3.3}$$

$$qp.i + r.i = -b.p + 1 \tag{3.4}$$

$$r.i = (-b + q.i)\,p + 1 \tag{3.5}$$

El grupo *G = {1, 2, 3,...., p-1}* bajo la *multiplicación módulo-p* resulta ser un *grupo multiplicativo*. Si *p* no es un número primo, *G* no es grupo bajo la *multiplicación módulo-p* [LIN83].

Definición 3.1.6: Se dice que *H* es subgrupo de *G*, si *H* es subconjunto de *G* cerrado bajo la operación de *G* y satisface todas las condiciones de un grupo. Si a es un elemento de un grupo, el conjunto de todas las potencias enteras de a es un subgrupo [ALL83]. Por ejemplo el conjunto de todos los números enteros es un subgrupo del grupo de números racionales bajo la operación de suma de números reales [HIL78].

Definición 3.1.7: Se dice que un grupo es *cíclico* si para algún elemento a que pertenece al grupo, todo elemento que pertenezca al grupo es de la forma a^m, con *m* perteneciente a los enteros [ALL83], [HIL78]. El elemento a se llama *generador* del grupo y es tal que $orda = ordG$. Todo grupo cíclico es *abeliano*. Un elemento a^t de un grupo cíclico finito de orden *n* es generador del grupo si se verifica *(t, n) = 1*, o sea *t* y *n* son primos relativos [ALL83].

Definición 3.1.8: Se denomina S_n al conjunto de las *n!* permutaciones de *n* símbolos. Este conjunto resulta ser un grupo respecto de la operación permutación (°). Como dicha operación no es conmutativa, un grupo de permutaciones es no abeliano [HIL78].

Definición 3.1.9: El teorema de *Lagrange* establece que el orden de cada subgrupo de un grupo finito *G* es divisor del orden de *G.* Por tanto si *G* es finito, el orden de cualquier elemento $a \in G$ (o

sea el orden del subgrupo cíclico generado por a) es divisor de *n*. También todo grupo de orden primo es cíclico [ALL83].

3.2 Campos

Definición 3.2.1: Sea un conjunto de elementos *F* sobre el cual se definen dos operaciones binarias denominadas adición (*+*) y multiplicación (.), *F* es un *campo* si se verifica [LIN83]:

- *F* es *grupo conmutativo* bajo la (*+*) con elemento identidad *0*.
- El conjunto de elementos distintos de *0* de *F* es un *grupo multiplicativo* bajo la (.) con elemento identidad *1*.
- La operación multiplicación (.) es distributiva respecto de la suma (*+*).

El número de elementos de un campo se denomina *orden*. Los campos finitos se conocen como *campos de Galois* y se denotan como *GF* (*Galois Field*) en honor a su descubridor.

Definición 3.2.2: Para cada elemento a perteneciente al campo *F* existe un inverso aditivo $-a$ y, si a es distinto de *0*, también existe un inverso multiplicativo a^{-1}, [MCE02].

El conjunto *{0, 1}* es *grupo conmutativo* bajo la *adición módulo-2* y *{1}* es *grupo multiplicativo* bajo la *multiplicación módulo-2*. Se puede verificar en este caso el cumplimiento de la propiedad distributiva, con lo cual *{0, 1}* es un campo de dos elementos bajo las operaciones mencionadas. Este campo se denomina *campo binario*, se denota *GF(2)* y juega un rol muy importante en la *Teoría de Códigos en Comunicaciones Digitales* [LIN83], [MCE02],[HEEM82], [HOE82], [IMM91], [HIL78].

El conjunto *{0, 1, 2,...., p-1}* es un campo de orden *p* bajo la *adición* y la *multiplicación módulo-p*, y se lo denota *GF(p)*. Para cualquier número primo *p* existe un campo finito de orden *p* [LIN83]. De hecho, para cualquier entero positivo *m*, es posible extender el campo *GF(p)* a un

campo de p^m elementos que se denomina *campo extendido de GF(p)* y se denota *GF(p^m)*. Se ha demostrado que el orden de cualquier campo finito es una potencia de un número primo, p^m [ALL83].

Definición 3.2.3: Dado un campo finito de *q* elementos *GF(q),* el menor entero positivo λ, tal que $\sum_{i=1}^{\lambda} 1 = 0$ se conoce como *característica* de *GF(q)* y siempre es un número primo [LIN83]. De esta definición se sigue que, dados dos números enteros positivos *k* y *m*, menores que λ, se verifica la relación que se presenta en Ec.(3.6) [LIN83].

$$\sum_{i=1}^{k} 1 \neq \sum_{i=1}^{m} 1 \tag{3.6}$$

La sucesión de diferentes sumatorias hasta llegar a λ, genera λ elementos diferentes de *GF(q)* y el conjunto de las mismas es en sí mismo un campo de λ elementos *GF(λ)*, *subcampo* de *GF(q)*. Se puede probar que si $q \neq \lambda$, entonces *q* es potencia de λ.

Se resaltan de este modo dos elementos importantes relativos a este tema:

- La *característica* de cualquier campo finito es un número primo.
- Todo campo finito de *característica p* contiene el campo de enteros *modulo-p* como *subcampo*.

Definición 3.2.4: Dado un elemento a distinto de *0*, perteneciente a *GF(q)*, si se toma la secuencia de potencias de a, se verifica que existen dos números positivos *k* y *m*, con $m > k$, tales que $a^k = a^m$. Si a^{-1} es el inverso multiplicativo de a, a^{-k} es el inverso multiplicativo de a^k, de tal modo que es factible verificar la Ec.(3.7) [LIN83]:

$$1 = a^{m-k} \tag{3.7}$$

Esta consideración implica que existe un entero positivo *n*, tal que $a^n = 1$ y todas las potencias de a, hasta la potencia *n-ésima* son diferentes entre sí. El *orden* de un elemento a es

el entero *n* y las potencias mencionadas forman un *grupo cíclico* bajo la operación de multiplicación de *GF(q)*.

Definición 3.3.5: Si a es cualquier elemento distinto de cero de *GF(q)*, entonces $a^{q-1} = 1$ y si el orden de a es *n*, *n* divide a *q-1*. En un campo finito *GF(q)*, se dice que un elemento a distinto de cero, es *primitivo* si el orden de a es *q-1* [LIN83].

3.3 Aritmética de Campo Binario

Los códigos y técnicas más utilizados en transmisión y almacenamiento cifrado de datos pertenecen a *GF(2)* o sus extensiones *GF(2^m)* [CAS96]. Se debe, entonces, profundizar el conocimiento y la interpretación de los datos como pertenecientes a alguno de los campos mencionados y, por consiguiente, de las operaciones a las que se ven sujetos [BLA94]. De este modo será posible comprender mejor diferentes posibilidades de funcionamiento y de implementación de los diversos diseños de circuitos utilizados para aplicar el procesamiento requerido.

Definición 3.3.1: Un polinomio *f(x)* sobre un campo *F* se puede expresar según la Ec.(3.8):

$$f(X) = f_0 + f_1 X + f_2 X^2 + ... + f_n X^n \tag{3.8}$$

donde cada uno de los coeficientes f_i es un elemento de *F*. *X* se denomina indeterminado y *n* es el grado del polinomio. El grado es el mayor número entero tal que $f_n \neq 0$. Los polinomios que verifican $f_n = 1$ se denominan *polinomios mónicos* [LIN83].

Se considera trabajar con polinomios cuyos coeficientes pertenecen a *GF(2)*, a los que se denomina *polinomios sobre GF(2)*. Se trata de aquellos polinomios para los que se verifica $f_i = 0$ ó 1, en el caso $0 \leq i \leq n$. Por ejemplo, existen dos polinomios de grado 1, X y $1 + X$, cuatro

polinomios de grado 2, X^2, $1+X^2$, $X+X^2$ y $1+X+X^2$. En general existen 2^n polinomios sobre *GF(2)* de grado *n*. Sobre cualquiera de ellos se puede operar con suma, resta, multiplicación y división. En la multiplicación y adición de coeficientes se considera la *aritmética módulo-2* (*EXOR*) y se puede demostrar que la suma es conmutativa, asociativa y distributiva con respecto al producto. Por otra parte, al dividir dos polinomios entre sí se obtiene un polinomio cociente y otro polinomio resto. Si este último es cero se dice que los dos polinomios son divisibles [ALL83].

Por otra parte, si a es raíz de un polinomio, dicho polinomio es divisible por $X+a$. Para polinomios sobre *GF(2)* que tienen un número par de términos, todos son divisibles por $X+1$ [LIN83].

Definición 3.3.2: Un polinomio de grado *m* se dice que es *irreducible* sobre *GF(2)* si no es divisible por ningún polinomio sobre *GF(2)* de grado menor que *m* pero mayor que *0* [HOE82]. Es decir que el concepto de irreducible para polinomios es análogo al concepto de primalidad para enteros [ALL83].

Un teorema importante referido a los polinomios irreducibles sobre *GF(2)* establece que un polinomio irreducible sobre *GF(2)* de grado *m* divide a $X^{2^m-1}+1$ [LIN83].

Dos importantes consecuencias de la definición de polinomios *irreducibles* son:

- Para cada polinomio irreducible *f(x)* sobre el campo *F*, existe un polinomio mónico irreducible *g(x)*, tal que $f(x)=a.g(x)$, siendo a un escalar, de tal modo que se cumple: $g(x)=a^{-1}f(x)$.
- Si *f(x)* es irreducible sobre el campo *F*, no necesariamente es irreducible sobre otro campo *G* [LIN83].

Definición 3.3.3: Si *f(x)* es un polinomio irreducible de grado *m* sobre *GF(p)* y *β* una raíz de dicho polinomio, claramente *β* no pertenece a *GF(p)*. El conjunto de polinomios grado < *m* en *GF(p)[β]*, junto con la multiplicación y adición módulo *f(β)*, forma un campo finito único respecto al isomorfismo. Se trata de una *extensión* de *GF(p)* con p^m elementos, *GF*(p^m). Esencialmente esta definición establece que existe un único campo finito con p^m elementos, para cualquier primo *p* y un entero *m*.

Definición 3.3.4: Se dice que un elemento a perteneciente a un campo *F* es un *elemento primitivo* de *F* si cualquier elemento distinto de cero de *F* se puede expresar como una potencia de a. Un polinomio irreducible sobre el subcampo primo de *F* que tenga un elemento primitivo de *F* como raíz se denomina *polinomio primitivo*. Un polinomio irreducible de grado *m* se verifica que es *primitivo* si el menor entero positivo *n* para el cual dicho polinomio divide a $X^n + 1$ es $n = 2^m - 1$. Existen tablas de polinomios primitivos donde se señalan los irreducibles [LIN83].

Un polinomio con coeficientes en *GF(2)* puede tener raíces que pertenezcan a un campo extensión de *GF(2)*. Por ejemplo el polinomio $X^4 + X^3 + 1$ es irreducible sobre *GF(2)* por lo que no posee raíces en *GF(2)*, aunque sí las posee en *GF*(2^4), siendo estas α^7, α^{11}, α^{13} y α^{14}.

Se puede demostrar que, siendo *f(X)* un polinomio con coeficientes en *GF(2)* y β un elemento del campo extendido de *GF(2)*, si β es raíz de *f(X)*, entonces para cualquier $l \geq 0$, β^{2^l} es también raíz de *f(X* [LIN83]*)*. Como β es un elemento de *GF*(2^m) se verifica que $\beta^{2^m - 1} = 1$. Esta relación también se puede expresar como $\beta^{2^m - 1} + 1 = 0$. De este modo, se cumple que β es raíz del polinomio $X^{2^m - 1} + 1$ cuyo grado es 2^m*-1*. Es decir que los 2^m*-1* elementos de *GF*(2^m) son las raíces de dicho polinomio. Por otro lado, todos los elementos de *GF*(2^m), incluyendo el *0*, son raíces de $X^{2^m} + X$.

Debido a lo anteriormente expresado, β puede ser raíz de un polinomio en *GF(2)* con grado menor a 2^m. Sea $\varphi(X)$ el polinomio de menor grado sobre *GF(2)* para el que se verifica $\varphi(\beta) = 0$. Este polinomio único se conoce como *polinomio mínimo* de β. Se puede demostrar que dicho polinomio es irreducible [LIN83]. También se puede demostrar que, si β es raíz de *f(X)*, entonces *f(X)* es divisible por $\varphi(X)$. El polinomio mínimo también divide a $X^{2^m} + X$.

Siendo β un elemento en $GF(2^m)$ y *e* el menor entero positivo tal que $\beta^{2^e} = \beta$, se verifica la Ec.(3.9):

$$f(X) = \prod_{i=0}^{e-1} (X + \beta^{2^i}) \tag{3.9}$$

De este modo, *f(X)* es un polinomio irreducible sobre *GF(2)*, que resulta ser el polinomio mínimo $\varphi(X)$. En particular, el grado de un polinomio mínimo de cualquier elemento en $GF(2^m)$ divide a *m* [LIN83].

3.4 Construcción de GF(2^m)

Un método para construir un campo $GF(2^m)$ a partir de *GF(2)*, consiste en comenzar con dos elementos *0* y *1* de *GF(2)* y un nuevo símbolo α para armar la secuencias de potencias de α a partir de la definición de la multiplicación (.). La Ec.(3.10) representa los elementos de dicho campo.

$$
\begin{aligned}
&0.0 = 0 \\
&0.1 = 1.0 = 0 \\
&1.1 = 1 \\
&0.\alpha = \alpha.0 = 0 \\
&1.\alpha = \alpha.1 = \alpha \\
&\alpha^2 = \alpha.\alpha \\
&\alpha^3 = \alpha.\alpha.\alpha \\
&. \\
&. \\
&. \\
&\alpha^j = \alpha.\alpha.....\alpha(jveces) \\
&. \\
&.
\end{aligned}
\tag{3.10}
$$

Se define un conjunto de elementos *F* que consiste en la serie de potencias de α anteriormente presentada y la restricción que *F* contenga sólo 2^m elementos. Si *p(X)* es un polinomio primitivo de grado *m* sobre *GF(2)* y se supone que verifica $p(\alpha) = 0$, dado que *p(X)* divide a $X^{2^m-1} + 1$, al reemplazar *X* por α resulta:

$$X^{2^m-1} + 1 = p(X).q(X) \tag{3.11}$$

$$\alpha^{2^m-1} + 1 = p(\alpha).q(\alpha) \tag{3.12}$$

$$\alpha^{2^m-1} + 1 = 0.q(\alpha) \tag{3.13}$$

$$\alpha^{2^m-1} = 1 \tag{3.14}$$

Es decir que, bajo la condición $p(\alpha) = 0$, el conjunto *F* se convierte en un campo finito de elementos $F^* = \{ 0, 1, \alpha, \alpha^2, ..., \alpha^{2^m-2} \}$ que a su vez es un grupo conmutativo de orden 2^m-1 bajo la multiplicación si se excluye el elemento *0*.

Se pretende definir una operación adición sobre F^* para que también se verifique la propiedad de grupo conmutativo. En el caso en que $0 \le i \le 2^m - 1$, al dividir el polinomio X^i por *p(X)* se obtiene:

$$X^i = q_i(X)\,p(X) + a_i(X) \tag{3.15}$$

$$a_i(X) = a_{i0} + a_{i1}X + a_{i2}X^2 + ... + a_{im-1}X^{m-1} \tag{3.16}$$

X y *p(X)* son primos relativos. X^i no es divisible por *p(X)* y se verifica que, para cualquier $i \geq 0$, es $a_i(X) \neq 0$. También se puede demostrar que, para $i \leq 0$, $j < 2^m - 1$, $i \neq j$, $a_i(X) \neq a_j(X)$. De esta manera se obtienen $2^m - 1$ polinomios $a_i(X)$ distintos de grado *m-1* o menor. Si se reemplaza *X* por α se verifica la Ec.(3.17):

$$\alpha^i = a_i(\alpha) = a_{i0} + a_{i1}\alpha + a_{i2}\alpha^2 + ... + a_{im-1}\alpha^{m-1} \tag{3.17}$$

Se ve que $2^m - 1$ elementos distintos de cero, α^0, α^1,...., α^{2^m-2} en F^*, se representan por $2^m - 1$ *polinomios distintos de cero* de α sobre *GF(2)* con grado *m-1* o menor.

Definida la (*+*), se puede comprobar que F^* es un grupo conmutativo, en el que *0* es la identidad y cada elemento es inverso de sí mismo. También los elementos no nulos de F^* forman un grupo conmutativo respecto de la multiplicación (.). Si se usa la representación polinomial para los elementos de F^* se verifica la distributividad de la multiplicación sobre la suma, con lo cual F^* es un *campo de Galois* de 2^m elementos, $GF(2^m)$, uno de cuyos subcampos es *GF(2)* y la característica es *2*. La representación por potencias es útil para las operaciones que impliquen una multiplicación, mientras que la representación de polinomios lo es para aquellas que se basen en sumas.

Ejemplo. Siendo *m* = 4 y el polinomio *p(X)* primitivo sobre *GF(2)*, $p(x) = 1 + X + X^4$, si se verifica α como raíz, resulta $p(\alpha) = 1 + \alpha + \alpha^4 = 0$ ó $1 + \alpha = \alpha^4$. Con estos supuestos podemos construir $GF(2^4)$:

Elementos de $GF(2^4)$ generados por $p(x) = 1 + X + X^4$

Representación en potencias	*Representación polinomial*	*Representación por 4-úpla*
0	0	(0 0 0 0)
1	1	(1 0 0 0)
α	α	(0 1 0 0)
α^2	α^2	(0 0 1 0)
α^3	α^3	(0 0 0 1)
α^4	$1 + \alpha$	(1 1 0 0)
α^5	$\alpha + \alpha^2$	(0 1 1 0)
α^6	$\alpha^2 + \alpha^3$	(0 0 1 1)
α^7	$1 + \alpha + \alpha^3$	(1 1 0 1)
α^8	$1 + \alpha^2$	(1 0 1 0)
α^9	$\alpha + \alpha^3$	(0 1 0 1)
α^{10}	$1 + \alpha + \alpha^2$	(1 1 1 0)
α^{11}	$\alpha + \alpha^2 + \alpha^3$	(0 1 1 1)
α^{12}	$1 + \alpha + \alpha^2 + \alpha^3$	(1 1 1 1)
α^{13}	$1 + \alpha^2 + \alpha^3$	(1 0 1 1)
α^{14}	$1 + \alpha^3$	(1 0 0 1)

Como ejemplo: $\alpha^5 . \alpha^7 = \alpha^{12}$; $\alpha^{12} . \alpha^7 = \alpha^{19} = \alpha^4$; $\alpha^4 / \alpha^{12} = \alpha^4 . \alpha^3 = \alpha^7$; $\alpha^5 + \alpha^7 = \alpha + \alpha^2 + 1 + \alpha + \alpha^3 = 1 + \alpha^2 + \alpha^3 = \alpha^{13}$.

3.5 Campo GF(2^8)

3.5.1 Operaciones con palabras de 8 bits

Varias operaciones presentes en el algoritmo *Rijndael* son de la forma orientada al *byte*. De este modo, los conjuntos de 8 bits pueden ser interpretados como elementos de *GF(2^8)*: polinomios cuyo grado máximo es *7* y cuyos coeficientes pertenecen a *GF(2)*. Otras operaciones del algoritmo

involucran palabras de *4 bytes* que se corresponden con polinomios de grado menor a *4* y coeficientes en *GF(2^8)*.

La diversas representaciones del campo *GF(2^8)* son isomórficas, pero a pesar de esta equivalencia, la complejidad de la implementación del algoritmo depende fuertemente de la representación elegida.

La Ec.(3.18) es la representación de un byte *b*, consistente de la cadena de bits $b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0$, como un polinomio con coeficientes en *{0, 1}*.

$$b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0 \qquad (3.18)$$

La suma de elementos se corresponde con la suma de elementos en *GF(2)*. Su resultado es un polinomio cuyos coeficientes resultan de la *suma módulo-2* de los coeficientes respectivos. Se cumplen así todas las condiciones de un *grupo Abeliano*.

Por ejemplo, el byte cuya representación binaria es (*01010111*) y cuya representación hexadecimal se corresponde con (*0x57*) tiene la representación polinomial dada por $x^6 + x^4 + x^2 + x + 1$. Con esta representación, la suma de los bytes (*0x57*) y (*0x58*) es el polinomio: $x^3 + x^2 + x + 1$. De este modo, se ve que la suma a nivel de bytes se corresponde claramente con una operación *EXOR*. En el Algoritmo Rijndael esta operación se realiza entre los bits de una clave o sub-clave y los correspondientes bits de una matriz de estado de resultados intermedios.

La multiplicación en *GF(2^8)* para la representación polinomial, se corresponde con la *multiplicación de polinomios módulo un polinomio binario irreducible de grado 8*. En *Rijndael* este polinomio se denomina *m(x)* y se presenta en la Ec.(3.19):

$$m(x) = x^8 + x^4 + x^3 + x + 1 \qquad (3.19)$$

El polinomio *m(x)* suele expresarse en representación hexadecimal como *0x11B*. La multiplicación módulo *m(x)* dará siempre como resultado un polinomio de grado menor que *8* de tal manera que se pueda representar como un *byte*. Además cumple con la propiedad asociativa y posee como elemento neutro (*0x01*). Por ejemplo, la multiplicación entre los bytes (*0x57*) y (0x58) se puede expresar por medio de las siguientes ecuaciones:

$$(0x57)\cdot(0x58) = (x^6 + x^4 + x^2 + x + 1)\cdot(x^6 + x^4 + x^3) \tag{3.20}$$

$$= (x^{12} + x^9 + x^3)\Big| \bmod(m(x)) \tag{3.21}$$

$$= (0x95) = x^7 + x^4 + x^2 + 1 \tag{3.22}$$

Se puede observar que la suma es una operación mucho más sencilla a nivel de bytes que la multiplicación. Por otra parte, aplicando el *Algoritmo Extendido de Euclides* es factible encontrar para cada polinomio *b(x)* de grado menor que *8*, un polinomio *a(x)* y otro *c(x)* tal que se verifiquen las Ec.(3.23) a Ec.(3.25). De este modo se puede decir que existe el elemento inverso en la multiplicación.

$$b(x)a(x) + m(x)c(x) = 1 \tag{3.23}$$

$$a(x).b(x) \bmod m(x) = 1 \tag{3.24}$$

$$b^{-1}(x) = a(x) \bmod m(x) \tag{3.25}$$

Existen entonces, *256* valores de bytes posibles y, si sobre ellos, se definen las operaciones suma y multiplicación mencionadas, el conjunto posee la estructura del campo finito *GF(2^8)*.

Puede pensarse la operación de multiplicación en un campo finito a partir de los elementos generadores de dicho campo. Las sucesivas potencias de estos elementos permiten generar progresivamente los *255* elementos diferentes de *0* de *GF(2^8)*. Cuando el valor del exponente llega a *255* se obtiene nuevamente el elemento unitario. Esta propiedad ofrece una manera de convertir la operación de multiplicación en una operación de suma.

Así, dos elementos a y b pertenecientes al campo pueden representarse por medio del elemento generador: $a = g^{\alpha}$ y $b = g^{\beta}$. Su producto $a \cdot b$ podría expresarse como $a \cdot b = g^{\alpha+\beta}$. Un elemento generador del campo *GF(2^8)* es el elemento (*0x03*)

Si se listaran en una tabla las diferentes potencias del elemento generador para cada elemento del campo finito, se podría conocer las potencias del generador, α y β, que originan los elementos a y b. La suma de ambos exponentes permitiría conocer la potencia del generador que corresponde al resultado de la multiplicación. Una nueva búsqueda en tabla permitiría hallar el resultado.

La consecuencia práctica más importante de la posibilidad de realizar la multiplicación por medio de una búsqueda en tabla es que dicha tabla también podría ser usada para hallar el elemento inverso de cada elemento del campo. Esto se debe a que el elemento g^{x} tiene como inverso al elemento g^{255-x}.

En el Algoritmo Rijndael estas propiedades se aplican a una transformación que los autores denominaron *SubBytes()* que posee las características de una sustitución nolineal [DAE99]. En los algoritmos criptográficos simétricos tipo bloque, este tipo de sustituciones suele llamarse *caja S*.

3.5.2. Multiplicación de un byte por *x*

Si se multiplica el polinomio *x*, cuya representación hexadecimal es *0x02*, por otro polinomio *b(x)* considerando la representación polinomial de un *byte*, se obtiene:

$$b_7 x^8 + b_6 x^7 + b_5 x^6 + b_4 x^5 + b_3 x^4 + b_2 x^3 + b_1 x^2 + b_0 x \qquad (3.26)$$

El producto representado en la Ec.(3.26) luego habrá que expresarlo en módulo *m(x)*. En el caso que b_7 sea *1* el cociente es *1* y el resto o resultado de la multiplicación se obtiene restando (por *suma módulo-2* ó *EXOR*) *m(x)* al dividendo. Si b_7 es *0* no hace falta dividir. De este modo, se

deduce que la multiplicación por *x* se puede implementar a nivel de byte como un corrimiento a la izquierda seguido de una operación *EXOR* condicionada con *0x1B*.

La operación se denota como $b = xtime(a)$ [DAE99] y en hardware dedicado sólo se implementaría mediante *4 EXORs*. La aplicación repetida de la misma sirve para efectuar multiplicaciones por potencias de *x*.

Por ejemplo, si se desea resolver el producto $(0x57)\cdot(0x13)$, como se verifica que $(0x57)\cdot(0x13) = (0x57)\cdot(0x01 \oplus 0x02 \oplus 0x10)$, se pueden realizan los siguientes cálculos:

$$(0x57)\cdot(0x02) = xtime(57) = AE \tag{3.27}$$

$$b_7 = 0 \Rightarrow leftshift \rightarrow 10101110 \tag{3.28}$$

$$(0x57)\cdot(0x04) = xtime(AE) = 47 \tag{3.29}$$

$$b_7 = 1 \Rightarrow leftshift, \oplus con1B \rightarrow 01011100 \oplus 0001011 = 01000111 \tag{3.30}$$

$$(0x57)\cdot(0x08) = xtime(47) = 8E \tag{3.31}$$

$$b_7 = 0 \Rightarrow leftshift \rightarrow 10001110 \tag{3.32}$$

$$(0x57)\cdot(0x10) = xtime(8E) = 07 \tag{3.33}$$

$$b_7 = 1 \Rightarrow leftshift, \oplus con1B \rightarrow 00011100 \oplus 00011011 = 00000111 \tag{3.34}$$

De este modo, la multiplicación entre los elementos del ejemplo resulta: $57 \cdot 13 = 57 \cdot (01 \oplus 02 \oplus 10) = 57 \oplus AE \oplus 07 = FE$. Así, la operación multiplicación por potencias de *x* puede generarse fácilmente mediante la aplicación repetida de la función *xtime()* y es particularmente útil en el caso en que dicha multiplicación se realice entre elementos tipo bytes y constantes. Sumando resultados parciales se puede generar de una manera sencilla la multiplicación por cualquier constante. Esta operación es usada en Rijndael y constituye un bloque básico para la implementación del algoritmo [DAE99].

3.5.3. Operaciones con palabras de 32 bits

Se pueden también definir polinomios de grado menor a *4* y con coeficientes en *GF(2^8)* para representar vectores de 4 bytes como lo indica la Ec.(3.35).

$$a(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0 \tag{3.35}$$

Esta representación es útil para aquellas operaciones de Rijndael que involucran palabras de 32 bits. Estos polinomios no son los mismos que los usados en la definición de elementos de campo finito, ya que los coeficientes en sí mismos son elementos de campo finito.

Se define la suma como la operación *EXOR* entre los respectivos coeficientes. En el caso de la multiplicación el resultado es un polinomio de mayor grado que no se puede representar por un vector de *4* bytes, por lo que hay que reducirlo mediante la operación módulo un polinomio de grado *4*. En Rijndael se utiliza para este propósito el polinomio de la Ec.(3.36) y el mismo verifica la Ec.(3.37).

$$M(x) = x^4 + 1 \tag{3.36}$$

$$x^j \bmod x^4 + 1 = x^{j \bmod 4} \tag{3.37}$$

Por otro lado, *M(x)* no es un polinomio irreducible sobre *GF(2^8)* por lo que la multiplicación por un polinomio fijo no es necesariamente invertible, pero en Rijndael el polinomio fijo elegido tiene inversa [DAE99]. Debido a la propiedad mencionada de *M(x)*, el producto *d(x) = a(x) ⊗ b(x)* queda expresado como:

$$d(x) = (a_3 x^3 + a_2 x^2 + a_1 x + a_0) \otimes (b_3 x^3 + b_2 x^2 + b_1 x + b_0) \tag{3.38}$$

$$d(x) = d_3 x^3 + d_2 x^2 + d_1 x + d_0 \tag{3.39}$$

donde

$$d_0 = a_0.b_0 \oplus a_3.b_1 \oplus a_2.b_2 \oplus a_1.b_3 \tag{3.40}$$

$$d_1 = a_1.b_0 \oplus a_0.b_1 \oplus a_3.b_2 \oplus a_2.b_3 \tag{3.41}$$

$$d_2 = a_2.b_0 \oplus a_1.b_1 \oplus a_0.b_2 \oplus a_3.b_3 \tag{3.42}$$

$$d_3 = a_3.b_0 \oplus a_2.b_1 \oplus a_1.b_2 \oplus a_0.b_3 \tag{3.43}$$

De este modo, se puede expresar la multiplicación "⊗" por un polinomio fijo *a*(x) como un producto de matrices:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{3.44}$$

3.5.4. Multiplicación de una palabra de 32 bits por x

Un caso especial lo constituye la multiplicación $x \otimes b(x)$. El resultado es un polinomio de grado *4*, $b_3 x^4 + b_2 x^3 + b_1 x^2 + b_0 x$, que debe reducirse módulo *M(x)*. Al realizar la reducción el polinomio resultante adopta la forma $b_2 x^3 + b_1 x^2 + b_0 x + b_3$. Si se piensa la operación desde el punto de vista matricial, es equivalente a la multiplicación por una matriz cuyos $a_j = 00$ excepto $a_1 = 01$, como se expresa en la Ec.(3.45)

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} 00 & 00 & 00 & 01 \\ 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{3.45}$$

De este modo, la multiplicación por *x* se convierte en un corrimiento cíclico de los bytes dentro del vector en cuestión. La función *RotWord()* [DAE99] que se utiliza en la generación de las diversas sub-claves para cada ronda corresponde a una multiplicación por x^3.

Capítulo 4

Cifrador Bloque Rijndael

El algoritmo *Rijndael* es un cifrador de bloque iterativo con una longitud variable, tanto de bloque como de clave, que se pueden establecer independientemente en *128*, *192* o *256* bits, en cada caso el número de rondas se fijará en *10*, *12* o *14* respectivamente. Fue ideado por *Joan Daemen* y *Vincent Rijmen* y enviado al *National Institute of Standards and Technology (NIST)* como propuesta para el nuevo algoritmo de cifrado estándar, *Advanced Encryption Standard (AES).* Luego de dos años de intenso trabajo, el *NIST* lo seleccionó de entre un grupo de varios candidatos y editó la *Federal Information Processing Standards Publications (FIPS PUBS) 197*, con fecha *Noviembre 26, 2001*, anunciando el *AES*. El nuevo estándar adopta el algoritmo Rijndael como un cifrador de bloque simétrico que puede procesar bloques de 128 bits usando claves de *128*, *192*, y *256* bits, referenciando los tres modos como *AES-128*, *AES-192* y *AES-256*.

4.1 La especificación

Se trata de un cifrador de bloque iterativo que toma un bloque de información por vez, conocido como *texto pleno* o *mensaje (plaintext)*, y lo transforma en otro bloque de la misma longitud, llamado *texto cifrado (ciphertext)*, utilizando un tercer bloque como base de la transformación: *clave de cifrado*, *llave* o simplemente *clave (cipher key*).

El cifrador fue diseñado teniendo en cuenta tres criterios: un máximo de resistencia frente a ataques conocidos, sencillez en el diseño y que su implementación pueda realizarse de manera compacta teniendo en cuenta a la vez características de velocidad y adaptabilidad a diferentes plataformas [DAE99]. Una ronda se compone de tres transformaciones individuales distintas que los autores llamaron capas o niveles, en inglés *layers*. Se trata de transformaciones uniformes en el

sentido que cada bit del bloque a encriptar se trata de la misma forma. A diferencia de *DES*, el cifrador no posee una estructura *Feistel*.

En el cifrador Rijndael los bloques de entrada y salida, así como también la clave, pueden especificarse independientemente en *128*, *192* ó *256* bits, pero el estándar *AES* define una longitud de bloque de entrada de *128* bits, mientras que la longitud de la clave puede ser de cualquiera de los valores mencionados, dando lugar a tres posibles implementaciones: *AES-128, AES-192* y *AES-256* [FIPS197].

La Fig. 4.1 presenta un diagrama de flujo del algoritmo.

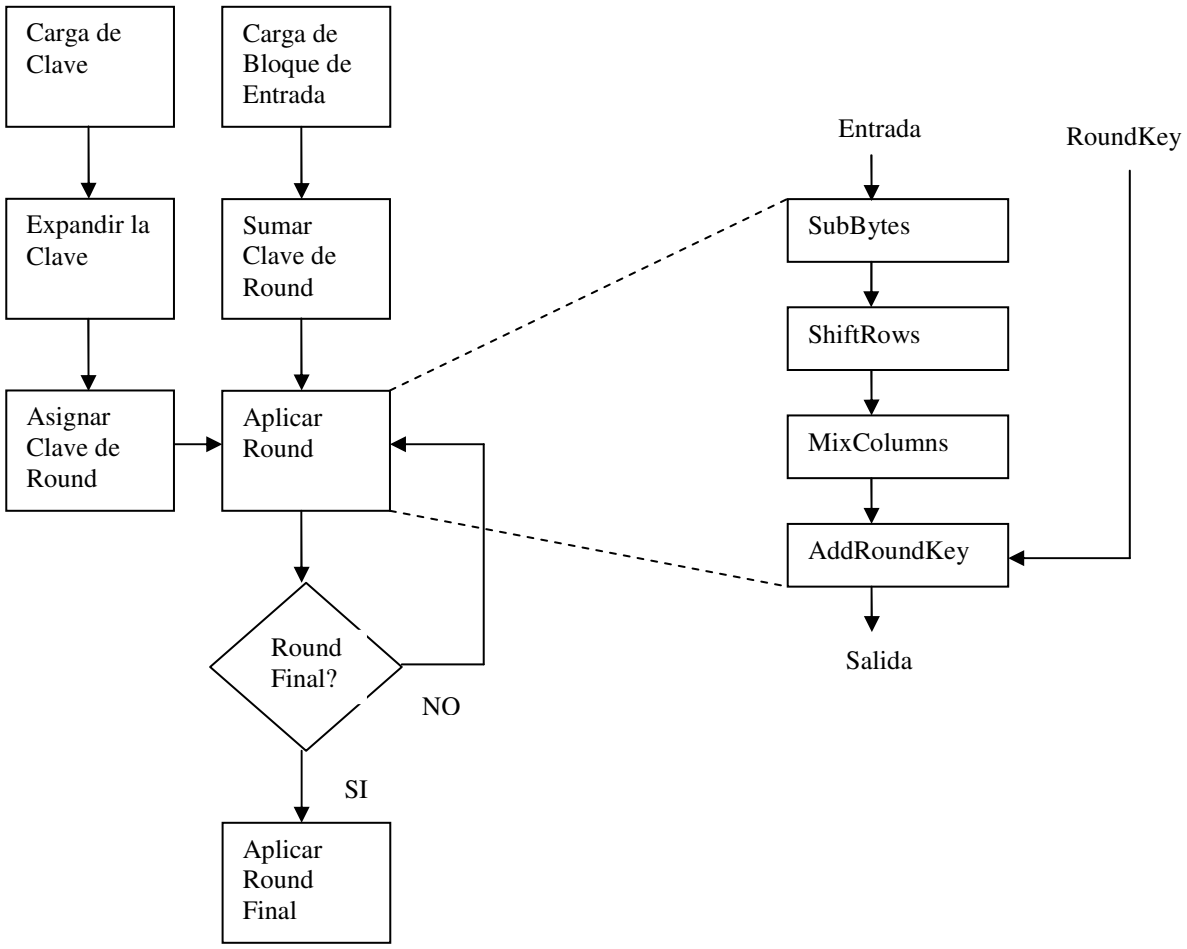

Figura 4.1. Cifrador Rijndael.

En un cifrador iterativo, el cifrado de los datos se realiza a través de la aplicación repetitiva de una función conocida como ronda, en inglés *round*, parametrizada por una transformación paralela que se realiza sobre la clave original. Las transformaciones que constituyen una ronda se asocian a las propiedades de confusión (*confussion*) y difusión (*difussion*) tan apreciadas en criptografía [MIT92].

Típicamente el cifrador se puede considerar dividido en dos partes con diferente funcionalidad: la transformación del mensaje por un lado y la transformación de la clave, en inglés *key scheduling*, por el otro. Cada ronda implica la aplicación de una clave diferente, conocida como sub-clave, derivada a partir de la clave principal. Por este motivo surge la necesidad de la expansión de la clave original en una cantidad de bits suficiente como para cubrir todas las rondas. La cantidad de rondas total es variable: *10*, *12* o *14*, dependiendo del tamaño de la clave elegido originalmente: *128*, *192* o *256* bits [DAE99], [BIH99].

Para el algoritmo *AES* la longitud del bloque, tanto de entrada como de salida, es de *128* bits representada a través de la variable $N_b = 4$. Este valor de la variable puede interpretarse como la aplicación del algoritmo a bloques de *4* palabras de *32* bits cada una.

Internamente las operaciones se realizan sobre un arreglo de dos dimensiones llamado Matriz de Estado, en inglés *State*. Este arreglo consiste de *4* filas cada una de N_b bytes, donde N_b resulta de dividir la longitud del bloque por *32*. Es decir que, en el caso *AES-128*, $N_b = 4$. Cada byte individual del *State* se refiere como $s_{r,c}$, donde r representa el número de fila, $0 \leq r \leq 4$ y c el número de columna, con $0 \leq c \leq 4$.

La clave de cifrado puede ser una secuencia de *128*, *192* o *256* bits y su longitud queda fijada por el valor de la variable $N_k = 4$, *6* u *8*, con la misma interpretación que N_b, es decir representa la cantidad de palabras de *32* bits que forman la clave.

