Implementación *hardware* del algoritmo Keccak para Hash-3 y comparación con Blake, Grøstl, JH y Skein

Hardware Implementation of Keccak Algorithm for Hash-3 and Comparison with Blake, Grøstl, JH, and Skein

Recibido: 02-10-2013 Aceptado: 22-10-2013

Melissa Ramírez[1]
César Augusto Pino[2]
Vladimir Trujillo Olaya[3]
Jaime Velasco Medina[4]

Resumen

Las funciones criptográficas Hash convierten un mensaje de longitud variable en un "resumen del mensaje", y son usadas para aplicaciones de firma digital, códigos de autenticación de mensajes y otras aplicaciones para seguridad informática. Recientes ataques realizados al estándar SHA-1 han sido publicados, por lo tanto, el *National Institute for Standards and Technology* (NIST) realizó una convocatoria pública para seleccionar el algoritmo SHA-3 para el nuevo estándar. Como resultado, entre los cinco algoritmos finalistas: Blake, Grøstl, JH, Keccak y Skein, el seleccionado fue Keccak. En este trabajo se implementaron en *hardware* los cinco algoritmos finalistas usando arquitecturas *pipeline* para las diferentes rondas del proceso iterativo. Las arquitecturas *hardware* fueron simuladas y sintetizadas usando *Modelsim-Altera* y *Quartus II* v.11.2, respectivamente. Los resultados de síntesis muestran que existen alternativas para ser usadas como IP *cores* en aplicaciones criptográficas embebidas en un SoC.

Palabras clave: Criptografía; NIST; algoritmos Hash; FPGA; SHA-3; firma digital.

Abstract

Cryptographic hash functions convert a variable-length message into a "message digest" and are used for digital signature applications, message authentication codes, and other information security applications. Recent attacks carried out on SHA-1 have been published; therefore, the National Institute for Standards and Technology (NIST) issued an open call to select the SHA-3 for the new standard. As a result, among the five finalists: Blake, Gröstl, JH, Keccak and Skein, Keccak was selected. In this paper, the five finalist algorithms were implemented in hardware using pipelined architectures for the different rounds of the iterative process. The hardware architectures were simulated and synthesized using ModelSim-Altera and Quartus II v.11.2, respectively. The synthesis results show that alternatives exist to be used as IP *cores* for cryptographic applications embedded in a SoC.

Keywords: Cryptography; NIST; Hash algorithms; FPGA; SHA-3; Digital Signature.

1 Colombiana, Ingeniera Electrónica, Grupo de Investigación Bionanoelectrónica, Universidad del Valle, Colombia. melissa160@gmail.com
2 Colombiano, Ingeniero Electrónico, Grupo de Investigación Bionanoelectrónica, Universidad del Valle, Colombia. cesarpino547@gmail.com
3 Colombiano, Ingeniero Electrónico, M. Sc(c) Ingeniería Electrónica, estudiante de doctorado Universidad del Valle, Grupo de Investigación Bionanoelectrónica, Universidad del Valle, Colombia. vladimir.trujillo@correounivalle.edu.co
4 Colombiano, Ph.D. Profesor titular, Universidad del Valle, director del Grupo de Investigación Bionanoelectrónica, Universidad del Valle, Colombia. jaime.velasco@correounivalle.edu.co

Introducción

Las funciones criptográficas Hash son utilizadas en los sistemas criptográficos, por ejemplo en las aplicaciones de firma digital, donde la implementación de un criptosistema puede incluir algoritmos criptográficos de clave pública y clave privada, tales como RSA, intercambio de claves Diffie-Hellman y curvas elípticas, lo cual es ilustrado en la Figura 1.

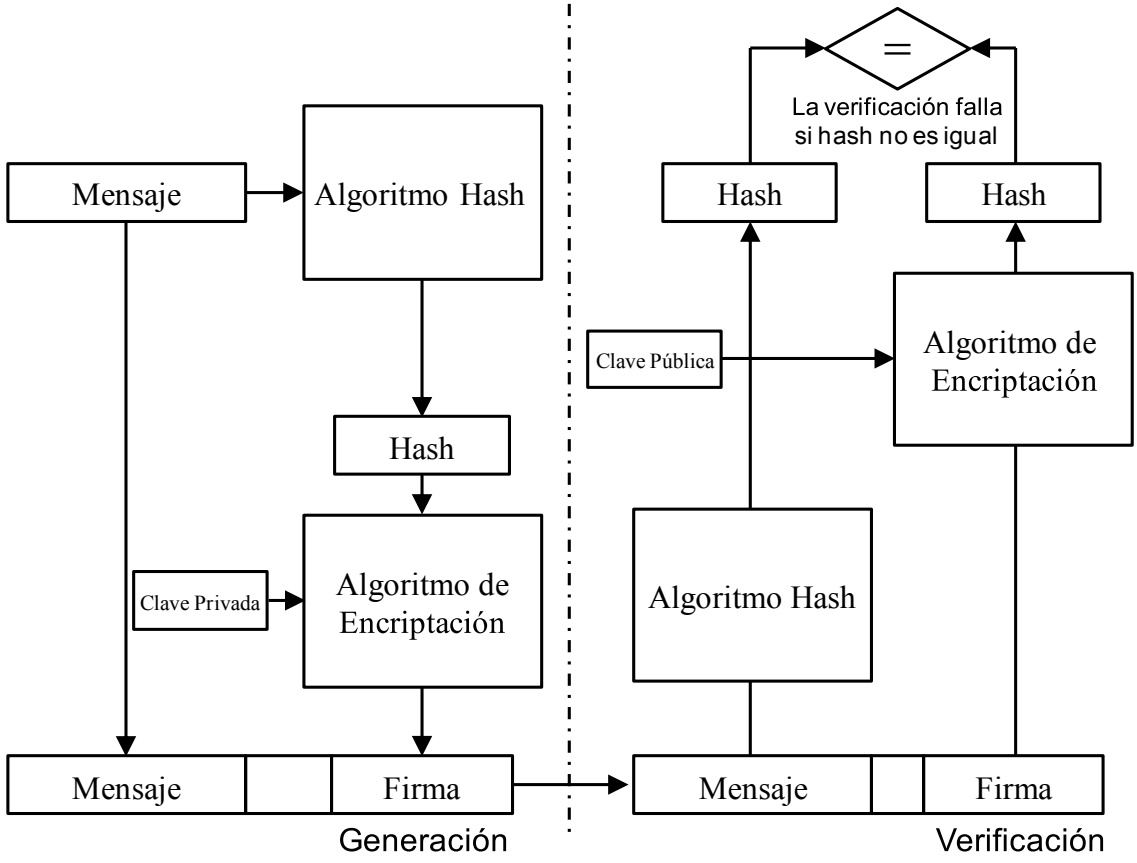

Figura 1. Generación y verificación de una firma digital

En la Figura 1 se observan las funciones criptográficas Hash usadas en la mayoría de las aplicaciones de seguridad informática, tales como autenticación, generación y verificación de firmas digitales. Las funciones se procesan en un bloque arbitrario de datos o mensajes, y retorna una cadena de bits de tamaño fijo llamada valor Hash o resumen del mensaje (*digest*), así, los algoritmos Hash son usados en el proceso de firma digital debido a que el valor Hash es por lo general más pequeño que el mensaje.

En este contexto, el estándar federal para el procesamiento de la información (FIPS 180-3: Federal Information Processing Standards, 2008), especifica cinco algoritmos criptográficos Hash ampliamente adoptados: SHA-1, SHA-224, SHA-256, SHA-384 y SHA-512. Sin embargo, varios ataques han sido publicados contra SHA-1 en los últimos años. Por lo tanto, en respuesta a los avances en el criptoanálisis de los algoritmos Hash, el Instituto Nacional de Estándares y Tecnologías (NIST: National Institute of Standards and Technology) realizó una convocatoria pública para seleccionar el algoritmo SHA-3, similar al proceso para seleccionar el estándar de encriptación avanzada (AES: Advanced Encryption Standard). Los cinco algoritmos finalistas fueron: Grøstl, JH, Keccak, Blake y Skein. El algoritmo seleccionado para el estándar SHA-3 fue Keccak (Federal Information Processing Standards, 2008).

