IMPLEMENTACIÓN ASÍNCRONA DE LAS FUNCIONES MIXCOLUMN E INVMIXCOLUMN DEL ALGORITMO DE RIJNDAEL

ASYNCHRONOUS IMPLEMENTATION OF RIJNDAEL ALGORITHM´S MIXCOLUMN AND INVMIXCOLUMN FUNCTIONS

RUBÉN DARIO NIETO LONDOÑO

Escuela de Ingeniería Eléctrica y Electrónica, Facultad de Ingenierías, Universidad del Valle, rnieto@univalle.edu.co

ÁLVARO BERNAL NOREÑA

Escuela de Ingeniería Eléctrica y Electrónica, Facultad de Ingenierías, Universidad del Valle alvaro@univalle.edu.co

Recibido para revisar Diciembre 9 de 2008, aceptado Julio 17 de 2009, versión final Agosto 26 de 2009

RESUMEN: Este artículo presenta resultados de la implementación asíncrona en FPGA (*Field Programmable Gate Array*) de las funciones de transformación de columna *MixColumn* e *InvmixColumn*, del algoritmo de Rijndael. Se usa la metodología para implementación de circuitos asíncronos de la herramienta conocida como Balsa, la cual permite descripción, síntesis y simulación funcional asíncrona de circuitos digitales. Se presentan resultados comparativos de área y desempeño de dos arquitecturas que realizan procesos de encriptación y desencriptación, la primera hace uso de las funciones sin optimizar y la segunda realiza ambos procesos usando recursos de hardware compartidos por ambas funciones. En la implementación asíncrona de todas las transformaciones se usan los protocolos de codificación *dual-rail* y *1-de-4*.

PALABRAS CLAVE: AES, Balsa, circuitos asíncronos, criptografía, FPGAs, protocolo de sincronismo.

ABSTRACT: This article presents the results of an asynchronous implementation of Rijndael Algorithm *MixColumn* and *InvMixColumn* transformation functions in *FPGA* (*Field Programmable Gate Array*). The methodology used for implementing asynchronous circuits is provided by the tool known as Balsa, which allows description, synthesis, and asynchronous functional simulation of digital circuits. Results are presented by comparing area and performance of two different architectures, which execute both encryption and decryption processes. The first architecture deploys the functions without optimization, whereas the second one executes encryption and decryption processes utilizing shared hardware resources for the two functions. The codification protocols, *dual-rail and 1-of-4*, are implemented in all asynchronous transformations.

KEYWORDS: AES, Balsa, asynchronous circuits, cryptography, FPGAs, handshake.

1. INTRODUCCIÓN

En enero de 1997, el Instituto Nacional de Estándares y Tecnología de Estados Unidos, *NIST* (*National Institute of Standards and Technology*) [1], realizó una convocatoria con el fin de establecer un nuevo algoritmo para el estándar AES (*Advanced Encryption Standard*) del *FIPS* (*Federal Information Processing Standard*) [2], y reemplazar el antiguo estándar de encriptación *DES* (*Data Encrytion Standard*). En octubre del 2000 el NIST, después de dos rondas de evaluación a 15 algoritmos candidatos, seleccionó el algoritmo de Rijndael como el nuevo estándar AES.

Son muchas las arquitecturas de circuito propuestas para el algoritmo AES; se ha evaluado su desempeño usando librerías de Circuitos Integrados.

De Aplicación Específica (*ASICs: Application Specific Integrated Circuits*) y *FPGAs* (*Field Programmable Gate Arrays*). La mayoría de las propuestas se han focalizado en obtener ahorro de área y en obtener mayor velocidad, algunas están dirigidas a reducir el consumo de potencia. Desde el punto de vista de seguridad, con el desarrollo de los ataques del lado de canal [3], han sido pocas las propuestas de arquitecturas orientadas a mejorar la resistencia a tales ataques y a la vez disminuir el consumo de potencia. La metodología de diseño asíncrono constituye una alternativa importante para mejorar la seguridad del algoritmo contra los llamados ataques de Análisis de Potencia Diferencial (*DPA: Differential Power Analysis*) [4].

Muchas de las implementaciones del algoritmo de Rijndael realizan de manera separada las transformaciones *MixColumn* e *InvMixColumn* y sólo en algunas publicaciones, [5]-[9], se presentan resultados sobre implementaciones en las que estas funciones comparten recursos lógicos a diferentes niveles. Este trabajo tiene como objetivo presentar los resultados de la implementación asíncrona de las funciones *MixColumn* e *InvMixColumn* del estándar AES. Las implementaciones de las funciones se realizaron de manera independiente y también compartiendo recursos, para este caso se implementó la función *EDMixColumn*. En los diseños se utilizó el estilo de diseño asíncrono usando para ello la herramienta de síntesis asíncrona conocida como Balsa.

La sección dos hace una introducción general del algoritmo de Rijndael y presenta la herramienta *Balsa*. En la sección tres se realiza un análisis detallado de la función de transformación de columnas para los procesos de encriptación/desencriptación del estándar AES; también se presentan distintas arquitecturas síncronas simplificadas para la implementación de la función *EDMixColumn,* en particular se hace énfasis en la arquitectura propuesta en [6] para su implementación asíncrona. También se presentan los resultados de la síntesis asíncrona de los circuitos en el sistema Balsa y su implementación en la FPGA Virtex XCV2P30-6ff896 de Xilinx. Finalmente, en la sección cuatro se consignan las conclusiones del trabajo.