Como se expresó anteriormente, el número de rondas a ser desarrolladas durante la ejecución es dependiente del tamaño de la clave y se representa en una variable N_r que puede tomar los valores *10*, *12* o *14* según que N_k valga *4*, *6* u *8* respectivamente. La tabla 4.1 presenta las únicas combinaciones de *Clave-Bloque-Ronda* aceptadas para el estándar [FIPS197]:

	N_k	N_b	N_r
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Tabla 4.1. Combinaciones de clave y tamaño de bloque

Tanto para el cifrado de datos como para la recuperación a partir del texto cifrado, el algoritmo utiliza cuatro transformaciones o sus inversas que operan sobre un resultado intermedio almacenado temporalmente en la Matriz de Estado. Todas las transformaciones son orientadas al byte y se conocen y definen en el algoritmo original como:

- *SubBytes()*: se trata de una función de sustitución que opera individualmente sobre cada byte de la Matriz de Estado. Realiza la inversión del byte en *GF(2^8)* y la aplicación de una transformación afín [RUD01]. Por similitud con otros algoritmos se suele denominar a esta transformación caja *S*, en inglés *S-box*.
- *ShiftRows()*: es una operación que realiza una rotación cíclica de diferente magnitud según la fila de la Matriz de Estado sobre la que opere.
- *MixColumns()*: por efecto de esta transformación, cada columna de la Matriz de Estado se representa como un polinomio de grado *3* sobre *GF(2^8)* y es multiplicado por un polinomio prefijado. La operación se realiza módulo otro polinomio, también fijo, $M(x) = x^4 + 1$.
- *AddRoundKey()*: consiste en la adición en *GF* de la clave con la Matriz de Estado. La aplicación práctica se traduce en la suma *EXOR* bit a bit entre bloques de *128* bits.

Se volverá sobre estas transformaciones más adelante en este capítulo.

4.2 Representación y Principios Matemáticos

Los bits de las secuencias o bloques a tratar se pueden numerar desde *0* hasta la longitud total de la secuencia menos *1*. La entrada, la salida y la clave de cifrado se procesan como arreglos de bytes, la unidad básica de procesamiento en el algoritmo *AES*. Los bytes del arreglo respectivo se referencian como a_n ó *a[n]* con *n* en alguno de los siguientes rangos dependiendo de la longitud de la clave original:

- *Longitud de la clave = 128 bits,* $0 \leq n < 16$.
- *Longitud de la clave = 192 bits,* $0 \leq n < 24$.
- *Longitud de la clave = 256 bits,* $0 \leq n < 32$.

De este modo el valor de *n* representa la cantidad de bytes de la Matriz de Estado. Por su parte, cada byte se representa como una secuencia de *8* bits {b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0} y cada bit puede interpretarse como un elemento del campo finito *GF(2)*. De este modo, puede trabajarse sobre los bytes a procesar mediante la representación polinomial de la Ec.(3.18).

Esta representación permite trabajar sencillamente con la operación suma, considerada como la *suma módulo-2* de los coeficientes de los polinomios en cuestión. Como se explicó anteriormente en el Capítulo 3, se puede pensar en esta operación como una simple *EXOR* entre bytes o conjuntos de *8* bits. Esta no es esta la única interpretación posible. A veces es conveniente recurrir a la representación hexadecimal. Por ejemplo, el byte {*0111 0011*}, tiene su representación polinomial en $x^6 + x^5 + x^4 + x + 1$ y una representación hexadecimal (*0x73*).

En el caso de la multiplicación (.) entre elementos de $GF(2^8)$, la operación se corresponde con la multiplicación entre polinomios módulo un polinomio binario irreducible de grado *8*. En

Rijndael dicho polinomio es $m(x) = x^8 + x^4 + x^3 + x + 1$ ó (*0x01*)(*0x1b*) en notación hexadecimal. La reducción modular con *m(x)* asegura que el resultado sea un polinomio de grado inferior a *8* y pueda, de esta manera, representarse como un byte. A diferencia de la adición, esta operación no es sencilla en cuanto a su implementación. De todas maneras, la multiplicación cumple con la propiedad asociativa, tiene un elemento identidad, (*0x01*), y para cada elemento no nulo *b(x)* de grado menor que *8,* puede hallarse la inversa multiplicativa $b^{-1}(x)$ *a*plicando el *Algoritmo Extendido de Euclides*. Es decir que el conjunto de *256* valores de bytes posibles junto con las operaciones de adición y multiplicación mencionadas tienen la estructura de un campo finito *GF(2^8)*.

Al trabajar con palabras de *32* bits, se pueden también definir polinomios de grado menor a *4*, con coeficientes en *GF(2^8)*. Se puede de este modo representar vectores de *4* bytes o palabras de 32 bits. Estos polinomios no son los mismos que los usados en la definición de elementos de campo finito, ya que los coeficientes en sí mismos son elementos de campo finito, es decir bytes. En este caso, se define la suma como antes, por adición de los coeficientes de campo finito de las potencias iguales de *x*. La multiplicación resulta en un polinomio de mayor grado que no se puede representar por un vector de *4* bytes, por lo que hay que reducirlo por medio de otro polinomio. En el algoritmo Rijndael este polinomio es *M(x)*, presentado en la Ec.(3.36), y posee la particularidad de no ser reducible pero poseer inversa.

Otra propiedad particular de las transformaciones presentes en la estructura del Algoritmo Rinjdael es que las operaciones de multiplicación allí definidas consisten en la multiplicación por un polinomio fijo cuya representación matricial es circulante. Esta propiedad se presenta en la transformación afín que forma parte de *SubBytes()* y en la multiplicación que define la operación *MixColumns()*. La diferencia es que en un caso se trabaja con bytes como elementos a transformar y en el otro con conjuntos de 4 bytes. La elección de estos polinomios se apoyó en los criterios de diseño perseguidos por los autores, sobre todo en cuanto a la realización de operaciones

invertibles, velocidad de procesamiento, simplicidad en la descripción y buenas características criptográficas [DAE99].

Otro polinomio fijo utilizado por el algoritmo es aquel con coeficientes $a_0 = a_1 = a_2 =$ (0x*00*) y $a_3 =$ (*0x01*). Se trata del polinomio x^3 y se utiliza en una de las funciones que componen la transformación de la clave. Si se inspecciona la representación matricial del producto, se ve que el efecto de multiplicar por x^3 una palabra de *32* bits, es la rotación de los bytes de entrada $[b_0, b_1, b_2, b_3]$ a $[b_1, b_2, b_3, b_0]$, lo cual facilita su implementación.

4.3 El Cifrador

La transformación por ronda de este cifrador no posee la estructura de Feistel característica de muchos de sus antecesores. A diferencia de ellos se considera compuesta de tres transformaciones uniformes e invertibles, llamadas capas o niveles. La estrategia de diseño con resistencia a criptoanálisis lineal y diferencial denominada *Wide Trail Strategy* constituyó la base de la elección de los distintos niveles, ya que cada nivel tiene su propia función [DAE99]. El *nivel de mezclado lineal* garantiza difusión sobre múltiples rondas. El *nivel alineal* consiste en la aplicación de la transformación *SubBytes()* y posee propiedades óptimas en cuanto al peor caso de alinealidad por la aplicación en paralelo de cajas *S*. El *nivel de agregado de clave* es simplemente una *EXOR* de la clave de ronda con la Matriz de Estado intermedio. Antes de la primer ronda se aplica un *nivel de agregado de clave* ya que cualquier nivel luego de la última adición de clave en el cifrador, o antes de la primera en el contexto de ataques de texto pleno conocido, puede quitarse sin el conocimiento de la clave y por tanto no contribuye a la seguridad del cifrador como sucede con las permutaciones inicial y final de *DES*. Simplemente se rescata la operación de adición de la clave por ser esta una operación común en casi todos los cifradores simétricos tipo bloque más conocidos.

Para que el cifrador y su inversa sean más similares en cuanto a estructura, se deben

diferenciar el *nivel de mezclado lineal* del último round respecto de los otros. En este sentido en la última ronda no se realiza la operación *MixColumns()*. Esto no modifica la seguridad del cifrador y es similar a la ausencia de rotación en la última ronda de *DES*.

Al comienzo del cifrado, el bloque de entrada de datos considerado como una sucesión de *16* bytes, ($in_0, in_1, ..., in_{15}$), simplemente se copia a la Matriz de Estado, *State*, referenciada con la letra *s*. Se realizan las operaciones correspondientes sobre este arreglo implementando la función de ronda *10*, *12* o *14* veces según cuál sea la longitud de la clave. La ronda final difiere un poco de las $N_r - 1$ rondas previas. Finalmente el resultado se copia en el arreglo de salida ($out_0, out_1, ..., out_{15}$), como se representa en la Fig. 4.2.

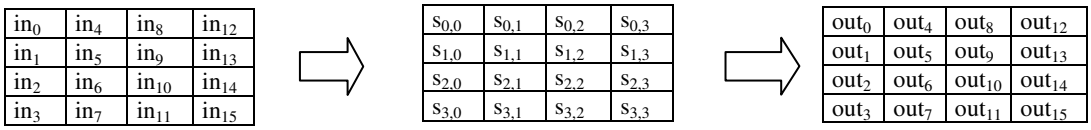

Figura 4.2. Bloque de entrada, resultado intermedio y bloque de salida.

Si se considera un bloque de entrada de *128* bits, los *4* bytes de cada columna de *s* forman palabras de *32 bits* y el número de fila *r* resulta un índice para los *4* bytes dentro de cada palabra. Por este motivo, *s* también puede interpretarse como un arreglo unidimensional de palabras w_c de *32 bits* con el número de columna *c* como índice. Las ecuaciones Ec.(4.1) a Ec.(4.4) presentan la representación matemática de las *4* columnas del bloque de *128* bits a ser procesado.

$$w_0 = s_{0,0} s_{1,0} s_{2,0} s_{3,0} \tag{4.1}$$

$$w_1 = s_{0,1} s_{1,1} s_{2,1} s_{3,1} \tag{4.2}$$

$$w_2 = s_{0,2} s_{1,2} s_{2,2} s_{3,2} \tag{4.3}$$

$$w_3 = s_{0,3} s_{1,3} s_{2,3} s_{3,3} \tag{4.4}$$

El siguiente pseudo-código representa la operación de cifrado:

```
Cipher (byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)] )
{
byte state [4, Nb];
state = in;

AddRoundKey(state, w[0, Nb-1]);

for (i =1; i < Nr ; i++)
        SubBytes(state);
        ShiftRows(state);
        MixColumns(state);
        AddRoundKey(state, w[round * Nb, (round+1)*Nb-1]);

SubBytes(state);
ShiftRows(state);
AddRoundKey(state, w[Nr* Nb, ( Nr +1)*Nb-1]);

out = state
}
```

La clave consiste de un arreglo uni-dimensional de palabras de *4 bytes* que surgen de la aplicación de la rutina de *Expansión de la Clave*. En cualquier ronda, una porción de dicho arreglo debe sumarse con los respectivos bits de la Matriz de Estado. De este modo, en el caso AES-128, la clave original de *128* bits debe expandirse mediante una serie de transformaciones hasta llegar a conformar una sucesión de *176* bytes. Se generan así *10* sub-claves, una para cada una de las rondas sobre las que itera el algoritmo. Como es posible apreciar en el pseudo-código presentado, todas las rondas, excepto la última, exigen la aplicación de *4* transformaciones: *SubBytes()*, *ShiftRows()*, *MixColumns()* y *AddRoundKey()*.

4.3.1 Transformación *SubBytes()*

Se trata de una sustitución alineal de bytes que opera de manera independiente sobre cada byte de la Matriz de Estado. Es una sustitución que posee su equivalente inverso para poder proceder al proceso de descifrado. Se construye a partir de dos transformaciones:

1) Como primer paso se debe obtener el inverso multiplicativo en el campo finito $GF(2^8)$ del elemento a transformar. El elemento (*0x00*) se transforma sobre sí mismo.
2) Luego se debe aplicar la siguiente transformación afín sobre *GF(2)*, válida para $0 \leq i < 8$ con b_i el bit i-ésimo del byte a transformar y c_i el bit i-ésimo del byte fijo *c* de valor (*0x63*) ó en binario (*01100011*):

$$b_i^{'} = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i \tag{4.5}$$

Esta segunda transformación se puede representar también de manera matricial como en la Ec.(4.6).

$$\begin{bmatrix} b_0^{'} \\ b_1^{'} \\ b_2^{'} \\ b_3^{'} \\ b_4^{'} \\ b_5^{'} \\ b_6^{'} \\ b_7^{'} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{4.6}$$

Se puede comprender entonces que esta sustitución puede ser implementada mediante circuitos que realicen la inversión sobre GF(2^8) que se apoyan en la aplicación del Algoritmo Extendido de Euclides [RIJ03].

También es posible, a partir de un elemento generador, desarrollar por medio de sus potencias, g^i con $0 \leq i < 255$, todos los elementos del campo y presentar esta información en una tabla. Esta propiedad ofrece una manera de convertir la operación de multiplicación en una operación de suma, como se explicó en el Capítulo anterior. De la misma forma es posible tabular los elementos inversos de cada elemento del campo finito, ya que en este caso se verificaría que las potencias g^x y g^{255-x} son elementos inversos entre sí. Si es posible tabular todos los elementos y sus inversos, también lo es tabular la transformación completa que representa *SubBytes()*, pues simplemente a cada elemento inverso se le aplica la transformación afín mencionada en el paso 2.

De este modo, se puede representar toda la transformación por medio de una tabla de valores fijos, denominada caja *S*. La misma se presenta en formato hexadecimal en la Tabla 4.2 [FIPS197]. Por ejemplo, si el byte a transformar es $s_{1,1}$ = (*0x53*), el valor de reemplazo se obtiene

de la intersección de la fila *5* y la columna *3*, es decir el resultado de la transformación es (*0xed*).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3ª	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7ª	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8ª
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Tabla 4.2. Caja *S*.

La transformación inversa de *SubBytes()*, *InvSubBytes()*, que se presenta en el caso del descifrado de los datos, resulta de aplicar la tabla inversa referenciada más adelante. La Fig. 4.3 representa el efecto de la aplicación de la operación *SubBytes()* sobre la matriz *s*: cada byte de la Matriz de Estado es reemplazado por el resultado de aplicar los dos pasos mencionados anteriormente.

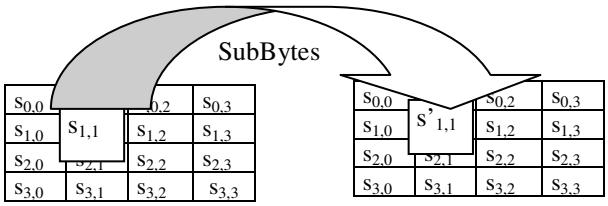

Figura 4.3. Transfomación *SubBytes()*.

4.3.2 Transformación *ShiftRows()*

En esta transformación se toman los bytes de la matriz *State* y se corren cíclicamente un número diferente de bytes u *offsets*, de acuerdo a la fila a la que pertenezcan. La primera fila no se rota. En general la transformación se puede expresar según la siguiente expresión, válida para $0 < r < 4$ y $0 \leq c < N_b$:

$$s'_{r,c} = s_{r,(c+shift(r,N_b)) \bmod N_b} \tag{4.7}$$

Para $N_b = 4$ el valor de *shift(r, N_b)* depende del número de fila *r*:

$$shift(1,4) = 1; shift(2,4) = 2; shift(3,4) = 3 \tag{4.8}$$

El efecto final de esta transformación es un corrimiento cíclico entre los bytes de una misma fila, resultando en una (C_1) posición para la primera, dos (C_2) para la segunda y tres (C_3) para la tercera, siempre que N_b sea 4. La Fig. 4.4 representa esta transformación.

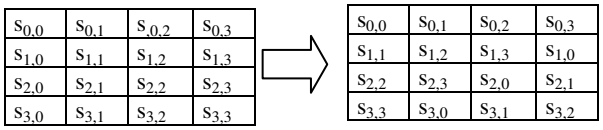

Figura 4.4. Corrimiento cíclico producido por *ShiftRows()*.

De la observación de la Fig. 4.4, puede inferirse que, de realizarse la operación de manera secuencial, no es posible continuar con la transformación siguiente hasta no haber rotado la última fila. Esto es debido a que *MixColumns()* opera sobre las columnas de la Matriz de Estado, comenzando con la primera, que no puede completarse hasta no haber rotado el byte $s_{3,3}$.

La transformación inversa de *ShiftRows()* es un corrimiento cíclico de las tres filas inferiores en $N_b - C_1$, $N_b - C_2$ y $N_b - C_3$ bytes respectivamente, de tal manera que un byte en la posición *j* en la fila *i* se mueve a la posición $(j + N_b - C_i)_{mod\ Nb}$. Específicamente, se puede expresar para $0 < r < 4$ y $0 \leq c < N_b$:

$$s'_{r,(c+shift(r,N_b)) \bmod N_b} = s_{r,c} \tag{4.9}$$

4.3.3 Transformación *MixColumns()*

Esta transformación se aplica sobre las columnas de la Matriz *State* interpretándolas como polinomios de cuatro términos sobre $GF(2^8)$. Se toman las columnas y se las multiplica módulo x^4+1 con un polinomio fijo y co-primo con el anterior, *a(x)*, de la forma:

$$a(x) = (0x03)x^3 + (0x01)x^2 + (0x01)x + (0x02) \tag{4.10}$$

Este producto expresado en forma matricial $s'(x) = a(x) \otimes s(x)$, para $0 \leq c < N_b$, se puede representar por medio de la Ec.(4.11).

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \tag{4.11}$$

De la inspección de la Ec. (4.11) es posible observar que la operación a nivel de bytes se traduce en la multiplicación por potencias de *x*, que pueden ser implementadas por aplicación repetida de la función *xtime()* mencionada en el Capítulo 3.

La Fig. 4.4 esquematiza la transformación *MixColumns()*. El reemplazo es por columnas de la Matriz de Estado. La obtención de cada byte de la columna resultante exige la presencia simultánea de los cuatro bytes de la columna original.

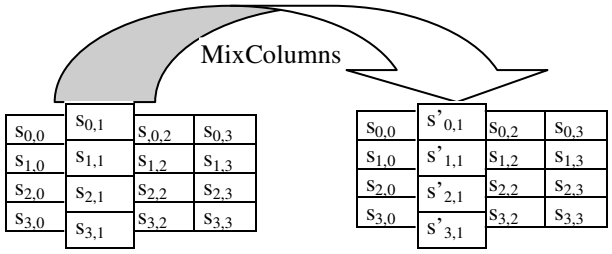

Figura 4.5. Transformación *MixColumns()*.

La inversa de la transformación *MixColumns()*, *InvMixColumns()*, exige la multiplicación de cada columna por un polinomio específico *d(x)* fijo, que verifica la Ec.(4.12). Se trata del polinomio inverso de *a(x)* que se presenta en la Ec.(4.13)

$$\left[(0x03)x^3 + (0x01)x^2 + (0x01)x + (0x02)\right] \otimes d(x) = (0x01) \tag{4.12}$$

$$=> d(x) = (0x0B)x^3 + (0x0D)x^2 + (0x09)x + (0x0E) \tag{4.13}$$

La transformación *InvMixColumns()* también tiene un equivalente matricial:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \tag{4.14}$$

4.3.4 Transformación *AddRoundKey()*

En esta transformación se agrega la clave de la ronda correspondiente a la matriz *State* mediante una operación *EXOR* que se puede expresar mediante la Ec.(4.15), válida para $0 \le c < N_b$ y $0 \le n^o\_de\_ronda < N_r$.

$$\begin{bmatrix} s'_{0,c} & s'_{1,c} & s'_{2,c} & s'_{3,c} \end{bmatrix} = \begin{bmatrix} s_{0,c} & s_{1,c} & s_{2,c} & s_{3,c} \end{bmatrix} \oplus \begin{bmatrix} w_{round*N_b+c} \end{bmatrix} \qquad (4.15)$$

Cada clave de ronda consiste en N_b palabras que provienen de la transformación que recibe la clave de cifrado. Para el cifrador, la suma inicial de la clave de ronda sucede previamente a la primer aplicación de la función de ronda. Esta operación es su propia inversa y esta propiedad es muy importante en cuanto a la eficiencia en el uso de componentes en la etapa de cifrado y la de descifrado.

4.4 Expansión de la clave

El algoritmo *AES* toma la clave de cifrado original, *K*, y realiza una rutina de expansión para generar la cantidad de claves necesarias para todas las rondas. La expansión genera un total de $N_b(N_r+1)$ palabras. El algoritmo utiliza un conjunto inicial de N_b palabras que provienen de la clave original elegida por el usuario o generada mediante alguna técnica de generación de claves. Cada una de las N_r rondas requiere también N_b palabras que se conocen como sub-claves. El resultado de la expansión consiste de un arreglo lineal de palabras de 4 bytes, denotado [w_i], con $0 \le i < N_b(N_r+1)$.

Se puede representar la expansión de la clave mediante el siguiente pseudo-código:

```
KeyExpansion (byte key[4*Nk],  word w[Nb*(Nr+1)], Nk )
begin
        word temp
        i = 0
        while ( i < Nk )
                w[i]= word (key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
                i = i+1
        end while
        i = Nk
        while ( i < Nb * (Nr + 1) )
                temp = w [i-1]
```

```
            if ( i mod Nk = 0)
                  temp = SubWord(RotWord(temp)) xor Rcon [i/Nk]
            else if ( Nk > 6 and i mod Nk =4)
                  temp = SubWord(temp)
            endif
            w[i] = w [i - Nk ] xor temp
            i = i+1
      endwhile
end
```

Notar que $N_k = 4, 6$ y 8 no tienen que implementarse completamente sino que se incluyen en la sentencia condicional anterior. La función *SubWord()* toma una palabra de *4* bytes de entrada y aplica la caja *S* a cada uno de los bytes para obtener una palabra de salida. *RotWord()* toma una palabra [a_0, a_1, a_2, a_3] de entrada y realiza sobre ella una permutación cíclica, retornando [a_1, a_2, a_3, a_0]. *Rcon[i]* contiene los valores dados por [x^{j-1}, (*0x00*), (*0x00*), (*0x00*)], siendo x^{j-1} las potencias de *x* en *GF(2*^{8}*)*, de tal manera que $x^0 = (0x01)$, $x^1 = (0x02)$, $x^2 = x\ .x = x.(0x02)$ y, en general $x^i = x.x^{i-1} = (0x02).x^{i-1}$.

Las primeras N_k palabras de la clave expandida se llenan con la clave de cifrado. Cada palabra subsiguiente *w[i]* se completa con una operación *EXOR* entre la palabra previa *w[i-1]* y la que está N_k posiciones antes *w[i-*N_k*]*. Para palabras en posiciones que son múltiplos de N_k se aplica una transformación a *w[i-1]* previo a la *EXOR*, y a continuación una *EXOR* con una constante propia de cada ronda, *Rcon[i]*. La transformación consiste de un corrimiento cíclico de bytes *RotWord()* seguido de la aplicación de una búsqueda en tabla para los cuatro bytes de la palabra *SubWord()*.

La rutina de expansión es diferente para claves de cifrado de *256* bits ($N_k = 8$). En este caso, si *(i-4)* es múltiplo de N_k, se aplica *SubWord()* a *w[i-1]* previo a la *EXOR*.

4.5 Descifrado

La operación de desencriptado se puede representar con el siguiente pseudocódigo:

```
InvCipher (byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)] )
{
byte state [4, Nb];
state = in;

AddRoundKey(state, w[Nr* Nb, (Nr +1)* Nb-1]);
for (i =Nr-1 ; i ≥ -1 ; i--)

        InvShiftRows(state);
        InvSubBytes(state);
        AddRoundKey(state, w[round * Nb, (round+1)*Nb-1]);
        InvMixColumns(state);

InvShiftRows(state);
InvSubBytes(state);
AddRoundKey(state, w[0, Nb-1]);

out = state
}
```

Como puede observarse, la secuencia de las transformaciones difiere del caso del cifrado. En particular las transformaciones de la caja *S* y la rotación se aplican en orden inverso. También la sub-clave se suma en cada ronda previamente a la aplicación de *MixColumns()*. El tratamiento de la clave es el mismo pero las sub-claves se utilizan en orden inverso respecto del caso del cifrador. Este orden invertido de las operaciones afecta la implementación del descifrador si su diseño se apoyara en el circuito cifrador. El aprovechamiento de los diversos componentes entre ambos circuitos se vería afectado negativamente.

Afortunadamente, varias propiedades del algoritmo permiten la existencia de un cifrador inverso equivalente que posee las mismas secuencias de transformaciones que el cifrador, aunque reemplazadas por sus inversas, pero presenta un cambio en el tratamiento de la clave.

Las propiedades que permiten el equivalente inverso son [DAE99]:

- *SubBytes()* y *ShiftRows()* pueden conmutar entre sí y lo mismo es cierto para sus inversas. De este modo se verifican las siguientes ecuaciones:

$$ShiftRows[SubBytes(State)] = SubBytes[ShiftRows(State)] \quad (4.16)$$

$$InvShiftRows[InvSubBytes(State)] = InvSubBytes[InvShiftRows(State)] \quad (4.17)$$

- *MixColumns()* e *InvMixColumns()* son lineales con respecto a la operación por columna.

Por lo que se verifica que:

$$InvMixColumns(State \oplus RoundKey) = InvMixColumns(State) \oplus InvMixColumns(RoundKey) \quad (4.18)$$

De este modo, es posible invertir el orden de aplicación de las transformaciones *InvSubBytes()* e *InvShiftRows()*. Lo mismo sucede con el orden de *AddRoundKey()* e *InvMixColumns()* siempre que las palabras o columnas de la clave de descifrado se modifiquen con la transformación *InvMixColumns()*, excepto para el caso de las primeras y últimas N_b palabras.

Así, el cifrador inverso equivalente mantiene el mismo orden de las operaciones que el cifrador original, permitiendo así mayor aprovechamiento de los componentes del circuito cifrador y otorgando mayor eficiencia en lo que respecta a la implementación. El pseudo-código para el cifrador se transforma entonces de la siguiente manera:

```
EqInvCipher(byte in[4*Nb], byte out[4*Nb], word dw[Nb*(Nr+1)])
{
    byte state[4,Nb]
    state = in

    AddRoundKey(state, dw[Nr*Nb, (Nr+1)*Nb-1])

        For round = Nr-1 step -1 downto 1
        InvSubBytes(state)
        InvShiftRows(state)
        InvMixColumns(state)
        AddRoundKey(state, dw[round*Nb, (round+1)*Nb-1])
        end for

    InvSubBytes(state)
    InvShiftRows(state)
    AddRoundKey(state, dw[0, Nb-1])

    out = state
}
```

4.6 Motivación de la elección para las Transformaciones

El algoritmo fue cuidadosamente diseñado, de tal manera que la elección de las

transformaciones que lo componen contribuye a la fortaleza del mismo frente a diversos tipos de ataques [DAE99].

Las cajas *S* han sido diseñadas para soportar varios tipos de ataques derivados tanto del criptoanálisis diferencial como del lineal. No se han descuidado tampoco aspectos relacionados con ataques que involucren manipulaciones algebraicas. En este sentido se ha elegido una caja *S* que reemplaza cualquier elemento de *GF(2^8)* por su elemento inverso, y, para evitar ataques de interpolación se ha modificado el mapeo mediante una transformación afín adicional que también es invertible. La transformación SubBytes() puede, de este modo, expresarse como:

$$b(x) = (x^7 + x^6 + x^2 + x) + a(x)(x^7 + x^6 + x^5 + x^4 + 1)\Big|_{\mathrm{mod}(x^8+1)} \qquad (4.19)$$

Se trata de una expresión algebraica en la que el módulo ha sido elegido entre los más sencillos posibles y el polinomio multiplicador, por su parte, es co-primo con el módulo. La constante es tal que la caja S no presenta puntos fijos ni puntos fijos opuestos, esto es $SubBytes(a) = a$ ó $SubBytes(a) = \overline{a}$.

La transformación *MixColumns()* ha sido elegida de entre el espacio de transformaciones lineales de *4* bytes a *4* bytes, teniendo fundamentalmente en cuenta su potencia de difusión, lo cual impone restricciones en los valores de los coeficientes de $a(x)$. La propiedad fundamental de la transformación lineal se basa en el número de ramas, en inglés *branch number*, que es una medida de su potencia de difusión. En este algoritmo dicha medida se ha llevado al máximo posible con la elección apropiada de los coeficientes. El módulo *M(x)* se eligió para reforzar propiedades tales como linealidad, simetría y simplicidad.

Los corrimientos que derivan de la transformación *ShiftRows()* se han elegido teniendo en cuenta sobre todo la resistencia frente a diversos ataques conocidos.

En cuanto a la Expansión de la Clave, se tuvieron en cuenta ataques del texto cifrado conocido o ataques de clave conocida o que pueda ser elegida para usos especiales del algoritmo en funciones de compresión o funciones *hash*. También se buscó cumplir la propiedad que dos claves de cifrado diferentes no tengan un gran conjunto de sub-claves en común para evitar ataques relacionados con la clave. La simetría por ronda que el algoritmo conlleva en cuanto a que todos los bytes de la Matriz de Estado son procesados de la misma manera es removida con la utilización de distintas constantes en el esquema de expansión. La simetría entre rondas, inevitable pues en todas las rondas se realizan las mismas operaciones, se elimina por medio de constantes por ronda que dependen del número de ronda en cuestión.

El número total de rondas se determinó teniendo en cuenta el número mínimo de rondas en que el algoritmo puede ser atacado con éxito mediante el empleo de técnicas conocidas.

Se espera entonces que el ataque más eficiente para recuperación de la clave se base en la búsqueda exhaustiva de la misma. De este modo el algoritmo se comporta de la mejor manera posible para un cifrador de bloque simétrico. El esfuerzo esperado con respecto a la búsqueda exhaustiva demandará 2^{127} aplicaciones del algoritmo cuando se trabaja con una clave de *128* bits, como es el caso de *AES-128*.

Capítulo 5

Estrategia de Diseño de Bloques Fundamentales

En este capítulo se desarrollan los aspectos más relevantes relacionados con la implementación en hardware de los diversos componentes del cifrador. Los autores del algoritmo, *Joan Daemen* y *Vincent Rijmen*, abordaron este aspecto cuando elevaron la propuesta para el nuevo estándar de cifrado de datos. Concentraron sus consideraciones en desarrollos sobre procesadores de *8* bits para tarjetas inteligentes y procesadores de *32* bits para *PC*. Para el desarrollo utilizaron lenguaje ensamblador en el primer caso y lenguaje *C* en el segundo. Midieron los resultados por cantidad de ciclos de reloj que llevaba correr el algoritmo para un bloque de datos y longitud de líneas de código expresadas en cantidad de bytes. En ambos casos basaron la realización en tablas de búsqueda y resaltaron el compromiso existente entre velocidad y cantidad de memoria ocupada. Consideraron el paralelismo propio de la transformación por ronda como una importante ventaja para futuros procesadores y tecnología de hardware dedicado [DAE99].

Como sucede en el caso del software, una implementación en hardware puede optimizarse en lo que respecta a los parámetros de tamaño o velocidad. Sin embargo, el tamaño de un circuito se traslada en forma más directa a su costo que en el caso de una implementación en software. Duplicar el tamaño de un programa de cifrado no hace tanta diferencia en una computadora de propósito general con mucha memoria, pero duplicar el área usada por un dispositivo probablemente más que duplicará su costo.

5.1 Tecnología de Arreglo de Compuertas Programables

Una alternativa más que interesante para la implementación se apoya en una nueva tecnología que recién se estaba imponiendo en el mercado para la época de la propuesta Rijndael. Se trata de la tecnología de dispositivos de hardware reconfigurables, también conocidos como *Field Programmable Gate Arrays* (*FPGAs*). Las *FPGAs* consisten de un arreglo de elementos circuitales, llamados bloques lógicos, en conjunto con diversos recursos interconectados. La configuración es realizada por el

usuario final, lo cual facilita enormemente cualquier cambio en el diseño [BRO96].

Esta nueva tecnología ofrece muchas ventajas, en particular a aquellos investigadores dedicados al diseño de equipos para criptografía, y se presenta como la mejor alternativa de diseño de hardware de la actualidad. Su gran flexibilidad, bajo costo de desarrollo y las propiedades inherentes a la seguridad desde un punto de vista físico, son fundamentales en esta área de investigación.

La familia *FPGA* seleccionada para el presente diseño es *Altera Flex 10K*, en particular *EPF10K20*. Los dispositivos *FLEX 10K* fueron diseñados específicamente para cubrir las necesidades impuestas por los diseños de arreglos de compuertas de mediana densidad [ALT98]. En este caso el dispositivo es parte del *UP1 Educational Board of the University Program Design Laboratory Package* de *Altera*. La herramienta de software usada para sintetizar una implementación en *VHDL* del algoritmo *AES-128* es *MAX+PLUS II Version 7.21 Student Edition*, que se provee en el mismo paquete educacional [MAX96]. Esta herramienta también se puede utilizar para realizar simulaciones de comportamiento en el tiempo del circuito sujeto a diseño y de sus partes componentes.

El dispositivo cuenta con *1.152 celdas lógicas* ó *elementos lógicos* (*Logic Cells, LCs* ó *Logic Elements, LEs*) agrupadas en *144 Bloques de Arreglos Lógicos* (*Logical Array Blocks, LABs*). Estos *LABs* forman una estructura de *6* filas y *24* columnas [ALT98]. Se trata de un conjunto de recursos lógicos que se encuentran agrupados físicamente. Un *LAB* consiste de un arreglo de *LCs* y, en algunas familias de dispositivos incluye un arreglo de expansores de términos producto. Los *Bloques de Arreglos Embebidos* (*Embedded Array Blocks, EABs*) consisten de un conjunto de *8* celdas embebidas, físicamente agrupadas, que sirven para implementar memoria *RAM* ó *ROM* ó lógica combinacional. Un único *EAB* puede implementar un bloque de *256 x 8*, *512 x 4*, *1,024 x 2,* ó *2,048 x 1* bits de memoria. Cada celda de *Entrada/Salida* (*Input/Output, I/O*) contiene un buffer de *I/O* bidireccional y un *flip flop* que puede usarse como registro de salida o de entrada. La *EPF10K20* también contiene *6* entradas rápidas dedicadas para señales de control de entrada sincrónicas con grandes *fan-outs*. Cuando se usan como salidas, los registros *I/O* proveen tiempos rápidos *Clock-to-output*. Como entradas se privilegia el tiempo de establecimiento, en inglés *setup time*.

En este tipo de dispositivo, una celda lógica o elemento lógico consta de una tabla de búsqueda (*look-up table*, *LUT*), o sea un generador de funciones con el que se puede generar rápidamente cualquier función de *4* variables, y un registro programable para poder realizar funciones secuenciales y que puede programarse como *latch*, *flip flop D*, *T*, *JK*, ó *SR*. El registro puede alimentar otras *LCs* o realimentar la *LC* propia. Se pueden asignar funciones lógicas específicas a *LCs* específicas o a bloques de arreglos lógicos y asegurarse que una función se implemente en una *LC* de un *LAB* particular. Por otra parte, existen *LCs* especializadas, llamadas *I/O LCs* que se encuentran en la periferia del dispositivo.

5.2 Metodología de Diseño

Un circuito electrónico puede describirse como un módulo con determinada cantidad de entradas y salidas, donde los valores eléctricos adoptados por las salidas son función de los valores adoptados por las entradas. Esta visión sencilla puede aplicarse a circuitos simples y extenderse a sistemas más complejos entendiendo por sub-módulos a los componentes más sencillos. Los sub-módulos se conectan entre sí a través de sus puertos de entrada/salida, generando de este modo, módulos más complejos. Esta manera de describir el funcionamiento de un circuito da origen a una descripción de tipo estructural [PAR99].

En algunos casos la descripción estructural no es la apropiada. Esto se aprecia claramente en el caso de sub-módulos que se encuentran en la base de la jerarquía de alguna descripción estructural. En estos casos muchas veces no es necesario describir la estructura interna sino simplemente basta con una descripción de la funcionalidad del componente. La descripción de funcionalidad, sin referencia a la real estructura interna, se convierte de este modo en una descripción de comportamiento del componente en cuestión. El ejemplo más concreto de este tipo de descripciones es el de las compuertas lógicas [GRE95].

En términos habituales, el diseño del circuito o sistema comienza de la forma más generalizada posible, habitualmente con una especificación algorítmica en un lenguaje de alto nivel. En cada nivel

se refina la descripción correspondiente y se compara con la de nivel superior. Este método implica la utilización de herramientas de diseño asistido por computadora para simulación y prueba en cada nivel, y por ello los lenguajes deben proporcionar compatibilidad y consistencia entre niveles. De esta manera se trabaja de forma flexible y se logra que en un determinado momento coexistan descripciones de diferentes niveles que se pueden probar conjuntamente [ASH98].

Teniendo en cuenta estas consideraciones, se ha adoptado una metodología de diseño sistemática que contemple ambas posibilidades. Se ha trabajado a partir del concepto global del sistema a diseñar, descomponiendo su estructura en una serie de componentes capaces de interactuar entre sí de manera cooperativa para producir el comportamiento final esperado. Cada componente, a su vez, puede también considerarse constituido por otros y así hasta llegar al nivel más bajo, donde se encuentran los componentes primitivos que realizan una única función acotada. El resultado de este proceso es un sistema final con estructura jerárquica, construido a partir de elementos primitivos. La ventaja de esta metodología de diseño es la posibilidad de trabajar sobre cada parte componente de manera independiente, concentrándose sólo en los aspectos relevantes al mismo [ASH02].

En este sentido, uno de los desafíos más importantes consiste en la partición del modelo en sus diversos elementos componentes. Esta división no es arbitraria, puesto que guarda relación directa con las diferentes transformaciones que forman parte del circuito a diseñar, en este caso el cifrador.

De este modo, el método de diseño adoptado para realizar la descripción de los diversos módulos componentes se conoce con el término *Bottom-Up* [PAR99]. Este criterio involucra la descripción de los componentes más sencillos, para más tarde agruparlos en diferentes módulos, y estos a su vez en otros módulos, hasta llegar a un único módulo que representa el sistema completo. Dentro de este esquema, es posible trabajar en diferentes niveles. Uno de ellos lo constituye el *Nivel de Transferencia de Registros*, en inglés *Register Transfer Level*, *RTL*. En este nivel descriptivo el sistema se considera compuesto por elementos básicos tales como registros, memorias, lógica

combinacional y buses [ASH02]. En general, se distingue entre elementos con capacidad de almacenamiento y elementos sin ella.

