Centro de Investigación y de
Estudios Avanzados del Instituto
Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Implementación ligera con VHDL del esquema COFB

TESIS

Que presenta

Manuel Rodríguez Camacho

Para obtener el grado de

Maestro en Ciencias en Computación

Directores de la Tesis:

Dr. Francisco José Rambó Rodríguez Henríquez

Dr. Cuauhtemoc Mancillas López

Ciudad de México Diciembre, 2018

A mi madre, mis hermanos y mis hijos.

Agradecimientos

Gracias a mi madre por todo su amor, a mi hermana por ser mi amiga y compañera, a mi hermano por su cariño y apoyo, a Emiliano y Santiago por ser mi más grande motivación.

A mi director, el Dr. Francisco Rodríguez-Henríquez, por su amable guía, sus consejos, su confianza, y por creer en mí. A mi codirector, el Dr. Cuauhtemoc Mancillas López, por su apoyo, su enseñanza, su paciencia y por confiar en mí.

A la Dra. Brisbane Ovilla y al Dr. Juan Carlos Ku, mis sinodales, por sus valiosos comentarios y observaciones que enriquecieron este trabajo.

Al CINVESTAV, por brindarme la oportunidad de ser parte de esta prestigiosa institución y realizar un posgrado. Al CONACyT, por el apoyo económico brindado durante mis estudios de maestría.

A mis compañeros del CINVESTAV por su amistad, ayuda y todos los buenos momentos con ellos.

A Sofi, que ha sido tan amable, siempre acudiendo al llamado de auxilio de los alumnos. ¡Muchas gracias Sofi!.

A todos los profesores del Departamento de Computación del CINVESTAV, por sus enseñanzas.

Resumen

Se denomina cifrado autenticado (*Authenticated Encryption* AE) a un algoritmo criptográfico de llave simétrica, que provee tanto los servicios de confidencialidad como de autenticidad. Un esquema AE con datos asociados (*Authenticated Encryption with Associated Data* AEAD), es un algoritmo de cifrado autenticado con datos adicionales que agregan seguridad al esquema.

COFB[1] es un esquema de cifrado autenticado con datos asociados, diseñado para ser eficiente con respecto a los recursos de hardware necesarios para su implementación. Utiliza únicamente el algoritmo de cifrado de un cifrador por bloques. Por otra parte, la aritmética más compleja consiste en realizar multiplicaciones con constantes pequeñas en campos finitos binarios.

Se dice que una implementación en hardware es ligera cuando se ocupa el menor área posible en un circuito electrónico. Cuando se trabaja con hardware reconfigurable, el objetivo es utilizar el menor número de recursos integrados (LUTs, bloques de memoria, etc) en un dispositivo FPGA.

En el presente trabajo de tesis se desarrolla un estudio experimental de COFB con dos cifradores por bloques diferentes: AES y Midori. Primero se realizan implementaciones ligeras de ambos cifradores, para luego utilizarlas como primitivas de cifrado dentro de COFB y finalmente se realiza un análisis del área ocupada en ambas implementaciones.

Abstract

Authenticated encryption (AE) is a symmetric key cryptographic algorithm, which provides both confidentiality and authenticity services. An AE schema with associated data (AEAD) is an authenticated encryption algorithm with additional data that adds security to the schema.

COFB is an authenticated encryption scheme with associated data designed to be efficient with respect to the hardware resources necessary for its implementation. It only uses the algorithm of encryption of a block-cipher. On the other hand, the most complex operations consist of multiplications with small constants in finite binary fields.

It is said that a hardware implementation is light when it deals with the smallest possible area in the circuit. When working with reconfigurable hardware, the goal is to use the least number of components integrated in the FPGA device.

In this thesis work an experimental study of COFB is developed with two blocks for different blocks: AES and Midori. First, both ciphers are implemented in the lightest possible version, and then they are used as cipher primitives in COFB. Finally, an analysis of the space occupied in both implementations is carried out.

Índice general

Índice de Figuras

Índice de Tablas

Índice de Códigos

Índice de Algoritmos

Capítulo 1

Introducción

1.1. Motivación

La cantidad de dispositivos con capacidad de conexión a Internet crece de manera importante. La gran diversidad de aparatos conectados incluye (en gran medida) aparatos reducidos en sus capacidades de procesamiento, memoria y energía; por ejemplo teléfonos, tabletas, computadoras portátiles de bajo desempeño y además, a todos los dispositivos que ofrecen funcionalidad específica como sensores y actuadores que conforman el denominado Internet de las cosas[1] (IoT), ver Figura 1.1. Todos ellos con una restricción en común: el suministro de energía depende de una batería. Para reducir el consumo y lograr un uso eficiente de sus baterías, los dispositivos portátiles reducen su tamaño y limitan su capacidad de procesamiento.

Cuando la comunicación entre dispositivos se realiza a través de un medio promiscuo como lo es Internet, existe la necesidad de proteger la información transmitida, es decir ocultar los datos de manera que no puedan ser leídos por personas (o entidades) no autorizadas. Además, se debe contar con un mecanismo con el cual sea posible verificar que los mensajes son enviados por una determinada persona (o entidad) autorizada y recibido por otra. Para ello la *criptografía* brinda los servicios de *privacidad* y *autenticación*, los cuales son ofrecidos a través de herramientas de *cifrado* (implementando cifradores por bloques por ejemplo) y *cifrado autenticado* (además de la

[1] Concepto que se refiere a la interconexión digital de objetos de uso cotidiano a través de Internet. Propuesto en 1999 por Kevin Ashtom en el Auto-ID Center del MIT, a partir de investigaciones en el campo de la identificación por radiofrecuencia en red (RFID) y tecnologías de sensores.

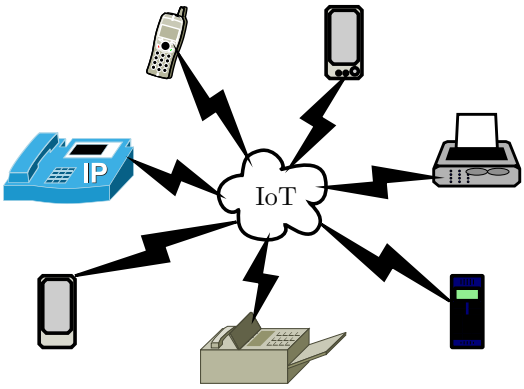

Figura 1.1: Ecosistema del internet de las cosas.

primitiva de cifrado, implementan verificación de la autenticidad) respectivamente.

Una solución económica (con respecto a la cantidad de recursos utilizados y al consumo de energía) que proporcione servicios de seguridad a dispositivos pequeños, necesita un esquema que pueda ser implementado en ellos a pesar de sus reducidos recursos. Dicha necesidad ha forjado el desarrollo de un área de investigación relacionada con la *criptografía ligera*.

En la *Conferencia sobre Hardware Criptográfico y Sistemas Empotrados* (*CHES*[2] por siglas en inglés) se presentan investigaciones y trabajos, con un enfoque en la implementación eficiente y segura de algoritmos criptográficos. Es patrocinada por la *Asociación Internacional de Investigación Criptográfica* (*IACR*[3] por sus siglas en inglés). En ella se han presentado trabajos que han brindado soluciones eficientes para dispositivos con recursos limitados. Un ejemplo es el cifrador por bloques PRESENT[2], propuesto en el año 2007 como alternativa a AES, y diseñado para ser implementado de manera eficiente en dispositivos pequeños.

La *Competencia para el Cifrado Autenticado: Seguridad, Aplicabilidad, y Robustez* (*CAESAR*[4] por sus siglas en inglés), convocó a la comunidad criptográfica a participar presentando soluciones de cifrado autenticado seguras,

[2] `https://ches.iacr.org/`.
[3] `https://www.iacr.org/`.
[4] `https://competitions.cr.yp.to/caesar.html`.

con una aplicación eficiente y robusta. Anunciada en 2013, ha recibido una gran cantidad de trabajos, entre los que se encuentran los modos de cifrado autenticado con datos asociados CLOC[3] y SILC[4], ambos orientados al bajo consumo de recursos tanto de hardware como de energía.

En la actualidad, continúa la investigación en el campo de la implementación eficiente de algoritmos criptográficos, en sistemas portátiles o de recursos restringidos. Uno de los trabajos publicados más recientes, es el esquema de cifrado COFB propuesto en el trabajo[1] de Chakraborti et. al. En su publicación, COFB fue implementado usando AES, sin embargo, es claro que el esquema resulta mucho más eficiente con algún otro cifrador ligero, ya que la primitiva de cifrado empleada no ha sido diseñada para ser ligera.

En este trabajo se presenta una solución para dispositivos portátiles o de recursos limitados: una implementación ligera del esquema COFB utilizando núcleos de AES y Midori, que ofrece servicios de confidencialidad y autenticidad de manera simultánea.

Con el desarrollo del presente trabajo se obtuvo un núcleo de sólo cifrado Midori64 (Midori en su versión de 64 bits), implementado de manera ligera para hardware reconfigurable. Por otra parte, se ha desarrollado un núcleo de sólo cifrado AES128 (AES en su versión de 128 bits) en una versión reducida en área. Además, se implementó una versión ligera del esquema COFB con VHDL, con lo que se proporciona una solución compacta para dar servicios de autenticidad y confidencialidad, empleando una menor cantidad de recursos de hardware. Finalmente, comparando el espacio ocupado por el esquema con los dos cifradores elegidos, se puede concluir cuál primitiva es la adecuada para una implementación reducida con respecto al área ocupada en el circuito.

1.2. Organización de la tesis

En el capítulo 2 del presente trabajo, se presentan los preliminares matemáticos que sustentan el desarrollo de la implementación propuesta, así como el estado del arte relacionado con cifradores por bloques y esquemas de cifrado ligeros.

En el capítulo 3, se aborda la implementación: la programación en lenguaje C del esquema COFB y de los núcleos de cifrado tanto de AES128

como de Midori64, su integración en COFB-AES128 y COFB-Midori64 (respectivamente), y la generación de sus respectivos vectores de prueba para comprobar la correctitud del posterior diseño en VHDL.

En el capítulo 4 se presentan las mediciones realizadas con respecto al área ocupada por cada uno de los diseños implementados (COFB-AES128 y COFB-Midori64) y su análisis.

En el capítulo 5 se concluye con una breve discusión sobre los resultados obtenidos y del trabajo a futuro.

Capítulo 2

Marco teórico

2.1. Preliminares matemáticos

Definición 1. Una *operación binaria* sobre un conjunto G es una función $* : G \times G \to G$. Para $a, b \in G$ se suele escribir $a * b$ en lugar de $*((a, b))$, e incluso solamente ab.

- Si para cualesquiera $a, b, c \in G$ se cumple que $(a * b) * c = a * (b * c)$, entonces se dice que una operación binaria es *asociativa*.
- Si para cualesquiera $a, b \in G$ se cumple que $a * b = b * a$, entonces se dice que es *conmutativa* .

Definición 2. Sea G un conjunto no vacío y $*$ una operación binaria sobre G; entonces el par $(G, *)$ recibe el nombre de:

- *Semigrupo*, si $*$ es asociativa.
- *Monoide*, si $*$ es asociativa y existe un *elemento neutro* $e_G \in G$ tal que $e_G g = g e_G = g$, para toda $g \in G$.
- *Grupo*, si $(G, *)$ es un monoide y se cumple la propiedad del inverso para la operación*, es decir, para cada $g \in G$ existe $g^{-1} \in G$ tal que $gg^{-1} = g^{-1}g = e_G$.
- *Grupo conmutativo* o *Abeliano* si $(G, *)$ es un grupo y la operación $*$ es conmutativa. En caso contrario se dice que el grupo no es conmutativo o que es *no abeliano*.

Definición 3. El orden de un grupo $(G, *)$, es la cardinalidad del conjunto G. Decimos que el grupo $(G, *)$ es finito si $\mid G \mid = n \in \mathbb{N}$

Teorema 1. Sea G un grupo y $a \in G$, entonces

$$H = \{a^n \mid n \in \mathbb{Z}\} = \{e, a, a^2, a^3, \dots\} \tag{2.1}$$

es un subgrupo de G.

Teorema 2. Sea H un subgrupo de un grupo G. Definimos la relación $\sim_I$ en G como

$$a \sim_I b \Leftrightarrow a^{-1}b \in H \tag{2.2}$$

y definimos $\sim_D$ como

$$a \sim_D b \Leftrightarrow ab^{-1} \in H \tag{2.3}$$

Entonces $\sim_I$ y $\sim_D$ son relaciones de equivalencias.

Definición 4. Sea H un subgrupo de un grupo G. El subconjunto $aH = \{ah \mid h \in H\}$ de G es llamado, *clase lateral izquierda* de H conteniendo al elemento a, mientras que el subconjunto $Ha = \{ha \mid h \in H\}$ de G es llamado, la *clase lateral derecha* de H, que contiene al elemento a.

Definición 5. Subgrupo normal Sea H un subgrupo de un grupo G, H es normal si sus clases laterales derechas e izquierdas coinciden, es decir:

$$gH = Hg \tag{2.4}$$

para toda $g \in G$. Denotamos que H es normal en G como $H \triangleleft G$ o $G \triangleright H$. Todos los subgrupos de un grupo abeliano son normales.

Teorema 3. Sea H un subgrupo de un grupo G, entonces la multiplicación de clases laterales izquierdas dadas por la ecuación 2.5 está bien definida si y sólo si H es un subgrupo normal de G.

$$(aH)(bH) = (ab)H \tag{2.5}$$

Teorema 4. Sea H un subgrupo normal de un grupo G, entonces las clases laterales de H forman un grupo G/H bajo la operación binaria $(aH)(bH) = (ab)H$.

Definición 6. Grupo cociente El grupo G/H en el Teorema 4 es el grupo cociente de G sobre H.

Definición 7. Un *Anillo* es una tupla $(R, +, \cdot)$ consistente de un conjunto R y dos operaciones binarias, $+$ y $\cdot$, llamadas *suma* y *multiplicación*, definidas en R con las siguientes propiedades:

- $(R, +)$ es un grupo abeliano.
- $(R, \cdot)$ es un semigrupo.
- Para todo $a, b, c \in R$, se cumplen *la ley distributiva izquierda*, $a \cdot (b + c) = (a \cdot b) + (c \cdot b)$, y la *ley distributiva derecha* $(a+b) \cdot c = (a \cdot c) + (b \cdot c)$.

Definición 8. Sea $(R, +, \cdot)$ un anillo.

- $(R, +, \cdot)$ es un *anillo conmutativo* si la operación $\cdot$ es conmutativa.
- $(R, +, \cdot)$ es un *anillo con unitario* si $(R, \cdot)$ es un monoide.

Definición 9. Sea R un anillo. Un *polinomio* $f(x)$ *con coeficientes en* R es una suma formal infinita

$$\sum_{i=0}^{\infty} a_i x^i = a_0 + a_1 x + \cdots + a_n x^n + \ldots, \tag{2.6}$$

donde cada $a_i \in R$ y $a_i = 0$ excepto para un número finito de valores de i. Los elementos a_i son llamados *coeficientes* de $f(x)$. Si se cumple para alguna i que $a_i \neq 0$ entonces, el mayor valor de i que cumple esto es llamado el *grado* del polinomio $f(x)$. Si todos los coeficientes son cero, entonces el grado del polinomio está indefinido. Si $a_i \neq 0$ únicamente para $i = 0$, entonces $f(x)$ es denominado *polinomio constante*. Todo elemento en R es un polinomio constante.

Es posible definir una operación de suma y una multiplicación en estos polinomios.

Definición 10. Sean $f(x) = a_0 + a_1 x + \cdots + a_n x^n + \ldots$ y $g(x) = b_0 + a_1 x + \cdots + b_n x^n + \ldots$ entonces la suma de polinomios está definida por

$$f(x) + g(x) = (a_0 + b_0) + (a_1 + b_1)x + \cdots + (a_n + b_n)x^n + \ldots \tag{2.7}$$

.

La multiplicación está dada por

$$f(x)g(x) = d_0 + d_1 x + \cdots + d_n x^n + \ldots, \text{ donde } d_n = \sum_{i=0}^{n} a_i b_{n-i}.$$

Teorema 5. El conjunto $R[x]$ de todos los polinomios en un indeterminada x con coeficientes en un anillo R, es un anillo con las operaciones definidas previamente. Si R es un anillo conmutativo, entonces $R[x]$ lo es, y si R es un anillo con unitario entonces $R[x]$ lo es.

Definición 11. Sea $GF(2)[x]$ el anillo de polinomios del campo $GF(2) = \{0, 1\}$. Sea $p(x)$ un polinomio irreducible de grado n en $GF(2)[x]$. El conjunto cociente $GF(2)[x]/\langle p(x)\rangle$ forma un campo con la suma usual de polinomios y la multiplicación módulo el polinomio $p(x)$. Es usual llamar a este campo , *campo binario de grado n* y lo denotamos por $GF(2^n)$. Decimos que el campo $GF(2^n)$ es una *extensión de grado n* del campo $GF(2)$.

Todas las cadenas de bits de longitud n serán consideradas como elementos del campo $GF(2^n)$ y se pueden representar como polinomios de grado a lo más $n-1$ con coeficientes en $\{0, 1\}$. La operación suma + dentro del campo $\mathrm{GF}(2^n)$ se define como la operación XOR. Para producto módulo $p(x)$ entre dos elementos del campo $\mathrm{GF}(2^n)$, $p(x)$ es un polinomio irreducible. $\{0, 1\}^*$ se refiere al conjunto de todas las cadenas binarias.

Definición 12. Sean n y l dos enteros[1] tales que $n, l \in \mathbb{Z}^+$, el espacio de mensajes M es $M = \{0, 1\}^n$ y el espacio de llaves K es $K = \{0, 1\}^l$.

Definición 13. Sea E_K un cifrador por bloques tal que $E : M \times K \to \{0, 1\}^n$. Un cifrador por bloques tiene una función inversa tal que $E_K^{-1}(E_K(M)) = M$.

Un cifrador por bloques sólo puede cifrar mensajes de tamaño n, en aplicaciones reales por lo general, los mensajes son de mayor longitud. Para poder procesar mensajes mayores a n bits se utilizan los modos de operación. *Núcleo de cifrado* (o *núcleo de descifrado*) se refiere a la función de sólo cifrado (o sólo descifrado) $E_k(x)$ (o $E_k^{-1}(y)$ respectivamente), donde $x \in M$, $y = E_k(x)$ y $k \in K$.

Definición 14. Sea $\Pi = (\mathcal{E}, \mathcal{D}, \mathfrak{K})$ un esquema de cifrado autenticado[5] donde $\mathfrak{K}$ es el generador de llaves, $\mathcal{E}$ es la función de cifrado autenticado y $\mathcal{D}$) es la función de descifrado verificado.

Asociados a Π se tiene el conjunto de los *Nonces*[2] $\mathfrak{N} = \{0, 1\}^N$ y el conjunto de los mensajes $\mathfrak{M} \subseteq \{0, 1\}^*$. El espacio de llaves $\mathfrak{K}$ es un conjunto no vacío y finito de cadenas binarias.

[1] Siendo ambos enteros n y l, las longitudes en bits del bloque del cifrador y de su llave, respectivamente. Para casos como AES128 y Midori128: $n = l = 128$.

[2] En criptografía es usado el término *Nonce* (inspirado en el concepto inglés *nonce word*) con el que se denomina a un número arbitrario que es utilizado una sola vez, y que además debe ser único para cada mensaje, de otro modo se compromete la seguridad del esquema.

$\mathcal{E}$ es un algoritmo determinista que toma cadenas $\mathcal{K} \in \mathfrak{K}$, $\mathcal{N} \in \mathfrak{N}$ (donde $\mathcal{N}$ debe ser único para cada $\mathcal{M}$) y $\mathcal{M} \in \mathfrak{M}$, y devuelve la cadena $\mathcal{C} = \mathcal{E}_{\mathcal{K}}^{\mathcal{N}}(\mathcal{M}) = \mathcal{E}_{\mathcal{K}}(\mathcal{N}, \mathcal{M})$.

$\mathcal{D}$ es un algoritmo determinista que toma cadenas $\mathcal{K} \in \mathfrak{K}$, $\mathcal{N} \in \mathfrak{N}$ (donde $\mathcal{N}$ debe ser único para cada $\mathcal{C}$) y $\mathcal{C} \in \{0,1\}^*$ y devuelve la cadena $\mathcal{Y} = \mathcal{D}_{\mathcal{K}}^{\mathcal{N}}(\mathcal{C})$, si $\mathcal{Y} \in \mathfrak{M}$ entonces $\mathcal{Y}$ es el mensaje, de lo contrario $\mathcal{Y} = \perp$, es decir, es inválido.

Es necesario que $\mathcal{D}_{\mathcal{K}}^{\mathcal{N}}(\mathcal{E}_{\mathcal{K}}^{\mathcal{N}}(\mathcal{M})) = \mathcal{M}$ se cumpla para toda $\mathcal{K} \in \mathfrak{K}$, $\mathcal{N} \in \mathfrak{N}$ y $\mathcal{M} \in \mathfrak{M}$.

Definición 15. Sea $\Pi = (\mathfrak{K}, \mathcal{E}, \mathcal{D})$ un esquema de cifrado autenticado con datos asociados[6].

Asociados a Π se tiene el conjunto de los *Nonces* $\mathfrak{N} = \{0,1\}^{\mathcal{N}}$, al conjunto de los mensajes $\mathfrak{M} \subseteq \{0,1\}^*$, y además el conjunto de las cabeceras $\mathfrak{H} \subseteq \{0,1\}^*$. El espacio de llaves $\mathfrak{K}$ es un conjunto no vacío y finito de cadenas binarias.

$\mathcal{E}$ es un algoritmo determinista que toma cadenas $\mathcal{K} \in \mathfrak{K}$, $\mathcal{N} \in \mathfrak{N}$ (donde $\mathcal{N}$ debe ser único para cada mensaje $\mathcal{M}$) , $\mathcal{H} \in \mathfrak{H}$ y $\mathcal{M} \in \mathfrak{M}$, y devuelve la cadena $\mathcal{C} = \mathcal{E}_{\mathcal{K}}^{\mathcal{N},\mathcal{H}}(\mathcal{M}) = \mathcal{E}_{\mathcal{K}}(\mathcal{N}, \mathcal{H}, \mathcal{M})$.

$\mathcal{D}$ es un algoritmo determinista que toma cadenas $\mathcal{K} \in \mathfrak{K}$, $\mathcal{N} \in \mathfrak{N}$ (donde $\mathcal{N}$ debe ser único para cada $\mathcal{M}$) , $\mathcal{H} \in \mathfrak{H}$ y $\mathcal{C} \in \{0,1\}^*$ y devuelve la cadena $\mathcal{Y} = \mathcal{D}_{\mathcal{K}}^{\mathcal{N},\mathcal{H}}(\mathcal{C})$, si $\mathcal{Y} \in \mathfrak{M}$ entonces $\mathcal{Y}$ es el mensaje, de lo contrario $\mathcal{Y} = \perp$, es decir, es inválido.

Es necesario que $\mathcal{D}_{\mathcal{K}}^{\mathcal{N},\mathcal{H}}(\mathcal{E}_{\mathcal{K}}^{\mathcal{N},\mathcal{H}}(\mathcal{M})) = \mathcal{M}$ se cumpla para toda $\mathcal{K} \in \mathfrak{K}$, $\mathcal{N} \in \mathfrak{N}$, $\mathcal{H} \in \mathfrak{H}$ y $\mathcal{M} \in \mathfrak{M}$.