Estos algoritmos pueden ser implementados en *hardware* o *software*. En el caso del *hardware*, existen varios trabajos presentados en la literatura para los algoritmos seleccionados en la ronda 2 y 3. Kris, Ekawat y Rogawski (2010) implementaron los algoritmos usando VHDL comportamental para evaluar su desempeño en los FPGA de Xilinx y Altera, y como resultado existen diferencias considerables para las implementaciones de 512-bits. Solamente Keccak y CubeHash tienen una mejor relación *throughput*-área

con respecto al estándar, SHA-512. Se implementaron los algoritmos de la segunda ronda usando tecnología ASIC para 90nm y 130nm. En este caso, la comparación *throughput*-área presenta como el mejor algoritmo a Luffa seguido por Keccak (Guo *et al*., 2010). Los algoritmos de la segunda ronda se implementaron usando VHDL comportamental y teniendo en cuenta métricas de diseño para hacer una comparación equitativa (Tillich *et al*., 2010). Los resultados mostraron que Keccak y Luffa presentan la mejor relación *throughput*-área. También se muestra que otros algoritmos tienen mejor *throughput* que el SHA-256. Se obtuvo que el algoritmo con mayor consumo de área es Grøstl (4525 slices), y el algoritmo con mayor frecuencia es JH (250.125 MHz) (Baldwin *et al*., 2010). Los algoritmos de la tercera ronda se implementaron usando tecnología ASIC, donde se presenta una metodología de evaluación para la implementación en un ASIC, y se reportan los resultados para los candidatos de la ronda final (Guo *et al*., 2011).

Se implementaron en *hardware* las funciones de compresión de algunos candidatos para el estándar SHA-3, donde las implementaciones de los algoritmos Skein y Blake presentan el mayor *throughput* (Namin y Hasan, 2011).

Los algoritmos de la ronda 3 se implementaron usando seis metodologías de diseño, y se seleccionó el algoritmo con mejor área, consumo de potencia y relación *throughput*-área (Homsirikamol, Rogawski and Gaj, 2011).

En este trabajo se implementó eficientemente en *hardware* los cinco algoritmos finalistas para la función Hash SHA-3, con el propósito de disponer de los dos mejores algoritmos que presenten las dos figuras de mérito desde el punto de vista de *throughput*, recursos de área y relación *throughput*-área, para ser implementados en los FPGA de Altera. Cada algoritmo seleccionado se usó como un bloque funcional en la implementación de un criptosistema embebido de curvas elípticas para firma digital. Los diseños se simularon y sintetizaron usando Modelsim y Quartus II v.11.0, respectivamente. La principal contribución de este trabajo es el diseño eficiente de dos algoritmos Hash que pueden ser re-usados como *IP cores* en aplicaciones de criptografía embebidas en *hardware*.

El artículo se organizó de la siguiente manera:

- Los conceptos básicos sobre la función Hash
- Las arquitecturas *hardware* para cada algoritmo
- Los resultados de síntesis y simulación
- Las conclusiones

Función criptográfica Hash

Las funciones Hash son algoritmos para proteger la autenticidad de la información; estas funciones comprimen una cantidad arbitraria de datos de entrada (cadenas de caracteres, archivos binarios o paquetes TCP), y generan una cantidad fija de datos de salida. Sin embargo, es posible que dos mensajes distintos puedan tener el mismo valor Hash, esto es llamado *colisión*. Entonces, con el propósito de diseñar una función Hash, Ralph C. Merkle (en Knopf, 2007) propone un esquema llamado "Merkle's Meta Method", el cual fue modificado y recibió el nombre de Merkle-Damgard. A lo largo de los años este esquema ha demostrado tener muy buenas características de seguridad, y ha sido adoptado en el diseño de muchas funciones Hash exitosas, tales como MD5 y SHA-2 (Figura 2).

En el esquema Merkle-Damgard, el mensaje es modificado (*padded*) y dividido en bloques de tamaño uniforme. Los bloques son procesados secuencialmente usando una función de compresión C que genera un valor Hash intermedio a partir de previos valores Hash. La salida de la compresión final es el valor Hash del mensaje inicial, conocido como *digest*.

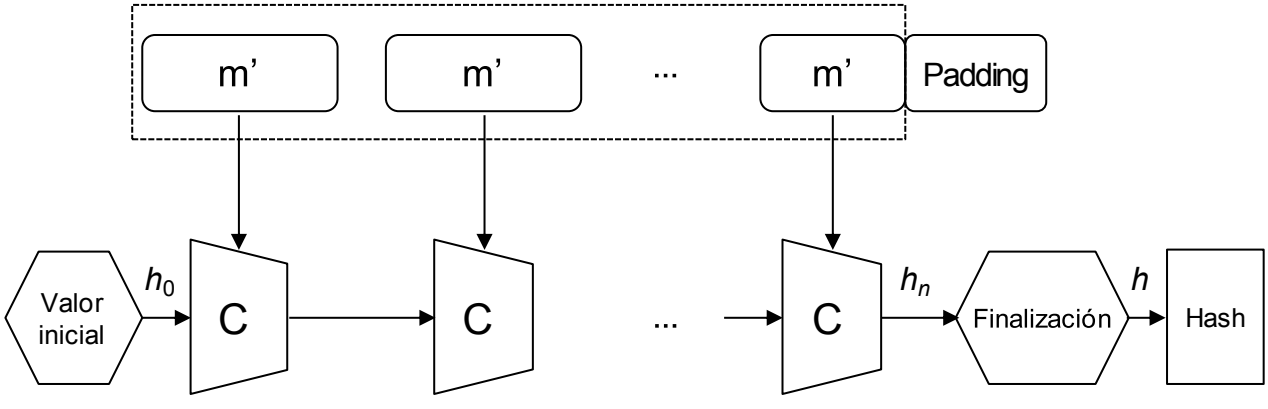

Figura 2. Función Hash

Diseño hardware para las funciones Hash

Se presentan las implementaciones *hardware* de los algoritmos JH, Blake, Skein, Grostl y Keccak.

Función Hash JH

La función Hash JH descrita por Hongjun (2011), sigue la metodología de diseño propuesta para AES, y se implementó usando una función de compresión llamada F8.

Descripción de la función Hash JH

La función F8 comprime el mensaje modificado M^i de 512 bits y el valor Hash anterior H^{i-1} de 1024 bits en un nuevo valor H^i de 1024 bits. La función F8 de Hash JH es presentada en el Algoritmo 1 (Hongjun, 2011).

Función F8

Entrada: $H^{(i-1)}$, $M^{(i)}$ **Salida**: $H^{(i)}$

1. $A^{(i)} = H^{(i-1),j} \oplus M^{(i),j}$ for $0 \le j \le 511$;
   $A^{(i)} = H^{(i-1),j}$ for $512 \le j \le 1023$;
2. $B = E8(A)$;
3. $H^{(i),j} = B^j$ for $0 \le j \le 511$;
   $H^{(i),j} = B^j \oplus M^{(i),j-512}$ for $512 \le j \le 1023$;

Algoritmo 1. Función F8

La función F8 es implementada a partir de la función bijectiva E8, la cual se presenta en el Algoritmo 2. La función E8 usa la metodología SPN (Substitution-Permutation Network) y códigos MDS (Maximum Distance Separable) para procesar un arreglo de bits en cada ronda. Esta función tiene como entrada y salida dos datos de 1024 bits, y calcula la función R8 durante 42 veces, usando el parámetro C_r de 256 bits. Una ronda de la función R8 ejecuta tres procesos: S-boxes, transformación lineal y permutación, y cuando se procesan las 42 rondas se genera el *digest* por truncamiento de H^i.

Función E8

Entrada: A **Salida**: B

1. Agrupar los bits de A en 256 bloques de 4 bits cada uno para obtener Q_0
   for $0 \le i \le 41$
   $\{ Q_{0,2i} = A^i \| A^{i+256} \| A^{i+512} \| A^{i+768}$;
   $Q_{0,2i+1} = A^{i+128} \| A^{i+384} \| A^{i+640} \| A^{i+896} \}$
2. $Q_{r+1} = R8(Q_r, C_r)$; for $0 \le r \le 41$;
3. Desagrupar los 256 bloques de 4 bits guardados en Q_{42} para obtener B
   for $0 \le i \le 41$
   $\{ B^i \| B^{i+256} \| B^{i+512} \| B^{i+768} = Q_{42,2i}$;
   $B^{i+128} \| B^{i+384} \| B^{i+640} \| B^{i+896} = Q_{42,2i+1} \}$

Algoritmo 2. Función E8

Arquitectura hardware de la función Hash JH

En la Figura 3 se muestra el diagrama de bloques de la unidad de flujo de datos de la implementación *hardware* de la función Hash JH. Los bloques de inicialización y finalización ejecutan los pasos 1 y 3 del Algoritmo 3, y los bloques de agrupar y desagrupar ejecutan una permutación de bits.

Los bloques Cnt0 y Cnt1 se usaron para contar el número de bloques de mensaje y las rondas, respectivamente. Estos bloques generaron las señales Z0 y Z1 que indicaron la finalización del procesamiento del mensaje y de las 42 rondas, respectivamente.