La elección de esta metodología de diseño se basa principalmente en las limitaciones inherentes a la *FPGA* seleccionada. Un diseño *Top-Down* implicaría partir de una descripción abstracta, del tipo funcional o algorítmico, para más tarde llegar a los elementos primarios [PAR99]. Este tipo de diseño permitiría la realización o síntesis a partir del concepto de sistema, en este caso un algoritmo. El problema del diseño del cifrador se reduciría de esta manera prácticamente a la posibilidad de manejo de bloques de *128* bits desde la entrada hacia la salida del circuito, situación imposible de llevar a la práctica dada la cantidad de pines del dispositivo final del que se dispone [ALT98].

El concepto de diseño *Top-Down* se asocia íntimamente a la construcción de diseños jerárquicos. Por otro lado, los módulos ENTITY como estructura en el lenguaje *VHDL* también se asocian a la realización de diseños jerárquicos [ASH98]. En este sentido se ha aplicado un criterio *Top-Down* en diversas particiones del diseño original, sobre todo en descripciones asociadas a las unidades de procesamiento más sencillas.

5.3 Elección del Lenguaje

El lenguaje elegido para la descripción del circuito es *VHDL* (ó *VHSIC*, *Very High Speed Integrated Circuits Hardware Description Language*). Se trata de un lenguaje de descripción y modelado, que actualmente también se utiliza en simulación y síntesis [ASH02]. La elección se fundamenta principalmente en la posibilidad de utilización sobre diversas herramientas de creación e implementación. Esta propiedad minimiza la probabilidad de re-diseño cuando se pretende utilizar diversas tecnologías.

Independientemente de los beneficios obtenidos a partir de las características concretas del lenguaje, VHDL es un estándar no sometido a ninguna patente o marca registrada. Al ser mantenido y documentado por el *IEEE*, existe una garantía de estabilidad y soporte [TER97].

Con este lenguaje es posible lograr la simplificación de un problema complejo mediante las técnicas de partición y jerarquía. Así, la arquitectura puede describirse en función de bloques, procesos y componentes externos. Los bloques de un nivel pueden describirse en función de los bloques, procesos y componentes de un nivel de jerarquía inferior, y así sucesivamente. Los procesos pueden estructurarse en procedimientos y funciones.

El uso de un lenguaje estándar estable permite la reutilización en diseños futuros de las descripciones y datos generados durante el diseño actual con el consiguiente ahorro de recursos. VHDL facilita la reutilización del diseño gracias a varios factores.

VHDL esta basado en los conceptos de simulación discreta dirigida por eventos [PAR99]. Por ello, los modelos expresados en este lenguaje son muy fáciles de simular. Este era el propósito inicial del lenguaje además del de documentación. Posteriormente se comenzó a utilizar cada vez más para otros propósitos, especialmente el de síntesis. Sin embargo, algunas de las primitivas del lenguaje son difíciles de sintetizar o no tienen una correspondencia clara con el hardware. Por eso, los vendedores de herramientas de síntesis han definido subconjuntos para sus sintetizadores. Distintos sintetizadores admiten distintos subconjuntos y por ello se pierde en gran medida la ventaja de lenguaje estandarizado. Estas herramientas permiten un aumento importante en la productividad del proceso de diseño y, en consecuencia, abordar diseños con la complejidad que actualmente soporta la tecnología.

En lo referente al diseño en sí, el lenguaje se caracteriza por permitir una descripción Top-Down, así como también la descomposición del diseño en bloques, característica conocida como modularidad. VHDL puede ser usado como un lenguaje común de *Netlist* (*Lista de conexiones*) o para descripción de comportamiento o funcional [PAR99]. Los estilos de descripción varían con el nivel de abstracción. La descripción *algorítmica* sigue una estructura muy similar a los lenguajes de programación convencionales y se asocia a la ejecución secuencial. La descripción por *flujo de datos* es más cercana a la realización física y se encuentra más asociada a la ejecución concurrente. La descripción *estructural* es el uso del lenguaje como *Netlist*.

En la implementación desarrollada se utilizan los tres tipos de descripciones. Los circuitos en su conjunto describen lógica combinacional y secuencial.

Por otro lado, el software *Max+Plus II* de *Altera* es un sistema completo de diseño de hardware, por lo que la herramienta de síntesis a partir de VHDL es sólo una de las herramientas del sistema. Una vez armada una jerarquía de diseño, se la puede compilar para luego proceder a la simulación. Esta última no es la del código programado, sino la del dispositivo seleccionado. La definición de estímulos es realizable a través de archivos de formas de onda (*.scf*) pero en el caso de modelos más complejos es posible definir un *banco de pruebas* modelado en *VHDL*. Por cuestiones de sencillez las simulaciones se realizan mediante el primer método mencionado.

5.4 Opciones de Arquitectura

La elección de una arquitectura apropiada es fundamental en el diseño de circuitos de cifradores de bloque. La arquitectura determina finalmente los parámetros característicos de funcionamiento y ocupación de área del circuito bajo consideración. Su elección es uno de los problemas a resolver más importantes al comienzo del diseño [WEA03].

El Algoritmo Rijndael es un cifrador de bloque con una arquitectura interna con realimentación, en inglés *looping*, en la que los datos pasan iterativamente a través de una función denominada ronda. Existen varias opciones en cuanto a la implementación de este tipo de arquitectura que pueden conducir a diseños optimizados.

En el caso en que un algoritmo presenta una estructura interna homogénea en cada ronda de su estructura iterativa, probablemente pueda implementarse en hardware como una estructura de ronda en lógica combinacional. Es decir su estructura constará de circuitos cuyas salidas dependan solamente de las entradas presentes. En un período de reloj del sistema los datos ingresarán al circuito a través de algún elemento multiplexor y la salida resultante se almacenará en algún registro. En cada período de reloj subsiguiente se evaluaría una ronda del algoritmo, precisándose para el

cifrado tantos períodos como rondas comprenda el procesamiento total. A este tipo de estructura se la denomina *arquitectura básica* y supone trabajar con un bloque completo de datos por vez [GAJ01], [WEA02]. Se trata de una arquitectura versátil en cuanto a su utilización en diversos modos criptográficos con o sin realimentación. Sin embargo no se explota en este tipo de arquitectura la posibilidad de utilizar varios componentes del circuito al mismo tiempo para introducir algún grado de paralelismo en el procesamiento.

La arquitectura básica no es adaptable de manera directa a algoritmos con rondas de estructura heterogénea, como es el caso Rijndael donde la ronda inicial y la última difieren de las intermedias. También sería preciso dotar a la estructura original de algún esquema de multiplexado para el ingreso de las sub-claves en cada ronda.

En algunos casos, la arquitectura básica puede modificarse por partición de la lógica dentro de una ronda mediante registros que separan porciones individuales de funcionalidad propia. Puede ser posible realizar la partición de manera que la latencia asociada con cada una de las porciones sea prácticamente la misma. Esta posibilidad representa la oportunidad de realizar algún grado de paralelismo entre los distintos bloques de datos a procesar. A este tipo de estructuras se las conoce con el nombre de arquitecturas con *paralelismo interno*, en inglés *internal pipelinning*, y sólo son aplicables en modos criptográficos sin realimentación [SHI02], [GAJ01], [HOD03], [WEA02]. De este modo se logra una mejora sustancial en el número de bloques procesados por unidad de tiempo, parámetro que aumenta casi linealmente con el número de etapas, pero el costo asociado es el aumento del área en término de registros internos. La dificultad más importante de este tipo de diseños se encuentra a la hora de efectuar la partición en bloques con retardos similares ya que esta posibilidad depende fuertemente de las operaciones involucradas [GAJ01].

Existe una variación de la arquitectura básica, conocida como desarrollo de lazo, en inglés *loop unrolling*, en la que se implementan *k* copias de una ronda, siendo *k* habitualmente un divisor del número total de rondas [ELB01], [GAJ01]. El efecto es un aumento del área y del período de reloj, básicamente en un factor proporcional a *k*. También se ve afectada la cantidad de memoria utilizada pues es preciso almacenar previamente *k* sub-claves para cada ciclo de reloj. Este tipo de

arquitectura suele utilizarse en modos criptográficos con realimentación para aumentar los parámetros de eficiencia relacionados con la velocidad del circuito.

Se podrían también colocar registros entre las *k* rondas desarrolladas, conformando de este modo un conjunto de *k* etapas con paralelismo externo, en inglés *external pipelining*. Como en el caso de *internal pipelining*, esto sólo sería aplicable a modos sin realimentar y la arquitectura resultante incluiría un aumento del área ocupada y la réplica de elementos tales como cajas *S*, generando una mejora en la velocidad del cifrador [HOD03].

Es posible también combinar ambos paralelismos, interno y externo, en una arquitectura conocida como híbrida [HOD04].

Generalmente lo que determina la elección de una arquitectura es el objetivo del diseño. La existencia de varios objetivos puede producir decisiones incompatibles. En este sentido, los objetivos de minimización de área y maximización de velocidad son incompatibles. Por ejemplo, una arquitectura que desarrolle todas las rondas de un algoritmo puede maximizar la velocidad de procesamiento pero incrementa notablemente el área ocupada, reduciendo de este modo la eficiencia general del diseño.

Al examinar los aspectos principales del *AES*, resulta obvio que una implementación capaz de manejar bloques completos de 128 bits en el flujo (*stream*) de datos, resultaría en un cifrador con capacidad de procesamiento de un bloque de 128 bits por ciclo del reloj del sistema. Suponiendo un reloj de *25 MHz*, esta capacidad se trasladaría a una velocidad o *throughput* de *3.2Gbps*. Este tipo de implementación consumiría demasiados recursos en términos de área. Además requeriría una gran cantidad de patas o *pins* de *I/O* (*entrada/salida*) y no se podría sintetizar en una *FPGA* pequeña [HOL04].

Dado que la *EPF10k20* es la *FPGA* utilizada y se encuentra disponible en un circuito de *240-pin* en empaquetado *RQFP package* con *183 I/O pins*, no es posible sintetizar sobre esta *FPGA* este tipo de arquitectura [ALT98].

Se presenta entonces el primer interrogante a responder respecto de la estructura interna del diseño. Esta decisión implica la elección de un número de bits como unidad de procesamiento interno del cifrador. En este sentido se produce un alejamiento respecto de las arquitecturas mencionadas previamente ya que todas ellas tal y como se han presentado utilizan buses internos de *128* bits.

Básicamente este alejamiento implica una pérdida en cuanto a las características de velocidad de cifrado del circuito resultante. Afortunadamente la mejora se da en términos de área ocupada. En este sentido una configuración más equilibrada en término de ambos parámetros de eficiencia involucraría un camino interno de datos de menor cantidad de bits.

Al igual que en el caso de *128* bits, la posibilidad de elegir un bus interno de *64* bits impondría la necesidad de utilizar un bloque de memoria embebida superior a la que el dispositivo disponible es capaz de ofrecer.

El trabajo interno con *32* bits parecería tentador ya que una de las principales operaciones internas se realiza sobre una columna de la Matriz de Estado, pero aún persistirían los problemas de falta de memoria.

Por otro lado, en una configuración mínima [FIS00], un cifrador debería usar la menor cantidad de bloques de memoria posibles y una interfaz básica para comunicación con un sistema externo ó *host system*, como para permitir su adaptación a diversos propósitos. Un bus interno de *16* bits se presenta factible en estos términos, pero la unidad básica de procesamiento interno del algoritmo es de *8* bits [MUR01].

En el diseño presentado, se ha optado por un camino de datos interno de *8* bits ya que un *byte* es la unidad básica de datos en las operaciones internas y además representa el caso de una configuración mínima en cuanto a memoria utilizada y se sitúa holgadamente dentro de las posibilidades de implementación del dispositivo. En este sentido se puede decir que la arquitectura

seleccionada es una *arquitectura básica de 8 bits*. La elección representa una ocupación mínima de recursos y puede dar lugar a una implementación conjunta de cifrador y descifrador en el mismo *chip*.

Una vez definido el ancho en bits de la arquitectura, el siguiente problema es determinar el conjunto de operaciones que definirán las unidades funcionales. Un primer análisis del algoritmo identifica operaciones primarias que conducen al desarrollo de las unidades funcionales buscadas. La estrategia de diseño es jerárquica, en el sentido que se implementan bloques básicos para luego interconectarlos para el armado del cifrador.

En el núcleo de cifrado, se implementa una ronda solamente y el cifrador debe iterar *10* veces para la realización del cifrado de un bloque. Esta aproximación se conoce como *Iterative Looping* (*LU-1*) y se trata de un subconjunto de *Loop Unrolling-i* (*LU-i*) [ELB01] en la que sólo una ronda es desarrollada. De esta manera generalmente se logra minimizar el hardware requerido para la implementación, debiéndose poner especial esfuerzo en el diseño para poder maximizar el aspecto relacionado con la velocidad.

Se implementa entonces una única ronda mediante lógica combinacional suplementada con registros, memorias y multiplexores, debido fundamentalmente al hecho de tener que distinguir entre rondas con diferentes secuencias de operaciones.

La cuestión pendiente es la posibilidad de introducir cierto grado de paralelismo que permita mejorar el comportamiento de la arquitectura básica elegida como para obtener un nivel apropiado en cuanto a la capacidad de procesamiento por unidad de tiempo.

Por otro lado, se debe definir el modo en que operará el cifrador en cuestión. El modo de operación seleccionado es el más simple: *Electronic Codebook* (*ECB*). La elección de una arquitectura básica junto con un modo de operación sin realimentación (*non-feedback*) es fácil de implementar y mejora la probabilidad de obtener los resultados esperados en términos de área [WEA02]. También con este modo es posible aplicar paralelismo interno y de esta manera mejorar condiciones relacionadas con la velocidad de operación del cifrador.

5.5 Estrategia Elegida para las Operaciones Fundamentales en Rijndael

Habiendo definido una estructura básica, las decisiones próximas deben concentrarse en la partición del problema original en sus partes constitutivas características. Esta cuestión es fácilmente apreciable en el caso del cifrador Rijndael pues las transformaciones básicas que componen las diversas rondas se pueden traducir en elementos fácilmente separables y distinguibles desde un punto de vista circuital. Queda decidir respecto de las posibilidades relativas a su implementación.

5.5.1 SubBytes()

Como se explicó en el Capítulo previo, la operación *SubBytes* consiste en una función de sustitución sobre cada *byte* del bloque de entrada o Matriz de Estado *s*. La posición del *byte* en dicha matriz no es un aspecto relevante y es independiente, en cuanto a su transformación, de la aplicación de ésta operación sobre los demás *bytes*. Matemáticamente, la operación consiste de dos pasos [O´DR01]:

1. Inversión del byte en $GF(2^8)$. En este paso el *byte* en cuestión se toma como un elemento del campo.
2. Aplicación de la transformación afín. En este paso la representación más apropiada para un *byte* es la de un vector de 8 bits:

$$x \rightarrow Ax \oplus b \tag{5.1}$$

con

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{5.2}$$

$$b = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}^T \tag{5.3}$$

La diferencia en la representación en ambos casos genera la relación de alinealidad que se busca que presente cualquier caja *S* aplicable a un algoritmo criptográfico. Vale la pena recalcar que

la operación equivalente de desencriptado significa trabajar con la matriz inversa de *A*, A^{-1}, y un vector *c*, diferente del *b* pero obtenido a partir de él. Es decir que, en lo que respecta al paso *2*, existe una discrepancia entre las operaciones de encriptado y desencriptado, mientras que el paso *1* es similar en ambas, pues consiste en el cálculo del elemento inverso.

Paar [O´DR01] demostró que el cálculo de la inversa de un elemento perteneciente a un campo de la forma $GF(2^{2m})$ se reduce a una serie de operaciones en el subcampo $GF(2^{m})$: cálculo de la inversa, suma, multiplicación y cuadrado de dos elementos [RIJ03]. La complejidad de estas operaciones es dependiente de la elección de los polinomios irreducibles en ambos campos. Mastrovito [O´DR01] desarrolló arquitecturas para las operaciones de multiplicación y cuadrado. O'Driscoll determinó los parámetros óptimos para la implementación de la inversión.

Existe otra arquitectura posible para la implementación de la inversión y se trata de la realización de una *look-up table* (*LUT*) o tabla de búsqueda. Esta opción posee la ventaja adicional de incorporar la transformación afín sin ningún costo extra en términos de área o de complejidad [O´DR01], [MRO01], [KAR01], [FIS00], [KER02].

La caja *S* puede entonces ser almacenada en una sola *ROM* de *256* elementos de *8* bits. De esta manera, la implementación resulta más apropiada en términos de costo de recursos ya que la síntesis puede realizarse sobre las zonas específicas de memoria de la FPGA (EAB´s) sin consumir celdas lógicas. También se simplifica el esfuerzo de diseño ya que simplemente la memoria debe ser inicializada. La limitación más importante de esta selección se presenta en términos del tiempo de acceso.

En términos de re-uso de este componente, la caja *S* o memoria *ROM* válida para la operación de encriptado es diferente de aquella que corresponde a la operación de desencriptado. Esto significa que, en términos de la operación inversa, dicha memoria debe reemplazarse por otra de igual tamaño pero cargada inicialmente con diferentes datos.

Siguiendo las recomendaciones de Altera, se usó una *lpm_rom* para implementar la caja *S*, ya que además la misma se encuentra disponible sólo para dispositivos *FLEX 10K* en la herramienta utilizada. Altera también recomienda instanciar el componente como se describe en la ayuda para la creación de *Megafunciones* con el *MegaWizard Plug-In Manager*. Este componente de la herramienta simplifica el proceso de especificación de parámetros ó constantes y crea archivos de diseño que sirven para instanciar variaciones de una megafunción estandarizada, como es el caso de las funciones de librería de módulos parametrizados (*LPM*).

En este sentido se optó por una instanciación sencilla con tres *ports*. Un port de direcciones de 8 bits, un port de salida del mismo ancho y un port de reloj para lograr sincronismo entre el contenido de memoria direccionado y el dato presente en el bus de salida. Estas constituyen las vías de comunicación del componente con el resto del circuito.

De esta manera, la lectura de la *ROM* se realiza de modo sincrónico. La Fig. 5.1 representa, a modo de ejemplo, el funcionamiento de la memoria *ROM* implementada y su relación con los ciclos de reloj del circuito.

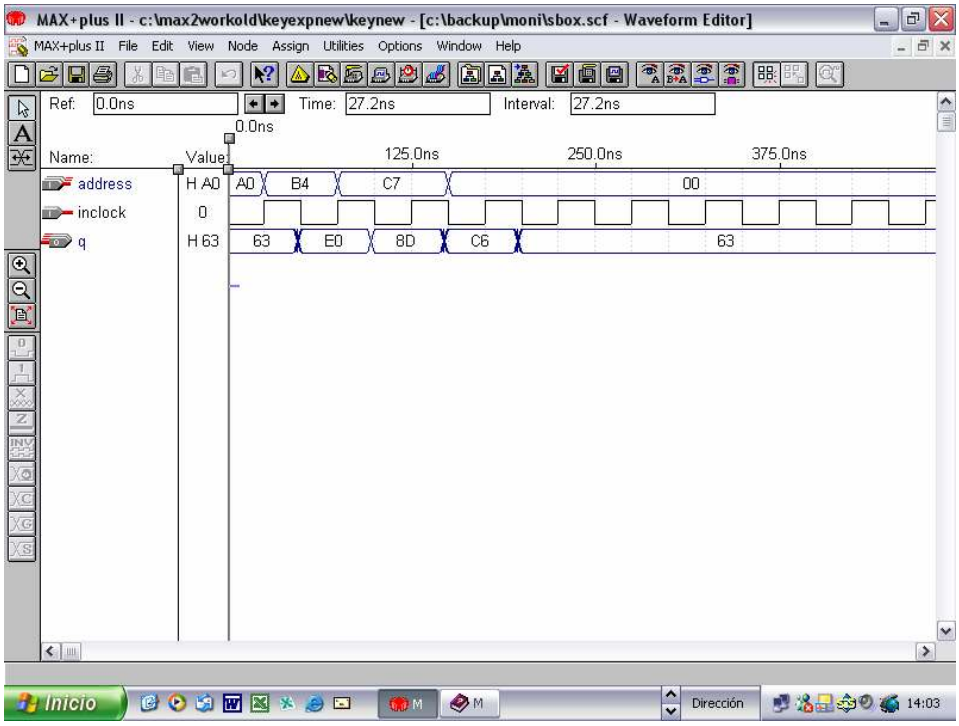

Figura 5.1. Caja S para Operación SubBytes

La caja se carga previamente con los datos correspondientes a la operación implementada que pueden verificarse consultando la Tabla 4.2. El funcionamiento es muy sencillo. Sobre el port de entrada a la memoria se coloca en el bus de *8* bits el byte correspondiente de la Matriz de Estado a transformar. Este dato constituye la dirección de almacenamiento en memoria y corresponde a los valores de fila y columna de la Tabla 4.2. El dato almacenado se presenta sobre el port de salida y corresponde al valor correspondiente al cruce de fila y columna de la tabla mencionada.

Es interesante observar en la Fig. 5.1 el tiempo que transcurre entre la presentación del dato, en este caso la dirección, y la obtención del resultado de la lectura de la memoria en el bus de salida. Medio ciclo de reloj es el retardo necesario para la obtención de un dato almacenado en memoria. Por otra parte se precisan 16 ciclos de reloj para obtener el resultado de la operación sobre la Matriz de Estado completa.

Por otra parte, al haber seleccionado un bus interno de *8* bits, sólo es necesaria una única caja *S*, la cual, en términos de memoria, representa *2048* bits ó un bloque *EAB* completo utilizado para el cifrado. La familia *Flex 10k* utilizada sólo posee *6* de estos bloques.

5.5.2 ShiftRows()

Según lo visto en el Capítulo previo, la operación en cuestión implica la rotación por filas de la matriz *State*. Muchas implementaciones de Rijndael optan por una implementación del tipo cableado que no tiene costo en términos de área ni de tiempo [SHI02], [HOD04]. La cuestión fundamental de esta elección se basa en que internamente trabajan con un bus de *128 bits*. En el caso de trabajar con un bus interno menor, la operación de rotación debe rediseñarse en términos de algún mecanismo de direccionamiento y recursos para almacenamiento temporal de los datos. La elección de un bus interno de *8 bits* limita la implementación hardware en términos del segundo caso planteado: almacenamiento y direccionamiento.

Otra cuestión importante de destacar es el orden en que se realiza esta operación. El cifrador, como hemos visto, coloca la operación *ShiftRows()* a continuación de *SubBytes()*, aunque nada impide que sea al revés ya que ambas son intercambiables y lo mismo se verifica para sus

operaciones inversas [DAE99]. Esta propiedad es importante pues puede condicionar la forma de implementar ambas operaciones. La elección de una u otra forma condicionará además la implementación del cifrador inverso o máquina de descifrado. El cifrador inverso común presenta las operaciones inversas de las mencionadas en orden inverso, esto es primero *InvShiftRows()* y luego *InvSubBytes().* La existencia de un cifrador inverso equivalente [DAE99], en donde el orden de ambas operaciones inversas puede ser exactamente al revés es importante en términos de re-uso de componentes, ya que el mismo bloque físico podría reemplazarse directamente por su equivalente inverso. Por este motivo, en esta implementación se eligió para el cifrador el orden habitual de las operaciones.

De este modo, la operación de rotación por filas se realiza por medio de dos componentes: una memoria *RAM* de *16 x 8* y un componente armado con lógica combinacional. La memoria *RAM* almacena temporalmente el resultado de la operación de rotación sobre el conjunto de *16 bytes* de la Matriz de Estado, *State*, sobre los que se trabaja en cada ronda. Se implementó con dos buses de entrada, uno para datos y otro para direccionamiento, el primero de *8 bits* y el segundo de *4 bits*, necesario para direccionar las *16* posiciones. Un bus de salida de *8 bits* permite la lectura de los datos almacenados.

El bus de direcciones es manejado por un circuito combinacional que se ha diseñado según la información que se presenta en la Tabla 5.1. Esta tabla representa la operación *ShiftRows()* en cuanto a la posición de cada *byte* en *State*. La fila *in(1..4)* de la Tabla representa la posición del *byte* previa a la rotación y *out(1..4)* la posición de ese mismo *byte* después de la misma. Ambas se expresan en formato hexadecimal.

in(1..4)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
out(1..4)	0	D	A	7	4	1	E	B	8	5	2	F	C	9	6	3

Tabla 5.1. Función de Circuito Combinacional de ShiftRows()

Se diseña según dicha Tabla un circuito de *4* entradas y *4* salidas que se aplican al bus de

direcciones de la *RAM* y cuyo diseño se realizó por simplificación de mapas de Karnaugh. El circuito resultante se presenta en la Fig. 5.2.

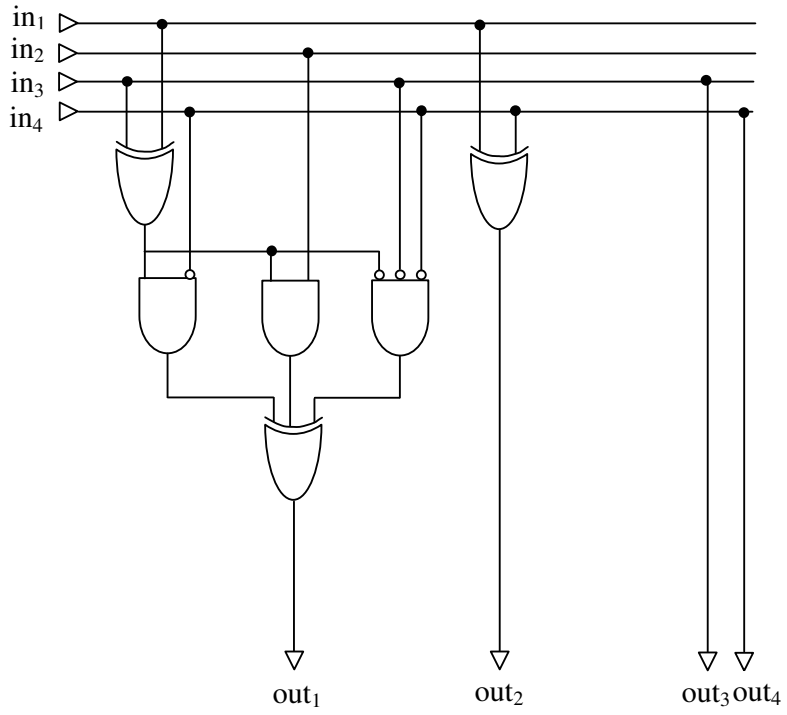

Figura 5.2. Circuito Combinacional para generación de ShiftRows()

El circuito de la Figura 5.2 toma los datos provenientes de una *Unidad de Control* y los transforma apropiadamente según se representa en la Tabla 5.1 en formato hexadecimal.

En consecuencia, el bus de datos de la memoria *RAM* recibe los *bytes* provenientes de la *caja S*, que se almacenan en la dirección apropiada de la *RAM* ya rotados según los datos presentes en su propio bus de direcciones, *out(1..4)* en la Fig. 5.2. Luego, esta memoria se podrá leer de manera secuencial, para lo cual ha de multiplexarse el bus de direcciones mencionado. La Fig. 5.3 presenta la simulación para la fase del cifrador en que los datos de salida de la *caja S* se graban sobre la *RAM*

Observar en la Fig. 5.3 que la escritura de la *RAM* se maneja a través de una señal de habilitación de escritura, *we*, en alto al momento de escribir la memoria. La señal *outclock* es

sencillamente la señal de reloj del sistema. Las señales *addressrom*, *monitordata* y *monitoraddress* representan:

- *addressrom* : los datos que ingresan a la *caja S*.
- *monitordata* : la salida de la *caja S*, que a su vez es la señal de entrada en el bus de datos de la *RAM*, es decir los *bytes* de *State* que se almacenarán rotados en memoria.
- *monitoraddress* : la señal presente en el bus de direcciones de la *RAM* que marcará la posición de almacenamiento en memoria.

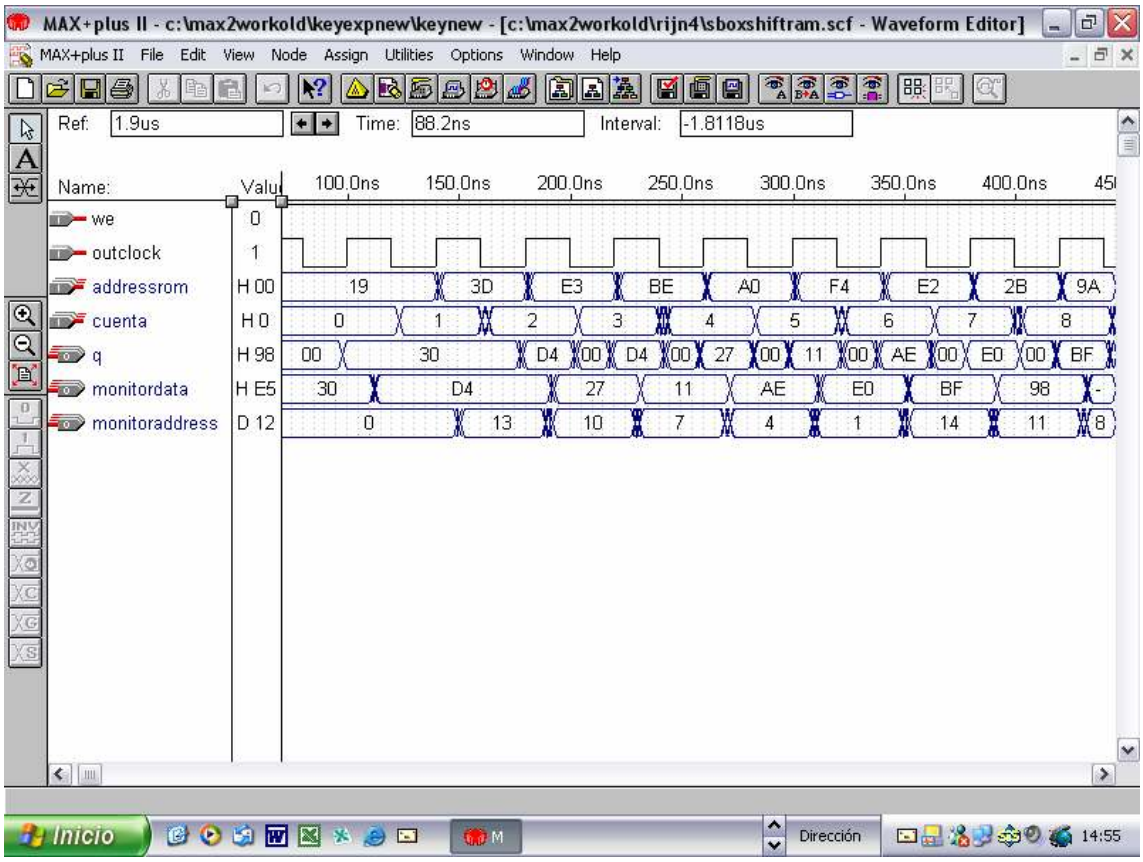

Figura 5.3. Operación ShiftRows(). Fase de Escritura.

Se puede observar en esta figura también, la demora de un ciclo en la lectura de la *caja S*, como así también el retardo introducido por el circuito combinacional de la Fig. 5.2, medido por la diferencia de tiempos entre las señales *cuenta* y *monitoraddress* que a su vez representan las dos filas de la Tabla 5.1.

También corresponde destacar que la operación que sigue a esta no puede realizarse hasta que *ShiftRows()* no haya sido realizada completamente. Esta situación, en el caso de un bus de *8* bits, dificulta la introducción de paralelismo interno.

5.5.3 MixColumns():

Como se ha visto anteriormente, esta operación consiste en una multiplicación de polinomios sobre GF(2^8), módulo $x^4 + 1$. Uno de los polinomios de la multiplicación se toma a partir de una columna de *State* y el otro es un polinomio fijo *a(x)*, presentado en la Ec.(4.10). Esta multiplicación tiene una representación matricial, como se ha desarrollado en la Ec.(4.11). Dicha ecuación puede re-escribirse de la siguiente forma:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} s_{0,c} + \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} s_{1,c} + \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} s_{2,c} + \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} s_{3,c} \tag{5.4}$$

Es de notar que las columnas de la Ec.(5.4) sólo presentan tres constantes: *01*, *02* y *03*, lo cual simplifica la operación pues se traduce en la multiplicación por potencias de *x* de bajo exponente.

A su vez, la Ec.(5.4) puede re-escribirse como cuatro ecuaciones:

$$s'_{0,c} = (0x02) \cdot s_{0,c} \oplus (0x03) \cdot s_{1,c} \oplus s_{2,c} \oplus s_{3,c} \tag{5.5}$$

$$s'_{1,c} = s_{0,c} \oplus (0x02) \cdot s_{1,c} \oplus (0x03) \cdot s_{2,c} \oplus s_{3,c} \tag{5.6}$$

$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (0x02) \cdot s_{2,c} \oplus (0x03) \cdot s_{3,c} \tag{5.7}$$

$$s'_{3,c} = (0x03) \cdot s_{0,c} \oplus s_{1,c} \oplus s_{2,c} \oplus (0x02) \cdot s_{3,c} \tag{5.8}$$

Puede observarse claramente que, para obtener un *byte* de cada columna de la Matriz de Estado como resultado de la transformación, es preciso contar con los *4 bytes* de la respectiva columna de entrada. Es decir que se trata de una operación simultánea sobre *32 bits* [ASH02].

La presencia de sólo dos constantes significativas en las multiplicaciones, *02* y *03*, permite aprovechar las propiedades mencionadas en el Capítulo 4 referidas a la operación *xtime()*. Desde el punto de vista de implementación en hardware, la operación mencionada se puede imaginar como el circuito representado en la Fig. 5.4.

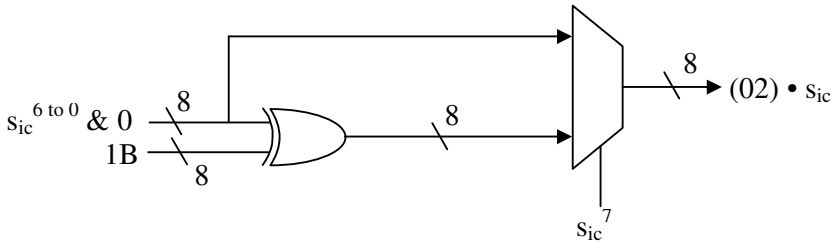

Figura 5.4. Circuito multiplicador por *(02)*. Funcionalidad *xtime()*

En el circuito de la Fig. 5.4 una compuerta *EXOR* realiza esta operación entre dos entradas: un *byte* de datos desplazado a la izquierda y un *byte* fijo, en hexadecimal la constante *0x1B*. El resultado de la multiplicación por *(0x02)* puede ser el *byte* de datos desplazado o la salida de la *EXOR*, según el valor del bit más significativo, s_{ic}^{7}, del *byte* de datos en cuestión. De esta manera con una función *EXOR* y un *multiplexor* de doble entrada se puede generar la operación *xtime()*. En líneas de código VHDL esto se traduce en una arquitectura muy sencilla.

El multiplicador mencionado sirve de base para la implementación de un multiplicador por *(0x03)*, como se representa en la Fig. 5.5. El componente surge de la descomposición de esta constante en la suma $0x01 \oplus 0x02 = 0x03$ y la aplicación de la propiedad distributiva de la multiplicación respecto de la suma.

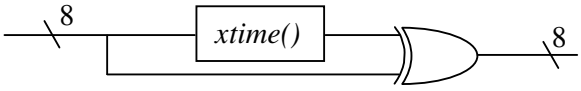

Figura 5.4. Circuito multiplicador por *(03)*. Funcionalidad *x3*.

A partir de estos dos componentes se puede generar el circuito básico para la multiplicación. Asi, el circuito que implementa la Ec.(5.5) queda representado por el circuito de la Fig. 5.5.

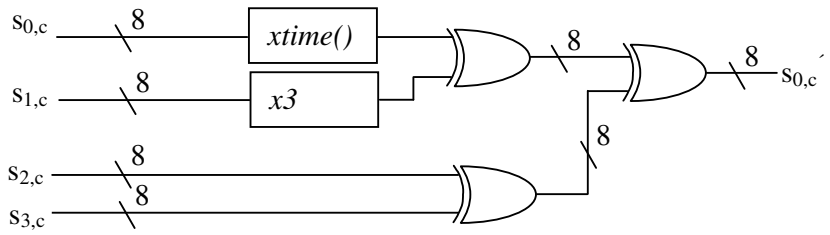

Figura 5.5. Obtención del primer *byte* de operación de multiplicación. Componente *multcolumn_4*.

Es decir que cuatro componentes *multcolumn_4*, como el representado en la Fig. 5.5, son necesarios para generar la multiplicación de una columna. El cableado debe adecuarse a las Ec.(5.5) a (5.8) como se indica en la Fig. 5.6. Se denomina al conjunto componente *mixcolumn_4*.

Todavía falta resolver una cuestión. Para generar un único *byte* de salida se precisa tener presente los cuatro *bytes* de la columna respectiva. Por este motivo es necesario, a partir del bus interno de *8 bits*, almacenar *4 bytes* consecutivos de *State*, y así presentarlos al circuito generador de la operación *MixColumns()*, componente *mixcolumn_4*. De la misma manera, luego de la operación se ha de regresar a la estructura de bus interno de *8 bits*.

Para lograr la presentación coordinada de los datos en el caso de la presentación de la columna al componente, se utiliza un registro de 32 bits de entrada serie y salida paralelo como el que se presenta en la Fig. 5.7.

El registro de salida tendrá una funcionalidad inversa para poder regresar al bus interno de *8* bits. El almacenamiento temporal produce una penalización de 8 ciclos de reloj para la obtención del primer *byte* de datos a la salida.