2. ALGORITMO DE RIJNDAEL Y HERRAMIENTA DE DISEÑO ASÍNCRONO

El algoritmo de encriptación de Rijndael está compuesto por cuatro funciones de transformación llamadas *ByteSub* (substitución de byte)*, ShiftRow* (desplazamiento circular de fila)*, MixColumn* (transformación de columna) *y AddRoundKey* (adición de clave de ronda). El proceso de desencriptación consiste en sustituir las transformaciones utilizadas en la encriptación por sus inversas e invertir el orden de aplicación de dichas transformaciones o funciones matemáticas, en este caso las funciones inversas se denominan *InvByteSub, InvShiftRow, InvMixColumn*. El diagrama de bloques del algoritmo de Rijndael, mostrado en la figura 1, combina las propuestas realizadas en [10],[11] para su implementación en hardware. Una descripción detallada de estas transformaciones y del algoritmo en general es presentada por los autores del mismo en [12].

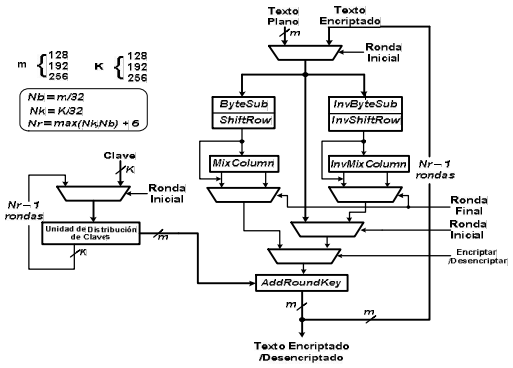

Figura 1. Diagrama de bloques de AES
Figure 1. AES Block diagram

En la implementación asíncrona de las funciones de transformación de columna del algoritmo de Rijndael se ha utilizado la herramienta de diseño llamada *Balsa*. Ésta cubre varias etapas del ciclo de diseño asíncrono y provee medios asíncronos de descripción y modelamiento junto con un medio de simulación funcional [13],[14]. Sin embargo, Balsa es primordialmente un sistema de

síntesis, el cual requiere acceder a herramientas CAD (*Computer Ayded Design*) comerciales para producir una implementación real en silicio o en arreglos de compuertas. Para este propósito, Balsa sólo produce un archivo de tipo lista de nodos (*netlist)* en formato apropiado para un sistema CAD que soporte la tecnología.

Un circuito descrito en Balsa se puede compilar en una red de comunicación de sincronismo (*handshake*) compuesta de un conjunto de componentes específicos de la herramienta. Los componentes se conectan a través de canales sobre los cuales toma lugar la comunicación; los canales pueden tener asociados rutas de datos o pueden ser sólo de control.

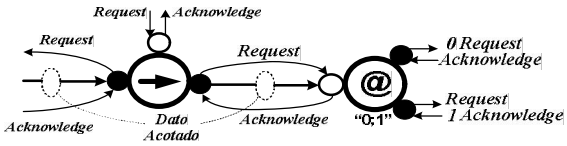

Figura 2. Conexión de componentes de *handshake*, Ref. [13],[14]
Figure 2. Connected *handshake* components, Ref. [13],[14]

La figura 2 ilustra un ejemplo de un diagrama de circuito conformado por componentes de *handshake*. Aquí se muestra un componente *fetch* (u operador de transferencia) denotado por '→' y un componente *Case* denotado por '@' conectados por medio de un canal orientado a datos. La acción del circuito es activada por un evento de solicitud (*request*) al componente *fetch* el cual a su vez establece un *request* a su entorno (al lado izquierdo del diagrama). El entorno provee el dato solicitado e indica su validez con una señal de reconocimiento (*acknowledge*). El componente *fetch* presenta el dato y una señal de *request* al componente *Case* usando un puerto activo (mostrado como un círculo negro), estos son recibidos en el puerto pasivo de *Case* (mostrado como círculo en blanco). Dependiendo del valor del dato, el componente *Case* establece un *handshake* en su entorno sobre el puerto inferior y superior del lado derecho. Finalmente, la señal de *acknowledge* es recibida por el componente *Case*, se retorna otro *acknowledge* a lo largo del canal original terminando así el *handshake*. Los protocolos de *handshake* utilizados en la comunicación entre módulos hacen que los circuitos asíncronos tengan la propiedad de ser insensibles a los retardos. Gracias a ello, han aparecido metodologías de diseño asíncrono como la que utiliza circuitos *Casi Insensibles a los Retardos, QDI* (*Quasy Delay Insensitive*). Esta metodología permite disminuir el consumo de potencia dinámica y hace que la implementación sea resistente a los ataques del lado de canal como por ejemplo el análisis de potencia diferencial *DPA* (*Differential Power Analisys*), tal como se reporta en [4],[15]. Los circuitos QDI controlan en forma precisa el número de transiciones eléctricas involucradas en una operación particular logrando, de esta manera, mejorar la seguridad del hardware contra los ataques *DPA*.

3. TRANSFORMACIONES DE COLUMNA

3.1 Transformación *MixColumn*

Esta transformación opera columna por columna sobre el bloque de datos de entrada (llamado matriz de estado), tratando cada columna como un polinomio de cuatro términos. Las columnas son consideradas como polinomios sobre $GF(2^8)$ (*Galois Fields*) y multiplicados módulo x^4+1 con un polinomio fijo $c(x)$, dado por:

$$c(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \quad (1)$$

La anterior ecuación se puede escribir como la multiplicación de matrices siguiente:

$$S = c(x) \otimes S(x) \quad (2)$$

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \quad (3)$$

para $0 \le c < Nb$

Los bytes de cada columna se pueden reemplazar por el resultado obtenido de la multiplicación de acuerdo con las siguientes expresiones:

$$\begin{aligned} S'_{0,c} &= (\{02\} \bullet S_{0,c}) \oplus (\{03\} \bullet S_{1,c}) \oplus S_{2,c} \oplus S_{3,c} \\ S'_{1,c} &= S_{0,c} \oplus (\{02\} \bullet S_{1,c}) \oplus (\{03\} \bullet S_{2,c}) \oplus S_{3,c} \\ S'_{2,c} &= S_{0,c} \oplus S_{1,c} \oplus (\{02\} \bullet S_{2,c}) \oplus (\{03\} \bullet S_{3,c}) \\ S'_{3,c} &= (\{03\} \bullet S_{0,c}) \oplus S_{1,c} \oplus S_{2,c} \oplus (\{02\} \bullet S_{3,c}) \end{aligned} \quad (4)$$

