Informe de Proyecto IPD432

Diego Facundo Lazcano 201321059-7 - Benjamin Opazo 201321022-8

I. Introducción

EN este informe se detalla el desarrollo del proyecto final del ramo IPD432. Inicialmente se buscaba implementar un minador de *criptomonedas* en FPGA, en particular de Bitcoin, sin embargo, a medida que se desarrolló el proyecto, y por temas de complejidad, se decidió sólo implementar la función SHA256, obteniendo como resultado un dispositivo de *hashing* que funciona como esclavo de otra maquina, por ejemplo, un minador de bitcoins.

Actualmente, los algoritmos de *hashing* son usados en una gran variedad de tecnologías y procesos [2], como por ejemplo,

- **Blockchain**: Las funciones *hash* se utilizan como bases de la tecnología, la que se utiliza ampliamente en las *criptomonedas*. [5]
- **Linux**. Se utiliza SHA256 para verificar que algunos instaladores o procesos estén en correcto estado. [1]
- **Ciberseguridad**: Se utiliza ampliamente las funciones SHA1, MD5, SHA2, etc.

SHA256 (Secure Hash Algorithm) corresponde a una función *hash* desarrollada por la NSA (National Security Agency) en Estados Unidos ([6] cap. 2). 256 corresponde a la cantidad de bits que tiene la salida, y es parte de la familia de funciones SHA-2. El algoritmo tiene propiedades que permiten su uso de forma segura, por ejemplo, su salida es determinística en función de la entrada, es decir para una entrada x siempre se obtiene una salida $h(x) = y$, y más importante, el algoritmo presenta resistencia frente a colisiones, esto quiere decir, que no existen métodos conocidos para encontrar dos entradas, x y z, que tengan las misma salidas $h(z) = h(x)$ [4]. En contraste, para el algoritmo SHA1 ya se han encontrado colisiones [8].

Se comentará la motivación detrás del trabajo en la primera sección, en la segunda sección se explicará el funcionamiento del diseño con tal de entenderlo como caja negra, y los primeros pasos para que un posible futuro usuario busque hacer mejoras, luego en la tercera sección se profundizará en la implementación del módulo *sha256*. En la cuarta sección se detalla el desempeño del circuito diseñadoen términos de latencia, throughput y uso de potencia. Finalmente en las conclusiones se comenta sobre los desafíos a la hora de trabajar en el proyecto, las razones de las decisiones más importantes, y futuras propuestas para mejorar el proyecto.

Junio, 2018

II. Motivación

El algoritmo SHA56, así como una gran cantidad de funciones *hash*, contiene muchas operaciones internas que están enfocadas al manejo directo de bits que no son necesariamente múltiplos de 8. En este contexto, y por la alta granularidad de las FPGA, es conveniente implementar una algoritmo con estas características en FPGA, y no en CPU. En consecuencia, debido a la naturaleza de las operaciones internas, las FPGAS sobre las CPU pueden tener un mayor desempeño en función de ciclos de reloj, y por la forma en que se tratan los bits y sus operaciones, es mas sencillo hacer la implementación en FPGA.

Para ello se busca implementar un módulo de cálculo de SHA256, que funcione como caja negra, y que logre en pocos ciclos calcular el *hash*. Por otra parte, como se busca que el módulo funcione como caja negra, es necesario generar un protocolo de comunicación con el exterior, y que este sea lo más sencillo posible, y de esta forma, es factible implementar muchos módulos de *hashing* en paralelo, o portar el módulo a otros proyectos.

Otro punto importante dentro del proyecto es generar el mecanismo adecuado de control para los módulos de *hash* (módulo maestro), pues el diseño considera que funcionan como esclavos, y a su vez, este maestro funciona como interfaz entre la FPGA y alguna máquina externa que pide los *hash* para entradas de largo arbitrario. Nótese que el mecanismo de control no es obligatorio para el uso del módulo de *hashing*, pero se implementó para que algún usuario futuro no tenga problemas al usarlo.