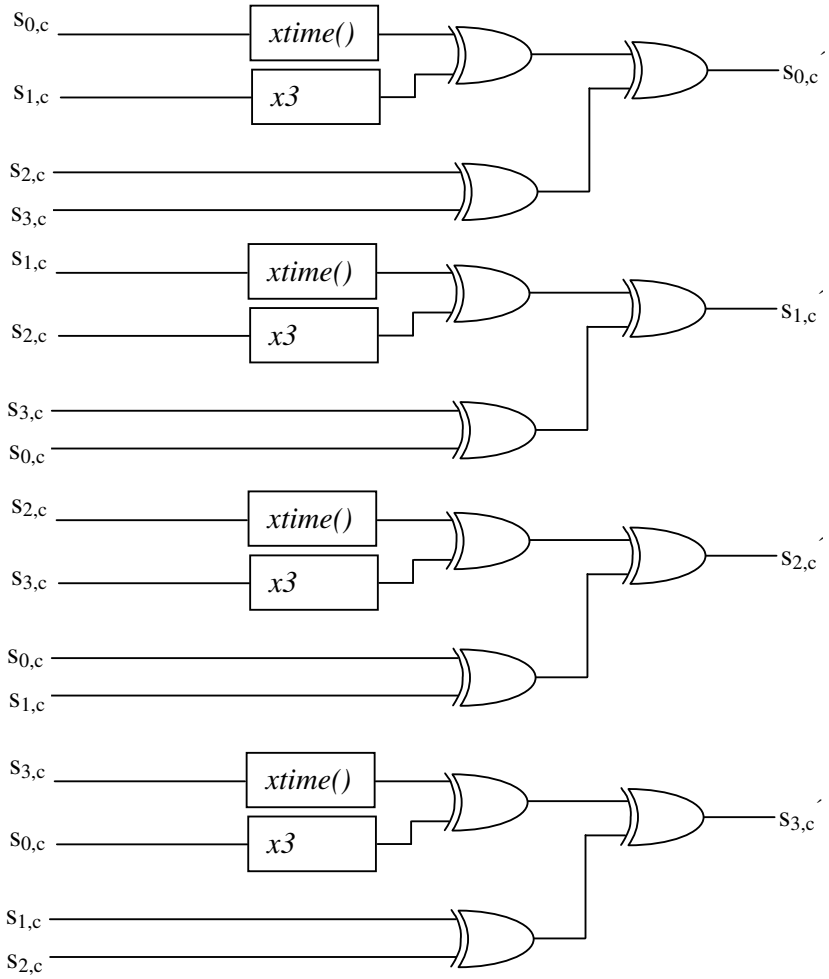

Figura 5.6. Obtención de una columna de operación de multiplicación. Componente *mixcolumn_4.*

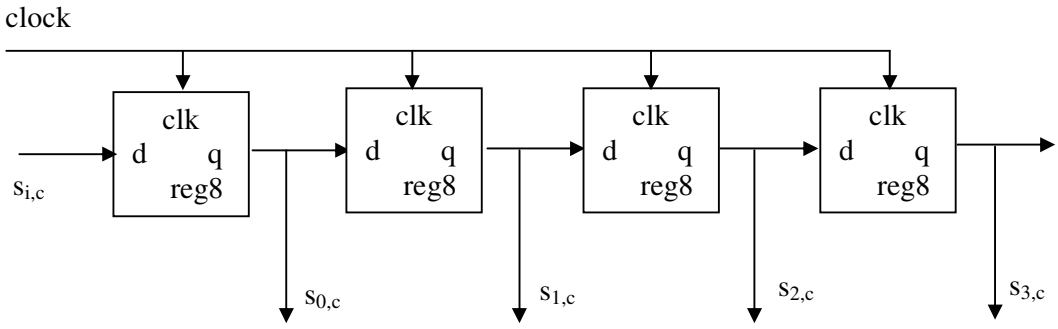

Figura 5.7. Registro de Desplazamiento para presentación de datos. Componente *reg32*.

El funcionamiento del circuito en su fase inicial ha sido simulado y el resultado se presenta en la Fig. 5.8. En esta figura, las señales *bytein* y *byteout* son simplemente la representación consecutiva de la Matriz de Estado *State* antes y después de la aplicación de *MixColumns()*. Las señales *state40* a *state43* en su conjunto representan la entrada de una columna de *State* al circuito. Cada una de ellas por separado ingresa a un circuito como el de la Fig. 5.5.

En la Fig. 5.8 también puede observarse el retardo inicial en la obtención de un byte de la operación. Una vez superado el mismo, los *bytes* del resultado se presentan a razón de uno por ciclo de reloj.

El trabajo coordinado de todos los componentes necesarios para realizar esta transformación depende de señales de control sincrónicas cuyo funcionamiento se explicará en los capítulos siguientes.

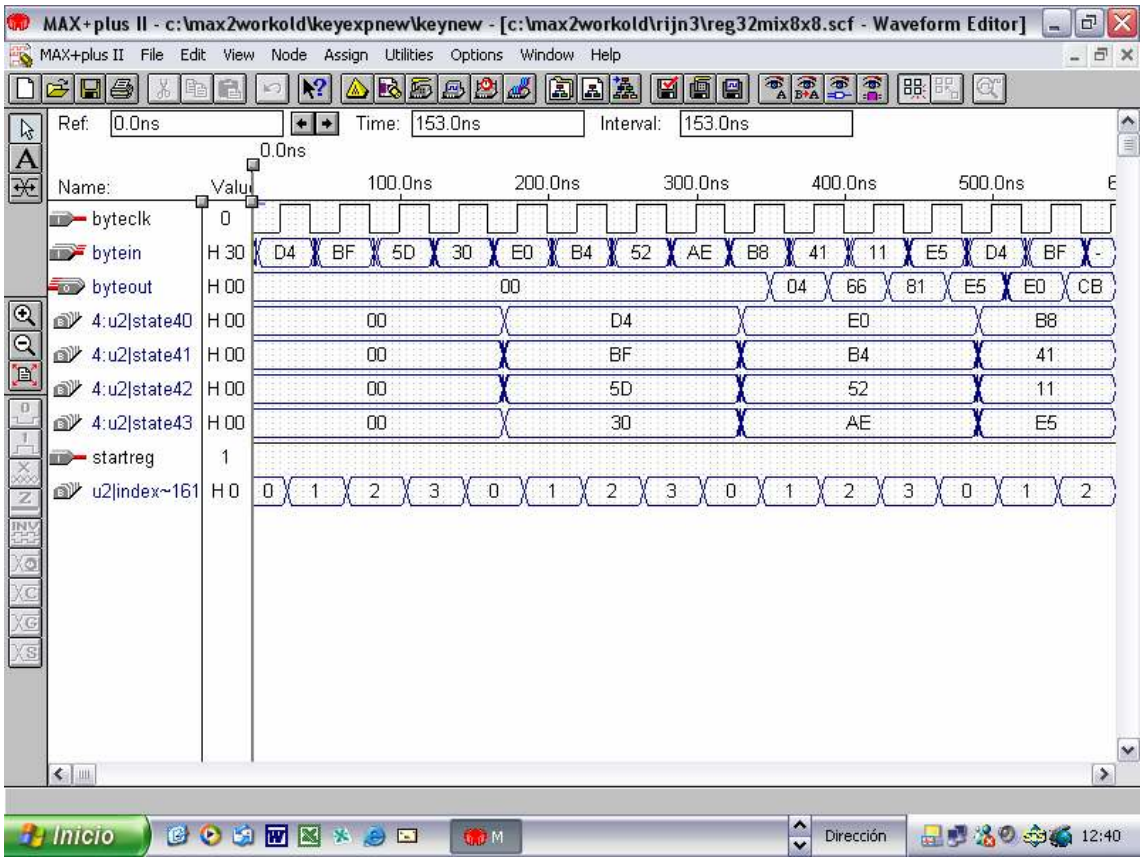

Figura 5.8. Operación *MixColumns()*.

5.5.4 AddRoundKey():

Esta operación es muy sencilla y consiste en la adición tipo *XOR* de los *128 bits* de la subclave de una ronda con los *128 bits* de *State*. Una implementación con bus interno de *8 bits* precisará *16* ciclos de *clock* para poder completarse. Una *EXOR* con dos buses de entrada de *8 bits* cada uno es una función muy sencilla de realizar en *VHDL*.

Por sí sola esta operación implica el desarrollo de una ronda inicial pero luego está presente en todas las demás rondas. No es necesario en ninguna de las rondas esperar la finalización de la operación *EXOR* sobre la Matriz de Estado completa puesto que la operación que, en todos los casos, le sigue a esta, *SubBytes()*, se realiza a nivel de bytes como ya se ha visto. Este hecho permite la realización de ambas prácticamente en simultáneo.

Capítulo 6

Descripción del Cifrador

Este capítulo desarrolla una explicación detallada del diseño completo para la implementación en hardware del cifrador *AES-128* sobre una *FPGA* de la Familia *Flex 10k* de *Altera*. La utilización del lenguaje *VHDL* permitiría trasladar este diseño sobre un chip *FPGA* de otra compañía de manera casi inmediata. La excepción en este sentido la constituyen aquellos componentes diseñados con ayuda de la *Biblioteca de Módulos Parametrizados* (*LPM*) que ofrece posibilidades para diseño independiente de la arquitectura sólo para aquellos dispositivos que soporta la herramienta utilizada, *MAX+Plus II*, es decir los fabricados por Altera.

El objetivo inicial de ocupación mínima en términos de área se tradujo en un circuito cuyos componentes son los necesarios para la implementación de una única ronda del cifrador. Un componente generador de las señales de control internas del dispositivo produce la iteración sobre las *10* rondas que conforman la realización del cifrado de un bloque de datos de *128* bits. Este objetivo se logra además mediante la realimentación apropiada del circuito por medio de multiplexores. La separación entre rondas se realiza mediante una memoria que actúa a modo de registro para almacenar los datos provenientes de la ronda anterior y dejarlos disponibles para su procesamiento en la ronda siguiente.

Se trata de una arquitectura de lazo de tipo iterativo, conocida como *Iterative Looping*, en la que sólo una ronda es desarrollada, *Loop Unrolling-1* (*LU-1*). La implementación de una ronda se logra mediante el empleo componentes de lógica combinacional suplementada con registros, memorias y multiplexores. El motivo fundamental del empleo de estos componentes es el cumplimiento del funcionamiento iterativo pero tampoco puede desconocerse que la diferencia entre las diversas rondas, característica intrínseca del cifrador Rijndael, implica la utilización de componentes adicionales.

6.1 Descripción General

El dibujo de la Fig. 6.1 es una representación esquemática del cifrador. Se puede observar en dicho gráfico los componentes principales del circuito. En términos de memoria, el circuito presenta un bloque de memoria *ROM* de *2048* bits y cuatro bloques *RAM* de *2432* bits. Es decir que el diseño de este cifrador implica un consumo de *4480* bits de memoria total. Entre los demás componentes se destaca una serie de multiplexores: uno de *3* entradas de *8* bits, otro de dos entradas de *8* bits y dos de dos entradas de *4* bits. En general estos multiplexores manejan buses de entrada de datos o de direcciones de los bloques de memoria. El circuito también presenta una interfaz de comunicación para el manejo de datos, un bloque específico para la realización de la transformación *MixColumns()*, un bloque combinacional que interviene en la operación de rotación de las filas de la Matriz de Estado y un par de componentes para ajuste de sincronismo en la presentación los datos sobre las diferentes memorias que conforman las operaciones de una ronda. Además de los mencionados, existe un componente que fiscaliza y sincroniza la operación del cifrador completo, realizando toda la funcionalidad de control necesaria para el funcionamiento correcto del circuito.

Las unidades de memoria *RAM* son cuatro y se trata de los componentes etiquetados como U_1, U_3, U_4 y U_{10} en la Fig. 6.1. En términos generales se las identifica como *ram16x8* y *ramkey*. La denominación *ram16x8* alude a la cuestión de la capacidad total de dichas memorias, organizadas como bloques de *16* palabras de *8* bits cada una, es decir *128* bits en total. Estas memorias se utilizan a modo de registro, ya sea para almacenamiento temporal de los datos de entrada del bloque de datos a ser cifrado, como es el caso de U_1, como así también de los resultados intermedios en el caso de U_{10}, y del resultado final en U_4. Por consiguiente actúan como registro temporal de las diversas transformaciones de la Matriz de Estado.

Por otro lado, el componente U_3 ó *ramkey* es una memoria también de tipo *RAM* pero de mayor capacidad que las anteriores. Se estructura internamente como un banco de *256* palabras de *8* bits, *256 x 8*, y se utiliza para almacenar las claves y subclaves necesarias para el desarrollo de las *10* rondas de cifrado. Esta memoria debe ser cargada antes de comenzar la operación de cifrado. Los primeros *16* bytes almacenados constituyen la clave original, a partir de la cual se generan *10* claves,

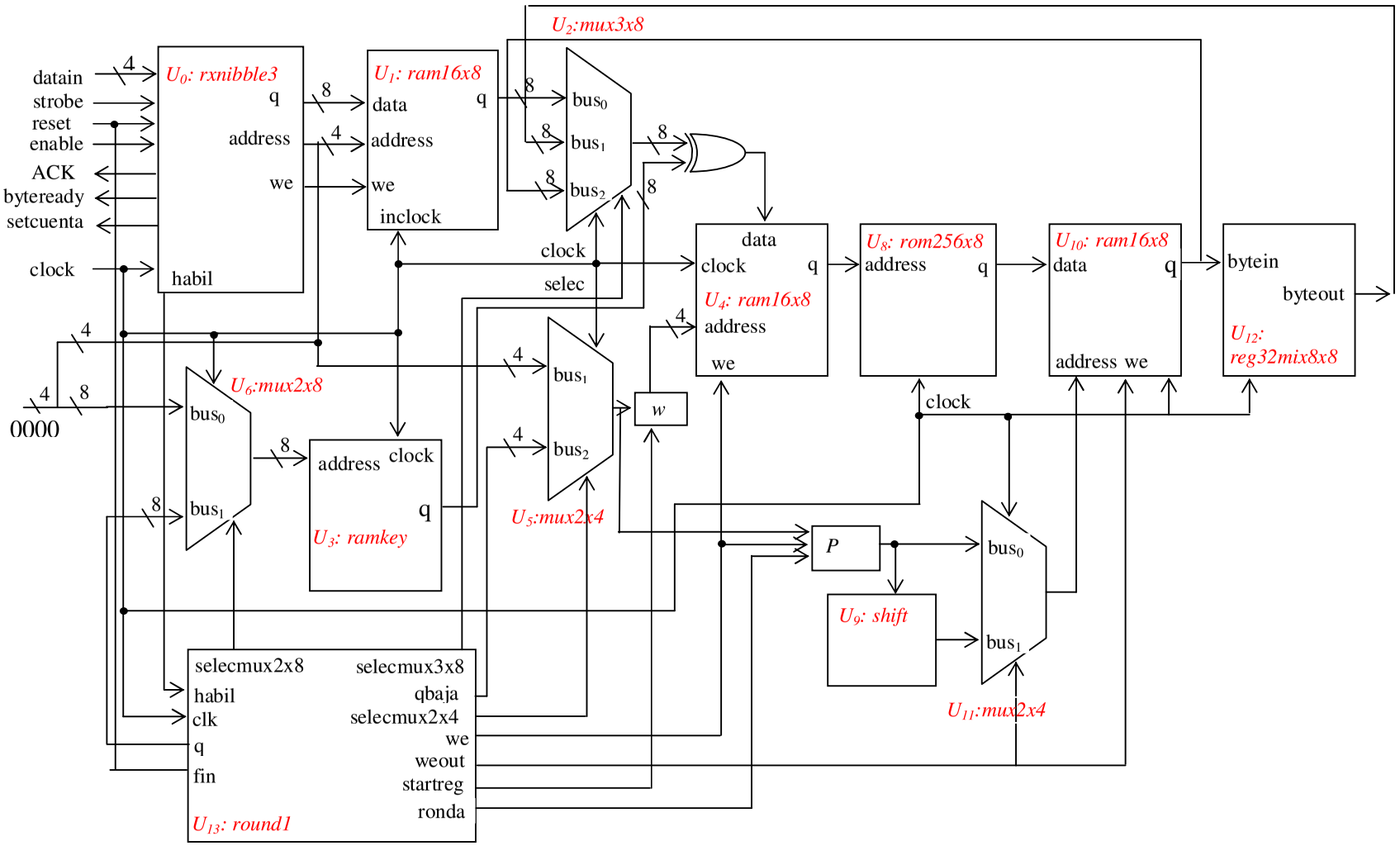

Figura 6.1. Diagrama en Bloques del Cifrador Implementado.

conocidas también como subclaves, de *16* bytes cada una. Es decir que la memoria para almacenamiento de claves y subclaves debe tener capacidad para almacenar al menos *176* bytes. En este sentido, la opción de diseño no ha sido el cálculo de la clave a medida que se desarrolla el algoritmo, en inglés *scheduling on the fly*, y esta decisión obedece al objetivo original de minimización de área.

Por su parte, la memoria *ROM* de *256x8*, componente U_8 ó *rom256x8* de la Fig. 6.1, sirve a los propósitos de implementación de la caja *S*, por haberse optado para dicha transformación el método de tabla de búsqueda.

Los componente U_{12} y U_9, también denominados *reg32mix8x8* y *shift* respectivamente, realizan las transformaciones *MixColumns()* y *ShiftRows()*. La segunda operación se realiza con el soporte del componente de memoria U_{10}.

La coordinación entre todos los componentes del cifrador se opera desde un único componente U_{13}, denominado *round1*. El componente U_0 ó *rxnibble3* es la interfaz del cifrador con el mundo exterior para la carga de datos. El cifrador se completa con algunos componentes adicionales en general con funcionalidad de multiplexores.

De este modo se puede distinguir en la estructura de la implementación una organización básica común a la implementación de cifradores en bloque simétricos [GAJ01]:

- Un *núcleo de cifrado*, en inglés *core*, para realizar las operaciones matemáticas necesarias, que en este caso lo componen U_1, U_2, U_4, U_5, U_6, U_8, U_9, U_{10}, U_{11} y U_{12}.
- Una unidad *de memoria interna* para almacenar la clave y las subclaves, U_3.
- Una *interfaz* de entrada para poder realizar la carga de bloques de datos y de la clave y subclaves, U_0.
- Una *memoria de almacenamiento* de los datos cifrados, U_4.
- Una *unidad de control* que genera las señales de control para el manejo del resto de los componentes, U_{13}.

Aparte de la modalidad iterativa, se ha elegido para este diseño el modo de operación de cifrador sin realimentación pues dicha elección representa seguridad en términos de menor ocupación de área [GAJ01]. Esta opción también ofrece la ventaja de permitir paralelismo externo, es decir permitiría replicar el diseño sobre algún integrado FPGA con mayor capacidad para poder realizar cifrado de diferentes bloques en paralelo. Este modo no realimentado por ejemplo se utiliza en muchos protocolos de autenticación que utilizan algoritmos de cifrado para el cifrado inicial de claves de sesión, previo al intercambio de las mismas.

6.2 Interfaz de Ingreso de Datos

El componente que oficia de interfaz entre el circuito y el anfitrión ó *host* que provea los datos a cifrar es *rxnible3*, U_0. Como el puerto paralelo es uno de los más comúnmente usados y más sencillos en cuanto a funcionamiento, se pensó la interfaz como aquella capaz de enfrentar un puerto de estas características.

La nueva estandarización del puerto paralelo fue publicada en 1994, bajo la denominación *IEEE 1284*. Este estándar define cinco modos de operación, de los cuales tres coinciden con las versiones de hardware estandarizadas originalmente y dos precisan hardware adicional para poder manejar mayores velocidades [PEA98]. Los tres modos compatibles se denominan *Compatibility*, *Nibble* y *Byte*. El primero sólo puede transmitir datos desde la PC al dispositivo en cuestión y, para poder recibir información en sentido inverso, se debe cambiar a modo *Nibble* ó *Byte*. Este último modo se encuentra en muy pocas placas.

El modo *Nibble* es el modo más usado para leer datos en el puerto paralelo y generalmente se combina con el modo *Compatibility* cuando se desea crear un canal bidireccional. Como todos los puertos paralelos estándar proveen cinco líneas para que los periféricos comuniquen su estado al anfitrión, se puede transmitir un byte de datos en esta dirección si el envío se divide en *2 nibbles*, cada uno de *4* bits. De este modo, un *nibble* se constituye en la unidad básica de transferencia sobre este tipo de interfaces. Su selección se debe a que *4* es el número de dígitos binarios necesarios para representar un elemento de la base hexadecimal de numeración. Los *nibbles* también son la base del

sistema de codificación *BCD*, *binary coded decimal*.

Bajo las consideraciones mencionadas, la interfaz fue concebida como una máquina de estados finitos, con un *bus* de datos de entrada de *4 bits*. Es importante ser capaz de describir este tipo de componentes de una manera que pueda ser sintetizable. El estilo seguido con este propósito separa la implementación en dos procesos, uno que describe la lógica combinacional para el cálculo del siguiente estado y los valores de las salidas, y otro que oficia de registro que almacena el estado de la máquina.

Por otro lado, para evitar problemas de estabilidad producidos por situaciones de carrera crítica, se provee al componente de una señal de sincronismo, de manera que la transición entre estados se produzca en los flancos de una señal de *reloj*. Como es de práctica normal, se ha dotado a la máquina de una señal de *reset* asincrónico que permita colocar la misma en un estado de partida conocido en cualquier momento. También se le han añadido señales para comunicación con el mundo exterior.

La interfaz se ha completado con el componente U_1. Se trata de una memoria *RAM* de *128* bits estructurados en *16* palabras de *8* bits, que oficia de registro de entrada de datos para el bloque a procesar y es accesible vía un bus de direcciones de *4 bits* cuyo control queda en manos de la propia interfaz de entrada.

El diagrama de estados del componente se representa en la Fig. 6.2. Se puede observar en dicha figura los diversos estados de transición que atraviesa la máquina de estados durante el traspaso de un byte de datos desde el exterior hacia el núcleo del cifrador.

Se trata de una máquina que evoluciona a lo largo de *10* estados, aunque en realidad, existen estados sin funcionalidad para permitir que las señales intercambiadas tengan la duración mínima requerida por el estándar. Se han creado de este modo, sólo para propósitos de la simulación, estados ¨virtuales¨ en los que la máquina carece de funcionalidad y sólo existen para acercar el funcionamiento de la misma a un funcionamiento en tiempo real. Aún así se ha simulado con una

velocidad de transferencia de datos muy superior a la que manejaría el más común de los *ports* paralelo.

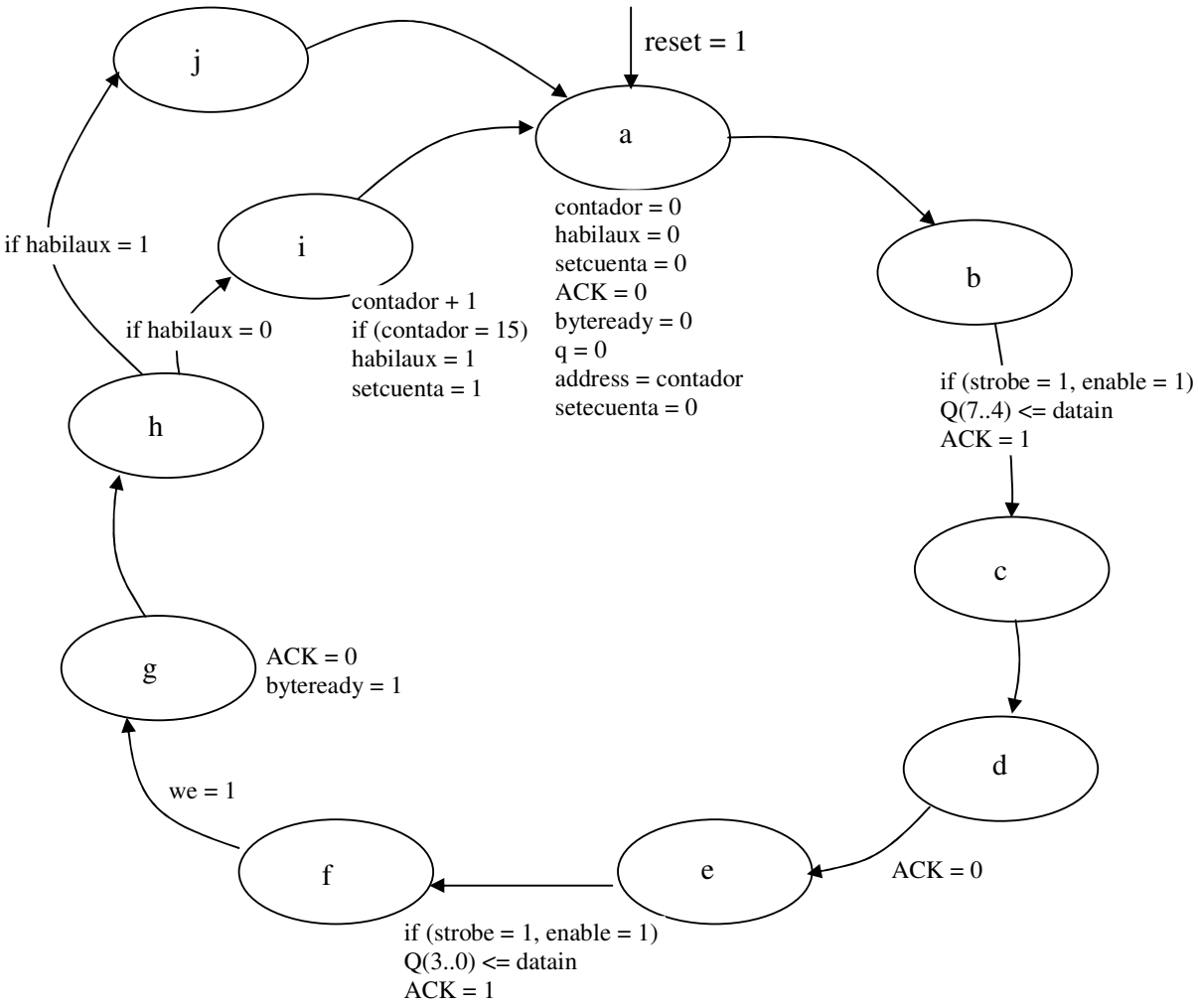

Figura 6.2. Diagrama de Estados de la Interfaz de Carga de Datos.

En el diagrama de la Fig. 6.2, la señal denominada *habilaux* es interna al circuito y se encuentra conectada al puerto de salida *habil* del componente U_0 de la Fig. 6.1.

En un estado inicial *a*, la interfaz inicializa sus puertos de salida *ACK*, *bytready* y *setecuenta* a nivel lógico *0*. En este estado también se inicializan el bus de salida de datos, *q* de *8* bits, y el de direcciones, *address* de *4* bits. Estos *buses* manejan los de entrada de igual nombre de la *RAM* U_1 que registrará el bloque de datos a cifrar. La máquina traspasa, mediante dos *nibbles*, cada byte de

entrada al bus *q* y posee un contador interno que genera las direcciones de la *RAM* donde dichos *bytes* serán almacenados. Cada *nibble* de entrada es un número codificado en hexadecimal que se presenta en el puerto de datos de entrada de *4* bits, *datain*.

La evolución de la máquina se puede seguir también a través de diagrama de tiempos que ofrece la herramienta de simulación, como se puede observarse en la Fig. 6.3.

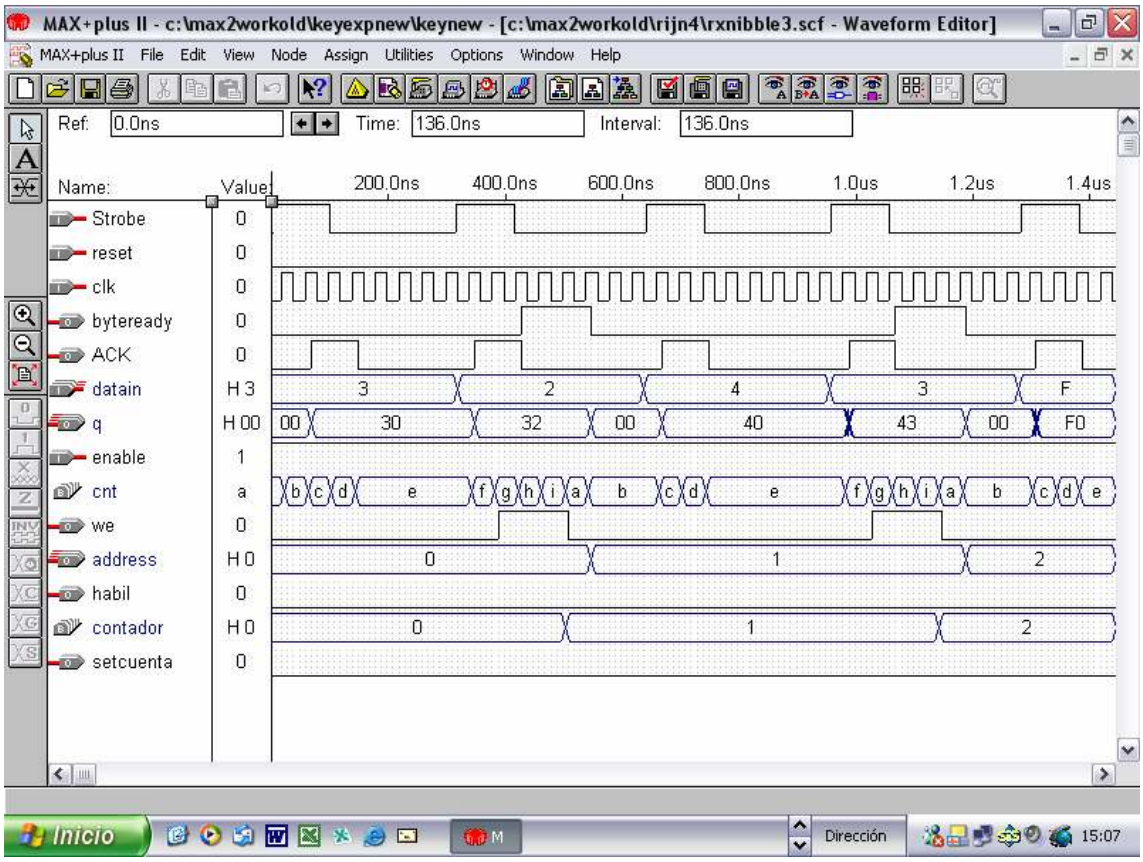

Figura 6.3. Simulación del comportamiento del circuito en el ingreso de datos.

La Fig. 6.3 presenta los datos codificados en formato hexadecimal. Por cada pulso de entrada de *Strobe*, indicando la presencia de un *nibble* en el *bus* de datos de entrada, se traspasa dicho *nibble* a la parte alta o baja del *bus* de salida *q* según se trate de la primera o la segunda mitad de un *byte*. La llegada del un *nibble* se avisa activando la señal de *ACK*. La duración de esta señal de reconocimiento debe tener un mínimo de tiempo para ser reconocida por el otro extremo de la

comunicación, de ahí la existencia de los estados virtuales, *c* y *f*. El traspaso del *nibble* al bus de salida se indica por la recuperación del nivel bajo en la misma señal.

Cuando un *byte* se completa sobre el bus de salida, la interfaz genera un pulso en la señal *we*, con el objeto de habilitar la escritura de dicho *byte* en la *RAM* U_1. También se transmite otro pulso sobre el puerto de salida *byteready*, como señal de aviso al *host* de esta situación. Se trata de una indicación similar a una señal de ocupado. Por otra parte la señal del bus *address* se genera desde un contador interno a la propia interfaz que se encarga de la numeración de los bytes ingresados y coincide con la posición que cada *byte* ocupará en memoria. Cuando este contador haya recorrido el rango completo desde *0* hasta *15* levantará la señal *setcuenta* hacia el *host*, indicando el inicio de proceso de cifrado. También esta situación se comunicará a la unidad de control del cifrador, *round1*, pues a partir de este momento será este componente el que se haga cargo de la situación en el sentido de la coordinación de las diversas operaciones.

Al finalizar el procesamiento completo de este primer bloque de *16* bytes, el cifrador, desde el componente de control, generará sobre la interfaz una señal de *reset* para preparar la recepción del próximo bloque de datos a cifrar.

6.3 El Núcleo del Cifrador y el Procesamiento Inicial

Como se ha explicado en capítulos anteriores, existe una primera ronda de cifrado en la que solamente se produce una operación sobre los datos. Se trata de la adición *EXOR bit* a *bit* del bloque de datos original con la clave seleccionada por el usuario, ambos de *128 bits*. Se ha optado en esta implementación por realizar esta operación en tiempo real, es decir, a medida que el bloque de datos ingresa al cifrador se los suma con la clave original almacenada en *RAM*, al ritmo o velocidad de ingreso de los mismos.

En la concreción de esta primer ronda aparecen implicados varios componentes: la interfaz de datos descripta anteriormente, tres memorias *RAM* que almacenan el bloque inicial, la clave y el resultado de la primer ronda, una *EXOR* de dos entradas de *8 bits*, tres multiplexores que acomodan

los buses de entrada y direcciones apropiados sobre las *RAM´s* mencionadas y la unidad de control que maneja la habilitación de los multiplexores y de lectura/escritura de las memorias.

En esta primer ronda el componente *rxnibble3* toma los datos de entrada de a *4 bits* y, una vez que obtiene un *byte* levanta la señal de escritura de la *RAM* U_1. El ritmo de escritura de esta *RAM* se maneja con el contador interno de la interfaz y coincide con el ritmo de lectura de los datos. Este ritmo de ingreso de datos se supone apenas superior a los *10 Mbps* en lo que respecta a las simulaciones, teniendo de este modo en cuenta la posibilidad de interacción con la más elemental tarjeta de red cableada o inalámbrica. Este detalle no es menor ya que restringe la velocidad de la primera ronda, determinando que finalice en un tiempo superior a los *10µs*. Con fines comparativos, se resalta el hecho que este consumo de tiempo resulta más de *5* veces superior a los *1,92 µs* que resultan de procesamiento en cada una de las rondas subsiguientes.

Durante el período mencionado se realiza la *EXOR* entre los datos de entrada y la clave original, a la vez que el resultado se almacena en la memoria *RAM* U_4 de *128* bits, a la espera de su próximo procesamiento.

Como toda la operación se hace en tiempo real, en esta primera ronda los multiplexores ofrecen los mismos datos a los *buses* de direcciones de las 3 *RAM*, es decir se barren las posiciones de memoria en una sucesión de *0* a *15*. Por consiguiente, el *bus* de direcciones de *rxnibble3* se presenta sobre el bus de direcciones de la *RAM* que almacena la clave, *ramkey*, a través del multiplexor U_6. La diferencia de *4 bits* en ambos buses se subsana generando un *nibble* ficticio de *4* ceros que se acoplan al *nibble* verdadero en una sencilla operación que permite el lenguaje *VHDL*. Se trata de la operación de concatenación, representada por el operador &. De esta forma se generan las primeras 16 de direcciones de la *RAM* U_3, posiciones donde se encuentra almacenada la clave original.

Cabe aclarar que el bus de direcciones de la *ramkey* se encuentra multiplexado puesto que, por un lado, los datos almacenados por dicha *RAM* U_3 serán leídos a la velocidad de procesamiento del cifrador en las rondas subsiguientes, situación distinta a la que corresponde a la primera ronda.

Por otro lado, la lectura de esta memoria es por bloques de *16 bytes*, constituyendo cada bloque una subclave para la ronda correspondiente. Por tanto, la generación de la secuencia particular de direcciones para cada ronda subsiguiente es tarea de la *unidad de control*. En la primera ronda dicha unidad sólo se encarga de proporcionar la señal adecuada al multiplexor U_6 para que éste ofrezca al bus de direcciones de la *RAM* U_3 el contenido presente en el bus superior de dicho componente.

También el bus de direcciones de la *RAM* U_4 se encuentra multiplexado y las consideraciones aplicadas anteriormente sobre distintas velocidades entre la primera ronda y las demás valen también para este caso, sobre todo por que este componente almacenará el resultado parcial del procesamiento de cada ronda de cifrado, como así también el resultado final. La selección del bus apropiado lo realiza la *unidad de control* por medio de la habilitación del multiplexor U_5. En la primera ronda el bus de direcciones de esta memoria proviene directamente de *rxnibble3*. En las rondas siguientes un bus de *4 bits* se genera internamente con ayuda de la *unidad de control*. El componente U_5 se agrega con este propósito y la unidad de control no sólo genera la señal de selector de bus sino también, durante la primera ronda mantiene la señal de escritura de la *RAM* U_4 habilitada para que los datos resultantes de la operación *EXOR* puedan ser almacenados en la misma.

6.4 El Núcleo del Cifrador y las Nueve Rondas Intermedias

El comienzo de las nueve rondas subsiguientes se identifica con la habilitación de la señal de lectura de la *RAM* U_4. Tanto en estas nueve rondas como en la ronda final, la velocidad de procesamiento la gobierna el reloj del sistema. En este diseño se eligió un reloj de *25 MHz* en la realización de las simulaciones por tratarse de la frecuencia del oscilador *on-board* del dispositivo *EPF10K20*, sobre el que se apuntó la síntesis de la implementación [ALT98].

A este ritmo se realiza la lectura mencionada previamente, a razón de un byte por ciclo. Existe una penalidad de retardo sobre el byte inicial de *50 ns*, algo más de un ciclo, propio de la operación de lectura de la memoria.

Cada uno de los bytes que se leen de U_4 se presenta sobre el bus de direcciones de U_8. El dispositivo U_8 es la memoria *ROM* que almacena los datos necesarios para la operación *SubBytes()*. Por lo tanto, cada byte de entrada direcciona su propio resultado de la transformación, obteniéndose un nuevo byte a la salida, cuya relación con el anterior es la representada en la Tabla 4.2. Entre la presentación del *byte* original y la obtención del transformado existe un retardo, propio de la lectura de la *ROM*, de un poco más de medio ciclo de reloj.

Una vez presente en el bus de salida de la *ROM* U_8 el primer byte válido resultante de la transformación *SubBytes()* debe aplicarse sobre el conjunto de *16* bytes de la Matriz de Estado, la operación de *ShiftRows()*. Esto se realiza escribiendo la memoria U_{10} en las direcciones explicitadas en la segunda fila de la Tabla 5.1. La sucesión de columnas de dicha fila se presenta al bus de direcciones de U_{10} por habilitación del bus correspondiente del multiplexor U_{11} y su generación se ofrece a través del componente *shift*, U_9. La presencia del multiplexor sobre dicho bus obedece al hecho de que la memoria se escribe en el orden indicado de la Tabla, pero se lee de manera secuencial ordenada, es decir desde la posición *0* avanzando hasta la posición *15*. El componente generador de esta última secuencia ordenada es la *unidad de control*. A través del puerto de salida *qbaja* del componente U_{13}, se obtiene esta secuencia en el momento de la lectura. Debido a que este puerto también produce el direccionamiento de la *RAM* U_4, el bus de direcciones de U_{10} se obtiene a través de dos multiplexores, U_5 y U_{11}. El retardo adicional generado por dichos componentes se compensa con la funcionalidad de *P* y *w* de la Figura 6.1.

El componente *P* es en realidad un proceso independiente dentro del cifrador que ajusta, de acuerdo al número de ronda, el bus de entrada de 4 bits al componente *shift*. De esta manera se distingue entre rondas con distinto procesamiento y, por ende, con diferente duración. Por su parte, el bloque *w* ajusta el bus de direcciones de la *RAM* U_4 según que se haya habilitado o no la transformación *MixColumns()*, debido al retardo inherente a la misma en cuanto a la conversión de bus serie de *8* bits a paralelo de *32* bits y su conversión inversa.

