# Resultado de imagen para javeriana bogota escudo"

Generación de bloques en una red blockchain

*TG 1826*

*~~~~~~~~~~~~~~~*

*Autoras:*

Adriana Andrea Aguirre Angel

Maria Paola Fonseca Páez

Silvana López Cuellar

*Directores:*

*Ing. Francisco Fernando Viveros*

*Ing. Jose Luis Uribe Aponte*

*~~~~~~~~~~~~~~~*

*Facultad de Ingeniería*

*Departamento de Electrónica*

*2019*

**AGRADECIMIENTOS**

# **ÍNDICE**

[1 INTRODUCCIÓN 4](#_Toc26029269)

[2 MARCO TEÓRICO 5](#_Toc26029270)

[3 GLOSARIO 7](#_Toc26029271)

[4 OBJETIVO DEL PROYECTO 7](#_Toc26029272)

[4.1 Objetivo general 8](#_Toc26029273)

[4.2 Objetivos específicos relevantes 8](#_Toc26029274)

[5 DESARROLLO 8](#_Toc26029275)

[5.1 Sistema de minado en](#_Toc26029276) *[hardware](#_Toc26029276)* [10](#_Toc26029276)

[5.1.1 Diseño e implementación del subsistema ‘función](#_Toc26029277) *[hash sha256’](#_Toc26029277)* [11](#_Toc26029277)

*[5.1.2](#_Toc26029278)* [Diseño e implementación del subsistema ‘generación](#_Toc26029278) *[nonce’](#_Toc26029278)* [17](#_Toc26029278)

[5.1.3 Diseño e implementación del subsistema ‘comunicación serial’ 21](#_Toc26029279)

[5.1.4 Diseño e implementación del subsistema ‘unidad de tiempo’ 26](#_Toc26029280)

[5.1.5 Diseño e implementación del subsistema ‘comparar ceros’ 28](#_Toc26029281)

[5.1.6 Diseño e implementación del subsistema ‘control global’ 28](#_Toc26029282)

[5.2 Sistema de minado en](#_Toc26029283) *[hardware](#_Toc26029283)*[, variación 30](#_Toc26029283)

[5.2.1 Planeamiento y variaciones para la implementación de las unidades de minado 32](#_Toc26029284)

[5.2.2 Diseño e implementación del subsistema ‘out select’ 33](#_Toc26029285)

[5.2.3 Diseño e implementación del subsistema ‘output sel’ 34](#_Toc26029286)

[5.2.4 Diseño e implementación del subsistema ‘control global’ para la variación 35](#_Toc26029287)

[5.3 Desarrollo en software 36](#_Toc26029288)

[6 PROTOCOLO DE PRUEBAS 39](#_Toc26029289)

[7 ANÁLISIS DE RESULTADOS 42](#_Toc26029290)

[8 CONCLUSIONES Y RECOMENDACIONES 42](#_Toc26029291)

[9 BIBLIOGRAFÍA 43](#_Toc26029292)

[10 ANEXOS 44](#_Toc26029293)

# **[INTRODUCCIÓN](#_ÍNDICE)**

La tecnología *blockchain* (cadena de bloques) se dio a conocer gracias a su más popular aplicación para el pago de transacciones vía virtual, es decir, por medio de la *cryptocurrency Bitcoin* [3]. Su funcionamiento se resume en que la información de cada transacción que se haya completado, se codifica y se comparte de forma que esté disponible para todos los nodos, es decir, se agrega a la cadena de bloques [1]. Para esto, es necesario primero validarla y empaquetarla, y es a este proceso al que se le conoce como minería [4]. El cifrado de la información se realiza por medio de un algoritmo de complejidad matemática conocido como función *hash* que permite codificar la información de forma que no pueda ser decodificada; el resultado se conoce como valor *hash* o solamente *hash* [4].

La complejidad del proceso de minado por medio del algoritmo de consenso *Proof-of-work* (PoW) ha generado necesidades con respecto a la capacidad de cómputo de los equipos usados [4]. Una de éstas es la reducción del gasto energético que requiere el proceso, pues se efectúan miles de operaciones matemáticas continuamente para realizar el *hash* en cada bloque [4]. Según Wang L, entre el 2010 y 2013, una GPU realizaba 750 M*hash*/seg generando un gasto energético de 410 W solo en esta función [4]. Otra, es la reducción del tiempo que se tarda cada transacción en ser validada. En promedio hasta el 24 de noviembre del 2019, el proceso de minado tarda en promedio 10 minutos para lograr la validación del bloque según la página oficial de gráficos de Bitcoin [5].

El tiempo estimado es producto de un equipo especializado en minado, pues no es recomendable realizar el proceso en un computador común [11], debido a la improductividad de correr un programa con esta funcionalidad, sobre un sistema operativo que dedica los recursos del procesador entre diferentes tareas. Esto resulta en que el proceso de minado se lleve a cabo en periodos muy prolongados de tiempo.

De esta forma, para esta tecnología con una gran proyección a futuro según la citada página [11], se genera la necesidad de invertir en la optimización del sistema, específicamente en la sección de validación de los bloques. En conjunto, también se requiere de investigación en la mejora de equipos de cómputo específicos aplicados a la minería. Se han realizado diferentes estudios sobre *blockchain* buscando mejorar el consumo generado por el proceso de minado y la aplicabilidad a otras áreas diferentes a las transacciones de divisa virtual, como el ambiente electoral, almacenamiento de datos, etc.

En el presente documento se encontrará el capítulo ‘Marco Teórico’, donde se definirán todos los conceptos necesarios para comprender la totalidad del informe. Posteriormente en el capítulo ‘Objetivos’ se presentan los objetivos y metas planteadas para el desarrollo del presente proyecto de trabajo de grado, junto con una descripción del resultado final alcanzado.

Una vez enunciado esto, se tiene el capítulo ‘Desarrollo’ donde se explica el proceso de concepción, diseño e implementación de cada parte del proyecto, el capítulo de ‘Protocolo de pruebas’ donde se detallan todas las pruebas que se realizaron junto con su descripción y justificación, el capítulo de ‘Análisis de los Resultados’ donde se presentan el procesamiento e interpretación de los resultados obtenidos en las pruebas realizadas.

Finalmente se encuentra el capítulo ‘Conclusiones’ donde se presentan los puntos concluyentes a partir de la interpretación de los resultados obtenidos, junto con los hallazgos y recomendaciones del proyecto. Todas las fuentes bibliográficas se encuentran debidamente citadas en el capítulo ‘Bibliografía’. Finalmente, todos los planos, esquemáticos, simulaciones, etc. se encuentran en los anexos de este documento.

# **[MARCO TEÓRICO](#_ÍNDICE)**

En este capítulo se presenta un resumen de los conceptos teóricos usados durante el desarrollo del proyecto que proporcionan las bases teóricas mínimas para comprender la totalidad del informe.

Uno de los conceptos más generales a tener en cuenta para el presente trabajo de grado es el concepto de *Blockchain,* conocido en español como ‘cadena de bloques’. Es una tecnología de bases de datos distribuida que mantiene una lista en continuo crecimiento de registro de datos vinculados por medio de la criptografía, que son confirmados por nodos (usuarios que poseen una copia de la información y participan en ella) [1][2]. Se caracteriza por brindar un alto nivel de anonimato y seguridad informática [2], por consiguiente, la información de cada transacción que se haya completado se codifica y se comparte de forma que está disponible para todos los nodos [1]. Cada vez que se realiza una nueva transacción, es necesario validar y empaquetar la información para agregar un nuevo bloque en la cadena: este proceso se conoce como minería [4]. El cifrado de la información se realiza por medio de un algoritmo de complejidad matemática conocido como función *hash*.

Con el fin de comprender la definición de función *hash*, se debe tener en cuenta que un *hash* de n bits es una función que desempeña un papel de mapa para mensajes de longitud arbitraria a valores resultantes de n bits. La definición del *hash* criptográfico de n bits se le añade que cumple con unidireccionalidad y resistencia a colisiones. La importancia de estas funciones criptográficas radica en su utilización para firmas digitales y protección de contraseñas, entre otras[]. Teniendo esta definición en cuenta, la función de compresión SHA-256 es esencialmente un algoritmo de cifrado que actúa en bloques de mensajes de 512 bits con un valor de *hash* intermedio de 256 bits, utilizando cada bloque de mensaje como clave[].

El mensaje que se va a codificar, para utilizar una cadena de bloques de 512 bits en la entrada, debe pasar por el proceso de ‘*padding*’ siguiendo con el estándar. Esto se realiza para que el resultado sea siempre múltiplo de 512 bits de longitud, analizándolos en una cadena de bloques de mensajes de igual longitud. Los bloques de mensajes se procesan uno a la vez.

El proceso de *padding* consiste en rellenar los espacios vacíos de un mensaje de entrada para cumplir con un tamaño fijo. Si la información ocupa un tamaño menor al máximo fijo establecido, después del último bit de información del mensaje se pone un bit en ‘1’ seguido de un mínimo de bits en ‘0’ (64 en este caso), y finalmente en los últimos bits se debe indicar la longitud del mensaje. Si el tamaño de la información supera el máximo establecido, se debe usar un bloque adicional y rellenar de la misma forma.

Ejemplo

Información en base hexadecimal: 616263

Información tras proceso de ‘*padding*’ en base hexadecimal: 61626380000000000000000000000000

000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000018

El número de ceros cumple con el mínimo de 64 bits, y los últimos bytes indican la cantidad de caracteres del mensaje inicial, es decir, 24 bits.

[k] Descriptions of SHA-256, SHA-384, and SHA-512.

<http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf>

En *Blockchain*, para agregar los llamados ‘bloques’ a la cadena cada nodo, se debe ‘demostrar’ que se ha realizado una cierta cantidad de trabajo, también conocido como Prueba de Trabajo (PoW). Esta consiste en el proceso de resolver el acertijo planteado (dependiente del protocolo que se esté usando de PoW) para encontrar un valor hash ‘ganador’ que satisface al acertijo. A este proceso se conoce como *minería*. El primer nodo o ‘minero’ en encontrar un hash ganador puede agregar su bloque propuesto a la cadena de bloques y también reclamar una recompensa de minería, y es por ello que ‘compiten’ para ser el primero en resolver el acertijo mediante sus máquinas especializadas *ASIC* dedicadas al minado.

<https://pdfs.semanticscholar.org/da8a/37b10bc1521a4d3de925d7ebc44bb606d740.pdf?_ga=2.21200635.1919538867.1522092864-1798624458.1520283070&source=post_page--------------------------->

Generalmente el ‘acertijo’ planteado por el sistema requiere que las transacciones al ser minadas, cumplan con una cierta cantidad de ceros en sus posiciones más significativas. Ya que una entrada siempre produce el mismo valor *hash* usando la misma función *hash*, con el fin de producir salidas diferentes adjuntan cierta información al inicio de la transacción, y la cambian hasta que el resultado satisfaga la cantidad del número de ceros deseados. A esta información se le llama *nonce*, término que proviene de la abreviación del inglés *‘number that can be only used once’*.

Este debe cambiarse cada vez que el resultado de la codificación no genere la salida esperada, y se espera que nunca se repita un mismo nonce, sino que por el contrario esta información siempre varíe de manera pseudo-aleatoria, hasta que se genere el número de ceros esperados en el resultado.

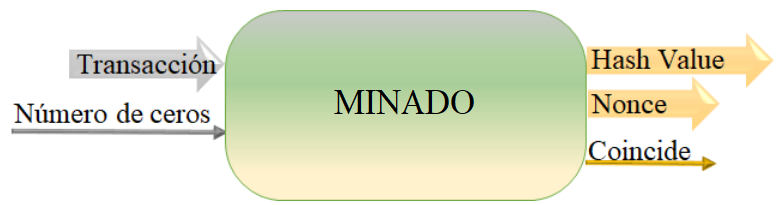


Figura 1. Esquema de entradas y salidas de un bloque de minado

En la Figura 1 se muestra el diagrama de entradas y salidas de un bloque de minado para un protocolo que exige únicamente número de ceros, el cual recibe como entradas la transacción a minar junto con el número de ceros esperados en el valor *hash*, y entrega como salidas el valor *hash* que satisface el número de ceros especificado, junto con el *nonce* usado para obtener el mismo y una señal que indica cuando el resultado es válido.

<http://www.cs.ucdavis.edu/~rogaway/papers/nonce.pdf>

Los *nonce* comúnmente se generan de manera pseudo-aleatoria con el fin de tener una secuencia de *nonce* replicable, y obtener siempre el mismo *nonce* al ingresar la misma entrada.

Los generadores de información pseudo-aleatoria se caracterizan por la utilización de una semilla inicial x0, la cual genera una sucesión de valores xn, mediante una relación de recurrencia establecida xn = T(xn−1)[ñ]. Cada uno de estos valores es en relación con la secuencia generada, un número pseudo-aleatorio definido a través de la relación establecida por el método[ñ].

Los generadores criptográficos de números pseudo-aleatorios (*PRNG*) se caracterizan por tomar una sola semilla segura y producir tantos números pseudo-aleatorios (aparentemente no predecibles) de esa semilla como sea necesario [2]. De forma tal que si las condiciones de entrada o semilla se replican, se pueda obtener la misma secuencia de números.

Uno de los métodos más populares es el método de los cuadrados medios, propuesto por los matemáticos John von Neumann y Nicholas Metropolis en los años 40. Comienza tomando un número x0 de 2n cifras, que al elevarlo al cuadrado resulta un número de 4n cifras (si es necesario se añaden ceros a la izquierda para que el número resultante tenga 4n cifras). x1 es el número resultante de seleccionar las 2n cifras centrales de x02. A continuación x2 se genera a partir de x1 del mismo modo[ñ].

[ñ]<https://webs.um.es/mpulido/miwiki/lib/exe/fetch.php?id=amio&cache=cache&media=wiki:simt1b.pdf>

# **[GLOSARIO](#_ÍNDICE)**

En este capítulo se definen las terminologías más utilizadas, según el significado que toman para el desarrollo del presente libro, dichas terminologías se presentan a continuación.

**Bloque** Unidad de información de 512 bits de longitud.

**Palabra** Unidad de información de 32 bits de longitud.

**Sistema** Unidad jerárquica compuesta por subsistemas.

**Subsistema** Unidad dedicada a la realización de funciones específicas en un sistema.

**Valor *hash / digest***Resultado de aplicar la función *hash* sobre una entrada*.*

**Función *hash sha256*** Función *hash* de cifrado *sha* 256

***Nonce*** Cadena de información que varía constantemente su valor

***Padding*** Acción de rellenar la información faltante de un bloque para cumplir con un tamaño específico.

**Rx** Abreviación de ‘recepción’

**Tx** Abreviación de ‘transmisión

**Byte** Unidad de información compuesta por 8 bits

**Bit de inicio** Bit que tomando el valor ‘0’, indica el inicio de una transmisión de un byte.

**Bit de parada** Bit que tomando el valor ‘1’, indica el fin de la transmisión de un byte.

**RAM** Memoria de acceso aleatorio

**Método** Secuencia de instrucciones que realizan una tarea específica en *python*

**Caractér** Byte de información cifrado en *ASCII* de 8 bits de longitud

# **[OBJETIVO DEL PROYECTO](#_ÍNDICE)**

En este capítulo se presentan los objetivos planteados en el anteproyecto del presente trabajo de grado, junto con una síntesis de lo alcanzado en el proyecto.

## **[Objetivo general](#_ÍNDICE)**

Desarrollar un sistema digital en FPGA que genere un nuevo bloque en una red *blockchain* utilizando el proceso de minado propuesto para el algoritmo de consenso: *proof-of-work*.

## **[Objetivos específicos relevantes](#_ÍNDICE)**

1. Diseñar una arquitectura del sistema que supla los requerimientos funcionales del algoritmo de consenso PoW.
2. Desarrollar al menos dos variaciones en la arquitectura para realizar una comparación entre éstas.
3. Evaluar las herramientas y dispositivos programables que permitan realizar una evaluación objetiva de las arquitecturas y sus variaciones.
4. Diseñar un entorno que permita la realización de la medición del tiempo para lograr la respuesta de nonce deseada para las diversas variaciones de la arquitectura.
5. Analizar el desempeño de las arquitecturas propuestas a partir del entorno planteado.

# **[DESARROLLO](#_ÍNDICE)**

En el presente capítulo se presenta la descripción de cada una de las partes que conforman el proyecto, junto con los dispositivos utilizados y los diseños realizados. A continuación se muestra el sistema principal desarrollado, en la figura X:

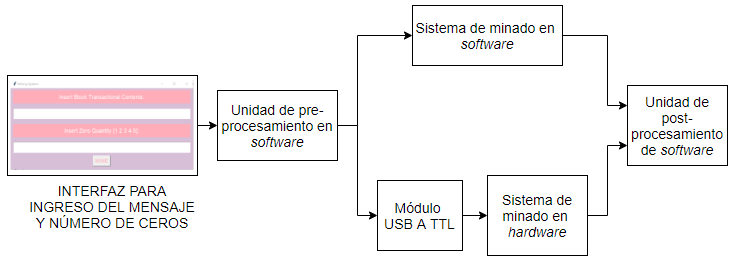
**

Figura 2. Diagrama general del proyecto

En el presente trabajo de grado se desarrolló un sistema digital en una FPGA stratix IV en una tarjeta de desarrollo ALTERA Terasic DE4, el cual cumple con la función de generar el valor *hash* de un bloque de información para una red *blockchain* utilizando el proceso de minado propuesto para el algoritmo de consenso: *proof-of-work*, en el cual se cumple con obtener un resultado que satisfaga el número de ceros especificado en la entrada del sistema.