Por lo anterior, el hardware de control debe ser capaz de recibir una cadena de mensajes de largo arbitrario para aplicarle *hashing*, guardar estas cadenas de mensajes, y comunicarse con el módulo de *hash* para pedir los resultados. El hardware de control no guarda ni procesa los resultados entregados por el módulo de *hash*.

III. Funcionalidad

Se explicará brevemente el funcionamiento general del diseño de los módulos *sha256* y *driver_ram*.

III-A. sha256

Este módulo fue diseñado de tal forma de que sea fácilmente portable a otros proyectos, esta parametrizado casi en su totalidad, de forma que es posible adaptarlo a distintas aplicaciones, dependiendo de los recursos que tenga la FPGA con la que se trabaje.

Los parámetros del módulo son:

- **N_length**: Cantidad de bits necesarios para representar el largo máximo de la entrada. Por ejemplo, si la entrada máxima que acepta el diseño es 1023, entonces N_length es 10, pues con 10 bits es posible representar los largos de 0 a 1023.

- **N_N**: Cantidad de bits necesarios para representar el N máximo (N_Max) del diseño.
- **MAX**: Largo Máximo del mensaje, que corresponde a $2^{\text{N\_length}} - 1$
- **N_MAX**: Cantidad máxima de bloques M de 512 bits que se crearán. Este valor depende del largo máximo y se calcula con la siguiente formula:

$$N\_MAX = [MAX/512] + f(MAX\,\%512)$$

Donde:

$$f(x) = \begin{cases} 1 & ; x + 1 < 448 \\ 2 & ; c.o.c. \end{cases}$$

Y, $[\cdot]$ corresponde a la función parte entera

El módulo tiene las siguientes banderas

- **start** (input): Se inicia el proceso cuando el maestro levanta esta bandera.
- **data_ready** (input): Indica que los datos que están en el bus de datos de entrada están listos para que el módulo los lea.
- **new_data** (output): El módulo pide nuevos datos. Esto se debe a que el largo de los datos que han entrado hasta el momento es menor al valor *length*.
- **hash_ok** (output): El módulo indica que termino de calcular el hash, y los datos que están en *hash* corresponden a la salida correctamente.
- **hash_save** (input): El maestro ya usó los datos de la salida *hash* y se puede empezar el proceso de nuevo (borrando lo que está en *hash*).

De forma general, el módulo permanecerá en espera hasta que se levante la bandera de *start*. Una vez levantada la bandera, el módulo leerá el largo del mensaje a través de *length*, este valor deberá mantenerse hasta que el módulo termine de trabajar. Si el largo de la entrada es mayor a 512, el módulo tendrá que pedir más de una vez los datos de entrada, pues el bus para ingresar los datos es de 512 bits, en consecuencia, el módulo levantará la bandera *new_data*, en ese momento se debe actualizar el bus *data* con los siguientes 512 bits correspondientes al mensaje de entrada. Una vez que los datos que están en el bus *data* son los que quiere ingresar al módulo, se debe levantar la bandera *data_ready*. Este proceso se debe repetir hasta que se ingrese el mensaje completo (según *length*).

Se ingresan los datos desde el bit más significativo al menos significativo, eso quiere decir, que los últimos datos que se entregan al módulo *sha256* corresponden a los menos significativos. Si faltan menos de 512 bits por entregar, el módulo desechará los bit menos significativos del último mensaje.

La bandera *hash_ok*, indica que el hash ya esta listo, y el módulo mantendrá la salida del hash hasta que se levante la bandera *hash_save*, que indica que ya se guardaron los datos que están en la salida.