Por otra parte, la escritura rotada primero y la lectura ordenada posterior de la *RAM* U_{10}, se realiza con la habilitación o deshabilitación de la señal *we* de la misma, funcionalidad a cargo de la

unidad de control, *round1*.

Una vez finalizada la escritura rotada de *16* ciclos de duración, se habilita la lectura ordenada o secuencial de la memoria U_{10}. Esta última operación denota el inicio de la transformación *MixColumn()* a cargo del componente U_{12}. Vale la pena recordar que esta operación se realiza sobre 4 bytes de entrada que generan otros tantos de salida. Por este motivo, la carga del circuito consume cuatro ciclos, tras el paso de los cuales la *unidad de control* habilita el funcionamiento de U_{12} a través de la señal *startreg*. Durante estos cuatro ciclos un registro interno del componente ha almacenado los primeros *4 bytes* a procesar. La transformación en sí consume un ciclo y finalmente se necesitan otros *4* ciclos para retornar al bus serie de *8 bits*. Esta es la razón de la presencia de un retardo de *9* ciclos entre la presencia del primer byte válido proveniente de U_{10} a la entrada de U_{12} y la salida del primer byte resultante de la transformación *MixColumn()* a la salida de este último componente.

Una vez que se dispone del primer byte del resultado de la transformación a salida de U_{12}, los siguientes *15* bytes se presentan en sucesión continua, a razón de uno por ciclo de reloj. La operación final de la ronda consiste en la suma *EXOR* de cada uno de estos bytes con la subclave correspondiente.

Con esta finalidad, el circuito se realimenta sobre el *bus1* del multiplexor de entrada U_2. Nuevamente la *unidad de control*, será la encargada de la coordinación respecto de la presentación de los bytes correspondientes al componente *EXOR*. En este sentido generará la señal de habilitación del *bus1* de U_2 y el barrido de las *16* direcciones correspondientes a la subclave que corresponda a esa ronda, almacenada en U_3. Esta operación se sincroniza con el almacenamiento del resultado en la *RAM* U_4 cuya habilitación y la generación del bus de direcciones es también función de U_{13}.

Al finalizar la escritura de la RAM U_4, los datos quedan disponibles para su procesamiento para la siguiente ronda.

El procesamiento completo de una ronda, desde la lectura original de U_4 hasta la finalización de la escritura del resultado de la última *EXOR* en el mismo componente, consume un tiempo total de 1,9212 µs. Este tiempo es el mismo para las nueve iteraciones siguientes a la ronda inicial.

6.5 El Núcleo del Cifrador y la Ronda Final

En esta ronda no se realiza la transformación *MixColumns()*. Este es el motivo de la realimentación de la salida de U_{10} hacia el *bus2* de multiplexor de entrada U_2. En esta última ronda la descripción coincide con los conceptos vertidos en el ítem anterior hasta que comienza la lectura de la *RAM* U_{10}.

La habilitación de lectura para U_{10} es la indicación de la finalización de la operación *ShiftRows()*. En esta última ronda la diferencia es que no se habilita la operación del componente U_{12}, sino que simplemente la *unidad de control* habilita el *bus2* de U_2 y se realiza la operación *EXOR* entre los *16 bytes* almacenados rotados en U_{10} y la última subclave guardada en U_3.

Como en las rondas anteriores, el resultado de la operación *EXOR* se almacena finalmente en la *RAM* U_4. En este caso el resultado se corresponde con el resultado final de procesamiento: el cifrado de un bloque de datos de *16 bytes* siguiendo el estándar *AES-128*.

Al no realizarse la transformación *MixColumns()* esta última ronda tiene menor duración que las anteriores, completándose en 1,52 µs. La duración total de las diez últimas diez rondas es de 18,7992 µs.

6.6 Unidad de Control

Esta unidad es la encargada de generar las señales necesarias para sincronizar el funcionamiento de los diversos componentes de la implementación. Para tal efecto se pensó su diseño en la forma de una máquina de estados que en total avance sobre 4 estados [ASH02].

El funcionamiento de este componente puede presentarse en un estilo descriptivo explícito, es decir un proceso combinacional describe la función de próximo estado y las asignaciones de salida (*process* con *case* en lenguaje *VHDL*) y un proceso secuencial describe las asignaciones sobre el registro de estados en la transición activa del reloj (*process* sensible a *clock* y *reset*). La entidad se define con puertos de *reloj* y *reset* y señales de entrada y salida. En la arquitectura se define la variable *estados* que enumera todos los posibles, una señal de estado y otra de próximo estado. Este estilo identifica la parte combinacional de los elementos de memoria según el modelo de la máquina de Huffman [PAR99]. Siguiendo este criterio la máquina opera a lo largo de los *4* estados mencionados previamente.

En un estado inicial se generan todas las señales necesarias para preparar el cifrador para procesar los datos en lo que se ha denominado Procesamiento Inicial. En este sentido se habilita el funcionamiento de la interfaz de entrada activando la señal *enable* del componente *rxnibble3*. También se generan señales de control sobre los multiplexores U_2 y U_6 de tal manera que los datos del bloque que ingresa se almacenen sobre la memoria U_4 luego de haberse sumado byte por byte mediante una *EXOR* con la clave de cifrado.

El segundo estado es el que maneja el procesamiento de las Rondas Intermedias. Para poder realizar esta funcionalidad la máquina posee una variable interna que actúa como contador de ciclos de reloj y que comienza en cero al empezar una ronda. Durante estas Rondas Intermedias el bus de entrada habilitado del multiplexor U_3 es el proveniente de la propia *unidad de control* y se utiliza para la lectura de la *RAM* que almacena la clave. Esta lectura se realiza en cada ronda sobre *16* posiciones diferentes de la memoria mencionada. Por ejemplo en la ronda inicial se leen las posiciones desde *15* a *30*, en la que sigue desde *31* a *46* y así sucesivamente hasta la última ronda donde se leen las posiciones *143* a *158*. De este modo se provee a cada ronda de la subclave apropiada. Esto se realiza por medio de una señal interna que se utiliza para identificar el número de ronda presente.

Por otro lado la variable interna que cuenta los ciclos de reloj sirve para identificar los períodos de tiempo en que se habilitarán los diversos componentes del cifrador. De este modo al comienzo de

cada ronda la *RAM* U_4 se mantiene habilitada para su lectura y recién se habilita su escritura al final de una ronda. También se controla de este modo la lectura y escritura de la *RAM* U_{10}. Por ejemplo, esta memoria se habilitará para su escritura a partir del segundo ciclo de reloj desde el inicio de una ronda pues es, a partir de ese instante, cuando se obtiene el resultado de la consulta a la Caja *S* y el byte debe afectarse por la transformación siguiente, *ShiftRows()*.

Luego de *16* ciclos de escritura donde el componente *shift* maneja el bus de direcciones de U_{10} se procede a la lectura de dicha memoria. El bus de direcciones de U_{10} se multiplexa entre dos buses que genera la misma unidad de control, uno proveniente directamente de ella y el otro a través de *shift*.

En los ciclos finales del estado que maneja las Rondas Intermedias se produce la habilitación del componente que maneja la transformación *MixColumns()* y se sincroniza el resultado de la misma con la escritura de U_4 previamente mencionada.

La terminación de cualquier ronda intermedia genera la activación de una señal por parte de la *unidad de control* en su puerto denominado *ronda* y el reset del contador interno que se ha mencionado.

La unidad de control genera de este modo las señales de habilitación de todos los multiplexores, fundamentalmente con el objetivo de presentar a cada memoria involucrada en el circuito el bus de direcciones que corresponda.

Este también es el propósito del tercer estado de la máquina pero, se ha dedicado el funcionamiento del mismo al control de la última ronda del algoritmo donde no se produce la transformación *MixColumns()*. En este estado también se utiliza el contador para marcar los ciclos de reloj en los que se debe leer y escribir las memorias RAM U_4 y U_{10}. La diferencia principal se da el la habilitación del tercer bus de entrada del multiplexor U_2 que es el que proviene directamente de la salida RAM U_{10}. La máquina permanece en este estado *10* ciclos menos que en el caso de cada ronda del ciclo de Rondas Intermedias.

Por último se introdujo un estado final en la máquina donde se distingue la finalización del procesamiento completo de cifrado. En este sentido se procede a leer el contenido de la memoria U_4 durante 16 ciclos y se activa la señal del puerto *fin* de la *unidad de control* como indicación de cierre.

Capítulo 7

Funcionamiento del Cifrador

En este capítulo se desarrolla una explicación detallada del funcionamiento del diseño propuesto para la implementación en hardware del cifrador *AES-128*. Para mayor claridad, se ha elegido explicar dicho funcionamiento a partir de los resultados de las simulaciones del diseño realizadas por medio de la herramienta que para ese propósito proporciona el software *MAX+PLUS II Version 7.21 Student Edition*.

Las pruebas de funcionamiento fueron realizadas sobre varios grupos de datos cuyos resultados de cifrado figuran como ejemplo en los apéndices de la especificación de la FIPS. Dichos ejemplos incluyen el resultado paso a paso de las diversas transformaciones del algoritmo de cifrado, permitiendo de este modo la comprobación parcial del funcionamiento y la relación entre los resultados intermedios con diversos aspectos de diseño.

Los resultados volcados en este capítulo muestran el progreso en el tiempo de la operación del cifrador en el caso de un bloque de entrada particular de *16* bytes, cifrado con una clave de la misma longitud. El bloque de entrada se ha denominado *datain* y se trata de la secuencia: $3243f6a8885a308d313198a2e0370734$, expresada en formato hexadecimal. Por su parte la *clave de cifrado* utilizada es: $2b7e151628aed2a6abf7158809cf4f3c$, expresada en el mismo formato. El diseño del circuito fue dotado de numerosas señales adicionales del tipo *monitoras* con la finalidad de poder seguir paso a paso su funcionamiento.

A lo largo de la descripción la mención de los diversos componentes y señales deben referirse al gráfico del cifrador de la Fig. 6.1.

7.1 Ingreso de los Datos. Primer Ronda

Los datos mencionados pueden considerarse como una matriz de *4x4* bytes cargada por columna, como se aprecia en la Fig. 7.1. La aplicación de la primera ronda o *round* del algoritmo exige el cumplimiento de una única operación a la cadena de datos original: la matriz que la representa se suma bit a bit mediante una operación *EXOR* con la matriz equivalente que representa a la clave original. En el caso del cifrador diseñado la suma se realiza simultáneamente sobre grupos de *8* bits. Así, el resultado de sumar el primer byte de datos, 32, con el primer byte de la clave, $2b$, origina el byte 19. La Fig. 7.1 presenta los componentes y el resultado de esta primer ronda.

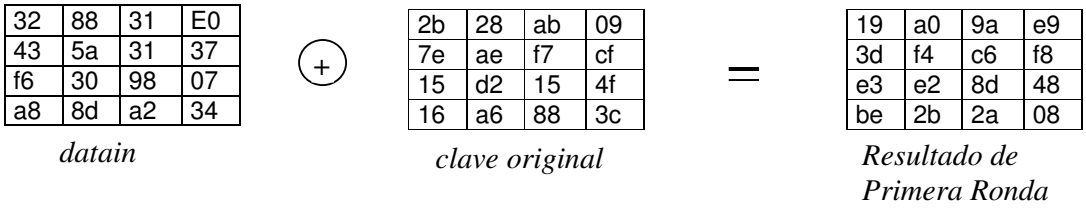

Figura 7.1. Operación en la Primer Ronda

El circuito cifrador diseñado, combina esta primera ronda del algoritmo con la carga de los datos a cifrar. Como se puede advertir en la Fig. *7.2*, los datos ingresan a una velocidad mucho más baja que la propia señal de sincronismo del circuito. Esto puede comprobarse a través de la comparación entre las señales *datain* y *clock* y la diferencia se debe a la simulación de la entrada de datos, como se ha explicado en el capítulo previo.

La Fig. *7.2* permite observar el ingreso al circuito por *nibble* de los primeros *5 bytes* de datos: $3243f6a888$, en la línea *datain*. Cada *nibble* que ingresa al circuito genera una señal de reconocimiento hacia el mundo exterior al activar el puerto *ACK* de la Interfaz de Entrada. La señal *byteready* cumple idéntica función pero relacionada con el reconocimiento de un byte completo. La señal *monitoraux4* es la representación por bytes de estos mismos datos, presente en el bus de salida del multiplexor de entrada U_2.

Esta sucesión de bytes se vuelca sobre la entrada de una *EXOR*, junto con los datos provenientes de las primeras *15* posiciones de la memoria que almacena el conjunto total de claves y

sub-claves, *RAM U_3*. La salida de esta última se representa en la Fig. *7.2* por la señal *monitoraux8*, donde se pueden observar los primeros *5* bytes de la clave original.

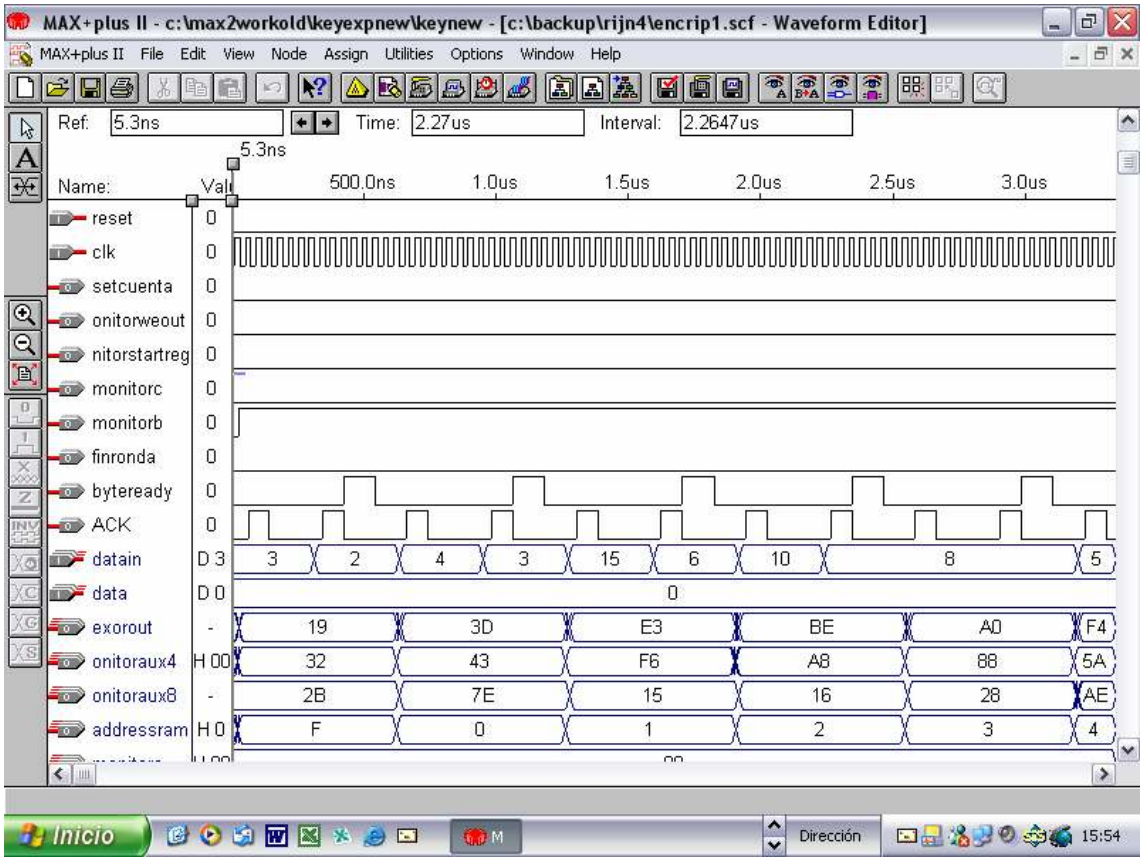

Figura 7.2. Detalle de Primer Ronda

El resultado de la operación *EXOR* es monitoreado mediante la señal *exorout* de la figura. Esta señal a su vez se corresponde con el bus de entrada de la primera *RAM* interna del cifrador, *U_4*. La sucesión de bytes de la señal *exorout* coincide con el resultado presentado en la Fig. *7.1*.

También se puede observar en la Fig. *7.2* la señal *monitorb* en estado alto a partir de la aparición del primer byte de datos válidos de la operación, esto es *19*. Esta señal es la de habilitación de escritura de la *RAM U_4* que proviene de la *unidad de control*.

No se presenta en esta figura el bus de direcciones de dicha memoria cuyo contenido, en esta fase del procesamiento, lo genera la interfaz de entrada *rxnibble3*. Se trata de la sucesión de los

números *0-15* que indican las posiciones de almacenamiento en memoria en consonancia con los correspondientes bytes de *exorout*. De esta manera se realiza la carga del bloque de datos inicial a cifrar, al mismo tiempo que se cumple con la primera operación del cifrado a la velocidad en que se ingresan dichos datos. Exactamente lo mismo sucede con el barrido de bus de direcciones de la *RAM* que almacena las claves: se maneja desde la interfaz de entrada de datos en esta primera ronda. Existe, por lo tanto, sincronismo de ambos barridos con la velocidad de ingreso de datos al cifrador.

La Fig. *7.3* presenta la situación del cifrador al finalizar la carga del bloque de datos inicial a cifrar. Se puede observar en esta figura el ingreso de los dos últimos *nibbles* de datos, 34, al mismo tiempo en que se obtiene el resultado de la *EXOR* de los mismos con el último byte de la clave original, $3c$, es decir 08.

La llegada del último *nibble* genera un pulso en la señal *setcuenta* y sendos cambios en las señales *monitorb* y *monitorc*. El pulso mencionado es un aviso para el circuito externo desde donde provienen los datos: el cifrador estará ocupado en la realización de las siguientes rondas del algoritmo y no podrá atender entonces la carga de datos externa hasta el próximo aviso.

A su vez, la señal *monitorb*, como se comentó anteriormente, controla la habilitación de escritura de la *RAM* U_4 donde se ha almacenado el resultado de la *EXOR* inicial. Se trata en realidad de una señal generada por la *unidad de control* que se corresponde con el puerto *we* del componente *round1* U_{13}. Esta señal adquirirá a partir de ahora un ritmo regular y se verá en alto durante las rondas subsiguientes cada vez que se almacenen los datos resultantes de la aplicación de una ronda del algoritmo.

Por su parte, la señal *monitorc* es una indicación desde la interfaz de entrada a la *unidad de control*. Se trata de un aviso de que ha finalizado la carga de los datos y ahora es la *unidad de control* la que debe hacerse cargo por completo de la situación. Se podría interpretar así el comienzo de las operaciones reiterativas o de ronda del algoritmo.

Es de destacar que la carga de los 128 bits del bloque original finaliza a los *10,1448* µseg. En

ese momento el resultado de la *EXOR* queda almacenado en la *RAM* U_4, disponible para su procesamiento en la siguiente ronda.

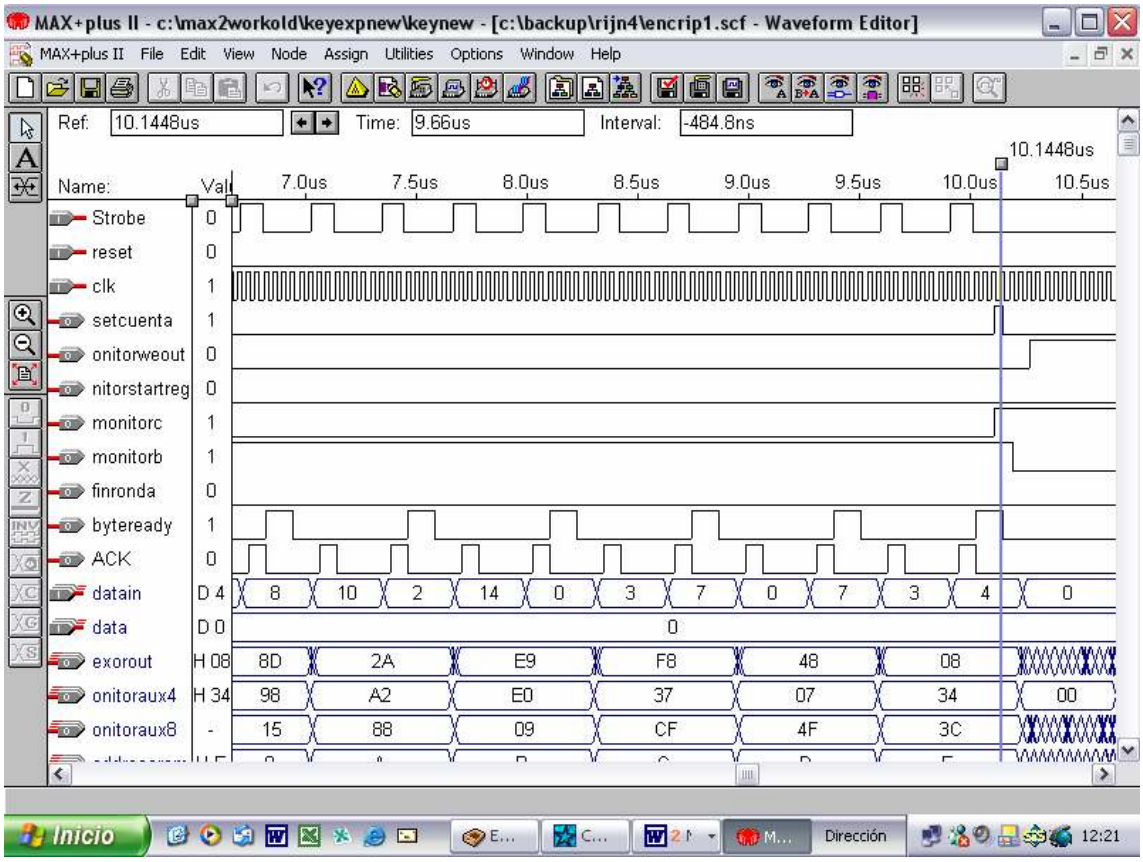

Figura 7.3. Finalización de Primer Ronda

7.2 Rondas Intermedias

La Fig. *7.4* presenta las operaciones a realizar durante el desarrollo de la primera ronda intermedia, explicitando los resultados numéricos parciales a obtener luego de cada operación.

Como se puede observar en dicha figura, se trata de la primera ronda en que se aplican todas las operaciones características del algoritmo *Rijndael*. Esta situación se repetirá durante nueve rondas consecutivas. De este modo, los comentarios vertidos para esta primera ronda intermedia serán válidos para las siguientes ocho rondas.

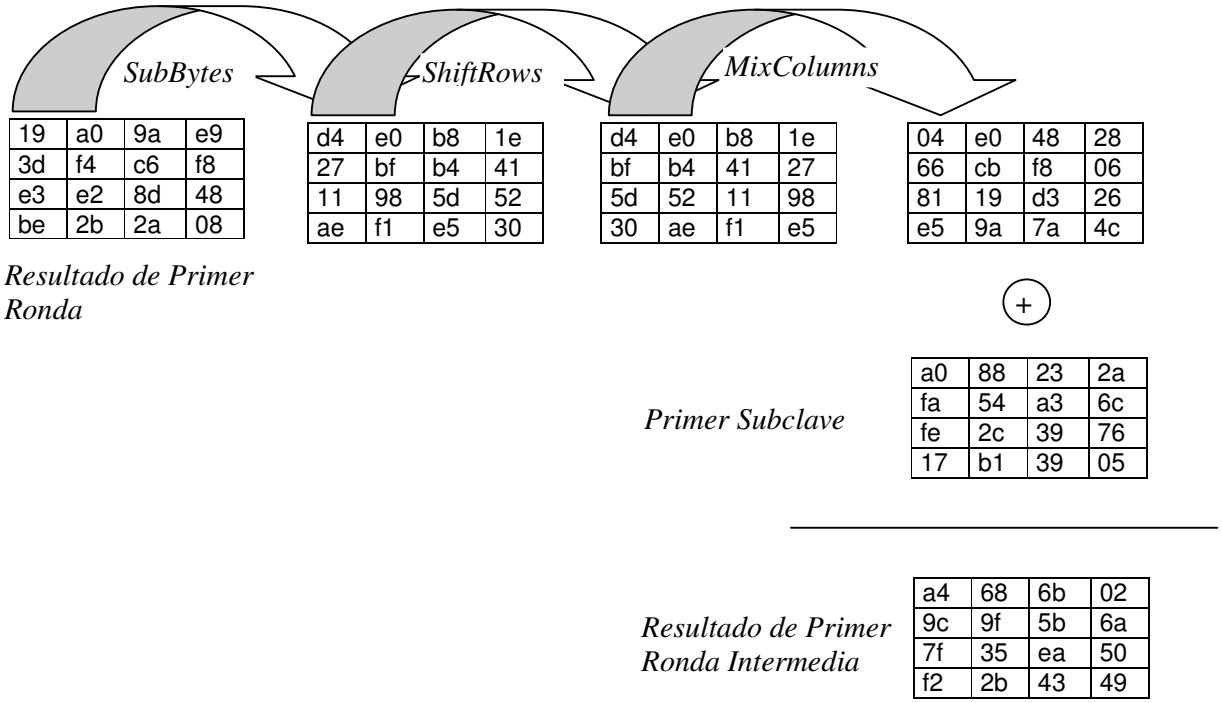

Figura 7.4. Operaciones de la Primer Ronda Intermedia y sus Resultados.

En la Fig *7.5* se observa que, al comenzar la ronda, baja la señal *monitorb* indicando el comienzo de dieciséis ciclos de lectura de la *RAM* U_4, donde se han almacenado los resultados de la ronda previa. La figura también permite apreciar que se ha cortado el flujo de entrada de datos en el bus *datain* y, por lo tanto, sobre la entrada de la *EXOR* que representa la señal *monitoraux4*. No importa en este momento lo que ingrese por el bus de datos de la *RAM* mencionada pues la misma se haya en situación de lectura. Esta operación exige el barrido de todas las posiciones de dicha memoria a través del bus de direcciones. La generación de la secuencia esperada pasa a estar en manos de la unidad de control, *round1* U_{13}, a través de la salida *qbaja*. Esta se inyecta sobre el bus correspondiente de la memoria U_4, a través del multiplexor U_5. Este último componente queda habilitado desde esta primera ronda intermedia para permitir el paso de la señal generada por la *unidad de control*.

Es también a partir de esta primera ronda intermedia que todas las operaciones se realizan a la velocidad del reloj ó *clock* del sistema.

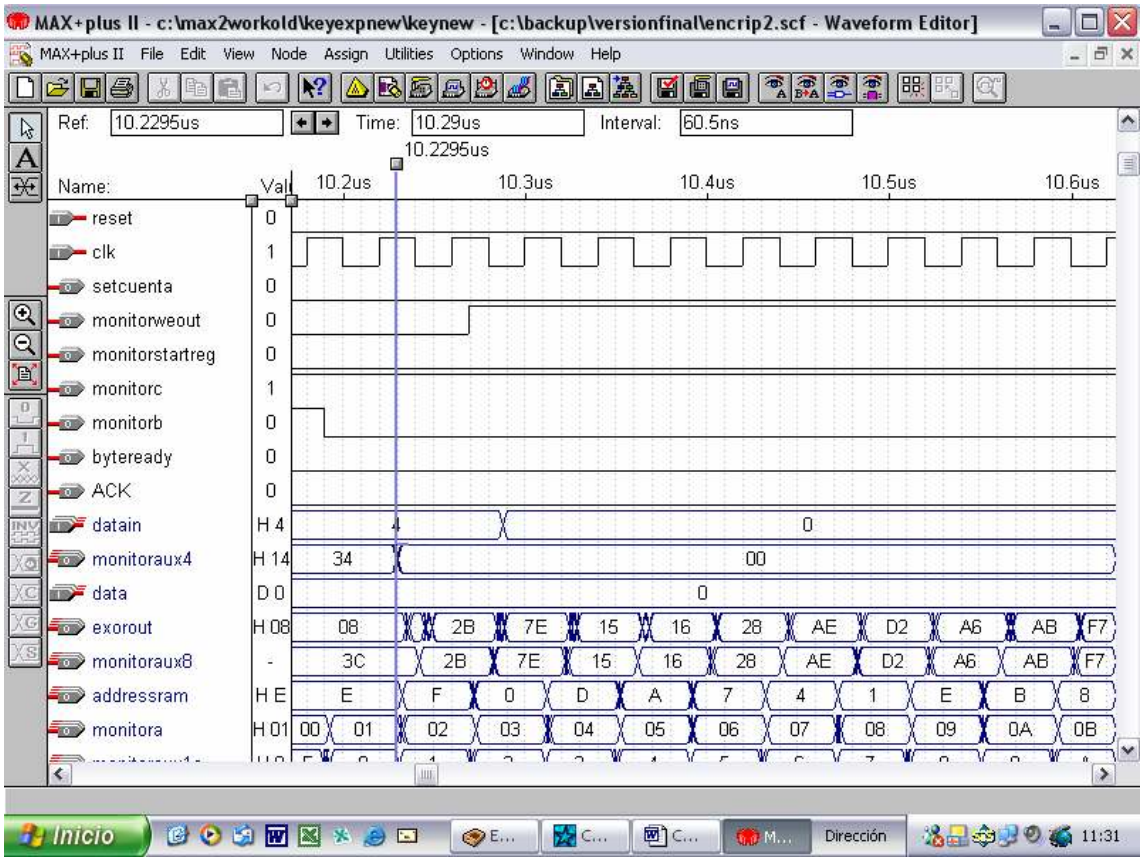

Figura 7.5 Comienzo de la Primer Ronda Intermedia.

La primera transformación que sufren los datos provenientes de la ronda previa es la consulta a la *Caja S*, es decir la operación *SubBytes()*. Como se ha mencionado en capítulos previos, este diseño desarrolla dicha operación por medio de la lectura de una memoria *ROM* U_8 de *256x8*. En esta memoria se guarda la tabla que representa la transformación. Los datos procedentes de la ronda previa, resultado de la lectura secuencial de la *RAM* U_4, ingresan por su bus de direcciones a la memoria *ROM* a razón de una posición por ciclo de reloj. La posición de memoria presentada contiene el dato transformado que le corresponde. Esta situación se puede observar en el bus de la señal monitora del bus de direcciones de U_8, *monitorauxrom* de la Fig. *7.6*. Se trata de la sucesión de bytes $193de3bea0f4e22b9ac6$.... que se almacenaron previamente en la *RAM* U_4.

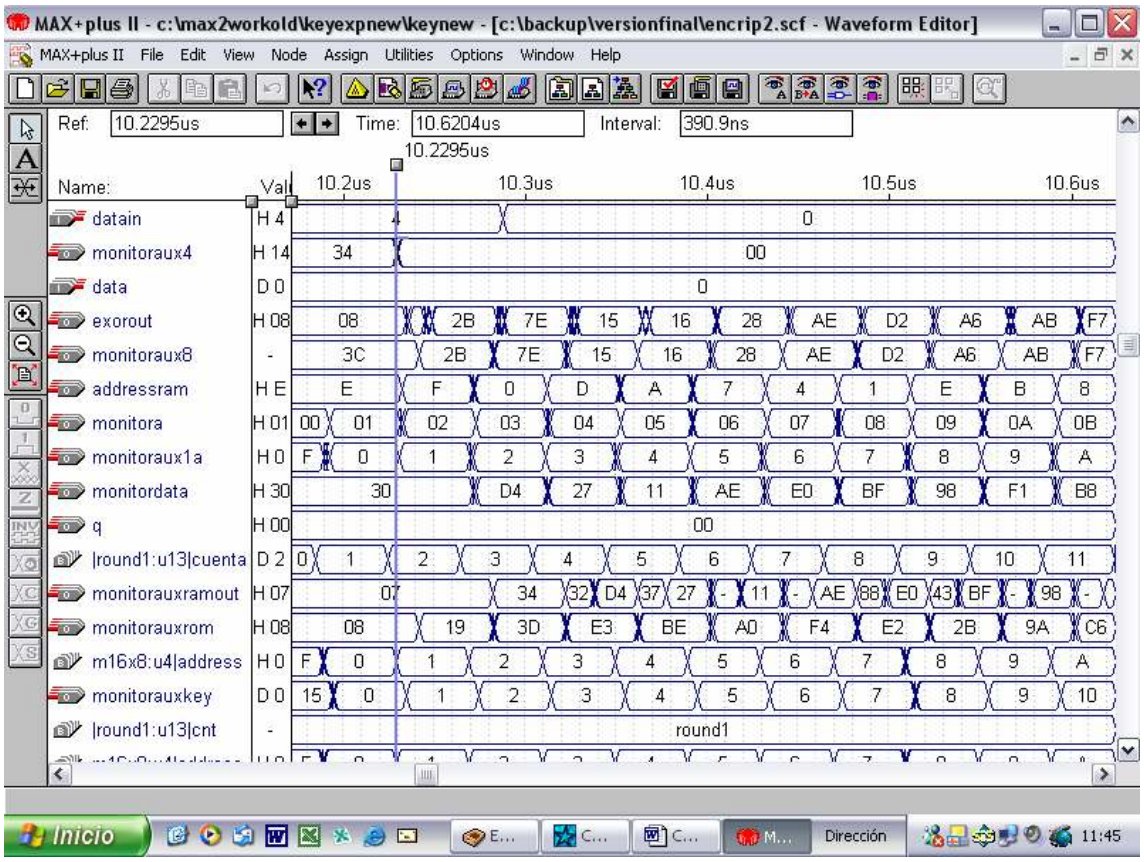

Figura 7.6. Otro aspecto del Comienzo de la Primer Ronda Intermedia.

El resultado de la consulta se presenta en la señal *monitordata* y se trata de la sucesión de bytes $d42711aee0...$. Se puede observar también en la figura el retraso de casi un ciclo inherente a la presentación de la dirección de memoria y la obtención del dato de dicha locación a la salida.

El primer *nibble* de un *byte* sobre *monitorauxrom* se interpreta como la posición en la fila correspondiente de la Tabla *4.2* y el segundo *nibble* como la posición en la columna de dicha tabla. Así el primer byte, *19*, representa un posición de la memoria *ROM*, donde se haya almacenado el byte resultante de su transformación, *d4*. Este será el dato que se obtendrá de la lectura de la *ROM*, sobre el bus *monitordata* de la figura. En cuanto a los tiempos, el primer byte proveniente de la lectura de la *RAM* U_4 aparece a los *96,6 nseg* luego de haberse iniciado el *round1.* En el siguiente flanco de subida del reloj, aproximadamente *30 nseg* después, se presenta sobre el bus de salida de la ROM el primer byte válido resultante de la transformación *SubBytes()*. En ese momento se habilita la escritura de la *RAM* U_{10}, como se puede apreciar en la Fig. *7.7*. Se trata de la señal *monitorweout* que se

corresponde con el port *weout* de la unidad de control y que permanecerá en estado alto a lo largo de 16 ciclos de *clock*, para permitir la escritura de los bytes provenientes de la consulta a la *ROM*.

Los datos deben ahora ser sometidos a la operación de rotación por filas, *ShiftRows()*. El cifrador implementado realiza esa operación haciendo coincidir la presencia de cada byte de datos con la posición apropiada de memoria *RAM* U_{10} en que debe almacenarse. De este modo el componente U_{10} oficia de registro para almacenar los datos intermedios de manera temporal, tal como sucede con U_4 al finalizar cada ronda.

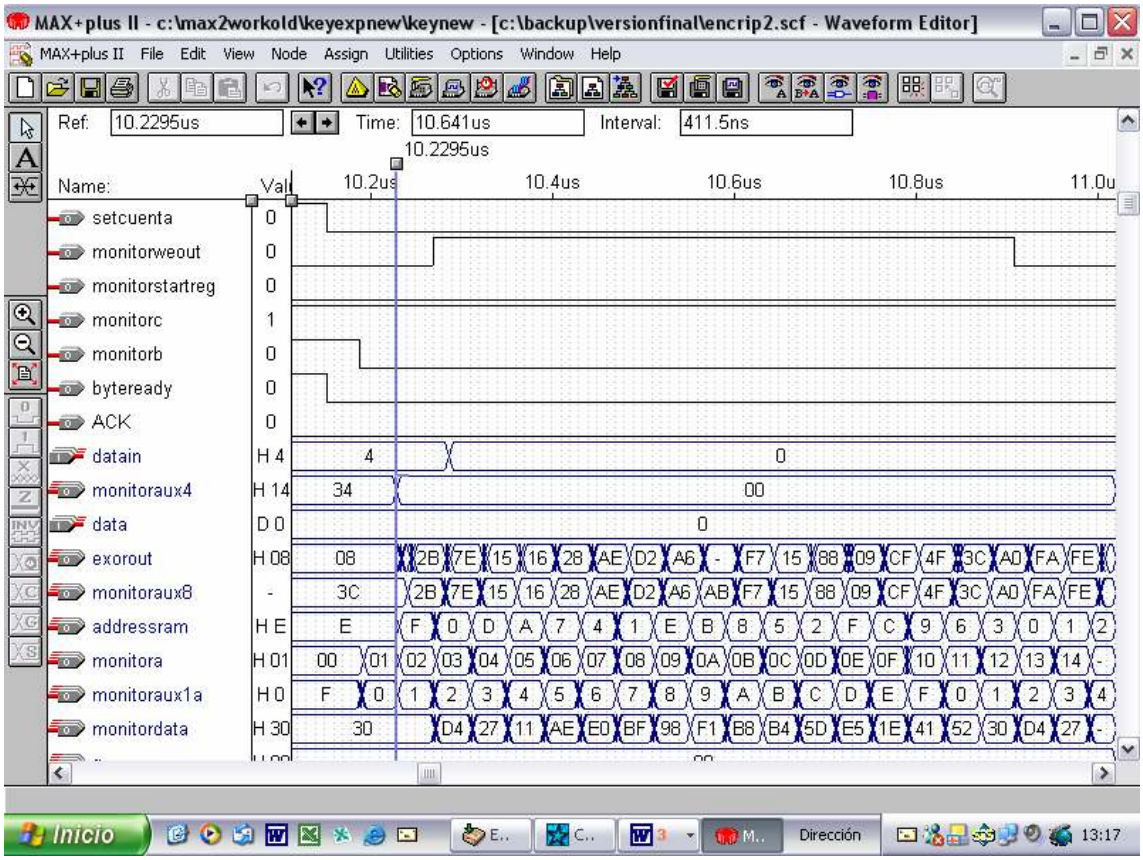

Figura 7.7. Primer Ronda Intermedia. Resultados de la Operación *SubBytes()*.