El objetivo del presente trabajo de grado tiene como fin la realización un sistema en *hardware,* el cual realizara la función de minado sobre una información de entrada, produciendo un valor *hash* con un determinado número de ceros en sus primeras posiciones.

No obstante, la obtención de un resultado sin referencia alguna no garantiza la validez del mismo, por lo que se desarrolló un entorno de verificación mediante la implementación de un código en *software,* el cuál realizara el mismo proceso de minado, con el fin de obtener una verificación desempeño del sistema de minado *hardware*.

El sistema total desarrollado se compone de una interfaz, desarrollada en *software,* en la que reciben las entradas del sistema, las cuales son la información que se desea minar y el número de ceros que se desean. Una vez se toman las entradas por el sistema, se activa la unidad de pre-procesamiento en *software*, desde la cual las entradas son procesadas y enviadas tanto al módulo de minado de *hardware* como al módulo de minado de *software*.

El envío de las entradas al módulo de minado en *hardware* se realiza de manera serial, por medio de un conversor TTL a USB CP2102 elevando la línea mediante tarjetas *MAX232*. Una vez ingresadas las entradas al módulo de minado en *hardware*, se realiza el proceso de minado y se reciben los resultados en la unidad de procesamiento en *software*, donde se realiza el mismo proceso en el módulo de minado de *software*.

Finalmente, una vez obtenidas las salidas de los dos sistemas, se comparan los dos resultados de forma que las salidas de cada módulo de minado satisfagan el número de ceros esperados en las posiciones más significativas, y sean iguales. Junto con los *nonce* que produjeron los correspondientes valores *hash.* Estos datos junto con el tiempo que le tomó a cada sistema procesar se almacenan en un archivo de texto (.txt) que indica los resultados de la prueba.

Para el protocolo de pruebas se desarrolló una versión de *software* especializada a pruebas, la cual será explicada en la sección del protocolo de pruebas.

Para el bloque de minado en *hardware* se desarrolló una arquitectura adicional basada en variaciones sobre la implementada originalmente, cumpliendo con el objetivo general N°2, la cual principalmente se diferencia en que posee 16 instancias de minado en paralelo dedicadas a la búsqueda del *nonce*.

Las unidades que componen el sistema presentado en la Figura 2 son las siguientes:

* Interfaz de *software*, la cual recibe la información que se desea minar junto con el número de ceros que se desea. La información aquí recopilada se organiza y se envía a la siguiente etapa. Todos los desarrollos de software se realizaron en lenguaje *Python* y se ejecutan en el entorno de desarrollo *Visual Studio Code*.
* Unidad de pre-procesamiento en *software*, desde el cual las entradas se procesan ejecutando un *‘padding’* sobre la entrada de información, se organizan y envían a cada etapa siguiente. Las etapas siguientes constan del módulo de minado de *hardware* y el módulo de minado de *software*.
* Módulo conversor TTL a USB CP2102 el cual soporta el envío de las entradas al módulo de minado *hardware*, urilizando tarjetas *MAX232*.
* Módulo de minado en *hardware* el cual recibe las entradas de la línea de transmisión, en el siguiente orden: primero el número de ceros deseados y luego el mensaje que se desea codificar. El módulo realiza el proceso de minado sobre esto y envía los resultados de manera serial por medio del TTL a USB de vuelta al *software*. Todos los desarrollos en *hardware* se realizaron en lenguaje descriptivo de *hardware Verilog* o *VHDL*, y se implementaron en una *FPGA* stratix IV dispositivo EP4SGX230KF40C2 en una tarjeta de desarrollo *ALTERA Terasic DE4*.
* Módulo de minado en *software* el cual realiza el mismo proceso algorítmico realizado por el módulo de minado en *hardware*, y envía los resultados obtenidos a la unidad de post procesamiento del sistema.
* Unidad de post procesamiento en *software* donde se reciben los resultados del módulo de *hardware* y de *software*, comparando que los resultados de la función *hash* y *nonce* sean iguales, junto con el tiempo que le tomó a cada sistema procesar. Aquí se genera el archivo de texto (.txt) que indica los resultados de la prueba.

## **[Sistema de minado en](#_DESARROLLO) *[hardware](#_hchruwq62c7r)***

En esta sección se presenta la concepción y diseño de todas las partes del proyecto que conforman la primera arquitectura para el módulo de minado en *hardware*. Inicialmente se describen los bloques en la capa jerárquica más alta del sistema.

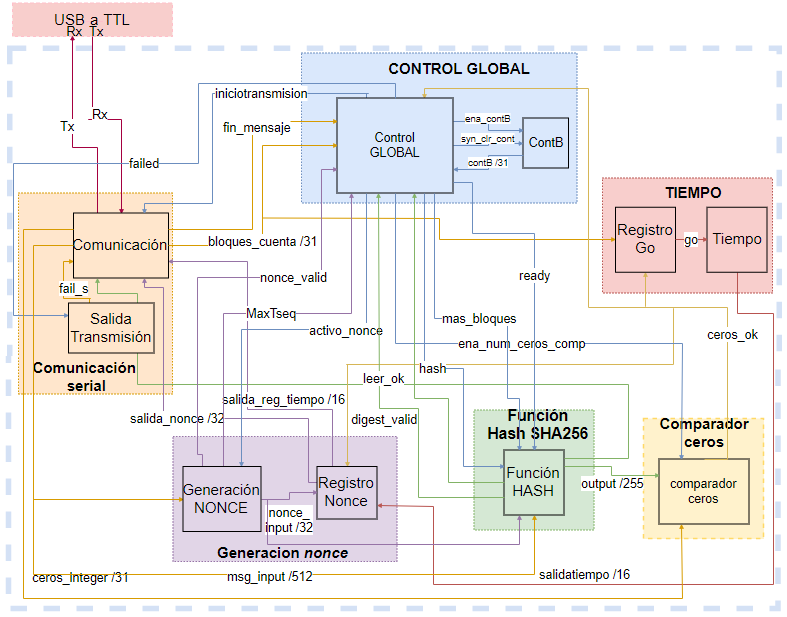


Figura 3. Diagrama de bloques del sistema de minado de hardware

En el sistema de minado de *hardware* ilustrado en la Figura 3, se reciben las entradas y transmiten sus salidas por medio del conversor **TTL a USB**, recibiendo toda la información de entrada del pin de Rx, llegando directamente sub sistema de **comunicación** donde la entrada es almacenada en la memoria *RAM* del sistema. Inmediatamente el sistema detecta que la totalidad de la información de entrada ha sido leída, inicia el procesamiento del subsistema de **tiempo**, e inicia el proceso de minado.

Inicialmente una parte del mensaje (información del primer y último byte) es enviada al subsistema **generación *nonce****,* donde una vez generado un *nonce*, es adjuntado al inicio del mensaje y enviado al subsistema **función *hash* *sha256***. Una vez el resultado es obtenido, se lleva al subsistema **comparador de ceros** donde se compara si las posiciones más significativas satisfacen el requerimiento de número de ceros. En caso de ser así el sistema detiene el conteo de la unidad de tiempo y toma el *nonce*, el resultado de la función *hash* y el tiempo. Estas variables se transmiten al computador desde el subsistema de **comunicación serial**, por medio del conversor **TTL a USB**.

En caso de que no satisfacer la condición del número de ceros, el sistema genera un nuevo *nonce* y se procesa de la misma forma anteriormente explicada, iterando hasta encontrar una coincidencia. En el caso de no hallar una coincidencia o que la información de entrada no constituya el mínimo de 1 bloque, la prueba es procesada como fallida y el sistema envía al computador una cadena de ‘f’ para indicar el estado de la prueba.

Todas las interacciones entre subsistemas y el manejo de la sincronización entre los mismos es orquestada por el **control Global.**

Los esquemáticos para cada vista jerárquica de este sistema se pueden encontrar en los anexos, ingresando en el siguiente enlace: [RTL jerárquico del sistema](https://livejaverianaedu-my.sharepoint.com/:b:/g/personal/fonseca_maria_javeriana_edu_co/ESbsF3GaCl5IgW6Pc4KvEjUBVKMsyxnZIsm4fHxFV_hmEg?e=QUQrRG).

### **[Diseño e implementación del subsistema ‘función](#_Módulo_de_minado) *[hash sha256’](#_37qjefsixob)***

A continuación, se presenta el diagrama de bloques del subsistema función *hash sha256*, el cual toma una entrada y la ingresa al módulo de cifrado donde se le aplica la función *hash sha256*, y posteriormente almacena los resultados en un registro de salida.

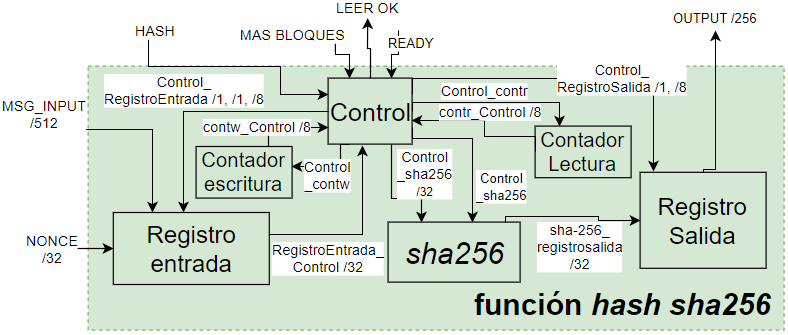
****

Figura 4. Diagrama de bloques de la unidad que implementa la función hash

El subsistema de función *hash sha256* mostrado en la Figura 4 recibe como entrada la información que se desea minar, junto con un *nonce*. Estas entradas son ingresadas al subsistema **registro entrada**, el cual en conjunto con el subsistema **contador escritura**, son coordinados por el **control,** y por medio de este se ingresa la información en el módulo ***sha256***. El **control** recibe como entrada las señales *ready* y *más bloques*, provenientes del control global, y se comunica con este por medio de la señal *leer ok*. Con base a esto se proporcionan las configuraciones necesarias tanto al módulo ***sha256*** como al subsistema **registro entrada**.

Al obtener un resultado total de toda la información recibida, el **control** coordina junto con el subsistema **contador lectura** el almacenamiento de la salida en el subsistema **registro salida**.

Los esquemáticos para cada vista jerárquica de este sistema se pueden encontrar en los anexos, ingresando en el siguiente enlace: [Diagrama de bloques función](https://livejaverianaedu-my.sharepoint.com/:b:/g/personal/fonseca_maria_javeriana_edu_co/EZJ8BfhlJ8pPha8FGxNKXU0BNYH3rIp2S6BnmAYYbHpa3A?e=TKhCgz) *[hash](https://livejaverianaedu-my.sharepoint.com/:b:/g/personal/fonseca_maria_javeriana_edu_co/EZJ8BfhlJ8pPha8FGxNKXU0BNYH3rIp2S6BnmAYYbHpa3A?e=TKhCgz)*.

**Concepción y Diseño**

Para la implementación de esta etapa inicialmente se planteó el uso de un módulo externo que realizara la función *hash sha*256, esto debido a que el diseño e implementación de este algoritmo de alta complejidad no es un objetivo de este trabajo de grado. Con base a los planteamientos iniciales se requería un sistema en *software*, equivalente al *hardware*, mediante el cual se pudiese comparar la veracidad de los resultados obtenidos.

Con base a esto, se investigó un módulo funcional que tuviese una versión para *software* y una para *hardware*. Finalmente se descargó un módulo desde el repositorio web *open sourced [GitHub](https://github.com/)*, en la que se encuentran diferentes códigos disponibles para uso público. Se optó por uno escrito tanto en lenguaje *Verilog* como en lenguaje *python*, lo cual suponía la investigación de su funcionamiento e instanciación en el proyecto.

El hipervínculo a la página de *GitHub* desde la cual se descargó la IP está disponible en el siguiente enlace*:* <https://github.com/secworks/sha256>.

Este módulo cuenta con un subsistema principal llamado *sha256* el cual cumple una función de envolvente o interfaz con los subsistemas que realizan la función *hash* sobre una entrada fija de 512 bits. El módulo cuenta con la opción de realizar el *hash sha256* versión 18.0 o el *sha224*, obteniendo un resultado a la salida de 256 bits. El subsistema principal *sha256* “*wrapper*” se encarga del manejo de entradas y salidas al cifrador, y en este es instanciado el subsistema principal que realiza la función *hash*, llamado *sha256\_core*. Este instancia los subsistemas *sha256\_w\_mem* y *sha256\_k\_constants* para su funcionamiento.

En el planteamiento del proyecto se especificó que el *hardware* sería implementado en una *FPGA*, por lo que se optó por usar los programas de *Altera* para el desarrollo de este, estableciendo el uso de *Quartus* para el desarrollo del código a un nivel de descripción *RTL (Register Tranfser Level)*, usando el lenguaje descriptivo de *hardware* *VHDL*; y el programa *ModelSim* para realizar simulaciones sobre los desarrollados realizados.

Una de las primeras metas o actividades planteadas para el desarrollo de este trabajo fue la instanciación del módulo ‘*sha256’* de *GitHub* en otro subsistema, de forma que se pudiese ingresar de forma sencilla la entrada deseada y almacenar el resultado obtenido de la función *hash*, llamado *digest*.

Se planteó como especificación del sistema permitir un tamaño máximo de 1 *Kbyte* para la información de entrada, sin embargo, el *sha256* recibe únicamente 512 bits a la entrada y a partir de los mismos produce un resultado de 256 bits a la salida. Teniendo en cuenta este dilema se investigó cómo cumplir con dicha especificación, y se llegó a lo siguiente:

Si se le quiere ingresar una entrada de información mayor a 512 bits, es necesario ingresar la información particionada en bloques de 512 bits, indicando si es el primero o el siguiente de la serie de bloques que conforman la entrada. Lo anterior suponía implementar un proceso para ingresar dicha serie de bloques hasta ingresar toda la información de entrada. Con el fin de cumplir con el formato de cadena de bloques requerido por el módulo, se encontró que la información debería pasar por un proceso de *padding* antes de ser ingresado al sistema de encriptado.

El *padding* utilizado por este está basado en particionar la entrada en 448 mod 512 bits, lo cual se refiere a particionar la entrada en bloques de 512 bits, donde la información válida en el último bloque de información sólo puede ocupar un máximo de 448 bits del bloque, y el resto corresponden al relleno. Para esto, se toman los bits de la entrada ubicándolos en orden de las posiciones más significativas a las menos significativas en cada bloque hasta llegar al último, el cual se rellena usando el procedimiento explicado de ‘*padding*’.

Instanciación de módulos escritos en lenguajes mixtos:

Para la realización de la instanciación de un módulo escrito en *verilog* en una entidad en *VHDL* se investigó en diferentes foros y se encontró el siguiente método. Declarar un *COMPONENT* para el módulo *sha256* y posteriormente llamar el módulo en la entidad, usando señales de tipo *ULOGIC* [1].

Con el fin de corroborar este método se planteó la realización de un sistema que recibiera cuatro entradas, dos para ejecutar una operación AND y las otras dos para ejecutar una operación OR, y obtuviera ambas salidas, a modo de prueba. El módulo para efectuar la función AND se realizó en *VHDL* y el módulo para efectuar la función OR en *verilog*.

A raíz de la implementación de la prueba anteriormente descrita, el resultado fue exitoso, obteniendo el desempeño esperado las dos compuertas lógicas sin complicaciones en la compilación ni en la simulación en *Quartus*.

Con base en esta prueba se obtuvo la confirmación de la funcionalidad del método de instanciación en *Quartus*, y al tiempo se descubrió que algunas versiones de *Modelsim* no soportan la compilación de proyectos escritos en lenguajes mixtos. Por lo tanto, se inició un proceso de pruebas de diferentes versiones de *Quartus* con su respectiva versión de *Modelsim*, con el fin de encontrar una que incluyera la biblioteca de las tarjetas *stratix IV*. Se llegó finalmente a *Quartus* *Prime* 15.0 y 18.1, los cuales cual poseen las versiones de *Modelsim* 10.1 y 10.5 respectivamente. Estas dos versiones suplen todos los requerimientos para la compilación de proyectos escritos en lenguajes mixtos.

Posteriormente se continuó con la investigación de la implementación del módulo ‘*sha256’* como instancia en un subsistema encargado del ingreso de las entradas y almacenamiento de las salidas en un registro.

Con el fin de determinar el procedimiento para manejar correctamente el módulo de *sha256*, se procedió a analizar el **código de prueba** *testbench* del módulo proveniente de *GitHub*, obteniendo los siguientes resultados del análisis.

Manejo del módulo *sha256*:

Adoptando la terminología en que una ‘palabra’ hace referencia a una partición de 32 bits de un mensaje más largo, al bloque se ingresa un bloque de entrada en 16 palabras. En el módulo *sha256* se espera que estas palabras sean almacenadas en las direcciones destinadas al almacenamiento de la entrada, es decir, de la 16 a la 31. Posteriormente se obtiene de la misma forma la salida (de 256 bits) en 8 palabras, las cuales se almacenan en las direcciones de la 32 a la 39 en el módulo al obtener el *digest*.

Además de las direcciones usadas para almacenar la entrada y el *digest*, el módulo cuenta con otras direcciones de memoria que cumplen con las siguientes funcionalidades: En las direcciones de memoria 0, 1 y 2 están almacenados los nombres y versiones del *hash sha256* que se utiliza. En la dirección 8 se almacenan las configuraciones del modo y si es el bloque inicial o el siguiente en una serie de bloques de un solo mensaje. Finalmente, en la dirección 9 se almacena el estado del *sha256*.

