Implementación hardware de la función Hash SHA3-256 usando una arquitectura Pipeline

Hardware implementation of SHA3-256 Hash function using a Pipeline architecture

Nathaly Nieto Ramirez^{1*}

Rubén Darío Nieto Londoño¹

Recibido 8 de mayo de 2017, aceptado 21 de enero de 2018

Received: May 8, 2017 Accepted: January 21, 2018

RESUMEN

Las funciones hash son parte fundamental en aplicaciones relacionadas con seguridad informática, como los sistemas de autenticación y las firmas digitales. Muchas de estas aplicaciones requieren una alta velocidad de procesamiento, por lo que es necesario estudiar alternativas que disminuyan el tiempo de cálculo de estas funciones. Este trabajo presenta una arquitectura segmentada de dos etapas para la función hash SHA3-256 que soporta el procesamiento de uno o varios mensajes. Cada etapa ejecuta dos rondas del algoritmo en un ciclo de reloj, permitiendo un rendimiento de 27,98 Gbps en una FPGA de la familia Virtex-5 de Xilinx. Esta implementación permitió obtener una mejora en el rendimiento general de la función *KECCAK pipeline* de dos etapas con respecto a implementaciones de ronda iterativa e incluso de otras implementaciones *pipeline* multietapa.

Palabras clave: FPGA, función hash criptográfica, implementación hardware, KECCAK, pipeline, SHA-3.

ABSTRACT

Hash functions are a fundamental part of computer security applications such as authentication systems and digital signatures. Many of these applications require a high processing rate; therefore, it is important to explore alternatives that contribute to reduce the computation time of these functions. This paper presents a two-stage pipelined architecture for SHA3-256 hash function that supports single and multi-message processing. Each pipeline stage executes two rounds of the algorithm in one clock cycle. The proposed pipelined architecture has been implemented in Xilinx Virtex-5 and achieves a throughput 27,98 Gbps. This implementation allowed to obtain an improvement in the overall performance of the two-stage pipelined KECCAK function regarding implementations of iterative round and even other implementations multi-stage pipelined.

Keywords: Cryptographic hash function, FPGA, hardware implementation, KECCAK, pipeline, SHA-3.

---

¹ Escuela de Ingeniería Eléctrica y Electrónica. Universidad del Valle. Ciudad Universitaria Meléndez. Calle 13 # 100-00 Edificio 354, 1010. A.A 25360. Cali, Colombia. E-mail: nathaly.nieto@correounivalle.edu.co; ruben.nieto@correounivalle.edu.co

INTRODUCCIÓN

Con el surgimiento de nuevas tecnologías, aparecen nuevas amenazas contra la seguridad de los sistemas informáticos, y a pesar que existen herramientas de protección para los sistemas computacionales, como cortafuegos, antivirus, sistemas de detección de intrusos (*Intrusion Detection System: IDS*), estos no son totalmente efectivos y deben complementarse con otros sistemas de carácter externo. Lo anterior ha impulsado el uso de la criptografía, que emplea métodos y técnicas tanto *hardware* como *software* para proteger la información.

Las funciones *hash* son una de las herramientas usadas en criptografía para resolver el problema de integridad de mensajes, así como para determinar la autenticidad y el origen de los mismos. Estas funciones también se usan para firmar digitalmente documentos, asociándoles una cadena de longitud de tamaño fijo, lo que los hace más manejables y seguros.

Bruce Schneier en [1] manifiesta que “*es tiempo de migrar lejos del SHA-1*” después de hacer referencia a la colisión encontrada por un grupo de investigadores, la mayoría de la Universidad Shandong en China, [2]. Estos investigadores demostraron que se puede romper la seguridad de las funciones SHA-1 en menos de 2^{69} operaciones. Situaciones como esta hicieron que el *National Institute of Standards and Technology*, NIST, convocara una competencia para escoger la siguiente generación de funciones *hash*, SHA-3. En octubre de 2012 la función *KECCAK* fue elegida como ganadora y en agosto de 2015 se publicó el nuevo estándar de funciones criptográficas *hash* SHA-3, bajo el Federal Information Processing Standards Publication, FIPS 202 [3].

