Proyecto 1 Microelectrónica

Leonardo Agüero Villagra
Escuela de Ingeniería Eléctrica
Universidad de Costa Rica
Carné: B70103

Gabriel Jiménez Amador
Escuela de Ingeniería Eléctrica
Universidad de Costa Rica
Carné: B73895

***Resumen*—Para este proyecto se trabaja con un sistema que recibe un bloque de 12 bytes y un target y que mediante una función llamada *micro_ucr_hash* devuelve en su salida un nonce de 4 bytes que genera una salida de 3 bytes en *micro_ucr_hash* donde los primeros dos bytes son menores que el target especificado. Este sistema se describe por medio de diagramas de bloques y modela mediante el lenguaje Golang, sus arquitecturas para optimizar velocidad y área.**

I. Introducción

Una función hash posee la habilidad de convertir cualquier entrada, indiferente de tamaño y devolver una firma de esta entrada de tamaño constante. Esta salida es única para la entrada, además (si es una función hash segura) la entrada no se puede reconstruir a partir de la salida. Es decir, es una función unidireccional.

A continuación se muestran dos implementaciones en código de alto nivel de un sistema que emplea cierta función hash llamada `micro_ucr_hash` para iterar sobre sus salidas para lograr generar salidas especiales según cierto bloque de entrada y *target* deseado. La vista general de bloques del sistema se observa en la Figura 1.

El sistema debe:

1. Recibir un bloque de 12 bytes, un target de 1 byte más una señal de inicio.
2. Concatenarle un nonce de 4 bytes.
3. Calcularle la firma de la concatenación del bloque + nonce, haciendo uso de la función hash `micro_ucr_hash`.
4. Si los dos primeros bytes de la salida:
   - No son menores al target: debe intentar de nuevo modificando el nonce y volviendo a concatenarlo con el bloque.
   - Son menores al target: el sistema debe retornar el nonce que hace cumplir la condición.
5. Al terminar, el sistema retorna una señal de terminación.

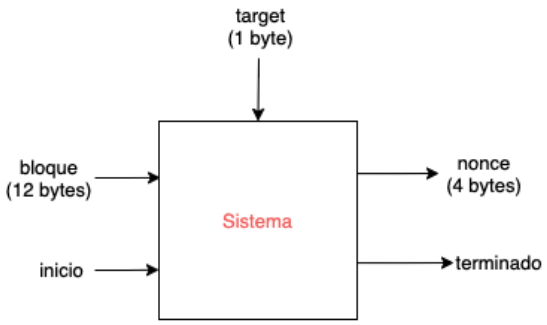

Figura 1. Sistema a describir funcionalmente en código de alto nivel.

Una implementación del sistema fue diseñada priorizando un reducido uso de área (i.e., menores componentes y lógica simplificada), y otra priorizando desempeño aunque utilice mayor área y más componentes. Ambos diseños fueron posteriormente puestos en práctica, creando un modelo en Go. El repositorio trabajado con ambos modelos se puede explorar en https://github.com/gjimeneza/micro-ucr-hash

II. Arquitectura optimizada en área

El principio del diseño de una arquitectura del Sistema presentado en la figura 1 consiste en lograr el funcionamiento correcto mediante la menor cantidad de bloques posibles, en este caso se debe evitar utilizar bloques redundantes para el funcionamiento del sistema ya que un bloque significa mayor área necesitada para construir el ASIC del sistema.

El diagrama de bloques del diseño de la arquitectura optimizada para el área se puede observar en la siguiente figura:

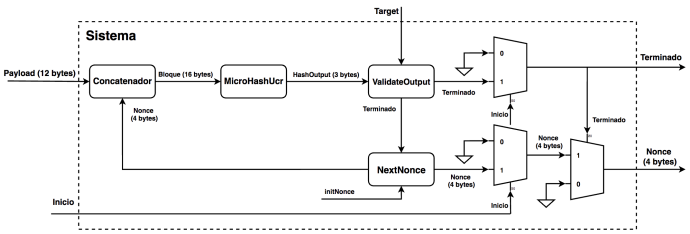

Figura 2. Arquitectura del sistema optimizada para área.