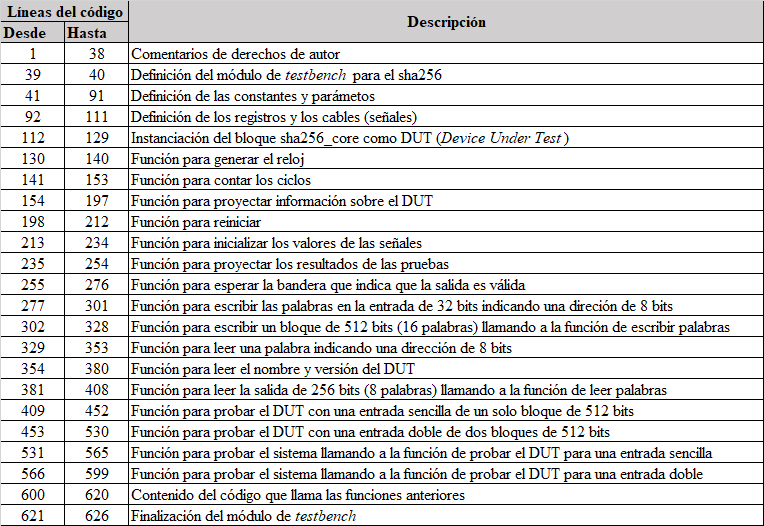


Figura 5. Tabla del análisis del código de pruebas de la IP de Github, dividida en líneas de codigo

Al realizar el análisis del código de pruebas del módulo sha256, se obtuvo la tabla mostrada en la Figura 5, en la cual se presenta la función de todas las secciones del código escrito en lenguaje SystemC, indicando las líneas en las que se encuentra cada sección.

Con base en el documento de pruebas del código de pruebas proporcionado en el siguiente enlace, [documento de pruebas](https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/examples/sha256.pdf). Se determinó la forma de ingresar entradas de tamaños mayores a los 512 bits de ongitud, es decir, la forma de ingresar varios bloques de entrada y como se conectan entre ellos para obtener un solo *digest*.

Se planteó el procedimiento para manejar correctamente el módulo, el cual se describe a continuación.

1. Inicializar las variables.
2. Reiniciar el módulo *sha256*.
3. Ingresar las palabras al módulo *sha256*.

**3.1.** Indicar la palabra en la señal *read\_data*, junto con su dirección, con las señales we=‘1’ y cs=‘1’.

**3.2.** Esperar un periodo de reloj.

**3.3.** Poner las señales we=‘0’ y cs=‘0’.

1. Ingresar las configuraciones de inicio.

**4.1.** Indicar la dirección en “8” y la palabra en: “5” para el bloque inicial y “6” para los que le siguen con las señales we=‘1’ y cs=‘1’.

**4.2.** Esperar un periodo de reloj.

**4.3.** Poner las señales we=‘0’ y cs=‘0’.

1. Esperar un periodo de reloj.
2. Esperar la bandera que indica que el *digest* es válido.

**6.1.** Indicar la dirección en “9”.

**6.2.** Esperar hasta que la salida sea diferente de cero.

1. Leer el *digest*.

**7.1.** Indicar la dirección a leer junto con las señales we=‘0’ y cs=‘1’.

**7.2.** Esperar un periodo de reloj.

**7.3.** Leer la salida de la señal *read\_data* y poner las señales we=‘0’ y cs=‘0’.

**7.4.** Guardar las palabras del *digest* en un registro de 256 posiciones.

En base a este procedimiento se planteó el funcionamiento específico para cada uno de los subsistemas del diagrama de bloques de función *hash sha256*, junto con la máquina de estados que se implementó, la cual incluye la funcionalidad de proporcionar varios bloques de entrada. La descripción de los demás subsistemas se presenta a continuación:

**Registro de entrada:** Registro que recibe una entrada en paralelo de 512 bits al estar activada la señal *write enable*, y la organiza en su salida en palabras de 32 bits, dependientes de una dirección. El envío de cada palabra debe estar en sincronía con la dirección de memoria, la cual es indicada por el control. Para realizar el envío de las palabras debe estar activada la señal *read enable*.

**Registro de salida:** Registro que recibe su entrada en palabras de 32 bits con su respectiva dirección, y las almacena dependiendo de la dirección indicada en un registro de 256 posiciones, el cual es la salida del subsistema función *hash sha256*. Este registro sólo guarda y muestra su salida mientras su habilitador se encuentre en ‘1’.

**Control IN OUT:** Bloque que se encarga de coordinar el tiempo para la escritura y lectura de datos y configuraciones para módulo de *sha256*, el registro de entrada y el registro de salida.

**Contador escribir:** Contador que va de 16 a 31 para indicar la dirección de escritura del registro de entrada al subsistema de sha256.

**Contador leer:** Contador que va de 32 a 39 para indicar las direcciones de lectura del módulo de sha256 al registro de salida.

Los esquemáticos específicos de este subsistema se pueden encontrar en los anexos ingresando en el siguiente enlace: [Diagramas de bloques Función](https://livejaverianaedu-my.sharepoint.com/:f:/g/personal/fonseca_maria_javeriana_edu_co/Eq0WTAzFD79OjdFSUuM4yFEBTMaiwWRdCwfZsPG7W-fbtA?e=X7YXcu) *[hash](https://livejaverianaedu-my.sharepoint.com/:f:/g/personal/fonseca_maria_javeriana_edu_co/Eq0WTAzFD79OjdFSUuM4yFEBTMaiwWRdCwfZsPG7W-fbtA?e=X7YXcu)*. De igual forma los códigos se pueden encontrar en la carpeta del proyecto de *Quartus Prime* en los anexos ingresando en el siguiente enlace: [Códigos en VHDL y Verilog](https://livejaverianaedu-my.sharepoint.com/:f:/g/personal/fonseca_maria_javeriana_edu_co/EmqhcnLXvdVNv2yg_r5Bg7MBGi_5dnp9d2-OOp3qNHCFFA?e=E61JAI).

A continuación, se presenta la máquina de estados desarrollada para coordinar las interacciones entre los subsistemas que componen la unidad función *hash sha256*.

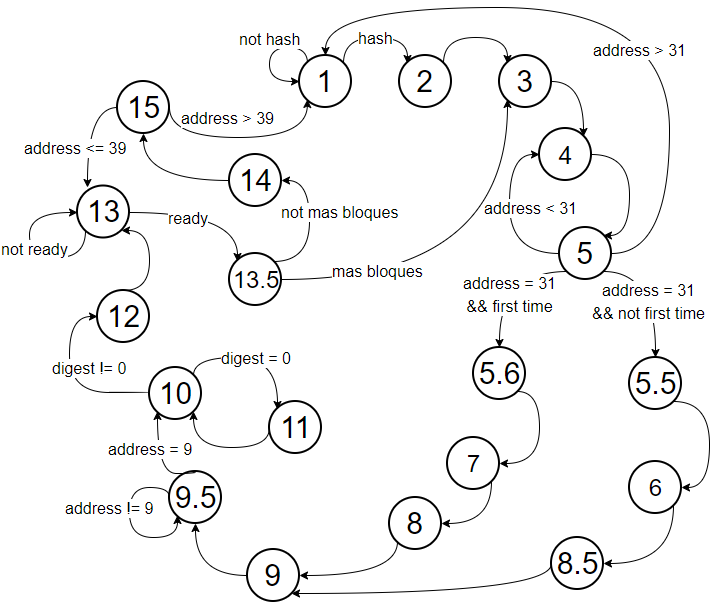


Figura 6. Máquina de estados finitos de la unidad que implementa la función hash

La máquina de estados mostrada en la Figura 6 inicia su funcionamiento en el estado uno donde se envía en todo momento el reinicio al módulo de *sha256*, y se inicializan las variables externas del sistema, en un estado en que la unidad de función *hash sha256* se considera inactiva. Con la llegada de la señal *‘hash’* se procede a activar esta unidad, pasando al estado dos y tres donde se habilita el almacenamiento de las entradas en el subsistema **registro de entrada**.

Acto seguido se llega al estado cuatro donde se ponen las señales cs = ‘1’ y we = ‘1’, junto con la dirección que toma el valor del subsistema **contador escribir** de la señal *contw* y la palabra de la entrada correspondiente a la dirección indicada. En este estado también se habilita el conteo del subsistema **contador escribir** y el modo de lectura del subsistema **registro de entrada**. Tras el estado cuatro el sistema sigue al estado cinco donde se ponen las señales cs = ‘0’ y we = ‘0’. Es en este estado se hace efectivo el conteo del contador *contw*. Mientras la dirección sea menor a 31 la máquina pasará al estado cuatro y se quedará en un ciclo de lectura entre los estados cuatro y cinco hasta llegar a la dirección 31.

Al alcanzar la dirección 31, del estado cinco seguirá al estado cinco\_cinco si la señal *primera vez* es cero, o al estado cinco\_seis si la señal *primera vez* es uno. La señal *primera vez* corresponde al estado del bloque, es decir, si el bloque es el inicial o no. En caso de ser el bloque inicial se ingresa en los estados cinco\_cinco, seis y ocho\_cinco donde se pone la configuración para el bloque inicial con las señales cs = ‘1’, we = ‘1’, la dirección en ‘8’ y la configuración ‘5’ en la entrada de datos del módulo *sha256*. En estos estados se pondrá la señal *primera vez* en ‘1’ y para el resto de los bloques, se ingresa en los estados cinco\_seis, siete y ocho donde se pone la configuración ‘6’ con las demás señales necesarias en la entrada de datos.

Los estados ocho y ocho\_cinco llegan al estado nueve donde se ponen las señales cs = ‘0’ y we = ‘0’. Sigue al estado nueve\_cinco donde se configura la dirección en ‘9’ con las señales cs = ‘1’ y we = ‘0’. Una vez ingresadas estas configuraciones al módulo, se sigue al estado diez, donde se entra en un pequeño ciclo hasta que la información en la salida de datos del módulo es diferente de cero.

Cuando esta información es diferente de cero, del estado diez se sigue al estado doce donde se ponen las señales cs = ‘1’ y we = ‘1’, y se llega al estado trece donde el sistema espera hasta que llegue la señal *ready*, proveniente del control global, la cual indica que ya es posible determinar si hay más bloques de información de la entrada o no.

Se procede al estado trece\_cinco donde se ponen las señales cs = ‘1’ y we = ‘0’, junto con la dirección que toma el valor del subsistema **contador leer** de la señal *contr* , y procede al estado tres nuevamente si hay más bloques de información, o al estado catorce si no hay más bloques de información. En el estado catorce se ponen las señales cs = ‘0’ y we = ‘0’ y se lee el *digest* en el subsistema **registro de salida**. Del estado catorce se sigue al estado quince donde se habilita el conteo del subsistema **contador leer** y mientras la dirección sea menor o igual a 39, se pasa al estado trece en un ciclo de lectura.

Cuando la dirección de lectura es mayor que 39, se pasa nuevamente al estado uno donde se desactiva la unidad función *hash sha256* hasta que vuelva a llegar la señal *‘hash’*.

El esquemático específico de este control se puede encontrar en los anexos, ingresando en el siguiente enlace: [Control Función](https://livejaverianaedu-my.sharepoint.com/:i:/g/personal/fonseca_maria_javeriana_edu_co/EQqnji4Y7ERBnwp4OPw78xoBwp3fF5a9wO_c3MfDzBqFug?e=32cpaG) *[hash](https://livejaverianaedu-my.sharepoint.com/:i:/g/personal/fonseca_maria_javeriana_edu_co/EQqnji4Y7ERBnwp4OPw78xoBwp3fF5a9wO_c3MfDzBqFug?e=32cpaG)*.

### **[Diseño e implementación del subsistema ‘generación](#_Módulo_de_minado) *[nonce’](#_37qjefsixob)***

A continuación, se presenta el diagrama de bloques de la unidad que genera salidas de *nonce* a partir de una semilla tomada de la información de entrada, de forma que la generación de las secuencias de *nonce* resultantes sea replicable.

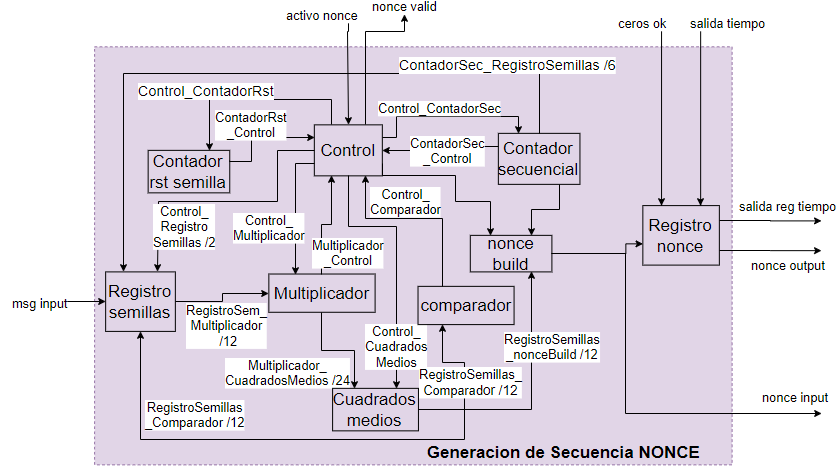
****

Figura 7. Diagrama de bloques de la unidad que implementa la generación de nonce

La unidad de generación *nonce* ilustrada en la Figura 7 recibe como entrada una parte del mensaje de entrada del sistema de minado, junto con la señal activo *nonce*, con la cual se inicia el procesamiento de la unidad. El subsistema de **registro semillas** toma para el primer caso la semilla inicial del mensaje de entrada, la cual se envía al subsistema **multiplicador**, cuya salida llega al subsistema de **cuadrados medios** donde se genera la primera salida pseudo-aleatoria. Esta salida se convierte en la nueva semilla para el siguiente *nonce.* A su vez, esta salida pasa por el subsistema **comparador** donde se corrobora que el resultado sea diferente de cero, y al subsistema ***nonce* build** donde junto con la salida del subsistema **contador secuencial**, se construye la salida de la unidad, es decir, el *nonce* que se usará por el sistema en esta iteración.

En el caso de que esta salida sea cero o hayan pasado 10 iteraciones partiendo de una semilla, determinada por el subsistema **contador reinicio semilla**, se lleva a cabo el reinicio de la semilla para el cual, la nueva semilla se tomará de la semilla anterior y del contador secuencial. El subsistema de **control** se encarga de coordinar y sincronizar todas las operaciones y configuraciones entre los demás subsistemas mencionados.

Los esquemáticos para cada vista jerárquica de este sistema se pueden encontrar en los anexos, ingresando en el siguiente enlace: [Diagrama de bloques generación](https://livejaverianaedu-my.sharepoint.com/:b:/g/personal/fonseca_maria_javeriana_edu_co/EfJAnbddrHdLk-y_-p1jtwYBk7YpGA_kDpPLW-6JmmVXLQ?e=PaSLDF) *[nonce](https://livejaverianaedu-my.sharepoint.com/:b:/g/personal/fonseca_maria_javeriana_edu_co/EfJAnbddrHdLk-y_-p1jtwYBk7YpGA_kDpPLW-6JmmVXLQ?e=PaSLDF)*.

**Concepción y Diseño**

Con el fin de realizar seguimiento a los resultados obtenidos en cada iteración del sistema, se realizó una investigación de diferentes procedimientos para generar secuencias de números pseudo-aleatorias para generar el *nonce*.

En base a dicha investigación se encontró que es recomendado tener tanto una porción pseudo-aleatoria como una porción secuencial. De manera que los bits más significativos pertenezcan a la porción pseudo-aleatoria mientras que los bytes menos significativos sean secuenciales[2].

El *nonce* en general tiene que ser único en cada iteración, por lo que en el caso de que la sección pseudo-aleatoria del *nonce* permanezca fija durante toda la vida útil de una semilla, la sección secuencial garantiza que el *nonce* siempre será único[2].

De los métodos investigados para la generación de secuencias pseudo-aleatorias se optó por la utilización del método de los cuadrados medios, detallado anteriormente en el [marco teórico](#_MARCO_TEÓRICO).

Para el diseño e implementación de una unidad generadora de *nonce* se decidió que el tamaño del *nonce* sería fijo, y se decidió utilizar 32 bits. Este con una porción pseudo-aleatoria de 12 bits y una porción secuencial de 20 bits, para el total de 32 bits de *nonce*.

Se realizó el planteamiento de entradas y salidas del subsistema, para el cual las entradas constan de una señal para activar su funcionamiento, y un bus de datos conteniendo la semilla inicial x0. De la salida se obtiene un bus de datos conteniendo el *nonce* generado, junto con una señal que indica que la salida del *nonce* en el bus de datos ha sido exitosamente generada y es válida.

Los principales inconvenientes del método de los cuadrados medios, es la fuerte tendencia a degradar a cero, y de la secuencia de números generados a repetirse de forma cíclica tras una secuencia corta. Con base a esto, al momento del diseño se tomaron en cuenta estos inconvenientes y se concertó para el diseño de la unidad generadora de *nonce* los siguientes lineamientos:

* El origen de la semilla provendría del mensaje de entrada a la función *hash*, tomando los 6 bits más significativos (MSB) y los 6 bits menos significativos (LSB) del mismo.
* La semilla inicial x0 estaría constituida inicialmente de 12 bits de forma que al elevar x0 al cuadrado, se obtendría una señal de 24 bits.
* Para evitar la degradación a cero y caer en un ciclo repetitivo, se reiniciaría la semilla inicial cada 10 salidas válidas de *nonce* y cada vez que la semilla sea degradada a cero.
* Para reiniciar la semilla, se toman los 6 LSB de la porción secuencial y se concatenan con los 6 MSB de la última semilla almacenada, posicionando los bits de la parte secuencial en las 6 posiciones más significativas de la nueva semilla, y los bits de la semilla anterior en las 6 posiciones menos significativas de la nueva semilla.
* La parte secuencial se realizaría sumando una unidad en cada ocasión. Para 20 bits, se contaría de 0 a 220-1, es decir, hasta 1048575.