Definición 16. Sea λ la cadena vacía, para algunos $X, Y \in \{0,1\}^*$ donde $\{0,1\}^*$ es el conjunto de todas las cadenas binarias (incluida λ), la longitud en bits de X y Y es $|X|$ y $|Y|$ respectivamente. Nótese que $|\lambda| = 0$. Para ambas cadenas binarias X y Y, la concatenación se denota como $X \| Y$.

Definición 17. Sea una cadena binaria un *bloque completo* (o *bloque incompleto*) X, si $|X| = n$ (o si $|X| < n$ respectivamente). Se denota al conjunto de todos los bloques completos (o incompletos) como $\mathcal{B}$ (o $\mathcal{B}^<$ respectivamente).

Definición 18. Sea $\mathcal{B}^{\leq} := \mathcal{B}^{<} \cup \mathcal{B}$ el conjunto de todos los bloques. Para $B \in \mathcal{B}^{\leq}$ se define a $\overline{B}$ de la siguiente manera:

$$\overline{B} := \begin{cases} 0^n & \text{si } B = \lambda \\ B \| 10^{n-1-|B|} & \text{si } B \neq \lambda \text{ y } |B| < n \\ B & \text{si } |B| = n \end{cases} \tag{2.8}$$

Definición 19. La función *techo* se define como:

$$y = \lceil x \rceil := y = \{y : y \in \mathbb{Z} \ \wedge \ x \in \mathbb{R} \ \wedge \ y - 1 < x \leq y\} \tag{2.9}$$

Definición 20. Dada la cadena binaria $Z \in \{0,1\}^*$ su procesamiento para separarla en bloques de n bits se define como:

$$(Z[1], Z[2], ..., Z[z]) \xleftarrow{n} Z \tag{2.10}$$

Donde el número de bloques está dado como $z = \left\lceil \frac{|Z|}{n} \right\rceil$, además la longitud para cada bloque es $\left|Z[i]\right| = n$ para todo $i < n$ y $1 \leq \left|Z[z]\right| \leq n$, es decir que la longitud de cada bloque es de al menos un bit o a lo sumo de n bits, tal que $Z = (Z[1] \| Z[2] \| ... \| Z[z])$. Si $Z = \lambda$, entonces $z = 1$ y $Z[1] = \lambda$. Denótese $\|Z\| = z$ al número de bloques en Z.

Definición 21. Dada cualquier secuencia $Z = (Z[1], ..., Z[s])$ y $1 \leq a \leq b \leq s$ para algunos enteros a, b y s; la subsecuencia $(Z[a], ..., Z[b])$ es representada como $Z[a...b]$.

Definición 22. Para dos cadenas binarias X y Y, donde $|X| \geq |Y|$, las operaciones XOR extendidas se definen como:

$$X \underline{\oplus} Y = X\left[1...|Y|\right] \oplus Y \tag{2.11}$$

y

$$X \overline{\oplus} Y = X \oplus (Y \| 0^{|X|-|Y|}) \tag{2.12}$$

donde $(X[1] \| X[2] \| ... \| X[x]) \xleftarrow{1} X$ (es decir, se separa en bits) y por lo tanto $X\left[1...|Y|\right]$ denota a los primeros $|Y|$ bits de X. Si $|X| = |Y|$, entonces las operaciones XOR extendidas se reducen a la operación XOR ordinaria.

2.2. Cifradores por bloques ligeros

Existe un gran número de trabajos que proponen cifradores por bloques ligeros. En seguida se mencionan a los de mayor importancia para el presente trabajo.

2.2.1. PRESENT

En 2007 se publica el trabajo de Andrey Bogdanov *et. al.*[2] en la Conferencia sobre Hardware Criptográfico y Sistemas Empotrados (*CHES* 2007), donde fue propuesto el cifrador por bloques *PRESENT* como una alternativa al estándar AES.

PRESENT fue pensado para ser eficiente con respecto al área ocupada al ser implementado en hardware y en el consumo de energía. Resultó una opción atractiva para ser implementado en sistemas con recursos restringidos.

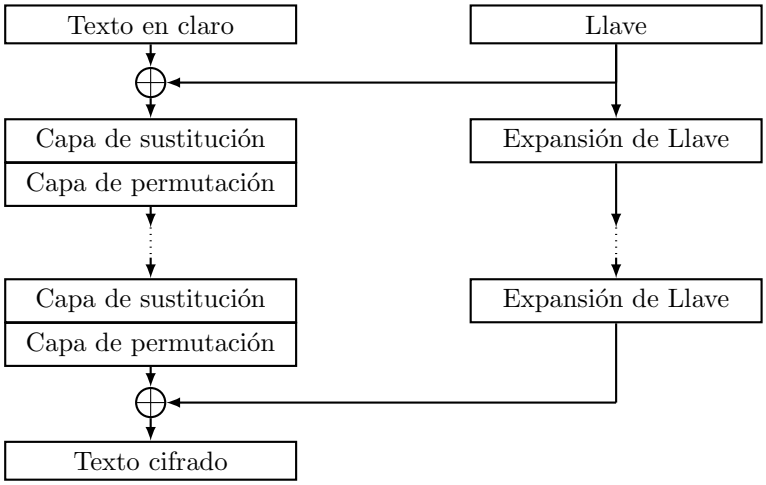

Figura 2.1: Arquitectura de PRESENT.

PRESENT es una red de sustituciones y permutaciones (SPN[3]) de 33 rondas. Trabaja con bloques de 64 bits y soporta llaves de 80 y 128 bits de

[3]Una red de sustituciones y permutaciones es un algoritmo que toma como entradas un bloque de texto en claro y una llave para aplicarles transformaciones de sustitución y permutación.

longitud. Cada ronda consta de una permutación a nivel de bits y una capa de sustitución no lineal, en la que se utiliza una Caja-S de 16 elementos de 4 bits de longitud para cada uno, tal que para cada elemento $Sb \in$ Caja-S se tiene que $Sb : \{0,1\}^4 \rightarrow \{0,1\}^4$. Su arquitectura se ilustra con el diagrama de bloques de la Figura 2.1.

Se considera una buena alternativa a AES, sin embargo no resulta ser la primitiva más económica, ya que ha sido optimizado en trabajos posteriores como GIFT [7] que se explica a continuación.

2.2.2. GIFT

Subhadeep Banik *et. al.* presentan en su publicación [7] el cifrador GIFT en el cual se revisó la construcción de PRESENT y se optimizó para ser más pequeño, se eligió una Caja-S más económica lo que lo hace más rápido y ligero.

En su trabajo se proponen dos versiones: GIFT-64-128 y GIFT-128-128, procesando bloques de entrada de 64 bits en 28 rondas, y de 128 bits en 40 rondas respectivamente. En ambas versiones la llave tiene una longitud de 128 bits. Su arquitectura se ilustra con el diagrama de bloques de la Figura 2.2.

GIFT obtuvo una ventaja sobre PRESENT (ambos en sus versiones de 64 bits) con respecto al área ocupada. 1345 GE[4] de GIFT contra 1560 GE de PRESENT.

GIFT resultó más ventajoso sobre su antecesor PRESENT (en su versión de 64 bits). Sin embargo, para su implementación en un esquema de cifrado de tipo *libre-de-inverso*[5] (COFB por ejemplo) la etapa de descifrado no es necesaria, existen otras alternativas (Midori[8] por ejemplo) en las que se utiliza la misma rutina tanto para la función de cifrado, como la de descifrado.

[4]El concepto de compuerta equivalente (GE) permite medir el área de un circuito independientemente de la tecnología utilizada. Una GE es interpretada como una compuerta NAND o NOR de dos entradas.

[5]El concepto de *Inverse-Free* se refiere a diseños que utilizan únicamente el núcleo de sólo cifrado para realizar el proceso de cifrado y descifrado del mensaje, de tal forma que nunca se utiliza el núcleo de sólo descifrado. Con lo anterior se tienen ventajas en el espacio ocupado por el circuito.

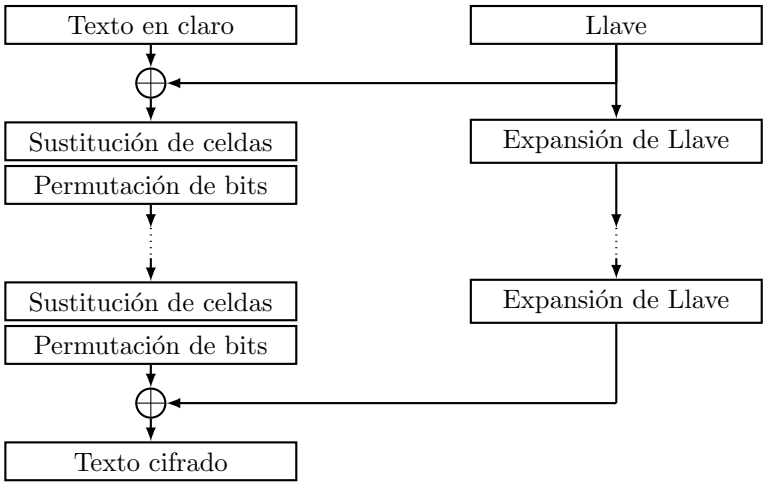

Figura 2.2: Arquitectura de GIFT.

2.2.3. SIMON

Ray Beaulieu *et. al.* presentan en 2013 una nueva familia de cifradores por bloques, en su trabajo publicado como *The SIMON and SPECK Families of Lightweight Block Ciphers*[9] son propuestos SIMON (pensado para ser eficiente en hardware) y SPECK (optimizado para ser implementado en software). Uno de los aspectos importantes en estas dos familias es su flexibilidad, ya que son capaces de trabajar con varios tamaños de bloque y de llave tal como lo muestra la Tabla 2.1.

Tamaño del bloque	**Tamaño de la llave**
32	64
48	72, 96
64	96, 128
96	96, 144
128	128, 192, 256

Tabla 2.1: Tamaños en los parámetros de SIMON y SPECK.

En los resultados reportados, la implementación de SIMON (el cifrador optimizado para hardware) ocupó 1000 GE, por lo que es una buena opción para implementaciones ligeras, sin embargo debe tenerse en cuenta que fue

desarrollado de manera directa por la NSA[6].

2.2.4. Midori

En 2015 Subhadeep Banik *et. al.* publican en su trabajo[8] un nuevo cifrador por bloques, pensado para trabajar en dispositivos con recursos limitados, logrando reducir su consumo de energía. Midori además de ser ligero, también es económico en su consumo de potencia ya que se desarrollaron capas lineales y no lineales eficientes en términos de energía.

Sin embargo, para garantizar su seguridad frente a varios tipos de ataques, se implementa el mayor número posible de rondas sin superar la cantidad necesaria en una construcción basada en una red de Feistel. La arquitectura elegida está basada en una red de sustituciones y permutaciones (SPN). Si el bloque a procesar es visto como una matriz cuadrada de elementos, en lugar de utilizar operaciones de desplazamiento de filas (como en el caso de AES), se eligen operaciones de permutación de los elementos del estado.

Una ventaja importante resulta de reducir el número de elementos en la Caja-S. Las operaciones con las Cajas-S son críticas en la ejecución del algoritmo, pues el cómputo depende en gran medida de ellas, por lo tanto se construyeron Cajas-S ligeras de 4 bits y de un retardo pequeño. Otra característica importante en Midori es el diseño de componentes involutivos, por lo que el proceso de descifrado utiliza los mismos elementos usados que el proceso de cifrado, por lo tanto se logra un menor área en la implementación del circuito.

Midori se presenta en dos versiones: Midori64 y Midori128, procesando bloques de 64 bits y 128 bits respectivamente. En ambos casos la llave tiene una longitud de 128 bits. Opera con un estado expresado como una matriz cuadrada de 4x4, donde cada uno de sus elementos tiene una longitud de 4 bits y 8 bits para su versión correspondiente (Midori64 y Midori128, respectivamente).

[6]Un grupo de expertos en criptografía se ha opuesto a los esfuerzos que la Agencia de la Seguridad Nacional (NSA) de los Estados Unidos ha realizado para estandarizar los algoritmos SIMON y SPECK. Argumentan debilidades deliberadas en los algoritmos, además de la participación anterior de la NSA en la creación del algoritmo criptográfico Dual-EC-DRBG con vulnerabilidades premeditadas. (Consultado el 7 de agosto de 2018, en `https://www.reuters.com/article/us-cyber-standards-insight/distrustful-u-s-allies-force-spy-agency-to-back-down-in-encryption-fight-idUSKCN1BW0GV`)

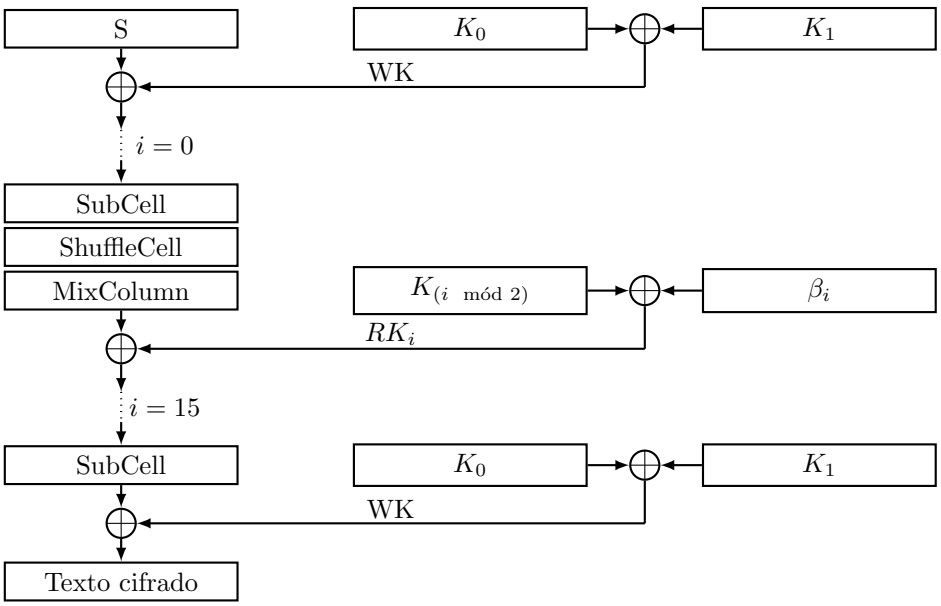

Figura 2.3: Arquitectura de Midori64.

La capa lineal consiste de una permutación de celdas ($ShuffleCell$) y operaciones con matrices cuadradas de 4x4 ($MixColumn$), en las cuales, todas sus operaciones se realizan dentro de los campos $GF(2^4)$ y $GF(2^8)$ según la versión (de 64 bits o de 128 bits, respectivamente). Su arquitectura se ilustra con el diagrama de bloques de la Figura 2.3.

En los resultados, Midori64 se implementó con 1542 GE para su versión de sólo cifrado, mientras que Midori128 ocupó 2522 GE para el núcleo de sólo cifrado.

Midori resulta ser una buena opción para ser implementado en soluciones que, debido a sus especificaciones, necesitan ser eficientes en términos del consumo de energía, sin comprometer la seguridad del sistema.

2.2.5. Atomic-AES

En el trabajo[10] publicado por Subhadeep Banik *et. al.* en 2016, se presenta una implementación optimizada de AES. En algunos modos de operación

(por ejemplo: CBC[7]) es necesario tener acceso tanto al núcleo de cifrado como al núcleo de descifrado. En otros trabajos (por ejemplo en la publicación de Moradi et al.[11]) se ha obtenido una mejora en la implementación, ya sea del núcleo de cifrado o del núcleo de descifrado, pero no en ambos a la vez.

En la Figura 2.4 se muestra la arquitectura general del cifrador AES en su versión de $n = 128$ bits. Recibe un mensaje M, una llave K y devuelve un mensaje cifrado C, todos ellos de longitud n bits.

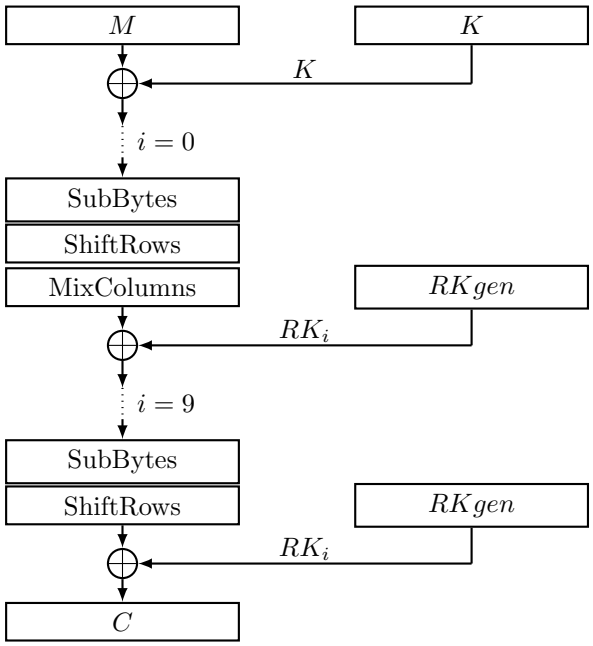

Figura 2.4: Arquitectura de AES128.

Una estrategia utilizada que resulta interesante, es la descomposición de la matriz inversa usada en el procedimiento denominado $MixColumn^{-1}$, en el trabajo publicado por Akashi Satoh *et. al.*[12] se propone la descomposición

[7]En el modo de operación CBC (*Cipher-block chaining*) en cada bloque se aplica una operación lógica XOR al bloque de texto con el bloque cifrado anterior, por lo que cada bloque cifrado depende de todos los bloques de texto en claro anteriores. Requiere de un vector de inicialización (IV). En su modo de cifrado su operación se define como $C_i = E_k(P_i \oplus C_{i-1})$ con $C_0 = IV$, mientras que en su modo de descifrado su definición es $P_i = D_k(C_i) \oplus C_{i-1}$ con $C_0 = IV$

ilustrada con la Ecuación 2.13, con la que se logró una implementación usando 193 compuertas XOR y un multiplexor de 32 bits:

$$\begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} + \begin{bmatrix} 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 \end{bmatrix} + \begin{bmatrix} 4 & 0 & 4 & 0 \\ 0 & 4 & 0 & 4 \\ 0 & 4 & 0 & 4 \\ 4 & 0 & 4 & 0 \end{bmatrix} \tag{2.13}$$

Sin embargo, en Atomic-AES Paulo Barreto *et. al.* presenta una descomposición más eficiente, como lo muestra la Ecuación 2.14:

$$\begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 5 & 0 & 4 & 0 \\ 0 & 5 & 0 & 4 \\ 4 & 0 & 5 & 0 \\ 0 & 4 & 0 & 5 \end{bmatrix} \tag{2.14}$$

Con lo que la implementación de la etapa $MixColumn$ requiere de sólo 108 compuertas XOR y un multiplexor de 32 bits, logrando un diseño pequeño para el circuito como lo ilustra la Figura 2.5.

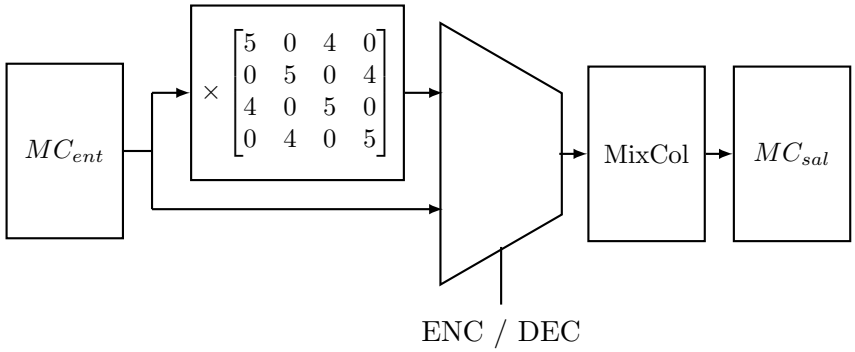

Figura 2.5: Diseño del circuito que implementa la etapa $MixColumn$.

El área total ocupada por la implementación de Atomic-AES es de 2645 compuertas equivalentes, por lo tanto aún con las mejoras propuestas, no resulta una alternativa competitiva frente a otras soluciones como PRESENT, GIFT o Midori (por mencionar algunos ejemplos).

2.3. Esquemas de cifrado autenticado

Existe un gran número de trabajos que proponen esquemas de cifrado diseñados para ser implementados en un circuito pequeño. A continuación se mencionan los de mayor importancia para el presente trabajo.

2.3.1. CLOC

CLOC[3] es un esquema de cifrado autenticado con datos asociados presentado en la competencia CAESAR. Diseñado para ser una optimización de los esquemas CCM[13], EAX[14] y EAX-prime[15], con respecto al exceso de recursos necesarios para su implementación, adicionales a los requeridos por la primitiva de cifrado, la complejidad en el cómputo previo y la memoria empleada para su ejecución. Los algoritmos de cifrado y descifrado se muestran en la Figura 2.6.

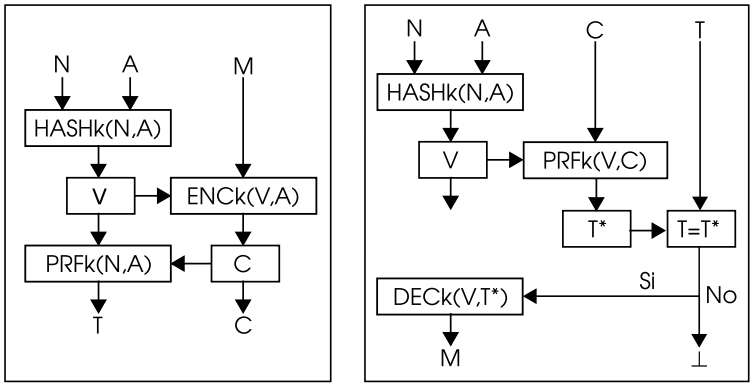

Figura 2.6: Algoritmo de cifrado (a la izquierda) y de descifrado (a la derecha) de CLOC.

CLOC es un esquema basado en un *Nonce* que brinda servicios de cifrado autenticado con datos asociados. Además, sólo requiere la función de cifrado del cifrador por bloques, tanto para el proceso de cifrado como el de descifrado.

En su algoritmo (de cifrado y descifrado), CLOC utiliza cuatro subrutinas: HASH, PRF, ENC y DEC, las cuales se ilustran en las Figuras 2.7, 2.8 y 2.9.

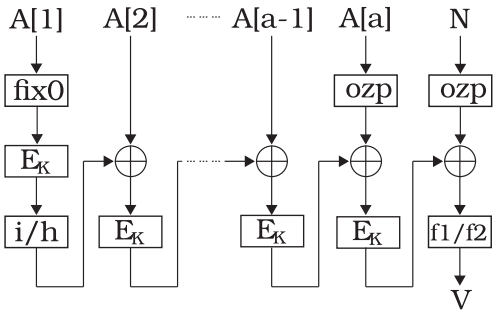

Figura 2.7: Algoritmo de la función picadillo (HASH) de CLOC.

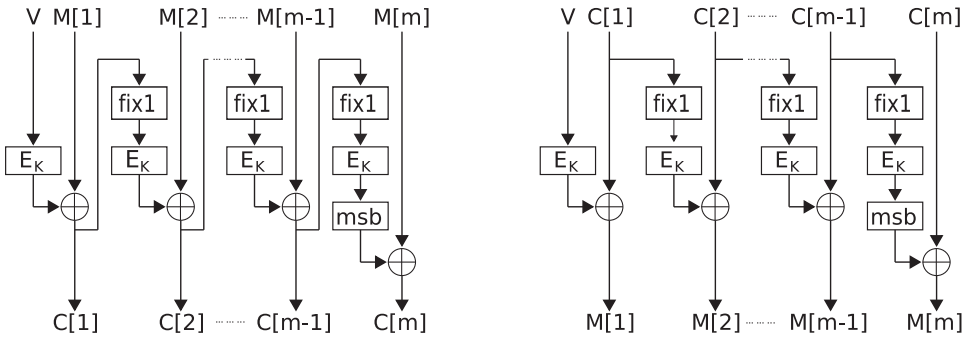

Figura 2.8: Algoritmos de las funciones de cifrado (izquierda) y descrifrado (derecha) de CLOC.