Las ecuaciones (4) muestran multiplicaciones de polinomios por potencias de x en $GF(2^8)$. Estas operaciones corresponden a multiplicaciones

módulo un polinomio binario irreducible de grado 8 llamado *m(x)* [12] dado por:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \qquad (5)$$

Para facilitar la implementación de la transformación *MixColumn* en hardware, los autores del algoritmo definieron la función llamada *xtime*, la cual simplifica la operación de un polinomio por potencias de *x*. Esta función consiste en aplicar un desplazamiento a la izquierda al valor que representa el polinomio además de una operación XOR con $1B_{16}$ (00011011_2) cuando el resultado de la multiplicación deba ser reducido módulo *m(x)*. Sea el polinomio definido por:

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

Al realizar la operación modular de (6) con *m(x)* se obtiene una expresión que permite implementar el circuito para *xtime* a nivel de bits de la siguiente manera:

$$xtime(b) = b(x) \bullet x \bmod m(x)$$
$$xtime(b) = b_7 x^8 + b_6 x^7 + b_5 x^6 + b_4 x^5 + b_3 x^4 + b_2 x^3 + b_1 x^2 + b_0 x \bmod x^8 + x^4 + x^3 + x + 1 \qquad (7)$$

Entonces

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

Si hacemos

$$xtime(b) = b'(x) = b'_7 x^7 + b'_6 x^6 + b'_5 x^5 + b'_4 x^4 + b'_3 x^3 + b'_2 x^2 + b'_1 x + b'_0 \qquad (9)$$

De (8) y (9) se deduce que:

$$b'_7 = b_6, \quad b'_6 = b_5, \quad b'_5 = b_4, \quad b'_4 = b_7 + b_3, \quad b'_3 = b_7 + b_2, \quad b'_2 = b_1, \quad b'_1 = b_7 + b_0, \quad b'_0 = b_7 \qquad (10)$$

La figura 3 representa el circuito combinatorio de la función *xtime* a nivel de bits implementado de acuerdo con los términos deducidos.

Se pueden realizar multiplicaciones por potencias mayores de *x* mediante la aplicación repetida de *xtime*.

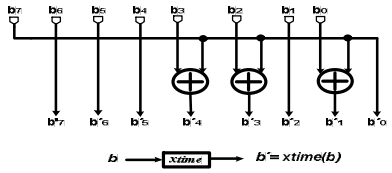

Figura 3. Circuito de función *xtime*, Ref. [8],[16]
Figure 3. *Xtime* function circuit, Ref. [8],[16]

La figura 4 muestra el código para la especificación asíncrona del circuito que implementa la función *xtime* en Balsa. Al realizar la síntesis del código se genera el circuito de *handshake* mostrado en la figura 5. Los diagramas de *handshake* generados son muy densos en detalles y de difícil comprensión, sin embargo se han incluido en este artículo para ofrecer al lector una idea general de su función.

```
import [balsa.types.basic]
procedure xtime1 (input i: byte; output o : byte) is
        variable u : byte
        variable x,y : array 0 .. 7 of bit
begin
        i -> u;  x := #u;
y[7] := x[6] ||  y[6] := x[5] ||  y[5] := x[4] ||
y[4] := x[3] xor x[7] ||  y[3] := x[2] xor x[7] ||
y[2] := x[1] ||  y[1] := x[0] xor x[7]  || y[0] := x[7]  ;
    o <-  (y as byte)
end
```

Figura 4. Código de función *xtime*
Figure 4. *Xtime* function code

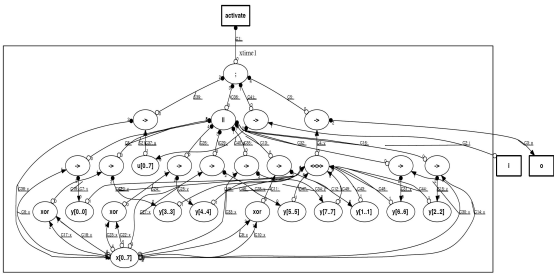

Figura 5. *Handshake* de función *xtime*
Figure 5. *Xtime* function *handshake*

Las ecuaciones (4) muestran multiplicaciones de polinomios por potencias de *x* (representadas por los coeficientes 02_{16} y 03_{16}). Como en $GF(2^8)$ la multiplicación es distributiva respecto de la adición, se puede reescribir la multiplicación de dos elementos como una combinación lineal de productos, por ejemplo:

$$x \bullet \{03\} = x \bullet (\{02\} \oplus \{01\}) = x \bullet \{02\} \oplus x$$
$$= xtime(x) \oplus x$$

Para cualquier $x \in GF(2^8)$.
La anterior distribución permite implementar fácilmente el circuito que realiza la transformación *MixColumn* a partir de (4) tal como se aprecia en la figura 6. Las figuras 7 y 8 muestran, respectivamente, el código Balsa y el diagrama de *handshake* correspondiente.