La descripción de la arquitectura es la siguiente:

- **Entradas**
  - **Payload**: bloque de 12 bytes, definido por el usuario que corresponde a los primeros 12 bytes de la entrada de la función `MicroHashUcr`.
  - **Inicio**: señal que cuando está en 1 indica que se ha iniciado el cálculo de un *Nonce* y cuando está en 0 mantiene el circuito apagado.
  - **Target**: Valor definido por el usuario en el cual se utiliza para validar la salida producida por la función `MicroHashUcr`.
- **Salidas**
  - **Nonce**: bloque de 4 bytes que corresponde a una combinación de 4 bytes por el cual la función `MicroHashUcr` ha podido generar un bloque de 3 bytes en el cual los dos primeros bytes son menores al *Target*.
  - **Terminado**: señal que al estar en 1 indica que se ha encontrado un *Nonce* por el cual la función `MicroHashUcr` ha podido producir un bloque de 3 bytes en el cual el valor de los dos primeros bytes son menor al *Target*.
- **Funciones**
  - **Concatenador**: recibe como entradas un bloque de 12 bytes que corresponde a una entrada del sistema definida por el usuario y un *Nonce* de 4 bytes. Su función es concatenar estas entradas para producir una salida de 16 bytes que posteriormente sean la entrada de la función `MicroHashUcr`.
  - **MicroHashUcr**: recibe en su entrada un bloque de 16 bytes en el cual los primeros 12 bytes corresponde a la entrada del sistema de 12 bytes definida por el usuario y sus últimos 4 bytes corresponden al *Nonce*. Esta función genera una salida de 3 bytes cuyo propósito es que sea posteriormente validada por la función `ValidateOutput`.
  - **ValidateOutput**: recibe en su entrada el bloque de 3 bytes generado por la función `MicroHashUcr` y el *Target* definido por el usuario. Esta función valida el *Nonce* utilizado mediante el criterio que el bloque de 3 bytes a la salida de `MicroHashUcr` tenga sus dos primeros bytes con un valor menor al *Target*.
  - **NextNonce**: recibe en su entrada la señal de *Terminado* y *initNonce* que corresponde al nonce inicial. Esta función se encarga de generar un bloque de 4 bytes llamado Nonce que entrará al concatenador para posteriormente ser utilizado en la función `MicroHashUcr` siempre y cuando no se haya validado un Nonce (señal de *Terminado* = 0). Su valor inicial (*initNonce*) es de [0,0,0,0] y al tener que pasar al nuevo valor de *Nonce* se le sumará 1.

La arquitectura consiste en las dos entradas que debe tener el sistema que es el bloque de 12 bytes y la señal de inicio. En caso de que la señal de inicio sea 0 el circuito estará apagado y no se tendrá ninguna señal en la salida.

Al ponerla la señal de inicio en 1 entra los 12 bytes de la entrada (en esta implementación se le llamará *Payload*) donde entrará al bloque `Concatenador` que concatena el *payload* de 12 bytes de la entrada y el *nonce* respectivo para la iteración (para la primera iteración el nonce inicial será [0,0,0,0]) y producirá el bloque de 16 bytes que entra a la función `MicroHashUcr`.

La función `MicroHashUcr` producirá una salida de 3 bytes que en la implementación realizada se nombra como *HashOutput* que entra al bloque `ValidateOutput` el cual revisa que el valor de los dos primeros bytes sea menor que el *Target* que recibe en la entrada. Si `ValidateOutput` valida que la salida producida por `MicroHashUcr` es correcta se coloca la señal de *Terminado* en 1 y se envía a la salida el *nonce* con el que se ha validado el resultado. En caso de ser incorrecta la salida de `MicroHashUcr` se procede al siguiente *nonce* calculado por el bloque `NextNonce` y se vuelve a realizar el proceso hasta que se valide un *HashOutput* de `MicroHashUcr`.