A modo de ejemplo, si se quiere calcular la función hash de un mensaje de largo 520 bits, la secuencia de banderas y datos sería la siguiente:

1. El maestro carga en *length* el valor 520
2. El maestro levanta la bandera *start*
3. El módulo levanta la bandera *new_data*
4. El maestro carga los 512 bits mas significativos del mensaje en el bus *data* y levanta la bandera *data_ready*
5. El módulo levanta nuevamente la bandera *new_data*, pues aún faltan 8 bits por cargar
6. El maestro carga los 8 bits **menos significativos de su mensaje** (los bits que faltan) en los 8 bits **más significativos del bus `data`** y levanta la bandera *data_ready*. Los 504 bits menos significativos funcionan como *don't care*.
7. El módulo calcula el hash, y levanta la bandera *hash_ok*, al mismo tiempo, en el bus *hash* se carga el resultado de la operación.
8. El maestro guarda el resultado y levanta la bandera *hash_save*
9. El módulo borra el resultado de *hash* y se queda en espera de un nuevo hash, con un nuevo largo.

Es importante destacar que los datos que se quieren cargar en el hash deben permanecer ahí al menos 4 ciclos (incluso si el módulo ya levanto la bandera *new_data*), pues la propagación de los datos no es instantánea.

III-B. driver_ram

Se diseñó este módulo con dos objetivos, primero, entregarle al usuario final un driver listo para hacer funcionar el módulo *sha256*, dado que los datos que se quieren cargar y sus largos respectivos están guardados cada uno en una Block RAM Dual Port distinta, y en segundo lugar, para poder medir el rendimiento del módulo sha256.

Este módulo tiene una máquina de estados hecha a la medida para comunicarse correctamente con el módulo *sha256*, es por eso que tiene como entradas y salidas todas las banderas del módulo *sha256*, excepto *hash_save*, considerando que el usuario final puede tener distintos usos con el resultado de la operación. El driver se comienza levantando la bandera *driver_start*.

El módulo no está parametrizado, sin embargo, es fácilmente modificable para las necesidades específicas del usuario final. Además, el módulo asume que:

- Se tienen dos memorias RAM, una que contiene los datos para calcular, y la otra contiene los largos de cada mensaje.
- La memoria RAM que contiene los datos, tiene 512 bits de ancho, y tiene los datos guardados secuencialmente en el orden que se cargaran en el módulo *sha256*.
- La memoria RAM que contiene los largos tiene un ancho igual o mayor que *N_length*. La salida de la RAM debe ingresarse correctamente al módulo *sha256*. Los largos están guardados secuencialmente en el orden que se utilizarán.

Nótese que las direcciones entre la memoria RAM de los datos y de los largos no necesariamente se corresponden uno a uno. Si quiere calcular el hash de dos mensajes, el primero de

520 bits de largo y el segundo de 528, entonces las direcciones de las memorias RAM serán las siguientes

Ram de Datos

- Dirección 0: 512 bits más significativos del **primer** mensaje.
- Dirección 1: 8 bits menos significativos del **primer** mensaje en los bits más significativos del bloque.
- Dirección 2: 512 bits más significativos del **segundo** mensaje.
- Dirección 3: 16 bits menos significativos del **segundo** mensaje en los bits más significativos del bloque.

Ram de Largos

- Dirección 0: Largo del **primer** mensaje (520)
- Dirección 1: Largo del **segundo** mensaje (528)

IV. Implementación

El diagrama del módulo *sha256* se muestra en la Figura 1, y el detalle de la implementación del módulo se puede consultar en la Figura 2.

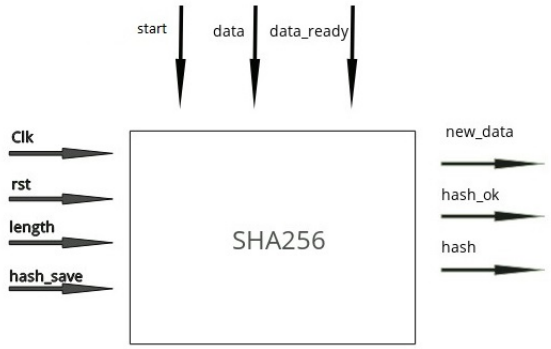

Figura 1. Modulo Sha256 visto por fuera

El módulo de *sha512* se divide en una serie de submódulos, que están relacionados a los subalgoritmos (o procesos internos) que tiene la función de hashing, un módulo controlador y funciones básicas especificas del hash (Ch, Maj, RotR, Sig0, Sig1, Sig_chi0, Sig_chi1) las que se pueden consultar en [7].