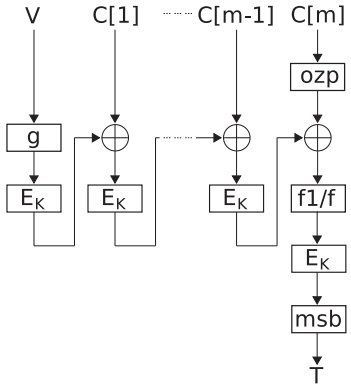

Figura 2.9: Algoritmo de la función pseudo aleatoria (PRF) de CLOC.

CLOC fue implementado con AES y probado sobre un procesador de propósito general (Intel Core i5-3427U a 1.80GHz), donde se aprovecharon las características de paralelización ofrecidas por la arquitectura. También se implementó sobre un microcontrolador Atmega128[8].

Además de implementarlo y probarlo en software, se realizó una implementación en hardware. Se empleó un dispositivo FPGA[9] Cyclone IV GX de Altera[10] (EP4CGX110DF31C7). La implementación se comparó con EAX utilizando para ambos casos un núcleo AES128. Los resultados con respecto al tamaño se reportaron en términos de elementos lógicos (LE). En la Tabla 2.2 se muestran los resultados.

	Tamaño (LE)	**Frecu. max. (Mhz)**	**Tasa de salida (Mbit/seg)**
CLOC	5628	82.1	400.7
EAX	6453	61.3	342.2
AES Cif	3175	98.7	971.7

Tabla 2.2: Resultados de la implementación de CLOC, EAX y el núcleo de sólo cifrado AES128.

2.3.2. SILC

Propuesto en la competencia CAESAR (junto con CLOC), SILC[4] (*SImple Lightweight CFB*) es un modo de operación de llave simétrica que ofrece servicios de cifrado autenticado con datos asociados. Ofrece una optimización con respecto al área ocupada en la implementación de CLOC. Los algoritmos de cifrado y descifrado de SILC resultan idénticos al caso de CLOC,

[8]El microcontrolador Atmega128 es un dispositivo programable que ejecuta un conjunto de instrucciones de 8 bits. Es parte de la familia de microcontroladores AVR de Atmel (recientemente adquirida por Microchip). Cuenta con un convertidor de señales analógicas a digitales (ADC), un convertidor de señales digitales a analógicas (DAC), reloj de tiempo real, temporizadores, e interfaces para comunicación serial síncronas y asíncronas operadas sobre los protocolos SPI e I2C. Tiene una capacidad de almacenamiento estático para memoria de programa de 4 KB, una memoria flash de 128 KB y 4 KB de memoria volátil. Atmega128 se hizo popular cuando fue incorporado en el diseño original de la placa de desarrollo Arduino UNO.

[9]Matriz de Compuertas Programables en Campo (FPGA).

[10]Altera es una empresa líder en la fabricación de dispositivos programables, pionera en la introducción de PLDs. Uno de sus productos más importantes es el entorno de desarrollo Quartus II, el cual cuenta con soporte para lenguajes de descripción de hardware como VHDL, Verilog y AHDL (desarrollado por Altera), además de un simulador. El 1 de junio de 2015 fue adquirida por Intel.

debido a que pertenecen a la misma familia. SILC utiliza cuatro subrutinas: HASH, PRF, ENC y DEC. Las cuales se ilustran en las Figuras 2.10, 2.11 y 2.12.

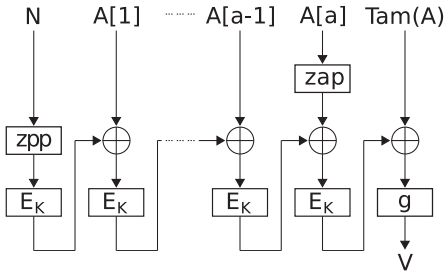

Figura 2.10: Algoritmo de la función picadillo (HASH) de SILC.

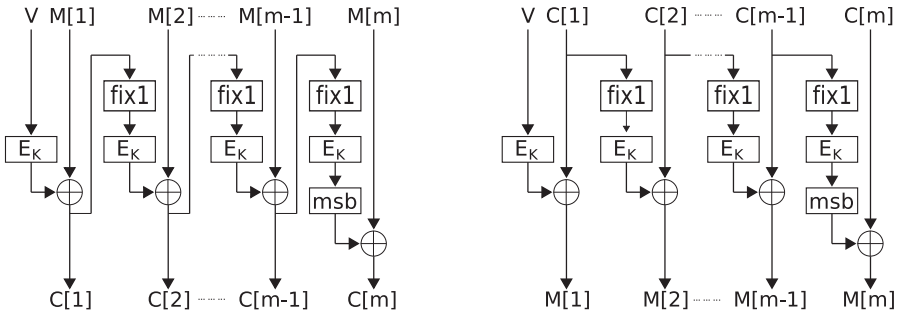

Figura 2.11: Algoritmos de las funciones de cifrado (izquierda) y descrifrado (derecha) de SILC.

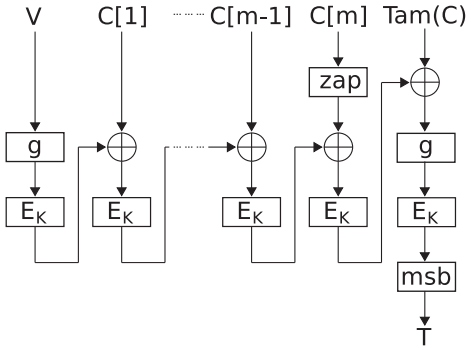

Figura 2.12: Algoritmo de la función pseudo aleatoria (PRF) de SILC.

Los logros de SILC con respecto a la confidencialidad del mensaje y su integridad, a los datos asociados y al *Nonce*, son los mismos que los ofrecidos por CLOC.

2.3.3. COFB

En 2017 Avik Chakraborti *et. al.* publicaron en su trabajo [1] un nuevo diseño para cifrado autenticado con datos asociados llamado COFB (*combined feedback*), orientado para ser implementado en sistemas área reducida (para hardware) y económicos en consumo de memoria (para software).

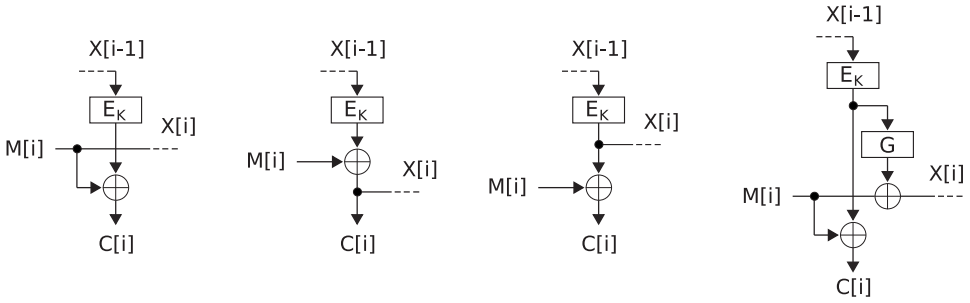

Figura 2.13: Tipos de modo de retroalimentación en esquemas anteriores. A la derecha el modo de retroalimentación propuesto en COFB.

Como característica importante en la seguridad de COFB, se propone un modo de retroalimentación donde se combinan modos utilizados en esquemas anteriores, tal como lo ilustra la Figura 2.13.

COFB utiliza un cifrador por bloques de n bits, como máscara necesita un bloque de $n/2$ bits y además, un *Nonces* que agrega seguridad al esquema. COFB soporta hasta $O(2^{n/2})$ consultas, lo que le da seguridad dentro de los límites acotados por la paradoja del cumpleaños[11]. Además, el esquema

[11]La paradoja del cumpleaños establece que dado un conjunto de n personas dentro de alguna habitación, la probabilidad de encontrar a una pareja que cumpla años el mismo día, está dada de la siguiente manera:

$$1 - p = \begin{cases} 1 - \frac{365!}{365^n (365-n)!} & \text{si } 1 \leq n \leq 365 \\ 1 & \text{si } 365 < n \end{cases} \tag{2.15}$$

Por ejemplo: supóngase un conjunto de 23 personas, la probabilidad de encontrar alguna pareja que cumpla años el mismo día está dada como: $1 - (\frac{364}{365})^n = 0{,}597$. Si pensamos en

sólo necesita el núcleo de sólo cifrado de la primitiva elegida, es decir, no es necesario implementar el núcleo de descifrado, lo que lo define como un esquema *libre-de-inverso*.

Se ha comparado la implementación en hardware de COFB con otros esquemas publicados en la base de datos ATHENA[12]. Se tomaron a los dispositivos Virtex 6 y Virtex 7 como plataformas para la implementación.

En la publicación[1] se muestran los resultados obtenidos de realizar una implementación en hardware con un cifrador por bloques AES (serializado y en una versión ligera), sin embargo, el esquema no ha sido implementado usando otras familias de cifradores (por ejemplo Midori[8]).

el conjunto de personas como el espacio de mensajes en claro y, al hecho de cumplir años el mismo día como una colisión de dos mensajes a los que les corresponde la misma cifra, entonces la paradoja del cumpleaños aporta una noción de la seguridad en el cifrador en cuestión.

[12]El proyecto, iniciado por la Universidad George Mason, es un conjunto de Herramientas Automatizadas para la evaluación de sistemas criptográficos diseñados para hardware (*Automated Tools for Hardware EvaluatioN: ATHENa*). Los sistemas criptográficos pueden ser implementaciones para FPGA, SoC y sistemas ASIC, con lo que facilita la comparación de trabajos propuestos en la competencia CAESAR. Las plataformas soportadas están basadas en dispositivos Xilinx Virtex 7 y Altera Stratix V (por ejemplo). Los diseños pueden ser desarrollados en lenguajes como VHDL y Verilog. Es compatible con proyectos de entornos de desarrollo como Xilinx ISE v. 14.7 y Xilinx Vivado 2015.2

2.4. Comparación de algunas implementaciones de esquemas en Hardware

En la Tabla 2.3 se compara el esquema COFB con otros esquemas publicados en la base de datos ATHENa, donde el área ocupada se expresa en LUTs[13] y Slices[14].

Esquema	**Primitiva**	**#LUT**	**#Slices**	**Gbps**	**Mbps/LUT**	**Mbps/Slices**
ACORN[16]	SC	455	135	3.112	6.840	23.052
AEGIS[17]	BC-RF	7592	2028	70.927	9.342	34.974
AES-COPA[18]	BC	7754	2358	2.500	0.322	1.060
AES-GCM[19]	BC	3175	1053	3.239	1.020	3.076
AES-OTR[20]	BC	5102	1385	2.741	0.537	1.979
AEZ[21]	BC-RF	4597	1246	8.585	0.747	2.756
ASCON[22]	Sponge	1271	413	3.172	2.496	7.680
CLOC[3]	BC	3145	891	2.996	0.488	1.724
DEOXYS[23]	TBC	3143	951	2.793	0.889	2.937
ELmD[24]	BC	4302	15840	3.168	0.736	2.091
JAMBU-AES[25]	BC	1836	652	1.999	1.089	3.067
JAMBU-SIMON[25]	BC (non-AES)	1222	453	0.363	0.297	0.801
Joltik[26]	TBC	1292	442	0.853	0.660	0.826
Ketje[27]	Sponge	1270	456	7.345	5.783	16.107
Minalpher[28]	BC (non-AES)	2879	1104	1.831	0.636	1.659
NORX[29]	Sponge	2964	1016	11.029	3.721	10.855
PRIMATES-HANUMAN[30]	Sponge	1012	390	0.964	0.953	2.472
OCB[5]	BC	4249	1348	3.122	0.735	2.316
SCREAM[31]	TBC	2052	834	1.039	0.506	1.246
SILC[4]	BC	3066	921	4.040	1.318	4.387
Tiaoxin[32]	BC-RF	7123	2101	52.838	7.418	25.149
TriviA-ck[33]	SC	2118	687	15.374	7.259	22.378
COFB	BC	1075	442	2.850	2.240	6.450

Tabla 2.3: Comparación de COFB con esquemas en ATHENa.

[13](*Look-Up Table*) también llamados "tablas de consulta", son componentes utilizados para implementar funciones booleanas, se pueden modelar como un multiplexor de n entradas de selección y una máscara de 2^n bits.

[14]Un Slice contiene un determinado número de LUTs, memoria, Flip Flops y multiplexores

Capítulo 3

Desarrollo

En este capítulo se explica de manera amplia el desarrollo del trabajo experimental. Primero se aborda la implementación en software tanto de los cifradores AES128 y Midori64, como del esquema COFB con cada uno de ellos. Luego, se detalla el proceso de diseño para hardware con VHDL, tanto de los cifradores AES128 y Midori64, como del esquema COFB. Además de revisar las estrategias utilizadas para la reducción de la cantidad de recursos utilizados por los diseños dentro del dispositivo FPGA.

3.1. Implementación en software

La publicación de Avik Chakraborti et. al.[1] donde es propuesto el esquema COFB no proporciona vectores de prueba, para obtenerlos resulta necesario desarrollar las versiones en software de los cifradores AES128 y Midori64, así como del esquema COFB. La correctitud de la posterior implementación en hardware se comprueba con los vectores de prueba generados por la implementación en software.

3.1.1. AES128

Para la implementación en software se revisó AES128, el cual se ilustra en el Algoritmo 3.1, donde $k = n = 128$ y $R = 10$.

AES-NI es una extensión para el conjunto de instrucciones de la arquitectura x86 en procesadores Intel y AMD. Fue propuesto en el año 2008 para proveer eficiencia en el proceso de cifrado o descifrado con AES.

Algoritmo 3.1 Algoritmo AES128

Require: $K \in \{0,1\}^k, RK_0 \in \{0,1\}^k, ..., RK_9 \in \{0,1\}^k, M \in \{0,1\}^n$
Ensure: $C \in \{0,1\}^n$
1: $S \leftarrow AddKey(M, K)$
2: **for** $i = 0$ to $R - 2$ **do**
3: $S \leftarrow SubBytes(S)$
4: $S \leftarrow ShiftRows(S)$
5: $S \leftarrow MixColumns(S)$
6: $S \leftarrow AddKey(S, RK_i)$
7: **end for**
8: $S \leftarrow SubBytes(S)$
9: $S \leftarrow ShiftRows(S)$
10: $S \leftarrow AddKey(S, RK_9)$

El núcleo de cifrado AES128 se implementó utilizando el conjunto AES-NI, incorporando una interfaz para su posterior integración en COFB.

Código 3.1: Rutina principal de la implementación de AES128.

```
1  byte aes(unsigned char *out,const unsigned char *in,const char *key)
2          {
3          unsigned char llave[0x10], estad[0x10], i=0, j;
4          __m128i tmp = mm_loadu_si128(&((__m128i*)in)[i]);
5
6          tmp = _mm_xor_si128(tmp,((__m128i*)key)[0]);
7          for(j=1; j<10; j++){
8                  tmp = _mm_aesenc_si128(tmp,((__m128i*)key)[j]);
9                  _mm_storeu_si128(&((__m128i*)llave)[i],((__m128i*)key)[j]);
10                 _mm_storeu_si128(&((__m128i*)estad)[i],tmp);
11                 }
12         tmp = _mm_aesenclast_si128(tmp,((__m128i*)llvRnd)[j]);
13         _mm_storeu_si128(&((__m128i*)out)[i],tmp);
14
15         return(0);
16         }
```

La primera ejecución del procedimiento $AddKey$ es implementada con la instrucción `_mm_xor_si128`, dando como resultado la operación: $S = M \oplus K$.

Para cada ronda $i \in \{0, ..., R-1\}$, los procedimientos $SubBytes$, $ShiftRows$, $MixColumn$ y $AddKey$ son implementados con la instrucción `_mm_aesenc_si128`. La última ronda se implementa con la instrucción `_mm_aesenclast_si128`.

Las instrucciones del conjunto AES-NI no cuentan con una función de derivación de llaves de ronda, por lo que es necesario implementar una función auxiliar con la cual se obtienen las llaves de ronda utilizadas posteriormente en la rutina principal mostrada en el Código3.1, En el Código 3.2 se muestra la función auxiliar para el cómputo de las llaves de ronda.

Código 3.2: Función auxiliar para la derivación de llaves de ronda.

```
1  inline m128 subLlavesAux(m128 temp1, m128 temp2)
2          {
3          m128 temp3;
4          temp2 = _mm_shuffle_epi32 (temp2 ,0xff);
5          temp3 = _mm_slli_si128 (temp1, 0x4);
6          temp1 = _mm_xor_si128 (temp1, temp3);
7          temp3 = _mm_slli_si128 (temp3, 0x4);
8          temp1 = _mm_xor_si128 (temp1, temp3);
9          temp3 = _mm_slli_si128 (temp3, 0x4);
10         temp1 = _mm_xor_si128 (temp1, temp3);
11         temp1 = _mm_xor_si128 (temp1, temp2);
12         return temp1;
13         }
```

El conjunto AES-NI ofrece instrucciones para implementar la rutina de descifrado, sin embargo no se ocupa en el esquema COFB, por lo tanto no fue implementada.

Los vectores de prueba obtenidos se muestran en el Código A.1 del Apéndice A.

El código fuente de la implementación está disponible en el Enlace B.1.1 del Apéndice B.

3.1.2. Midori64

Para la implementación en software se revisó Midori64, el cual se ilustra en el Algoritmo 3.2, donde $k/2 = n = 64$ y $R = 16$.

El programa fue realizado en lenguaje C, el dato de entrada M es almacenado en un entero de n bits. La llave K se almacena en un arreglo de 2 enteros de n bits. La salida se devuelve a través de un entero de n bits.

Algoritmo 3.2 Algoritmo Midori64.

Require: $M \in \{0,1\}^n, WK \in \{0,1\}^{k/2}, RK_0 \in \{0,1\}^{k/2}, ..., RK_{R-2} \in \{0,1\}^{k/2}$
Ensure: $C \in \{0,1\}^n$
1: $S \leftarrow KeyAdd(M, WK)$
2: **for** $i = 0$ to $R - 2$ **do**
3: $S \leftarrow SubCell(S)$
4: $S \leftarrow ShuffleCell(S)$
5: $S \leftarrow MixColumn(S)$
6: $S \leftarrow KeyAdd(S, RK_i)$
7: **end for**
8: $S \leftarrow SubCell(S)$
9: $C \leftarrow KeyAdd(S, WK)$

Los vectores de prueba obtenidos se muestran en el Código A.2 del Apéndice A.

El código fuente de la implementación está disponible en el Enlace B.1.2 del Apéndice B.

3.1.3. COFB

Para la implementación en software se revisó COFB, el cual se ilustra en el Algoritmo 3.3.

En los programas realizados en lenguaje C de ambas versiones (COFB-AES128 y COFB-Midori64), los datos de entrada K, N, A y M (de longitudes k, $n/2$, a y m bits respectivamente) son almacenados en arreglos de enteros de 8 bits de longitud variable (con el uso de la instrucción *realloc*() de la biblioteca *stdlib.h* de C). Las salidas T y C (de longitudes $n/2$ y $c = m$ respectivamente), son devueltos también en arreglos de enteros de 8 bits de longitud variable.

De acuerdo con la versión (COFB-AES128 y COFB-Midori64) implementada, la llamada al cifrador se realiza con el parámetro M de la longitud correspondiente ($n = 128$ o $n = 64$ bits respectivamente). La longitud de la llave K es constante para ambas versiones: $k = 128$ bits.

Algoritmo 3.3 Algoritmo COFB

Require: $K \in \{0,1\}^k, N \in \{0,1\}^{n/2}, A \in \{0,1\}^r, M \in \{0,1\}^s$
Ensure: $C \in \{0,1\}^m, T \in \{0,1\}^n$
1: $(\Delta, Y[0]) \leftarrow MaskGen(K, N)$
2: $(A[1], ..., A[a]) \xleftarrow{n} A$
3: $(M[1], ..., M[m]) \xleftarrow{n} M$
4: $l \leftarrow a + m$
5: $((B[1], t[1]), ..., (B[l], t[l])) \leftarrow Frm(A, M)$
6: **for** $i = 1$ to l **do**
7: $\quad X[i] \leftarrow (B[i] \oplus G \cdot Y[i-1]) \mathbin{\overline{\oplus}} mask_\Delta(t[i])$
8: $\quad Y[i] \leftarrow E_K(X[i])$
9: $\quad$ **if** $i > a$ **then**
10: $\quad\quad C[i-a] \leftarrow Y[i-1] \underline{\oplus} M[i-a]$
11: $\quad$ **end if**
12: **end for**
13: $T \leftarrow Y[l]$

Los vectores de prueba obtenidos para COFB-AES128 y COFB-Midori64 se muestran en los Códigos A.3 y A.4 del apéndice A, respectivamente.

Los códigos fuente de las implementaciones de COFB-AES128 y COFB-Midori64 están disponibles en los Enlaces B.1.3 y B.1.4 del apéndice B, respectivamente.

3.2. Implementación en hardware

Si S es el estado a procesar por el cifrador por bloques, tal que $|S| = n$ es su longitud en bits, entonces se dice que una implementación procesa el ancho total de la entrada S cuando es utilizado hardware de n bits; es decir, se procesa el bloque completo en cada ciclo de reloj.

Por otra parte, se dice que un cifrador por bloques procesa la entrada S de manera serializada cuando es utilizado hardware de n/m bits; es decir se procesan fragmentos de longitud n/m bits de la entrada S en cada ciclo de reloj.

Cuando un algoritmo hace uso de datos constantes, puede utilizarse para su implementación un registro (o memoria) para almacenarlos previamente,

o calcularlos al vuelo. El método que hace uso del almacenamiento agiliza la ejecución del algoritmo, sin embargo cuando se trata de una gran cantidad de datos, compromete el área ocupada por el diseño. El método que ocupa un diseño combinacional, por lo general permite producir circuitos compactos aunque lentos.

Para la implementación en hardware, se utilizan la estrategia de serialización, y los métodos tanto de almacenamiento como de diseño combinacional en el diseño de los cifradores y del esquema. La estrategia de diseño considerando el ancho total no es implementada, ya que por definición no es una estrategia ligera.

3.2.1. AES128

En el Algoritmo 3.1 se realiza el cómputo previo de las llaves de ronda $RK_0, ..., RK_{R-1}$, en el cual se emplean componentes utilizados en los procedimientos $SubBytes$ y $ShiftRows$. Para reutilizar los componentes de la implementación, es diseñada una unidad de control, la cual consiste en una máquina de estados que controla las señales de los componentes de la implementación.

Algoritmo 3.4 AES128 reutilizando componentes.