Cada vez que el *hardware* procesó un nuevo mensaje, las constantes de inicialización H^0 se calcularon usando los bloques *SBox*, *TLinear* y permutaron. La función F8 usó la Ecuación 1 para calcular la constante H^0. La ecuación tuvo como entradas un mensaje M^0 de 512 bits, los cuales son todos cero, y la constante H^{-1} con el valor 0x02.

$$H^0 = F8(H^{-1}, M^0) \quad (1)$$

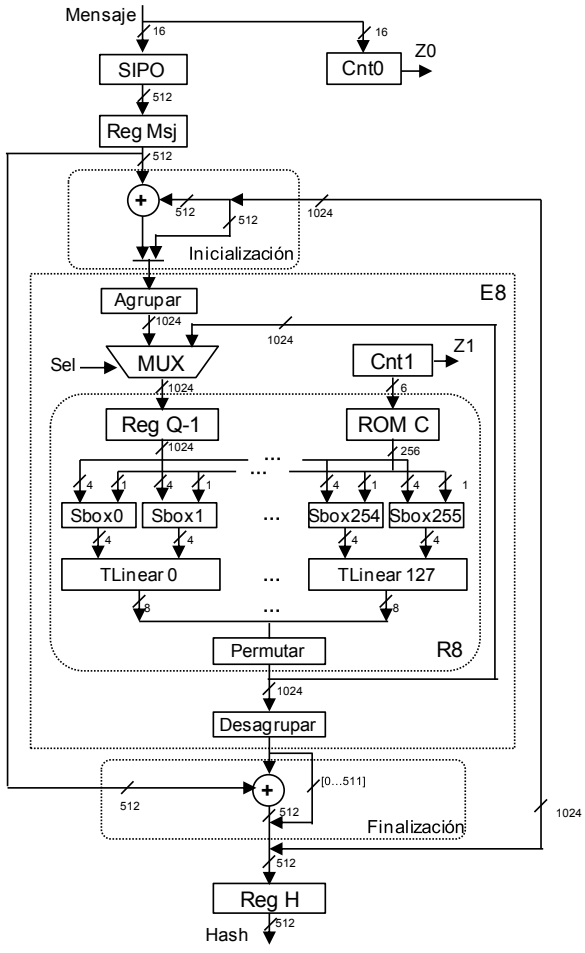

Figura 3. Diagrama de bloques de la unidad de flujo de datos de la función Hash JH

Función Hash Blake

La función Hash Blake descrita por Aumasson (2011), usa un algoritmo de compresión basado en una versión modificada del algoritmo de Bernstein de cifrado llamado ChaCha (Aumasson, 2008).

Descripción de la función Hash Blake

Blake usa la función de compresión presentada en el Algoritmo 3 que contó con los siguientes parámetros:

- Valor Hash anterior de 512 bits
- Bloque de mensaje *m* de 1024 bits
- Parámetro salpicador *s* de 256 bits
- Un contador *t* de 128 bits

Esta función usó tres bloques funcionales: inicialización, rondas y finalización.

Función compresión

Entrada: h, m, s, t **Salida**: h^N

$h^0 \leftarrow IV$

for $0 \leq i \leq N-1$

$\{ \quad h^{i+1} = compress(h^i, m^i, s, t) \quad \}$

return h^N

Algoritmo 3. Función compresión

Bloque de inicialización

Blake inició el proceso de compresión usando 8 valores de inicialización IV y 16 constantes *c*, donde el bloque de inicialización usó una matriz de estados de tamaño 4x4, y los parámetros *h*, *s*, *t* y las constantes *c*, tal como se describe en la Ecuación (2).

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} = \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7 \end{pmatrix} \quad (2)$$

Bloque de rondas

Este bloque usó la función compresión durante 16 rondas *r*, y una ronda es una transformación en la matriz de estado que calculó la función G_i con los siguientes parámetros de entrada:

$$G_0(v_0, v_4, v_8, v_{12}) \quad G_1(v_1, v_5, v_9, v_{13}) \quad G_2(v_2, v_6, v_{10}, v_{14}) \quad G_3(v_3, v_7, v_{11}, v_{15})$$
$$G_4(v_0, v_5, v_{10}, v_{15}) \quad G_5(v_1, v_6, v_{11}, v_{12}) \quad G_6(v_2, v_7, v_8, v_{13}) \quad G_7(v_3, v_4, v_9, v_{14})$$

En cada ronda *r* se calcularon los valores G_i usando la función G presentada en el Algoritmo 4.

Función G

Entrada: a, b, c, d **Salida**: a, b, c, d

1. $a \leftarrow a + b + (m_{\sigma r(2i)} \oplus c_{\sigma r(2i+1)})$
2. $d \leftarrow (d \oplus a) >> 32$
3. $c \leftarrow c + d$
4. $b \leftarrow (b \oplus c) >> 25$
5. $a \leftarrow a + b + (m_{\sigma r(2i+1)} \oplus c_{\sigma r(2i)})$
6. $d \leftarrow (d \oplus a) >> 16$
7. $c \leftarrow c + d$
8. $b \leftarrow (b \oplus c) >> 11$

Algoritmo 4. Función G

La función G usó la tabla de permutaciones (sigma σ) para seleccionar el bloque de mensaje y la constante que se procesó en cada función G. En la Figura 4 se muestra la transformación de la matriz de estado usando 8 funciones G, donde la transformación para las funciones $G_0 \ldots G_3$ y $G_4 \ldots G_7$ se llamaron etapa columna y diagonal, respectivamente.

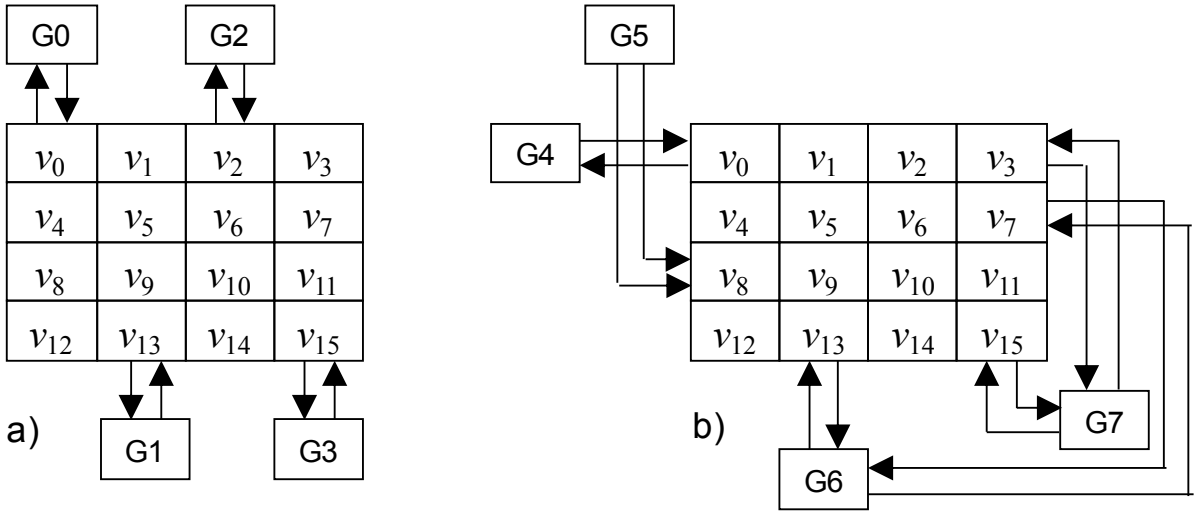

Figura 4. Matriz y transformaciones G: a) etapa columna, b) etapa diagonal

Bloque de finalización

Este bloque calculó el valor final Hash, $h'_0 \ldots h'_7$, usando los valores de la matriz de estado, los valores iniciales $h_0 \ldots h_7$, y los valores generados por el salpicador $s_0 \ldots s_3$ tal como se presenta en el Algoritmo 5.

Algoritmo Hash final

Entrada: h, s, v **Salida**: h'

1. $h'_0 \leftarrow h'_0 \oplus s_0 \oplus v_0 \oplus v_8$
2. $h'_1 \leftarrow h'_1 \oplus s_1 \oplus v_1 \oplus v_9$
3. $h'_2 \leftarrow h'_2 \oplus s_2 \oplus v_2 \oplus v_{10}$
4. $h'_3 \leftarrow h'_3 \oplus s_3 \oplus v_3 \oplus v_{11}$
5. $h'_4 \leftarrow h'_4 \oplus s_0 \oplus v_4 \oplus v_{12}$
6. $h'_5 \leftarrow h'_5 \oplus s_1 \oplus v_5 \oplus v_{13}$
7. $h'_7 \leftarrow h'_7 \oplus s_3 \oplus v_7 \oplus v_{15}$

Algoritmo 5. Algoritmo Hash final

Arquitectura hardware de la función Hash Blake

En la Figura 5 se muestra un diagrama de bloques de la unidad de flujo de datos de la implementación *hardware* de la función Hash Blake, donde el bloque *msj-bank* almacena el mensaje, los bloques etapa columna y etapa diagonal reorganizan la matriz de estado, la memoria ROM C almacena las constantes *c*, y la memoria *ROM perm* almacena la tabla de permutaciones (sigma σ) en dos sectores, etapa columna y etapa diagonal. Los contadores cnt0 y cnt1 cuentan el número de bloques del mensaje y las rondas, respectivamente.