Muchas de las funciones *hash* criptográficas se construyen bajo un mismo principio de diseño que consiste en la aplicación iterativa de una función de compresión. El mecanismo de iteración de la función *KECCAK* se conoce como construcción esponja (*Sponge* en inglés) [4], como se muestra en la Figura 1.

La construcción esponja es una construcción de iteración simple en la que se aplica una función de entrada de longitud variable obteniendo una salida de longitud arbitraria basada en una permutación de longitud fija y operando un número fijo de bits. El núcleo del

algoritmo se basa en una función de permutación f que se aplica repetidamente a un estado de longitud fija de $b = r + c$ bits. El parámetro r se conoce como la tasa de bits y el parámetro c como la capacidad. Los valores más altos de r mejoran la velocidad y valores mayores de c su nivel de seguridad [5].

En el flujo de procesamiento de los datos, los b bits de estado se inicializan en cero y se obtiene el producto de la construcción esponja en dos fases, la primera es la de absorción seguida por la fase de compresión. En la fase de absorción se realiza la operación lógica XOR de los r bits de cada bloque de mensaje con los primeros r bits del bloque de estado y después se aplica la función f . Cuando todos los bloques de mensaje han sido procesados, se pasa a la siguiente fase. En la fase de compresión los primeros r bits de estado se toman como parte de bloque de salida Z_i . Si se requieren más bloques de salida de r bits, se aplica iterativamente la función f hasta lograr los bloques de r bits deseados. Los c bits restantes no son afectados directamente por los bits del mensaje, ni se toman como salida; estos bits determinan el nivel de seguridad de la construcción [4].

Figura 1. Construcción Esponja [4].

El rendimiento de las implementaciones de las funciones *hash* iterativas, ya sean hardware o software, disminuye cuando aumenta la cantidad de datos que se tienen en la entrada, esto debido a que los bloques de mensajes se procesan de forma secuencial y a que para el procesamiento de un nuevo bloque de mensaje se requiere que todos los bloques anteriores hayan sido procesados [6]. Esto lleva a plantear la necesidad de concebir un sistema hardware digital que haga uso de nuevas estrategias de paralelismo para calcular el resumen del mensaje.

Existen trabajos sobre implementaciones hardware de los algoritmos candidatos a ser SHA-3. La mayoría de estos presentan resultados de implementaciones iterativas en las que el objetivo principal es obtener un mayor rendimiento [7–10], o un diseño compacto para reducir el área ocupada en hardware [11–13]. Otras implementaciones como [9, 14–17] presentan

un enfoque *pipeline* básico en las que se busca mejorar el rendimiento por área.

Este trabajo presenta resultados de la implementación de una estructura de interconexión que aprovecha los niveles de paralelismo en hardware ofrecidos por el estándar *SHA-3* a la vez que propone una arquitectura segmentada (*pipeline*) que mejora el rendimiento con respecto a las implementaciones iterativas y facilita el procesamiento de varios mensajes de forma paralela (*Multi-Message Hashing: MMH*).

Este artículo se ha organizado de la siguiente manera: en la sección dos resume los trabajos previos encontrados de implementaciones *pipeline* del algoritmo *KECCAK*. La tercera sección describe brevemente el algoritmo *KECCAK*. En la sección cuatro se expone la arquitectura propuesta y en la sección cinco se presentan los resultados de la implementación usando una FPGA de la familia Virtex-5.

TRABAJOS PREVIOS

En [9] los autores presentan una comparación de todos los candidatos a la ronda 3 *SHA-3* y la norma *SHA-2*. *KECCAK* fue implementado usando tres arquitecturas, la básica iterativa, básica iterativa con dos etapas *pipeline* y básica iterativa usando cuatro etapas *pipeline*, Figura 2. El mejor resultado de rendimiento lo obtuvo la implementación básica iterativa con dos etapas *pipeline* con 15,362 Gbps.