Require: $M \in \{0,1\}^n, K \in \{0,1\}^k$
Ensure: $C \in \{0,1\}^n$
1: $RK_0 \leftarrow K$
2: **for** $i = 0$ to $R - 2$ **do**
3: $\quad RK_{i+1} \leftarrow RoundKey(RK_i, i)$
4: **end for**
5: $S \leftarrow KeyAdd(M, RK_0)$
6: **for** $i = 0$ to $R - 2$ **do**
7: $\quad S \leftarrow SubBytes(S)$
8: $\quad S \leftarrow ShiftRows(S)$
9: $\quad S \leftarrow MixColumns(S)$
10: $\quad S \leftarrow AddKey(S, RK_{i+1})$
11: **end for**
12: $S \leftarrow SubBytes(S)$
13: $S \leftarrow ShiftRows(S)$
14: $S \leftarrow AddKey(S, RK_{i+1})$

El Algoritmo 3.1 es reorganizado para lograr una implementación orientada al ahorro de área como lo ilustra el Algoritmo 3.4, en donde puede observarse que el cálculo de las llaves de ronda es considerado dentro de la misma implementación, con lo que se tiene en cuenta la reutilización de los componentes en el sistema.

El Algoritmo para el cálculo de las llaves involucra al conjunto de constantes de ronda α definido en la Tabla 3.1.

i	0	1	2	3	4	5	6	7	8	9
α_i	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80	0x1b	0x36

Tabla 3.1: Constantes de ronda en AES128

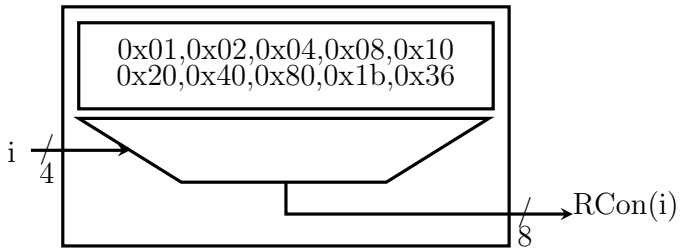

Figura 3.1: *RconGen* con almacenamiento de AES128.

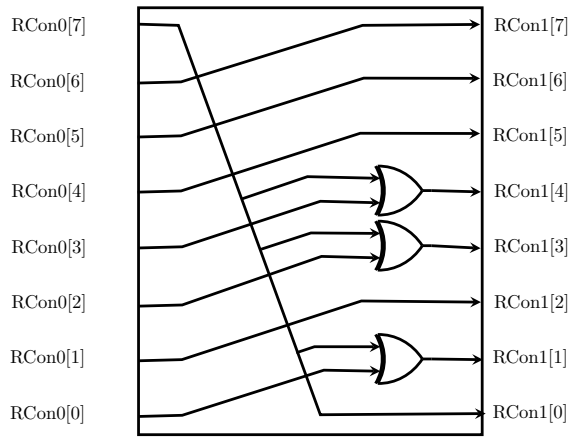

Figura 3.2: *RconGen* combinacional de AES128.

Las constantes α_i resultan de la operación 2^i dentro del campo $GF(2^8)$, con $i \in 0, ..., R-1$. Cuando se utiliza como método de implementación el uso de almacenamiento, son cargadas en una memoria de solo lectura, organizada en un vector bidimensional de 16 datos, con longitud de 8 bits para cada uno de ellos. El diagrama de la Figura 3.1 ilustra su arquitectura.

Algoritmo 3.5 Cálculo de las constantes de ronda de AES128.

Ensure: $Rcon_0 \in \{0,1\}^8, ..., Rcon_{R-1} \in \{0,1\}^8$

1: $Rcon_0 \leftarrow 0x01$
2: **for** $i = 0$ to $R-2$ **do**
3: $\left(Rcon_i[0], ..., Rcon_i[7]\right) \overset{1}{\leftarrow} Rcon_i$
4: $Rcon_{i+1}[0] \leftarrow Rcon_i[7]$
5: $Rcon_{i+1}[1] \leftarrow Rcon_i[0]$ **or** $Rcon_i[7]$
6: $Rcon_{i+1}[2] \leftarrow Rcon_i[1]$
7: $Rcon_{i+1}[3] \leftarrow Rcon_i[2]$ **or** $Rcon_i[7]$
8: $Rcon_{i+1}[4] \leftarrow Rcon_i[3]$ **or** $Rcon_i[7]$
9: $Rcon_{i+1}[5] \leftarrow Rcon_i[4]$
10: $Rcon_{i+1}[6] \leftarrow Rcon_i[5]$
11: $Rcon_{i+1}[7] \leftarrow Rcon_i[6]$
12: $Rcon_{i+1} \leftarrow Rcon_{i+1}[0]|, ..., |Rcon_{i+1}[7]$
13: **end for**

Por otra parte, si el cálculo de las constantes de ronda se realiza al vuelo, se implementa un circuito combinacional que calcula un doblado dentro del campo $GF(2^8)$, como lo explica el Algoritmo 3.5 y lo ilustra el diagrama de la Figura 3.2.

Componente KMEM. El componente $KMEM$ almacena la llave inicial K y las llaves de ronda RK_i.

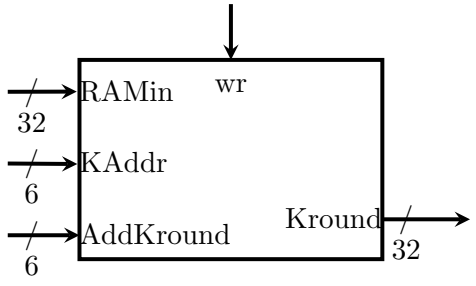

Figura 3.3: Interfaz del componente KMEM de AES128.

Diseñado con registros de 32 bits con un puerto de salida: $Kround$, y dos puertos $Kaddr$ y $AddKrond$ para direccionar escritura y lectura respectivamente. Se han empleado recursos de memoria RAM del FPGA para el almacenamiento. El diagrama en la Figura 3.3 ilustra su interfaz.

El procedimiento para el cálculo de las llaves de ronda, $RK_0, ..., RK_{R-2}$, a partir de la llave inicial K y del número de ronda $i \in \{0, ..., R-2\}$ en el que se encuentra la ejecución, se muestra en el Algoritmo 3.6, para el cual, se implementan los procedimientos $Rotar$, $SubCell$ y $Expandir$.

Algoritmo 3.6 Cálculo de llaves de ronda de AES128.

Require: $i \in \{0, ..., R-2\}, RK_i \in \{0,1\}^k$
Ensure: $RK_{i+1} \in \{0,1\}^k$
1: $RK_{i+1} \leftarrow Rotar(RK_i)$
2: $RK_{i+1} \leftarrow SubCell(RK_i)$
3: $RK_{i+1} \leftarrow RK_{i+1} \oplus Expandir(Rcon(i))$
4: $RK_{i+1} \leftarrow RK_{i+1} \oplus RK_i$

El procedimiento $Rotar(RK_i)$ consiste en aplicar un desplazamiento de 8 bits hacia la la izquierda a la llave inicial RK_i. Para su implementación es diseñado un registro de corrimiento, el cual es reutilizado en el procedimiento $ShiftRows$.

El procedimiento $SubCell(RK_i)$ consiste en aplicar una sustitución de los elementos de la llave con los elementos de una denominada Caja-S, tal procedimiento se explica a detalle más adelante.

El procedimiento $Expandir(Rcon(i))$ consiste en concatenar $k-8$ ceros como la parte menos significativa en $Rcon(i)$, como lo explica el Algoritmo 3.7. Para su implementación, se unen las señales en $Rcon$ a señales con valor lógico de 0.

Algoritmo 3.7 Expansión de la constante de ronda $Rcon$.

Require: $RconIn \in \{0,1\}^8$
Ensure: $RconOUT \in \{0,1\}^k$
1: $RconOut \leftarrow RconIn|0^{k-8}$

Componente KeyAdd. El procedimiento $KeyAdd$ consiste en sumas binarias dentro de $GF(2^8)$, del estado S y la llave K si se trata de la primera ronda, o de la llave RK_i para la ronda i.

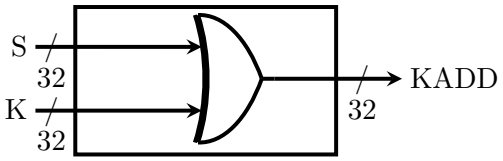

Figura 3.4: Componente KeyAdd de AES128.

Es el componente más pequeño, consiste en 32 compuertas XOR con entradas: la llave de ronda RK_i y el estado S.

Componente SubCell. El procedimiento $SubCell$ consiste en una sustitución de los elementos del estado S con los elementos de una Caja-S, los cuales resultan de calcular un inverso multiplicativo en el campo $GF(2^8)$ y una transformación afín del elemento a sustituir, el componente se ilustra en la Figura 3.5.

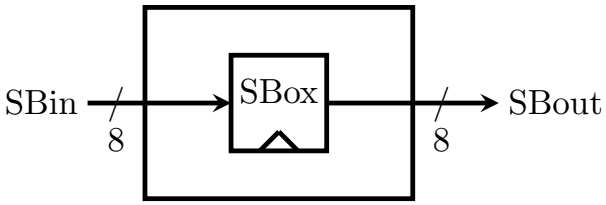

Figura 3.5: Componente SubCell con almacenamiento de AES128.

Para su implementación son considerados dos enfoques, el primero basado en almacenamiento del cómputo previo de los elementos de la Caja-S como un arreglo de 256 posiciones con longitud 8 bits cada uno, donde la sustitución se realiza accediendo al byte que se encuentra en la posición indicada por el elemento del estado a sustituir, para el cual se implementa un circuito secuencial con un registro que almacena en sus localidades de memoria los elementos de la Caja-S mostrado en la Figura 3.5, su descripción con VHDL se muestra en el Código 3.3.

Código 3.3: $Caja - S$ de AES128 almacenada en ROM

```
1  SIGNAL ROM : MEM := (
2          X"637c777bf26b6fc53001672bfed7ab76",
3          X"ca82c97dfa5947f0add4a2af9ca472c0",
4          X"b7fd9326363ff7cc34a5e5f171d83115",
5          X"04c723c31896059a071280e2eb27b275",
6          X"09832c1a1b6e5aa0523bd6b329e32f84",
7          X"53d100ed20fcb15b6acbbe394a4c58cf",
8          X"d0efaafb434d338545f9027f503c9fa8",
9          X"51a3408f929d38f5bcb6da2110fff3d2",
10         X"cd0c13ec5f974417c4a77e3d645d1973",
11         X"60814fdc222a908846eeb814de5e0bdb",
12         X"e0323a0a4906245cc2d3ac629195e479",
13         X"e7c8376d8dd54ea96c56f4ea657aae08",
14         X"ba78252e1ca6b4c6e8dd741f4bbd8b8a",
15         X"703eb5664803f60e613557b986c11d9e",
16         X"e1f8981169d98e949b1e87e9ce5528df",
17         X"8ca1890dbfe6426841992d0fb054bb16"
18         );
```

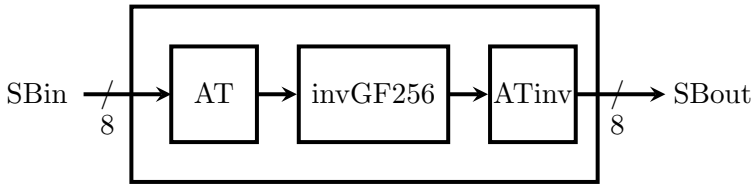

Figura 3.6: Componente SubCell secuencial de AES128.

Por otra parte, el segundo enfoque consiste en realizar el cálculo al vuelo de los elementos de la Caja-S definido por la Ecuación 3.1, para ello se implementa un componente combinacional ilustrado en la Figura 3.6 que utiliza una instancia del componente **invGF256** con sus respectivos módulos, además de los módulos para la transformación afín de la entrada y la salida.

$$subCell(S_i) = \begin{cases} AT(S_i^{-1}) & \text{si } S_i > 0 \\ 0x63 & \text{en otro caso} \end{cases} \tag{3.1}$$

Componente invGF256. Cada elemento en $GF(2^8)$ puede verse como un polinomio $bx + c$ de grado a lo más 7 con coeficientes en $\{0, 1\}^4$, donde b

es el nibble[1] mas significativo y c el nibble menos significativo, y para el cual su inverso multiplicativo puede calcularse como se define [34] en la Ecuación 3.2.

$$(bx + c)^{-1} = b(b^2B + bcA + c^2)^{-1}x + (c + bA)(b^2B + bcA + c^2)^{-1} \quad (3.2)$$

Ha sido elegido a $x^2 + x + \lambda$ como el polinomio irreducible, donde puede observarse que $A = 1$ y $B = \lambda$ [35]. Por lo que sustituyendo en la Ecuación 3.2 se tiene la Ecuación 3.3.

$$(bx + c)^{-1} = b(b^2\lambda + c(b + c))^{-1}x + (c + b)(b^2\lambda + c(b + c))^{-1} \quad (3.3)$$

Para la implementación de la Ecuación 3.3, es diseñado un componente para calcular un mapeo isomorfo para cada elemento S_i del estado S a través de la función δ (y δ^{-1}) para iniciar (y finalizar) con el cálculo del inverso multiplicativo en $GF(2^8)$, definidas en las Ecuaciones 3.4 y 3.5 respectivamente.

$$\delta(S_i) = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} S_i[7] \\ S_i[6] \\ S_i[5] \\ S_i[4] \\ S_i[3] \\ S_i[2] \\ S_i[1] \\ S_i[0] \end{bmatrix} \quad (3.4)$$

$$\delta(S_i)^{-1} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} S_i[7] \\ S_i[6] \\ S_i[5] \\ S_i[4] \\ S_i[3] \\ S_i[2] \\ S_i[1] \\ S_i[0] \end{bmatrix} \quad (3.5)$$

[1]También llamado Cuado o Cuarteto es el conjunto de cuatro bits que equivalen a medio octeto (o medio byte).

Resolviendo para 3.4 y 3.5, se tienen las Ecuaciones 3.6 y 3.7

$$\delta(S_i) = \begin{bmatrix} S_i[7] \oplus S_i[1] \\ S_i[7] \oplus S_i[6] \oplus S_i[4] \oplus S_i[3] \oplus S_i[2] \oplus S_i[1] \\ S_i[7] \oplus S_i[5] \oplus S_i[3] \oplus S_i[2] \\ S_i[7] \oplus S_i[5] \oplus S_i[3] \oplus S_i[2] \oplus S_i[1] \\ S_i[7] \oplus S_i[5] \oplus S_i[2] \oplus S_i[1] \\ S_i[7] \oplus S_i[4] \oplus S_i[3] \oplus S_i[2] \oplus S_i[1] \\ S_i[6] \oplus S_i[4] \oplus S_i[1] \\ S_i[6] \oplus S_i[1] \oplus S_i[0] \end{bmatrix} \tag{3.6}$$

$$\delta(S_i)^{-1} = \begin{bmatrix} S_i[7] \oplus S_i[6] \oplus S_i[5] \oplus S_i[1] \\ S_i[6] \oplus S_i[2] \\ S_i[6] \oplus S_i[5] \oplus S_i[1] \\ S_i[6] \oplus S_i[5] \oplus S_i[4] \oplus S_i[2] \oplus S_i[1] \\ S_i[5] \oplus S_i[4] \oplus S_i[3] \oplus S_i[2] \oplus S_i[1] \\ S_i[7] \oplus S_i[4] \oplus S_i[3] \oplus S_i[2] \oplus S_i[1] \\ S_i[5] \oplus S_i[4] \\ S_i[6] \oplus S_i[5] \oplus S_i[4] \oplus S_i[2] \oplus S_i[0] \end{bmatrix} \tag{3.7}$$

Suma de dos elementos. La suma de dos elementos en $GF(2^4)$ es implementada con compuertas XOR.

Multiplicación de dos elementos. El resultado de la multiplicación de dos elementos en el campo $GF(2^4)$ se muestra en la Tabla 3.2.

	0x0	**0x1**	**0x2**	**0x3**	**0x4**	**0x5**	**0x6**	**0x7**	**0x8**	**0x9**	**0xa**	**0xb**	**0xc**	**0xd**	**0xe**	**0xf**
0x0	**0x0**	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0
0x1	0x0	**0x1**	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x2	0x0	0x2	**0x3**	0x1	0x8	0xa	0xb	0x9	0xc	0xe	0xf	0xd	0x4	0x6	0x7	0x5
0x3	0x0	0x3	0x1	**0x2**	0xc	0xf	0xd	0xe	0x4	0x7	0x5	0x6	0x8	0xb	0x9	0xa
0x4	0x0	0x4	0x8	0xc	**0x6**	0x2	0xe	0xa	0xb	0xf	0x3	0x7	0xd	0x9	0x5	0x1
0x5	0x0	0x5	0xa	0xf	0x2	**0x7**	0x8	0xd	0x3	0x8	0x9	0xc	0x1	0x4	0xb	0xe
0x6	0x0	0x6	0xb	0xd	0xe	0x8	**0x5**	0x3	0x7	0x1	0xc	0xa	0x9	0xf	0x2	0x4
0x7	0x0	0x7	0x9	0xe	0xa	0xd	0x3	**0x4**	0xf	0x8	0x6	0x1	0x5	0x2	0xc	0xb
0x8	0x0	0x8	0xc	0x4	0xb	0x3	0x7	0xf	**0xd**	0x5	0x1	0x9	0x6	0xe	0xa	0x2
0x9	0x0	0x9	0xe	0x7	0xf	0x6	0x1	0x8	0x5	**0xc**	0xb	0x2	0xa	0x3	0x4	0xd
0xa	0x0	0xa	0xf	0x5	0x3	0x9	0xc	0x6	0x1	0xb	**0xe**	0x4	0x2	0x8	0xd	0x7
0xb	0x0	0xb	0xd	0x6	0x7	0xc	0xa	0x1	0x9	0x2	0x4	**0xf**	0xe	0x5	0x3	0x8
0xc	**0x0**	**0xc**	**0x4**	**0x8**	**0xd**	**0x1**	**0x9**	**0x5**	**0x6**	**0xa**	**0x2**	**0xe**	**0xb**	**0x7**	**0xf**	**0x3**
0xd	0x0	0xd	0x6	0xb	0x9	0x4	0xf	0x2	0xe	0x3	0x8	0x5	0x7	**0xa**	0x1	0xc
0xe	0x0	0xe	0x7	0x9	0x5	0xb	0x2	0xc	0xa	0x4	0xd	0x3	0xf	0x1	**0x8**	0x6
0xf	0x0	0xf	0x5	0xa	0x1	0xe	0x4	0xb	0x2	0xd	0x7	0x8	0x3	0xc	0x6	**0x9**

Tabla 3.2: Multiplicación de dos elementos en $GF(2^4)$.

Multiplicación de dos elementos. Para cualquier k y $\lambda = 0xc$, ambos elementos en $GF(2^4)$, a partir de la Tabla 3.2 se obtienen los valores de su producto, mostrados en la Tabla 3.3.

k	**0x0**	**0x1**	**0x2**	**0x3**	**0x4**	**0x5**	**0x6**	**0x7**	**0x8**	**0x9**	**0xa**	**0xb**	**0xc**	**0xd**	**0xe**	**0xf**
$k\lambda$	0x0	0xc	0x4	0x8	0xd	0x1	0x9	0x5	0x6	0xa	0x2	0xe	0xb	0x7	0xf	0x3

Tabla 3.3: Multiplicación de un elemento por $k\lambda$.

De la Tabla 3.3 (considerada como tabla de verdad) se obtienen las Ecuaciones 3.8, 3.9, 3.10, 3.11, con las que es implementado el componente para calcular el producto $k\lambda \in GF(2^4)$.

$$k\lambda_3 = k_2 \oplus k_0 \tag{3.8}$$

$$k\lambda_2 = k_3 \oplus k_2 \oplus k_1 \oplus k_0 \tag{3.9}$$

$$k\lambda_1 = k_3 \tag{3.10}$$

$$k\lambda_0 = k_2 \tag{3.11}$$

Cuadrado de un elemento. De la Tabla 3.2, el cuadrado de un elemento en $GF(2^4)$ se obtiene de la diagonal, los resultados se muestran en la Tabla 3.4.

k	**0x0**	**0x1**	**0x2**	**0x3**	**0x4**	**0x5**	**0x6**	**0x7**	**0x8**	**0x9**	**0xa**	**0xb**	**0xc**	**0xd**	**0xe**	**0xf**
k^2	0x0	0x1	0x3	0x2	0x6	0x7	0x5	0x4	0xd	0xc	0xe	0xf	0xb	0xa	0x8	0x9

Tabla 3.4: Cuadrado de un elemento $k \in GF(2^4)$.

De la Tabla 3.4 (considerada como tabla de verdad) se obtienen las Ecuaciones 3.12, 3.13, 3.14, 3.15, con las que es implementado el componente para calcular el cuadrado $k^2 \in GF(2^4)$.

$$k^2{}_3 = k_3 \tag{3.12}$$

$$k^2{}_2 = k_3 \oplus k_2 \tag{3.13}$$

$$k^2{}_1 = k_2 \oplus k_1 \tag{3.14}$$

$$k^2{}_0 = k_3 \oplus k_1 \oplus k_0 \tag{3.15}$$

Inverso multiplicativo de un elemento. De la Tabla 3.2, el inverso multiplicativo de un elemento en $GF(2^4)$ se obtiene buscando el elemento identidad, es decir el número $0x1$ en cada columna (o cada fila), donde el número de fila (o número de columna) corresponde al elemento en el campo y el número de columna (o número de fila) a su inverso multiplicativo (respectivamente, sin pérdida de generalidad). Los resultados se muestran en la Tabla 3.5.

k	**0x0**	**0x1**	**0x2**	**0x3**	**0x4**	**0x5**	**0x6**	**0x7**	**0x8**	**0x9**	**0xa**	**0xb**	**0xc**	**0xd**	**0xe**	**0xf**
k^{-1}	0x0	0x1	0x3	0x2	0xf	0xc	0x9	0xb	0xa	0x6	0x8	0x7	0x5	0xe	0xd	0x4

Tabla 3.5: Inverso multiplicativo de un elemento $k \in GF(2^4)$.

De la Tabla 3.5 (considerada como tabla de verdad) se obtienen las Ecuaciones 3.16, 3.17, 3.18, 3.19, con las que es implementado el componente para calcular el inverso multiplicativo $k^{-1} \in GF(2^4)$.

$$k^{-1}{}_3 = k_3 \oplus k_3k_2k_1 \oplus k_3k_0 \oplus k_2 \tag{3.16}$$

$$k^{-1}{}_2 = k_3k_2k_1 \oplus k_3k_2k_0 \oplus k_3k_0 \oplus k_2 \oplus k_2k_1 \tag{3.17}$$

$$k^{-1}{}_1 = k_3 \oplus k_3k_2k_1 \oplus k_2 \oplus k_2k_0 \oplus k_1 \tag{3.18}$$

$$k^{-1}{}_0 = k_3k_2k_1 \oplus k_3k_2k_0 \oplus k_3k_1 \oplus k_3k_1k_0 \oplus k_3k_0 \oplus k_2 \oplus k_2k_1 \oplus k_2k_1k_0 \oplus k_1k_0 \tag{3.19}$$

Componente MixColumn. Para la implementación del componente **MixColumn** se considera el producto de la Ecuación 3.20, donde el estado S es representado como un arreglo de cuatro elementos (S_0, S_1, S_2, S_3) y todas las operaciones se realizan en $GF(2^8)$.

$$MixColumn(S) = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} \tag{3.20}$$

De la Ecuación 3.20 se obtienen las Ecuaciones 3.21, 3.22, 3.23 y 3.24.

$$MixColumn(S)_3 = 2 \cdot S_0 \oplus 3 \cdot S_1 \oplus 1 \cdot S_2 \oplus 1 \cdot S_3 \tag{3.21}$$

$$MixColumn(S)_2 = 2 \cdot S_1 \oplus 3 \cdot S_2 \oplus 1 \cdot S_3 \oplus 1 \cdot S_0 \tag{3.22}$$

$$MixColumn(S)_1 = 2 \cdot S_2 \oplus 3 \cdot S_3 \oplus 1 \cdot S_0 \oplus 1 \cdot S_1 \tag{3.23}$$

$$MixColumn(S)_0 = 2 \cdot S_3 \oplus 3 \cdot S_0 \oplus 1 \cdot S_1 \oplus 1 \cdot S_2 \tag{3.24}$$

Multiplicaciones en MixColumn. El doblado de un elemento en $GF(2^8)$ es implementado con el módulo **xTimes** ilustrado en la Figura 3.7. El triple de un elemento en $GF(2^8)$ se calcula como $k+2k$ para cualquier $k \in GF(2^8)$.