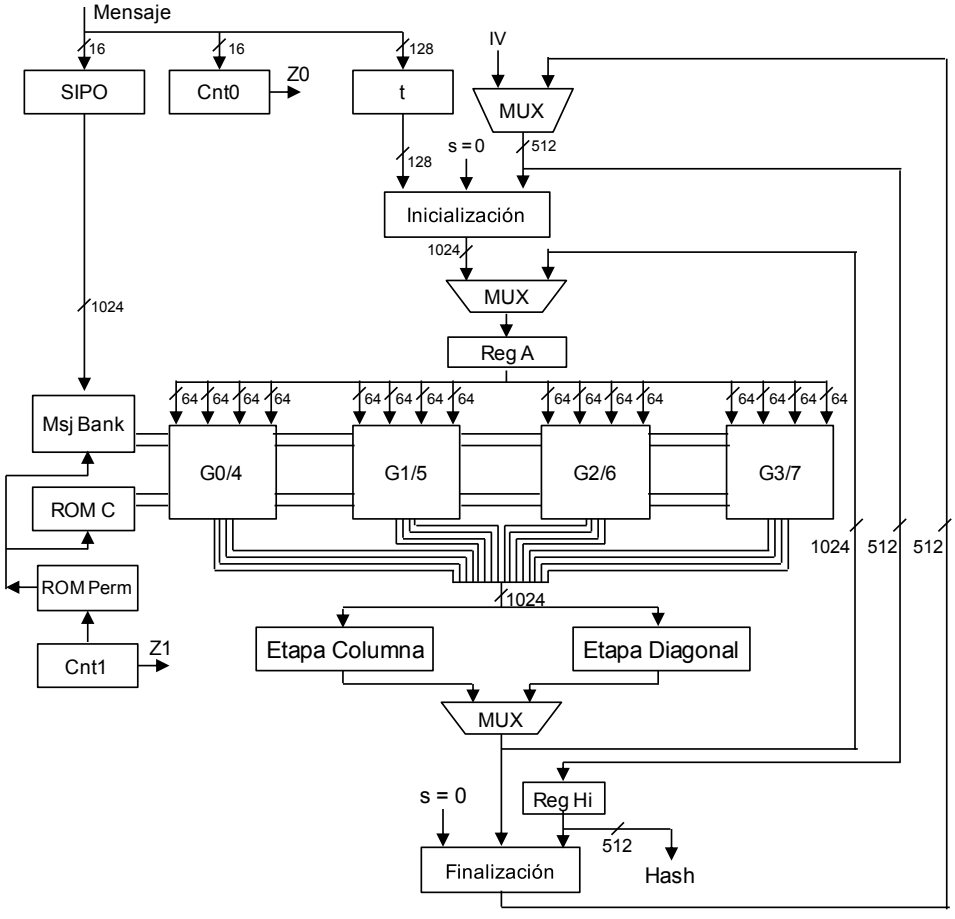

Figura 5. Diagrama de bloques de la unidad de flujo de datos de la función Hash Blake

Función Hash Skein

Esta función se basó en la implementación de una función Hash a partir de un bloque de cifrado reconfigurable que permite obtener valores de compresión únicos en cada bloque (Gauravaram, 2011). La función Hash Skein se implementó usando tres bloques funcionales: *Threefish*, *Unique Block Iteration* (UBI) y *Optional Argument System*. *Threefish*, bloque principal donde se llevó a cabo el cifrado reconfigurable para un mensaje de 256, 512 o 1024 bits. El bloque UBI usó una cadena de bloques *Threefish* para realizar la compresión que mapeó el mensaje de entrada de tamaño arbitrario en un mensaje de salida de longitud fija. El bloque *Optional Argument System* permitió que Skein soportara una variedad de características opcionales.

Descripción de la función Hash Skein

El bloque de cifrado *Threefish* se basó en la realización de un gran número de rondas simples (Skein-512 usa 72 rondas), lo que ofreció mayor seguridad que realizar pocas rondas complejas. El bloque de cifrado E(*K*, *T*, *P*) tiene los siguientes argumentos de entrada:

K: Clave del bloque de cifrado de 64 bytes para Skein-512

T: *Tweak* de 16 bytes

P: Texto plano de 64 bytes para Skein-512

La clave *K* se calculó por el bloque de subclaves, que generó una secuencia de 18 subclaves, cada una formada por 8 palabras de 64 bits, las que son denotadas como (*k*_(s,0),…,*k*_(s,*Nw*-1)) con *Nw* = 8. Las subclaves se adicionaron en cada ronda usando el Algoritmo 6, donde el parámetro v_(d,i) es el valor de la i-ésima palabra de un estado de encriptación después de *d* rondas. En la ronda *d*=0 la palabra v_(0,i) corresponde al mensaje de entrada o texto plano, luego de cada 4 rondas se suma una subclave.

Algoritmo adición de subclaves

Entrada: $v_{d,i}, k_{d/4,i}$ **Salida:** $e_{d,i}$

for $0 < d < 71$

for $0 < i < 7$

$v_{0,i} = p_i$

if $(d \bmod 4 = 0)\{$

$e_{d,i} = \left(v_{d,i} + k_{\frac{d}{4},i}\right) \bmod 2^{64}\}$

Else

$e_{d,i} = v_{d,i}$

Algoritmo 6. Algoritmo adición de subclaves

En cada ronda se realizó una función MIX y una función de permutación, tal como se muestra en el Algoritmo 7.

Algoritmo de ronda

Entrada: $e_{d,i}$ **Salida:** $f_{d,i}, v_{d+1,i}$

for $0 < j < 3$

$(f_{d,2j}, f_{d,2j+1}) = MIX_{d,j}(e_{d,2j}, e_{d,2j+1})$

for $0 < i < 8$

$(v_{d+1,i}) = f_{d,\pi(i)}$

Algoritmo 7. Algoritmo de ronda

Donde $f_{d,i}$ son los datos de salida de la función MIX. En una ronda es obtenida la palabra de permutación y la permutación π(i) se muestra en la Tabla 1.

Tabla 1. Permutaciones para Skein

i	0	1	2	3	4	5	6	7
π(i)	2	1	4	7	6	5	0	3

La función MIX se presenta en el Algoritmo 8 y usó tres operaciones de 64 bits: XOR, suma y rotaciones de constantes.

Función MIX

Entrada: x_0, x_1 **Salida:** y_0, y_1

1. $y_0 \leftarrow (x_0 + x_1) \bmod 2^{64}$
2. $y_1 \leftarrow \left(x_1 \ll R_{(d \bmod 8),j}\right) \oplus y_0$

Algoritmo 8. Función MIX

Donde el número rotaciones $R_{d,j}$ cambió de acuerdo con la ronda *d*, tal como se muestra en la Tabla 2.

Tabla 2. Constantes de rotación $R_{d,j}$ para Skein

	j	0	1	2	3
d =	0	46	36	19	37
	1	33	27	14	42
	2	17	49	36	39
	3	44	9	54	56
	4	39	30	34	24
	5	13	50	10	17
	6	25	29	39	43
	7	8	35	56	22

Las subclaves se generaron usando el Algoritmo 9.

Algoritmo de generación de subclaves

$k_{[Nw]} = C_{240} \oplus \bigoplus_{i=0}^{Nw-1} \mathrm{k_i}$ y $t_2 = t_0 \oplus \mathrm{t_1}$

$k_{s,i} = k_{(s+i)mod(N_w+1)}$ For $i = 0, \ldots, N_w - 4$

$\mathrm{k_{s,i}} = \mathrm{k_{(s+i)mod(N_w+1)}} + \mathrm{t_{s\ mod\ 3}}$ For $i = N_w - 3$

$k_{s,i} = k_{(s+i)mod(N_w+1)} + t_{(s+1)mod\ 3}$ For $i = N_w - 2$

$k_{s,i} = k_{(s+i)mod(N_w+1)} + s$ For $i = N_w - 1$

Algoritmo 9. Algoritmo generación de subclaves

Donde s es el número de subclave, C_{240} es una constante de 64 bits, t es el *Tweak* actual y k es la clave anterior.