III. Arquitectura optimizada en velocidad

Para la arquitectura enfocada en desempeño y velocidad en la generación de nonces válidos se tomó el diseñó preliminar de área y se decidió paralelizar ciertos componentes en lugar de realizar todo serialmente. Claro, esto significa mayor cantidad de módulos en su implementación en un circuito integrado, y por lo tanto mayor área del ASIC que habría que cubrir.

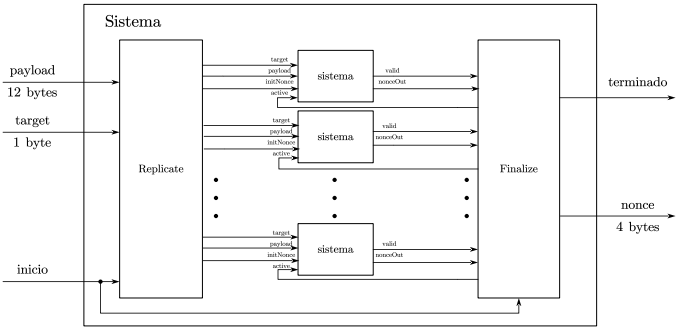

Figura 3. Arquitectura del sistema optimizada para desempeño.

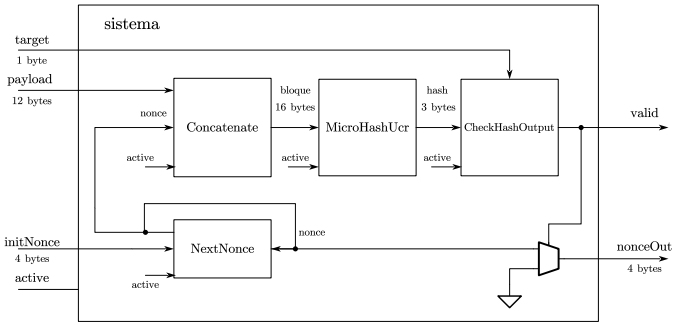

Figura 4. Submódulos paralelizados de Sistema.

El diagrama general de bloques propuesto para el sistema se observa en la Figura 3. En él se observan n submódulos `sistema` que correrán y procesarán nonces diferentes en paralelo. Estos submódulos son descritos por el diagrama de la Figura 4.

El sistema en general tiene las mismas entradas y salidas que el diseño optimizado en área[1], siguiendo las especificaciones indicadas:

- **Entradas**:
  - **Payload**: bloque de 12 bytes. Entrada principal de datos.
  - **Inicio**: señal que indica si se debe empezar a buscar un nonce válido para el payload.
  - **Target**: bloque de 1 byte que corresponde a la condición a buscar en la salida de la función hash.
- **Salidas**:
  - **Nonce**: bloque de 4 bytes del nonce válido que cumple con el target de entrada.
  - **Terminado**: señal que indica la finalización del proceso, indica que se encontró un nonce válido.

Los módulos y submódulos que componen a `Sistema` también son similares al diseño anterior con la diferencia clave de paralelización para eso se requiere replicar señales y modulos:

- **Funciones/Módulos que componen el Sistema**
  - **Concatenate**: recibe el *payload* (12 bytes) y lo concatena con un nonce de entrada (4 bytes) para generar el bloque (16 bytes) a ingresar al módulo de la función hash.
  - **MicroHashUcr**: módulo que procesa un bloque (16 bytes) y genera la firma hash de este bloque (3 bytes) según la función hash `micro_ucr_hash`. Esta salida debe verificar que cumple con el target.
  - **CheckHashOutput**: verifica que los dos primeros bytes del módulo de la función hash (3 bytes) son menores al target de entrada.
  - **NextNonce**: recibe un nonce (4 bytes), lo incrementa en uno, y lo devuelve. En el primer ciclo de reloj luego de activado el submódulo (`posedge`), debe enviar `initNonce`, luego hasta desactivado debe enviar el nonce incrementado.
  - **sistema**: módulo que realiza el proceso principal de:

    $$\text{payload} + \text{nonce inicial} \rightarrow \text{MicroUcrHash} \rightarrow \text{verificacion de salida} \rightarrow \text{nonce válido}$$