En la arquitectura de ronda iterativa se implementa una sola ronda de la función de compresión *KECCAK*. Esta es la manera más simple de implementación ya que existe una relación directa entre el hardware y las operaciones de internas de la función de compresión.

Figura 2. Arquitecturas implementadas por los autores en [9] (a) Básica iterativa, (b) básica iterativa con dos etapas *pipeline*, (c) básica iterativa con cuatro etapas *pipeline*.

En la arquitectura iterativa con varias etapas de *pipeline*, la lógica combinacional de una ronda se divide en el número de etapas que se tenga. Para calcular el tiempo requerido para procesar un solo bloque de mensaje de las arquitecturas implementadas, los autores utilizan las relaciones de la Tabla 1. Notación: *b* - tamaño de bloque, *r* - número de rondas, *T*-periodo de reloj.

Tabla 1. Procesamiento de los mensajes en la organización propuesta por [9].

Arquitectura	Tiempo necesario para procesar un solo mensaje	Rendimiento
Básica Iterativa	$T_{Blk} = r \cdot T$	$T_p = b/T_{Blk}$
Iterativa con 2 etapas <i>pipeline</i>	$T_{Blk} = (2 \cdot r) \cdot T$	$T_p = 2 \cdot b/T_{Blk}$
Iterativa con 4 etapas <i>pipeline</i>	$T_{Blk} = (4 \cdot r) \cdot T$	$T_p = 4 \cdot b/T_{Blk}$

El trabajo de [14] propone una implementación *pipeline* de *KECCAK* que consta de cuatro etapas. Cada una de las funciones de transformación, θ , $\rho\pi$, χ , ι , está representada por una etapa del *pipeline*. En el diseño presentado en este trabajo, se operan simultáneamente cuatro bloques de datos mediante un buffer asíncrono que realiza el procesamiento dependiendo de la iteración que se esté ejecutando. En cada ciclo se procesa un bloque de datos diferente por lo que para el ciclo 4 ya han ingresado los cuatro bloques de datos y el primer bloque ha terminado la operación *iota* (ι). La arquitectura fue implementada en una FPGA Virtex 5 obteniendo un rendimiento de 7,7 Gbps.

Los autores de [16] reportan la implementación una arquitectura que soporta los cuatro modos de operación del estándar *SHA-3*, *SHA3-224*, *SHA3-256*, *SHA3-384* o *SHA3-512*. La arquitectura propuesta implementa un *pipeline* de dos etapas en el que se ejecuta una ronda por etapa y es capaz de manejar un solo mensaje o múltiples bloques de mensajes. Los autores reportan los resultados de rendimiento de 16,896 Gbps para el modo *SHA3-512* usando la FPGA Virtex-5 XC5VLX330T.

En [17] se evalúan seis organizaciones *pipeline* y dividen la función θ en funciones intermedias θ_1 , θ_2 , θ_3 . En cada una de las seis organizaciones la coma (,) indica la separación de las etapas *pipeline*. Las seis organizaciones que se evalúan son:

- P2:** $\theta_1 \theta_2 \theta_3$ y $\varrho \pi \chi \iota$;
P3: $\theta_1 \theta_2 \theta_3$, $\varrho \pi$ y $\chi \iota$;
P4: θ_1 , $\theta_2 \theta_3$, $\varrho \pi$ y $\chi \iota$;
P5: θ_1 , θ_2 , θ_3 , $\varrho \pi$ y $\chi \iota$;
P6: θ_1 , θ_2 , θ_3 , $\varrho \pi$ χ e ι ;
P7: θ_1 , θ_2 , θ_3 , ϱ , π , χ e ι .

Según los autores, P2, P3 y P4 presentan un mayor rendimiento por unidad de área que la organización propuesta por Akin y otros en [15], que fue nombrada como P5, Figura 3. Se nota que el rendimiento aumentó a medida que el número de etapas *pipeline* también lo hizo, hasta llegar a un máximo en P4.