Con base en estos lineamientos se planteó el funcionamiento específico para cada uno de los subsistemas del diagrama de bloques de generación *nonce*, junto con la máquina de estados que se implementó. La descripción de los subsistemas se presenta a continuación:

**Registro Semillas:** Registro que selecciona entre la entrada de semilla xo externa al bloque, la semilla proveniente de la salida anterior, y la semilla generada internamente para su reinicio. En este subsistema también se realiza la concatenación de las señales que componen la semilla generada para el reinicio de la misma.

**Multiplicador:** Subsistema que se encarga de elevar al cuadrado la entrada proveniente del subsistema de registro semillas.

**Cuadrados medios:** Subsistema que implementa la selección de los bits centrales del resultado de la multiplicación.

**Comparador:** Subsistema que revisa si la semilla se debe reiniciar, al revisar si la semilla fue degradada a cero o si ya han pasado 10 salidas válidas desde el último reinicio de la semilla.

**Constructor de *nonce*:** Subsistema que construye el *nonce* al concatenar la parte secuencial con la parte pseudo-aleatoria del *nonce*, en una sola salida.

**Control:** Subsistema que se encarga de coordinar el proceso completo de la generación del *nonce*, con el fin de que el mismo funcione de acuerdo con los lineamientos estipulados. También se encarga de que solo se genere un nuevo *nonce* cada vez que sea requerido por el sistema de minado, de forma que en el caso contrario el subsistema de la generación del *nonce* debe permanecer inactivo.

**Contador secuencial:** Contador que va desde 0 hasta 2n-1, con n=20, cuenta hasta 1048575.

**Contador reinicio semilla:** Contador que cuenta cada 10 salidas válidas diferentes de *nonce*, para reiniciar la semilla.

**Registro *nonce*:** Registro que almacena la salida tanto del *nonce* como del tiempo, con la llegada de la señal *ceros ok*. Es desde este subsistema que se obtiene el valor final resultante del *nonce* para transmitir a la salida del sistema de minado en *hardware*.

Los esquemáticos específicos de esta unidad se pueden encontrar en los anexos, ingresando en el siguiente enlace: [Diagramas de bloques generación](https://livejaverianaedu-my.sharepoint.com/:b:/g/personal/fonseca_maria_javeriana_edu_co/ESEdHUcy4P5Aqd1gVZBvUrkBPkT6ZjMuKgVOj3Z48pp6gw?e=qG0PgF) *[nonce](https://livejaverianaedu-my.sharepoint.com/:b:/g/personal/fonseca_maria_javeriana_edu_co/ESEdHUcy4P5Aqd1gVZBvUrkBPkT6ZjMuKgVOj3Z48pp6gw?e=qG0PgF)*. De igual forma los códigos se pueden encontrar en la carpeta del proyecto de *Quartus Prime* en los anexos ingresando en el siguiente enlace: [Códigos en](https://livejaverianaedu-my.sharepoint.com/:f:/g/personal/fonseca_maria_javeriana_edu_co/EmqhcnLXvdVNv2yg_r5Bg7MBGi_5dnp9d2-OOp3qNHCFFA?e=E61JAI) *[VHDL](https://livejaverianaedu-my.sharepoint.com/:f:/g/personal/fonseca_maria_javeriana_edu_co/EmqhcnLXvdVNv2yg_r5Bg7MBGi_5dnp9d2-OOp3qNHCFFA?e=E61JAI)* [y](https://livejaverianaedu-my.sharepoint.com/:f:/g/personal/fonseca_maria_javeriana_edu_co/EmqhcnLXvdVNv2yg_r5Bg7MBGi_5dnp9d2-OOp3qNHCFFA?e=E61JAI) *[Verilog](https://livejaverianaedu-my.sharepoint.com/:f:/g/personal/fonseca_maria_javeriana_edu_co/EmqhcnLXvdVNv2yg_r5Bg7MBGi_5dnp9d2-OOp3qNHCFFA?e=E61JAI)*.

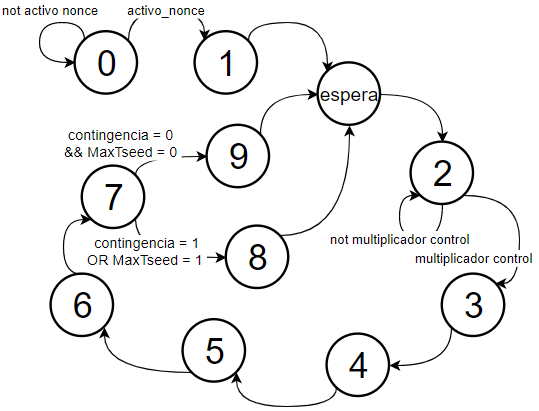


Figura 8. Máquina de estados finitos de la unidad que implementa la generación del nonce

La máquina de estados de la Figura 8 inicia su funcionamiento en el estado cero donde se envía en todo momento el reinicio a contadores y registros, y se inicializan las variables externas del sistema, en un estado en que la unidad de generación *nonce* se considera inactiva. Con la llegada de la señal *activo nonce* se procede a activar la unidad, pasando al estado uno y al estado espera, donde se habilita el almacenamiento de la semilla inicial externa x0 en el subsistema **registro semillas**. En el estado espera se habilita un conteo para en contador secuencial. Posteriormente pasa al estado dos donde el subsistema **multiplicación** toma la entrada del subsistema **registro semilla** y eleva este número al cuadrado.

Por medio de la señal *multiplicador\_control* se determina que la multiplicación ha terminado y se pasa al estado tres, donde el subsistema **cuadrados medios** toma el resultado de la multiplicación, y luego al estado cuatro donde se seleccionan los bits que conforman la sección pseudo-aleatoria del *nonce*. Posteriormente se pasa al estado cinco, el cual es un estado de espera, y al seis en el cual se activa la toma de la entrada para el subsistema ***nonce* build**. Tras el estado seis, se procede al siete donde se pone en ‘1’ la señal *nonce* valid, y la unidad de generación *nonce* entra en un estado de inactividad mientras la señal *activo* *nonce* sea cero.

Una vez reactivada la unidad con la señal activo *nonce* en ‘1’, se procede a generar el próximo *nonce*, de forma que si no se ha detectado ninguna de las condiciones para el reinicio de semilla, es decir, el contador de reinicio de semilla o el resultado de la semilla anterior como cero, se procede al estado ocho donde la nueva semilla se toma del resultado anterior, y en el caso contrario se procede al estado nueve tomando la semilla del proceso de reinicio de semilla anteriormente explicado. Se sigue tanto al estado ocho o al nueve dependiendo del valor que debe tomar la nueva semilla, y se vuelve al estado espera donde se itera en el mismo ciclo hasta generar el *nonce* “ganador”.

En el caso en que el *nonce* ganador no se encuentre cuando el contador secuencial llegue a su máximo conteo, esta señal sale directamente del subsistema **contador secuencial** y llega al control global donde se determina que el sistema ha fallado en encontrar un nonce válido para una entrada, ya que en este punto no se puede asegurar que el siguiente *nonce* generado sea único con respecto a los producidos anteriormente.

El esquemático específico de este control se puede encontrar en los anexos ingresando en el siguiente enlace: [Control generación](https://livejaverianaedu-my.sharepoint.com/:i:/g/personal/fonseca_maria_javeriana_edu_co/ERuZwQ37emJAgje3vEvKw-MBV5DPXtTB4eifUkJDWlMH_A?e=LQrSZF) *[nonce](https://livejaverianaedu-my.sharepoint.com/:i:/g/personal/fonseca_maria_javeriana_edu_co/ERuZwQ37emJAgje3vEvKw-MBV5DPXtTB4eifUkJDWlMH_A?e=LQrSZF)*.

### **[Diseño e implementación del subsistema ‘comunicación serial’](#_Módulo_de_minado)**

A continuación, se presenta el diagrama de bloques de la unidad que implementa la comunicación serial, el cual comprende la recepción de la entrada al sistema de minado (el mensaje y el número de ceros a comparar) desde un computador externo hacia la *FPGA*. Y la transmisión la salida del sistema de la *FPGA* hacia el computador.

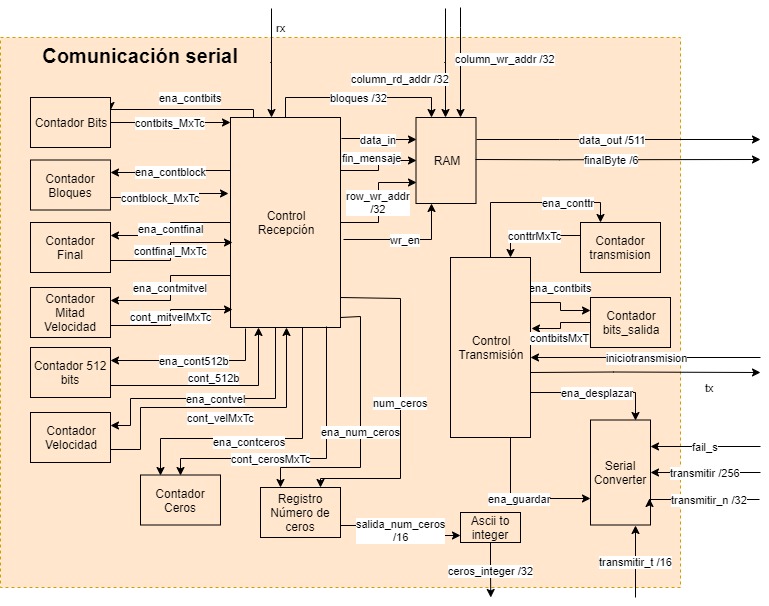


Figura 9. Diagrama de bloques del subsistema que implementa la comunicación serial.

El subsistema de la comunicación serial ilustrado en la Figura 9 consta de dos secciones. La sección encargada de la recepción que recibe como entrada la señal *rx*, que es de donde provienen los datos del mensaje de forma serial. Esta señal inicialmente está en ‘1’ y comienza a recibir cuando se pone en ‘0’, al llegar un bit de inicio que indica el comienzo de la recepción. La segunda sección corresponde a la transmisión, cuya entrada es la señal *tx* por donde se envían los datos de salida de forma serial.

Los esquemáticos para cada vista jerárquica de este sistema se pueden encontrar en los anexos ingresando en el siguiente enlace: [Diagrama de bloques comunicación serial](https://livejaverianaedu-my.sharepoint.com/:b:/g/personal/fonseca_maria_javeriana_edu_co/EbUNMYSuexhDmxu6hE7eulEBUQwaneA1N5MycDqC8yvC7A?e=ZeYzax).

**Concepción y Diseño**

Para la implementación del bloque encargado de la comunicación serial se decidió implementar una comunicación *UART* (*Universal Asyncronos Receiver and Transmitter)* entre el computador y la *FPGA,* haciendo necesaria la utilización de un conversor USB a TTL, que es un elemento electrónico programado para convertir el protocolo de comunicación serial al protocolo de comunicación *USB* y establecer la conexión, en este caso entre la *FPGA* y el computador. Este elemento es necesario ya que tiene niveles de voltaje *TTL* compatibles con la *FPGA* (0V - 5V), con pines específicos de transmisión (*Tx*) y recepción (*Rx*) y voltajes de salida de 5 V y 3.3 V.

Para efectos de comunicación serial se debe establecer una velocidad de transmisión, que en este caso es de 9600 Baudios [Bit/s]. Esta es la velocidad a la cual se recibirá y se transmitirá desde la unidad de comunicación. Inicialmente se planteó el uso de velocidades más altas, sin embargo, durante la realización pruebas sencillas con el conversor se presentaron considerables problemas de ruido, atribuidos las condiciones de la línea de transmisión. Con base a esto se decidió utilizar la velocidad anteriormente enunciada, 9600 baudios.

El proceso de la recepción se realiza por medio de un contador de velocidad que cuenta ciclos de reloj hasta llegar a 104 µs (tiempo de permanencia de un bit al ser transmitido a 9600 Baudios). Antes de esto es necesario implementar un contador que cuente hasta la mitad de la velocidad, con el fin de que el contador del tiempo completo inicie tomando los valores de la señal aproximadamente en la mitad de cada bit, garantizando la toma de los bits de forma correcta. También se debe establecer el paquete de datos que en este caso es de la forma: Bit de inicio + 8 bits de datos + 1 bit de parada.

Inicialmente se utilizó el software *Realterm* para enviar y recibir los datos, por su bajo nivel de complejidad al usar. Posteriormente, con el fin de dar cumplimiento a los objetivos del proyecto, se implementó una interfaz en *software*, con la cual fuera posible el ingreso de las entradas y obtención de las salidas de manera sencilla. El *software* se expone en la sección: *[Desarrollo en software](#_Desarrollo_en_software)*.

Para el desarrollo de esta unidad fue necesario dividirla en dos secciones relevantes:

* + - 1. La recepción:

Para esta se implementaron los siguientes subsistemas: el subsistema **control recepción**, el subsistema **contador de ceros**, el subsistema **contador de 512 bits**, el subsistema **contador de bloques**, el subsistema **contador de bits**, el subsistema **contador de velocidad**, el subsistema **registro de número de ceros**, el subsistema **conversión de ascii a integer** y el subsistema **memoria RAM**.

Para la recepción de los datos se definió el orden de la información del mensaje de entrada, de la siguiente forma: los 16 bits que llegan inicialmente, indican el número de ceros, posteriormente llega el mensaje resultante del proceso de ‘*padding*’ (desde 512 bits hasta 4096 bits), el cual incluye al inicio del mismo el número de bits que serán reemplazados por el *nonce,* es decir 32 bits de información en cero, sobre la cual será reemplazado el *nonce* adentro del sistema*.*

Después de definir este orden, se estableció el procedimiento a seguir: los bits que llegan inicialmente, como se definió anteriormente son los del número de ceros, los cuales se almacenan en el registro **número de ceros** mediante el subsistema **contador de ceros**, el subsistema **contador de velocidad** y el subsistema **contador de bits.** La salida de este registro llega a la salida del subsistema **conversor de ascii a integer**, para que pueda ser interpretado con el subsistema **comparador de ceros** que será explicado más adelante.

Luego de recibir el número de ceros, se comienza con la recepción del mensaje que será almacenado en el subsistema **memoria RAM**. Esta memoria comprende 512 filas por 17 columnas, de la siguiente forma: Se almacenan los bloques en cada columna mediante la salida del subsistema **contador de bits**, el subsistema **contador de 512 bits**, el subsistema **contador de bloques** y el subsistema **contador de velocidad**. Una vez finalizada la función de los mismos, el mensaje que se va a recibir activa un contador final, con el cual, si se cumple un tiempo de inactividad en la línea de transmisión, se indica el fin del mensaje y se espera a que se acabe todo el proceso de minado del sistema, para volver al estado inicial y esperar un nuevo mensaje.

Los subsistemas planteados fueron los siguientes:

**Control recepción:** Es el subsistema encargado de coordinar la velocidad de transmisión con la recepción de datos, por medio de la activación y desactivación de los distintos contadores para recibir el mensaje y guardarlo en la memoria RAM y en el registro de número de ceros.

**RAM:** Es el subsistema encargado de almacenar todo el mensaje en filas de 512 bits y columnas de 16 posiciones dependiendo de la longitud del mensaje que llega.

**Registro número de ceros:** Es el subsistema encargado de recibir la información correspondiente al número de ceros requerido y almacenarla para luego enviarla al conversor de ascii a integer.

**Ascii to integer:** Es el subsistema encargado de recibir el número de ceros requerido en ascii y convertirlo a integer para ser procesado de manera correcta por el sistema.

**Contador bits:** Es el subsistema encargado de contar de 0 a 7 bits para saber cuándo se obtuvo la recepción del byte completo.

**Contador mitad velocidad:** Es el subsistema encargado de contar hasta la mitad de un conteo de la velocidad total, es decir, si un conteo es culminado al pasar 104 µs, este cuenta hasta 52 µs. Esto con el fin de poder garantizar a partir de la cuenta de la mitad de la velocidad, que siempre se tomará el bit en el momento correcto.

**Contador velocidad:** Es el subsistema encargado de contar a la velocidad de recepción (9600 baudios), ya que a partir de que se cuenta por primera vez la mitad se un conteo con el contador mencionado anteriormente, al contar la velocidad total se garantiza la toma cada bit aproximadamente en la mitad de su transmisión, asegurando que fue tomado correctamente.

**Contador 512 bits:** Es el subsistema encargado de contar los 512 bits que constituyen cada bloque, para almacenar en la memoria dependiendo de la salida de este contador.

**Contador bloques:** Es el subsistema encargado de contar las veces en que se recibe un bloque, para almacenar cada bloque en la memoria RAM dependiendo del número de columna proveniente de la salida de este contador.

**Contador final:** Es el subsistema encargado de contar el doble de ciclos de reloj que toma el subsistema de la velocidad de transmisión, mientras espera un nuevo bloque de mensaje. Si se cumple este tiempo y no ha llegado más información, se da el fin del mensaje.

**Contador ceros:** Es el encargado de contar la recepción de los primeros 16 bits del mensaje recibido por la comunicación serial, para almacernarlos en el registro del número de ceros.

Los esquemáticos específicos de este subsistema se pueden encontrar en los anexos ingresando en el siguiente enlace: [Diagrama de bloques de la recepción](https://livejaverianaedu-my.sharepoint.com/:b:/g/personal/fonseca_maria_javeriana_edu_co/EexYHqTJRcxHibUd3-btAREBbZx1kodhFukwX3Pfm1SEiw?e=CoamRE).

A continuación, se presenta la máquina de estados desarrollada para coordinar las interacciones entre los bloques que componen la recepción serial.