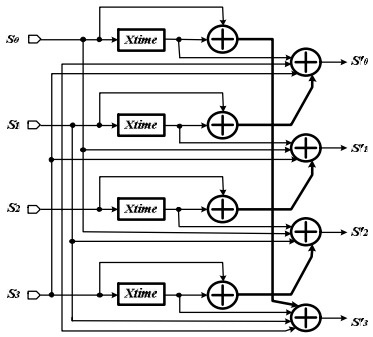

Figura 6. Circuito para la transformación *MixColumn*, Ref. [10].
Figure 6. Circuit for *MixColumn* transformation, Ref. [10].

```
-- Programa Balsa Transformación MixColumn
import [balsa.types.basic]
import [Xtime]

procedure mixcolumn4 (input i3,i2,i1,i0: byte ; output o3,o2,o1,o0:
byte) is
variable x3,x2,x1,x0 : byte
variable y3,y2,y1,y0 : byte
variable z3,z2,z1,z0 : byte
channel  ch3,ch2,ch1,ch0 : byte

begin
                i3 -> x3 ||  i2 -> x2 ||  i1 -> x1 ||  i0 -> x0 ||

                xtime1(i3,-> y3)   || xtime1(i2,-> y2)   ||
                xtime1(i1,-> y1)   || xtime1(i0,-> y0)   ;

                z3 := x3 xor y3  || z2 := x2 xor y2    ||
                z1 := x1 xor y1  || z0 := x0 xor y0    ;

                o3 <- x0 xor x1 xor y3 xor z2   ||
                o2 <- x0 xor x3 xor y2 xor z1   ||
                o1 <- x3 xor x2 xor y1 xor z0   ||
                o0 <- x1 xor x2 xor y0 xor z3
end
```

Figura 7. Código de *MixColumn*
Figure 7. *MixColumn* Code

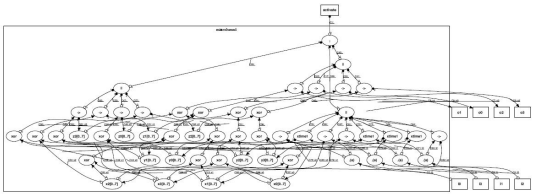

Figure 8. *Handshake* de *MixColumn*
Figure 8. *MixColumn handshake*

3.2 Transformación *InvMixColumn*

InvMixColumn es la inversa de la transformación *MixColumn*, y opera sobre la matriz de estado columna por columna, tratando cada columna como un polinomio de cuatro términos. Las columnas son consideradas como polinomios sobre $GF(2^8)$ multiplicados módulo $x+1$ con un polinomio fijo $c^{-1}(x)$, en adelante llamado $d(x)$, dado por:

$$d(x) = c^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\} \quad (11)$$

La ecuación (11) se puede escribir como una multiplicación de matrices de la siguiente forma:

$$S' = c^{-1}(x) \otimes S(x) \quad (12)$$

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \quad \text{para} \quad 0 \le c < Nb \quad (13)$$

Al resolver la multiplicación anterior se obtienen las siguientes expresiones para reemplazar cada byte de la columna c [13]:

$$\begin{aligned} S'_{0,c} &= (\{0e\} \bullet S_{0,c}) \oplus (\{0b\} \bullet S_{1,c}) \oplus (\{0d\} \bullet S_{2,c}) \oplus (\{09\} \bullet S_{3,c}) \\ S'_{1,c} &= (\{09\} \bullet S_{0,c}) \oplus (\{0e\} \bullet S_{1,c}) \oplus (\{0b\} \bullet S_{2,c}) \oplus (\{0d\} \bullet S_{3,c}) \\ S'_{2,c} &= (\{0d\} \bullet S_{0,c}) \oplus (\{09\} \bullet S_{1,c}) \oplus (\{0e\} \bullet S_{2,c}) \oplus (\{0b\} \bullet S_{3,c}) \\ S'_{3,c} &= (\{0b\} \bullet S_{0,c}) \oplus (\{0d\} \bullet S_{1,c}) \oplus (\{09\} \bullet S_{2,c}) \oplus (\{0e\} \bullet S_{3,c}) \end{aligned} \quad (14)$$

Nuevamente, basados en el hecho que en $GF(2^8)$ la multiplicación es distributiva respecto de la adición, el coeficiente $0b_{16}$ multiplicado por x, por ejemplo, se puede descomponer así:

$$\begin{aligned} x \bullet \{0b\} &= x \bullet (\{08\} \oplus \{02\} \oplus \{01\}) \\ &= x \bullet \{08\}\, x \bullet \{02\} \oplus x \bullet \{01\} \end{aligned}$$

para cualquier $x \in GF(2^8)$.

Teniendo en cuenta el ejemplo anterior, los coeficientes $09_{16}, 0b_{16}, 0d_{16}, 0e_{16}$ en (14) se pueden expresar como sumas de elementos en potencias de dos, lo cual facilita la implementación del circuito que genera tales coeficientes a partir de la apropiada interconexión de circuitos *xtime*. El circuito mostrado en la figura 9 es un

multiplicador que genera los coeficientes requeridos y constituye la base de la implementación del circuito para *InvMixColumn*. La especificación en Balsa del multiplicador y el diagrama de *handshake* resultante se observan en las figuras 10 y 11 respectivamente.