El bloque UBI es un modo de conectar bloques *Threefish* para implementar una función de compresión. Este bloque tiene una entrada de tamaño arbitrario y una salida de tamaño fijo, y cada UBI procesa un bloque de mensaje usando un diferente valor de *Tweak* de 128 bits.

Arquitectura hardware de la función Hash Skein

El diseño en *hardware* de esta función consistió en implementar las 72 rondas usando cuatro rondas en paralelo teniendo en cuenta que la función Hash adiciona cada cuatro rondas una subclave que se generó por la unidad de subclaves. En la Figura 6 se muestra el diagrama de bloques de la unidad de flujo de datos de la implementación *hardware* de la función Hash Skein, donde el bloque FF configura el bloque MIX con el parámetro d entre 0 y 3 cuando su señal de salida es cero, en caso contrario, entre 4 y 7.

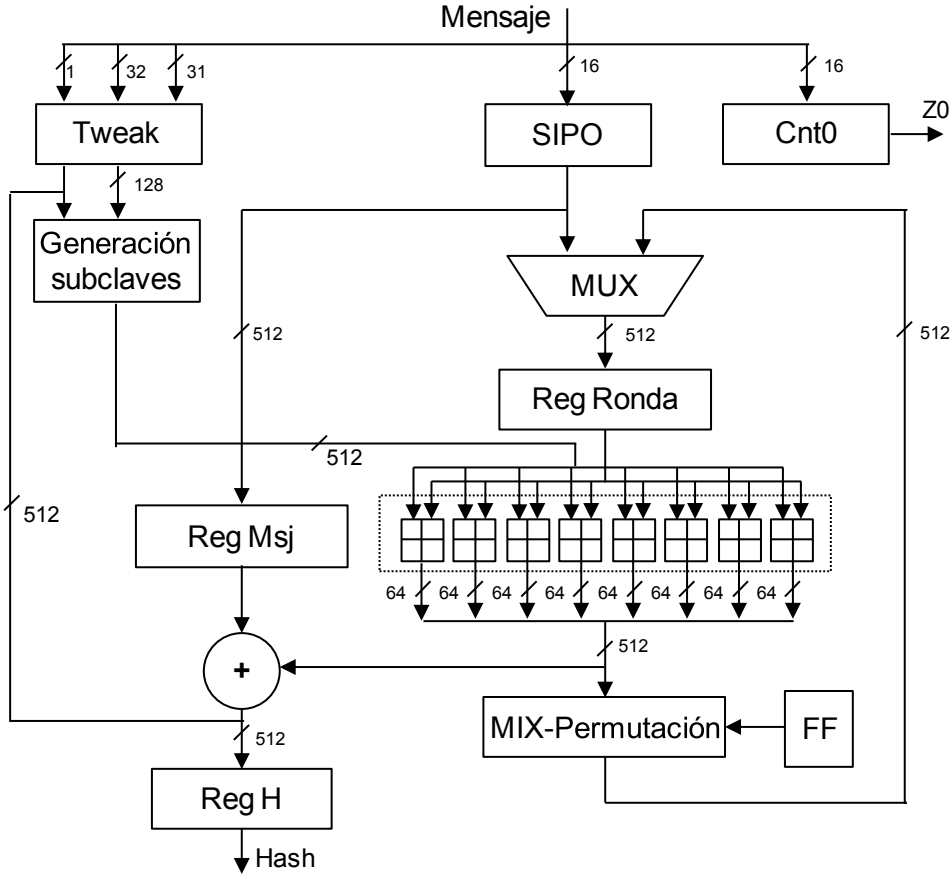

Figura 6. Diagrama de bloques de la unidad de flujo de datos de la función Hash Skein

Función Hash Grostl

El algoritmo propuesto por Lars *et al*., basó su diseño en la metodología SPN que usó componentes del estándar AES. Grostl pudo retornar un resumen del mensaje de tamaño 224, 256, 384 y 512 bits. A continuación se describen las características más importantes de este algoritmo.

Descripción del algoritmo de la función Hash Grostl

La función Grostl procesa secuencialmente un bloque m_i del mensaje M. En este caso, los valores h_i son calculados usando un valor anterior h_{i-1} y un bloque de mensaje m_i, cuando todo el mensaje M ha sido procesado, el valor Hash es obtenido por truncamiento de h_t, este proceso se ilustra en el Algoritmo 10.

Algoritmo Grostl

Entrada: m_i, h_{i-1} **Salida**: h_i

for $0 < i < t$

$h_i = f(h_{i-1}, m_i)$

if $(i = t)\{H(M) = \Omega(h_t)\}$

Algoritmo 10. Algoritmo Grostl

Donde $\Omega(x) = trunc_n(P(x) \oplus x)$ y $trunc_n(x)$ y es truncar los n bits menos significativos de x.

Para obtener el valor Hash de 512 bits, Grostl-512 llevó a cabo 14 rondas, donde la función f usó un par de bloques de cifrado P y Q de 1024 bits cada uno, y los bloques usaron cuatro tipos de transformaciones para la ronda R descrita por la Ecuación 3 y compuesta por cuatro transformaciones de ronda, las cuales son *AddRoundConstant*, *SubBytes*, *ShiftBytesWide* y *MixBytes* (Aumasson, 2011).

$$R = MixBytes(ShiftBytesWide(SubBytes(AddRoundConstant))) \quad (3)$$

En el Algoritmo 11 se describe brevemente la transformación de la ronda.

Algoritmo transformación de ronda

1. *AddRoundConstant*

for $0 < i < 13$

$A = A \oplus C(i)$

2. *SubBytes*

for $0 < i < 8$

for $0 < j < 15$

$a_{i,j} = S(a_{i,j})$

3. ShiftBytesWide

$A = \sigma(A)$

4. *MixBytes*

$A = B \times A$

Algoritmo 11. Algoritmo de transformación de ronda

La transformación *AddRoundConstant* realizó una operación XOR en cada ronda entre la matriz A y la constante $C_{(i)}$, donde la constante $C_{(i)}$ es diferente para bloque de cifrado P ($Cp_{(i)}$) y Q ($Cq_{(i)}$). La transformación *SubBytes* realizó la sustitución de cada byte de la matriz A por un valor de la tabla S-box. La transformación *ShiftBytesWide* realizó la rotación a la izquierda de los bytes de una fila de la matriz A. Para los bloques P y Q, el número de bytes a rotar se especificó con los vectores σ = [0,1,2,3,4,5,6,11] y σ = [1,3,5,11,0,2,4,6], respectivamente. La transformación *MixBytes* realizó la multiplicación entre las matrices A y B sobre GF(2^8), es decir, cada columna de la matriz A se multiplicó por una constante de la matriz B. En este caso, el polinomio irreducible del campo finito es $x^8+x^4+x^3+x^1$.

Arquitectura hardware de la función Hash Grostl

En la Figura 7 se muestra el diagrama de bloques de la unidad de flujo de datos de la implementación *hardware* de la función Hash Grostl. El diseño de los bloques P(ShiftBytesP) y Q(ShiftBytesQ) fue muy similar, lo que redujo significativamente el área y aumentó el *throughput*.

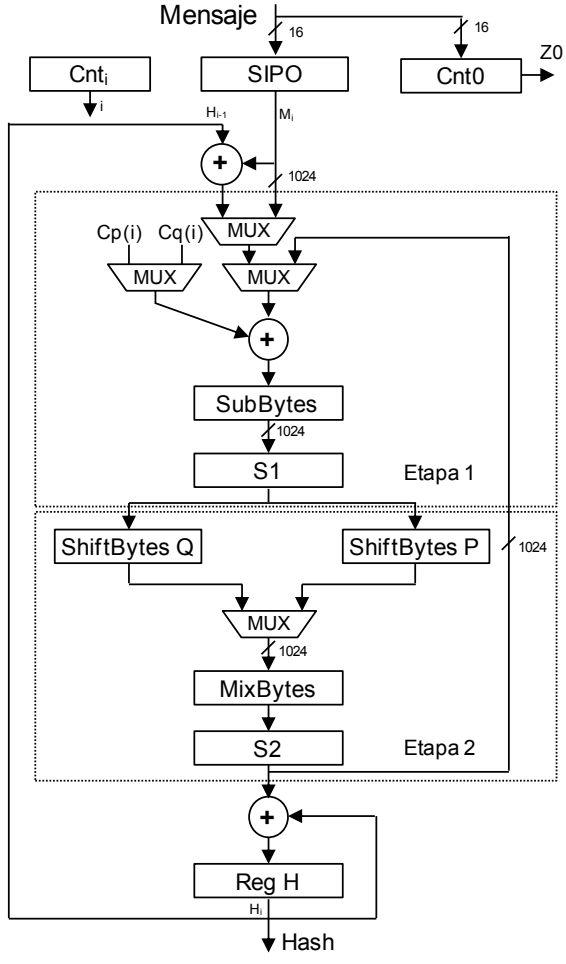

Figura 7. Diagrama de bloques de la unidad de flujo de datos de la función Hash Grostl

Las transformaciones *AddRoundConstant* y *ShiftBytesWide* son configuradas para trabajar con los bloques P o Q. La etapa 1 se procesó en el bloque P, y la etapa 2 se procesó simultáneamente en los bloques P y Q.