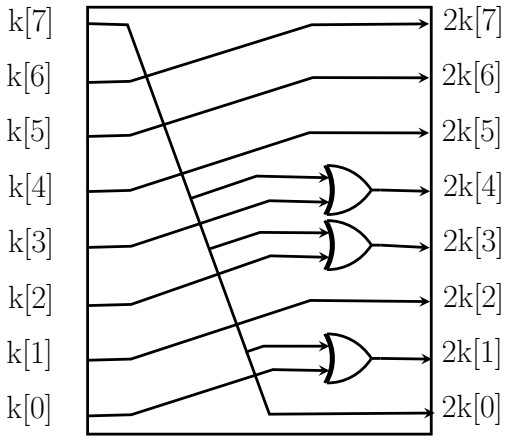

Figura 3.7: Módulo **xTimes**.

La implementación del componente **MixColumn** es realizada con un circuito combinacional mostrado en la Figura 3.8 que utiliza instancias del componente **xTimes** para realizar el cálculo de las Ecuaciones 3.21, 3.22, 3.23 y 3.24. Las entradas A, B, C y D del componente toman valores del estado S de la siguiente manera:

- Para calcular $MixColumn(S_3)$: A= S_0, B= S_1, C= S_2 y D= S_3.
- Para calcular $MixColumn(S_2)$: A= S_1, B= S_2, C= S_3 y D= S_0.
- Para calcular $MixColumn(S_1)$: A= S_2, B= S_3, C= S_0 y D= S_1.
- Para calcular $MixColumn(S_0)$: A= S_3, B= S_0, C= S_1 y D= S_2.

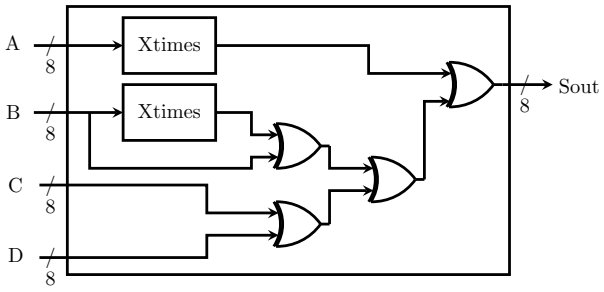

Figura 3.8: Componente SubCell combinacional de AES128.

Componente ShiftRows. Si el estado S es visto como una matriz cuadrada de elementos, el procedimiento $ShiftRows$ consiste en desplazamientos aplicados a cada fila del estado. Su implementación se realiza con registros de desplazamiento organizados y controlados con la unidad de control (explicada más adelante), los cuales almacenan el estado S para cada ronda. La Figura 3.9 ilustra la implementación del componente.

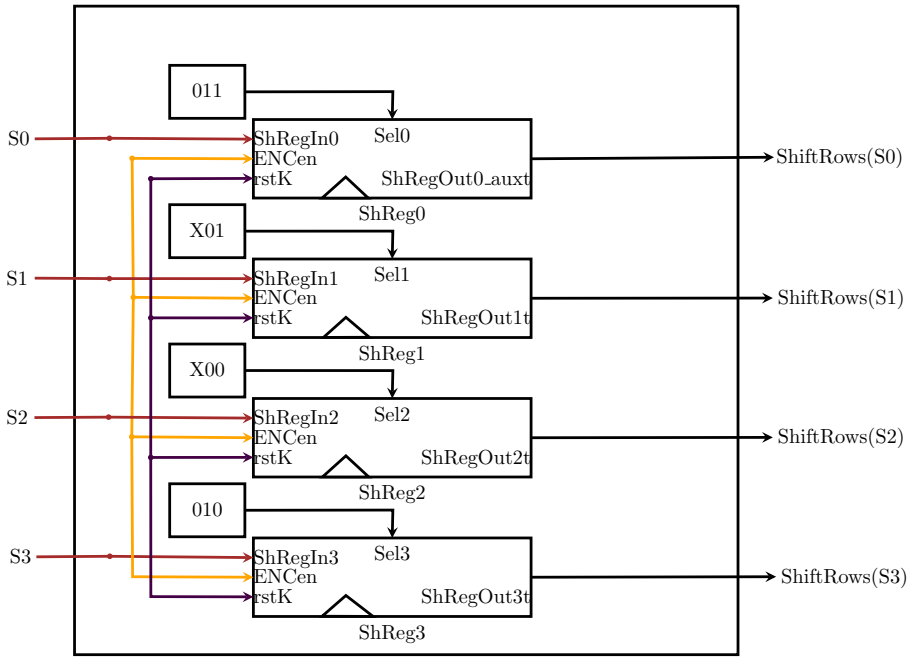

Figura 3.9: Componente ShiftRows de AES128.

Unidad de control. Para la implementación de la unidad de control se considera la máquina de estados mostrada en la Figura 3.10.

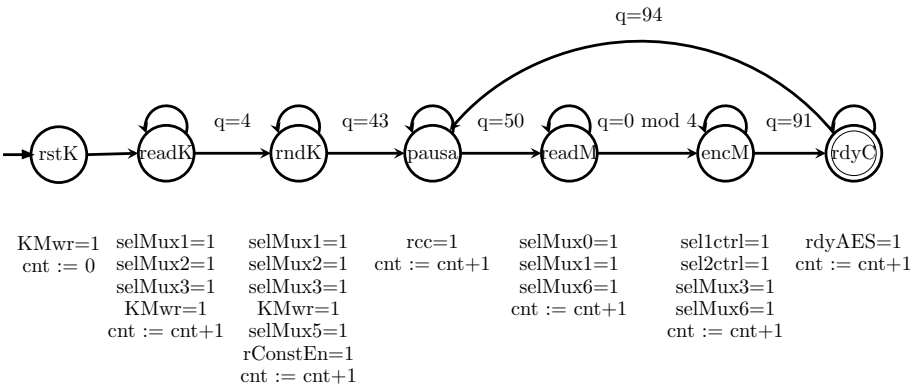

Figura 3.10: Máquina de estados de la unidad de control de AES128.

Para su implementación es diseñado un circuito secuencial con siete estados definidos como:

- *rsK*: El estado inicial. Todos los registros son reinicializados, se prepara a los componentes de almacenamiento para comenzar la carga de los datos, los contadores son inicializados en cero y el control pasa de manera incondicional al estado *readK*.

- *readK*: La llave K es leída con hardware de 32 bits y almacenada con el componente $KMEM$. El proceso de almacenamiento de la llave tarda cuatro ciclos de reloj, cuando ocurre que el contador $q = 4$, el control pasa al estado *rndK*.

- *rndK*: En este estado, se ejecuta la derivación de las llaves de ronda K_i, para ello se controlan las señales que habilitan la escritura en los registros de desplazamiento, además se seleccionan las entradas adecuadas en los multiplexores correspondientes, la escritura en la memoria del componente $KMEM$ permanece activa. Cuando ocurre que el contador $q = 44$ el proceso de derivación ha terminado y por lo tanto el control pasa al estado *pausa*.

- *pausa*: Una vez concluido el proceso de derivación de llaves de ronda, los registros de desplazamiento necesitan ser reinicializados, los multiplexores que alimentan los registros de desplazamiento deben conmu-

tar sus entradas para trabajar ahora con el estado S, se preparan los componentes necesarios para el proceso de cifrado. Cuando ocurre que el contador $q = 50$ el control pasa al estado *readM*.

- *readM*: Se realiza la lectura de los bloques del estado S con hardware de 32 bits. El almacenamiento se realiza en los registros de desplazamiento, los cuales ejecutan además el procedimiento $shiftRows$ durante el cifrado. La lectura del estado toma cuatro ciclos de reloj, cuando el contador $q = 54$ el control pasa al estado *encM*.

- *encM*: Son ejecutados en cada ciclo de reloj los procedimientos $keyAdd$, $subCell$, $shiftRow$ y $mixColumn$ para cada ronda con hardware de 32 bits. El resultado de la cifra del estado S en cada ronda es dado como entrada para la siguiente ronda. El proceso de cifrado ocupa 36 ciclos de reloj sin considerar la última ronda, cuando ocurre que el contador $q = 90$ el control pasa al estado *rdyC*.

- *rdyC*: En este estado se ejecuta la última ronda, la cual omite la aplicación del procedimiento $mixColumn$ al estado S, además se enciende la bandera $rdyC$ que indica que la salida en ese instante corresponde a la cifra del bloque. El diseño del cifrador contempla el cifrado consecutivo de un número indeterminado de bloques, lo cual es aplicado en la implementación de los modos de operación (como lo es COFB), por lo tanto cuando el contador $q = 94$ se reinicializa como $q = 44$, el control regresa al estado de *pausa*, luego inicia el proceso de lectura de un nuevo estado S para ser cifrado con la llave K proporcionada desde el principio de la ejecución.

En la implementación con VHDL se tienen en consideración aspectos propios del lenguaje, por ejemplo para enumerar los procesos llevados a cabo en cada estado de la máquina se utiliza una estructura de casos, la cual evalúa el valor de una señal. Además, aspectos que permiten optimizar el espacio ocupado tales como utilizar un contador con valor máximo de 44, en lugar de un valor de 94 (como lo ilustra la máquina de estados propuesta en la Figura 3.10), ayudan a evitar el uso excesivo de recursos.

Las señales que son controladas corresponden a las del diseño de la implementación en hardware ilustrado en la Figura 3.11.

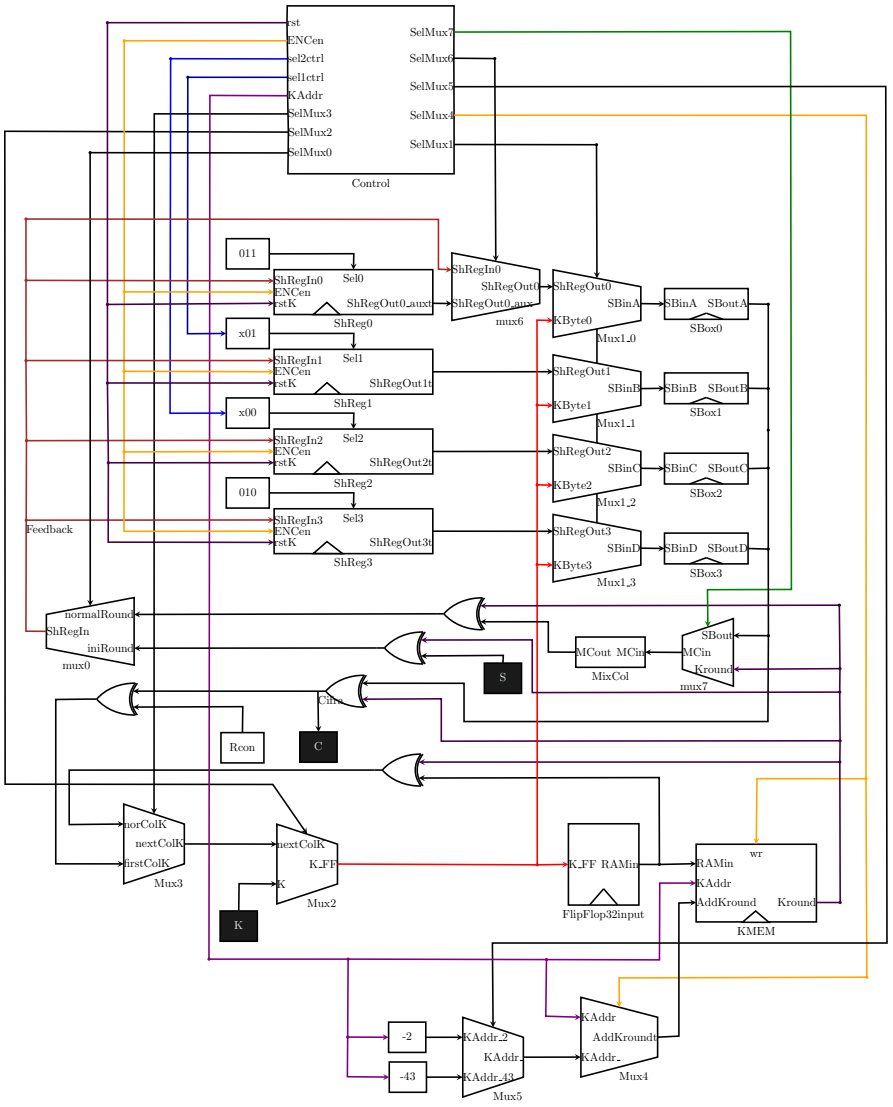

Figura 3.11: Arquitectura de AES128.

La implementación tarda 94 ciclos de reloj en procesar un bloque. En cada ciclo se procesa el mensaje serializado, es decir en porciones de longitud 32 bits, lo que permite el uso de componentes más pequeños, sin embargo al reducir el tamaño de la implementación aumenta el número de ciclos que tarda el proceso de cifrado.

Las tablas 4.3 y 4.4 del Capítulo 4 (Resultados) muestran los resultados obtenidos de la implementación utilizando el método combinacional y con almacenamiento respectivamente. En ellas se ilustra también el área ocupada en términos de LUTs, FlipFlops y bloques de RAM por sus respectivos componentes.

El tamaño de la implementación de AES128 es de 376 LUTs y 1.5 bloques de RAM para la estrategia que utiliza almacenamiento.

El código fuente de la implementación está disponible en el enlace del apéndice B.2 correspondiente.

3.2.2. Midori64

En el Algoritmo 3.2 se realiza el cómputo previo de la llave de trabajo WK y de las llaves de ronda $RK_0, ..., RK_{R-2}$, lo que implica el uso de almacenamiento. El Algoritmo 3.2 es reorganizado considerando una implementación ligera en hardware.

Para evitar el uso de almacenamiento, no se utiliza cómputo previo y en su lugar se realiza el cálculo de las llaves de ronda al vuelo, como lo ilustra el Algoritmo 3.8.

Algoritmo 3.8 Midori64 sin cómputo previo.

Require: $M \in \{0,1\}^n, K \in \{0,1\}^k$
Ensure: $C \in \{0,1\}^n$
1: $\left(K\left[0\right], K\left[1\right]\right) \xleftarrow{n/2} K$
2: $WK \leftarrow K\left[0\right] \oplus K\left[1\right]$
3: $S \leftarrow KeyAdd(M, WK)$
4: **for** $i = 0$ to $R - 2$ **do**
5: $\quad S \leftarrow SubCell(S)$
6: $\quad S \leftarrow ShuffleCell(S)$
7: $\quad S \leftarrow MixColumn(S)$
8: $\quad RK_i \leftarrow Expandir(\beta_i) \oplus K\left[i \mod 2\right]$
9: $\quad S \leftarrow KeyAdd(S, RK_i)$
10: **end for**
11: $S \leftarrow SubCell(S)$
12: $C \leftarrow KeyAdd(S, WK)$

El algoritmo para el cálculo de las llaves involucra al conjunto de constantes de ronda β definido en la Tabla 3.6.

i	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
β_i	0x15b3	0x78c0	0xa435	0x6213	0x104f	0xd170	0x0266	0x0bcc
i	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
β_i	0x9481	0x40b8	0x7197	0x228e	0x5130	0xf8ca	0xdf90	0x7c81

Tabla 3.6: Constantes de ronda en Midori64

Las constantes β_i se derivan de los dígitos hexadecimales de la parte fraccionaria del número π (3.243f6a8885a3...), y debido a la complejidad de su cálculo, son almacenadas previamente en una memoria de solo lectura, organizada en un vector bidimensional de 64 datos, con longitud de 4 bits para cada uno de ellos.

Componente KMEM. El componente $KMEM$ almacena la llave inicial K y calcula la llave de trabajo WK. Diseñado como un registro de 128 bits con dos puertos de salida: $KMout$ y $WKMout$ (para la llave K y la llave de trabajo WK respectivamente). El diseño se ilustra con la Figura 3.12.

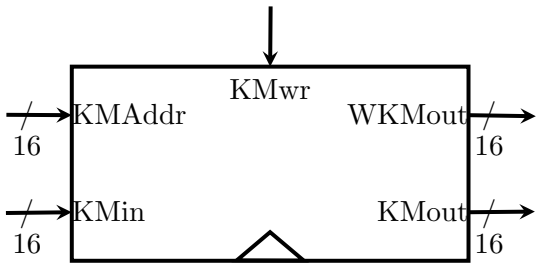

Figura 3.12: Interfaz del componente KMEM de Midori64.

El procedimiento para el cálculo de la llave de trabajo WK y de las llaves de ronda, $RK_0, ..., RK_{R-2}$ a partir de la llave inicial K y del número de ronda $i \in \{0, ..., R-2\}$ en el que se encuentra la ejecución, se muestra en el Algoritmo 3.9.

Algoritmo 3.9 Cálculo de llaves de ronda de Midori64.

Require: $i \in \{0, ..., R-2\}, K \in \{0,1\}^k$
Ensure: $WK \in \{0,1\}^{k/2}, RK_i \in \{0,1\}^{k/2}$
1: $\left(K\left[0\right], K\left[1\right]\right) \overset{n}{\leftarrow} K$
2: $RK_i \leftarrow Expandir(\beta_i) \oplus K\left[i \mod 2\right]$
3: $WK \leftarrow K\left[0\right] \oplus K\left[1\right]$

Donde el procedimiento $Expandir(\beta_i)$ consiste en concatenar tres ceros como la parte más significativa de cada bit en β_i, como lo explica el Algoritmo 3.10. La implementación de la expansión de las constantes β_i no es eficiente en software, ya que el acceso a todos los bits de manera individual de cualquier dato, es siempre complejo, sin embargo en hardware la complejidad se reduce de manera importante, ya que el acceso a las señales que representan a los bits es directo.

Algoritmo 3.10 Expansión de β_i.

Require: $\beta_i \in \{0,1\}^{n/4}$
Ensure: $\beta_i \in \{0,1\}^n$
1: $\left(\beta_i\left[0\right], ..., \beta_i\left[n/4\right]\right) \overset{1}{\leftarrow} \beta_i$
2: $\beta_i \leftarrow \left(0^3||\beta_i\left[0\right], ..., 0^3||\beta_i\left[n/4\right]\right)$

Componente KeyGen. El componente $KeyGen$ calcula al vuelo la llave de ronda RK_i para cada ronda en la rutina de cifrado. El almacenamiento de las constantes β_i se definen en una señal $BETA$ como lo ilustra el Código 3.4 y su expansión se realiza de manera directa manipulando las señales de cada bit. El diseño del componente se muestra en la Figura 3.13.

Código 3.4: $KeyGen$ de Midori64.

```
1  signal BETA : Bs := (
2          "0000", "0000", "0000", "0000", "0001", "0101", "1011", "0011",
3          "0111", "1000", "1100", "0000", "1010", "0100", "0011", "0101",
4          "0110", "0010", "0001", "0011", "0001", "0000", "0100", "1111",
5          "1101", "0001", "0111", "0000", "0000", "0010", "0110", "0110",
6          "0000", "1011", "1100", "1100", "1001", "0100", "1000", "0001",
7          "0100", "0000", "1011", "1000", "0111", "0001", "1001", "0111",
8          "0010", "0010", "1000", "1110", "0101", "0001", "0011", "0000",
9          "1111", "1000", "1100", "1010", "1101", "1111", "1001", "0000"
10         );
```

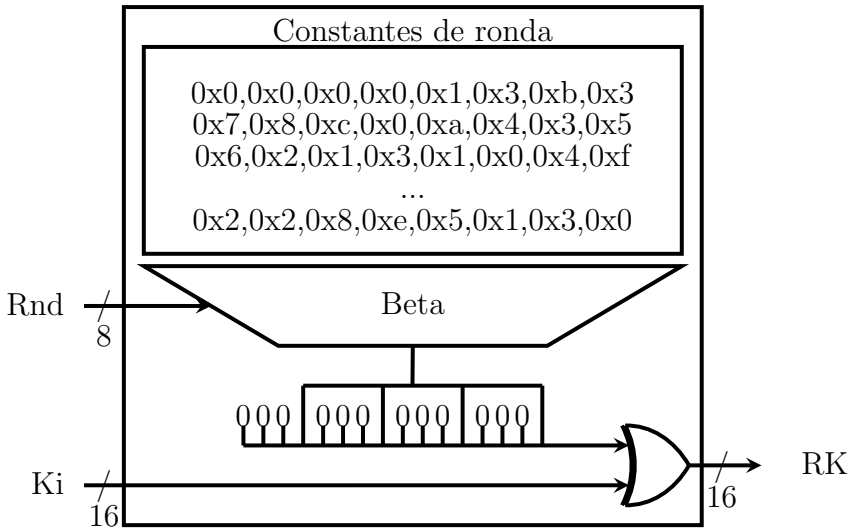

Figura 3.13: Componente KeyGen de Midori64.

Componente KeyAdd. El procedimiento *KeyAdd* consiste en la suma binaria en GF(2^m), con $m = 4$, del estado S y la llave WK si se trata de la primera o última ronda, o de la llave RK_i para la ronda i.

Es el componente más pequeño, consiste en 16 compuertas XOR con entradas: la llave de ronda RK_i y el estado S, su diseño lo ilustra la Figura 3.14.

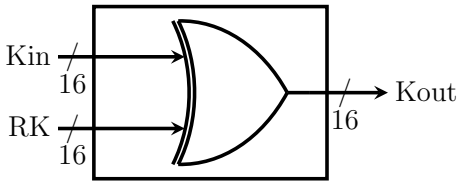

Figura 3.14: Componente KeyAdd de Midori64.

Componente SubCell. El procedimiento *SubCell* consiste en una sustitución de los elementos del estado S por los elementos de una Caja-S con valores definidos en la Tabla 3.7. Para su implementación son considerados dos enfoques, el primero orientado en el almacenamiento de los valores de la Caja-S como un arreglo de 16 elementos con longitud 4 bits cada uno (el Código 3.5 lo ejemplifica), donde la sustitución se realiza accediendo al

nibble que se encuentra en la posición indicada por el elemento del estado a sustituir. Dicho enfoque lo ilustra el diagrama de la Figura 3.15.

k	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
SBox(k)	c	a	d	3	e	b	f	7	8	9	1	5	0	2	4	6

Tabla 3.7: Caja-S de Midori64

Código 3.5: SubCell con Caja-S de Midori64.

```
1 signal Sb0 : Bs := 0x"cad3ebf789150246";
```

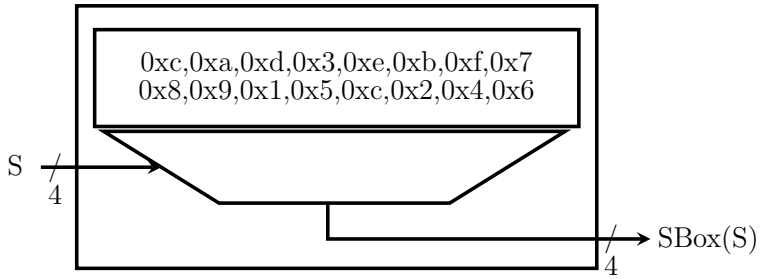

Figura 3.15: Módulo $CajaS$ secuencial de Midori64.