El submódulo *N_Calculator*, calcula la cantidad N de bloques M de 512 bits para desarrollar en el hashing. El submódulo depende del largo del mensaje *n_length* y es puramente combinacional, lo que disminuye la latncia.

El submódulo *Padding* realiza el padding de la cadena de entrada que consiste en adecuar los N bloques M de una forma particular que la función exige. Este procedimiento siempre se realiza, independiente del largo del mensaje.

El submódulo *hash_i* es el corazón del algoritmo, ya que aquí se digiere un bloque M, para generar el hash. Este proceso se itera *N* veces, que es igual a la cantidad de bloques M. El valor de la primera iteración del primer bloque M es un valor constante conocido, mientras que el valor de la primera iteración desde el segundo bloque M en adelante corresponde al hash de la última iteración del bloque M anterior.

El submódulo *hash_mannager*, corresponde al mecanismo de control del hashing y cumple con dos funciones, en primer lugar sincroniza los submódulos de *sha256* levantado las banderas especificas para cada módulo, y en segundo lugar, es el encargado de comunicarse con el exterior, controlando e interpretando las banderas de entrada y salida del módulo.

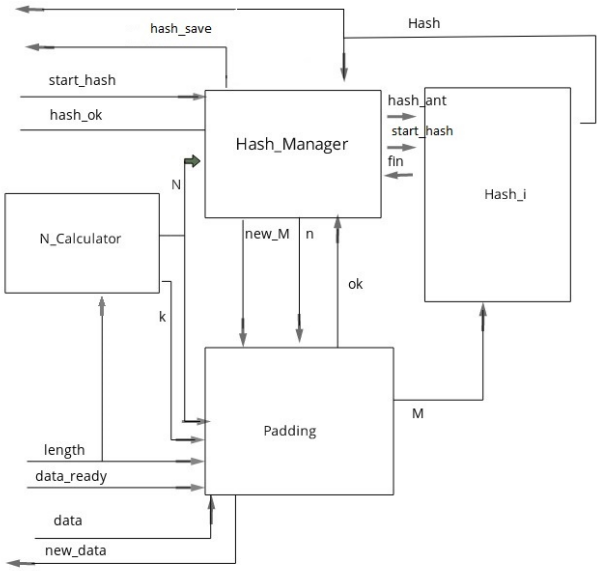

Figura 2. Modulo Sha256 visto por dentro

V. Desempeño

Se considerarán las siguientes características a medir

- **Latencia**: Tiempo que se demora el módulo en calcular el hash desde que se levanta la bandera de *start* y se tiene el resultado en la salida *hash*.
- **Recursos**: Flip-Flops, LUT, Block RAM, etc. que utiliza el módulo *sha256*, y el *overhead* que agrega el driver *driver_ram*

V-A. Latencia

La latencia tiene directa relación con el largo del mensaje de entrada, pues la cantidad de bloques M que digiere el submódulo *hash_i* es función del largo. El primer bloque M que se calcula depende sólo de los primeros 512 bits del mensaje, de forma que se empieza a calcular en cuanto se carga la primera parte del mensaje. El Cuadro I resume las latencias para distintos largos.

Se observa que la latencia es proporcional a N, la cantidad de bloques M que digiere el submódulo *hash_i*. Cada bloque M demora 3010 [ns], ahorrándose 60 [ns] el segundo bloque y 70 [ns] el tercer bloque.