Para comprender cómo se genera la rotación se debe observar en la Fig. *7.7* las señales *monitordata* y *addressram*. Ambas señales están en sincronismo, la segunda representa las posiciones de memoria donde se guardan los datos que presenta la primera. De este modo, el primer byte (*d4*) queda en su posición original (*0*) mientras que el segundo (*27*) se almacena en la posición

13, presentada en la figura en formato hexadecimal (*0x0d*), el tercero (*11*) pasa a ocupar la posición *10* (*0xa*), el cuarto (*ae*) la *7* y así sucesivamente, siguiendo el orden establecido en la Fig. *7.2*.

Como se puede observar los bytes primero, quinto, noveno y treceavo no cambian de posición pues pertenecerían a la primera fila de la matriz *State* que no se ve afectada por la rotación. Es conveniente recordar que la generación de la dirección de almacenamiento apropiada queda en manos de la *Unidad de control* U_{13} y de los componentes *P* y *shift* de la Fig. *6.1*.

De este modo en *16* ciclos de *clock* se combina la realización de dos transformaciones: *SubBytes()* y *ShiftRows()*.

La lectura ordenada de la *RAM* U_{10} presentará los bytes rotados de la Matriz de Estado a la siguiente operación. Nuevamente la *unidad de control* presentará el bus de direcciones a apropiado a la *RAM* mencionada a través de la habilitación del bus de entrada del multiplexor U_{11} que no pasa por el componente *shift*.

La ronda se completa por la aplicación de la transformación *MixColumns()* y una operación *EXOR* con los bytes de la sub-clave correspondiente. Como dicha transformación exige el trabajo con *4* bytes de modo simultáneo el componente *reg32mix8x8*, encargado de la realización de *MixColumns()*, posee dos registros internos para transformación serie-paralelo del flujo de bytes original en la entrada y transformación paralelo-serie para regresar al bus interno de *8 bits* propio del cifrador. El primero de dichos registros genera una demora inicial en la transformación *MixColumns()* al momento de la carga de los primeros *4 bytes*. El segundo implica una demora del mismo tenor en la presentación final. A partir de allí la generación del resultado es una cadena continua de *bytes*.

En la Fig. *7.8* se puede observar una demora de aproximadamente cinco ciclos entre el flanco de bajada de la señal *monitorweout*, indicación de lectura de la *RAM* U_{10}, y el flanco de subida de la señal *monitorstatreg* para la habilitación de la transformación *MixColumns()*. Como se explicó, este cambio en *monitorweout* se produce ni bien se termina de escribir el último byte proveniente de la *caja S* (*30*) en la señal *monitordata*, en su correspondiente posición de memoria ¨rotada¨ (*3*) de

addressram.

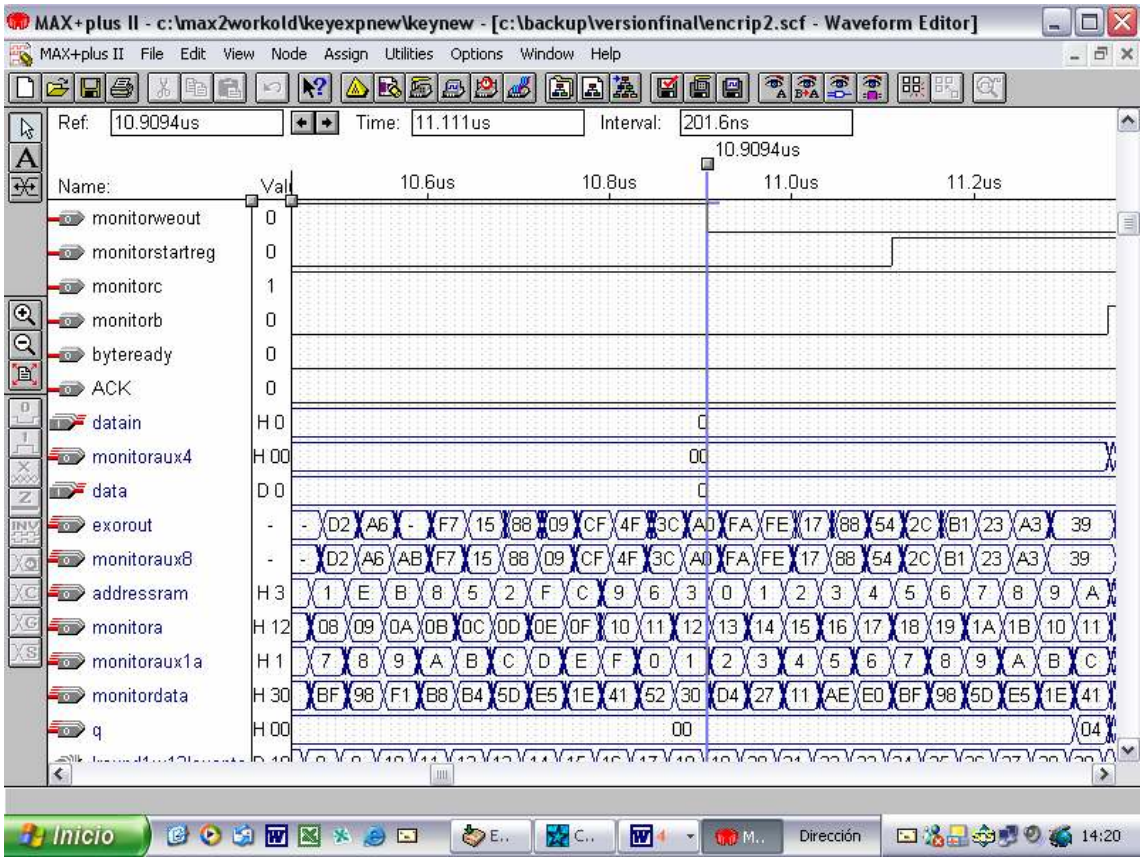

Figura 7.8. Inicio de *MixColumn()* para Primer Ronda Intermedia

A continuación se deberán tomar los cuatro primeros bytes del resultado intermedio o Matriz de Estado, que se corresponden a los cuatro primeros bytes almacenados en la memoria *RAM* U_{10} para poder aplicar la transformación subsiguiente. Por este motivo la señal *monitorstartreg* no habilita la operación del componente *reg32mix8x8* sino hasta cinco ciclos después, uno de establecimiento de lectura de la *RAM* y cuatro más para cargar el registro interno con los cuatro primeros bytes presentes en *monitorauxramout*: $d4bf5d30$.

La Fig. 7.9 se incluye para graficar el retardo intrínseco a la elección de trabajar con un bus interno de *8 bits*. Esta elección se enfrenta con su peor penalización en el retardo, también llamado latencia, que existe entre la presentación del primer byte al componente que implementa la

transformación *MixColumns()* y la salida del primer *byte* válido de resultado. Este retardo es el existente entre el byte *d4* de la señal *monitorauxramout* y el byte de idéntico valor sobre la señal presentada como *q* en la figura.

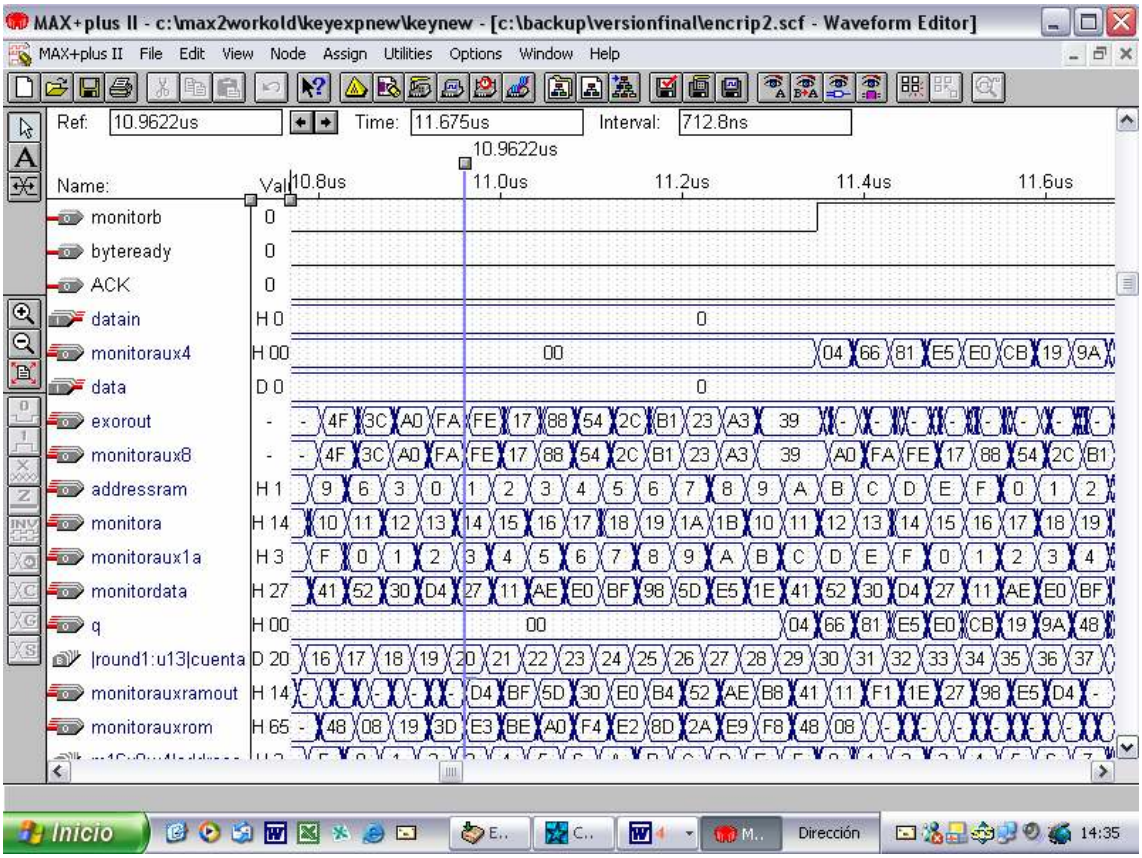

Figura 7.9. Retardo propio de *MixColumn()*

La latencia incluye las dos conversiones ya mencionadas: una serie-paralelo para poder realizar la operación por columnas de la Matriz de Estado y otra paralelo-serie al final de la operación para regresar al procesamiento sobre el bus original de *8 bits*. La operación genera en total un retardo adicional de *10* ciclos, contados desde que se comienza a leer la memoria U_{10} hasta que se obtiene el primer byte a la salida del componente U_{12}.

La desventaja mencionada queda compensada por el hecho de que el resultado finalmente se presenta como un flujo continuo de 16 ciclos de reloj, tiempo durante el cual se puede realizar al mismo tiempo la operación final de la ronda, consistente en la *EXOR* con los respectivos bytes de la

sub-clave.

Por este motivo se incluye en la figura anterior la presencia de la señal *monitorb*. Esta señal maneja la escritura de la *RAM* U_4, y se genera para almacenar el resultado final de la aplicación de una ronda completa a la cadena de *128 bits*. Se puede observar que se habilita la escritura de dicha *RAM* en el mismo momento que aparece el primer byte válido sobre la señal *monitoraux4*.

Las señales *q* y *monitoraux4* son iguales, con la salvedad que la segunda se encuentra retrasada un ciclo respecto de la primera. Esto es debido a la presencia del multiplexor de entrada U_2, necesario para realimentar sobre la entrada de la *EXOR* la señal apropiada. Este componente diferencia tres situaciones de ingreso de datos al cifrador propiamente dicho: el flujo de datos inicial de *128 bits*, la realimentación desde la operación *MixColumns()* en las *8* rondas subsiguientes y la realimentación desde la operación *ShiftRows()* en la última ronda del algoritmo.

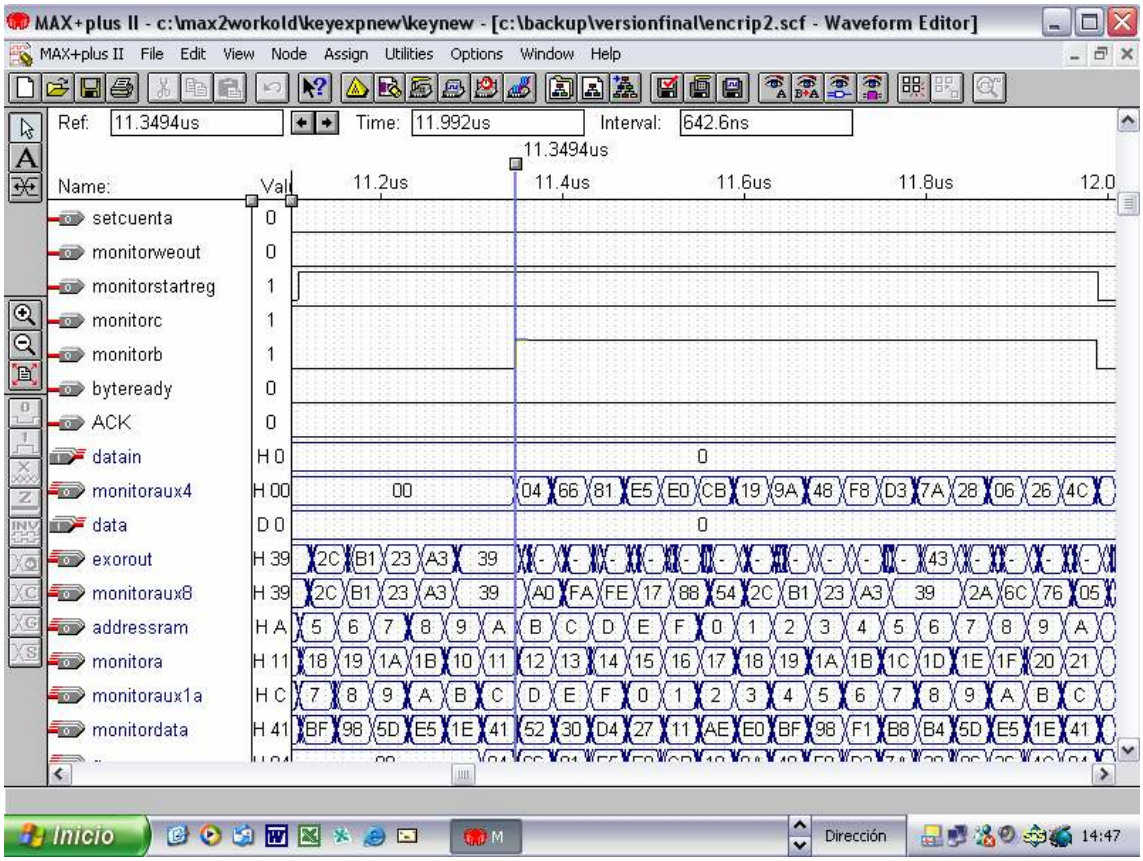

Figura 7.10. Resultado de *MixColumn()*

Complementando las operaciones mencionadas, se puede advertir en la Fig. *7.10* que las señales que, por un lado habilitan la realización de la transformación *MixColumns()* y por el otro permiten la escritura de la *RAM U₄*, señales *monitrostartreg* y *monitorb*, bajarán al mismo tiempo. Es decir que la presencia simultánea de estos cambios puede usarse para generar la señal de finalización de la ronda.

Durante todo el tiempo en que la señal *monitorb* se encuentra en alto se realiza la *EXOR* entre los datos realimentados desde la transformación *MixColumns()* y la sub-clave de la primera ronda, a la vez que se escribe el resultado en la *RAM U₄*.

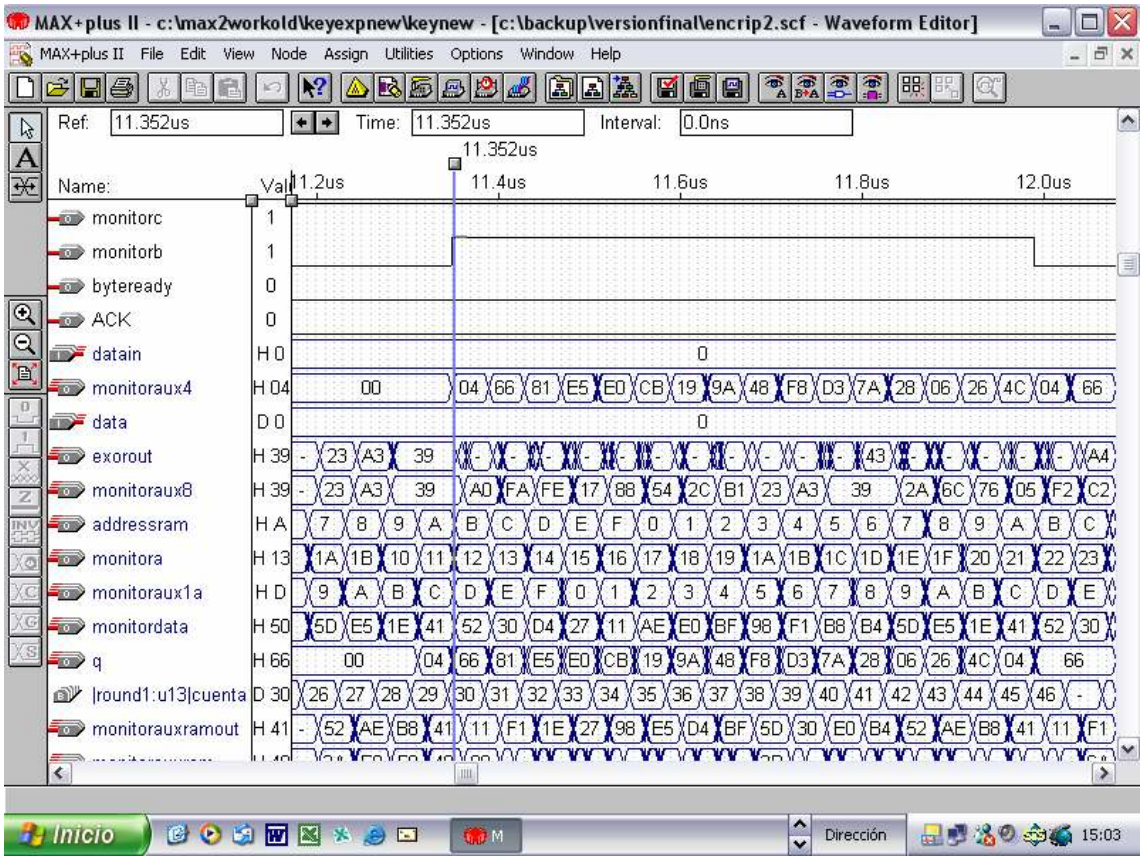

Figura 7.11. *EXOR* final.

La sub-clave mencionada es la sucesión de *bytes* que se presenta en la señal *monitoraux8* de la Fig. *7.11*. Es la primer sub-clave generada previamente a partir de la clave original $a0\,fafe1788\ldots$

Se obtiene de la lectura de las posiciones *16* a *31* de la memoria que almacena la clave y sus derivadas, *RAM* U_3. Las direcciones las genera la *unidad de control* a través de su puerto de salida *q*.

De este modo finaliza la primera ronda intermedia con el *Resultado de la Primer Ronda Intermedia* (ver Fig. *7.4*) almacenada temporalmente en la *RAM* U_4 para su utilización posterior en la siguiente ronda.

Las siguientes ocho rondas serán exactamente iguales a la descripta coincidiendo inclusive en su duración. La única variación se reflejará en el hecho que cada ronda se caracteriza por una sub-clave diferente. En este diseño la diferencia se resuelve en la generación por parte de la *unidad de control* de los números apropiados sobre el bus de direcciones de la *RAM* U_3. Se trata de leer cada vez las *16* posiciones consecutivas de dicha memoria que correspondan al momento de realización de la *EXOR* final de la ronda que se esté procesando.

7.3 Ronda Final

La Fig. *7.12* representa las operaciones a realizar en la ronda final para el cifrado de los 128 bits originales que se han utilizado como ejemplo en este capítulo. Si se contrasta esta figura con la Fig. *7.4*, se podrá observar que la última ronda difiere de las anteriores ya que no se realiza en la misma la transformación *MixColumns()*.

El resto de las transformaciones son iguales y su orden no varía. Para el diseño adoptado, esta característica se traduce de una manera muy sencilla. Simplemente se permite la realimentación directa del resultado de la operación *ShiftRows()* sobre la *EXOR* de entrada habilitando una tercera entrada del multiplexor U_2 . A su vez, para evitar componentes de ruido, en esta ronda se deshabilita la operación *MixColumns()*, situación manejada por la señal *startreg* proveniente de la *unidad de control*.

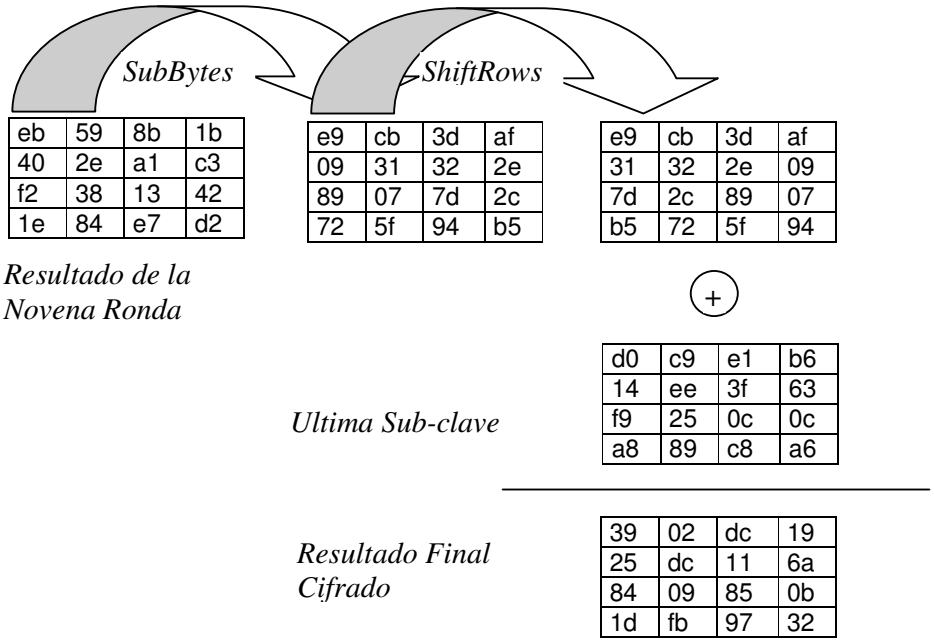

Figura 7.12. Operaciones en la Ultima Ronda.

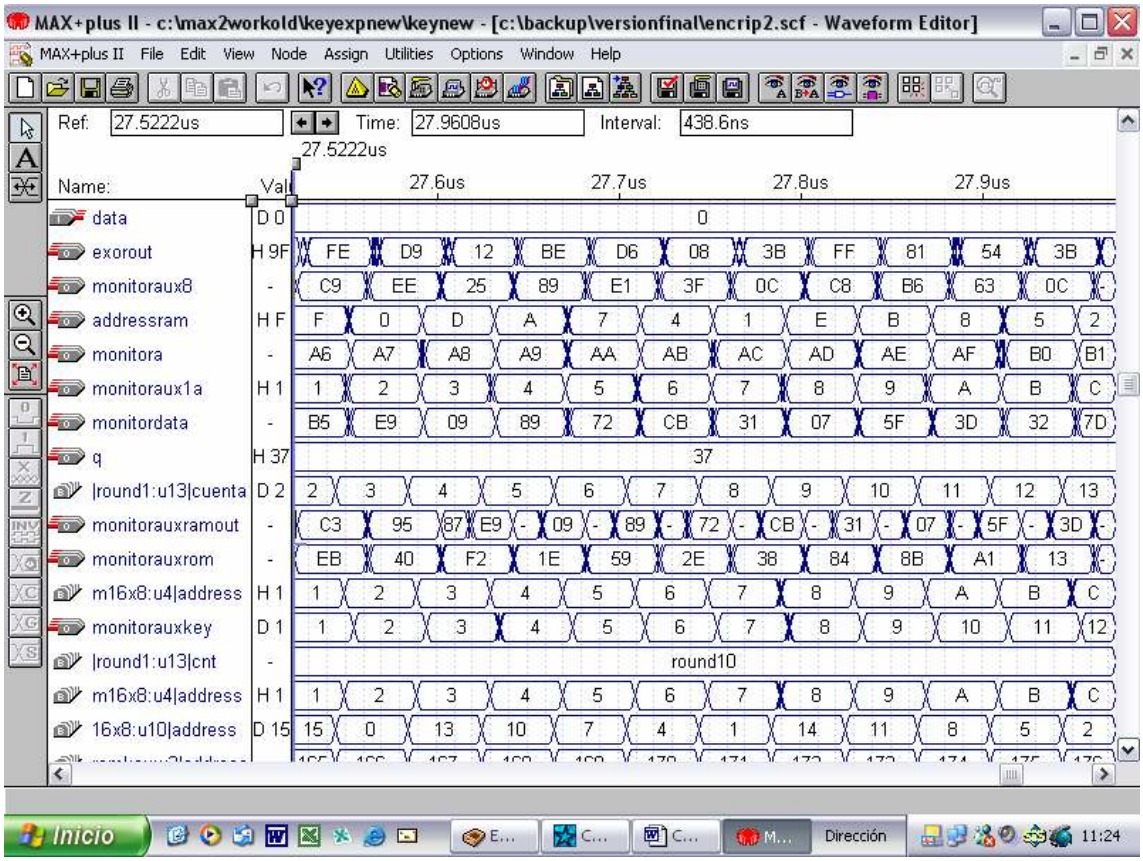

Figura 7.13. Comienzo de la Ultima Ronda.

En la Fig. *7.13* se puede observar el barrido secuencial del bus de direcciones de la *RAM* U_4 que permite la lectura de los datos allí almacenados, representados por la señal *monitorauxrom*. Esta sucesión de bytes deberá pasar la transformación *SubBytes()* a través del acceso a la *ROM* U_8. En la figura se puede observar la señal *monitordata* que verifica la transformación mencionada con un ciclo de retraso respecto de la sucesión de entrada debido al inevitable retardo introducido en la lectura de la *ROM*. De este modo, $e9$ es el byte transformado de eb, 09 lo es de 40, y así sucesivamente.

A la par de la realización de dicha transformación se escribe la memoria *RAM* U_{10} con el resultado afectado por la siguiente operación, *ShiftBytes()*. El bus de direcciones de U_{10} es generado a través de la unidad de control y el componente *shift* de tal manera de almacenar los bytes de la Matriz de Estado en orden rotado. La lectura de esta memoria se efectuará de manera secuencial y en los dieciséis ciclos posteriores a la escritura.

En la Fig. 7.14 se puede observar la aparición del byte $e9$ sobre la señal *monitorauxramout* marcando el inicio de la lectura ordenada de la *RAM* U_{10}. Dicha sucesión de bytes deberá ahora realimentarse directamente hacia la entrada. De este modo, se puede observar la aparición de la misma sucesión de bytes sobre la salida del multiplexor de entrada, en la señal *monitoraux4.* Existe un retardo de un ciclo entre ambas señales justamente debido a la presencia del multiplexor.

La señal *monitoraux4* es una de las entradas de la *EXOR*, la última operación a realizar en la ronda. Debe coordinarse la presencia de esta señal con la de la sub-clave de la última ronda sobre la señal *monitoraux8.* La coordinación mencionada se realiza a través de la habilitación de la lectura de las posiciones *160* a *175* de la *ramkey* U_3. Como se explicó anteriormente, esta funcionalidad queda en manos de la *unidad de control*.

Por otro lado, debe coordinarse la operación *EXOR* con la escritura de la *RAM* U_4 pues en esta memoria queda almacenado el resultado del cifrado del bloque. En este sentido la *unidad de control* comunicará al anfitrión la disponibilidad del resultado a través de la señal de *fin*, indicando de este modo la finalización del ciclo de aplicación del algoritmo sobre un bloque de datos de *128 bits*.

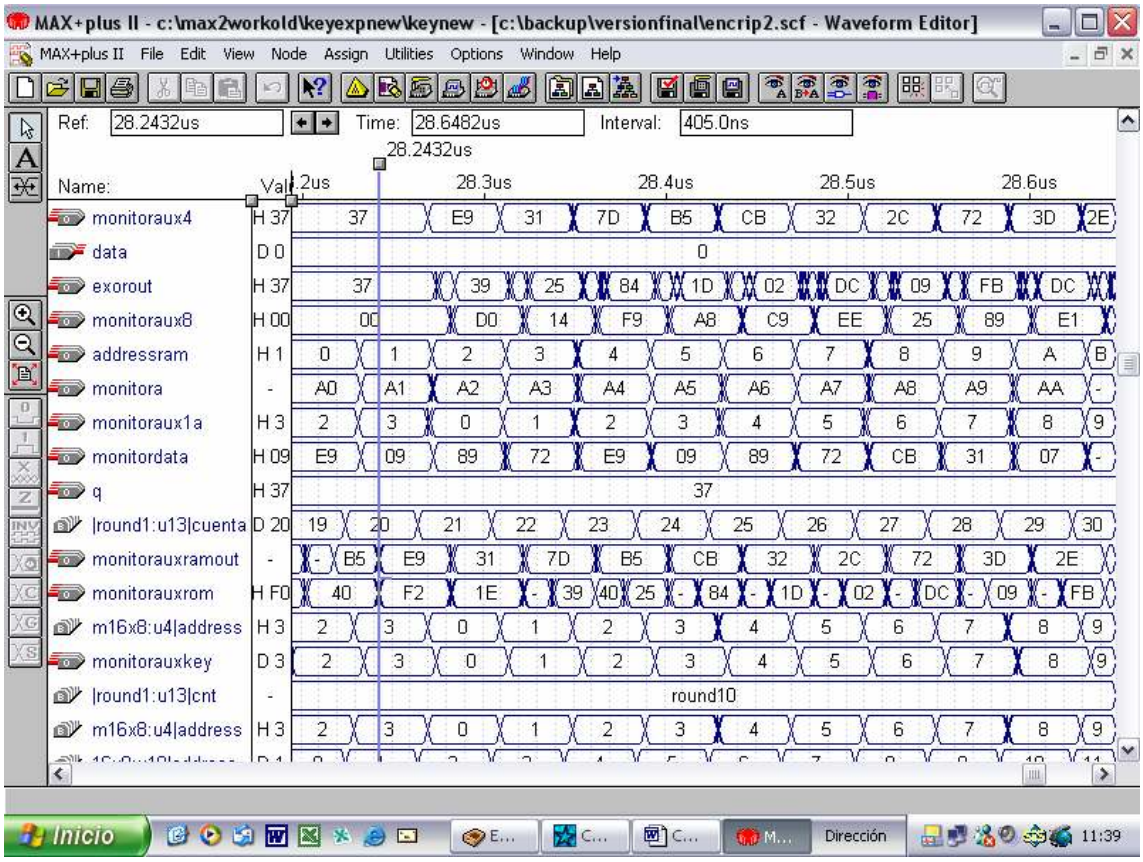

Figura 7.14. Realimentación de *ShiftRows()* sobre la entrada.

Capítulo 8

Resultados

Este capítulo presenta los resultados asociados al cifrador diseñado. Antes de entrar plenamente en el tema es conveniente aclarar los parámetros adoptados comúnmente para medir las prestaciones que una dada implementación en *FPGA* puede ofrecer.

8.1. Parámetros de Medición de Resultados

Suele ser frecuente en términos de circuitos criptográficos la comparación de implementaciones FPGA según el número de celdas lógicas (*LC´s*) que consumen y el número de bits de memoria embebida que utilizan. También se usan la latencia, la frecuencia de reloj y la velocidad ó *throughput* del circuito como factores de evaluación de calidad. En este sentido es primordial entender a qué se refiere cada uno de los parámetros mencionados como para valorar convenientemente la calidad de una implementación. Interesa, una vez definidos los parámetros de evaluación, la aplicabilidad del circuito en cuestión ya que esto fijará las prioridades en la implementación.

La interpretación de la estructura interna de la *FPGA* utilizada como objetivo del proyecto es muy importante para la interpretación de los reportes que ofrecen las herramientas de diseño.

En este sentido conviene recordar que la estructura interna del dispositivo *Flex10k* contiene un arreglo embebido que consiste en una serie de *EABs*. Cada una de las *EABs* provee *2.048* bits de memoria pero también pueden usarse para implementar funciones lógicas más complejas y en este sentido pueden pensarse como un conjunto de entre *100* y *600* compuertas.

Por otro lado el dispositivo está compuesto de un arreglo lógico que consiste de una serie de *LABs*. Cada *LAB* contiene *8* elementos lógicos (*LE*) y una interconexión local. Cada *LE* consiste de una tabla de búsqueda (*LUT*) de *4* entradas más un *FF* programable y caminos de señal dedicada para acarreo, en inglés *carry*, y funciones cascada. Un *LAB* puede usarse para crear bloques lógicos

de tamaño mediano, tales como contadores de *8* bit, decodificadores de direcciones o máquinas de estado. Los *LABs* pueden combinarse entre sí para crear bloques lógicos más importantes. Cada *LAB* se puede pensar como un conjunto de *96* compuertas. Esta estructura se representa en la Fig. 8.1 [BRO96].

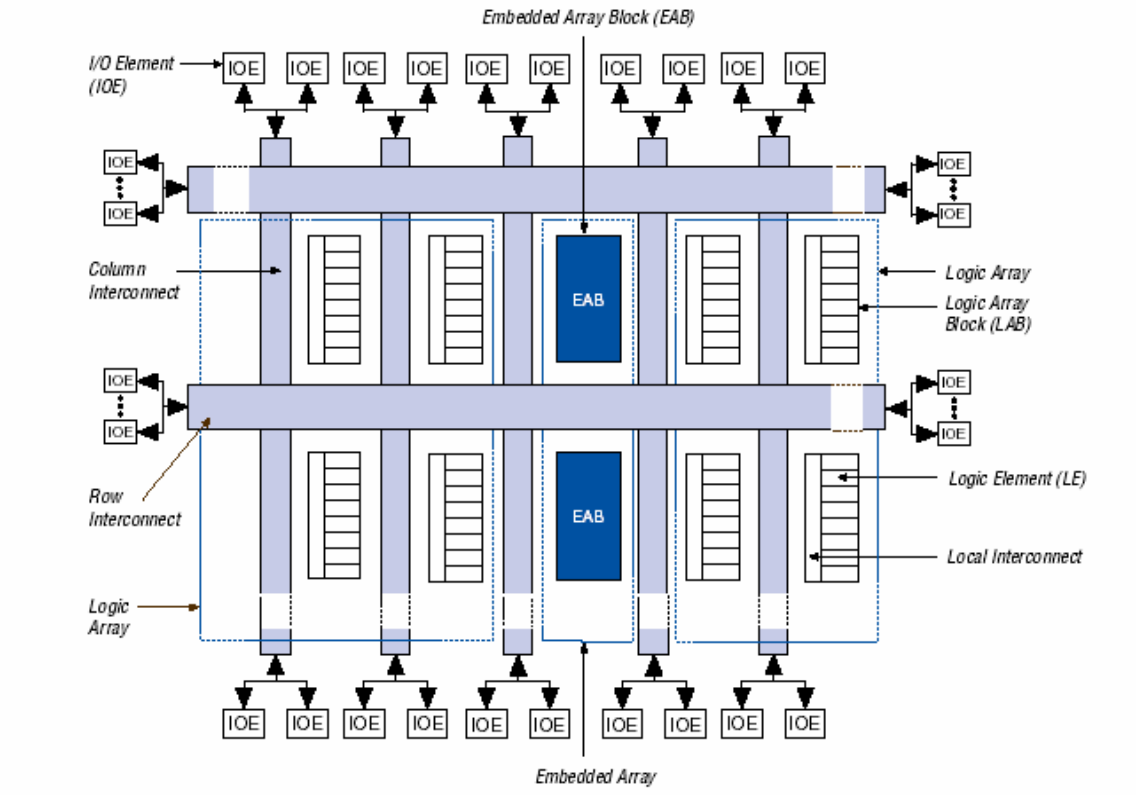

Figura 8.1. Diagrama en Bloques del Dispositivo Flex10k.

Una implementación particular generalmente se evalúa en algunos aspectos de acuerdo a los resultados obtenidos a partir de un sintetizador lógico ofrecido por la misma herramienta utilizada para el diseño original. El sintetizador es un módulo del compilador que utiliza una serie de algoritmos para sintetizar la lógica de un proyecto particular, minimizando la cuenta de compuertas, removiendo lógica redundante y, en general, utilizando la arquitectura del dispositivo elegido de la manera más eficiente posible.

El procesamiento puede ser adaptado con opciones lógicas y asignaciones de estilo de síntesis lógica. La adaptación puede ser de dos tipos: síntesis multinivel ó estándar. La síntesis multinivel

ofrece hasta *20* opciones individuales de síntesis. La estándar solo un conjunto limitado de estas [MAX96].

Una vez realizada la síntesis, el sintetizador generará un reporte con los resultados. Se trata de un archivo dividido en varias secciones, una de las cuales, *Tiempos y Ajustes de la Compilación*, en inglés *Compilation Settings & Times*, habrá registrado el estilo de síntesis lógica seleccionado previamente para la compilación, así como también el tiempo empleado durante el procesamiento del módulo particular.

El proceso de síntesis de este cifrador fue realizado con tipo multinivel y estilo normal para la familia *Flex10K*. El estilo normal define el ajuste de las opciones lógicas de manera que el uso de los recursos del dispositivo en cuestión sea el más eficiente posible, sin el agregado de retardos de tiempo excesivos. Del conjunto global de opciones conviene destacar que la optimización fue realizada en términos similares en cuanto a área y velocidad.

Es importante entender los conceptos mencionados para poder efectuar hacer una interpretación correcta de los archivos de reporte generados con la herramienta. En este sentido los reportes de Altera miden la cantidad de recursos utilizados en términos de celdas lógicas o elementos lógicos, *LC´s* ó *LE´s*, pues se trata del bloque de construcción básico en un dispositivo de Altera [ALT98]. También se genera numéricamente la cantidad total de bits de memoria utilizados. Otro parámetro interesante ofrecido en estos reportes es el porcentaje de recursos usados dentro del dispositivo en cuestión o porcentaje de utilización, tanto de memoria como de *LC´s*. Este número se basa en la cuenta total de bits de memoria y de *LC´s* respectivamente. De la obtención de estos resultados depende la medición de la calidad de la implementación en términos de área.

En general, un reporte de Altera es un archivo consistente de varias secciones. Los parámetros mencionados previamente así como la cantidad de patas de entrada/salida y el porcentaje de memoria utilizado se generan en la sección de resumen del dispositivo. En otra sección se especificará las memorias sintetizadas como arreglos embebidos ó *EAB´s*.