Para el segundo enfoque, el cómputo al vuelo de los elementos de la Caja-S no está definido en la publicación [8] donde ha sido propuesto, sin embargo se implementa como un componente combinacional para la sustitución de cada elemento del estado S, evitando el uso de almacenamiento y de un multiplexor.

De la Tabla 3.7 (considerada como tabla de verdad) se obtienen las Ecuaciones 3.25, 3.26, 3.27 y3.28.

$$SubCell(S_3) = \overline{S_3} \cdot \overline{S_1} \oplus \overline{S_2} \cdot \overline{S_1} \oplus \overline{S_3} \cdot \overline{S_0} \tag{3.25}$$

$$SubCell(S_2) = \overline{S_3} \cdot \overline{S_0} \oplus S_2 \cdot S_1 \oplus S_3 \cdot S_1 \cdot S_0 \tag{3.26}$$

$$SubCell(S_1) = \overline{S_3} \cdot S_0 \oplus \overline{S_3} \cdot S_2 \oplus S_2 \cdot S_0 \tag{3.27}$$

$$SubCell(S_0) = \overline{S_3} \cdot S_1 \oplus \overline{S_2} \cdot S_1 \oplus \overline{S_3} \cdot S_2 \cdot S_0 \oplus S_3 \cdot \overline{S_2} \cdot S_0 \tag{3.28}$$

De las Ecuaciones 3.25, 3.26, 3.27 y 3.28 se obtiene el diseño mostrado en la Figura 3.16.

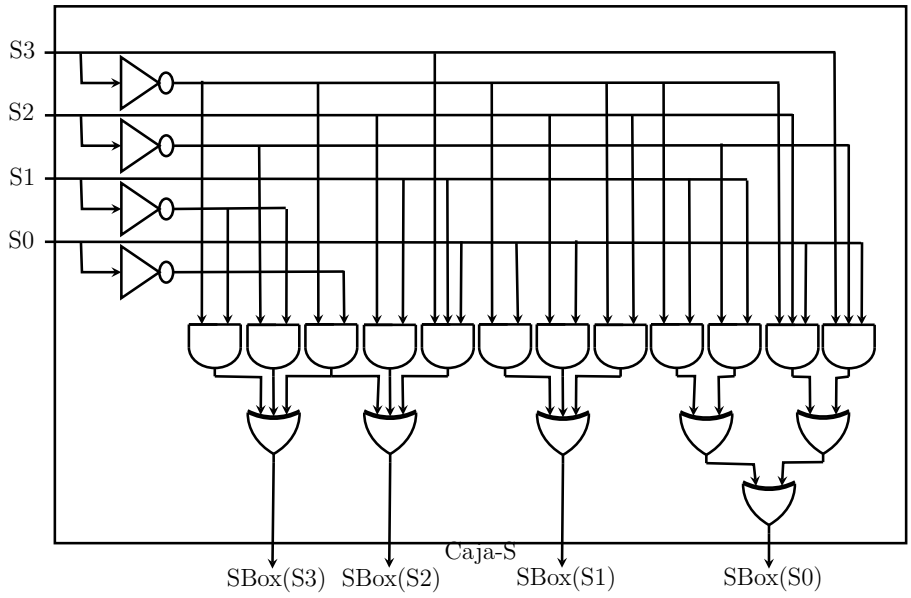

Figura 3.16: Módulo $CajaS$ combinacional de Midori64.

El componente $SubCell$ procesa cuatro datos en una ronda utilizando una instancia del módulo $CajaS$ como lo muestra la Figura 3.17. En cada ciclo de reloj procesa un elemento del estado S, la entrada $SCaddr$ selecciona uno de los cuatro elementos a sustituir.

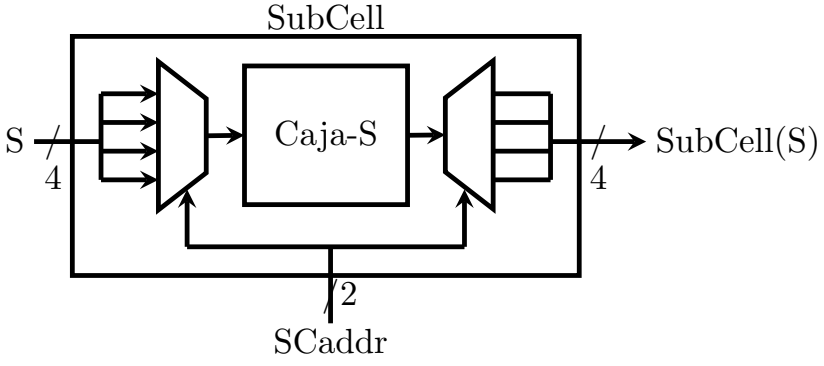

Figura 3.17: Componente $SubCell$ de Midori64.

Componente ShuffleCell. El procedimiento $ShuffleCell$ es una permutación entre los elementos del estado S, definida como:

$$(S_0, ..., S_{15}) \leftarrow (S_0, S_7, S_{14}, S_9, S_5, S_2, S_{11}, S_{12}, S_{15}, S_8, S_1, S_6, S_{10}, S_{13}, S_4, S_3) \tag{3.29}$$

Para la rutina de descifrado, el único elemento que no es involutivo es precisamente la permutación en $ShuffleCell$; para descifrar un mensaje, es necesario implementar la permutación inversa. Sin embargo, para el presente trabajo no es necesario implementar tal inverso, a pesar de ser una paso sin costo significativo. El diagrama de su diseño se muestra en la Figura 3.18.

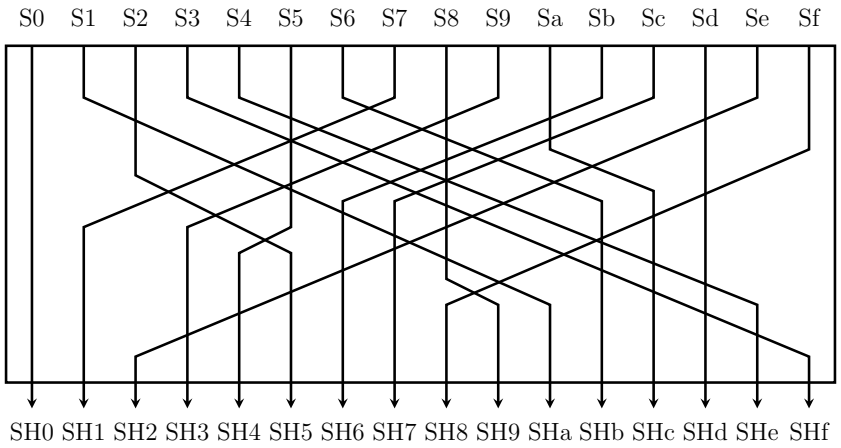

Figura 3.18: Componente ShuffleCell de Midori64.

Componente SMEM. Es el componente donde se almacena el estado S. Utiliza como elemento de memoria a un registro síncrono de 64 bits. La señal $SMwr$ habilita (o deshabilita) la escritura en la dirección del registro MEM indicada por el bus $SMaddr$ de longitud 4 bits. El bus de salida $SMout$ devuelve el dato almacenado en la dirección apuntada por $SMaddr$ cuando $SMwr = 0$, o el dato de entrada en $SMin$ de otro modo. El bus $SMsh$ devuelve la permutación del estado S almacenado en el registro MEM correspondiente al procedimiento $ShuffleCell$, la cual es calculada en el componente **ShuffleCell**. Su diseño lo ilustra la Figura 3.19.

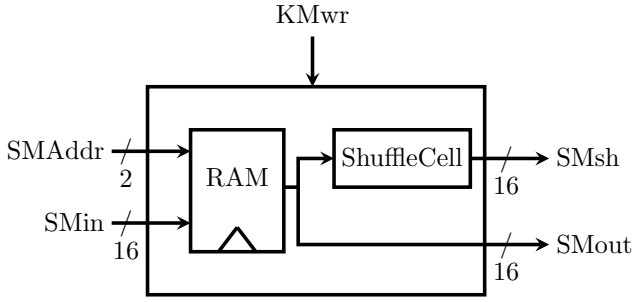

Figura 3.19: Módulo $SMEM$ de Midori64.

Componente MixColumn El procedimiento $MixColumn$ está definido en la Ecuación 3.30.

$$S' = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} S_0 & S_4 & S_8 & S_{12} \\ S_1 & S_5 & S_9 & S_{13} \\ S_2 & S_6 & S_{10} & S_{14} \\ S_3 & S_7 & S_{11} & S_{15} \end{bmatrix} \tag{3.30}$$

Donde todas las operaciones se realizan dentro del campo GF(2^4), se obtiene:

$$S' = \begin{bmatrix} S_1 \oplus S_2 \oplus S_3 & S_5 \oplus S_6 \oplus S_7 & S_9 \oplus S_{10} \oplus S_{11} & S_{13} \oplus S_{14} \oplus S_{15} \\ S_0 \oplus S_2 \oplus S_3 & S_4 \oplus S_6 \oplus S_7 & S_8 \oplus S_{10} \oplus S_{11} & S_{12} \oplus S_{14} \oplus S_{15} \\ S_0 \oplus S_1 \oplus S_3 & S_4 \oplus S_5 \oplus S_7 & S_8 \oplus S_9 \oplus S_{11} & S_{12} \oplus S_{13} \oplus S_{15} \\ S_0 \oplus S_1 \oplus S_2 & S_4 \oplus S_5 \oplus S_6 & S_8 \oplus S_9 \oplus S_{10} & S_{12} \oplus S_{13} \oplus S_{14} \end{bmatrix} \tag{3.31}$$

De la Ecuación 3.31, se implementa el procedimiento $MixColumn$ en este componente con siete sumas binarias, como lo muestra el Código 3.6 y lo ilustra el diagrama de la Figura 3.20.

Código 3.6: Cálculo de MixColumn.

```
1 MC3 <= MCin(15 downto 12) xor MCin(11 downto 8) xor MCin(7 downto 4) xor
      MCin(3 downto 0);
2 MCout(15 downto 12) <= MC3 xor MCin(15 downto 12);
3 MCout(11 downto 8) <= MC3 xor MCin(11 downto 8);
4 MCout(7 downto 4) <= MC3 xor MCin(7 downto 4);
5 MCout(3 downto 0) <= MC3 xor MCin(3 downto 0);
```

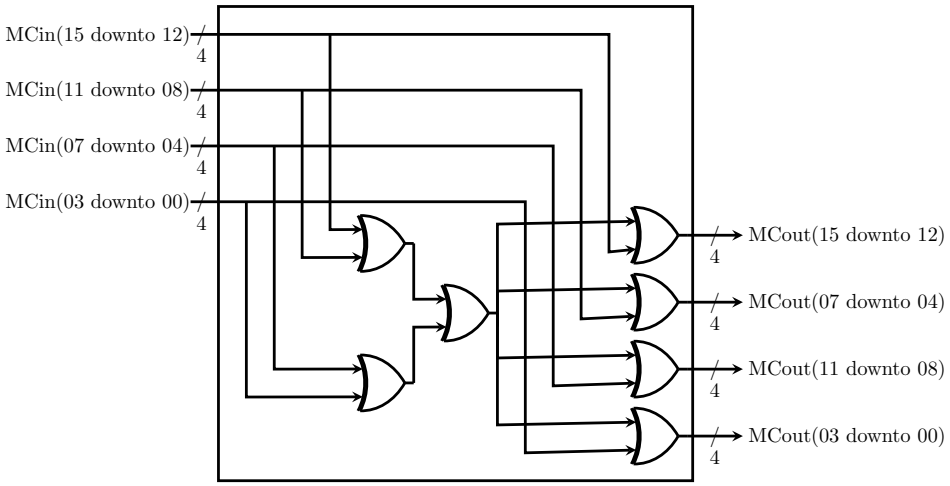

Figura 3.20: Componente $MixColumn$ de Midori64.

Se retoma el Algoritmo 3.8 considerando los siguientes aspectos propios de la implementación:

- El algoritmo inicia y termina con una suma realizada en el procedimiento $KeyAdd$, donde los parámetros son el mensaje en claro M y la llave de trabajo WK. Al final de cada ronda, también se ejecuta el procedimiento $KeyAdd$ con la salida $S = MCout$ del procedimiento $MixColumn$ y la llave RK_i (correspondiente a la ronda i) como parámetros de entrada. El componente $KMEM$ resuelve $RK = WK$ para la primer y última ronda, $RK = RK_i$ para cualquier otra ronda.

- El componente $SMEM$ recibe como entrada $SMin = S$, a la salida $S = SCout$ del procedimiento $SubCell$ en la última ronda, o a la salida $S = KAout$ del procedimiento $KeyAdd$ en cualquier otra ronda.

- Se sabe que los procedimientos $SubCell$ y $ShuffleCell$ corresponden a la capa lineal del algoritmo, por lo que el orden en el que se ejecutan no afecta el resultado. Por otra parte, la permutación del estado S en el procedimiento $ShuffleCell$ requiere el bloque completo de n bits, es decir, el procedimiento $ShuffleCell$ no permite su serialización, sin embargo existe un momento en el que el estado S permanece completamente almacenado en cada ronda. Por lo tanto, para motivos de implementación utilizando estrategia de serialización, se invierte el orden de los procedimientos $SubCell$ y $ShuffleCell$.

- Los elementos de la salida $SMout$ del procedimiento $SMEM$ son permutados dentro de la memoria, luego son sustituidos por sus respectivos valores en el componente $SubCell$, donde se encuentra implementada la Caja-S de Midori64.

- El procedimiento $MixColumn$ recibe como entrada $MCin = S$ a la salida $S = SCout$ del procedimiento $SubCell$, donde se realiza el cómputo definido por la Ecuación 3.31.

De lo anterior, se obtiene el Algoritmo 3.11.

Algoritmo 3.11 Midori64 sin cómputo previo.

Require: $M \in \{0,1\}^n, K \in \{0,1\}^k$
Ensure: $C \in \{0,1\}^n$

1: $\left(K\left[0\right], K\left[1\right]\right) \xleftarrow{n/2} K$
2: $WK \leftarrow K\left[0\right] \oplus K\left[1\right]$
3: $S \leftarrow M$
4: **for** $i = 0$ to 16 **do**
5: $\quad RK_i \leftarrow Expandir(\beta_i) \oplus K\left[i \mod 2\right]$
6: $\quad$ **if** $i = 0$ **then**
7: $\quad\quad RK \leftarrow WK$
8: $\quad\quad KAin \leftarrow S$
9: $\quad\quad SMin \leftarrow KAout$
10: $\quad$ **else if** $i = 16$ **then**
11: $\quad\quad RK \leftarrow WK$
12: $\quad\quad SMin \leftarrow SCout$
13: $\quad\quad C \leftarrow SMout \oplus KAout$
14: $\quad$ **else**
15: $\quad\quad RK \leftarrow RK_i$
16: $\quad\quad KAin \leftarrow MCout$
17: $\quad\quad SMin \leftarrow KAout$
18: $\quad$ **end if**
19: $\quad KAout \leftarrow KeyAdd(KAin, RK)$
20: $\quad SMout \leftarrow SMEM^{wr}(ShuffleCell(SMin))$
21: $\quad$ *Pulso de reloj*
22: $\quad SCout \leftarrow SubCell(SMout)$
23: $\quad MCout \leftarrow MixColumn(SCout)$
24: **end for**

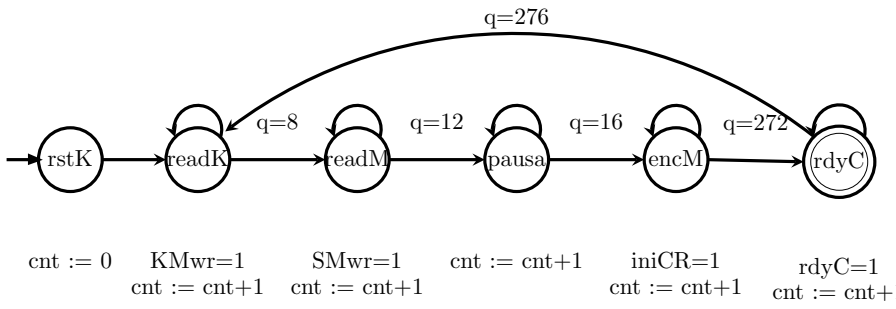

Figura 3.21: Máquina de estados de la unidad de control de Midori64.

Unidad de Control. Para la implementación de la unidad de control se considera la máquina de estados mostrada en la Figura 3.21.

La unidad de control implementa un circuito combinacional para gestionar las señales de control de los componentes durante la rutina de cifrado.

El diseño de Midori64 permite implementar el control prescindiendo de la implementación con un circuito secuencial para la máquina de estados, lo cual resulta en un circuito ligero.

De la Figura 3.21 se tienen los siguientes estados identificados por las señales que representan cada bit del valor del contador *cnt* dentro del circuito de la unidad de control:

- *rstK*: El estado inicial. Todos los registros son reinicializados, se prepara a los componentes de almacenamiento para comenzar la carga de los datos, los contadores son inicializados en cero y el control pasa de manera incondicional al estado *readK*.
- *readK*: La llave K es leída con hardware de 16 bits y almacenada con el componente $KMEM$. El proceso de almacenamiento de la llave tarda ocho ciclos de reloj, cuando ocurre que el contador $q = 8$, el control pasa al estado *readM*.
- *readM*: El estado S es leído con hardware de 16 bits y almacenado con el componente $SMEM$. El proceso de almacenamiento del estado tarda cuatro ciclos de reloj, cuando ocurre que el contador $q = 12$, el control pasa al estado *pausa*.

- *pausa*: Son preparados los componentes para iniciar con el proceso de cifrado. Los multiplexores para la selección del elemento del estado a ser sustituido con el componente SubCell conmutan sus entradas con los datos adecuados. La llave de ronda se prepara para ser utilizada en la primera ronda. Se ejecuta la primera permutación del estado con el procedimiento $ShuffleCell$. Además, el contador llega al valor $q = 0x10$, con lo que resulta fácil indicar el inicio del proceso de cifrado, ya que el quinto bit más significativo es visto como bandera para el inicio de las rondas. Cuando ocurre que el contador $q = 16$, el control pasa al estado *encM*.

- *encM*: Cada ronda tiene una duración de 16 ciclos de reloj, se procesa un elemento del estado en cada ciclo, cuando se tiene la ultima actualización en el registro del componente SMEM, se ejecuta la permutación correspondiente al procedimiento $ShuffleCell$. La sustitución $SubCell$ se ejecuta con hardware de cuatro bits, es por ello que la ejecución de la ronda tarda 16 ciclos, pues es procesado sólo un elemento del estado por ciclo. Los procedimientos $MixColumn$ y $KeyAdd$ se ejecutan con hardware de $n/4 = 16$ bits. Cuando ocurre que el contador $q = 272$, el control pasa al estado *rdyC*.

- *rdyC*: Es ejecutada la última ronda, además se activa la bandera que indica que el bloque en el puerto de salida C corresponde a la cifra del mensaje M. Cuando ocurre que el contador $q = 276$, el control pasa al estado *readK* para procesar un nuevo bloque. Como puede verse en el Algoritmo 3.9, el cómputo de las llaves de ronda no implica un diseño excesivo con respecto al área, y tampoco compromete el tiempo de ejecución, por lo que la ejecución es reinicializada por completo.

Con base en el Algoritmo 3.11 y en el diseño de la máquina de estados ilustrada en la Figura 3.21, se obtiene el diagrama 3.22 que describe la arquitectura en hardware de Midori64, implementado con la estrategia de serialización, utilizando hardware de 16 bits. Los componentes SMEM y KMEM son los únicos que incorporan registros de 64 bits, el componente SMEM almacena el estado S y KMEM las llaves $\left(K\left[0\right], K\left[1\right]\right) \overset{n}{\leftarrow} K$ y $WK \leftarrow K\left[0\right] \oplus K\left[1\right]$.

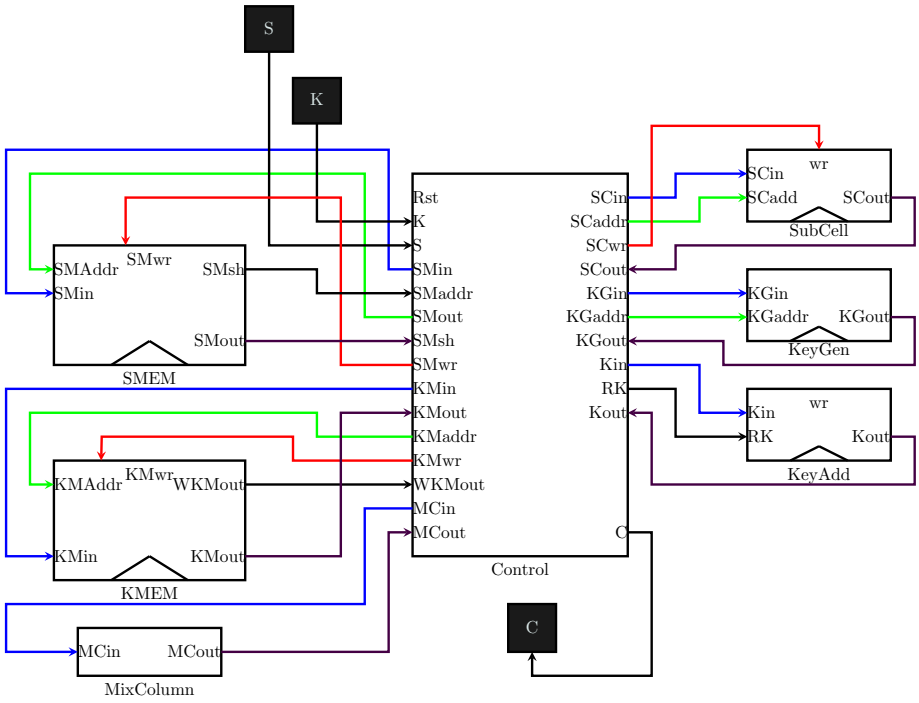

Figura 3.22: Arquitectura de Midori64.

Las tablas 4.1 y 4.2 del Capítulo 4 (Resultados) muestran los resultados obtenidos de la implementación utilizando el método combinacional y con almacenamiento respectivamente. En ellas se ilustra también el área ocupada en términos de LUTs, FlipFlops y bloques de RAM por sus respectivos componentes.

El código fuente de la implementación está disponible en el enlace del apéndice B.2 correspondiente.

3.2.3. COFB

Para la implementación en hardware del esquema COFB, se consideran aspectos propios de cada cifrador utilizados como primitiva (AES128 y Midori64). Un ejemplo es la longitud de las llaves, a pesar de que AES128 y Midori64 trabajan con longitudes del estado S distintas ($n_a = 128$ y $n_m = 64$ bits respectivamente), la longitud de sus llaves $K_a = Km = 128$ es la misma. Otra particularidad radica en el número de ciclos que tarda cada uno

de ellos en procesar un bloque; para tratar con el problema se implementan señales que indican el fin del cifrado de cada bloque (señal $rdyC$).

Teniendo en cuenta lo anterior y con base en el Algoritmo 3.3 se sigue el desarrollo de la presente implementación.