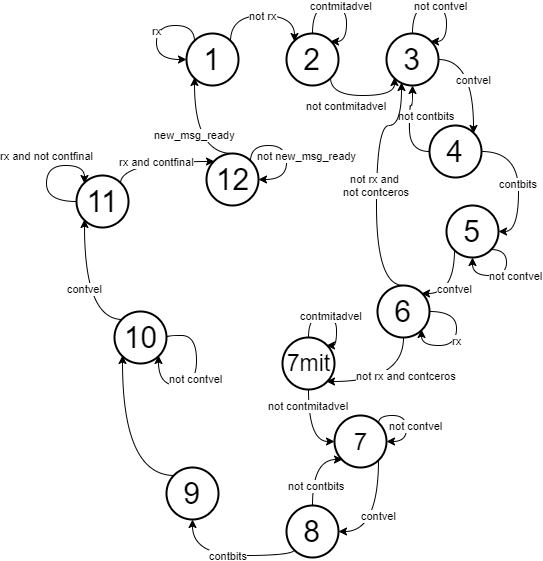


Figura 10. Máquina de estados finitos que implementa la recepción de la comunicación serial.

La máquina de estados mostrada en la Figura 10 comienza en el estado uno mientras la señal de recepción *rx* está en ‘1’. Cuando se pone en ‘0’ cambia al estado dos, el cual indica un primer bit de inicio de datos y se activa el contador de la mitad de la velocidad. Una vez termina de contar la mitad, se pasa al estado tres, en donde tras haber asegurado el muestreo de los datos en la mitad de los bits, se activa el contador de la velocidad para tomar el primer bit. Posteriormente se pasa al estado cuatro donde se almacenan los 16 bits que indican el número de ceros, en el registro de número de ceros.

Del estado cuatro se vuelve al estado tres si no ha terminado de almacenar los 16 bits del número de ceros, y se queda en un pequeño ciclo entre estos estados. Una vez se almacenaron los 16 bits, pasa al estado cinco donde espera que pase una última vez el tiempo de la velocidad de transmisión que indica el bit final, es decir, *rx* en ‘1’. Después de almacenar en el registro el número de ceros, pasa al estado seis en donde vuelve a esperar un bit de inicio en ‘0’ proveniente de la señal *rx*, y luego hace el mismo proceso de la adaptación de las velocidades en los estados siete mit y siete. Entre el estado siete y ocho se comienza a almacenar el mensaje en bloques de 512 bits, con un máximo de16 bloques por medio del contador de 512 bits que indica la fila y el contador de bloques que indica la columna de la memoria RAM. Cada vez que llegan 512 bits, se aumenta el contador de bloques y si llegan más se vuelve a hacer el mismo proceso descrito. De lo contrario se activa el contador final con el cual se termina la recepción del mensaje.

El esquemático específico de este control se puede encontrar en los anexos ingresando en el siguiente enlace: [Control recepción](https://livejaverianaedu-my.sharepoint.com/:b:/g/personal/fonseca_maria_javeriana_edu_co/EexYHqTJRcxHibUd3-btAREBbZx1kodhFukwX3Pfm1SEiw?e=CoamRE).

* + - 1. La transmisión

Para la transmisión se utilizaron algunos de los contadores de la recepción, también se usó un registro serial y conversor con el cual se desplazan los datos y se envían a la misma velocidad de 9600 baudios indicada anteriormente. También utiliza un control para establecer en qué momento transmitir.

Para el inicio de la transmisión se debe tener en cuenta que sólo se va a transmitir cuando la salida del subsistema función *hash sha256* sea la esperada y se haya encontrado el *nonce* ganador que satisface el requerimiento del número de ceros que se ingresó a la entrada del sistema de minado. Cuando esto sucede la señal *comparar\_ceros* se pone en ‘1’. Una vez se recibe esta señal, se empieza con la transmisión del valor *hash* y el *nonce* encontrado junto con el tiempo en que se demoró el sistema de minado en encontrar el valor *hash* ‘ganador’. Adicionalmente, en el registro serial donde se almacena esta información, se incluye para cada byte el respectivo bit de inicio y bit de parada, con el fin de la realización efectiva de la transmisión.

Los bloques planteados son los siguientes:

**Control transmisión:** Es el subsistema encargado de coordinar transmisión de los datos, a una velocidad de transmisión igual a la del receptor, hasta que se termine de enviar la totalidad de la salida.

**Contador transmisión:** Es el subsistema encargado de contar a la velocidad de transmisión (9600 baudios), para enviar los bits a la velocidad que va a recibir el receptor.

**Contador bits de salida:** Es el subsistema encargado de contar los bits enviados; un total son 390 contando los bits de inicio y fin de parada requeridos en cada byte a transmitir. El mensaje a enviar se compone de 256 bits provenientes del hash, 32 bit provenientes del respectivo *nonce* y 16 bits provenientes del tiempo total.

**Serial converter:** Es el subsistema encargado de guardar en paralelo los bits que se van a transmitir, con su respectivo bit de inicio y bit de parada, e ir desplazándose en las posiciones de la salida serial para realizar la transmisión.

Los esquemáticos específicos de este subsistema se pueden encontrar en los anexos ingresando en el siguiente enlace: [Diagrama de bloques transmisión](https://livejaverianaedu-my.sharepoint.com/:b:/g/personal/fonseca_maria_javeriana_edu_co/EavwQa3C0g9Cn_p8JIjBPmABLJ9pVTRmPxg8-vbqVGLPXQ?e=MBJgPg).

A continuación, se presenta la máquina de estados desarrollada para coordinar las interacciones entre los bloques que componen la transmisión serial.

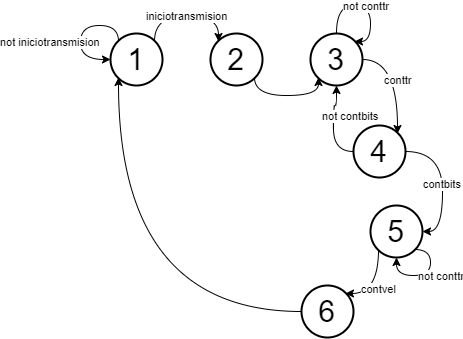


Figura 11. Máquina de estados finitos que implementa la transmisión de la comunicación serial.

La máquina de estados finitos de la transmisión mostrada en la Figura 11 comienza en el estado uno, en el cual se espera a que la señal *iniciotransmision* se ponga en ‘1’ para comenzar la transmisión de datos. A continuación, se sigue al estado dos y tres, donde se comienzan a contar los ciclos de reloj necesarios para la velocidad de transmisión 9600 baudios. Una vez termina, se pasa al estado cuatro donde realiza el envío y desplazando de los bits mientras la cuenta de bits transmitidos sea menor a 390. Primero transmiten los 256 bits del hash final, seguidos por los 32 bits del *nonce* y finalizando con los 16 bits que indican el tiempo que le tomó al sistema minar la entrada. Una vez se transmiten todos los bits, se espera el último bit final y se vuelve al estado uno.

El esquemático específico de este control se puede encontrar en los anexos ingresando en el siguiente enlace: [Control transmisión](https://livejaverianaedu-my.sharepoint.com/:i:/g/personal/fonseca_maria_javeriana_edu_co/EUWgzTaRa-pNts0wV4F_H9cBrxaVW7dmn_pPBJ6qI5rCwQ?e=bsHj7a).

### **[Diseño e implementación del subsistema ‘unidad de tiempo’](#_Módulo_de_minado)**

En la presente sección se describe el sistema dedicado a medir el tiempo que demora el módulo de minado de *hardware* en encontrar el *hash* y *nonce* válidos para satisfacer el requerimiento del número de ceros deseados.

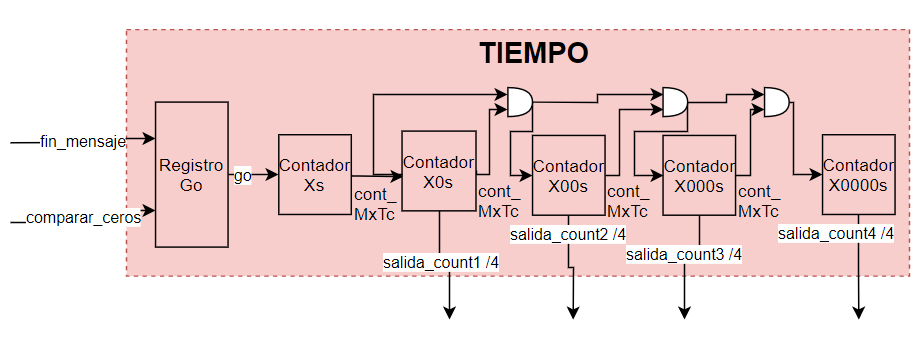
****

Figura 12. Diagrama de bloques de la unidad de tiempo.

*.*

En el diagrama de bloques presentado en la Figura 12 se presenta el subsistema que toma el tiempo en que se demora el sistema en realizar el proceso de minado. Este tiempo se toma desde que el mensaje se ha guardado totalmente en la memoria, hasta que se encuentra la salida correspondiente al hash del mensaje con el número de ceros correcto, es decir, no se toma el tiempo de transmisión y recepción de la información.

Para iniciar con el conteo del tiempo, el sistema espera a que la señal *fin\_mensaje,* es decir, la señal que indica cuando ya se recibió todo el mensaje de entrada, este en ‘1’, y se para el tiempo cuando la señal *comparar\_ceros*, es decir, la señal que indica que ya se encontró el número de ceros deseados, esté en ‘1’. Para el conteo del tiempo se estableció una cuenta máxima de 1000 µs, y para ello se implementó un primer contador que contaba hasta 1 µs, luego se implementó un segundo contador que contaba de 1 - 9 el número de veces que transcurría 1 µs, es decir contaba hasta 9 µs. Luego se implementó un tercer contador que contaba el número de veces que transcurrían 9 µs, es decir que contaba hasta 99 µs , y finalmente un cuarto contador que contaba hasta 999 veces, es decir 9999 µs. Esta fue la medición del tiempo que se planteó inicialmente, no obstante, durante la implementación del protocolo de pruebas fue necesario un ajuste del tiempo dependiendo del número de ceros requerido y el número de información de entrada, ya que con el aumento de estos dos parámetros, se generaron necesidades con respecto a la escala del tiempo.

Ya que el tiempo fue diseñado dependiente del primer contador, y los siguientes contadores aumentan en unidades, se tomó el tiempo ajustando el valor de conteo máximo del primer contador de la siguiente manera:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#Bloque/#ceros** | 1 | 2 | 3 | 4 |
| 1 | Xs: | Xs: | Xs: | Xs: |
| 2 | Xs: | Xs: | Xs: | Xs: |
| 3 | Xs: | Xs: | Xs: | Xs: |
| 4 | Xs: | Xs: | Xs: | Xs: |
| 5 | Xs: | Xs: | Xs: | Xs: |
| 6 | Xs: | Xs: | Xs: | Xs: |
| 7 | Xs: | Xs: | Xs: | Xs: |
| 8 | Xs: | Xs: | Xs: | Xs: |

Figura 13. Tabla de valores del contador inicial de la unidad de tiempo por bloque de ceros por número de ceros.

La realización de este subsistema se llevó a cabo por medio de distintos contadores:

**Contador Xs:** Este contador se realizó inicialmente para una cuenta 100 conteos de reloj, lo que equivale a 1 µs, el cual al llegar a su máximo, activa una bandera que es el habilitador de conteos del siguiente contador. Posteriormente se editó como se indica en la Figura 13.

**Contador X0s:** Este contador tiene como habilitador la bandera de 1 µs, con la cual, al recibir dicha señal cuenta 1 vez en cada ocasión hasta llegar a 9, valor en el cual activa una bandera cuando cuenta 9 µs.

**Contador X00s:** Este contador tiene como habilitador la bandera de 10 µs con el cual, al recibir dicha bandera cuenta 1 vez en cada ocasión hasta llegar a 9, valor en el cual activa una bandera cuando cuenta 99 µs.

**Contador X000s:** Este contador tiene como enable la bandera de 100 µs con el cual, al recibir dicha bandera cuenta 1 vez en cada ocasión hasta llegar a 9, valor en el cual activa una bandera cuando cuenta 999 µs.

Cada contador tiene como salida una señal de 4 bits con la cual se indica el número del dígito correspondiente, los cuales son transmitidos como resultados de tiempo medido para ser recibidos por el *software*.

Los esquemáticos específicos de este subsistema se pueden encontrar en los anexos ingresando en el siguiente enlace: [Diagrama de bloques tiempo](https://livejaverianaedu-my.sharepoint.com/:b:/g/personal/fonseca_maria_javeriana_edu_co/EcH04YDjX6NGnvwzabn2vjwBE-IWklYfncHalX-Fht-WrA?e=ctwwRz).

### **[Diseño e implementación del subsistema ‘comparar ceros’](#_Módulo_de_minado)**

En la presente sección se describe el funcionamiento del subsistema que se encarga de hacer la comparación del número de ceros iniciales de la salida de la función *hash*, y determina si se ha satisfecho el requerimiento del número deseado de ceros.

Para la comparación se reciben los bits más significativos de la salida de la función *hash* junto con la variable ‘ceros integer’, que indica la cantidad de ceros que se espera tener a la salida. Con base en esto, si la cantidad de ceros corresponde con los ceros presentes de la entrada proveniente de la función *hash*, se pone en ‘1’ la señal de ceros ok, de lo contrario ceros ok permanece en ‘0’.

El esquemático específico de este subsistema se puede encontrar en los anexos ingresando en el siguiente enlace: [Diagrama de bloques comparar ceros](https://livejaverianaedu-my.sharepoint.com/:b:/g/personal/fonseca_maria_javeriana_edu_co/EZR3x_kP0f1JmRwM5zAax-YBz6_a3CjPjWAy4qAXRwESsw?e=cwFRwV).

### **[Diseño e implementación del subsistema ‘control global’](#_Módulo_de_minado)**

En la presente sección se describe el funcionamiento del Control Global, el cual utiliza un contador externo como apoyo para el procesamiento de cadenas de bloques en la entrada. Ya que este bloque es meramente compuesto por el control y el contador de apoyo, se describirá únicamente el funcionamiento de la máquina de estados finitos.

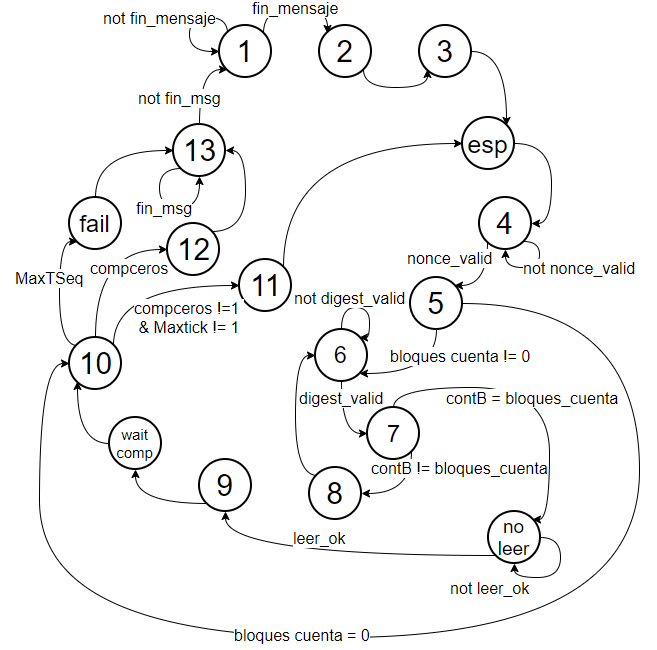
****

Figura 14. Máquina de estados finitos del Control Global.

La máquina de estados ilustrada en la Figura 14 inicia su funcionamiento en el estado uno, donde se manda la señal de reinicio al contador de apoyo contB, el sistema permanece en este estado mientras la señal *fin mensaje* sea cero, en el momento que la señal *fin mensaje* es uno, significa que todo el mensaje de entrada ha sido leído y almacenado en la memoria, y se da inicio a la operación del sistema. En el estado dos, el sistema pone la señal *go* en ‘1’, iniciando el conteo del tiempo, y se manda un pulso de reinicio al subsistema función *hash sha256* y al subsistema generación *nonce*.

A continuación, se pasa al estado tres donde la señal de activo *nonce* se pone en ‘1’ y se habilita el conteo del contador contB. Una vez realizado esto, se avanza directamente al estado de espera, y posteriormente al estado cuatro, en el cual permanece mientras la señal *nonce\_valid* sea cero. Cuando *nonce\_valid* es ‘1’ avanza al estado cinco. En el caso de que en el estado cinco el conteo de bloques de entrada sea menor a la cantidad mínima 1, se sigue al estado de *fail*, el cual será explicado más adelante.

En el estado cinco se pone la señal *hash* en ‘1’ tomando la posición de memoria equivalente al valor del contador contB-1, y si no hay problemas con la cantidad de bloques se avanza al estado seis, donde el sistema espera hasta que llegue la señal *digest\_valid* diferente de cero*.* Del estado seis se sigue al estado siete, donde se pone la señal *ready* en ‘1’ y si el valor del contador es diferente del número de bloques totales leídos a la entrada, se avanza al estado ocho donde se pone la señal *más bloques* en ‘1’ y se habilita un nuevo conteo del contador auxiliar. Del estado ocho se llega al estado seis, donde se queda en un pequeño ciclo entre los estados seis, siete y ocho hasta que se lea y procese la totalidad de los bloques de entrada.

En el estado siete, cuando el valor del contador contB es igual al número de bloques totales de la entrada, se avanza al estado no leer, en el cual se espera hasta que las posiciones más significativas de la salida de la función *hash* sean almacenadas en el registro de salida. Luego se procede al estado nueve, en el cual se pone nuevamente la señal *activar nonce* en ‘1’ y se habilita el subsistema **comparador ceros** poniendo la señal *ena\_num\_comp\_ceros* en ‘1’. En este punto también se manda una señal de reinicio al contador auxiliar contB. Del estado nueve se sigue al estado espera comparación, y después se sigue al estado diez.