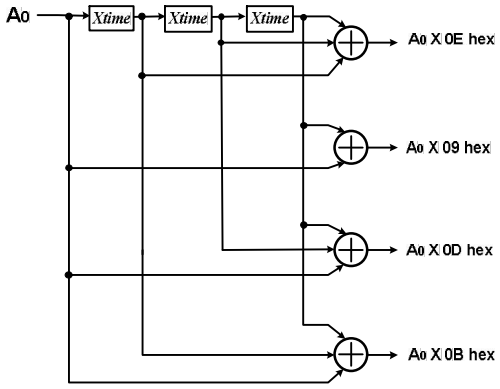

Figura 9. Multiplicador para *InvMixColumn*. Ref. [10]
Figure 9. Multiplier for *InvMixColumn*. Ref. [10]

```
import [balsa.types.basic]
import [Xtime]
procedure mult_GF(input i: byte ; output  o3,o2,o1,o0:
byte) is
variable z3,z2,z1,z0 : byte
channel  ch2,ch1,ch0 : byte
begin
i -> z3 ||
xtime1 (i,->z2) ||
xtime1(i,ch2) || xtime1 (ch2,->z1) ||
xtime1(i,ch1) || xtime1 (ch1,ch0) || xtime1 (ch0,->z0);
o3 <- z2 xor z1 xor z0 ||
o2 <- z3 xor z0        ||
o1 <- z3 xor z1 xor z0 ||
o0 <- z3 xor z2 xor z0
end
```

Figura 10. Código del multiplicador
Figure 10. Multiplier code

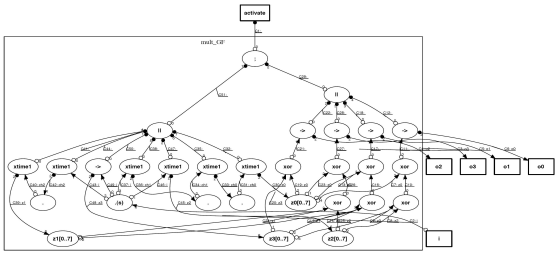

Figura 11. *Handshake* de multiplicador
Figure 11. Multiplier *handshake*

Como la transformación *InvMixColumn* opera sobre una columna a la vez, se requieren cuatro circuitos multiplicadores para realizar la transformación en paralelo de los cuatro bytes de la columna. Otra opción de diseño para InvMixColumn consiste en implementarla de manera secuencial pero, como se verá en la sección 3.3, a pesar que tal opción permite el ahorro de puertas la velocidad será menor que la implementación en paralelo.

El código en Balsa para *InvMixColumn*, mostrado en la figura 12, hace uso de cuatro multiplicadores (*mult_GF*), cuyas salidas se conectan como entradas a las diferentes puertas XOR de salida, tal como lo expresan las ecuaciones en (14). La figura 13 muestra el diagrama de *handshake* correspondiente. En este diagrama se observa al operador paralelo '||' conectando los cuatro procesos asociados con los multiplicadores y sus conexiones de *handshake* con las variables internas. Las variables a su vez indican las conexiones hacia las puertas XOR y las salidas respectivas.

```
import [balsa.types.basic]
import [Multiplicador_GF]

procedure invmixcolumn4 (input i3,i2,i1,i0: byte ; output
o3,o2,o1,o0: byte) is
variable u15,u14,u13,u12,u11,u10,u9,u8,u7,u6,u5, u4, u3,
u2,u1,u0 : byte
begin
mult_GF(i3,->u15,->u14,->u13,->u12)  ||
mult_GF(i2,->u11,->u10,->u9,->u8)  ||
mult_GF(i1,->u7,->u6,->u5,->u4)  ||
mult_GF(i0,->u3,->u2,->u1,->u0)  ;
o3 <- u15 xor u8 xor u5 xor u2  ||
o2 <- u14 xor u11 xor u4 xor u1 ||
o1 <- u13 xor u10 xor u7 xor u0 ||
o0 <- u12 xor u9 xor u6 xor u3
end
```

Figura 12. Código de *InvMixColumn*
Figure 12. *InvMixColumn* Code

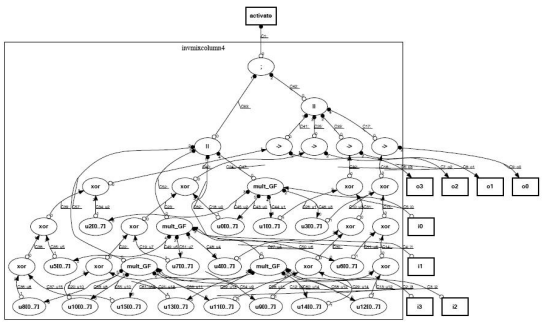

Figura 13. *Handshake* de *InvMixColumn*
Figure 13. *InvMixColumn handshake*

El circuito para realizar la transformación *InvMixColumn* (mostrado en la figura 14) se deduce de la ecuación (14) teniendo como base las salidas del multiplicador.

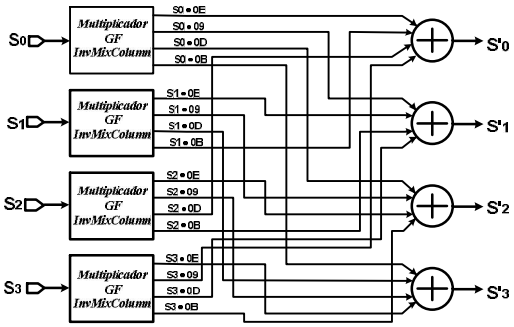

Figura 14. Circuito para transformación *InvMixColumn*. Ref. [10]
Figure 14. Circuit for *InvMixColumn* transformation. Ref. [10]

3.3 Transformación de columna para encriptación/desencriptación

De las ecuaciones (4) y (14) se puede ver que los coeficientes de *d(x)* son más complejos que los de *c(x),* por lo que la implementación hardware de la transformación *InvMixColumn* es más compleja, más lenta y ocupa mayor área que la transformación *MixColumn*. Al Implementar un circuito para encriptar/desencriptar, como se ilustra en la figura 15, el costo en hardware es alto si se utilizan ambas transformaciones por separado.