Una de las secciones más interesantes del reporte se refiere a la información específica del dispositivo y un diagrama de la misma muestra las patas ó *pins* usados del dispositivo elegido como objetivo del diseño. Se aclara de esta manera en dicho diagrama si la pata en cuestión no está conectada (N.C.), conectada a fuente (VCC), a masa (GND) o no usada (RESERVED). Esta última, ya sea de entrada o de salida debe dejarse desconectada.

El porcentaje de recursos utilizados suele describirse con mayor granularidad al presentar el dispositivo en cuestión mediante una tabla donde se discrimina los detalles de implementación por número de arreglo lógico *LAB* dentro del dispositivo. Si éste contiene *EABs*, se desplegará la misma información anterior pero en términos de los diferentes *EAB´s* utilizados.

Una segunda parte de la sección del dispositivo presentará un resumen de la utilización de los recursos. El sintetizador también informa si tuvo que crear celdas sin correspondencia con el diseño original. La parte final permitirá apreciar la asignación de celdas lógicas por fila y columna.

Fabricantes de otras marcas disponen de herramientas similares y generan reportes del mismo tipo. La diferencia principal se encontrará en el concepto que cada fabricante asigna a la unidad básica constituyente de la *FPGA*. Esta definición va de la mano de la arquitectura particular que difiere en el caso de distintos fabricantes y aún entre modelos del mismo fabricante. Por ejemplo, en el caso de *Xilinx*, la arquitectura básica también tiene la forma de un arreglo de dos dimensiones de bloques lógicos interconectados.

La primer familia introducida en el mercado por esta compañía es la de la serie *XC2000* a la que luego siguieron nuevas generaciones y, entre ellas, la más popular fue la serie *XC4000*. La arquitectura de esta familia se apoya en un Bloque Lógico Programable (*CLB*) que contiene *3 LUT´s*: *2 LUT´s* de 4 entradas y una tercera *LUT* de dos entradas que puede usarse en combinación con las anteriores. Esto permite generar funciones lógicas de hasta nueve entradas. Como en el caso de los *LE´s* de *Altera*, cada *CLB* posee registros pero el número de los mismos es de 2 *FF´s*. Las implementaciones en *FPGA* realizadas sobre esta familia de dispositivos se miden en cantidad de *CLB´s* en términos de área [WEA02], [VU03].

Por su parte la serie *Virtex* de *Xilinx* (familias *Virtex*, *Virtex-E*, *Spartan II* y *Spartan IIE*) define una unidad básica distinta, denominada *slice*, que consta de dos *LUT´s* de 4 entradas. En estas familias, dos *slices* y *4 FF´s* determinan una *CLB*, que además posee lógica de *carry* y multiplexado.

Estas diferencias de arquitecturas y su respectiva traducción en la cantidad de bloques utilizados en la síntesis de circuitos deben ser tenidas en cuenta ya que no pueden compararse directamente implementaciones de un mismo circuito sintetizadas sobre chips de diferentes fabricantes o de diferentes familias. Lo que sí es comparable en términos de área es la cantidad de bloques de memoria utilizados en distintos diseños, medidos como bits de almacenamiento.

Como se mencionó anteriormente, existen otras medidas de la calidad de un cifrador. Una de las figuras de mérito más utilizadas es la de velocidad, denominada por muchos investigadores *throughput*, que se mide como la cantidad de bits cifrados por el circuito en una unidad de tiempo y se encuentra muy relacionada con las características de *unrolling* y *pipelining* elegidas al enfrentar el diseño [ELB01].

Como generalmente interesa también maximizar la velocidad de procesamiento que soporta el circuito y minimizar el área ocupada por este, la relación $[\text{velocidad} / \text{área}]_{\text{Máx}}$ suele ser un parámetro bastante referenciado en la presentación de resultados [HOD04], [WEA02].

Otro parámetro mencionado por varios autores es la capacidad del circuito de compartir recursos entre los procesos de cifrado y descifrado. Es lo que se suele denominar *resource sharing* y medir como la lógica extra precisada para realizar el descifrado respecto de la lógica presente para el cifrado. Se expresa en porcentajes y se suele medir en términos de relación de áreas [GAJ01], [PAN03].

Algunos investigadores utilizan la relación entre *throughput* y cantidad de elementos lógicos básicos que conforman el circuito para comparar implementaciones entre sí. Esta cantidad se conoce como *TPS*, *throughput per slice* [ELB01].

Muchos autores mencionan la *latencia* como otro parámetro del circuito y expresan este parámetro como el tiempo necesario para cifrar un bloque de datos [SHI02], [ASH02], [PAN03], [WEA02], [HOD03], [KHO03]. Generalmente esta característica del circuito se interpreta mejor en la medida de su parámetro de velocidad o *throughput*.

8.2 Resultados

Se presentarán los resultados del diseño presentando los aspectos más relevantes de los archivos de reporte generados por la herramienta utilizada. En general se presentarán los resultados del reporte final, generado desde la simulación del circuito completo.

El reporte comienza con un resumen desde el punto de vista del dispositivo. Esta sección permite observar en forma de un vistazo rápido las características más importantes del cifrador desde el punto de vista de recursos utilizados. Aquí se detallan cantidad de pines de entrada, salida y bidireccionales en uso, la cantidad de bits de memoria ocupados y su relación respecto del total del dispositivo y la cantidad de celdas lógicas utilizadas y su relación con el total. Esta información se ha consignado en la Tabla 8.1.

Dispositivo	Input Pins	Output Pins	Bidir Pins	Memoria Bits	Memoria Utilizada %	LCs	LCs Utilizados %
EPF10K20TC144-3	15	87	0	4480	36 %	460	39 %

Tabla 8.1. Resumen del dispositivo

De la inspección de la Tabla 8.1 resalta la gran diferencia entre la cantidad de pines de entrada y de salida. En este sentido cabe la aclaración que la información consignada como pines de entrada es fidedigna pero aquella que representa los pines de salida no lo es debido a la gran cantidad de señales monitoras que hubo que adicionar al circuito original para poder observar el comportamiento de diversas partes del mismo.

Todos los bloques de memoria del circuito se han creado sobre los bloques *EAB´s* que incluye la *FPGA* para este tipo de propósito. De esta manera se obtiene en la Tabla 8.2 el porcentaje del total que representa cada componente utilizado.

Componente	Descripción	Comentario	Cantidad de bits Utilizados	% sobre el disponible	% sobre el cifrador
U_1	RAM 16x8	Registro de entrada	128	1,03	2,85
U_3	RAM 256x8	Ramkey	2048	16,45	45,71
U_4	RAM 16x8	Registro por round	128	1,03	2,85
U_8	ROM 256x8	Caja S	2048	16,45	45,71
U_{10}	RAM 16x8	Registro intermedio	128	1,03	2,85
Cifrador Completo			**4480**	**36%**	**100%**

Tabla 8.2. Distribución de memoria

En el mapa de distribución de memoria ofrecido por la Tabla 8.2 se destaca que algo más del 90% de la memoria consumida se utiliza en partes iguales en la Caja S y en el almacenamiento de la clave y sus correspondientes subclaves.

En este último caso el bloque de memoria no llega a ocuparse plenamente puesto que sólo es necesario almacenar *176* palabras de *8* bits. De este modo, un porcentaje de esta memoria, el *31,25*%, queda sin utilizarse. La ventaja respecto del caso de calcular la clave al mismo tiempo que se realiza el cifrado es que no existirían diferencias en recursos utilizados si se pretendiera utilizar el circuito adaptado como descifrador. En este caso las claves se utilizan en orden inverso a su generación para el cifrado y la re-utilización de componentes entre ambos procesos sería del 100%. Por otro lado, un cifrador con procesamiento interno de la clave, para ser usado como descifrador, primero debería generar todas las subclaves, pues a partir de la obtención de la última recién es posible iniciar el proceso de descifrado, adicionando en este sentido una gran latencia al proceso inverso.

Se destaca que la memoria utilizada para las Caja S es la mínima posible por dos motivos particulares de la elección de diseño. El primero se refiere al bus interno de *8* bits. Este flujo de datos interno impone la cantidad de memoria pues la consulta de la misma se realiza sobre la base de

bytes. El segundo se refiere al objetivo inicial de este diseño: la generación de un cifrador de mínima área. Por otra parte el componente no puede re-utilizarse si el circuito se usara como descifrador puesto que se precisaría otro bloque de similares características para almacenar la tabla correspondiente a la transformación inversa. De todos modos, el agregado de otra memoria de similares características es posible debido al bajo porcentaje total utilizado.

El resto de los componentes de memoria se utilizan con el fin de agregar al diseño registros de datos intermedios y cierto grado de procesamiento paralelo ó *pipelining* interno. Estas memorias no representan un gran consumo de recursos, apenas el *8,55*% del total. Su reutilización es completa para el mismo circuito utilizado como descifrador.

En términos de elementos lógicos se usa menos del *40*% de los recursos disponibles en la *FPGA* designada como objetivo del diseño. Es interesante observar cómo se reparten estos elementos entre los diversos componentes del circuito. Los datos presentados en la Tabla 8.3 se obtuvieron de los reportes individuales de los componentes mencionados. Existe una diferencia de *4* elementos lógicos con respecto a la versión completa si se efectúa la suma de los *LC´s*. La diferencia es debida a una muy probable eliminación de redundancias en la síntesis final, razón por la cual también baja el porcentaje de utilización de la *FPGA*. Por este motivo, el cálculo de porcentaje de ocupación sobre el cifrador de cada uno de los componentes, presentado en la última columna de la Tabla 8.3, se ha calculado en base a la ocupación de elementos lógicos por componente.

Se puede observar en esta Tabla que la Unidad de Control es el dispositivo más costoso en términos de área, consumiendo el *30*% de los recursos asignados al cifrador. El otro componente comparablemente significativo en estos términos es el encargado de ejecutar la transformación *MixColumns()*.

Componente	Descripción	Comentario	LC´s usados	% sobre disponible	% sobre el cifrador
U_0	rxnibble3	Interfaz	62	10	13,47
U_2	Mux 3x8	Multiplexor realimentación	32	5	6,95
U_5	Mux 2x4	Multiplexor direcciones U_4	4	---	0,87
U_6	Mux 2x8	Multiplexor direcciones U_3	16	2	3,47
U_9	shift	Direcciones de U_{10}	4		0,87
U_{11}	Mux 2x4	Direcciones de U_{10}	4		0,87
U_{12}	Reg32mix8x8	MixColumns()	160	27	34,78
U_{13}	Round1	Unidad de Control	174	30	37,82
Cifrador Completo			**460**	**39%**	**100%**

Tabla 8.3 Reportes de Componentes Individuales

Vale la pena aclarar que la Unidad de Control fue sometida a varios procesos de ajuste. Inicialmente se diagramó como una máquina de estados desarrollada completamente y de este modo consumía muchos más recursos en términos de elementos lógicos. La versión final consignada en este informe, es una reducción de la original. La máquina de estados reducida a su expresión mínima ocupa casi el *50*% menos que la original en términos de área. Se logró reducir el número de *LC´s* utilizados de *305* en la versión original a *174* como se muestra en la Tabla 8.3.

Se podría utilizar esta Unidad de Control para el descifrado de datos si dicha operación se realizara como la de un cifrador equivalente. En este caso la máquina pasaría por la misma cantidad de estados y cada uno de ellos conservaría su funcionalidad. La principal diferencia consistiría en el orden inverso de lectura de la memoria que almacena la clave y, probablemente, en algún ajuste de tiempos que permitiera sincronizar las operaciones apropiadamente. Seguramente la misma máquina utilizada para los fines conjuntos de cifrado y descifrado aumentaría su tamaño en término de *LC´s* pero no llegaría a doblarlo debido a la funcionalidad común mencionada.

En cuanto al componente U_{12} para la realización de *MixColumns()*, incluye internamente otros *3* componentes: un registro de entrada para transformar el camino interno de *8* bits en un bus de *32* bits y poder de este modo operar sobre cada columna, un componente para realizar la operación sobre la columna propiamente dicha y un registro de salida para reconvertir el camino de los datos a un bus de *8* bits. De los tres componentes mencionados los más costosos en términos de área son los registros internos de *32* bits que prácticamente consumen todos los *LC´s* destinados al componente final, ya que la síntesis final de cada uno de estos registros reporta una cantidad de *LC´s* en el orden de las *60*

unidades. El componente equivalente para el descifrado de datos incluiría los mismos registros pero debería modificarse la parte de la operación en sí puesto que la matriz equivalente de descifrado utiliza otras constantes. De este modo el componente se reutilizaría en el descifrado pero incluyendo otro circuito interno, similar al de cifrado, para realizar la operación por columnas. En este sentido aumentaría su tamaño en un orden cercano al 30%.

El componente responsable de la generación de la operación *ShiftRows()* es despreciable en términos de gasto de celdas lógicas. En esta operación la utilización de recursos se mide más apropiadamente en cantidad de bits de memoria, como sucede con *SubBytes()*. En el caso de incluir en el mismo circuito la operación de descifrado, el componente debería reemplazarse por otro que permitiera la rotación en sentido inverso por cada fila del la matriz de estado. En términos de área este componente sería enteramente similar al presentado en la Tabla y, por lo tanto no representaría un consumo importante de área.

Los demás componentes que conforman el núcleo del cifrador son enteramente reutilizables en la operación de descifrado y no representan un impacto significativo en términos del área total del circuito.

El reporte del circuito terminado también presenta una descripción más detallada del uso de los recursos disponibles dentro del dispositivo. Esta información se despliega en forma de una tabla que muestra el número de celdas lógicas, pines I/O (incluyendo los dedicados), expansores compartibles e interconexiones externas. Toda la información se consigna en forma de relación o porcentaje de recursos disponibles en el dispositivo. La misma información se consigna para los *EAB´s*. De este modo se puede obtener un mapa interno completo del dispositivo en cuanto a la distribución de los recursos.

Como se mencionó anteriormente, una de las figuras de mérito más utilizadas para medir la calidad del cifrador es la de velocidad ó *throughput*, medida como la cantidad de bits cifrados por el circuito en una unidad de tiempo. La Tabla 8.4 presenta un resumen de tiempos de procesamiento para el cifrador diseñado. En la misma se incluyen los ciclos de reloj que le lleva al cifrador realizar la

ronda inicial de suma EXOR de los datos con la clave. En realidad este tiempo no influye sobre la capacidad de procesamiento del cifrador puesto que surge de la suposición de una velocidad de entrada para los datos. Esta carga y la operación podrían haberse realizado a la velocidad de reloj, consumiendo de este modo 16 ciclos solamente.

Se puede observar una diferencia de tiempos entre las rondas intermedias y la final. Esto se debe a la ausencia de la Transformación *MixColumns()* al final del proceso de cifrado. En términos generales una ronda completa consume *48* ciclos de reloj. La incidencia de cada transformación sobre dicho consumo no se detalla en la tabla. Estos valores de incidencia se podrían estimar respecto del total en el caso en que la operación no existiera ya que existe un gran grado de simultaneidad entre diversas operaciones. Por ejemplo, la consulta a la caja S consume un ciclo por cada byte ingresado, pero a la vez que se obtiene el byte transformado se lo escribe en una memoria temporal afectándolo al mismo tiempo por la operación de rotación por filas. La lectura de dicha memoria permite disponer del conjunto original de bytes afectado por la operación mencionada. De este modo, a los *18* ciclos de iniciada la ronda se dispone del primer byte saliente de la transformación *ShiftRows()*. La lectura de esta memoria temporal significa la preparación subsiguiente para la operación sobre cada columna de la matriz de estado. En este sentido la operación *MixColumns()* sufre de una latencia inicial de *10* ciclos debido a la carga de los primeros cuatro bytes componentes de la columna, la realización de la operación en sí y, finalmente la disposición de los cuatro bytes en forma seriada. De este modo, *28* ciclos luego de ingresado el primer dato, se obtiene el primer byte de salida de la Transformación *MixColumns()*. A partir de ese momento se puede proceder a realizar la operación final de sumar cada byte obtenido con la subclave correspondiente, este último paso consumirá 16 ciclos y se aprovechará este tiempo para almacenar el resultado en memoria temporal, listo para su lectura como primer paso de la ronda siguiente. *48* ciclos transcurren desde el ingreso del primer byte al núcleo del cifrador y la salida del primer byte cifrado.

Número de Ronda	Ciclos de Clock	Comentario
Inicial	253 = 10,144μseg * 25 MHz	Carga de datos a velocidad elegida (cercana a 10 Mbps). Operación EXOR con la clave original.
1 a 9	48	Se realizan las 4 Transformaciones
10	38	No se realiza la Transformación MixColumns()

Tabla 8.4 Detalle por round de latencia del circuito

En el caso de la última ronda, el tiempo de 18 ciclos transcurridos desde el inicio de la misma hasta la aparición del primer byte resultante de la Transformación *ShiftRows()* se mantiene. Un ciclo después, debido al retardo del multiplexor de entrada, dicho byte se suma con el primero de la subclave de esta ronda. *18* ciclos más tarde se obtiene el primer byte resultante del cifrado del conjunto original de datos. En total esta ronda consume *10* ciclos de reloj menos que los anteriores.

Debido a la elección adoptada en la forma de ingresar los datos se puede considerar la ronda inicial como parte de latencia de ingreso y el tiempo de cifrado total se puede entonces calcular considerando el tiempo total de procesamiento de todas las rondas, a excepción de la primera. La relación resultante sería:

$$T_{encrypt} = (48x9 + 38)xT_{clock}$$

$$T_{encrypt} = (48x9 + 38)x\frac{1}{25MHz} = 18.8\mu s$$

$$Throughput = \frac{128bits}{18.8\mu s} = 6.8Mbps$$

Esta es la verdadera capacidad de procesamiento del cifrador. A su vez, si se considerara la ronda inicial a la velocidad de los datos de entrada superior a 10Mbps para una aproximación a funcionamiento con una interfaz básica, el *Throughput* bajaría a $4.42Mbps$. Sin embargo, si se considerara la ronda inicial de ingreso de datos pero a la velocidad del reloj que maneja el circuito se adicionarían sólo *16* ciclos más a la latencia total, resultando de este modo un *Throughput* mejorado respecto del anterior y cercano al valor original: $6.58Mbps$.

La comparación de los resultados en términos de área y velocidad con otras implementaciones conocidas se presenta en la Tabla 8.5. Se vuelcan sobre dicha tabla sólo implementaciones que han usado como objetivos de síntesis dispositivos de Altera.

La primera cuestión que resalta de la comparación es la diferencia en términos de área en cuanto a la selección de un camino interno para los datos. La decisión de trabajar internamente en el cifrador con mayor cantidad de bits al mismo tiempo produce los mejores resultados en términos de velocidad con la consiguiente penalidad en la utilización de recursos. Los resultados correspondientes al bloque mínimo de *16* bits del diseño de *Fischer* son los más cercanos a este trabajo. En este sentido, la decisión propia de trabajar con *8* bits por vez produce una mejora de más de tres veces y media en lo que respecta a los recursos ocupados. La penalidad se puede observar en los resultados obtenidos en lo que respecta a la velocidad del circuito, casi cuatro veces y medio menor.

Diseño	Bits de Memoria	LE´s	Velocidad (Mbps)	Familia
Panato [PAN03] Bloques de 32 bits y 128 bits	16384	2114	182	Altera Acex1K
Panato [PAN03] Bloques de 32 bits y 128 bits	0	4057	256	Altera Cyclone
Mroczkowski [MRO01] Bloques de 128 bits	40960	1032	268	Altera Flex 10K250A
Fischer Fast [FIS00] Bloques de 128 bits	65536	1585	232,7	Altera Flex 10KE
Fischer Fair [FIS00] Bloques de 64 bits	32768	1604	121,9	Altera Flex 10KE
Fischer Minimum [FIS00] Bloques de 16 bits	4096	1673	31,6	Altera Flex 10KE
Este diseño **Bloques de 8 bits**	4480	460	6,8	Altera Flex EPF10K20TC144-3

Tabla 8.5. Comparación con otras implementaciones

Sin embargo, el resultado más interesante de este trabajo es que el diseño podría tener como objetivo de síntesis el dispositivo de menor área de la familia Flex 10K. Se trata de EPF10K10 que cuenta con *6144* bits de memoria y sólo *576* celdas lógicas. En este caso los demás diseños presentados en la tabla no serían factibles de implementar.

La elección del dispositivo EPF10K20 como objetivo de diseño se debe a su disponibilidad en la placa de desarrollo. Los resultados obtenidos permiten suponer la posibilidad cierta de

implementación de un circuito con doble funcionalidad de cifrado y descifrado, debido al alto grado de reutilización de los componentes que conforman el diseño.

Capítulo 9

Conclusiones

La implementación de cifradores de bloques enfrenta al diseñador con diversas estrategias. La decisión que influye sobre la elección se relaciona con la característica o parámetro de diseño que se desee optimizar. En el caso de diseño de cifradores en hardware, la elección buscará optimizar el área ocupada por el circuito, la velocidad de procesamiento del mismo o la latencia asociada. Cada elección de diseño impone diferentes restricciones sobre la implementación final, existiendo un compromiso particular entre dos parámetros característicos del diseño final: área y velocidad. En este sentido no puede mejorarse uno sin la consiguiente penalización en términos del otro.

Sin embargo, cualquiera sea la estrategia, todas comparten a una estructura básica en cuanto a la organización de los diversos componentes del circuito. Esa estructura básica podrá distinguirse por el reconocimiento de unidades comunes a todos los circuitos cifradores: una unidad de cifrado/descifrado siempre presente, una unidad opcional de tratamiento de la clave que podrá reemplazarse por una unidad de memoria para almacenamiento de la clave original y las subclaves derivadas de la misma, una interfaz para comunicación con el mundo exterior e ingreso de los datos, una interfaz para almacenamiento de los datos cifrados y su envío posterior al exterior y una unidad de control que coordina el funcionamiento de las demás unidades mencionadas. A través de anteriores capítulos se han presentado y detallado estos componentes en el cifrador diseñado.

Por tratarse del diseño de un cifrador simétrico, una de las primeras decisiones al respecto se relaciona con el modo de operación. Básicamente existen dos modos: no realimentado ó ECB y realimentado con sus diversas variantes ya comentadas. En este caso, la selección de un cifrador sin realimentación se basa en la posibilidad de realizar un cifrado secuencial de datos para introducir, de este modo, un grado máximo de procesamiento paralelo que se traduce en mejoras en cuanto a la velocidad de procesamiento. Una de las aplicaciones prácticas de este tipo de circuitos ECB se encuentra en el cifrado de claves de sesión en la fase previa de distribución de la clave.

Otra decisión importante al momento de diseño es el ancho del bus interno de datos que

manejará el circuito. Esta decisión queda restringida principalmente por la cantidad de pines de la FPGA elegida para la fase de síntesis y por la interfaz que poseerá el circuito para su comunicación con el mundo exterior. En este caso, la selección inicial de una interfaz básica y sencilla y la limitación de 144 patas de la FPGA limitaba la elección a valores de bus internos pequeños.

Otro condicionamiento lo constituye la cantidad de cajas S que poseerá el cifrador ya que su número va en directa relación con el ancho del camino interno de datos. La unidad básica de procesamiento del algoritmo Rijndael es de 8 bits y esto determinó la selección original, utilizándose de este modo la menor cantidad de memoria posible para la caja S. La contrapartida de esta elección se da en una penalización de velocidad de procesamiento que se trató de mejorar con la introducción de técnicas de procesamiento paralelo.

La elección de una arquitectura interna encuentra sus variantes más conocidas en el desarrollo de un circuito que implemente varias rondas, en inglés *loop unrolling*, y en la introducción de registros internos para obtener capacidad de procesamiento paralelo. Esta última arquitectura se conoce en inglés como *inner-round pipelining*. También existe una combinación de ambas llamada en inglés *outer-round pipelining*. Debido a los requerimientos iniciales de diseño restringidos a optimizar el consumo de recursos en términos de área, la elección de la arquitectura se basó en el desarrollo de una única ronda, también conocida como arquitectura básica. Se trata de una arquitectura sencilla que funciona de manera iterativa ofreciendo de esta manera requerimientos mínimos en cuanto a área y modestos en lo relativo a velocidad de procesamiento. La introducción de procesamiento paralelo interno sobre esta arquitectura permitió mejorar los parámetros de velocidad. En general, la combinación de modo sin realimentar con procesamiento paralelo interno permite obtener los mejores resultados en la relación [velocidad/área].

Como se ha comentado en otros Capítulos, el diseño del circuito cifrador posee un alto grado de reutilización de componentes respecto de su adaptación para un circuito de descifrado. El porcentaje final de utilización de área avala la posibilidad de incorporar en el mismo chip ambas funcionalidades: cifrado y descifrado. El orden de las operaciones no se ve modificado en el caso de usar un descifrador equivalente. Sólo se verá afectado el procesamiento de la subclave

correspondiente a cada ronda.

El cifrador diseñado con las características mencionadas es un cifrador Rijndael para cifrado de datos cuyos parámetros más sobresalientes son su área reducida y bajo costo. El circuito ideado usa una arquitectura iterativa de 8 bits programada en el lenguaje VHDL, cuyo objetivo de síntesis ha sido dirigido sobre la familia Flex 10K de los chips FPGA de Altera. En particular, la selección del dispositivo Altera Flex EPF10K20TC144-3 se debe a su disponibilidad en la placa de desarrollo.

El cifrador fue sintetizado utilizando la herramienta Altera MAX+PLUS II Version 7.21 Student Edition. La implementación VHDL se realizó sobre una estrategia base de diseño tipo *bottom-up* y una metodología de prueba provista por la herramienta mencionada. La elección del lenguaje de programación se basó en adicionar características de portabilidad al código de tal manera de poder trasladar el diseño a dispositivos de otros vendedores.

El código VHDL fue simulado utilizando los vectores de prueba provistos en el paquete de algoritmo Rijndael que fuera presentado a la FIPS en ocasión de su comparación frente a otros cifradores de similares características. Los resultados son funcionalmente correctos.

La arquitectura seleccionada precisa menos celdas lógicas que otros cifradores y utiliza la menor cantidad posible de bloques de memoria. Utiliza sólo un 39% en término de celdas lógicas y un 36% de bits de memoria disponibles en el dispositivo usado como objetivo de diseño. Se logra de este modo un diseño compacto de sólo 460 elementos lógicos y 4480 bits de memoria. La velocidad de procesamiento ha sido calculada en 6,8Mbps. Tanto la clave como las subclaves se encuentran almacenadas en memoria. El camino crítico incluye componentes mencionados en Capítulos anteriores. La operación más costosa en términos de latencia es la transformación *MixColumns()*. El período mínimo de reloj es función del tiempo de acceso a las memorias utilizadas y ha sido fijado en el valor correspondiente al reloj externo a la FPGA que se encuentra en la placa de de desarrollo.

El análisis de los resultados y la comparación con otras arquitecturas conocidas que usan dispositivos de la Familia Altera como objetivo de síntesis permite posicionar este diseño como el más

compacto en términos de utilización de recursos. Esta característica abre la posibilidad de su implementación sobre los dispositivos de menores recursos de ésta u otras familias, otorgando de este modo al diseño presentado efectividad en cuanto a costo.

El trabajo a futuro debería concentrarse en la mejorar los resultados de velocidad para la aumentar los valores de capacidad de procesamiento obtenidos. Una manera de lograrlo podría presentarse en estudiar la posibilidad de trabajar con un camino interno de datos mixto con valores variables entre 16 a 32 bits según la operación enfrentada. Esto adicionaría pines al circuito original en lo que respecta a la intercomunicación con el mundo externo e implicaría un aumento en la cantidad de recursos de memoria utilizados.

Otra sugerencia a futuro contemplaría la posibilidad de diseñar cifrador y descifrador en el mismo dispositivo FPGA de bajo volumen. Los resultados obtenidos en este trabajo respecto de los porcentajes de ocupación de área presentan esta posibilidad como sumamente factible.

Apéndice A

Código VHDL

Unidad de Control *round1*

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

ENTITY round1 IS
        PORT
        (
                clk                     : IN    STD_LOGIC;
                reset           : IN    STD_LOGIC;
                habil           : IN    STD_LOGIC;
                q                       : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);
                qbaja           : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0);
                selecmux3x8     : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0);
                selecmux2x8     : OUT   STD_LOGIC;
                selecmux2x4     : OUT   STD_LOGIC;
                weout           : OUT   STD_LOGIC;
                we                      : OUT   STD_LOGIC;
                fin                     : OUT   STD_LOGIC;
                ronda   : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0);
                enable          : OUT   STD_LOGIC;
                startreg  : OUT   STD_LOGIC);
END round1;

ARCHITECTURE a OF round1 IS
        TYPE estado is (roundini,round1, round10,roundfin);
        SIGNAL cnt: estado:=roundini;
        SIGNAL auxselecmux2x4 : STD_LOGIC;
        SIGNAL auxq     : STD_LOGIC_VECTOR (7 downto 0);
        SIGNAL auxqbaja, auxronda       : STD_LOGIC_VECTOR (3 downto 0);
        SIGNAL auxwe, auxenable, auxfin : STD_LOGIC;
        SIGNAL auxfinronda1 : STD_LOGIC;
        SIGNAL intronda : INTEGER RANGE 0 TO 15;



BEGIN

selecmux2x4 <= auxselecmux2x4;
q <= auxq;
qbaja<= auxqbaja;
we <= auxwe;
enable <= auxenable;
fin <= auxfin;
ronda <= auxronda;

PROCESS (clk, reset)
BEGIN
                IF reset = '1' THEN
                        cnt <= roundini;
```

```
                ELSIF (clk'EVENT AND clk = '1') THEN
                        CASE cnt IS
                                WHEN roundini =>
                                        IF habil ='1' THEN
                                        cnt <= round1;
                                        END IF;

                                WHEN round1 =>
                                        IF   (auxfinronda1 = '1' AND intronda < 9) THEN
                                                cnt <= round1;

                                        ELSIF (auxfinronda1 = '1' AND intronda = 9) THEN
                                                cnt <= round10;
                                        END IF;

                                WHEN round10 =>

                                        IF auxfinronda1 = '1'THEN cnt <= roundfin;
                                        END IF;

                                WHEN roundfin =>

                                        IF auxenable = '1'THEN cnt <= roundini;
                                        END IF;


                        END CASE;
                END IF;
        END PROCESS;



PROCESS(clk, reset)

VARIABLE cuenta: INTEGER RANGE 0 TO 255;
VARIABLE address : INTEGER RANGE 0 TO 255;
VARIABLE addressbaja : INTEGER RANGE 0 TO 15;

BEGIN



IF reset = '1' THEN
        auxenable <= '1';
        auxq <= "00000000";
        auxqbaja <= "0000";
        selecmux2x8 <= '0';
        selecmux3x8 <= "00";
        auxselecmux2x4 <= '0';
        weout <= '0';
        auxwe <= '0';
        startreg <= '0';
        auxfinronda1 <= '0';
        cuenta:= 0;
        address:=0;
        addressbaja:=0;
```

```
ELSIF (clk'EVENT AND clk = '1') THEN
                CASE cnt IS
                                        WHEN roundini =>
                                                auxenable <= '1';
                                                selecmux2x8 <= '0';
                                                selecmux3x8 <= "00";
                                                auxselecmux2x4 <= '0';
                                                weout <= '0';
                                                auxwe <= '1';
                                                startreg <= '0';
                                                auxfinronda1 <= '0';
                                                cuenta:= 0;
                                                address:=0;
                                                addressbaja:=0;
                                                auxq <= "00000000";
                                                auxqbaja <= "0000";

        WHEN round1 =>

                                                auxenable <= '0';
                                                selecmux2x8 <= '1';
                                                selecmux3x8 <= "01";
                                                auxselecmux2x4 <= '1';
                                                address := address +1;
                                                addressbaja := addressbaja +1;
                                                auxq<= CONV_STD_LOGIC_VECTOR(address,8);
                                                auxqbaja<=
CONV_STD_LOGIC_VECTOR(addressbaja,4);
                                                cuenta := cuenta+1;

                                                IF cuenta<16 THEN
                                                auxwe <= '0';
                                                ELSIF ((cuenta > 29) AND (cuenta < 46)) THEN -- era 27
                                                auxwe <= '1';
                                                ELSE auxwe <= '0';
                                                END IF;

                                                IF ((cuenta >2) AND (cuenta < 19)) THEN
                                                weout <= '1';
                                                ELSE weout <='0';
                                                END IF;

                                                IF ((cuenta>23) AND (cuenta <46)) THEN
                                                startreg <= '1';
                                                ELSE startreg <= '0';
                                                END IF;

                                                IF cuenta = 47 THEN
                                                auxfinronda1 <= '1';
                                                        cuenta:= 255;
                                                        address:=0;
                                                        --addressbaja := 0;
                                                ELSE auxfinronda1 <= '0';
                                                END IF;

                                                IF cuenta = 27 THEN --  era 27
```

```
                                        CASE intronda IS

                                                WHEN 0 => address := 15;
                                                WHEN 1 => address := 31;
                                                WHEN 2 => address := 47;
                                                WHEN 3 => address := 63;
                                                WHEN 4 => address := 79;
                                                WHEN 5 => address := 95;
                                                WHEN 6 => address := 111;
                                                WHEN 7 => address := 127;
                                                WHEN 8 => address := 143;
                                                WHEN OTHERS =>

                                        END CASE;
                                END IF;


                WHEN round10 =>
                                        auxenable <= '0';
                                        selecmux2x8 <= '1';
                                        auxselecmux2x4 <= '1';
                                        address := address+1;
                                        addressbaja := addressbaja+1;
                                        auxq<= CONV_STD_LOGIC_VECTOR(address,8);
                                        auxqbaja<=
CONV_STD_LOGIC_VECTOR(addressbaja,4);

                                        cuenta := cuenta+1;

                                        IF cuenta<16 THEN
                                        auxwe <= '0';
                                        ELSIF ((cuenta > 20) AND (cuenta < 37)) THEN -- era 27
                                        auxwe <= '1';
                                        ELSE auxwe <= '0';
                                        END IF;
                                        IF cuenta=19 THEN
                                        addressbaja:=15;
                                        END IF;
                                        IF ((cuenta >2) AND (cuenta < 19)) THEN
                                        weout <= '1';
                                        ELSE weout <='0';
                                        END IF;
                                        IF  (cuenta > 19) THEN
                                        selecmux3x8 <= "10";
                                        ELSE selecmux3x8 <= "01";
                                        END IF;



                                        IF cuenta=37 THEN
                                        auxfinronda1 <= '1';
                                        cuenta:= 255;
                                        ELSE auxfinronda1 <= '0';
                                        END IF;

                                        IF cuenta = 18 THEN
                                        address := 159;
                                        END IF;
```

```
                WHEN roundfin =>

                                                IF cuenta = 16 THEN
                                                --auxenable <= '1';
                                                auxfin<= '1';
                                                END IF;


                                                --address := address+1;
                                                --addressbaja := addressbaja+1;
                                                --auxq<= CONV_STD_LOGIC_VECTOR(address,8);
                                                --auxqbaja<=
CONV_STD_LOGIC_VECTOR(addressbaja,4);

                                                --cuenta := cuenta+1;


                END CASE;

END IF;
END PROCESS;

PROCESS(clk)
VARIABLE sum: INTEGER RANGE 0 TO 15;
BEGIN
        IF (clk'EVENT AND clk = '1') THEN
                IF(auxfinronda1='1') THEN
                sum:=sum+1;
                intronda <= sum;

                auxronda <= CONV_STD_LOGIC_VECTOR(CONV_INTEGER(sum), 4);
                END IF;
        END IF;
END PROCESS;