En el estado diez, si la señal *ceros ok* es cero, se indica que no se han hallado coincidencias en el resultado, así que se prosigue a iterar nuevamente, con un nuevo *nonce*. Esto se realiza pasando al estado once en el cual se habilita el conteo inicial del contador auxiliar contB y se sigue al estado espera para realizar nuevamente el proceso descrito, con un nuevo *nonce*.

Para otro de los casos de transición del estado diez, la señal *max tick* del contador secuencial indica que el mismo ha llegado a su límite, y se pasa al estado fail, donde se pone en ‘1’ la señal *failed* junto con la señal *inicio transmisión*, enviando a la salida una cadena de caracteres ‘f’.

El último de los casos de transición del estado diez es en el cual la señal *ceros ok* es uno, lo cual significa que la cantidad de ceros a la salida corresponde con la cantidad de ceros deseada, y el sistema sigue al estado doce, en el que se pone la señal *inicio transmisión* en ‘1’ y se da inicio a la transmisión de la salida en el bloque de comunicación serial. En cuanto al control global, en este punto sigue al estado trece, donde se espera a que la señal fin mensaje sea ‘0’ para proseguir al estado uno nuevamente a la espera deun nuevo mensaje para minar. Tanto desde el estado doce como del estado *fail* se puede llegar al estado final trece.

El esquemático específico de este control se puede encontrar en los anexos, siguiendo el siguiente enlace: [Control Global](https://livejaverianaedu-my.sharepoint.com/:i:/g/personal/fonseca_maria_javeriana_edu_co/Eaoc3kADHghFr3KahP2XRVoBTk1OaFc2k3aoww-7MaIufw?e=7cAmS6). Del mismo modo los esquemáticos de los registros y contadores auxiliares se puede encontrar en los anexos siguiendo el siguiente enlace: [Diagrama de Bloques del control Global](https://livejaverianaedu-my.sharepoint.com/:f:/g/personal/fonseca_maria_javeriana_edu_co/EkiEodk9S5lMtyFtCkCdcBsB0TVBZkSE1s9IJgD63071eA?e=j18Msa).

## **[Sistema de minado en](#_DESARROLLO) *[hardware](#_DESARROLLO)*[, variación](#_DESARROLLO)**

En esta sección se presenta la concepción y diseño de todas las partes del proyecto que conforman la segunda arquitectura para el sistema de minado en *hardware*. En la variación se implementaron cambios sobre la primera arquitectura desarrollada con el fin de comparar el desempeño de los sistemas resultantes. A continuación, se describen los bloques en la capa jerárquica más alta del sistema.

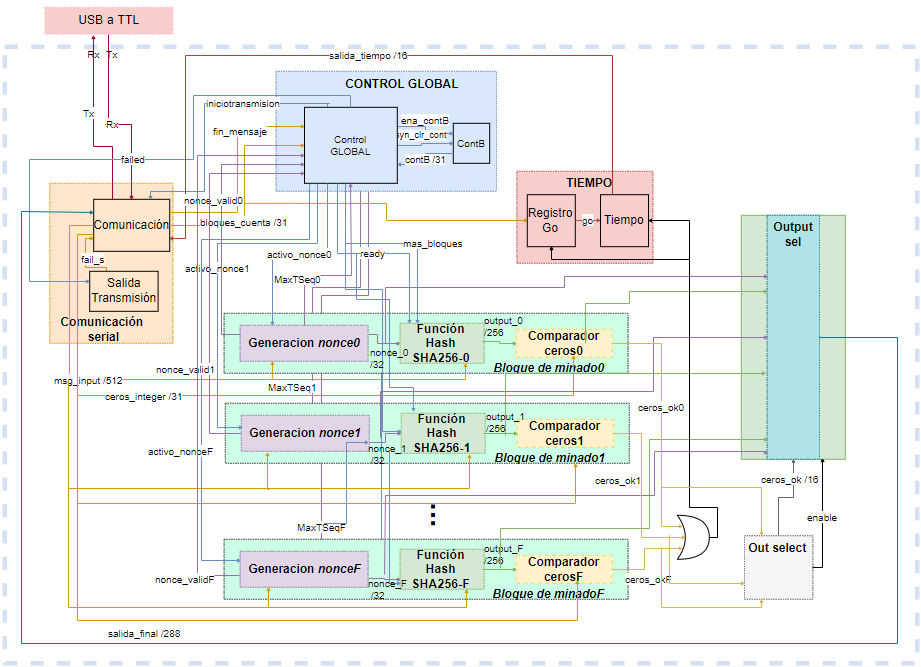


Figura 15. Diagrama de bloques para la variación del hardware.

La variación realizada a la arquitectura original consiste en la implementación de múltiples unidades de minado, es decir que los subsistemas que cumplen la función de minar (compuesto por los subsistemas de generación *nonce*, función *hash* y comparador ceros) fueron replicados varias veces con el fin de encontrar el *nonce* que proporciona el número de ceros requeridos al inicio del *hash,* de una forma más rápida. En total fueron replicadas 16 unidades, desde la unidad de minado 0 hasta la 15, numerando en base hexadecimal del 0 a la F.

Así como la arquitectura inicial, el sistema de minado de *hardware* con variaciones recibe sus entradas y salidas del conversor **TTL a USB**, recibiendo toda la información de entrada del pin de *Rx*, llegando directamente al subsistema **comunicación serial** donde la entrada es almacenada en la memoria *RAM*. De la misma forma inmediatamente el sistema detecta que la totalidad de la información de entrada ha sido leída, inicia el procesamiento del subsistema **tiempo**, e inicia el proceso de minado.

Una parte del mensaje es enviada a todos los subsistemas de **generación nonce** desde el 0 hasta el F*,* donde cada unidad de minado inicia el contador secuencial del *nonce* en un valor distinto y al ser generado un *nonce* diferente por cada una de las unidades, cuando se procesan junto con el mensaje en los subsistemas de **función *hash* *SHA256,*** para cada uno de las unidades de minado se obtiene un valor *hash* diferente. Una vez los resultados son obtenidos, cada unidad pasa al subsistema **comparador de ceros** donde se compara si las posiciones más significativas satisfacen el requerimiento de número de ceros. Las salidas de cada **comparador de ceros** de las unidades de minado (0 - F) llegan a un subsistema llamado **out select**, cuya función es seleccionar la primera salida en obtener el resultado válido en caso de presentarse en alguna de las unidades de minado, la señal *ceros ok*.

En este caso el sistema detiene el conteo de la unidad de tiempo y toma el *nonce* y el resultado de la función *hash* por medio del subsistema **Output sel**, el cual, con base en la información proporcionada por el subsistema **out select**, toma las salidas correctas junto con el tiempo resultante. Estas variables se transmiten al computador desde el subsistema de **comunicación serial** al computador, por medio del conversor **TTL a USB**.

En caso de que no satisfacer la condición del número de ceros, el sistema genera un nuevo *nonce* en cada unidad de minado y se procesa de la misma forma anteriormente explicada, iterando hasta encontrar una coincidencia. De igual manera que la arquitectura inicial, en el caso de no hallar una coincidencia o que la información de entrada no constituya el mínimo de 1 bloque, la prueba es procesada como fallida y el sistema envía al computador una cadena de ‘f’ para indicar el estado de la prueba.

Todas las interacciones entre subsistemas y el manejo de la sincronización entre los mismos es orquestada por el **control Global.** A continuación, se explicará el diseño e implementación de los subsistemas que tuvieron variaciones (**control global**, **generación *nonce***), y los nuevos subsistemas (**out select**, **output sel**).

Los esquemáticos para cada vista jerárquica de este sistema se pueden encontrar en los anexos, ingresando en el siguiente enlace: [RTL jerárquico de](https://livejaverianaedu-my.sharepoint.com/:b:/g/personal/fonseca_maria_javeriana_edu_co/EWbK4kJGlGdOhjKdv4ug5XQBX3M67im4RtCAiABhWnGfew?e=wiXY8T) la variación de la arquitectura.

### **[Planeamiento y variaciones para la implementación de las unidades de minado](#_Módulo_de_minado_1)**

En la presente sección se describen los lineamientos planteados para la implementación de las unidades de minado paralelas en el sistema de variación de la arquitectura del sistema de minado en *hardware*.

Para la implementación de varias unidades de minado, se llegó a la conclusión de que solo era necesario realizar cambios sobre el subsistema de generación *nonce*, de forma que en los subsistemas de función *hash* y comparador ceros, solo sería necesario replicarlos para cada unidad de minado.

El cambio principal que se realizó a los subsistemas de generación *nonce,* fue a sus configuraciones más no a su arquitectura, de forma que todos los módulos generación *nonce* comenzaran su contador secuencial en un punto diferente, con el fin de realizar el barrido de *nonce* de forma más rápida.

Con el fin de llevar sincronía entre los contadores y alterar en un mínimo posible el funcionamiento del sistema anterior, se planteó que todos los subsistemas de *nonce* debían tener su sección pseudo-aleatoria igual, de forma que la diferencia radicara únicamente en la sección secuencial, y del mismo modo, que todas las secciones secuenciales estuvieran sincronizadas de tal forma que alcanzaran su máximo valor simultáneamente.

Estos dos planteamientos fueron posibles de implementar al definir que con un número *m* de unidades de minado, cada contador debía contar un total de 2k/m veces, siendo *k* el número de bits del contador. En este orden de ideas, el número de unidades de minado a implementar debería ser un múltiplo de 2n. En base al espacio de almacenamiento ocupado por la primera arquitectura se inició implementando 8 unidades de minado siguiendo por 16 unidades de minado, donde se determinó una utilización razonable de los elementos lógicos de la tarjeta, los cuales serán expuestos en la sección de [resultados](#_ANÁLISIS_DE_RESULTADOS).

Con el fin de conservar la homogeneidad y sincronización del sistema, a todos los subsistemas de generación *nonce* les llega como entrada la misma semilla inicial, y considerando que las posiciones menos significativas de todos los contadores secuenciales estarían en todo momento en sincronía, se aseguró que la sección pseudo-aleatoria del nonce fuera la misma para todas las unidades en cada iteración.

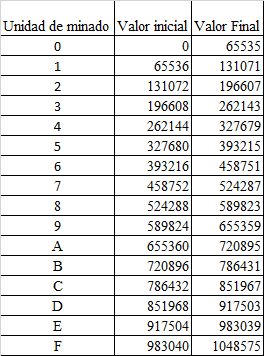


Figura 16. Configuraciones de los contadores secuenciales para unidades paralelas de minado.

Las configuraciones iniciales para cada contador secuencial de cada subsistema de generación *nonce* de sus respectivas unidades de minado se especifica en la Figura 16. Esta configuración es para 16 unidades de minado y 20 bits de contador secuencial.

Los códigos se pueden encontrar en la carpeta del proyecto de Quartus Prime en los anexos ingresando en el siguiente enlace: [Códigos en VHDL y Verilog VAR](https://livejaverianaedu-my.sharepoint.com/:f:/g/personal/fonseca_maria_javeriana_edu_co/EjkivjxQjb9Bp1q4cA9NYcEB92K4fFDn6-LjDdWOxra9-w?e=GXJycH).

### **[Diseño e implementación del subsistema ‘out select’](#_Módulo_de_minado_1)**

En la siguiente sección se describe el fundamento y funcionamiento del subsistema encargado de detectar la ‘posición’ de una salida en el conjunto de unidades paralelas de minado, en la llamada arquitectura variación.

Al plantear el uso de unidades en paralelo de minado, surgió un problema principal, el cual sería el de determinar la salida que corresponde a su señal *ceros ok*, ya que el control global solo detecta si alguna de todas ellas ha sido uno, más no exactamente cuál.

Al tener repartidos en cada unidad de minado un par de registros conteniendo la salida de la función *hash* con su respectivo *nonce*. Se decidió tomar estos registros separados como una memoria y realizar un subsistema capaz de determinar la posición de las salidas en este arreglo de registros, tomando las señales de ceros ok en cascada del registro de la unidad 0 a la f. En este orden de ideas, en el caso de obtener dos o más salidas con un par valor *hash - nonce* válido para la cantidad de ceros deseada, sólo se tomaría la posición de la primera salida válida hallada, y esta misma sería indicada al subsistema **output select** para su posterior procesamiento en el subsistema **comunicación**.

El esquemático específico de este subsistema se puede encontrar en los anexos ingresando en el siguiente enlace: [Diagrama de bloques out select](https://livejaverianaedu-my.sharepoint.com/:b:/g/personal/fonseca_maria_javeriana_edu_co/EWwI41UaAJFFv8sm5xDhUGcBO7DEy-jDDqyu9qFDZlbUWg?e=tsCyXZ).

### **[Diseño e implementación del subsistema ‘output sel’](#_Módulo_de_minado_1)**

En la presente sección se describe el funcionamiento del subsistema que se encarga de seleccionar la salida correcta del sistema de unidades de minado, para enviar a la transmisión serial

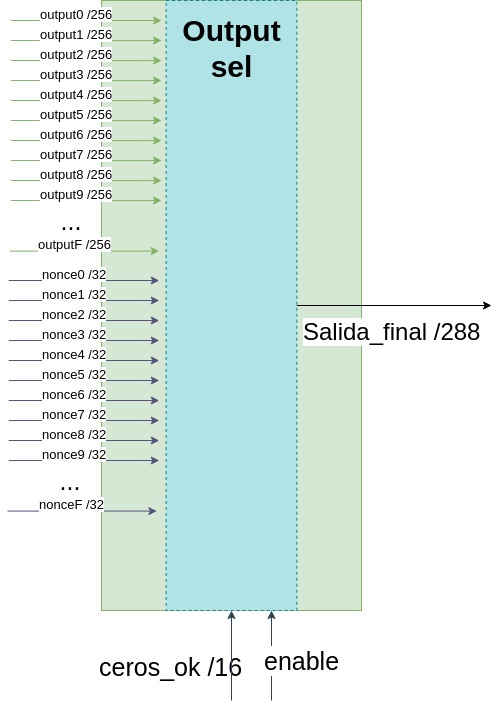


Figura 17. Diagrama de bloques del subsistema Output sel.

Para el registro de salida **output sel**, se tienen como entradas todas las salidas de las unidades de las funciones *hash* de cada subsistema de minado, es decir output 0 hasta output F, Las salidas del *nonce* de todas las unidades de minado, el registro *ceros\_ok* proveniente del subsistema **out select** y el habilitador. Se tiene como salida la información que va hacía el subsistema **comunicación serial** para realizar su transmisión.

El funcionamiento del subsistema es el de un registro al cual le entran todas las salidas de *hash* y *nonce,* y se alnacenan las que corresponden a la señal *ceros\_ok* proveniente del subsistema **out select**. Esto con el fin de seleccionar la salida correcta que se va a transmitir, es decir, la que corresponde con el número de ceros requeridos, con el valor *hash* y *nonce* correctos.

El esquemático específico de este subsistema se puede encontrar en los anexos ingresando en el siguiente enlace: [Diagrama de bloques output sel](https://livejaverianaedu-my.sharepoint.com/:i:/g/personal/fonseca_maria_javeriana_edu_co/Ea6jDtANGadCmTi-_OlBFEIBHy_D6X1HEtRGDeJ-8SoWSw?e=31CKUt).

### **[Diseño e implementación del subsistema ‘control global’ para la variación](#_Módulo_de_minado_1)**

En la presente sección se describe el funcionamiento del Control Global para la variación de la arquitectura original, el cual funciona de manera similar al planteado para la arquitectura inicial, utilizando un contador externo como apoyo para el procesamiento de cadenas de bloques en la entrada. Ya que este subsistema es meramente compuesto por el control y el contador de apoyo, se abordará únicamente el funcionamiento de la máquina de estados finitos.

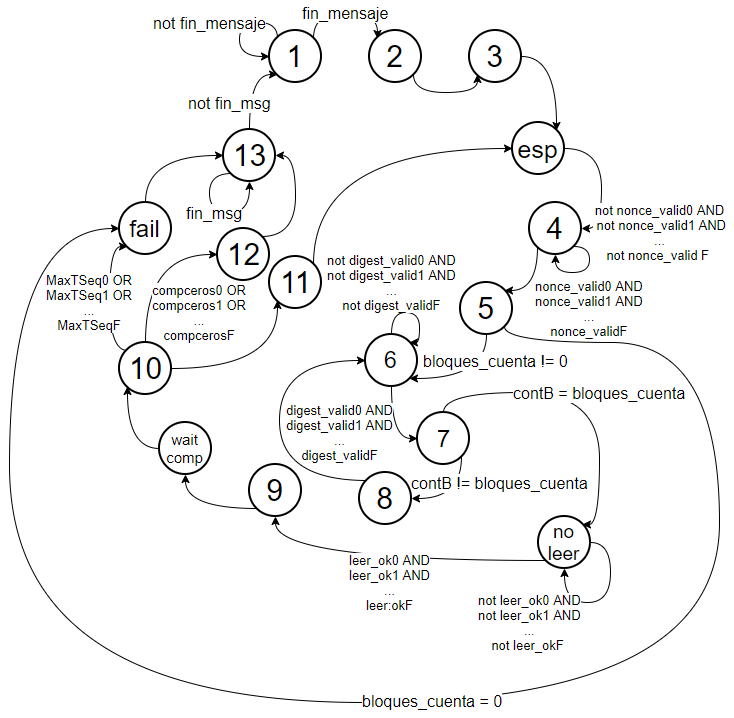
****

Figura 18. Máquina de estados finitos del Control Global para la variación.