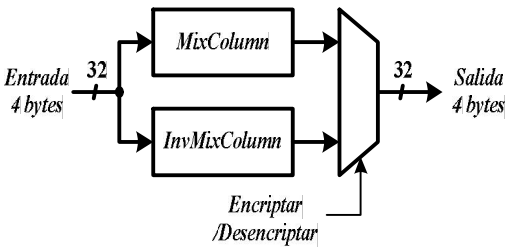

Figura 15. Circuito para transformación de columna sin optimización.
Figure 15. Circuit for column transformation without optimization

Para reducir el costo del hardware, la transformación *InvMixColumn* se puede descomponer en forma serial para que comparta recursos lógicos con *MixColumn* a nivel de palabras de 32 bits. Esta idea se ha planteado en varias publicaciones [6],[8],[9].

En [6] se analiza, además, una descomposición paralela de *InvMixColumn*.

3.3.1 Descomposición paralela de InvMixColumn, EDMixColumn_paralelo

Esta idea, propuesta por [17], se basa en el hecho que *d(x)* se puede expresar usando *c(x)* de la siguiente manera:

$$d(x) = c(x) + e(x) \qquad (15)$$

Donde *e(x)* es una extensión polinomial definida como:

$$e(x) = \{08\}x^3 + \{0c\}x^2 + \{08\}x + \{0c\} \qquad (16)$$

3.3.2. *Descomposición serial de InvMixColumn, EDMixColumn_serie*

Propuesta en [6], esta idea se sustenta en el hecho que el inverso *d(x)* está dado por la fórmula:

$$d(x) = c^{-1}(x) = c^3(x) \qquad (17)$$

Esta ecuación sugiere que la operación *InvMixColumn* se puede realizar repitiendo tres veces *MixColumn*. La ecuación (17) se puede expresar como:

$$d(x) = c(x) + f(x) \qquad (18)$$

Donde $f(x) = c^2(x) = \{04\}x^2 + \{05\}$ (19)

De esta ecuación se deduce que la transformación *InvMixColumn* puede implementarse usando la función *MixColumn* y el polinomio *f(x).* Al comparar *f(x)* con *c(x),* dada en (1), se puede ver que *f(x)* es más simple por cuanto dos de sus coeficientes son cero. La ecuación (19) también se obtuvo en [18] de la siguiente forma:
Ya que

$$c(x) \bullet d(x) = \{01\} \qquad (20)$$

Si se multiplica a ambos lados de (20) por *d(x)* se obtiene:

$$c(x) \bullet d^2(x) = d(x) \qquad (21)$$

Donde

$$d(x) = \{04\}x^2 + \{05\} \qquad (22)$$

La ecuación (22) se puede implementar en hardware como se muestra en la figura 16.

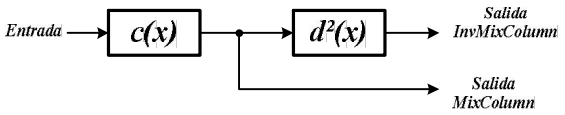

Figura 16. Implementación de *MixColumn* e *InvMixColumn*. Ref [18]
Figure 16. *MixColumn* and *InvMixColumn* Implementation. Ref [18]

Los circuitos para descomposición serial y descomposición paralela de *InvMixColumn* [6], se aprecian en las figuras 17 y 18 respectivamente, en ambos casos se observa la implementación de la función *MixColumn* encerrada en líneas punteadas.

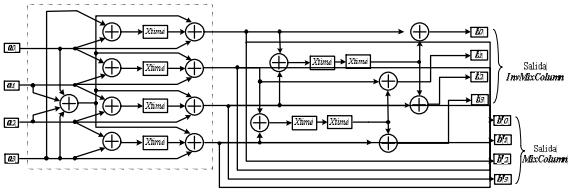

Figura 17. *EDMixColumn* basada en descomposición serial de *InvMixColumn*, Ref. [6].
Figure 17. *EDMixColumn* based on *InvMixColumn* serial decomposition. Ref. [6].

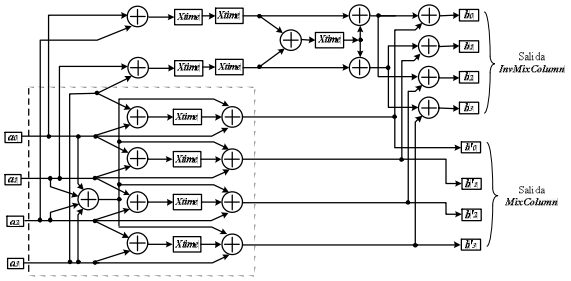

Figura 18. *EDMixColumn* basada en descomposición paralela de *InvMixColumn*, Ref. [6].
Figure 18. *EDMixColumn* based on *InvMixColumn* parallel decomposition. Ref. [6].

3.4 Transformación *EDMixColumn* asíncrona

La figura 19 ilustra la arquitectura general de la transformación *EDMixColumn* asíncrona utilizando protocolos de codificación *dual-rail* ó *1-de-4*. Esta arquitectura se basa en las transformaciones serial y paralela descritas anteriormente.

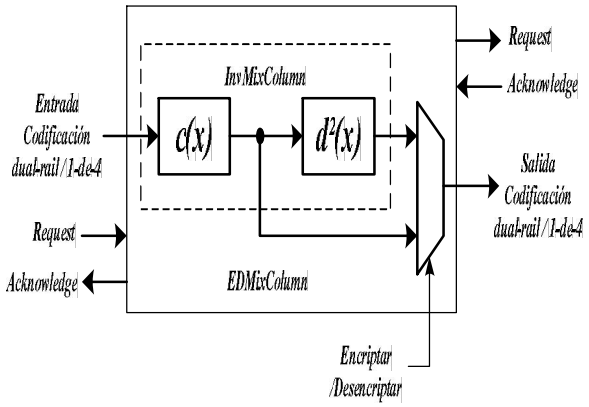

Figura 19. Arquitectura de *EDMixColumn*
Figure 19. *EDMixColumn* architecture

Los diagramas de *handshake* generados en Balsa para *EDMixColumn_serie* y *EDMixColumn_paralelo,* son más densos en detalles que los diagramas mostrados en secciones anteriores, por lo cual se omiten en este trabajo. Sin embargo, para el lector interesado en realizar un análisis más detallado, estos diagramas junto con otros aspectos importantes pueden ser consultados en [13].