Figura 3. Organización pipeline propuesta por Akin y otros (P5) en [15].

EL ALGORITMO KECCAK

El estándar FIPS 202 [3], define la familia de funciones hash criptográficas *SHA3-224*, *SHA3-256*, *SHA3-384* y *SHA3-512*, en estas, el número después del guion indica el tamaño del resumen. Este estándar también especifica la permutación *KECCAK*, que actúa como núcleo de la función *hash* *SHA-3*. *KECCAK* es una familia de funciones esponja con miembros *KECCAK* [r, c]. La suma determina el ancho de la permutación *KECCAK-f* y se encuentra limitada a los valores {25, 50, 100, 200, 400, 800, 1600}. Estas permutaciones son construcciones iteradas que consisten en una secuencia de rondas casi idénticas. El número de rondas, N_r , depende del ancho de la permutación, y está dado por $N_r = 12 + 2^\ell$, donde $2^\ell = b/25$. Esto indica que para la función *KECCAK-f* se requieren 24 rondas.

Una ronda *KECCAK-f* consta de una secuencia de cinco transformaciones identificadas por letras griegas: θ (theta), ϱ (rho), π (pi), χ (chi), ι (iota), ver Figura 4. La permutación se especifica mediante un arreglo de bits, llamado estado. A su vez, cada estado se encuentra organizado como un arreglo de líneas de tamaño w . En la Tabla 2 se definen los valores posibles para el estado con los respectivos tamaños de w y e .

Figura 4. Estructura interna de la función *KECCAK-f* [18].

Tabla 2. Anchos de la permutación *KECCAK-p* según el tamaño de estado [3]

b	25	50	100	200	400	800	1600
w	1	2	4	8	16	32	64
ℓ	0	1	2	3	4	5	6

El Algoritmo 1 describe las funciones de transformación de *KECCAK-f* usando las siguientes convenciones:

- Todas las operaciones de las funciones de transformación emplean aritmética módulo 5.
- La matriz de estado completa se identifica como A y $A[x, y]$ es una línea particular en ese estado.
- $B[x, y]$, $C[x]$ y $D[x]$ son variables intermedias.
- El símbolo $\oplus$ corresponde a la función OR y *ROT* (W, r) se refiere la operación de rotación de W un total de r veces.

Algoritmo 1. Funciones de Transformación de Ronda

Round[b](A, RC)

Transformación θ

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

$$D[x] = C[x - 1] \oplus \text{ROT}(C[x + 1], 1), \quad \forall (x) \in (0 \dots 4)$$

$$A[x, y] = A[x, y] \oplus D[x], \quad \forall (x, y) \in (0 \dots 4, 0 \dots 4)$$

Transformaciones ρ y π

$$B[y, 2x + 3y] = \text{ROT}(A[x, y], r[x, y]), \quad \forall (x, y) \in (0 \dots 4, 0 \dots 4)$$

Transformación χ

$$A[x, y] = B[x, y] \oplus (\text{NOT } B[x + 1, y]) \text{ AND } B[x + 2, y], \quad \forall (x, y) \in (0 \dots 4, 0 \dots 4)$$

Transformación ι

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

Return A

Función Theta (θ)

La función *theta* calcula la paridad de cada una de las de $5 \times w$ líneas (320, cuando $w=64$), y realiza la operación XOR con las dos columnas más cercanas (columnas de la izquierda de la parte delantera bits $(x-1)$ y derecha $(x+1, z-1)$). En la Figura 5 se muestra el diagrama de bloques de la función *theta* a nivel de bit.

Funciones Rho (ρ) y Pi (π)

La permutación *rho* rota los bits de cada línea en un valor constante conocido como *offset*, el cual depende de los valores que tomen las coordenadas x y y . La Tabla 3 especifica las 25 constantes de rotación, $r[x,y]$, para $w = 64$.