Función Hash Keccak

Cualquier implementación de una familia de funciones esponja Keccak usa una de las siete permutaciones Keccak-f (Bertoni *et al*., 2001), denotada Keccak-f[b], donde el ancho de la permutación $b \in \{25, 50, 100, 200, 400, 800, 1600\}$. Las permutaciones Keccak-f son estructuras que constan de una secuencia de rondas casi idénticas. El número de rondas n_r depende de b, y está dada por $n_r = 12 + 2l$, donde $2^l = b/25$. Por ejemplo, Keccak-f [1600] usa 24 rondas. Un resumen de la función Hash Keccak-f[b] se presenta en el Algoritmo 12 (Bertoni, G., *et. al*., 2011)

Algoritmo Keccak- *f* [*b*](A)

for *i* in 0 . . . n_r - 1

A = Round[*b*](*A*,RC[*i*])

return *A*

Algoritmo 12. Algoritmo Keccak- *f* [*b*](A)

Descripción del algoritmo de la función Hash Keccak

La ronda Keccak-f consistió de una secuencia de transformaciones invertibles, donde cada una realizó un proceso sobre la matriz de estado A de 5x5 líneas (*lanes*), y cada línea tiene una longitud $w \in \{1, 2, 4, 8, 16, 32, 64\}(b = 25w)$. El Algoritmo 13 describe las transformaciones de la ronda de Keccak-f.

Transformaciones de ronda

Entrada: b, A, RC

Salida: A $\forall\ x\ in\ 0 \ldots 4$

1. Transformación θ $\forall\ x\ in\ 0 \ldots 4$

$C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4],$ $\forall\ (x,y)\ in\ (0 \ldots 4, 0 \ldots 4)$

$D[x] = C[x-1] \oplus ROT[C[x+1],1],$

$A[x,y] = A[x,y] \oplus D[x],$

2. Transformaciones ρ y π $\forall\ (x,y)\ in\ (0 \ldots 4, 0 \ldots 4)$

$B[y, 2x+3y] = ROT[A[x,y], r[x,y]],$

3. Transformación χ

$A[x,y] = B[x,y] \oplus (\ (NOT\ B[x+1,y])\ AND\ B[x+2,y]),$ $\forall\ (x,y)\ in\ (0 \ldots 4, 0 \ldots 4)$

4. Transformación ι

$A[0,0] = A[0,0] \oplus RC[i],$

Return A

Algoritmo 13. Transformaciones de ronda

Todas las operaciones de las transformaciones se realizaron usando aritmética módulo 5. Donde, A es la matriz completa de estado y A [x, y] es una línea particular en la matriz de estado. B [x, y], C [x] y D [x] son variables intermedias; el símbolo ⊕ es la función OR exclusiva; ROT (W, r) es un desplazamiento cíclico de r-bits, donde r es el módulo de la longitud de una línea de la matriz A, y las constantes r [x, y] son los off-sets de los desplazamientos cíclicos, especificados en la Tabla 3.

Tabla 3. Constantes r [x, y]

	x = 3	*x* = 4	*x* = 0	*x* = 1	*x* = 2
y = 2	25	39	3	10	43
y = 1	55	20	36	44	6
y = 0	28	27	0	1	62
y = 4	56	14	18	2	61
y = 3	21	8	41	45	15

Las constantes RC[i] son las contantes de ronda, cuyos valores en formato hexadecimal para una línea de tamaño 64 son mostrados en la Tabla 4.

Tabla 4. Constantes RC[i]

RC[0]	0x0000000000000001	RC[12]	0x000000008000808B
RC[1]	0x0000000000008082	RC[13]	0x800000000000008B
RC[2]	0x800000000000808A	RC[14]	0x8000000000008089
RC[3]	0x8000000080008000	RC[15]	0x8000000000008003
RC[4]	0x000000000000808B	RC[16]	0x8000000000008002
RC[5]	0x0000000080000001	RC[17]	0x8000000000000080
RC[6]	0x8000000080008081	RC[18]	0x000000000000800A
RC[7]	0x8000000000008009	RC[19]	0x800000008000000A
RC[8]	0x000000000000008A	RC[20]	0x8000000080008081
RC[9]	0x0000000000000088	RC[21]	0x8000000000008080
RC[10]	0x0000000080008009	RC[22]	0x0000000080000001
RC[11]	0x000000008000000A	RC[23]	0x8000000080008008

La función esponja keccak [*r*,c], con parámetros capacidad *c* y rata de bit *r,* es obtenida cuando se aplica la transformación esponja a keccak-f[r+c] y se realizó el relleno (*padding*) del mensaje de entrada M. El Algoritmo14 describe la función esponja keccak [*r*,c], para el caso de mensajes que tienen como relleno un número completo de bytes, y donde la rata de bit *r* es un múltiplo del tamaño de la línea.

Función esponja Keccak [r,c](M)

1. Padding

$P = M \parallel \text{pad}[r](|M|)$

2. Initialización

$s = 0^b$

3. Fase de absorción

for $i = 0$ to $|P|_r - 1$ **do**

$s = s \oplus (P_i \parallel 0^{b-r})$

$s = Keccak - f[b](s)$

end for

4. Fase de exprimir

$Z = \lfloor s \rfloor_r$

while $|Z|_r r < l$ do

$s = Keccak - f[b](s)$

$Z = Z \parallel \lfloor s \rfloor_r$

end while

Return $\lfloor Z \rfloor_l$

Algoritmo 14. Función esponja keccak [r,c]

Donde S indica el estado que consiste de un arreglo de líneas; *P* es el mensaje con el relleno, el cual es organizado como un arreglo de bloques P_i que también son organizados como matrices de líneas; y el operador *jj* denota la concatenación de cadenas de bytes.

Arquitectura hardware de la función Hash Keccak

La implementación en *hardware* de la función Hash Keccac consiste en diseñar las respectivas transformaciones del bloque de cifrado f. En la Figura 8 se muestra un diagrama de bloques de la unidad de flujo de datos de la implementación *hardware* de la función Hash Keccak.

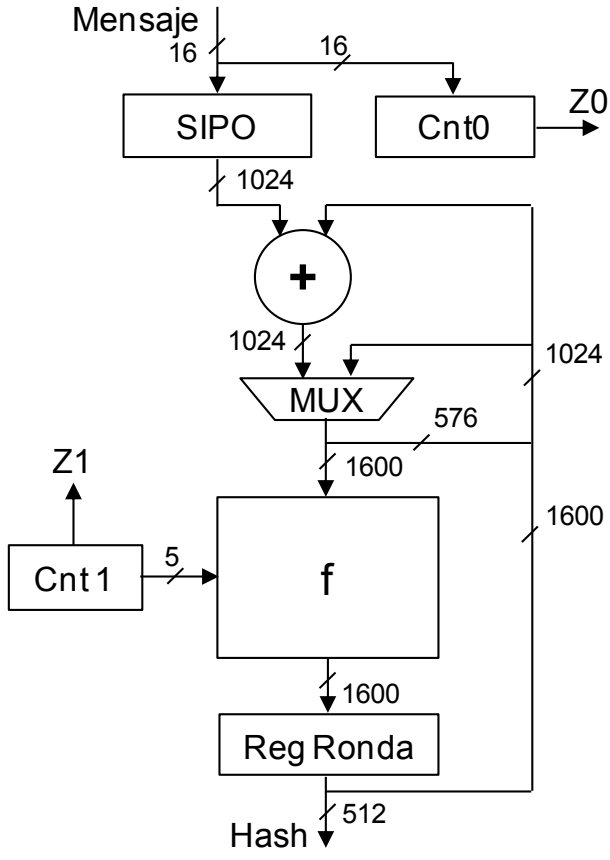

Figura 8. Diagrama de bloques de la unidad de flujo de datos de la función Hash keccak

El bloque de cifrado *f* es implementado usando los siguientes bloques funcionales:

***Bloque de la transformación theta* (θ)**

La Figura 9 muestra el diagrama de bloques de la implementación *hardware* de la transformación theta, donde cada registro X-1, X y X+1 son vectores de 5 datos de w bits.