Largo [bits]	Tiempo [ns]
$L \leq 447$	3010
$447 < L \leq 959$	5960
$959 < L \leq 1023$	8900

Cuadro I
LATENCIA EN FUNCIÓN DEL LARGO DE ENTRADA

En base a lo anterior, por cada módulo *sha256* que se tenga en paralelo, se tiene una tasa de [hash/s] de aproximadamente 330 [Khash/s], 167 [Khash/s], o 112 [Khash/s], para los 3 casos que se calculó.

El largo medio de los bloques de Bitcoin actualmente es de 522 bytes aproximadamente[1] lo que entrega una latencia media de 23320 [ns] en este entorno, o aproximadamente 42 [Khash/s]. En comparación, una CPU con un procesador Intel i5-3470 tiene un throughput de 200 [hash/s][2].

Con respecto al uso de recursos, la herramienta indica, para una Nexys 4 DDR, los siguientes valores

Site Type	Used	Fixed	Available	Util %
Slice LUTs	6716	0	63400	10.59
LUT as Logic	6716	0	63400	10.59
LUT as Memory	0	0	19000	0.00
Slice Registers	3533	0	126800	2.79
Register as Flip Flop	3533	0	126800	2.79
Register as Latch	0	0	126800	0.00
F7 Muxes	675	0	31700	2.13
F8 Muxes	320	0	15850	2.02

Cuadro II
REPORTE DE USO DE RECURSOS DE MÓDULO *sha256*

Site Type	Used	Fixed	Available	Util %
Slice LUTs	35	0	63400	0.06
LUT as Logic	35	0	63400	0.06
LUT as Memory	0	0	19000	0.00
Slice Registers	25	0	126800	0.02
Register as Flip Flop	25	0	126800	0.02
Register as Latch	0	0	126800	0.00
F7 Muxes	0	0	31700	0.00
F8 Muxes	0	0	15850	0.00

Cuadro III
REPORTE DE USO DE RECURSOS DE MÓDULO *driver_ram*

Lo que significa que se pueden implementar 9 bloques *sha256* en paralelo, dejando espacio para un driver de RAM propiamente dimensionado, y mecanismos de control y de entrada/salida en una Nexys 4 DDR. En consecuencia, se puede calcular una tasa máxima de 2.97 [Mhash/s] en el mejor de los casos, y en el entorno de Bitcoin, este valor se reduce a una tasa promedio de 0.378 [Mhash/s].

El módulo *sha256*, además utiliza 0.08 [W]. El Cuadro IV muestra el uso de potencia por tipo de señal dentro del módulo.

Clocks [W]	Signals [W]	Data [W]	Clock Enable [W]
0.016	0.036	0.035	0.001
Set/Reset [W]	**Logic [W]**	**BRAM [W]**	**I/O [W]**
$< 0{,}001$	0.028	$< 0{,}001$	$< 0{,}001$

Cuadro IV
REPORTE DE USO DE POTENCIA DEL MÓDULO *sha256*

[1]https://bitcoin.stackexchange.com/q/31974
[2]https://www.xmrstak.com/tag/i5-3470/

VI. CONCLUSIONES

El proyecto cumple con el objetivo principal, que es generar un módulo multipropósito que calcula hash, en particular SHA256. Para mejorar el proyecto en el futuro, en primer lugar hay que mejorar el módulo de hashing, ya que, aún tiene una latencia muy alta para algunas aplicaciones, pues gran cantidad de los subalgoritmos de la función son exclusivamente secuenciales, en consecuencia, es conveniente encontrar técnicas para paralelizar y expandir lo mas posible estos procesos. Por ejemplo, en [3] se implementa SHA256 de una forma más óptima que el diseño nuestro, pues utilizando 6 veces menos *slices*, logran un troughput 4 veces mayor al que se logró en el proyecto