Algoritmo 3.12 MaskGen

Require: $K \in \{0,1\}^k, N \in \{0,1\}^{n/2}$
Ensure: $\Delta \in \{0,1\}^{n/2}, Y\,[0] \in \{0,1\}^n$
1: $S \leftarrow 0^{n/2}||N$
2: $Y\,[0] \leftarrow E_k(S)$
3: $(Y_0[0], ..., Y_3[0]) \xleftarrow{n/4} Y\,[0]$
4: $\Delta \leftarrow Y_2[0]||Y_3[0]$

El primer paso en el Algoritmo 3.3 es calcular el valor de Δ, el procedimiento $MaskGen$ concatena $0^{n/2}$ ceros como parte más significativa del Nonce y lo cifra obteniendo como resultado a $Y\,[0]$, de donde se obtiene Δ, tal y como lo muestra el Algoritmo 3.12.

El procedimiento Frm da formato a los datos de entrada A y M devolviendo su concatenación en un arreglo B, tal como lo ilustra la Ecuación 3.32:

$$(B[1], ..., B[l]) \leftarrow (A[1], ..., A[a])||(M[1], ..., M[m]) \tag{3.32}$$

Además calcula un arreglo bidimensional $t[i]$ (donde $i \in \{1, ..., a+m\}$) de vectores con los exponentes utilizados en el cálculo de la máscara $mask$. Sin embargo, el procesamiento para separar en bloques a las cadenas A y M se obvia, ya que el lenguaje permite moldear datos en la memoria de almacenamiento.

Por otra parte, el cálculo de la máscara $mask$ se realiza al vuelo para cada bloque que es procesado, por lo que los elementos del arreglo $t[i]$ son calculados al vuelo de manera subyacente en el cálculo de la función $mask_\Delta(i)$ del Algoritmo 3.13.

Algoritmo 3.13 COFB reorganizado sin el arreglo $t[i]$

Require: $K \in \{0,1\}^n, N \in \{0,1\}^{n/2}, A \in \{0,1\}^r, M \in \{0,1\}^s$
Ensure: $C \in \{0,1\}^m, T \in \{0,1\}^n$
1: $(\Delta, Y[0]) \leftarrow MaskGen(K, N)$
2: $(A[1], ..., A[a]) \xleftarrow{n} A$
3: $(M[1], ..., M[m]) \xleftarrow{n} M$
4: $l \leftarrow a + m$
5: $(B[1], ..., B[l]) \leftarrow (A[1], ..., A[a])||(M[1], ..., M[m])$
6: **for** $i = 1$ to l **do**
7: $\quad X[i] \leftarrow (B[i] \oplus G \cdot Y[i-1]) \ \oplus \ mask_\Delta(i)||0^{n/2}$
8: $\quad Y[i] \leftarrow E_K(X[i])$
9: $\quad$ **if** $i > a$ **then**
10: $\quad\quad C[i-a] \leftarrow Y[i-1] \ \oplus \ M[i-a]$
11: $\quad$ **end if**
12: **end for**
13: $T \leftarrow Y[l]$

La máscara $mask : \{0,1\}^{n/2} \times \mathbb{N}^2 \mapsto \mathbb{N}^2$ se define como:

$$mask_\Delta(t_0, t_1) = \alpha^{t_0} \cdot (1+\alpha)^{t_1} \cdot \Delta \tag{3.33}$$

Donde:

$$t(i) = \begin{cases} (i, 0) & \text{si } i < a \\ (a-1, \delta_A) & \text{si } i = a \\ (i-1, \delta_A) & \text{si } a < i < a+m \\ (a+z-2, \delta_A + \delta_M) & \text{si } i = a+m \end{cases} \tag{3.34}$$

Para una cadena binaria $B \in \{0,1\}^*$, δ se define como:

$$\delta_B = \begin{cases} 1 & \text{si } B \neq \lambda \text{ y } 0 \equiv |B| \mod n \\ 2 & \text{en otro caso} \end{cases} \tag{3.35}$$

sin embargo, bajo la suposición de que $B \neq \lambda$ y $0 \equiv |B| \mod n$ tomada del hecho de leer bloques completos a partir de la entrada estándar, $\delta_B = 1$ siempre se cumple.

Las operaciones de multiplicación $\cdot$ y suma $+$, se calculan módulo $p(x)$, elegido como un polinomio de la forma $p(x) = x^n + g$, donde g es el mínimo

polinomio y el primero en orden lexicográfico. El polinomio $p(x)$ utilizado para la reducción, se obtuvo para ambas versiones (COFB-AES128 y COFB-Midori64) con la instrucción `pn<x>:=IrreducibleLowTermGF2Polynomial(n);` de la herramienta *MAGMA Calc*[2]. α denota al elemento primitivo del campo, que de acuerdo a Rogaway[36] $\alpha = 2$.

Luego, de las Ecuaciones 3.34 y 3.35 se obtiene la Ecuación 3.36.

$$t'(i) = \begin{cases} (i, 0) & \text{si } i < a \\ (a-1, 1) & \text{si } i = a \\ (i-1, 1) & \text{si } a < i < a+m \\ (a+z-2, 2) & \text{si } i = a+m \end{cases} \tag{3.36}$$

Para después obtener la Ecuación 3.37 a partir de las Ecuaciones 3.36 y 3.33.

$$mask_{\Delta}(i) = \begin{cases} 2^i \cdot 3^0 \cdot \Delta & \text{si } i < a \\ 2^{a-1} \cdot 3^1 \cdot \Delta & \text{si } i = a \\ 2^{i-1} \cdot 3^1 \cdot \Delta & \text{si } a < i < a+m \\ 2^{a+m-2} \cdot 3^2 \cdot \Delta & \text{si } i = a+m \end{cases} \tag{3.37}$$

Del análisis de la Ecuación 3.37 se tiene la Tabla 3.8, donde se toma en cuenta la manera en la que el lenguaje VHDL enumera los elementos de un arreglo (el primer elemento es el 0), por lo que el último elemento de un arreglo V de longitud v, es el elemento $V[v-1]$. Además, se considera la propiedad conmutativa para la operación producto $\cdot$ en el campo.

i	**msk**	**mx2**	**mx2x3**	**mx2x3x3**	**mask**
0	$2^0 \cdot 3^0 \cdot \Delta$	delta	gtriple(mx2)	gtriple(mx2x3)	mx2
1	$2^1 \cdot 3^0 \cdot \Delta$	gdoble(mx2)	gtriple(mx2)	gtriple(mx2x3)	mx2
...	$2^i \cdot 3^0 \cdot \Delta$	gdoble(mx2)	gtriple(mx2)	gtriple(mx2x3)	mx2
a	$2^{a-1} \cdot 3^1 \cdot \Delta$	mx2	gtriple(mx2)	gtriple(mx2x3)	mx2x3
$a+1$	$2^{i-1} \cdot 3^1 \cdot \Delta$	gdoble(mx2)	gtriple(mx2)	gtriple(mx2x3)	mx2x3
...	$2^{i-1} \cdot 3^1 \cdot \Delta$	gdoble(mx2)	gtriple(mx2)	gtriple(mx2x3)	mx2x3
$a+m$	$2^{a+m} \cdot 3^2 \cdot \Delta$	mx2	gtriple(mx2)	gtriple(mx2x3)	mx2x3x3

Tabla 3.8: Cálculo de la máscara de COFB.

[2] `http://magma.maths.usyd.edu.au/magma/`

Las señales $iniD$, $finA$ y $finD$ reemplazan al contador de bloques $i \in \{0, ..., a+m\}$ de la Tabla 3.8, tal como lo indica la Tabla 3.9.

iniD‖finA‖finM	**msk**	**mx2**	**mx2x3**	**mx2x3x3**	**mask**
000	$2^0 \cdot 3^0 \cdot \Delta$	delta	gtriple(mx2)	gtriple(mx2x3)	mx2
100	$2^1 \cdot 3^0 \cdot \Delta$	gdoble(mx2)	gtriple(mx2)	gtriple(mx2x3)	mx2
...	$2^i \cdot 3^0 \cdot \Delta$	gdoble(mx2)	gtriple(mx2)	gtriple(mx2x3)	mx2
110	$2^{a-1} \cdot 3^1 \cdot \Delta$	mx2	gtriple(mx2)	gtriple(mx2x3)	mx2x3
110	$2^{i-1} \cdot 3^1 \cdot \Delta$	gdoble(mx2)	gtriple(mx2)	gtriple(mx2x3)	mx2x3
...	$2^{i-1} \cdot 3^1 \cdot \Delta$	gdoble(mx2)	gtriple(mx2)	gtriple(mx2x3)	mx2x3
111	$2^{a+m} \cdot 3^2 \cdot \Delta$	mx2	gtriple(mx2)	gtriple(mx2x3)	mx2x3x3

Tabla 3.9: Cálculo de la máscara de COFB con las señales $iniD$, $finA$ y $finD$.

Algoritmo 3.14 COFB con señales $iniD$, $finA$ y $finM$.

Require: $K \in \{0,1\}^n, N \in \{0,1\}^{n/2}, A[0] \in \{0,1\}^n, M[0] \in \{0,1\}^n, iniD \in \{0,1\}, finA \in \{0,1\}, finM \in \{0,1\}$
Ensure: $C \in \{0,1\}^m, T \in \{0,1\}^n$

$(\Delta, Y_0) \leftarrow MaskGen(K, N)$
$i \leftarrow 0$
$enc \leftarrow 0$
while $finM \neq 1$ **do**
  if $enc = 0$ **then**
    $B \leftarrow A[0]$
  else
    $B \leftarrow M[0]$
    $C[i] \leftarrow Y_0 \oplus B$
  end if
  $s \leftarrow mask_\Delta(s, iniD, finA, finM)$
  $X \leftarrow (B \oplus G \cdot Y_0) \oplus s || 0^{n/2}$
  $Y_1 \leftarrow E_K(X)$
  if $finA = 0$ **then**
    $enc \leftarrow 1$
  end if
  $Y_0 \leftarrow Y_1$
end while
$T \leftarrow Y_1$

El Algoritmo 3.13 es reorganizado considerando la sustitución en la Tabla 3.9, además, se propone que los bloques de las cadenas A y M sean procesados al vuelo, es decir, que no exista un almacenamiento previo (ni su respectivo procesamiento en bloques ilustrado en la ecuación 3.32). La manera en la que se conoce la evolución del procesamiento de los bloques, es a través de las señales $iniD$, $finA$ y $finM$. De lo anterior se obtiene el Algoritmo 3.14, en el cual se basa la implementación del esquema.

Componente CKMEM El componente **CKMEM** almacena la llave inicial K utilizada para el cifrado de todos los bloques tanto del *Nonce* como de los datos asociados A y del mensaje M. Para el caso de COFB-Midori64, corresponde al componente KMEM de Midori64; para el caso de COFB-AES128, se implementa como un registro de 128 bits. Su diseño se ilustra con la Figura 3.23 para COFB-AES128 y en la Figura 3.24 para COFB-Midori64.

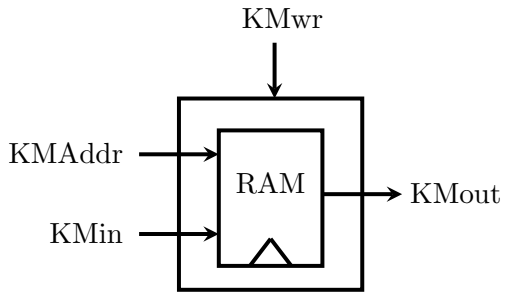

Figura 3.23: Componente CKMEM de COBF-AES128.

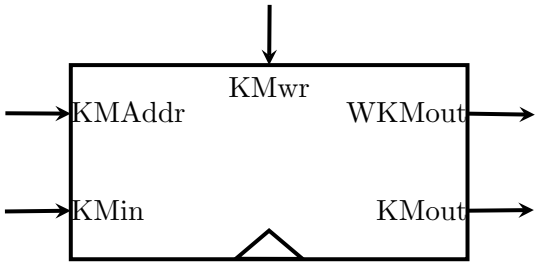

Figura 3.24: Componente CKMEM de COBF-Midori64.

Con base en el Algoritmo 3.14 es retomada la Ecuación 3.37 para ser definida la Ecuación 3.38, en donde se contemplan las señales de control

$iniD$, $finA$ y $finM$ mencionadas en la Tabla 3.9.

$$mask_{\Delta}(s, iniD, finA, finM) = \begin{cases} 2 \cdot \Delta & \text{si } iniD \| finA \| finM = 000 \\ 2 \cdot s & \text{si } iniD \| finA \| finM = 100 \\ 3 \cdot s & \text{si } iniD \| finA \| finM = 010 \\ 2 \cdot 3 \cdot s & \text{si } iniD \| finA \| finM = 110 \\ 2 \cdot 3 \cdot 3 \cdot s & \text{si } iniD \| finA \| finM = 111 \end{cases} \tag{3.38}$$

Componente CMASK De la Ecuación 3.38 se implementa el componente de enmascaramiento CMASK ilustrado en el diagrama de la Figura 3.25 con sus respectivos componentes (GDoble y GTriple), donde la aritmética implementada consiste en sumas y doblados dentro del campo GF($2^{n/2}$)(el producto por 3 resulta de una suma y un doblado).

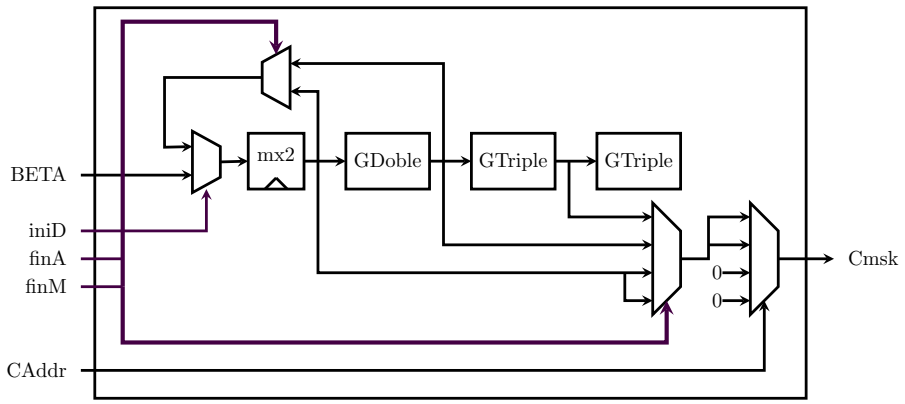

Figura 3.25: Componente CMASK de COFB.

La señal $BETA$ recibe el valor Δ calculado en el Algoritmo 3.12, las señales $iniD$, $finA$ y $finD$ son proporcionadas como entradas para la implementación e indican cuando comienza el proceso de cifrado, cuando es proporcionado el último bloque de datos asociados A y el último bloque del mensaje M respectivamente. La señal $CAdrr$ determina el fragmento (parte alta, baja o valor de ceros) de la máscara que es devuelto a la salida.

Componente CDMEM El componente **CDMEM** (su diseño lo muestra la Figura 3.26) almacena el valor de Δ. Es diseñado como un registro de $n/2$ bits.

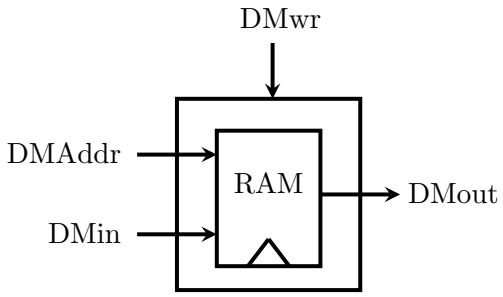

Figura 3.26: Componente CDMEM de COBF.

La operación $G \cdot Y_0$ se explica en el Algoritmo 3.15.

Algoritmo 3.15 Operación $G \cdot Y_0$

Require: $Y_0 \in \{0,1\}^n$
Ensure: $G \cdot Y_0 \in \{0,1\}^n$
1: $(Y_0[0], Y_0[1], Y_0[2], Y_0[3]) \xleftarrow{n/4} Y_0$
2: $G \cdot Y_0 \leftarrow (Y_0[0], Y_0[1], Y_0[2], Y_0[3] \oplus Y_0[0])$

De la Ecuación 2.12, se tiene que $(B \oplus G{\cdot}Y_0) \,\overline{\oplus}\, mask_\Delta(s, iniD, finA, finM)$ equivale a $(B \oplus G \cdot Y_0) \oplus mask_\Delta(s, iniD, finA, finM)||0^{n/2}$, por lo tanto, queda justificada la Ecuación 3.39.

$$X \leftarrow (B \oplus G \cdot Y_0) \oplus mask_\Delta(s, iniD, finA, finM)||0^{n/2} \tag{3.39}$$

para la cual, su implementación consiste en operaciones con compuertas XOR ordinarias.

Después de calcular la cifra $Y_1 \leftarrow E_k(X)$, se evalúan las señales de control $iniD$, $finA$ y $finM$, si $finM \neq 1$ los datos asociados A aún no terminan de ser procesados, de lo contrario, comienza a devolverse el mensaje cifrado. La Ecuación 3.40 ilustra el cómputo del mensaje cifrado.

$$C[i] \leftarrow Y_0 \underline{\oplus} M[0] \tag{3.40}$$

De la Ecuación 2.11 y del hecho de que todos los bloques son completos (por el formato dado desde la entrada estándar), se obtiene la Ecuación 3.41

$$C[i] \leftarrow Y_0 \oplus M[0] \tag{3.41}$$

para la cual, su implementación consiste en operaciones con compuertas XOR ordinarias.

El último bloque cifrado Y_1 es devuelto como el bloque verificador $T \in \{0,1\}^n$ (también llamado *Tag*).

Componente CYMEM El componente **CYMEM** (su diseño lo muestra la Figura 3.27) almacena el valor de Y_i (con $i \in \{0,1\}$). Es diseñado como un registro de $n/4$ bits.

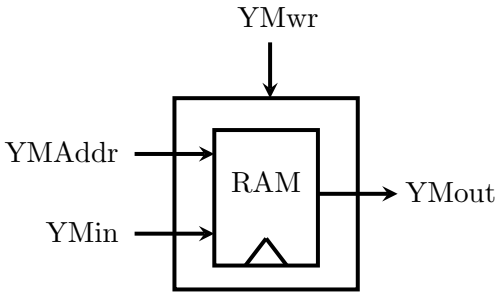

Figura 3.27: Componente CYMEM de COBF.

Unidad de Control Para el control de los componentes, se considera el diseño de una máquina de estados con la que es implementada la unidad de control, el diagrama se ilustra en la Figura 3.28.

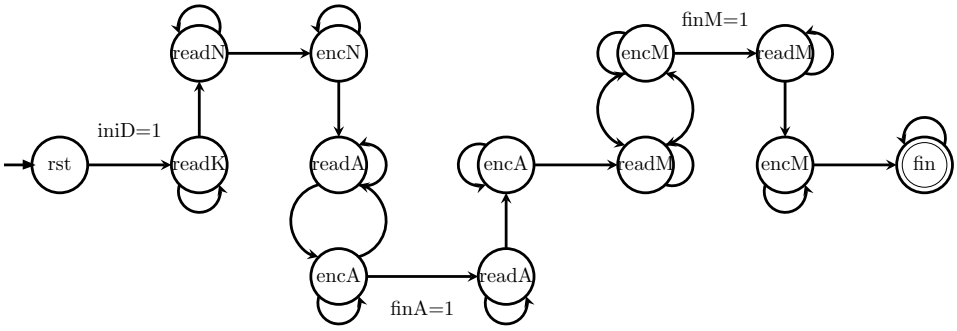

Figura 3.28: Máquina de estados de COFB.

COFB permite la implementación ligera de su control, las señales que controlan la ejecución son las dadas desde la entrada ($iniD$, $finA$ y $finM$) y

las señales que envía el cifrador ($getK$, $getS$ y $rdyC$). Para ello se consideran los siguientes estados:

- *rst*: Es el estado de inicio. Todas las señales son inicializadas, los contadores puestos a valor de cero y los registros son preparados. Todo ello ocurre en un ciclo de reloj, luego pasa de manera incondicional al estado *readK*.

- *readK*: Es leída la llave que se utilizará para cifrar el *Nonce* y todos los bloques. Es almacenada en el componente **CKMEM** y proporcionada al cifrador cuando éste envía la señal $getK$. El proceso tarda cuatro ciclos de reloj en el caso de COFB-AES128 y ocho ciclos para el caso de COFB-Midori64, luego el control pasa al estado *readN*.

- *readN*: Es leído el *Nonce* y dado como entrada al cifrador, el proceso tarda cuatro ciclos de reloj para ambos casos (COFB-AES128 y COFB-Midori64), luego el control pasa al estado *encN*. *encN*: Con el resultado de cifrar el *Nonce* concatenado con ceros, se obtienen Δ y Y_0, luego son almacenados en los componentes **CDMEM** y **CYMEM** respectivamente. Cuando ocurre que la señal $rdyC = 1$ el control pasa al estado *readA*.

- *readA*: Es un bloque de la cadena de datos asociados A, cuando ocurre que la señal $getS = 0$, el control pasa al estado *encA*.

- *encA*: Es cifrado el bloque leído de la cadena A, cuando ocurre que la señal $rdyC = 1$ entonces: si la señal $finA = 0$ entonces el control regresa al estado *readA*, de otro modo (si $finA = 1$) el control pasa al estado *readM*.

- *readM*: Es un bloque de la cadena del mensaje M, cuando ocurre que la señal $getS = 0$, el control pasa al estado *encM*.

- *encM*: Es cifrado el bloque leído de la cadena M, cuando ocurre que la señal $rdyC = 1$ entonces: si la señal $finM = 0$ entonces el control regresa al estado *readM*, de otro modo (si $finM = 1$) el cifrado del mensaje termina devolviendo el como bloque verificador T a la última cifra calculada.

El diagrama de la Figura 3.29 muestra la arquitectura de la implementación del esquema.

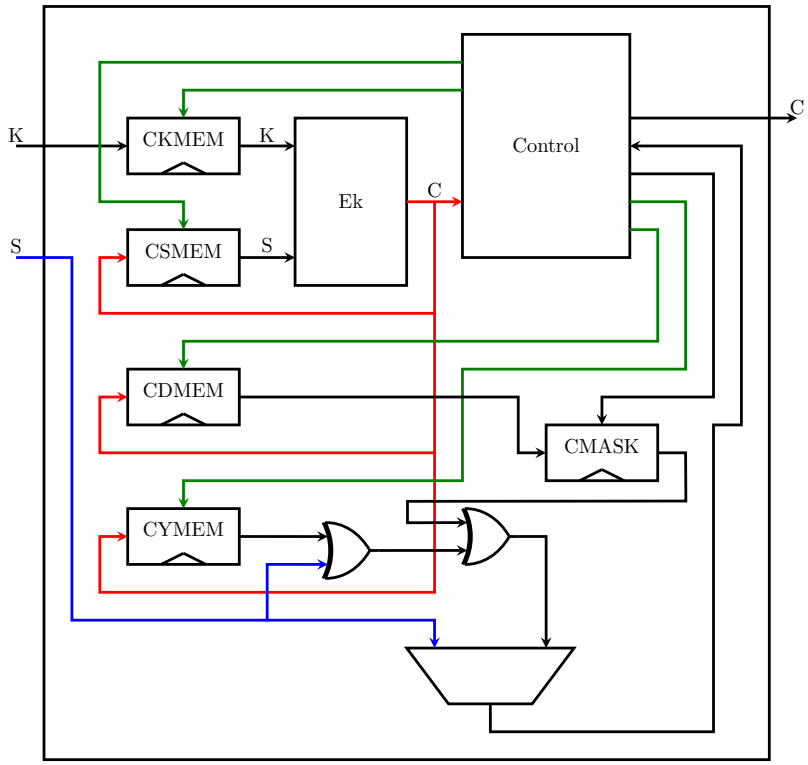

Figura 3.29: Arquitectura de COFB.