3.5 Resultados

La tabla 1 muestra los resultados obtenidos de las implementaciones asíncronas tanto para las arquitecturas tradicionales *MixColumn* e *InvMixColumn* de 128 bits, como para las implementaciones basadas en las arquitecturas serial y paralela desarrolladas en [6].

La utilidad *breeze-cost* de Balsa, se ha usado para estimar el área ocupada por cada circuito de *handshake* en los diseños realizados. Las unidades retornadas por *breeze-cost* son micrómetros (µm) lineales de celdas estándar sobre un viejo proceso 2LM CMOS de 1µm; la precisión de los valores es relativa al tamaño de las celdas usadas por *breeze-cost*, no a la precisión de estimación de *breeze-cost*. Las celdas primarias de esta librería básica presentan una grilla (*pitch*) de 45µm, adicionando un 50% de espacio de ruteo extra (el cual es típicamente requerido en los procesos 2LM) [19].

Tabla 1: Costo de área y retardo de simulación funcional para *MixColumn*, *InvMixColumn* y *EDMixColumn* asíncronas.
Table 1: Area cost and functional simulation delay for asynchronous *MixColumn*, *InvMixColumn* and *EDMixColumn*.

Transformación	*xor*	*Costo Área*	*Retardo (ns)*
MixColumn	*28*	*11746.25*	*17000*
InvMixColumn	*76*	*17195.00*	*35000*
EDMixColumn	*104*	*30715.5*	*Enc: 25900 Dec: 43900*
EDMixColumn_ser	*45*	*23722.75*	*Enc: 48600 Dec: 49400*
EDMixColumn_prl 1	*51*	*25721.5*	*Enc: 43400 Dec: 45000*

La implementación de la transformación *EDMixColumn* se hizo integrando las funciones *MixColumn* e *InvMixColumn* mediante un multiplexor y sin compartir recursos, por lo cual el número de puertas usadas, respecto de las transformaciones *EDMixColumn* serie y paralela, es mayor. Sin embargo, como se observa en la tabla 1, también presenta menor retardo en los procesos de encriptación/desencriptación.

Aunque las funciones *EDMixColumn_serie* y *EDMixColumn_paralelo* requieren de menos compuertas que la función *EDMixColumn*, éstas presentan mayores retardos debido a que las señales deben recorrer un número mayor de niveles de puertas. Además, debido a que la comunicación se realiza por medio de protocolos de *handshake*, la diferencia de retardos se hace más significativa.

Balsa soporta las tecnologías AMS de 350nm y SGS-ST de 180nm, las cuales generan formatos del tipo *'netlist'* para producir implementaciones en Silicio. Balsa también soporta la tecnología apropiada para producir formatos utilizados en FPGAs de Xilinx. Los resultados de la implementación en FPGA usando los protocolos asíncronos de codificación *dual-rail* de dos fases y *1-de-4*, se observan en la tabla 2.

Tabla 2: Resultados de implementación en Virtex XCV2P30-6ff896 de Xilinx
Table 2: Implementation results in Xilinx`s Virtex XCV2P30-6ff896

Transformación	*Retardo*	
	Dual-rail	*1-de-4*
MixColumn	***86.701ns*** *Lógica 37.029ns (42.7%) Rutas 49.672ns (57.3%)*	***107.992ns*** *Lógica 46.025ns (42.6%) Rutas 61.967ns (57.4%)*
InvMixColumn	***176.884ns*** *Lógica 73.650ns (41.6%) Rutas 103.233ns (58.4%)*	***217.393ns*** *Lógica 90.472ns (41.6%) Rutas 126.921ns (58.4%)*
EDMixColumn	***276.258ns*** *Lógica 114.260ns (41.4%) Rutas 161.999ns (58.6%)*	***284.885ns*** *Lógica 118.01ns (41.4%) Rutas 166.869ns (58.6%)*
EDMixColumn_ serie	***355.077ns*** *Lógica 145.952ns (41.1%) Rutas 209.125ns (58.9%)*	***415.565ns*** *Lógica 171.233ns (41.2%) Rutas 244.342ns (58.8%)*
EDMixColumn_ paralelo	***337.292ns*** *Lógica 138.754ns (41.1%) Rutas 139.781ns (58.9%)*	***382.711ns*** *Lógica 157.765ns (41.2%) Rutas 224.946ns (58.8%)*

Los retardos en ambos tipos de codificación se distribuyen entre las rutas de interconexión y las puertas lógicas que conforman cada circuito. Se puede observar que alrededor del 58% de los retardos en los circuitos son introducidos por las rutas de interconexión de la FPGA, lo cual refleja un enorme esfuerzo por parte de las herramientas de ruteo al tratar de equilibrar los tiempos de retardo sobre rutas de igual longitud y garantizar así la ausencia de riegos estáticos. Lo anterior es una evidencia de que los recursos de interconexión de las FPGAs comerciales están concebidos para aplicaciones que usan el tradicional estilo de diseño síncrono y no el estilo asíncrono.