Para la máquina de estados de la Figura 18, el funcionamiento de la misma es bastante similar al anteriormente descrito, con la diferencia de que este maneja en simultáneo todas las unidades de minado. Comienza en el estado uno donde se manda la señal de reinicio al contador de apoyo contB, el sistema permanece en este estado mientras la señal fin mensaje sea cero, en el momento que la señal fin mensaje es uno se da inicio a la operación del sistema. En el estado dos el sistema pone la señal go en ‘1’, iniciando el conteo del tiempo, y se manda un pulso de reinicio a todas las unidades de minado del 0 - F, compuestas por las unidades de función *hash sha256*, generación *nonce* y comparar ceros.

Se pasa al estado tres donde la señal de activo *nonce* se pone en ‘1’ para todas las unidades de minado, y se habilita el conteo del contador contB. Una vez realizado esto se avanza directamente al estado de espera, y posteriormente al estado cuatro, en el cual permanece mientras la señal *nonce\_valid* de todas las unidades de minado sea cero. Cuando todas estas señales son ‘1’, se avanza al estado cinco. En el caso de que en el estado cinco el conteo de bloques de entrada sea menor a la cantidad mínima 1, se sigue al estado de *fail*.

En el estado cinco se pone la señal *hash* en ‘1’ para todas las unidades de minado, tomando la posición de memoria contB-1, y si no hay problemas con la cantidad de bloques se avanza al estado seis, donde el sistema espera hasta que la señal *digest\_valid* de todos los bloques de minado sea diferente de cero*.* Del estado seis se sigue al estado siete, donde se pone la señal *ready* en ‘1’, y si el valor del contador es diferente del número de bloques totales leídos a la entrada, se avanza al estado ocho donde se pone la señal de más bloques en ‘1’ y se habilita un nuevo conteo del contador auxiliar. Del estado ocho se regresa al estado seis donde se queda en un pequeño ciclo entre los estados seis, siete y ocho hasta que se lea y procese la totalidad de los bloques de entrada.

En el estado siete, cuando el valor del contador contB es igual al número de bloques totales de la entrada, se avanza al estado no leer, en el cual se espera hasta que las posiciones más significativas de la salida de cada uno de los bloques de función *hash* sean almacenados en sus respectivos registros de salida. Luego se procede al estado nueve, en el cual se pone nuevamente la señal de activar nonce en ‘1’ para todas las unidades de minado, y se habilitan los subsistemas comparador ceros poniendo la señal *ena\_num\_comp\_ceros* en ‘1’. En este punto también se manda una señal de reinicio al contador auxiliar contB. Del estado nueve se sigue al estado de espera comparación y después se sigue al estado diez.

En el estado diez, si todas las señales ceros ok de los sistemas de minado son cero, se indica que no se han hallado coincidencias en ningún resultado, así que se prosigue a iterar nuevamente, con un nuevo *nonce* en cada unidad de minado. Esto se realiza pasando al estado once en el cual se habilita el conteo inicial del contador auxiliar contB y se sigue al estado espera para realizar nuevamente el proceso descrito.

En otro de los casos de transición del estado diez, la señal de max tick de los contadores secuenciales indica que se ha llegado a su límite, y se pasa al estado de fail, donde se pone en ‘1’ la señal *failed*, junto con la señal *inicio transmisión*, enviando a la salida una cadena de caracteres ‘f’.

En el último de los casos de transición del estado diez, si alguna de las señales ceros ok provenientes de los bloques de minado es uno, el sistema sigue al estado doce, en el que se pone la señal inicio transmisión en ‘1’ y se da inicio a la transmisión de la salida en el bloque de comunicación serial. En cuanto al control global, en este punto sigue al estado trece, donde se espera a que la señal fin mensaje sea ‘0’ para proseguir al estado uno nuevamente a la espera un nuevo mensaje para minar. Tanto del estado doce como del estado fail se puede llegar al estado final trece.

Los esquemáticos y diagramas para la arquitectura de la variación se puede encontrar en los anexos ingresando en el siguiente enlace: [Diagramas y el proyecto para ASIC mining](https://livejaverianaedu-my.sharepoint.com/:f:/g/personal/fonseca_maria_javeriana_edu_co/Eiwugo4NxmhCn9atnVKks2gB3H2P3xCnoaKL0k82pXE08Q?e=CWJCJM).

## **[Desarrollo en software](#_DESARROLLO)**

En esta sección se comprende la concepción e implementación de un código de seguimiento funcional en *software,* con el fin de verificar el funcionamiento del diseño del sistema base en *hardware*. De igual forma, se ilustra el proceso de adaptación de entradas para cada sistema en conjunto con el manejo de la transmisión serial de cara al *software*. A continuación, se describe la lógica implementada en alto nivel en conjunto con las principales funciones.

Para corroborar la funcionalidad en *hardware* se realizó el diseño de la arquitectura planteada, en *software,* tomando como base de referencia un *script* descrito en *python* con la funcionalidad de realizar el cálculo de *hash sha256* y *sha224*. Se requirió realizar la adecuación del código base, pues este por defecto realizaba la función *hash* con mensajes de entrada fijos codificados en hexadecimal. Cada mensaje de entrada se ingresaba como un arreglo de 16 posiciones, donde cada posición está compuesta por un número de 8 dígitos hexadecimales. Esto conlleva al ingreso de 32 bits por campo en 16 posiciones, es decir, 512 bits de igual manera que en el sistema en *hardware*.

La primera adecuación que se realizó fue ingresar el mensaje de entrada desde un archivo externo con *padding* de 64\*n caracteres (64, 128, 192, etc) con un mínimo de 64. Para ello se realizó la lectura del archivo externo y se almacenó en una variable como una cadena de caracteres (*string*). Se realizó el barrido de la cadena, se codificó a entero en base hexadecimal y se separó en grupos de 8 dígitos donde cada 2 dígitos *HEX* son la representación de un carácter del mensaje original en código *ASCII*. De esta forma, se buscó obtener el *hash* de un mensaje parametrizable por medio de un archivo externo.

Posterior al proceso de adecuación de entrada de la funcionalidad de *hash*, se procedió a la implementación de la función para realizar la generación de un valor *nonce* pseudo-aleatorio. La generación del *nonce* (compuesto por 16 o 32 bits - 2 o 4 dígitos en *ASCII*) surge a partir del mensaje que se ingresa. Se realizó la codificación del mensaje de entrada a binario y se obtuvieron los seis primeros y últimos bits. Estos se concatenaron para obtener un valor de semilla inicial de doce bits. El procedimiento de generación consiste en elevar una semilla al cuadrado, obtener los bits intermedios y concatenarlos con un contador para tener una parte secuencial. A partir de los bits intermedios se genera la nueva semilla que será elevada al cuadrado. De igual forma que en hardware se tiene una sección de verificación para volver a generar una semilla a partir del contador secuencial en caso de que la semilla tienda a converger a 0 o caer en un ciclo repetitivo. En *software* el resultado del *nonce* es concatenado con el mensaje de entrada.

Para realizar el cálculo del *hash*, en la entrada se requiere un mensaje proporcional a 64\*n caracteres *ASCII*. Al concatenar el mensaje de entrada con el *nonce*, se adicionan entre dos o cuatro caracteres dependiendo del tamaño del *nonce*. Las posiciones adicionales que se retiran para cumplir con el número requerido de caracteres son las dos o cuatro posiciones anteriores a los últimos 3 caracteres, debido a que estos últimos 3 caracteres hacen referencia el número de bits que contiene el mensaje inicial antes de realizar *padding*, a este número fue necesario adicionar el número de bits que componen el *nonce*. La función que realiza este procedimiento se llama *nonce* y no requiere entradas por parámetro, no obstante, su funcionamiento depende de variables globales como la semilla inicial que se va modificando y el resultado del *nonce.*

El objetivo de la implementación en software es corroborar los resultados del *hardware*, para ello se requiere ingresar la misma entrada con un *padding* previo en los dos sistemas de minado. De igual forma, el ingreso manual del mensaje a la *FPGA* está limitado por el número de pines de entrada. Por tanto se realizó la implementación de la adaptación del mensaje por medio de *software*, y por medio del mismo ingresar la misma información a los dos sistemas.

//(Agregar el nombre de la función y los parámetros de entrada tanto para HW como para SW)

Al sistema tanto en *software* como en *hardware* se requiere ingresar una entrada estándar con un número de caracteres múltiplo de 64, es decir bloque de 512 bits. Para ello, si la entrada contiene menor número de caracteres se requiere rellenar el mensaje hasta obtener 64. Este proceso se realiza en *software* por medio de una función llamada *padding,* a la que le ingresa el nombre de un documento de texto como parámetro de entrada de la función y se realiza una copia con el mismo nombre del archivo con un .*padded* adicional como se muestra a continuación: *‘archivo.txt’+.padded*.

Una vez creado el archivo se lee línea por línea y se genera el carácter euro (€) para determinar el fin del mensaje al final de la línea, se realiza el cálculo de la cantidad de ceros (carácter *null*) con las cuales se debe rellenar el mensaje y se calcula su tamaño. Para cálculo de número de ceros, se halló el tamaño de la cadena de mensaje de la línea y se realizó la operación que se muestra en la Ecuación 1.

Ecuación 1

A partir del resultado se generaron arreglos de *bytes* de ceros para ser almacenados antes de la transmisión a la *FPGA*. Posteriormente a partir del tamaño la cadena de línea multiplicado por 8 se halló el número de *bits* que componen el mensaje, y se utilizó del módulo *structure* específicamente la función *pack* para representarlo como un binario *unsigned* del tipo *big-endian*. El resultado se almacenó en una variable global a la cual se le adicionaron 32 bit más de *nonce*.

Los resultados de la variable de ceros, la representación del carácter euro en hexadecimal y la del tamaño del mensaje se concatenaron y se almacenaron para ser escritos en el documento con la función *f.write()* donde “*f”* es el documento en el cual se guarda la información. El orden de concatenación es el siguiente: mensaje + carácter euro + relleno de ceros (*null*) + tamaño del mensaje. La función *padding* retorna el nombre del nuevo documento para así ingresarlo a las funciones de adecuación de entrada del *software* y *hardware*.

Para el funcionamiento del *hardware* se requirió una entrada de 66 caracteres compuesta por el parámetro de número de ceros en sus dos primeros caracteres y 4 caracteres (32 bits) de *nonce* concatenados con el mensaje original, en conjunto con los caracteres adjuntados por el *padding* para obtener 64. De igual forma, la entrada del *hardware* debe contener en el último carácter el tamaño del mensaje en bits en conjunto con los 32 bits del *nonce*.

El número de ceros ingresa al *hardware* de forma invertida, por ejemplo, si el número de ceros es 12 bits de cero, los caracteres ingresan primero el 2 y luego el 1,+(64 caracteres adicionales). En este sentido, se llevó a cabo la adaptación de este parámetro en función de una variable la cual se encarga del manejo de ceros en el programa, en hexadecimal (1, 2, 3 o 4), y para ello se realizó la equivalencia para tomar el número en binario multiplicando la variable por 4 y se invirtieron los dígitos de los resultados para concatenarlo con los 4 caracteres *ASCII* de *nonce* y el resto del mensaje de entrada.

Como el sistema en *hardware* es el que genera internamente los valores de *nonce* en cada iteración, a la entrada se ingresa como valor inicial de este parámetro los 4 caracteres en cero (‘0000’). La entrada se compone de la siguiente manera: 2 caracteres de ceros + 4 caracteres de *nonce* + 59 caracteres de mensaje con relleno + 1 carácter con el tamaño del mensaje. Esta cadena de caracteres se recorrió posición por posición para convertir cada carácter de *ASCII* a entero y posteriormente a arreglo de bytes para enviarlo a través del puerto serial.

Para enviar el mensaje de entrada a la *FPGA* y recibir el resultado del valor *hash*, se importó la librería *serial* y se utilizaron los métodos .*read()* para leer el *hash* y .*write()* para transmitir la entrada. Así mismo, se importaron y adecuaron las funciones get\_ports() y findArduino() descargadas de *GitHub*. La primera función get\_ports() se utilizó para obtener los puertos seriales que están siendo utilizados, y la segunda función findArduino() para hallar en qué puerto se encuentra el dispositivo usado para transmitir.

La función inicialmente estaba configurada para detectar un Arduino, sin embargo en el proyecto el objetivo de esta función fue hallar el conversor USB a TTL, para ello se le modificó el nombre del dispositivo por: 'Silicon Labs CP210x USB to UART Bridge'. El objetivo de estas funciones era obtener el puerto del dispositivo y así poder configurar los parámetros de la transmisión. La configuración del puerto se enuncia a continuación: nombre del puerto, velocidad de transmisión, tamaño del byte y bit de parada.

Para la adecuación de la entrada en *software*, se utilizó como referencia las entradas fijas de prueba con las cuales estaba implementada en la *IP* original de *hash* como se muestra a continuación. El bloque TC1 representa el mensaje “abc” para un mensaje de entrada menor a 512 bits.

TC1\_block = [0x61626380, 0x00000000, 0x00000000, 0x00000000,

0x00000000, 0x00000000, 0x00000000, 0x00000000,

0x00000000, 0x00000000, 0x00000000, 0x00000000,

0x00000000, 0x00000000, 0x00000000, 0x00000018]

Figura 19. Entrada original TC1 IP de hash.

La adecuación consiste principalmente en leer el mensaje de entrada, concatenar el valor de *nonce* generado por el *software* al inicio del mensaje, y almacenar el valor correspondiente en entero hexadecimal a cada carácter *ASCII* en un arreglo. Estos valores se pasaron a cadenas de caracteres para así concatenarlos y agruparlos en grupos de 8 dígitos hexadecimales en los cuales cada 2 representa un carácter *ASCII* como se muestra en la Figura 19. De igual forma que el *hardware*, al adjuntar el *nonce* al mensaje se requirió adicionar los 32 bits a los caracteres finales que indican el tamaño del mensaje.

A continuación, en la Figura 20, se ilustra la secuencia de funcionamiento del script en *software*:

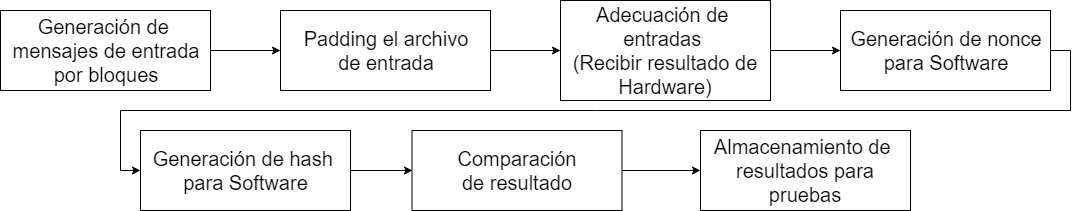


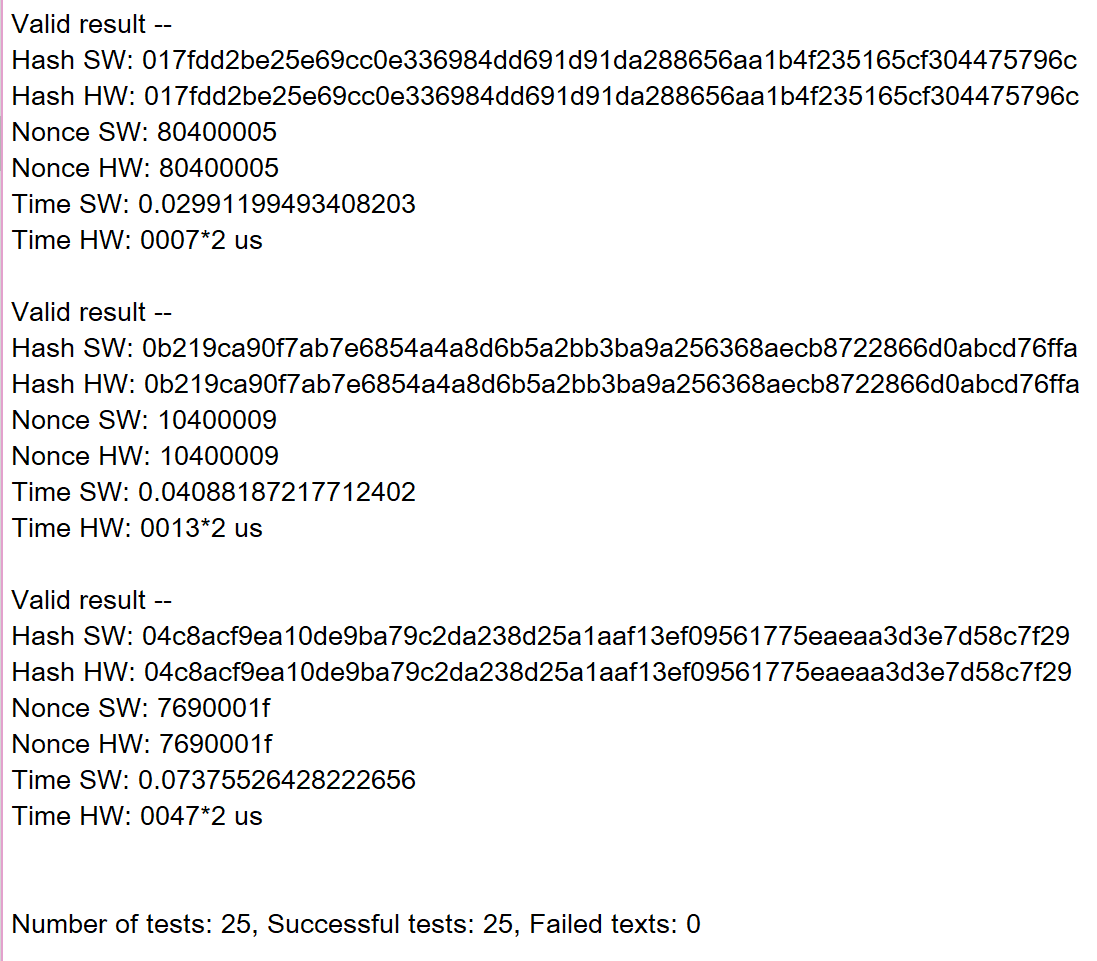
Figura 20. Secuencia de funcionamiento software.