Por otro lado, la implementación actual del hashing tiene espacios para mejoras, disminuyendo ciclos de procesos internos, pues por ejemplo, existen ciertos procesos que tienen buffers previos o posteriores para asegurar que la señal está estable, y no se trabajó en optimizar estos buffers. En tercer lugar, es posible mejorar el hardware que controla el banco de módulos de hashing, este punto es bastante complejo, ya que se busca optimizar el throughput (maximizar [hash/s]), esto consiste en desarrollar un método de asignación de tareas y recursos apropiada según su complejidad y prioridad, que a su vez no aumente la latencia de tareas de hashing particulares (o que las aumente lo menos posible), por ejemplo, si se decide realizar los hash mas cortos primeros, o paralelizar los cálculos redundantes. Una solución de este estilo igual genera problemas, pues se maximizará el throughput, pero alguna tarea que posea una cadena muy larga permanecerá eternamente en espera, gastando espacio en RAM y registros, lo que entrega un amplio espacio de soluciones por explorar.

Por último, el control externo de memoria en la FPGA puede generar un proyecto en si mismo. La implementación que se hizo, es una implementación que cubre lo básico, pero tiene sentido generar un sistema de archivos que optimice el uso de memoria, además de generar un sistema de comunicación que permita recibir mensajes a una tasa más alta, que en este momento, es el mayor cuello de botella del proyecto.

Durante el transcurso del proyecto, se enfrentaron diversos problemas, como por ejemplo tener que generar un protocolo de comunicación del hash con el exterior, o el problema de imaginar e implementar el driver del manejo de memorias. En este último caso se eploraron posibilidades como incluso generar un sistema de archivos básico, aunque eso significaría agregar un overhead muy grande a la lógica del módulo de hash. Se utilizó intensamente las herramientas de simulación de comportamiento (behavioural simulation), especialmente al final del proyecto, que permitió *debuguear* mucho más rápidamente los errores relacionados a la lógica. En relación al protocolo, hubo muchos problemas en el proceso de juntar los módulos *sha256* y *driver_ram*, pues es en ese momento cuando empiezan a encontrarse algunos problemas espúreos en el protocolo, o incluso problemas de comunicación entre los integrantes (se desechan los bits menos significativos o los más significativos?). Gracias a las herramientas de simulación,

y análisis escrupuloso de las señales, se logró detectar a tiempo errores relacionados con condiciones de carrera.

Finalmente, al enfrentarnos por primera vez a un código tan complejo y con tantas señales de control, aprendimos a tratar las señales de *reset* con mayor rigurosidad, propagándolas de una forma más conveniente por el circuito, y no generar una señal de *reset* con un *fanout* de varias miles de señales. Esto incluye generar lógicas complejas sólo para *resetear* de forma fidedigna las señales.

REFERENCIAS

[1] Ubuntu Comunity. Howtosha256sum. https://help.ubuntu.com/community/HowToSHA256SUM. Visitado 2018-06-14.

[2] J. Edward. An overview of cryptographic hash functions and their uses. https://www.sans.org/reading-room/whitepapers/vpns/overview-cryptographic-hash-functions-879. Visitado 2018-06-14.

[3] K. Lee P. Leong K. Ting, S. Yuen. An fpga based sha-256 processor. http://phwl.org/papers/sha_fpl02.pdf. Visitado 2018-06-15.

[4] Florian Mendel, Tomislav Nad, and Martin Schläffer. Improving local collisions: New attacks on reduced SHA-256. In *Advances in Cryptology – EUROCRYPT 2013*, pages 262–278. Springer Berlin Heidelberg, 2013.

[5] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf, 2008.

[6] Gerard Tel. *Cryptography in Context*. 2008.

[7] Unknown. The cryptographic hash function sha-256. https://www.researchgate.net/post/Can_anyone_-help_me_in_implementing_SHA_256_for_class. Visitado 2018-06-15.

[8] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology – CRYPTO 2005*, pages 17–36. Springer Berlin Heidelberg, 2005.