En la literatura no hemos encontrado reportes de resultados de implementaciones asíncronas del estándar AES sobre *FPGAs*. Hasta el momento sólo se han reportado dos implementaciones asíncronas del algoritmo basadas en *ASICs* las cuales solo reportan resultados para el proceso de encriptación y cuyos retardos, como es de esperarse, son mucho menores, que los obtenidos en este trabajo:

en [4] se reporta una arquitectura QDI balanceada con codificación *1-de-N* y un retardo máximo de 850ns; en [15] se reporta una arquitectura *pipeline* con codificación *dual-rail* con un retardo máximo de 800ns. Estos reportes no mencionan resultados de rendimientos de bloques individuales del algoritmo.

4. CONCLUSIÓN

El trabajo realizado constituye el primer esfuerzo para lograr la implementación asíncrona de las funciones de transformación de columna del algoritmo de Rijndael usando para ello herramientas de síntesis automática de circuitos sobre FPGAs de Xilinx. A pesar de la buena integración ofrecida por el sistema Balsa con el paquete ISE de Xilinx para lograr la implementación circuitos asíncronos, dichas implementaciones presentaron retardos elevados debido a que el mayor porcentaje de retardo lo introducen las rutas de interconexión de las FPGAs comerciales utilizadas.

Es claro que, dadas las condiciones de avance de las herramientas de diseño asíncrono, parte del trabajo futuro debe orientarse en el corto plazo a estudiar estrategias que permitan adecuar eficientemente las plataformas comerciales a las metodologías asíncronas existentes.

REFERENCIAS

[1] NIST, National Institute of Standards and Technology. 2000. Disponible en: http://csrc.nist.gov/CryptoToolkit/aes/rijndael

[2] FIPS, Federal Information Processing Standards, Specification for the ADVANCED ENCRYPTION STANDARD (AES). Publication 197, November 26, 2001.

[3] KURLD and SCARD Consortium. Intermediate report, Side Channel Analysis Resistant Design Flow. SCARD, Information Society, Project Number: IST-2002-507270, 15 January 2005.

[4] BOUESSE, RENAUDIN, F. M. and GERMAIN, F. Asynchronous AES Crypto-Processor Including Secured and Optimized Blocks. TIMA Laboratory, Grenoble, France; SGDN/DCISS, Paris, France. Journal of Integrated Circuits and Systems, Vol 1, No 1, pp 5-13. March 2004.

[5] SATOH, A., MORIOKA, S., TAKANO, K., and MUNETOH, S. A Compact Rijndael Hardware Architecture with S-Box Optimization. Advances in Cryptology – ASIACRYPT 2001, LNCS Vol. 2248, pp. 239–254, 2001.

[6] FISCHER, V., DRUTAROVSKÝ, M., CHODOWIEC, P. and GRAMAIN, F. InvMixColumn Decomposition and Multilevel Resource Sharing in AES Implementations. *IEEE Transactions* on Very Large Scale Integration (VLSI) Systems, Vol. 13, No 8, August 2005.

[7] CHODOWIEC, P., and GAJ, K. Very Compact FPGA Implementation of the AES Algorithm. CHES 2003.

[8] HUANG, Y.H., LIN, Y.S., HUNG, K.Y., and LIN, K.C. Efficient Implementation of AES IP. *IEEE, Asia Pacific, Conference on Circuits and Systems APCCAS-2006*, pp 1418-1421. Dec 2006

[9] HSIAO, S.F., CHEN, M.C., and TU, C.H. Memory-Free Low-Cost Designs of Advanced Encryption Standard Using Common Subexpression Elimination for Subfunctions in Transformations. *IEEE Trans. on circuits and systems-I: Regular papers*, vol. 53, No. 3, pp. 615-626, March 2006.

[10] CORDERO, V., and SILVA, C.. Design of an ASIC CORE for data encryption and decryption Using the NIST Advanced Encryption standard. Microelectronics Group – Pontificia Universidad Católica del Perú. IX Workshop de Iberchip, La Habana, Cuba. 2003.

[11] VERBAUWHEDE, I., SCHAUMONT, P., and KUO, H. Design and performance testing of a 2.29-GB/s Rijndael processor. *IEEE Journal of Solid-State Circuits*, Volume: 38 Issue:3, March 2003.

[12] DAEMEN, J., and RIJMEN, V. AES Proposal: Rijndael. Proton Worl Int. I, Zweefvliegtuigstraat, Brussel, Belgium; Katholieke Universiteit Leuven, ESAT-COSIC, Belgium. Document V. 2. 1999.

[13] NIETO, R. Diseño e implementación de un cripto-procesador asíncrono de bajo consumo basado en el algoritmo de Rijndael [Tesis Doctoral]. Cali, Universidad del Valle, 2009.

[14] EDWARDS, D., BARDSLEY, L., Janin, L., and TOMS, W. Balsa: a Tutorial Guide. Version 3.5.The Advanced Processor Technologies Group, APT Group. Manchester University . Manchester, England. 2006.

[15] SHANG, D., et al. High-security asynchronous circuit implementation of AES. *IEE Proceedings* online no. 20050088, doi:10.1049/ip-cdt: 20050088. *IEE Proc.-Comput. Digit. Tech., Vol.* 153, No. 2, March 2006.

[16] Li, H., and FRIGGSTAD, Z. An Efficient Architecture for the AES MixColumns Operation. IEEE International Symposium on Circuits and Systems, 2005, ISCAS. Vol. 5, pp. 4637-4640. May 2005.

[17] WOLKERSTORFER, J. An ASIC implementation of the AES MixColumn operation. *Proceedings Austrochip 2001*, pp. 129-132. Vienna, Austria, Oct. 12, 2001,

[18] CHODOWIEC, P. and GAJ, K. Very Compact FPGA Implementation of the AES Algorithm. CHES 2003, LCNS 2779, pp. 319-333, *Springer-Verlag* Berlin Heidelberg 2003.

[19] BARDSLEY, A. Implementing Balsa handshake Circuits. [PhD Thesis], University of Manchester, 2000.