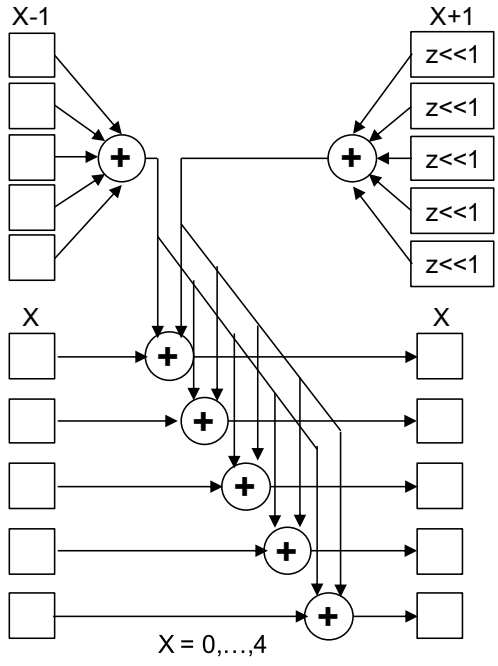

Figura 9. Transformación Theta

***Bloque de la transformación chi* (χ)**

La Figura 10 a) muestra el diagrama de bloques de la implementación *hardware* de la transformación chi, la cual realiza la operación

$$A[x,y]=B[x,y]\oplus((NOT\ B[x+1,y])\ \ AND\ B[x+2,y]).$$

***Bloque de la transformación iota* (ι)**

La Figura 10 b) muestra el diagrama de bloques de la implementación *hardware* de la transformación iota, la cual realiza una XOR entre la constante RC[i] y la palabra 0,0 de la matriz de estado anterior. Las constantes RC [i] son almacenadas en una ROM de 24 palabras de 64 bits.

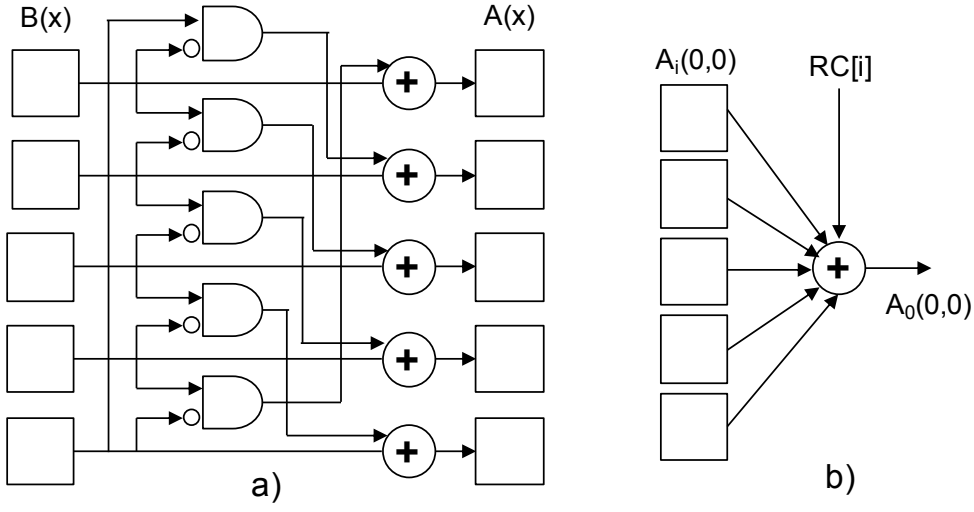

Figura 10. a) Transformación chi b) Transformación iota

***Bloque de las transformaciones* rho (ρ) y pi (π)**

Estas transformaciones son funciones de desplazamiento r, donde r es el número de desplazamientos módulo w. Estos bloques no requieren *hardware*.

Resultados de verificación y comparaciones

Para la verificación de los diseños de cada función Hash se implementaron los algoritmos en *software* usando Java, y sus resultados de simulación son comparados con los resultados obtenidos en *hardware* usando los respectivos vectores de prueba, los cuales se encuentran en la documentación de los algoritmos.

Los algoritmos son descritos en VHDL, simulados en Modelsim-Altera y sintetizados en diferentes familias de FPGAS de Altera usando Altera Quartus II v11. Las familias de FPGAS usadas son: I Cyclone II (EP2C-35F672C6), III (EP3C40F484C6) y IV GX (EP4CGX30C-F23C6), las cuales son de bajo costo y consumo de potencia; Arria II GX (EP2AGX45DF29C4) de gama media; y Statrix II (EP2S30F672C3), III (EP3SE50F484C2) y IV (EP4SE230F29C2), de alto desempeño. En la Tabla 5 se muestran los parámetros de cada función Hash con *digest* de 512 bits.

Tabla 5. Parámetros para cada función Hash

Función	Longitud de mensaje [bits]	Longitud de estado [bits]	Número de rondas	HTime
JH	512	1,024	42	44
Blake	1,024	1,024	16	32
Skein	512	512	18	19
Grostl	1,024	1,024	14	30
Keccak	576	1,600	24	24

En la Tabla 5, *HTime* es el número de ciclos de reloj necesarios para calcular el valor Hash de un solo bloque de mensaje.

En la Figura 11 se muestran los resultados de desempeño para los 5 finalistas de Hash-3, es decir, el *throughput*, los recursos de área, el consumo de potencia y la relación *throughput* / área (Tp / A), considerando diferentes familias de FPGAS de Altera.

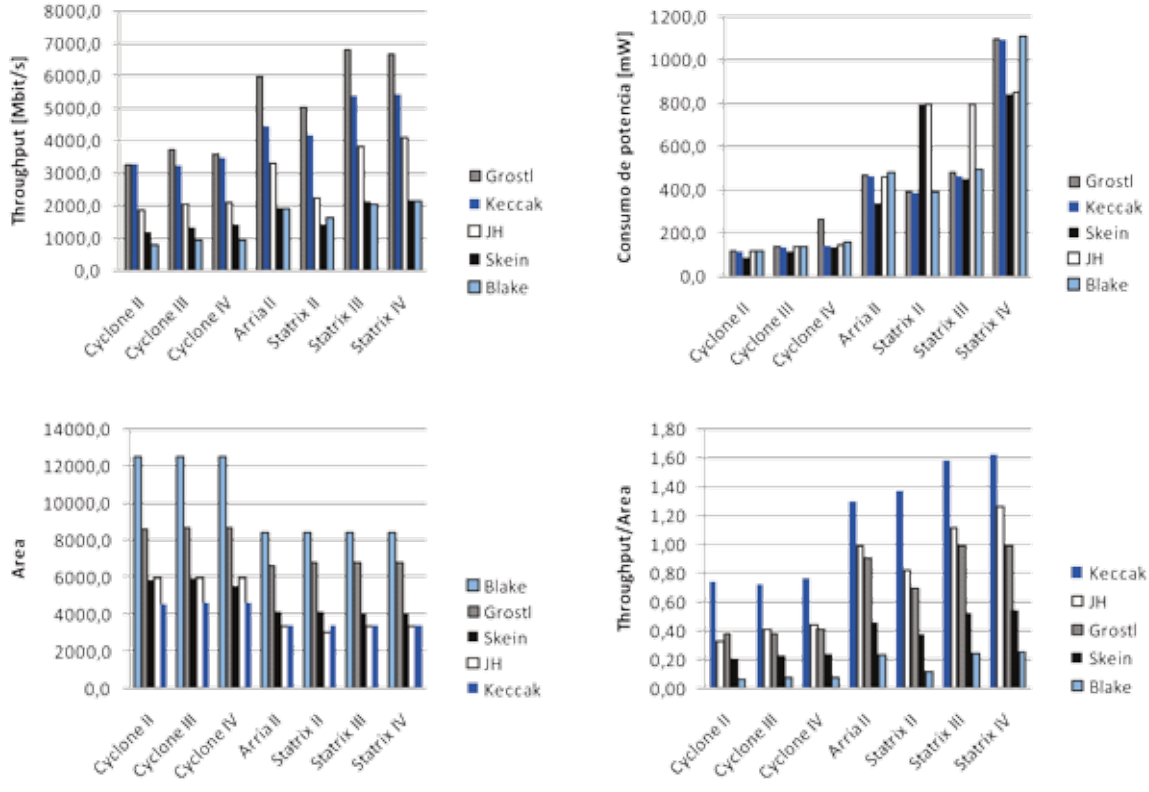

Figura 11. Comparación de resultados área, consumo de potencia, *throughput* y Tp/A

El *throughput* es el número de bits procesados por unidad de tiempo, y se calcula usando la Ecuación 4 (Homsirikamol, Rogawski, 2011).

$$Throughtput=(N*block\_size)/(T*HTime(N)) \quad (4)$$

En la Figura 11 se observó que la función que tiene el mayor *throughput* en todas las familias de FPGA es Grostl, seguida por la función Keccak, y el menor *throughput* lo tiene la función Blake.