    La idea es tener varios módulos de estos que verifiquen salidas, partiendo e incrementando desde nonces diferentes, todo en paralelo.
  - **Replicate**: este módulo se encarga replicar las señales para cada uno de los módulos `sistema` creados, repartiéndoles el nonce inicial para empezar a verificar salidas. El nonce inicial estaría definido por:

    $$\text{nonce inicial} = \frac{\text{nonce max}}{n} \cdot i$$
    $$\text{nonce max} = \text{0xFFFFFFFF}$$
    $$n = \text{número de sistemas}$$
    $$i = \text{id del sistema (0,1,2,...)}$$

  - **Finalize**: este módulo se encarga de recibir el primer nonce válido que genere alguno de los sistemas, apenas reciba este valor, se manda una señal a los sistemas de que paren de calcular y se devuelve el nonce válido con la señal de *terminado*.

De nuevo cabe recalcar que la funcionalidad de este diseño en teoría es la misma que la de la optimizada en área, salvo que logrará encontrar un nonce válido mucho más rápido gracias a la paralelización del proceso principal.

IV. Simulación y resultados

Se creó una descripción funcional de ambos sistemas en Go. Se eligió Go para aprovechar su facilidad de uso tanto para crear el modelo principal con las módulos siendo funciones, para probar funcionalidad con tests de Go, como para simular paralelización de componentes con las *goroutines* y concurrencia de Go.

Se crearon pruebas unitarias para comprobar funcionalidad correcta de la función hash y del Sistema. Además se creó una CLI (*command line interface*) para fácil uso del programa. El proyecto se puede explorar en https://github.com/gjimeneza/micro-ucr-hash

IV-A. Área

Utilizando el modelo optimizado en área, luego de compilado, para generar un nonce válido con un target de 10 y un payload de `0x397d9f2f40ca9e6c6b1f3324fd` se ejecuta el binario con los siguientes argumentos:

```
./micro-ucr-hash area -p 397d9f2f40ca9e6c6b1f3324fd -t 10
```

Lo que genera una salida de:

```
Entered payload: [0x39 0x7d 0x9f 0x2f 0x40 0xca 0x9e 0x6c
0x6b 0x1f 0x33 0x24 0xfd]
Desired target: 10
Elapsed time: 1.0567611s
Generated HashOutput: [0x09 0x09 0x04]
Generated Nonce: [0x01 0x00 0x1b 0x26]
```

Se observa que se obtuvo un nonce de `0x01001b26` para generar una salida de la función hash de `0x090904` en un tiempo de 1,057 s. Se puede ver que los dos primeros bytes de la salida de la función hash cumplen con el requerimiento del target, ambos son menores a 10.

IV-B. Velocidad

Ahora con el sistema optimizado en desempeño con 6 sistemas procesando concurrentemente, para generar un nonce válido con un target de 10 y un payload de `0x397d9f2f40ca9e6c6b1f3324fd` se ejecuta el binario con los siguientes argumentos:

[1]Para mayor detalle de estos, se sugiere revisar este apartado en la sección del diseño optimizado en área.

```
./micro-ucr-hash speed -p 397d9f2f40ca9e6c6b1f3324fd -t 10
```

Lo que genera una salida de:

```
Entered payload: [0x39 0x7d 0x9f 0x2f 0x40 0xca 0x9e 0x6c 0x6b
0x1f 0x33 0x24 0xfd]
Desired target: 10
Elapsed time: 460.6µs
Generated HashOutput: [0x09 0x09 0x40]
Generated Nonce: [0xd5 0x55 0x6f 0x09]
```

Se observa que ahora se obtuvo un nonce de `0x0d5556f09` para generar una salida de la función hash de `0x090940` en un tiempo de $460{,}6\,\mu s$. Comparando tiempos de ejecución contra el modelo de área se tiene que este sistema es $\frac{1{,}057}{460{,}6 \times 10^{-6}} = 2300$ veces más rápido, utilizando sólo 6 sistemas trabajando concurrentemente.