Tabla 3. Constantes $r[x,y]$ [3]

	$x = 3$	$x = 4$	$x = 0$	$x = 1$	$x = 2$
$y = 2$	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

Figura 5. Diagrama de Bloques de la Función *Theta*.

La función *pi*, por su parte, reorganiza la posición de las líneas usando un patrón fijo. Por su naturaleza, tanto *rho* como *pi* son funciones de transformación que no requieren *hardware*.

Función Chi (χ)

La Función *chi* es una combinación bit a bit a lo largo de las filas. *Chi* es el único mapeo no lineal en

la permutación *KECCAK-f*. Sin ella, *KECCAK-f* sería lineal. La Figura 6 muestra el diagrama de bloques de función *chi*.

Figura 6. Diagrama de bloques de la función *Chi*. [3]

Función Iota (ι)

La Permutación *iota* consiste en la adición de constantes de ronda y está dirigida a perturbar la simetría. Los valores de las constantes de ronda $RC[i]$ se encuentran definidos en [19].

Para obtener el resultado de la función *iota* se realiza la operación XOR entre las constantes RC y los w bits de la palabra (0,0) del estado inmediatamente anterior.

ORGANIZACIÓN PROPUESTA

El algoritmo *KECCAK* presenta dependencia de datos pues el resultado del procesamiento de cada una de las funciones de transformación ($\theta, \rho, \pi, \chi, \iota$) depende del resultado de la función que se ha procesado en el estado anterior, Figura 7. El presente trabajo tiene en cuenta esta característica y propone una organización que integra un diseño *pipeline* de dos etapas con la técnica de implementación *Loop Unrolling*; lo anterior con el fin de aumentar el rendimiento y también permitir el procesamiento paralelo de diferentes bloques de mensajes.

Figura 7. Funciones de transformación del algoritmo *KECCAK*

La transformación *Loop Unrolling* con un factor n , expande el núcleo de la función reproduciéndolo en

$n-1$ iteraciones. Esta aproximación se utiliza cuando se desea aumentar el rendimiento de un solo mensaje largo (mensaje que es mayor al doble del tamaño del *rate*, es decir, que tiene más de dos bloques de mensaje [20]), mientras que la segmentación adiciona paralelismo al permitir el cálculo de las transformaciones de ronda de múltiples mensajes o de múltiples bloques de mensaje de manera simultánea.

En el diseño propuesto, la lógica combinacional de una ronda se replica en un factor de cuatro y se agregan dos registros para segmentación después de la segunda y cuarta ronda. Con esto se logrará aumentar el rendimiento total con respecto a la arquitectura básica iterativa ya que disminuirá del número de ciclos de reloj para calcular el resumen.

La funcionalidad principal de la estructura propuesta está contenida en las cinco transformaciones (θ , ρ , π , χ , ι) de la matriz de estado. Como se mencionó anteriormente, esta implementación consta de dos etapas *pipeline*. Cada etapa contiene un núcleo de procesamiento que ejecuta dos rondas del algoritmo *KECCAK*, una ronda par y una ronda impar, ya que el área de cada ronda se ha replicado, por lo tanto en cada iteración el número de las rondas que se ejecutan varía de dos en dos. Ver la Figura 8.

En la Figura 9 se puede observar el camino de datos de la estructura propuesta. El registro de salida del primer núcleo se utiliza para alimentar la entrada del segundo núcleo una vez procesadas las seis iteraciones del primer núcleo. El control del número de iteraciones necesarias para el procesamiento de datos y el manejo de las constantes de ronda se realiza mediante la implementación de un contador de 3 bits para cada núcleo. Los multiplexores seleccionan la entrada de cada núcleo; en el caso del primer núcleo selecciona entre el resultado de la operación XOR

entre el mensaje rellenado y el estado anterior o el *Vector Inicial* (VI) y el resultado de la ronda anterior, y en el caso del núcleo 2 entre el resultado de la ronda anterior o la salida del primer núcleo.