END a;
```

Unidad de Cifrado *encrip2*

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;


ENTITY encrip2 IS
        PORT
        (
                datain          : IN    STD_LOGIC_VECTOR(3 DOWNTO 0);
                data            : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
                clk                     : IN    STD_LOGIC;
                Strobe          : IN    STD_LOGIC;
                reset           : IN    STD_LOGIC;
                setcuenta       : OUT STD_LOGIC ;
                exorout         : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);
                ronda   : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0);
                monitorstartreg : OUT STD_LOGIC ;
                monitorweout    : OUT STD_LOGIC ;
```

```
                monitoraux4     : OUT   STD_LOGIC_VECTOR( 7 DOWNTO 0);
                monitoraux8     : OUT   STD_LOGIC_VECTOR( 7 DOWNTO 0);
                addressram      : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0);
                ACK                     : OUT   STD_LOGIC;
                monitorb        : OUT   STD_LOGIC;
                monitorc        : OUT   STD_LOGIC;
                byteready       : OUT   STD_LOGIC;
                monitora        : OUT   STD_LOGIC_VECTOR( 7 DOWNTO 0);
                monitorauxramout        : OUT   STD_LOGIC_VECTOR( 7 DOWNTO 0);
                monitorauxrom   : OUT   STD_LOGIC_VECTOR( 7 DOWNTO 0);
                monitoraux1a: OUT       STD_LOGIC_VECTOR( 3 DOWNTO 0);
                monitorauxkey: OUT      STD_LOGIC_VECTOR( 3 DOWNTO 0);
                monitordata     : OUT   STD_LOGIC_VECTOR( 7 DOWNTO 0);
                q                       : OUT   STD_LOGIC_VECTOR( 7 DOWNTO 0));
END encrip2;

ARCHITECTURE a OF encrip2 IS


        COMPONENT rxnibble3
        PORT(
                datain          : IN    STD_LOGIC_VECTOR(3 DOWNTO 0);
                clk                     : IN    STD_LOGIC;
                Strobe          : IN    STD_LOGIC;
                reset           : IN    STD_LOGIC;
                enable          : IN    STD_LOGIC;
                we                      : OUT   STD_LOGIC;
                habil           : OUT   STD_LOGIC;
                setcuenta       : OUT   STD_LOGIC;
                address         : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0);
                ACK                     : OUT   STD_LOGIC;
                byteready       : OUT   STD_LOGIC;
                q                       : OUT   STD_LOGIC_VECTOR( 7 DOWNTO 0));
END COMPONENT;

        COMPONENT ram16x8
                PORT (
                address         : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
                we              : IN STD_LOGIC ;
                inclock         : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                q               : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
        );
END COMPONENT;

COMPONENT mux2x4
PORT ( bus1 : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
                bus2 : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
                clockR: IN STD_LOGIC;
                busout : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
                selec : IN STD_LOGIC);
END COMPONENT;

COMPONENT mux3x8

PORT ( bus0 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                bus1 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                bus2 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                clockR: IN STD_LOGIC;
                busout : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
```

```
                selec : IN STD_LOGIC_VECTOR(1 DOWNTO 0));
END COMPONENT;

COMPONENT ramkey
        PORT(
                address         : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                we              : IN STD_LOGIC ;
                inclock         : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                q               : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
        );
END COMPONENT;

COMPONENT mux2x8


PORT ( bus0 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                bus1 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                clockR: IN STD_LOGIC;
                busout : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
                selec : IN STD_LOGIC);
END COMPONENT;

COMPONENT rom256x8
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                outclock        : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
        );
END COMPONENT;

COMPONENT shift
        PORT (entrada   : IN  STD_LOGIC_VECTOR (3 DOWNTO 0);
                salida  : OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
END COMPONENT;

COMPONENT reg32mix8x8
        PORT(
                bytein  : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
                byteclk         : IN  STD_LOGIC;
                startreg        : IN  STD_LOGIC;
                byteout : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;

COMPONENT round1
        PORT
        (
                clk                     : IN    STD_LOGIC;
                reset           : IN    STD_LOGIC;
                habil           : IN    STD_LOGIC;
                q                       : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);
                qbaja           : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0);
                selecmux3x8     : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0);
                selecmux2x8     : OUT   STD_LOGIC;
                selecmux2x4     : OUT   STD_LOGIC;
                weout           : OUT   STD_LOGIC;
                we                      : OUT   STD_LOGIC;
                fin                     : OUT STD_LOGIC;
                ronda   : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0);
```

```
                enable          : OUT    STD_LOGIC;
                startreg  : OUT    STD_LOGIC);
END COMPONENT;

SIGNAL aux0, aux7, auxinv, auxi, auxil, auxclk, auxenable, auxweout, auxstartreg, auxfin : STD_LOGIC;
SIGNAL aux1,aux1a, auxcontab, auxresta, auxaddressram, auxmux, auxkey, auxronda:
STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL aux5: STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL aux2, aux3, aux4, aux6, aux8, aux9,  aux1e, auxconca, auxdata, auxrom,
                auxromout, auxramout, auxsalida: STD_LOGIC_VECTOR(7 DOWNTO 0);

BEGIN

u0: rxnibble3
        PORT MAP(datain => datain, clk => clk, Strobe => Strobe, reset =>auxfin,
        enable => auxenable, we => aux0, habil => auxclk, setcuenta => setcuenta,
        address => aux1, ACK => ACK, byteready => byteready,
        q => aux2);

u1: ram16x8
        PORT MAP(address => aux1, we => aux0, inclock => clk, data => aux2, q =>aux3);

u2:mux3x8
        PORT MAP( bus0 => aux3, bus1 => auxsalida, bus2 => auxramout,
                clockR => clk, busout =>aux4, selec => aux5);

u3: ramkey
        PORT MAP(address => aux6, we => aux7, inclock => clk, data => data, q => aux8);

u4: ram16x8
        PORT MAP(address => auxkey, we => auxinv, inclock => clk, data => aux9, q =>auxrom);

u5: mux2x4
        PORT MAP(bus1 => aux1, bus2 => auxcontab, clockR =>clk, busout => aux1a, selec => auxi);

u6: mux2x8
        PORT MAP(bus0 => aux1e, bus1 => auxconca, clockR => clk, busout => aux6, selec => auxil);

U8: rom256x8
        PORT MAP(address => auxrom, outclock => clk, q =>auxromout       );
u9: shift
        PORT MAP(entrada =>auxresta, salida => auxaddressram);

u10: ram16x8
        PORT MAP(address => auxmux, we => auxweout, inclock => clk,
        data => auxromout, q =>auxramout);

u11: mux2x4
        PORT MAP(bus2 => auxaddressram, bus1 => auxresta, clockR =>clk,
        busout => auxmux, selec => auxweout);

u12: reg32mix8x8
        PORT MAP(bytein         => auxramout, byteclk => clk, startreg =>auxstartreg,
        byteout => auxsalida);

u13: round1
        PORT MAP (clk => clk, reset       => reset, habil => auxclk, q => auxconca, qbaja => auxcontab,
                selecmux3x8 => aux5, selecmux2x8 => auxil, selecmux2x4 => auxi, weout => auxweout,
                we => auxinv, ronda => auxronda, enable => auxenable, startreg => auxstartreg, fin=> auxfin);
```

```
--auxresta <= CONV_STD_LOGIC_VECTOR((CONV_INTEGER(aux1a))-1, 4);
--auxcontab <= auxconca(3 DOWNTO 0);

aux1e<= "0000"&aux1;

aux9 <= aux8 xor aux4;
aux7 <= '0';

q<= auxsalida;
exorout<= aux9;
addressram<= auxmux;
monitora<=auxconca;
monitorb<= auxinv;
monitorc<=auxclk;
monitordata<=auxromout;
monitoraux4<= aux4;
ronda <= auxronda;

auxkey <= CONV_STD_LOGIC_VECTOR((CONV_INTEGER(aux1a))+3, 4) WHEN auxstartreg='1' ELSE
aux1a;

PROCESS
BEGIN
        IF(auxronda/=9) THEN
                auxresta <= CONV_STD_LOGIC_VECTOR((CONV_INTEGER(aux1a))-1, 4);
        ELSIF ((auxronda=9) AND (auxinv='1'))  THEN
                auxresta <= CONV_STD_LOGIC_VECTOR((CONV_INTEGER(aux1a))+3, 4);
        ELSE auxresta <= CONV_STD_LOGIC_VECTOR((CONV_INTEGER(aux1a))-1, 4);
        END IF;
END PROCESS;

monitoraux8<= aux8;
monitoraux1a<= aux1a;
monitorstartreg <= auxstartreg;
monitorweout <= auxweout;
monitorauxramout <= auxramout;
monitorauxrom <= auxrom;
monitorauxkey <= auxkey;
END a;
```

Unidad de Ingreso de Datos *rxnibble3*

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

ENTITY rxnibble3 IS
        PORT
        (
                datain          : IN    STD_LOGIC_VECTOR(3 DOWNTO 0);
                clk                     : IN    STD_LOGIC;
                Strobe          : IN    STD_LOGIC;
                reset           : IN    STD_LOGIC;
                enable          : IN    STD_LOGIC;
                we                      : OUT   STD_LOGIC;
                habil           : OUT   STD_LOGIC;
                setcuenta       : OUT   STD_LOGIC;
```

```
                address         : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0);
                ACK                     : OUT   STD_LOGIC;
                byteready       : OUT   STD_LOGIC;
                q                       : OUT   STD_LOGIC_VECTOR( 7 DOWNTO 0));
END rxnibble3;

ARCHITECTURE a OF rxnibble3 IS
        TYPE estado is (a, b, c, d, e, f, g, h,i,j);
        SIGNAL cnt: estado:=a;
        SIGNAL contador: INTEGER RANGE 0 TO 15;
        SIGNAL habilaux : STD_LOGIC;
BEGIN
habil <= habilaux;
PROCESS (clk, reset)
        BEGIN
                IF reset = '1' THEN
                                cnt <= a;
                                contador <= 0;
                                habilaux<='0';
                                setcuenta <='0';

                ELSIF (clk'EVENT AND clk = '1') THEN
                                CASE cnt IS
                                        WHEN a =>
                                                cnt <= b;               --inicializo los ports de salida

                                        WHEN b =>
                                                IF Strobe = '1'AND  enable = '1' THEN
                                                        cnt <= c;
                                                END IF;
                                        WHEN c =>
                                                        -- para aumentar la duracion a dos pulsos
                                                        cnt <= d;
                                        WHEN d =>
                                                        cnt <= e;
                                        WHEN e =>
                                                IF Strobe = '1' AND  enable = '1' THEN
                                                        cnt <= f;
                                                END IF;
                                        WHEN f =>
                                                        -- para aumentar la duracion a dos pulsos
                                                        cnt <= g;
                                        WHEN g =>
                                                        cnt <= h;
                                        WHEN h =>
                                                        IF habilaux ='0' THEN
                                                        cnt <= i;
                                                        ELSE cnt<= j;
                                                        END IF;
                                        WHEN i =>
                                                        cnt <= a;
                                        WHEN j =>
                                                        cnt <= a;

                                END CASE;
                END IF;
        END PROCESS;
PROCESS(cnt, clk)
```

```
BEGIN

IF (clk'EVENT AND clk = '1') THEN

        CASE cnt IS
                WHEN a =>       --inicializo los ports de salida
                        ACK <= '0';
                        byteready <= '0';
                        q <= "00000000";
                        address <= CONV_STD_LOGIC_VECTOR(contador,4);
                        setcuenta <='0';
                WHEN b =>
                        IF Strobe = '1'AND  enable = '1' THEN
                                q(7 DOWNTO 4) <= datain;
                                ACK <= '1';
                        END IF;
                WHEN c =>
                                -- para aumentar la duracion a dos pulsos

                WHEN d =>
                                ACK <= '0';

                WHEN e =>
                        IF Strobe = '1' AND  enable = '1' THEN
                                q(3 DOWNTO 0) <= datain;
                                ACK <= '1';


                        END IF;
                WHEN f =>
                                -- para aumentar la duracion a dos pulsos
                                we <= '1';

                WHEN g =>
                                ACK <= '0';
                                byteready <='1';

                WHEN h =>

                WHEN i =>
                                we <= '0';
                                contador <= contador +1;

                                IF contador = 15 THEN
                                habilaux <= '1';
                                setcuenta <= '1';
                                END IF;
                WHEN j =>
                                we <= '0';
                                contador <= contador +1;

        END CASE;


END IF;
END PROCESS;
END a;
```

Componente ram16x8

```
-- megafunction wizard: %LPM_RAM_DQ%
-- GENERATION: STANDARD
-- VERSION: WM1.0
-- MODULE: LPM_RAM_DQ

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY lpm;
USE lpm.lpm_components.all;

ENTITY ram16x8 IS
	PORT
	(
		address		: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
		we		: IN STD_LOGIC ;
		inclock		: IN STD_LOGIC ;
		data		: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
		q		: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
	);
END ram16x8;


ARCHITECTURE SYN OF ram16x8 IS

	SIGNAL sub_wire0	: STD_LOGIC_VECTOR (7 DOWNTO 0);



	COMPONENT LPM_RAM_DQ
	GENERIC (
		LPM_WIDTH		: POSITIVE;
		LPM_WIDTHAD		: POSITIVE;
		LPM_ADDRESS_CONTROL		: STRING;
		LPM_INDATA		: STRING;
		LPM_OUTDATA		: STRING;
		LPM_FILE		: STRING
	);
	PORT (
			address	: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
			inclock	: IN STD_LOGIC ;
			q	: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
			data	: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
			we	: IN STD_LOGIC
	);
	END COMPONENT;

BEGIN
	q    <= sub_wire0(7 DOWNTO 0);

	LPM_RAM_DQ_component : LPM_RAM_DQ
	GENERIC MAP (
		LPM_WIDTH => 8,
		LPM_WIDTHAD => 4,
		LPM_ADDRESS_CONTROL => "REGISTERED",
		LPM_INDATA => "REGISTERED",
		LPM_OUTDATA => "UNREGISTERED",
		LPM_FILE => "C:\\max2work\\rijn3\\inicia1.mif"
	)
```

```
        PORT MAP (
                address => address,
                inclock => inclock,
                data => data,
                we => we,
                q => sub_wire0
        );


END SYN;
```

Componente mux3x8

```
-- un multiplexor de 3 entradas de 8 bits cada una

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux3x8 IS

PORT ( bus0 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                bus1 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                bus2 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                clockR: IN STD_LOGIC;
                busout : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
                selec : IN STD_LOGIC_VECTOR(1 DOWNTO 0));
END mux3x8;

ARCHITECTURE a OF mux3x8 IS

BEGIN
PROCESS(clockR)

BEGIN
                 IF (clockR'EVENT AND clockR = '1') THEN
                CASE selec IS
                        WHEN "00" => busout <= bus0;
                        WHEN "01" => busout <= bus1;
                        WHEN "10" => busout <= bus2;
                        WHEN OTHERS => null;
                END CASE;

                 END IF;
END PROCESS;
END a;
```

Componente ramkey

```
-- megafunction wizard: %LPM_RAM_DQ%
-- GENERATION: STANDARD
-- VERSION: WM1.0
-- MODULE: LPM_RAM_DQ


LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY lpm;
```

```
USE lpm.lpm_components.all;

ENTITY ramkey IS
	PORT
	(
		address		: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
		we		: IN STD_LOGIC ;
		inclock		: IN STD_LOGIC ;
		data		: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
		q		: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
	);
END ramkey;


ARCHITECTURE SYN OF ramkey IS

	SIGNAL sub_wire0	: STD_LOGIC_VECTOR (7 DOWNTO 0);



	COMPONENT LPM_RAM_DQ
	GENERIC (
		LPM_WIDTH		: POSITIVE;
		LPM_WIDTHAD		: POSITIVE;
		LPM_ADDRESS_CONTROL		: STRING;
		LPM_INDATA		: STRING;
		LPM_OUTDATA		: STRING;
		LPM_FILE		: STRING
	);
	PORT (
			address	: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
			inclock	: IN STD_LOGIC ;
			q	: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
			data	: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
			we	: IN STD_LOGIC
	);
	END COMPONENT;

BEGIN
	q    <= sub_wire0(7 DOWNTO 0);

	LPM_RAM_DQ_component : LPM_RAM_DQ
	GENERIC MAP (
		LPM_WIDTH => 8,
		LPM_WIDTHAD => 8,
		LPM_ADDRESS_CONTROL => "REGISTERED",
		LPM_INDATA => "REGISTERED",
		LPM_OUTDATA => "UNREGISTERED",
		LPM_FILE => "C:\\max2work\\rijn3\\key.mif"
	)
	PORT MAP (
		address => address,
		inclock => inclock,
		data => data,
		we => we,
		q => sub_wire0
	);
END SYN;
```

Componente mux2x4

```
-- un multiplexor de dos entradas de 4 bits cada una

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux2x4 IS

PORT ( bus1 : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
                bus2 : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
                clockR: IN STD_LOGIC;
                busout : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
                selec : IN STD_LOGIC);
END mux2x4;

ARCHITECTURE a OF mux2x4 IS

BEGIN
PROCESS(clockR)

BEGIN
                IF (clockR'EVENT AND clockR = '1') THEN
                CASE selec IS
                        WHEN '0' => busout <= bus1;
                        WHEN '1' => busout <= bus2;
                        WHEN OTHERS => null;
                END CASE;

                END IF;
END PROCESS;
END a;
```

Componente mux2x8

```
-- un multiplexor de 3 entradas de 8 bits cada una

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux2x8 IS

PORT ( bus0 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                bus1 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                clockR: IN STD_LOGIC;
                busout : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
                selec : IN STD_LOGIC);
END mux2x8;

ARCHITECTURE a OF mux2x8 IS

BEGIN
PROCESS(clockR)



BEGIN
                IF (clockR'EVENT AND clockR = '1') THEN
```

```
                CASE selec IS
                        WHEN '0' => busout <= bus0;
                        WHEN '1' => busout <= bus1;
                        WHEN OTHERS => null;
                END CASE;

                 END IF;
END PROCESS;
END a;
```

Componente rom256x8

```
-- megafunction wizard: %LPM_ROM%
-- GENERATION: STANDARD
-- VERSION: WM1.0
-- MODULE: LPM_ROM


LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY lpm;
USE lpm.lpm_components.all;

ENTITY rom256x8 IS
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                outclock        : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
        );
END rom256x8;


ARCHITECTURE SYN OF rom256x8 IS

        SIGNAL sub_wire0        : STD_LOGIC_VECTOR (7 DOWNTO 0);



        COMPONENT LPM_ROM
        GENERIC (
                LPM_WIDTH               : POSITIVE;
                LPM_WIDTHAD                     : POSITIVE;
                LPM_ADDRESS_CONTROL             : STRING;
                LPM_INDATA              : STRING;
                LPM_OUTDATA                     : STRING;
                LPM_FILE                : STRING
        );
        PORT (
                        outclock: IN STD_LOGIC ;
                        address  : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                        q        : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
        );
        END COMPONENT;

BEGIN
        q    <= sub_wire0(7 DOWNTO 0);

        LPM_ROM_component : LPM_ROM
```

```
	GENERIC MAP (
		LPM_WIDTH => 8,
		LPM_WIDTHAD => 8,
		LPM_ADDRESS_CONTROL => "UNREGISTERED",
		LPM_INDATA => "UNREGISTERED",
		LPM_OUTDATA => "REGISTERED",
		LPM_FILE => "C:\\max2work\\rijn4\\inicia.mif"
	)
	PORT MAP (
		outclock => outclock,
		address => address,
		q => sub_wire0
	);



END SYN;
```

Componente reg32mix8x8

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;


ENTITY reg32mix8x8 IS
	PORT(
		bytein	: IN	STD_LOGIC_VECTOR(7 DOWNTO 0);
		byteclk		: IN	STD_LOGIC;
		startreg	: IN	STD_LOGIC;
		byteout	: OUT	STD_LOGIC_VECTOR(7 DOWNTO 0));
END reg32mix8x8;

ARCHITECTURE a OF reg32mix8x8 IS
COMPONENT reg32mix
		PORT(
		bytein	: IN	STD_LOGIC_VECTOR(7 DOWNTO 0);
		byteclk	: IN	STD_LOGIC;
		statemix0, statemix1, statemix2, statemix3	: OUT	STD_LOGIC_VECTOR(7 DOWNTO 0);
		reg_clk	:IN STD_LOGIC);
END COMPONENT;
 COMPONENT reg32parser
		PORT(
		wordin	: IN	STD_LOGIC_VECTOR(31 DOWNTO 0);
		byteclk, wordclk,startreg		: IN	STD_LOGIC;
		byteout	: OUT	STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;


COMPONENT gen_wordclk
	PORT(
		byteclk		: IN	STD_LOGIC;
		start		: IN	STD_LOGIC;
		gen_clk		: OUT	STD_LOGIC);
END COMPONENT;


SIGNAL aux0, aux1, aux2, aux3 : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL aux4 : STD_LOGIC;
```

```
BEGIN

u1: reg32mix

PORT MAP (bytein => bytein,
                        statemix0 => aux0,
                        statemix1 => aux1,
                        statemix2 => aux2,
                        statemix3 => aux3,
                        byteclk => byteclk,
                        reg_clk => aux4);

u2: reg32parser
PORT MAP (  wordin(31 downto 24) => aux0,
                        wordin(23 downto 16) => aux1,
                        wordin(15 downto  8) => aux2,
                        wordin( 7 downto  0) => aux3,
                        byteclk => byteclk,
                        wordclk => aux4,
                        byteout => byteout,
                        startreg => startreg);

u3: gen_wordclk
        PORT MAP(
                byteclk => byteclk, start => startreg,
                gen_clk  => aux4);


END a;
```

Componente reg32mix

```
-- este circuito es la conexión entre el registro de 32 bits
-- y el circuito mixcolumn4

LIBRARY ieee;
USE ieee.std_logic_1164.all;


ENTITY reg32mix IS
        PORT(
                bytein    : IN   STD_LOGIC_VECTOR(7 DOWNTO 0);
                byteclk          : IN   STD_LOGIC;
                statemix0, statemix1, statemix2, statemix3          : OUT  STD_LOGIC_VECTOR(7
DOWNTO 0);
                reg_clk  :IN STD_LOGIC);
END reg32mix;

ARCHITECTURE a OF reg32mix IS
COMPONENT reg32
                PORT(
                bytein    : IN   STD_LOGIC_VECTOR(7 DOWNTO 0);
                byteclk  : IN   STD_LOGIC;
                reg_clk          : IN  STD_LOGIC;
                state0, state1, state2, state3                  : OUT  STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;
 COMPONENT mixcolumn4
PORT(
```

```
                state40                                                 : IN    STD_LOGIC_VECTOR
(7 DOWNTO 0);
                state41                                                 : IN    STD_LOGIC_VECTOR
(7 DOWNTO 0);
                state42                                                 : IN    STD_LOGIC_VECTOR
(7 DOWNTO 0);
                state43                                                 : IN    STD_LOGIC_VECTOR
(7 DOWNTO 0);
                stateout40, stateout41, stateout42, stateout43          : OUT   STD_LOGIC_VECTOR (7
DOWNTO 0));
END COMPONENT;

SIGNAL aux0, aux1, aux2, aux3 : STD_LOGIC_VECTOR(7 DOWNTO 0);

BEGIN

u1: reg32

PORT MAP (bytein => bytein,
                        state0 => aux0,
                        state1 => aux1,
                        state2 => aux2,
                        state3 => aux3,
                        byteclk => byteclk,
                        reg_clk => reg_clk);

u2: mixcolumn4

PORT MAP (state40 => aux0,
                        state41 => aux1,
                        state42 => aux2,
                        state43 => aux3,
                        stateout40 => statemix0,
                        stateout41 => statemix1,
                        stateout42 => statemix2,
                        stateout43 => statemix3);


END a;
```

Componente reg32

```
-- este es el registro de columna de 32 bits de state

LIBRARY ieee;
USE ieee.std_logic_1164.all;


ENTITY reg32 IS
        PORT(
                bytein          : IN   STD_LOGIC_VECTOR(7 DOWNTO 0);
                byteclk         : IN   STD_LOGIC;
                reg_clk         : IN  STD_LOGIC;
                state0, state1, state2, state3                  : OUT  STD_LOGIC_VECTOR(7 DOWNTO 0));
END reg32;

ARCHITECTURE a OF reg32 IS
```

```
COMPONENT reg8
		PORT(
		d		: IN   STD_LOGIC_VECTOR(7 DOWNTO 0);
		clk		: IN   STD_LOGIC;
		q		: OUT  STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;

SIGNAL aux0, aux1, aux2, aux3 : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL aux4 : STD_LOGIC;

BEGIN

u1: reg8

PORT MAP (d => bytein, q => aux0, clk => byteclk);

u2: reg8

PORT MAP (d => aux0, q => aux1, clk => byteclk);

u3: reg8

PORT MAP (d => aux1, q => aux2, clk => byteclk);

u4: reg8

PORT MAP (d => aux2, q => aux3, clk => byteclk);

aux4 <= reg_clk;
	PROCESS
	BEGIN
		WAIT UNTIL aux4 = '1';
		state0 <= aux3;
		state1 <= aux2;
		state2 <= aux1;
		state3 <= aux0;

	END PROCESS;
END a;
```

Componente reg8

```
-- MAX+plus II VHDL Example
-- User-Defined Macrofunction
-- Copyright (c) 1994 Altera Corporation

LIBRARY ieee;
USE ieee.std_logic_1164.all;


ENTITY reg8 IS
	PORT(
		d		: IN   STD_LOGIC_VECTOR(7 DOWNTO 0);
		clk		: IN   STD_LOGIC;
		q		: OUT  STD_LOGIC_VECTOR(7 DOWNTO 0));
END reg8;

ARCHITECTURE a OF reg8 IS
BEGIN
```

```
	PROCESS
	BEGIN
		WAIT UNTIL clk = '1';
		q <= d;
	END PROCESS;
END a;
```

Componente mixcolumn4

```
-- genera columna de la operacion mixcolumn

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mixcolumn4 IS
		PORT(
		--gen_clk                                                    : IN	STD_LOGIC;
		state40                                                     : IN	STD_LOGIC_VECTOR
(7 DOWNTO 0);
		state41                                                     : IN	STD_LOGIC_VECTOR
(7 DOWNTO 0);
		state42                                                     : IN	STD_LOGIC_VECTOR
(7 DOWNTO 0);
		state43                                                     : IN	STD_LOGIC_VECTOR
(7 DOWNTO 0);
		stateout40, stateout41, stateout42, stateout43		: OUT	STD_LOGIC_VECTOR (7
DOWNTO 0));
END mixcolumn4;

ARCHITECTURE a OF mixcolumn4  IS

-- SIGNAL AUX0, AUX1, AUX2, AUX3 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
	COMPONENT multcolumn_4		PORT(
		state0		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
		state1		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
		state2		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
		state3		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
		stateout	: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0));
	END COMPONENT multcolumn_4;
BEGIN
		u1: multcolumn_4

	PORT MAP (state0 => state40, state1 => state41, state2 => state42, state3 => state43,
				stateout => stateout40);

	u2: multcolumn_4

	PORT MAP (state0 => state41, state1 => state42, state2 => state43, state3 => state40,
				stateout => stateout41);

	u3: multcolumn_4

	PORT MAP (state0 => state42, state1 => state43, state2 => state40, state3 => state41,
				stateout => stateout42);

	u4: multcolumn_4

	PORT MAP (state0 => state43, state1 => state40, state2 => state41, state3 => state42,
				stateout => stateout43);
```

```
END a;
```

Componente multcolumn4

```
-- MAX+plus II VHDL Template
-- multiplica una columna de state por {01}, {02} y {03} y genera byte
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY multcolumn_4 IS

	PORT
	(
		state0		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
		state1		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
		state2		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
		state3		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
		stateout	: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0)

	);

END multcolumn_4;

ARCHITECTURE a OF multcolumn_4 IS
	COMPONENT xtime		PORT(
			inxtime		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
			outxtime			: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0));
	END COMPONENT xtime;

	COMPONENT x3		PORT(
			inx3		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
			outx3		: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0));
	END COMPONENT x3;

	signal aux0: std_logic_vector (7 downto 0);
	signal aux1: std_logic_vector (7 downto 0);
	signal aux2: std_logic_vector (7 downto 0);
	signal aux3: std_logic_vector (7 downto 0);

BEGIN
		u1: xtime

			PORT MAP (inxtime => state0,
				outxtime => aux0);
		u2: x3

			PORT MAP (inx3 => state1,
				outx3 => aux1);

		aux2 <= aux0 xor aux1;
		aux3 <= state2 xor state3;
		stateout <= aux2 xor aux3;

END a;
```

Componente xtime

```
-- MAX+plus II VHDL Template
-- xtime

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY xtime IS

	PORT
	(
		inxtime		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
		outxtime			: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0)

	);

END xtime;

ARCHITECTURE a OF xtime IS
begin
Process (inxtime)

BEGIN
		IF inxtime(7)= '0'THEN

			outxtime <= inxtime(6 downto 0) & '0';
		ELSE
		   outxtime <= inxtime(6 downto 0) & '0' xor "00011011" ;

		END IF;
end process;
END a;
```

Componente x3

```
-- MAX+plus II VHDL Template
-- xtime

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY x3 IS

	PORT
	(
		inx3		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
		outx3		: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0)

	);

END x3;

ARCHITECTURE a OF x3 IS
	signal aux: std_logic_vector (7 downto 0);
begin
Process (inx3)

BEGIN
```

```
                IF inx3(7) = '0'THEN

                        aux <= inx3 (6 downto 0) & '0';
                    ELSE
                       aux <= inx3(6 downto 0) & '0' xor "00011011" ;

                END IF;
end process;
outx3 <= aux xor inx3;
END a;
```

Componente reg32parser

```
-- este es el registro de columna de 32 bits de state

LIBRARY ieee;
USE ieee.std_logic_1164.all;


ENTITY reg32parser IS
        PORT(
                wordin   : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);
                byteclk, wordclk,startreg              : IN   STD_LOGIC;
                byteout  : OUT  STD_LOGIC_VECTOR(7 DOWNTO 0));
END reg32parser;

ARCHITECTURE a OF reg32parser IS
COMPONENT reg8
                PORT(
                d                       : IN   STD_LOGIC_VECTOR(7 DOWNTO 0);
                clk                     : IN   STD_LOGIC;
                q                       : OUT  STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;

SIGNAL aux0, aux1, aux2, aux3 : STD_LOGIC_VECTOR(7 DOWNTO 0);

BEGIN

u1: reg8

PORT MAP (d => wordin(31 downto 24), q => aux0, clk => wordclk);

u2: reg8

PORT MAP (d => wordin(23 downto 16), q => aux1, clk => wordclk);

u3: reg8

PORT MAP (d => wordin(15 downto 8), q => aux2, clk => wordclk);

u4: reg8

PORT MAP (d => wordin(7 downto 0), q => aux3, clk => wordclk);

PROCESS(byteclk)
        variable index: integer range 0 to 3;
BEGIN
IF startreg = '0' THEN      index :=0;
```

```
                ELSE

        IF (byteclk'EVENT AND byteclk = '1') THEN

                        CASE index IS
                                WHEN 0 =>
                                   byteout <= aux0;
                                WHEN 1 =>
                                   byteout <= aux1;
                                WHEN 2 =>
                                   byteout <= aux2;
                                WHEN 3 =>
                                   byteout <= aux3;
                        END CASE;

                        index := index + 1;
                END IF;
END IF;
END PROCESS;
END a;
```

Componente gen_wordclk

```
-- contador modulo 4 para generar wordclk
LIBRARY ieee;
USE ieee.std_logic_1164.all;


ENTITY gen_wordclk IS
        PORT(
                byteclk         : IN   STD_LOGIC;
                start           : IN   STD_LOGIC;
                gen_clk         : OUT  STD_LOGIC);
END gen_wordclk;


ARCHITECTURE a OF gen_wordclk IS
SIGNAL aux :  STD_LOGIC;

BEGIN
        PROCESS (byteclk)
        VARIABLE    cnt     : INTEGER RANGE 0 TO 3;
        BEGIN
        IF start = '0' THEN         cnt :=0;
                        aux <= '0';

                        ELSE

                IF (byteclk'EVENT AND byteclk = '1') THEN
                                -- IF start = '1' THEN
                                cnt := cnt + 1;
                                -- ELSE cnt :=0;
                                -- END IF;
                        END IF;
                IF (cnt=0 ) THEN
                        aux <= '1';
                        ELSE
                        aux <= '0';
                END IF;
```

```
END IF;

        END PROCESS;
gen_clk<=aux AND  (NOT byteclk);
END a;
```

Bibliografía

ADAMEK, J., ¨*Foundation of Coding*¨. John Wiley & Sons. 1991.

ALLENBY, R.B.J.T., ¨*Rings, Fields and Groups. An Introduction to Abstract Algebra*¨. Edward Arnold Publishers. 1983.

ALTERA. "*MAX + PLUS II AHDL*". Altera Corporation.

ALTERA. "*MAX + PLUS II getting started*". Altera Corporation.

ALTERA. ¨*Data Book 1998*¨. Altera Corporation 1998.

ASHENDEN P. ¨The VHDL Cookbook¨. First Edition, 1998. http://tech-www.informatik.uni-hamburg.de/vhdl/doc/cookbook/VHDL.html

ASHENDEN, P., ¨*The Designer´s Guide to VHDL*¨. Second Edition. Morgan Kauffmann Publishers. 2002.

ASHRUF, R., GAYDADJIEV, G., VASSILIADIS, S., ¨*Reconfigurable implementation for the AES algorithm*¨. Computer Engineering Laboratory, Electrical Engineering Department, Delft University of Technology, http://ce-serv.et.tudelft.nl/~molen/publications/2002/ash/ash_prorisc2002.pdf.

BIHAM E., ¨*A note on Comparing the AES Candidates*¨. Second AES conference, 1999. http://www.cs.technion.ac.il/~biham/

BLAHUT, R. E., COSTELLO, D. J., MAURER, U., MITTEHOLZER, T., ¨*Communications and Cryptography. Two Sides of One Tapestry*¨. Kluwer Academic Publishers. 1994.

BROWN, S., ROSE, J., ¨*Architecture of FPGAs and CPLDs: A Tutorial*¨. IEEE Design and Test of Computers, Vol. 12, Nº 2. Summer 1996, pp. 42-57. http://www.eecg.toronto.edu/jayar/pubs/pubs.html.

CASTINEIRA MOREIRA, J., "*Improvement of Write/Read Characteristics in Optical Storage Systems (E. G. Compact Discs and Cd-Roms)*" Tesis para obtener el grado de "MSc in Digital Signal Processing Applications in Communication Systems",1996 DIRECTORES DE TESIS: DR. G. MARKARIAN, PROF. B. HONARY.

DAEMEN, J., RIJMEN V., ¨*AES Proposal: Rijndael*¨. Document version 2. Date: 03/09/99. NIST´s AES home page. http://www.nist.gov/aes.

ELBIRT, A., YIP, W., CHETWYND, B., PAAR, C., ¨*An FPGA Implementation and Performance Evaluation of the AES block cipher candidates algorithm finalists*¨. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, August 2001, pp. 545-557.

FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION (FIPS) 197. ¨*Specification for the Advanced Encryption Standard (AES)*¨. November 26, 2001. NIST´s AES home page. http://www.nist.gov/aes.

FISCHER, V., ¨*Realization of the round 2 AES candidates using Altera FPGA*¨. March 2000 http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html.

GAJ, K., CHODOWIEC, P., ¨*Comparison of the hardware performance of the AES candidates using reconfigurable hardware*¨. Proceedings of RSA Security Conference - Cryptographer's Track, San Francisco, CA, 2001 April 8-12, pp. 84-99.

GREEN MOUNTAIN COMPUTING SYSTEMS ¨*An Introductory VHDL Tutorial*¨. 1995. http://www.gmvhdl.com/VHDL.html

HEEMSKERK, J.P.J., IMMINK, K.A.S., "*Compact Disc: System Aspects and Modulation*". Philips Technical Review,Volume 40, pp 157-164,1982.

HILLMA, A. P., ALEXANDERSON, G. L., *"A First Undergraduate Course in Abstract Algebra,"* Second Edition. Wadsworth Publishing Company, Inc. Belmont, California, 1978.

HODJAT, A., VERBAUWHEDE, I., *¨Speed-Area Trade-off for 10 to 100 Gbits/s Throughput AES Processor¨* 2003 IEEE Asilomar Conference on Signals, Systems, and Computers, November 2003. http://www.ee.ucla.edu/~ahodjat/AES/asilomar_paper_alireza.pdf.

HODJAT, A., VERBAUWHEDE, I., *¨A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA¨*. Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM´04). http://www.ee.ucla.edu/~ahodjat/AES/hodjat_fccm.pdf.

HOEVE, H., TIMMERMANS, J., VRIES, L. B., "*Error correction and concealment in the Compact Disc System*" Philips tech. Rev. 40, 166-172,1982, No 6.

HONARY, B., DARNELL, M., FARRELL, P., ¨*Communications Coding and Signal Processing*¨. Research Studies Press Ltd. John Wiley & Sons Inc. 1997.

IMMINK, K.A.S., ENGLEWOOD CLIFFS, N.J., ¨*Coding Techniques for Digital Recorders*¨ Prentice Hall International. (1991).

KARRI, R., KIM, Y., ¨*Field Programmable Gate Array implementation of Advanced Encryption Standard*¨. 2001. http://www.eeweb.poly.edu/dream-it/publications/Rijndael.pdf.

KERINS, T., PPOPOVICI, A., DALY, A., MARNANE, W., ¨*Hardware encryption engines for e-commerce*¨. Proceedings of Irish Signals and Systems Conference, 2002, pp 89-94.

KHOA, V., ZIER, D., ¨*FPGA Implementation AES for CCM Mode Encryption Using Xilinx Spartan II* ¨ ECE 679, Advanced Cryptography, Oregon State University, Spring 2003.

LIN, S., COSTELLO, D., ¨*Error Control Coding: Fundamentals and Applications*¨. Prentice Hall, Englewood Cliffs, 1983.

MAX+PLUS II. ¨*Programmable Logic Development System. VHDL*¨. Altera Corporation. 1996.

MCELIECE, R. ¨*The Theory of Information and Coding. Encyclopedia of Mathematics and its Applications*¨. Addison-Wesley Publishing. 2002.

MENEZES, A., VAN OORSCHOT, P. C., VANSTONE, S. A., ¨*Handbook of Applied Cryptography*¨. CRC Press. 1997.

MITCHELL, C., ¨*Cryptography and Coding II*¨ Oxford University Press. 1992.

MROCZKOWSKI, P., ¨*Implementation of the block cipher Rijndael using Altera FPGA*¨. http://csrc.nist.gov/encryption/aes/round2/comments/200 00510-pmroczkowski.pdf. Aug. 2001.

MURPHY, S., ROBSHAW, M., ¨*Essential Algebraic Structure within AES*¨. Second NESSIE. New European Schemes for Signature, Integrity and Encryption Workshop. September 2001.

O´DRISCOLL, C., ¨*Hardware Implementations Aspects of the Rijndael Block Cipher*¨. A Thesis Submitted to the National University of Ireland in Fulfillment of the Requirements for the Degree of M. Eng. Sc. October 2001.

PANATO, A., BARCELOS, M., REIS, R., *¨A Low Device Occupation IP to Implement Rijndael Algorithm¨* Design, Automation and Test in Europe Conference and Exhibition (DATE'03 Designers' Forum), 2003. http://www.inf.ufrgs.br/%7Epanato/artigos/designforum03.pdf.

PARDO, F., BOLUDA, J., ¨*VHDL. Lenguaje para síntesis y modelado de circuitos*¨. Editorial Ra-Ma. Edición 1999.

PEACOCK, C. ¨*Interfacing the Standard Parallel Port*¨. 1998. http://www.senet.com.au/cpeacock.

PEEK, J.B.H., "*Communications Aspects of the Compact Disc Audio System*" IEEE Communications Magazine. February 1985. Vol. 23, No 2.

RIJMEN, V., ¨*Efficient Implementation of the Rijndael S-Box*¨. CHES 2003, LNCS 2779, pp 334-350.

RUDRA, A., DUBEY, K., JUTLA, C., ¨*Efficient implementation of Rijndael encryption with composite field arithmetic*¨. Workshop on Cryptographic Hardware and Embedded Systems, CHES 2001, May 13-16, Paris, France.

SCHNEIER, B., ¨*Applied Cryptography*¨, John Wiley & Sons, Inc., second edition, 1996.

SHIM, J., KIM, D., KANG, Y., KWON, T., CHOI, J., ¨*Inner-pipelining Rijndael cryptoprocessor with on-the-fly key scheduler*¨. http://www.ap-sic.org/2002/proceedings/2B/2B-3.PDF.

SMITH, R. E. ¨*Internet Cryptography*¨. Addison Wesley, 1995.

STALLINGS, W., ¨*Cryptography and Network Security*¨, 2nd Edition, 1999. Prentice Hall.

STINSON, D. R., ¨*Cryptography. Theory and Practice*¨. CRC Press. 1995.

TERÉS, L., TORROJA, Y., OLCOZ, S., VILLAR, E., ¨*VHDL. Lenguaje Estándar de Diseño Electrónico*¨. Editorial Mc Graw Hill/Interamericana de España, S.A.U. 1997.

VU, K., ZIER, D., *¨FPGA Implementation AES for CCM Mode Encryption Using Xilinx Spartan-II¨* Spring 2003. http://www.internetjournals.net/journals/transactions_on_advanced_research/2005/january/TAR.pdf

WEAVER, N., WAWRZYNEK, J., *¨High Performance, Compact AES Implementation in Xilinx FPGAs¨.* September 27, 2002. http://www.cs.berkeley.edu/ ~nweaver/rijndael