Se puede ver que los dos primeros bytes de la salida de la función hash cumplen con el requerimiento del target, ambos son menores a 10. Cabe denotar que tanto el `HashOutput` como el `Nonce` generado en la salida no son los mismos que los generados en el otro sistema, esto es esperado y se debe a que un sistema encontró un nonce válido diferente primero antes del mismo que se encontró en el sistema optimizado en área.

V. Papel del sistema en las criptomonedas, optimización y impacto en el ecosistema de hardware

El sistema de funcionamiento de las criptomonedas está basado en una tecnología llamada *Blockchain*. El *blockchain* es definido como un servidor de marcas de tiempo distribuido de forma "peer-to-peer" basada en bloques y cadenas como una forma de organizar y proteger registros de manera que al entrar a una basa de datos compartida puedan ser validados de forma matemática y permanecer intactos sin cambio alguno. [1]

El sistema diseñado y modelado para este proyecto corresponde a una función de *Hashing*. Este tipo de funciones son de suma importancia para las tecnologías *Blockchain* debido a que hace cumplir medidas de seguridad en los nodos (celulares o computadores de los usuarios) que verifican transacciones (una transacción corresponde al movimiento de activos de criptomonedas) además de introducir capas o bloques de seguridad. [2]

La característica de descentralización de esta tecnología hace que muchos factores se toman en cuenta en un sistema de *Blockchain* entre ellos está la eficiencia, el consumo energético, costo, seguridad, infraestructura, tiempo de ejecución entre muchas otras. Existen diversas maneras de optimizar estos factores y cada uno tiene sus compensaciones y que rubro optimizar depende de la aplicación que se está realizando [3]. Un ejemplo muy claro es de la seguridad y los tiempos de ejecución, si se quiere un algoritmo de *hashing* que sea más seguro y menos propenso a ataques normalmente va significar un mayor aumentos en los tiempos de ejecución y computaciones realizadas por transacciones que a cambio va aumentar otros rubros como el consumo energético y el costo [2]. Debido a esto es que se debe conocer las prioridades de la aplicación que se está realizando, el tema de la optimización ha generado un gran interés en investigadores y practicantes de la industria que buscan maneras de mejorar los sistemas de *blockchain*.

Para darle seguimiento al nivel computacional generado por los sistemas de *blockchain* los anfitriones y mantenedores de servidores de *blockchain* emplean infraestructuras de hardware que puedan manejar las altas cargas computacionales como por ejemplo CPU, GPU, ASIC, entre otros... Y que ha su paso consumen excesivas cantidades de energía [3]. Esto se ha vuelto un reto dentro de este campo puesto que los niveles de consumo energético de infraestructuras de *blockchain* han llegado a igualar el consumo de energético de varios países y más que un reto se ha vuelto en una controversia, un caso reciente y muy famoso fue la decisión del CEO de la compañía automovilística Tesla, Elon Musk, de eliminar la opción de compra de los vehículos producidos por la compañía mediante la criptomoneda *Bitcoin* citando la problemática que genera los altos niveles de consumo energético de las tecnologías Blockchain y el impacto de su huella ambiental [4].