El tamaño de los buses de entrada del mensaje rellenado y del resumen es de 64 bits por ser el tamaño del *lane* para *KECCAK* y para no exceder el número de pines de entrada salida de que dispone en la FPGA. Además se agregan los registros SIPO (*Serial-in-Parallel-out*) y PISO (*Parallel-in-Serial-out*) para el manejo de entradas y salidas. En la versión SHA3-256, un bloque de mensaje de 1088 bits tarda 17 ciclos de reloj en estar disponible para su procesamiento.

El procesamiento se realiza de la siguiente manera: el mensaje M_1 ingresa al primer núcleo para efectuar las operaciones desde la ronda 0 a la 11. De la ronda 12 a la 23, M_1 se procesa en el segundo núcleo, al tiempo que M_2 ingresa al primer núcleo para ejecutar las rondas 0 a 11. En la Tabla 4 se muestra el procesamiento de los mensajes en cada núcleo. El resultado del mensaje M_1 se obtiene después de 12 ciclos de reloj y el resultado para el mensaje M_2 y los siguientes mensajes, se obtendrán 6 ciclos de reloj después del resultado inmediatamente anterior. Para el caso de un mensaje compuesto por diferentes bloques, el procedimiento sigue siendo el mismo, la diferencia radica en la fase de compresión y la función *Truncated* según como se define en el algoritmo de la función *Sponge* [5].

Tabla 4. Procesamiento de los mensajes en la organización propuesta.

Cant. Rondas	12 rondas						12 rondas					
Clk	1	2	3	4	5	6	7	8	9	10	11	12
Núcleo 1	M_1						M_2					
Núcleo 2	Vacío						M_1					

Figura 8. Funciones de transformación del algoritmo *KECCAK* en cada núcleo de la organización propuesta.

Figura 9. Camino de datos para el diseño propuesto.

RESULTADOS OBTENIDOS

La organización presentada ha sido especificada en VHDL utilizando Xilinx ISE 14.7 y usando la FPGA Virtex-5 XC5VLX110T-1ff1136. El funcionamiento de los diseños fueron completamente verificados mediante un banco de pruebas empleando los vectores de prueba (KAT, *Known Answer Test*), distribuidos por los autores en su página web [21].

En la Tabla 5 se presenta un resumen de los resultados obtenidos del diseño propuesto y trabajos previos reportados en la literatura de implementaciones *pipeline* de la función KECCAK en FPGAs Virtex 4 y 5, indicando frecuencia, área, rendimiento y la relación rendimiento/área de la función SHA-3 en sus variantes 256 y 512 bits. El rendimiento de la propuesta realizada en este trabajo se calcula utilizando la ecuación (1):

$$Rendimiento = \frac{No.Bits}{No.Ciclos} \times f, \quad (1)$$

Donde:

- *No. Bits*: número de bits del mensaje que están siendo procesados por la función *hash*.
- *No. Ciclos*: ciclos de reloj requeridos entre mensajes sucesivos para obtener el resultado.
- *f*: la frecuencia de operación del diseño.

Como se puede observar, el diseño propuesto (*Pipeline de dos etapas + Unrolling x4*) presenta un aumento significativo en el área ocupada, lo cual es compensado con un factor de aumento de 1,82 en el rendimiento del circuito con respecto del diseño *pipeline* de 2 etapas [9] y en un factor de 2,19 con respecto de la implementación de ronda iterativa [9], ambas realizadas en una Virtex 5. El rendimiento del diseño propuesto aumenta un factor de 1,25 con relación a la implementación *pipeline* de 5 etapas-MMH [15], ya que el número de ciclos requerido para procesar un mensaje ha disminuido de 25 a 6 ciclos.

La propuesta [17] incluye gráficos comparativos entre seis organizaciones diferentes y la planteada

Tabla 5. Resultados de SHA3-256 [9], [14–16].