Se observó que la función Blake consumió la mayor cantidad de recursos de área, seguida por la Grostl. La que consume la menor cantidad de recursos de área es Keccak, seguida por la función JH. La que presenta la mejor relación *throughput* / área es Keccak, seguida por JH. El menor consumo de potencia corresponde a la función Skein, y las funciones Grostl y Keccak presentaron un consumo de potencia muy similar.

Los resultados muestran que Grostl tiene el mayor *throughput* en todos los dispositivos, pero esta función usa una cantidad considerable de recursos de área. Es importante mencionar que la función Blake es la peor opción para ser implementada en *hardware*, porque presenta la mayor cantidad de recursos de área y tiene el menor *throughput*. Finalmente, se concluyó que Keccak, JH y Grostl son las funciones Hash más adecuadas para ser implementadas en *hardware*, porque presentan la mejor relación *throughput* / área.

Comparación con otros trabajos

Los resultados del trabajo y los presentados por Homsirikamol, Rogawski y Gaj (2011), se presentan en la Tabla 6. En este caso, los diseños se sintetizaron sobre un FPGAS Statrix III de Altera, y las comparaciones se realizaron teniendo en cuenta que los diseños de JH, Keccak y Skein usan una arquitectura *hardware* básica iterativa presentada en la Figura 12.a). La función Skein se implementó usando cuatro rondas (4R). El diseño de la función Grostl usó la arquitectura básica iterativa con dos etapas *pipeline* de media ronda, tal como se muestra en la Figura 12.b). El diseño de la función Blake usó la arquitectura básica iterativa con una etapa de media ronda presentada en la Figura 12.c).

Tabla 6. Resultados de síntesis para las funciones Hash

	Homsirikamol, Rogawski y Gaj. 2011			Los autores		
	Área [ALUTs]	*Throughput* [Mbits/s]	Thr/ Área	Área [ALUTs]	*Throughput* [Mbits/ s]	Thr/Área
JH	3,548	4,912	1.38	3,047	3,598	1.18
Blake	7,127	2,905	0.41	8,447	2,059	0.24
Skein	3,632	1,082	0.30	4,084	2,150	0.53
Grostl	22,570	17,050	0.76	6,929	6,706	0.97
Keccak	3,477	6,859	1.97	3,399	5,305	1.56

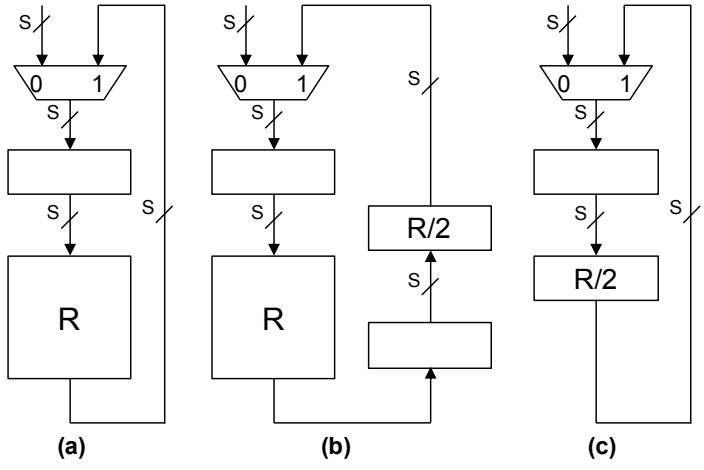

Figura 12. (a) Arquitectura básica iterativa (b) Arquitectura básica iterativa *pipeline* con dos etapas de media ronda (c) Arquitectura básica iterativa con una etapa de media ronda

Conclusiones

En este trabajo se implementaron en *hardware* los cinco algoritmos finalistas de la competencia SHA-3, donde los diseños de las funciones Hash usan una arquitectura básica iterada y *pipeline*. Cada algoritmo se sintetizó en siete FPGAs de Altera de diferentes familias, y para cada algoritmo se obtienen las métricas de desempeño como *throughput*, área, frecuencia y consumo de potencia.

Las métricas son graficadas, y se observó que la implementación del algoritmo Grostl tiene el mayor *throughput* en todas las FPGAs, no obstante, usó la mayor cantidad de recursos de área después de Blake. Los algoritmos que usaron la menor área son: Keccak y JH, sin embargo, es importante mencionar que Keccak usa la menor área en cyclone y JH en Arria y Statrix. También es fundamental observar que Keccak es el segundo algoritmo con mayor *throughput*. Entonces, Keccak tiene la mayor relación Tp/A, seguida por JH, y en el último lugar la menor relación Tp/A la tiene Blake.

El consumo de potencia fue similar en los cinco algoritmos para los diferentes FPGAs, con una excepción para JH en Cyclone IV, en donde este algoritmo presenta un valor de consumo de aproximadamente la mitad. Los algoritmos Blake y Grostl presentan el mayor consumo de potencia y área en la mayoría de los dispositivos.

Finalmente, se puede concluir que Keccak, JH y Grostl son las funciones Hash más adecuadas para ser implementadas en *hardware*, debido a que presentan la mejor relación *throughput*/área.

Referencias

ADVANCED ENCRYPTION STANDARD. [On-line] Avalaible at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

AL-SHAIKHLI, I. F, ALAHMAD, M. A., MUNTHIR, K. Hash Function of Finalist SHA-3: Analysis Study. In: International Journal of Advanced Computer Science and Information Technology (IJACSIT) Vol. 2, No. 2 (Apr., 2013) pp. 1-12, ISSN: 2296-1739.

AUMASSON, J. P., FISCHER, S., KHAZAEI, S., MEIER, W., and RECHBERGER, C. New features of Latin dances: analysis of Salsa, ChaCha and Rumba. In: FSE (2008).

AUMASSON, J.P., HENZEN, L., MEIER, W. and PHAN, R. C. SHA-3 proposal BLAKE, 2011.

BALDWIN, B.BALDWIN B., HANLEY N., HAMILTON M., LU L., BYRNE A., O'NEILL M. AND MARNANE W. P.. FPGA Implementations of the Round Two SHA-3 Candidates. In: The Second SHA-3 Candidate Conference (Aug., 2010).

BERTONI, G., DAEMEN, J., PEETERS, M., VAN ASSCHE, G. and VAN KEER, R. Cryptographic sponge functions. [On-line] Available at http://sponge.noekeon.org/ [Citado en enero, 2011].

BERTONI, G., DAEMEN, J., PEETERS, M., VAN ASSCHE, G. and VAN KEER, R. Keccak implementation overview. [On-line] Avalaible at http://keccak.noekeon.org/Keccak-implementation-3.2.pdf

FERGUSON, N. *et al*. The Skein Hash Function Family, 2011.

GAJ, K., HOMSIRIKAM, E. and ROGAWSKI, M. Comprehensive Comparison of Hardware Performance of Fourteen Round 2 SHA-3 Candidates with 512-bit Outputs Using Field Programmable Gate Arrays. In: The Second SHA-3 Candidate Conference (Aug., 2010).

GAURAVARAM, P. *et al*. Grostl-a SHA-3 candidate, 2011.

GUO, X., HUANG, S., NAZHANDALI, L. and SCHAUMONT, P. Fair and Comprehensive Performance Evaluation of 14 Second Round SHA-3 ASIC Implementations. In: The Second SHA-3 Candidate Conference (August 2010).

GUO, X. *et al*. Silicon Implementation of SHA-3 Finalists: Blake, Grostl, JH, Keccak and Skein. Center for Embedded Systems for Critical Applications (CESCA), Virginia Tech, 2011.

HENZEN, Luca, *et al*. Developing a hardware evaluation method for sha-3 candidates. In: Mangard and Standaert [21], pp. 248-263.

HOMSIRIKAMOL, E., ROGAWSKI, M. and GAJ, K. Comparing Hardware Performance of Round 3 SHA-3 Candidates using Multiple Hardware Architectures in Xilinx and Altera FPGA. George Mason University, 2011.

HOMSIRIKAMOL, E. Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGA. George Mason University, 2011.

HONGJUN, W. The Hash Function JH. Singapore, Institute for Infocomm Research y Nanyang Technological University, 2011.

KNOPF, C. Cryptographic Hash Functions. Leibniz Universität Hannover, in section 3.4 - The Compression Function (Nov., 2007).

NAMIN, A. H. and HASAN, M. A. Hardware Implementation of the Compression Function for Selected SHA-3 Candidates. Department of Electrical and Computer Engineering, University of Waterloo, Canada, 2011.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Announcing the Secure Hash Standard. In: Federal Information Processing Standards Publication 180-3 (Oct., 2008). [On-line] Avalaible at http://csrc.nist.gov/publications/PubsFIPS.html#fips180-4

TILLICH, S. *et al*. Uniform Evaluation of Hardware Implementations of the Round-Two SHA-3 Candidates. In: The Second SHA-3 Candidate Conference (Aug., 2010).