El código fuente de la implementación está disponible en el enlace del apéndice B.2 correspondiente.

El código fuente de la implementación de COFB-AES128 está disponible en el Enlace B.2.3 del apéndice B.2.

El código fuente de la implementación de COFB-Midori64 está disponible en el Enlace B.2.4 del apéndice B.2.

Capítulo 4

Resultados

Las implementaciones del esquema COFB usando Midori64 y AES128 se realizaron utilizando la estrategia de serialización, además se utilizaron los métodos de diseño combinacional y con almacenamiento. No fue utilizada la estrategia de ancho completo, porque por definición no es una técnica para diseño de hardware ligero.

El diseño se implementó en FPGAs de las familias Artix 7 y Virtex 7, se procedió a generar resultados para medir el área ocupada así como la velocidad. Artix 7 es una familia de FPGAs e bajo consumo energético mientras que los Virtex 7 son de alto rendimiento. En el artículo donde se presenta COFB [1] los resultados resultados experimentales se obtuvieron en FPGA de la familia Virtex 7, para tener una comparación justa implementamos nuestro diseño en el mismo dispositivo Virtex 7 que los autores de COFB.

En la tabla 4.1 se ilustra la cantidad de recursos ocupados por la implementación combinacional de Midori64.

	Componente	**LUTs**	**FlipFlops**	**RAM**
	CNT (contador)	148	9	0
	SubCell	7	0	0
	KMEM	38	0	0
	SMEM	32	64	0
Total	Midori64	225	73	0

Tabla 4.1: Resultados de la implementación combinacional de Midori64.

En la tabla 4.2 se ilustra la cantidad de recursos ocupados por la implementación con almacenamiento de Midori64.

	Componente	**LUTs**	**FlipFlops**	**RAM**
	CNT (contador)	95	9	0
	SubCell	42	16	0
	KMEM	38	0	0
	SMEM	42	64	0
Total	Midori64	217	73	0

Tabla 4.2: Resultados de la implementación con almacenamiento de Midori64.

En la tabla 4.3 se ilustra la cantidad de recursos ocupados por la implementación combinacional de AES128.

	Componente	**LUTs**	**FlipFlops**	**RAM**
	Unidad de control	127	9	0
	KMEM	97	32	0
	SBOX0	70	19	0
	SBOX1	91	24	0
	SBOX2	68	32	0
	SBOX3	80	24	0
	MixColumn	32	8	0
	SHReg0	12	8	0
	SHReg1	16	8	0
	SHReg2	19	8	0
	RoundConst	32	8	0
Total	AES128	614	212	0

Tabla 4.3: Resultados de la implementación combinacional de AES128.

En la tabla 4.4 se ilustra la cantidad de recursos ocupados por la implementación con almacenamiento de AES128.

	Componente	LUTs	FlipFlops	RAM
	Unidad de control	108	9	0
	KMEM	144	32	0
	MixColumn	32	8	0
	SBOX0	0	0	0.5
	SBOX1	0	0	0.5
	SHReg0	12	19	0
	SHReg1	12	24	0
	SHReg2	12	32	0
	SHReg2	12	24	0
	RoundConst	22	8	0
Total	AES128	342	180	0

Tabla 4.4: Resultados de la implementación con almacenamiento de AES128.

En la Tabla 4.5 presenta los resultados obtenidos, donde AT denota la estrategia de ancho total, SE la estrategia de serialización, AL el método con almacenamiento y CO el método combinacional. .

Versión	Dispositivo	Estrategia	LUTs	Ciclos por bloque	Frecuencia (MHz)	Tamaño de bloque (bits)	Ancho de bus (bits)	Velocidad (Gbps)
COFB	Virtex 7	AT	1456	12	264.24	128	128	2.82
COFB-AES128	Virtex 7	SE / AL	594	96	277.00	128	32	0.092
COFB-AES128	Virtex 7	SE / CO	675	96	222.00	128	32	0.074
COFB-Midori64	Virtex 7	SE / AL	360	279	294.00	64	16	0.017
COFB-Midori64	Virtex 7	SE / CO	351	279	263.00	64	16	0.015
COFB-AES128	Artix 7	SE / AL	597	96	125.00	128	32	0.041
COFB-AES128	Artix 7	SE / CO	679	96	112.35	128	32	0.037
COFB-Midori64	Artix 7	SE / AL	358	279	153.84	64	16	0.009
COFB-Midori64	Artix 7	SE / CO	351	279	142.85	64	16	0.008

Tabla 4.5: Tabla de resultados de la implementación de COFB.

De los resultados obtenidos puede observarse que la implementación más ligera obtenida es COFB-Midori64 implementado con la estrategia de serialización utilizando un método combinacional para el cálculo de la caja-S y así evitar el uso de memorias.

Por otra parte, la implementación menos ligera fue COFB-AES128, publicada en el trabajo [1] de Chakraborty *et. al.* la cual emplea una estrategia de ancho total.

4.1. Análisis de los resultados

Midori64 y COFB son implementaciones orientadas al uso reducido de recursos de hardware, por lo que COFB-Midori64 resulta una solución ventajosa sobre la implementación de la publicación original.

Midori64 ocupó 136 LUTs en su diseño, mientras que AES128 con el método de almacenamiento ocupó 376 LUTs. El esquema COFB-Midori64 fue diseñado con hardware de 16 bits, mientras que COFB-AES128 se realizó utilizando hardware de 32 bits. La implementación COFB original se implementó con hardware de 128 bits, por lo que su área ocupada es la mayor.

Para COFB-Midori64 SE/AL se tiene que el espacio ocupado es 1.01 (sobre Artix 7) y 1.02 (sobre Virtex 7) veces mayor que el espacio ocupado por COFB-Midori64 SE/CO, con lo que puede verse que en este caso, el método combinacional de implementación ha resultado en una mejora con respecto al tamaño del circuito. Por otra parte, para COFB-AES128 SE/AL se tiene que el espacio ocupado es 1.13 (sobre ambos: Artix 7 y Virtex 7) veces mayor que el espacio ocupado por COFB-128 SE/AL, con lo que se observa que en este caso, el método de almacenamiento es el que resultó más ventajoso.

Un diseño ligero no contempla el tiempo que se necesita para la ejecución del sistema, sin embargo de los resultados presentados puede verse que la implementación más compacta es precisamente la mas lenta, procesando 0.008 Gbps, la más veloz es la implementación original COFB del trabajo [1], logrando procesar 2.82 Gbps.

Capítulo 5

Conclusiones

En el presente trabajo ha sido presentada la implementación ligera con VHDL del esquema COFB utilizando como primitivas de cifrado tanto a Midori64 como AES128, de las cuales (según los resultados obtenidos) la versión más ligera es COFB-Midori64 logrando una optimización empleando estrategia de serialización y el método combinacional para el diseño de los componentes que utilizan almacenamiento.

Tanto los algoritmos de los cifradores AES128 y Midori64 así como el del esquema COFB fueron reorganizados (las transformaciones lineales únicamente por lo que la correctitud de la salida se mantiene) para lograr una optimización adicional en la implementación. Además fueron considerados aspectos propios de cada diseño que aportaron una mejora en el uso de elementos en los circuitos.

5.1. Discusión

A pesar de que el objetivo del presente trabajo ha sido alcanzado, es posible optimizar aún más al diseño propuesto tanto de COFB-Midori64 como de COFB-AES128. Por ejemplo, para el diseño de Midori64 se implementó un componente para la sustitución $SubCell$ que procesa un elemento del estado S a la vez, mientras que para AES128 la implementación del procedimiento $SubBytes$ procesa cuatro elementos del estado S en cada ciclo de reloj, lo que compromete el espacio ocupado (no así con el tiempo de ejecución).

El número de ciclos ocupados para el procesamiento de cada bloque se ha visto comprometido en gran medida (un factor de casi de 2^8 veces), sin embargo el objetivo de reducir el espacio de la implementación se alcanzó, la versión más compacta mide aproximadamente una cuarta parte del área ocupada por la implementación COFB original.

Existen técnicas para la implementación sobre cada dispositivo en particular, la presente implementación es mas bien genérica, por lo que el diseño para un dispositivo específico puede ofrecer una optimización adicional que resulte en una solución aún más compacta.

5.2. Trabajo a futuro

En el presente trabajo se proponen dos implementaciones del esquema COFB: COFB-Midori64 y COFB-AES128. En el proceso de diseño se han empleado técnicas y métodos que reducen el área ocupada por las implementaciones, además del análisis y reorganización de sus algoritmos para evitar el uso excesivo de recursos por los circuitos.

Sin embargo, como ocurre en la mayoría de las implementaciones, la presentada en éste trabajo ofrece varias oportunidades de mejora.

La implementación de AES128 aún puede optimizarse evitando el almacenamiento de las llaves de ronda y en su lugar implementar una etapa de control que sea capaz de reutilizar los componentes (como lo hace la implementación presentada) para calcular las llaves de ronda al vuelo sin comprometer con ello el espacio ocupado. El diseño de un control para la serialización de algún procedimiento determinado, representa un costo tanto de espacio como de tiempo de ejecución. En el método combinacional, es necesario adecuar las ecuaciones booleanas (y en general el diseño completo) obtenidas de manera que sea aprovechada la capacidad ofrecida por las 6-LUTs, de otro modo no representa una mejora con respecto al uso de espacio en implementaciones sobre FPGAs; una implementación con método combinacional resulta óptima para diseños ASIC.

La implementación de Midori64 resultó en una solución compacta, pero aún es posible lograr optimizaciones, por ejemplo, manipulando el diseño

combinacional para adecuar sus ecuaciones a una implementación sobre 6-LUTs, lo que resultaría en una implementación para un dispositivo específico pero con una mejora con respecto al uso reducido de recursos.

Para el caso del esquema, según el cifrador (Midori64 o AES128) la longitud del hardware empleado en el diseño de sus componentes es variable (16 o 32 bits respectivamente), sin embargo es posible realizar un diseño compacto independiente del tamaño del bloque que procesa su primitiva de cifrado. Además de la adecuación de su diseño para un dispositivo específico, el análisis de la aritmética empleada puede ofrecer mejoras.

Por otra parte, el diseño presentado puede implementarse sobre FPGAs o incluso otros dispositivos (diseño ASIC por ejemplo), con ellos existe la oportunidad de realizar un análisis detallado sobre el consumo de potencia, o vulnerabilidad ante ataques por canales laterales, por ejemplo. La implementación sobre un dispositivo físico ofrece grandes oportunidades para realizar trabajo de investigación seguido de la presente tesis.

Bibliografía

[1] Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, and Mridul Nandi. Blockcipher-based Authenticated Encryption: How Small Can We Go? Cryptology ePrint Archive, Report 2017/649, 2017. `http://eprint.iacr.org/2017/649`.

[2] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466, Vienna, Austria, September 10–13, 2007. Springer, Heidelberg, Germany. `http://lightweightcrypto.org/present/present_ches2007.pdf`.

[3] Tetsu Iwata, Kazuhiko Minematsu, Jian Guo, and Sumio Morioka. CLOC: Authenticated Encryption for Short Input. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption – FSE 2014*, volume 8540 of *Lecture Notes in Computer Science*, pages 149–167, London, UK, March 3–5, 2015. Springer, Heidelberg, Germany. `https://eprint.iacr.org/2014/157.pdf`.

[4] Tetsu Iwata, Kazuhiko Minematsu, Jian Guo, Sumio Morioka, and Eapeelido Kobayashi. SILC: SImple Lightweight CFB. `http://competitions.cr.yp.to/round1/silcv1.pdf`.

[5] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. In *ACM CCS 01: 8th Conference on Computer and Communications Security*, pages 196–205, Philadelphia, PA, USA, November 5–8, 2001. ACM Press. `http://web.cs.ucdavis.edu/~rogaway/papers/ocb-full.pdf`.

[6] Phillip Rogaway. Authenticated-Encryption With Associated-Data. In Vijayalakshmi Atluri, editor, *ACM CCS 02: 9th Conference on Computer and Communications Security*, pages 98–107, Washington D.C., USA, November 18–22, 2002. ACM Press. `http://web.cs.ucdavis.edu/~rogaway/papers/ad.pdf`.

[7] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Siang Meng Sim, Yosuke Todo, and Yu Sasaki. GIFT: A Small Present. Cryptology ePrint Archive, Report 2017/622, 2017. `http://eprint.iacr.org/2017/622`.

[8] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A Block Cipher for Low Energy. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015, Part II*, volume 9453 of *Lecture Notes in Computer Science*, pages 411–436, Auckland, New Zealand, November 30– December 3, 2015. Springer, Heidelberg, Germany. `https://eprint.iacr.org/2015/1142`.

[9] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404, 2013. `http://eprint.iacr.org/2013/404`.

[10] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. Atomic-AES: A Compact Implementation of the AES Encryption/-Decryption Core. In Orr Dunkelman and Somitra Kumar Sanadhya, editors, *Progress in Cryptology - INDOCRYPT 2016: 17th International Conference in Cryptology in India*, volume 10095 of *Lecture Notes in Computer Science*, pages 173–190, Kolkata, India, December 11–14, 2016. Springer, Heidelberg, Germany. `https://eprint.iacr.org/2016/927.pdf`.

[11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 69–88, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany. `https://iacr.org/archive/eurocrypt2011/66320067/66320067.pdf`.

[12] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A compact rijndael hardware architecture with S-box optimization. In Colin Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, Gold Coast, Australia, December 9–13, 2001. Springer, Heidelberg, Germany. https://www.iacr.org/archive/asiacrypt2001/22480241.pdf.

[13] D. Whiting, R. Housley, and N. Ferguson. Counter with CBC-MAC (CCM). RFC 3610 (Informational), September 2003. http://www.ietf.org/rfc/rfc3610.txt.

[14] Mihir Bellare, Phillip Rogaway, and David Wagner. The EAX Mode of Operation. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption – FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 389–407, New Delhi, India, February 5–7, 2004. Springer, Heidelberg, Germany. https://www.iacr.org/archive/fse2004/30170391/30170391.pdf.

[15] Kazuhiko Minematsu, Stefan Lucks, Hiraku Morita, and Tetsu Iwata. Attacks and Security Proofs of EAX-Prime. Cryptology ePrint Archive, Report 2012/018, 2012. http://eprint.iacr.org/2012/018.

[16] Hongjun Wu. Acorn: A lightweight authenticated cipher (v1). *CAESAR First Round Submission, competitions. cr. yp. to/round1/acornv1. pdf*, 2014. https://competitions.cr.yp.to/round3/acornv3.pdf.

[17] Hongjun Wu and Bart Preneel. AEGIS: A Fast Authenticated Encryption Algorithm. Cryptology ePrint Archive, Report 2013/695, 2013. http://eprint.iacr.org/2013/695.

[18] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Elmar Tischhauser, Kan Yasuda, and DTU Compute. AES-COPA v. 2. *CAESAR submission*, 2015. https://competitions.cr.yp.to/round2/aescopav2.pdf.

[19] Morris J Dworkin. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. *Special Publication (NIST SP)-800-38D*, 2007. https://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf.

[20] K Minematsu. AES-OTR. Submission to the CAESAR Competition (Round 2), 2016. https://competitions.cr.yp.to/round3/aesotrv31.pdf.

[21] Viet Tung Hoang, Ted Krovetz, and Phillip Rogaway. AEZ v1: Authenticated-Encryption by Enciphering. *CAESAR 1st Round, competitions. cr. yp. to/round1/aezv1. pdfl*, 2014. https://competitions.cr.yp.to/round3/aezv42.pdf.

[22] C Dobraunig, M Eichlseder, F Mendel, and M Schläffer. Ascon. Submission to the CAESAR competition (2014), 2014. https://competitions.cr.yp.to/round3/asconv12.pdf.

[23] Jérémy Jean, Ivica Nikolic, Thomas Peyrin, and Yannick Seurin. Deoxys v1. 4. *CAESAR candidate*, 2016. https://competitions.cr.yp.to/round3/deoxysv141.pdf.

[24] N Datta and M Nandi. ELmD v1. 0. Submission to the Caesar competition, 2015. https://competitions.cr.yp.to/round2/elmdv21.pdf.

[25] Authentication Encryption Mode. The JAMBU Lightweight Authentication Encryption Mode (v2. 1). 2016. https://competitions.cr.yp.to/round3/jambuv21.pdf.

[26] Jérémy Jean, I Nikolić, and Thomas Peyrin. Joltik v1. 3. *CAESAR Round*, 2, 2015. https://competitions.cr.yp.to/round2/joltikv13.pdf.

[27] G Bertoni, J Daemen, M Peeters, G Van Assche, and R Van Keer. CAESAR submission: Ketje v1. CAESAR First Round Submission, March 2014, 2016. https://competitions.cr.yp.to/round3/ketjev2.pdf.

[28] Yu Sasaki, Yosuke Todo, Kazumaro Aoki, Yusuke Naito, Takeshi Sugawara, Yumiko Murakami, Mitsuru Matsui, and Shoichi Hirose. Minalpher v1. *CAESAR Round*, 1, 2014. https://competitions.cr.yp.to/round2/minalpherv11.pdf.

[29] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. NORX8 and NORX16: Authenticated Encryption for Low-End Systems. Cryptology ePrint Archive, Report 2015/1154, 2015. http://eprint.iacr.org/2015/1154.

[30] Elena Andreeva, Begül Bilgin, Andrey Bogdanov, Atul Luykx, F Mendel, B Mennink, N Mouha, Q Wang, and K Yasuda. PRIMATEs v1. 02 submission to the CAESAR competition, 2014. https://competitions.cr.yp.to/round2/primatesv102.pdf.

[31] Shai Halevi, Don Coppersmith, and Charanjit S. Jutla. Scream: A Software-Efficient Stream Cipher. In Joan Daemen and Vincent Rijmen, editors, *Fast Software Encryption – FSE 2002*, volume 2365 of *Lecture Notes in Computer Science*, pages 195–209, Leuven, Belgium, February 4–6, 2002. Springer, Heidelberg, Germany. `https://eprint.iacr.org/2002/019.pdf`.

[32] Ivica Nikolić. Tiaoxin-346. *Submission to the CAESAR competition*, 2015. `competitions.cr.yp.to/round3/tiaoxinv21.pdf`.

[33] A Chakraborti and M Nandi. TriviA-ck-v2, 2015. `https://competitions.cr.yp.to/round2/triviackv2.pdf`.

[34] Vincent Rijmen. Efficient Implementation of the Rijndael S-box. 2000.

[35] P. V. S. Shastry, A. Agnihotri, D. Kachhwaha, J. Singh, and M. S. Sutaone. A combinational logic implementation of S-box of AES. In *2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1–4, Aug 2011.

[36] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In Pil Joong Lee, editor, *Advances in Cryptology – ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 16–31, Jeju Island, Korea, December 5–9, 2004. Springer, Heidelberg, Germany.

Apéndice A

Vectores de prueba

Código A.1: Vectores de prueba AES128.

```
1  K:  00000000000000000000000000000000
2  M:  00000000000000000000000000000000
3  C:  66e94bd4ef8a2c3b884cfa59ca342b2e
4
5  K:  2b7e151628aed2a6abf7158809cf4f3c
6  M:  6bc1bee22e409f96e93d7e117393172a
7  C:  3ad77bb40d7a3660a89ecaf32466ef97
8
9  K:  6abf7158809cf4f3c2b7e151628aed2a
10 M:  13198a2e03707343243f6a8885a308d3
11 C:  5ce3c920fe13da7d7be5b89840aaacd0
```

Código A.2: Vectores de prueba Midori64.

```
1  K:  00000000000000000000000000000000
2  M:  0000000000000000
3  C:  3c9cceda2bbd449a
4
5  K:  687ded3b3c85b3f35b1009863e2a8cbf
6  M:  0123456789abcdef
7  C:  c0d5183edc908fd2
8
9  K:  85b3f35b1009863e2a8cbf687ded3b3c
10 M:  5b1009863e2a8cbf
11 C:  86774f649915ed85
```

Código A.3: Vectores de prueba COFB-AES128.

```
K:  00000000000000000000000000000000
N:  0000000000000000
D:  00000000000000000000000000000000
M:  00000000000000000000000000000000
C:  4b595bfc5ff8cd704b0643d96b3d7a3b
T:  22837bdb243ad9b6d003b58a461da502

K:  00000000000000000000000000000000
N:  0000000000000000
D:  000000000000000000000000000000000000000000000000000000000000000000000000
M:  000000000000000000000000000000000000000000000000000000000000000000000000
C:  78844173f71efe093ebe1142590041af98c3f2cd06aa002fb1d7a21c68f755c5
T:  0dc9297d5fa82d9776084dd8c62beada

K:  2b7e151628aed2a6abf7158809cf4f3c
N:  abf7158809cf4f3c
D:  e93d7e117393172a6bc1bee22e409f96abf7158809cf4f3c2b7e151628aed2a6
M:  2b7e151628aed2a6abf7158809cf4f3ca6bc1bee22e409f96abf7158809cf4f3
C:  d5cb9804bc03e4e97ff5ec50551f8723945ce736f109d10c8550e2c385b0d243
T:  e4d68d6f53a9a09ba1a5b49bee74e48e
```

Código A.4: Vectores de prueba COFB-Midori64.

```
K:  00000000000000000000000000000000
N:  00000000
D:  0000000000000000
M:  0000000000000000
C:  cbd9e519dd0858f7
T:  698515ea663d7a10

K:  00000000000000000000000000000000
N:  00000000
D:  00000000000000000000000000000000
M:  00000000000000000000000000000000
C:  f2e5f8ae1c38c987a2f7220a2862a3c9
T:  808f5a435e35d9a4

K:  00000000000000000000000000000000
N:  00000000
D:  000000000000000000000000000000000000000000000000
M:  000000000000000000000000000000000000000000000000
C:  45f2162c86eaf911ce63d351e90a9296247e9b5856adb90e
T:  9e95f93d93943a07
```

Donde:

- K: llave privada.
- D: datos asociados.
- M: mensaje en claro.
- C: mensaje cifrado.
- N: número de un solo uso (*nounce*).
- T: verificador (*Tag*).

Apéndice B

Códigos fuente

En seguida se listan los enlaces a los códigos fuente desarrollados para el presente trabajo de tesis.

B.1. Implementaciones en software.

B.1.1. AES128 en lenguaje C:

`http://computacion.cs.cinvestav.mx/~mrodriguez/master/thesis/software/AES128/`

B.1.2. Midori64 en lenguaje C:

`http://computacion.cs.cinvestav.mx/~mrodriguez/master/thesis/software/Midori64/`

B.1.3. COFB-AES128 en lenguaje C:

`http://computacion.cs.cinvestav.mx/~mrodriguez/master/thesis/software/COFB-AES128/`

B.1.4. COFB-Midori64 en lenguaje C:

`http://computacion.cs.cinvestav.mx/~mrodriguez/master/thesis/software/COFB-Midori64/`

B.2. Implementaciones en hardware.

B.2.1. AES128 en lenguaje VHDL:

http://computacion.cs.cinvestav.mx/~mrodriguez/master/thesis/hardware/AES128/

B.2.2. Midori64 en lenguaje VHDL:

http://computacion.cs.cinvestav.mx/~mrodriguez/master/thesis/hardware/Midori64/

B.2.3. COFB-AES128 en lenguaje VHDL:

http://computacion.cs.cinvestav.mx/~mrodriguez/master/thesis/hardware/COFB-AES128/

B.2.4. COFB-Midori64 en lenguaje VHDL:

http://computacion.cs.cinvestav.mx/~mrodriguez/master/thesis/hardware/COFB-Midori64/