Diseño	Dispositivo	Área Slices	Frecuencia (MHz)	Rendimiento (Gbps)	Rendimiento/Área (Mbps/Slice)
Ronda Iterativa [9]	XC5VLX110T	1.395	-	12,78	9,16
Pipeline 4 Etapas [14]	XC5VLX50	3.117	452,00	7,70	2,47
Pipeline 2 Etapas [9]	Virtex 5	1.980	-	15,36	7,76
Pipeline 4 Etapas [9]	Virtex 5	3.849	-	12,65	3,29
Pipeline x2 Etapas + Unrolling x2[16]*	XC5VLX330T	1.581	382,00	9,17	5,80
Pipeline 2 Etapas, SHA3-512 [16]*	XC5VLX330T	2.652	352,00	16,90	6,40
Ronda Iterativa-SMH [15]	Virtex 4	2.024	143,00	6,07	3,00
Pipeline 5 Etapas-MMH [15]	Virtex 4	4.356	509,00	22,33	5,13
Pipeline x2 + Unrolling x4 MMH	XC5VLX110T	5.029	154,32	27,98	5,56

*Implementación para la variante de 512-bits

por los autores del algoritmo Keccak, pero no presenta valores precisos que permitan realizar un análisis. Sin embargo, de las gráficas se puede observar que el rendimiento de las diferentes organizaciones desarrolladas no superan los 7 Gbps, un rendimiento mucho menor que el de este trabajo, 27,98 Gbps.

Se resalta el hecho que al aumentar el área del circuito esta se ve afectada en un factor de 3,6 y la relación rendimiento/área en un factor de 1,65 con respecto de la implementación básica de ronda iterativa, no obstante, el porcentaje de utilización de recursos en la FPGA no supera el 30%. Se debe tener en cuenta que al replicar la lógica combinacional de una ronda, el área total del circuito aumenta en un factor cercano al factor de *unrolling*. Por tal razón, este tipo de arquitecturas son útiles cuando el objetivo es aumentar la velocidad y también procesar mensajes largos, es decir, mayores a 2.176 bits para el caso de SHA3-256, siempre que no se tengan limitaciones de área.

CONCLUSIONES

Se presentan los resultados de implementación de la función *hash* SHA3-256 en una configuración *pipeline* de dos etapas en la que cada etapa ejecuta dos rondas. Esta implementación es muy útil en aplicaciones de procesamiento de múltiples bloques de mensaje y múltiples mensajes. En la estructura *pipeline* propuesta se pueden procesar dos bloques del mismo mensaje o dos mensajes diferentes simultáneamente. Cada etapa *pipeline* ejecuta las operaciones de dos rondas en un ciclo de reloj, logrando obtener el resumen de un mensaje 6 ciclos de reloj después de obtener el resultado del mensaje inmediatamente anterior. El diseño propuesto fue implementado usando la FPGA Virtex 5 (XC5VLX110T) de Xilinx. Esta implementación permitió obtener una mejora en el rendimiento general de la función *KECCAK pipeline* de dos etapas con respecto de la de [9], aumentándola de 15,36 Gbps a 27,98 Gbps, lo que representa una mejora del 82,16%.

AGRADECIMIENTOS

El presente trabajo de investigación fue realizado gracias al apoyo de la Universidad del Valle con la financiación del proyecto “Análisis de Desempeño de Estrategias de Implementación del Estándar

SHA3-256 Usando Hardware Reprogramable,” Código: 2893.