Actualmente debido a la pandemia del COVID-19 se ha presentado una escasez de chips a nivel mundial, industrias como las de computadores, automovilística, médica, entre muchas otras que necesitan de estos elementos se les han presentado atrasos y paros totales en sus líneas de producción debido a que la demanda de chips es mayor a la cantidad que hay en el mercado [5]. La competencia en la minería de criptomonedas ha empeorado esta escasez. Como se ha explicado, los sistemas de *blockchain* necesitan de alta capacidad computacional y mucha de la competencia se resume en quien tiene los mejores y más eficientes sistemas. Muchas empresas productoras de CPU y GPU han empezado a adoptar medidas para aislar sus productos para el público en general de los criptomineros y de esta forma alivianar la carga en sus líneas de producción. Un ejemplo de esta práctica es el de la compañía desarrolladora de GPU, NVIDIA, la cual para evitar que criptomineros compren de sus GPU han desarrollado "Cryptocurrency Mining Processor" (CMP) que es una versión de sus tarjetas gráficas optimizada para esta actividad y por el otro lado sus tarjetas gráficas diseñadas para videojuegos y consumidores en general se les ha ralentizado a propósito partes de las tarjetas que son utilizadas por los mineros y de esta manera dividir el mercado [6].

Debido a las situaciones antes mencionadas, es que las criptomonedas y las tecnologías *blockchain* tienen mucho camino que recorrer y aspectos que mejorar, no cabe duda que su aparición en el mundo de la tecnología ha estado marcado de polémicas. Estas tecnologías tienen mucho potencial para no solo ayudar al mundo, sino de cambiarlo por completo pero si quieren que la comunidad mundial los vea con buenos ojos deben seguir optimizando sus tecnologías y reducir su huella ambiental. Son tecnologías muy recientes y en los próximos años se podrá ir viendo sus avances y su influencia en el mundo, su presencia ha sido todo un reto para la comunidad tecnológica principalmente para la industria de manufactura de chips de computadores pero eventualmente las industrias se

acostumbrarán a su presencia y los usuarios también, el tiempo dirá si estas tecnologías llegaron para quedarse o pasarán a ser otra moda pasajera.

VI. Conclusiones

- Se pudo diseñar dos sistemas, uno enfocado en reducción de área y otro en desempeño, que calcula nonces válidos según un `payload` de entrada y siguiendo la condición de `target` aplicada a la salida de una función hash.
- Se pudo observar cómo, a partir de los modelos de los sistemas en lenguaje de alto nivel se comprueba el funcionamiento de ambos diseños creados.
- Se pudo comparar el tiempo de ejecución de ambos modelos del sistema en Go, donde el modelo enfocado en desempeño pudo calcular un nonce válido en un tiempo de 2300 veces menor.
- Se logró desarrollar un modelo que necesite menos área pero que a cambio reduce significativamente su velocidad para obtener un nonce válido.
- Se explica el impacto que tienen las tecnologías de *blockchain* y la criptominería en el ecosistema de hardware y los retos que necesitan resolver para que la comunidad mundial perciba únicamente los grandes beneficios que pueden brindar.

Referencias

[1] T. Laurence, *Introduction to Blockchain Technology*. Van Haren Publishing, 2019. [Online]. Available: https://books.google.co.cr/books?id=uD-4DwAAQBAJ

[2] F. Wang, Y. Chen, R. Wang, A. O. Francis, B. Emmanuel, W. Zheng, and J. Chen, "An experimental investigation into the hash functions used in blockchains," *IEEE Transactions on Engineering Management*, vol. 67, no. 4, pp. 1404–1424, 2020.

[3] M. Fang and J. Liu, "Toward low-cost and stable blockchain networks," in *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, 2020, pp. 1–6.

[4] Bitcoin Mining Adds to Existing Shortage in Semiconductor Market, Chip Prices Surge. BBC. [Online]. Available: https://www.bbc.com/news/business-57096305

[5] Bitcoin Mining Adds to Existing Shortage in Semiconductor Market, Chip Prices Surge. Anatol Antonovici. [Online]. Available: https://finance.yahoo.com/news/bitcoin-mining-adds-existing-shortage-120348278.html

[6] The chip shortage is pitting bitcoin miners against gamers. Nicolás Rivero. [Online]. Available: https://finance.yahoo.com/news/chip-shortage-pitting-bitcoin-miners-200508812.html

[7] [Online]. Available: https://twitter.com/elonmusk/status/1284291528328790016?s=21