REFERENCIAS

- [1] B. Schneier. “Schneier on Security: Cryptanalysis of SHA-1.” *SHA-1 Broken*, 2005. Online. Available: <http://www.schneier.com/>.
- [2] X. Wang, Y. L. Yin and H. Yu. “Finding Collisions in the Full SHA-1,” *Adv. Cryptol. – CRYPTO 2005*. N° 90304009, pp. 17–36. 2005.
- [3] National Institute of Standards and Technology (NIST). “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions,” Gaithersburg, MD. Julio, 2015.
- [4] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche. “Cryptographic Sponge Functions”. 2011.
- [5] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche. “The Keccak Sponge Function Family”. 2009.
- [6] G. Bertoni, J. Daemen, P. Michael and G. Van Assche. “Sufficient conditions for sound tree and sequential hashing modes,” *IACR Cryptol. ePrint Arch.*. Vol. 2009, pp. 210. 2009.
- [7] B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O’Neill and W. P. Marnane. “FPGA Implementations of the Round Two SHA-3 Candidates”. *F. Program. Log. Appl. FPL 2010 Int. Conf.*. Vol. 3, pp. 400–407. 2010.
- [8] S. Tillich, M. Feldhofer, M. Kirschbaum, T. Plos, J.-M. Schmidt and A. Szekely. “High-Speed Hardware Implementations of BLAKE, Blue Midnight Wish, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein,” *eprint.iacr.org*. 2009. Online. Available: <http://eprint.iacr.org/2009/510.pdf>.
- [9] E. Homsirikamol, M. Rogawski and K. Gaj. “Comparing Hardware Performance of Round 3 SHA-3 Candidates using Multiple Hardware Architectures in Xilinx and Altera FPGAs,” *Ecrypt II Hash Work*. N° 60. 2011.
- [10] K. Latif, M. M. Rao, A. Mahboob and A. Aziz. “Novel Arithmetic Architecture for High Performance Implementation of SHA-3 Finalist Keccak on FPGA Platforms,” in

- Reconfigurable Computing: Architectures, Tools and Applications*. pp. 372–378. 2012
- [11] S. Kerckhof, F. F. Durvaux, N. Veyrat-Charvillon, F. Regazzoni, G. M. de Dormale and F.-X. Standaert. “Compact FPGA implementations of the five SHA-3 finalists,” *Smart Card Res. Adv. Appl.*. Vol. 7079, pp. 217–233. 2011.
- [12] B. Jungk and J. Apfelbeck. “Area-Efficient FPGA Implementations of the SHA-3 Finalists,” in *IEEE International Conference on Reconfigurable Computing and FPGAs*. pp. 235–241. 2011
- [13] N. M and B. Mrs. A. “Area-Efficient FPGA Implementation of Cryptographic SHA3-512,” in *International Journal of Engineering Trends and Technology (IJETT)*. Vol. 21 N° 9, pp. 455–460. 2015
- [14] F. D. Pereira, E. Ordonez, I. D. Sakai and A. M. SE Sourz. “Exploiting Parallelism on Keccak : FPGA and GPU Comparison,” in *Parallel & Cloud Computing 2.1*. Vol. 2, pp. 1–6. 2011
- [15] A. Akin, A. Aysu, O. C. Ulusel and E. Savaş. “Efficient hardware implementations of high throughput SHA-3 candidates keccak, luffa and blue midnight wish for single- and multi-message hashing,” in *Proceedings of the 3rd international conference on Security of information and networks - SIN '10*. pp. 168. 2010
- [16] L. Ioannou, H. E. Michail and A. G. Voyiatzis. “High Performance Pipelined FPGA Implementation of the SHA-3 Hash Algorithm,” in *IEEE 4th Mediterranean Conference on Embedded Computing (MECO)*. pp. 68–71. 2015
- [17] Y. Ayuzawa, N. Fujieda and S. Ichikawa. “Design Trade-offs in SHA-3 Multi-Message Hashing on FPGAs,” in *TENCON 2014 - 2014 IEEE Region 10 Conference*. pp. 1–5. 2014
- [18] C. Paar and J. Pelzl. “SHA-3 and The Hash Function Keccak,” in *Understanding Cryptography — A Textbook for Students and Practitioners*. Springer Berlin Heidelberg. 2010.
- [19] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche. “The keccak reference,” *Submiss. to NIST (Round 3)*. pp. 1–14. 2011.
- [20] S. S. Keller and L. E. Bassham Iii. “The Secure Hash Algorithm 3 Validation System (SHA3VS)”. 2016.
- [21] Guido Bertoni, J. Daemen, M. Peeters, G. Van Assche and R. Van Keer. “Team Keccak.” Online. Available: <https://keccak.team/index.html>.