# **[PROTOCOLO DE PRUEBAS](#_ÍNDICE)**

En el presente capítulo se presenta la descripción del protocolo de pruebas planteado para obtener las evidencias del cumplimiento tanto del objetivo del proyecto, como del desempeño funcional del proyecto desarrollado, con los respectivos procedimientos realizados y su justificación.

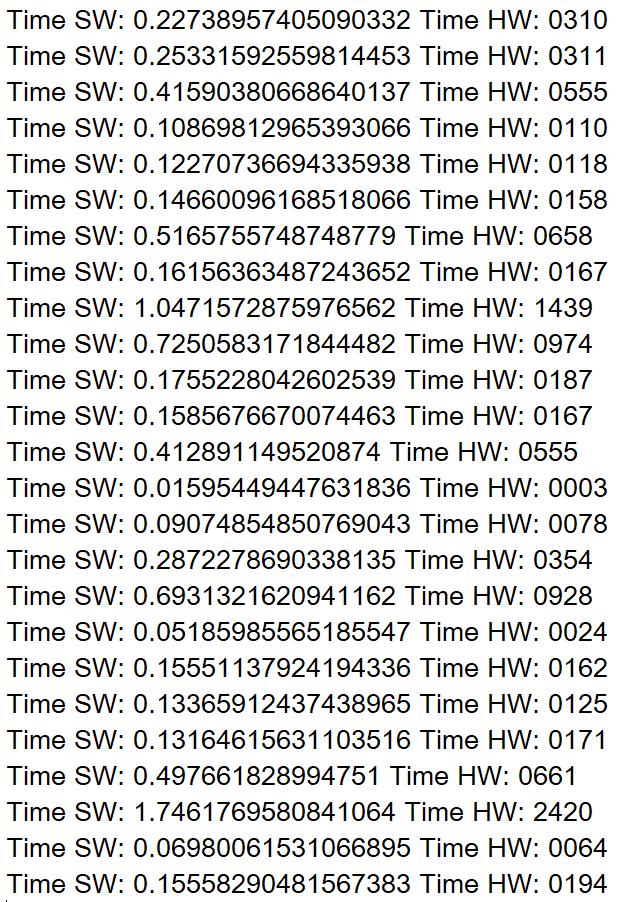
Se planteó un entorno de pruebas basado en *software* para realizar una comparación funcional, automatizar el proceso y realizar una recopilación de datos para su posterior análisis. El objetivo del trabajo de grado consistió principalmente en realizar un sistema en *hardware,* el cual ejecutara y encontrara un valor *hash,* para un determinado número de ceros al inicio del resultado a partir de la variación de un *nonce*. No obstante, obtener el resultado sin referencia alguna no garantiza la validez del resultado. Por ello se realizó la implementación de un código en *software* basado en la IP de *python* que se encuentra adjunta a la IP de *hardware* obtenida de *Github*.

Para la verificación del funcionamiento del *hardware*, se planteó recibir el resultado de la *FPGA* de forma serial por medio de la librería *pyserial*, y con ello comparar el resultado recibido por el puerto serial con el resultado generado en el sistema de minado del *software*, para así obtener un chequeo automático. De igual forma se estipuló almacenar los dos resultados en arreglos para recopilar todos los casos de prueba en un documento y analizar la salida de los sistemas. Para la arquitectura base de *hardware* y el *software*, las variables que se almacenan principalmente en los documentos de salida fueron el resultado del *hash*, el tiempo tanto en *hardware* como en *software*, el valor de nonce y si la prueba fue válida o no (*software* igual *hardware*) como se muestra a continuación en la figura Y.



*Figura Y. Archivo de salida para pruebas de funcionamiento para hardware original.*

Adicionalmente se verificó cuantas pruebas se estaban realizando, el número de pruebas fallidas y el número de pruebas exitosas para corroborar el funcionamiento; si existía una prueba fallida se procedía a revisar los sistemas de *software*-*hardware* para verificar y arreglar la lógica errónea para un resultado diferente. Asimismo se generó un documento adicional con el tiempo de cada prueba para realizar un análisis estadístico posterior, como se muestra en la figura W. Con respecto a la variación de la arquitectura original se tomaron los datos de tiempo para comparar entre la arquitectura original y su variación. El resultado de *hardware* y *software* se fue obteniendo y almacenando por mensaje.



*Figura Y. Pequeña fracción de archivo de salida de tiempo para hardware original.*

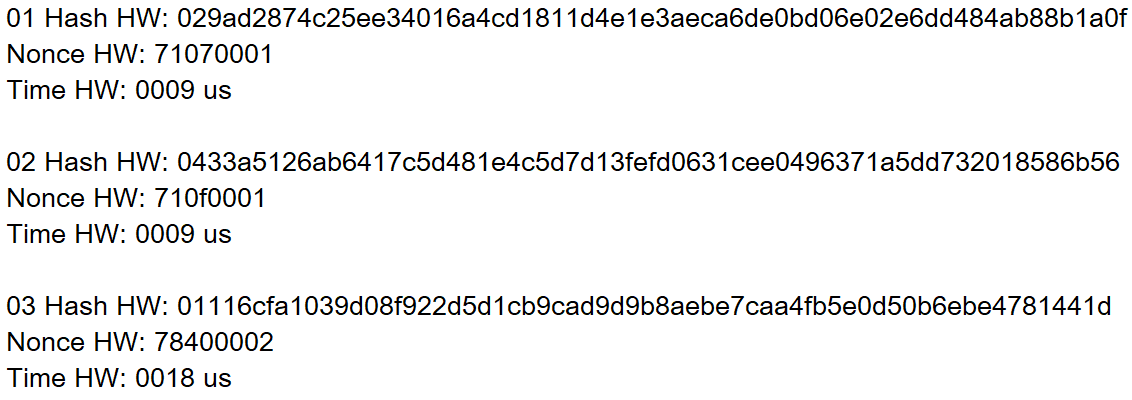
Para la automatización del proceso de pruebas se decidió implementar una generación de entradas aleatorias con diferentes tamaños. Al variar los tamaños de la entrada se buscó como objetivo verificar que el sistema proporcionara un resultado valido para todas las posibilidades de bloques. Para ingresar la entrada se implementó la generación de cadenas de caracteres aleatorios de tamaños múltiplos a 55 caracteres con la librería random, y se almacenaron en un document. Cada documento contiene 25 líneas, es decir 25 mensajes de entrada cada mensaje se divide por medio de un fin de línea como se muestra en la figura Ñ. El nombre del documento fue proporcionado por parámetro a la función que realiza el *padding* para ejecutar todo el proceso de pruebas en un solo comando.



*Figura Ñ. Ejemplo archivos generados por el entorno de pruebas como entrada (mensajes) para 1 y dos 2 bloques.*

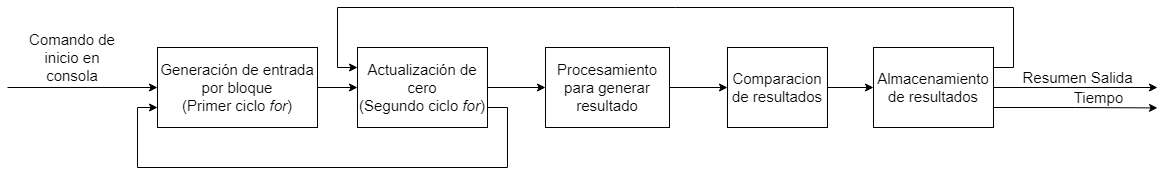
Para abarcar todas las posibilidades de prueba, el código base se implementó dentro de dos ciclos *for,* el ciclo externo para el manejo del tamaño de los bloque y un ciclo interno para el manejo del número de ceros por bloque. El ciclo del número de ceros inicia en 1 cero y finaliza con 4 ceros. Para el número de bloques de entrada se propuso inicialmente de 0 a 16 bloques. Sin embargo en el proceso de pruebas se evidenció diferencia en la salida del hash para una entrada de más de 8 bloques. Así pues, al evidenciar entre el resultado de hash entre software y hardware se verificó las posibles variaciones en el módulo para generar el valor de nonce, no obstante no se encontró diferencia alguna. Posterior a verificar el *nonce* se decidió confirmar el funcionamiento de las máquinas de estados y contadores pues la diferencia solo se visualizaba después de 8 bloques, a pesar de ello los bloques analizados funcionaban según lo esperado, por lo tanto se verificó la sección de generación del hash. Se visualizaron los resultados de hash para cada iteración por bloque, es decir si la entrada tenía 10 bloques se mostraban 10 resultados de hash donde el último era el válido para todo el mensaje si el mensaje. Al correr esta prueba (más de 8 bloques visualizando la salida para cada iteración por bloque) se evidencio que para todos los bloques a excepción del último, basado en ese planteamiento se decidió observar la variable que va actualizando su valor por cada hash dentro de la IP de python y se evidenció que este era el resultado que sal**Í**a del sistema de hash original. Al realizar el análisis descrito anteriormente se decidió contactar a por medio de Github al autor original de la IP para informar de la diferencia presente en la generación del *hash* entre las IP de *software* y *hardware* para mensajes de entrada superiores a 4096 bits.

Con respecto al entorno de pruebas del sistema en hardware basado en la paralelización de la arquitectura original, se decidió realizar cambios en la metodología de pruebas para **EL** análisis propuest**O**, pues realizar la comparación entre resultado de software y este sistema en hardware no tiene sentido debido a la forma en que se obtiene el resultado (distinto resultado). Siendo la implementación en hardware más eficiente al no realizar el proceso de manera secuencial como el código en software, se decidió descartar la opción de realizar la comparación entre resultados y la toma de muestras del software al mismo tiempo con el *hardware*, a diferencia de la arquitectura original. Para este sistema se generaron dos documentos, el primero solo con los parámetros obtenidos del hardware como el resultado, el tiempo y el nonce, como se muestra en la figura Q. Adicionalmente se produjo un documento con el tiempo individual para su posterior análisis con el tiempo de la arquitectura original. //Poner la figura para el tiempo cuando esté corregido



*Figura Y. Pequeña fracción de archivo de salida de para la variación hardware.*

En consecuencia, como protocolo de pruebas final con el fin de analizar estadísticamente los resultados, se estipularon 640 pruebas: 20 mensajes a 1, 2, 3, y 4 ceros con bloques de tamaño de entrada entre 1 y 8, donde cada mensaje es una cadena de caracteres alfanuméricos basados en los caracteres imprimibles de *ASCII*. Los resultados de salida se almacenaron en dos documentos principales, uno para almacenar el tiempo de cada solución de *hash* y otro para un resumen del resultado de los parámetros con los cuales se obtuvo el resultado como hash, nonce y tiempo para *hardware* y *software*. En síntesis, la idea fue automatizar el proceso de pruebas por medio de la herramienta *python* en *software*.



*Figura M. Flujo de funcionamiento de pruebas.*

//Adicionar la parte de la transacción

# **[ANÁLISIS DE RESULTADOS](#_ÍNDICE)**

En este capítulo debe realizarse el análisis y la interpretación de los resultados obtenidos al seguir el protocolo de pruebas desarrollado en el punto anterior; la interpretación de los datos constituye el soporte de las conclusiones.

En el presente capítulo se presentan los resultados obtenidos del análisis e interpretación de los resultados obtenidos al seguir el protocolo de pruebas desarrollado en el capítulo anterior, explicando en cada instancia la información presentada a continuación.

# **[CONCLUSIONES Y RECOMENDACIONES](#_ÍNDICE)**

En este capítulo se presentan los puntos concluyentes, producto del análisis e interpretación de los datos pertenecientes a los resultados presentados en la sección anterior. De igual manera se incluyen los hallazgos encontrados durante la realización del trabajo de grado, junto con las recomendaciones del mismo.

Relacionar los puntos concluyentes producto del análisis y la interpretación de resultados; así mismo, deben escribirse los hallazgos encontrados al realizar el trabajo de grado. En este contexto se entienden los hallazgos como descubrimientos inesperados durante el desarrollo del proyecto. Las recomendaciones son de suma importancia en todo el trabajo, pues es el aporte que realizan los autores al avance del conocimiento. Tenga en cuenta que las conclusiones no son una lista de los resultados finales.

# **[BIBLIOGRAFÍA](#_ÍNDICE)**

Se deben relacionar todos los libros, revistas, artículos, documentos y diversas fuentes de información consultadas.

BIBLIOGRAFÍA ADRIANA

**<https://www.youtube.com/watch?v=D4_SQawLwQs>**

BIBLIOGRAFÍA PAOLA

[1]<https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ism_p_instantiating_verilog_module_mixedlang.htm>

[2] Secure Programming Cookbook for C and C++by Matt Messier; John ViegaPublished by O'Reilly Media, Inc., 2003

<https://learning.oreilly.com/library/view/secure-programming-cookbook/0596003943/ch11s01.html>

BIBLIOGRAFÍA ANTEPROYECTO

[1] "Blockchains: The great chain of being sure about things". The Economist. 31 October 2015. [Accessed 19 Jul. 2018].

[2] A. Bartolomé Pina, C. Bellver Torlà, L. Castañeda Quintero and J. Adell Segura, "BLOCKCHAIN EN

EDUCACIÓN: INTRODUCCIÓN Y CRÍTICA AL ESTADO DE LA CUESTIÓN", EDUCTEC. Revista Digital de Tecnología Educativa, vol. 61, no. 1135-9250, pp. 1-14, 2017.

[3] "La generación de bitcóin utiliza más energía que un país entero", Dinero, 2018. [Online]. Available at:

https://cioperu.pe/articulo/25261/los-5-principales-problemas-de-blockchain/ [Accessed: 26- Jul- 2018].

[4] Wang L, Liu Y. Exploring Miner Evolution in Bitcoin Network. In: Mirkovic J, Liu Y, editors. Passive and

Active Measurement. vol. 8995 of Lecture Notes in Computer Science. Springer International Publishing; (2018). p. 290–302. [online] Available at: http://dx.doi.org/10.1007/978-3-319-15509-8\_22 [Accessed 19 Jul. 2018].

[5] Blockchain.com. (2018). Gráficos de Bitcoin - Blockchain.info. [online] Available at:

https://www.blockchain.com/es/charts [Accessed 19 Jul. 2018].

[6] REYES MACEDO, Víctor Gabriel, SALINAS ROSALES, Moisés y GALLEGOS GARCÍA, Gina "Bitcoin: Una visión general ". Revista Digital Universitaria, (2017), Vol. 18, Núm. 1. [online] Available at: http://www.revista.unam.mx/vol.18/num1/art11/index.html ISSN: 1607-6079 [Accessed 19 Jul. 2018].

[7] Yli-Huumo J, Ko D, Choi S, Park S, Smolander K (2016) Where Is Current Research on Blockchain Technology?—A Systematic Review. PLoS ONE 11(10): e0163477. [online] Available at: https://doi.org/10.1371/journal.pone.016347 6079 [Accessed 19 Jul. 2018].

[8] Digiconomist. (2018). Bitcoin Energy Consumption Index - Digiconomist. [online] Available at:

https://digiconomist.net/bitcoin-energy-consumption [Accessed 27 Jul. 2018].

[9] Blockchain.com. (2018). Distribución de tasas de hash. [online] Available at:

https://www.blockchain.com/es/pools [Accessed 27 Jul. 2018].

[10] Blockchain.com. (2018). Gráficos de Bitcoin - Blockchain.info. [online] Available at:

https://www.blockchain.com/es/charts [Accessed 26 Jul. 2018].

[11] Bitcoin WARNING: Mining for cryptocurrency could 'DESTROY' your PC, experts warn(2018). EXPRESS Home of the dialy and Sunday express. [online] Available at:

https://www.express.co.uk/life-style/science-echnology/932903/Bitcoin-warning-cryptocurrency-destroy-PC-McAfee [Accessed 25 Sep. 2018].

[12] M. Bedford Taylor, "The Evolution of Bitcoin Hardware," in Computer, vol. 50, no.9, pp. 58-66, 2017. doi:

10.1109/MC.2017.3571056. [online] Available at:

http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8048662&isnumber=8048614 [Accessed: 09- Sep- 2018].

[13] Ticbogota.gov.co, INFORME FINAL DE RESULTADOS PROTOTIPO BLOCKCHAIN, secretaria distrital.

[online] Available at: http://ticbogota.gov.co/sites/default/files/documentos/blockchain\_web.pdf [Accessed: 09- Sep- 2018].

[14] "LEY 697 DE 2001", www2.igac.gov.co, 2018. [Online] Available at:

http://www2.igac.gov.co/igac\_web/normograma\_files/LEY6972001.pdf [Accessed: 09- Sep- 2018].

[15] Aponte J., González A., Forero A., Giraldo J. y Viveros F., “Metodología de diseño sobre FPGA en un curso de sistemas digitales”. pp. 1-4, 2015.

[16] Chen L., Xu L., Shah N., Gao Z., Lu Y., Shi W., On Security Analysis of Proof-of-Elapsed-Time (PoET). In: Spirakis P., Tsigas P. (eds) Stabilization, Safety, and Security of Distributed Systems. SSS 2017. Lecture Notes in Computer Science, vol 10616. Springer, Cham

[17] Brownworth, A. (n.d.). Block. [online] Blockchain.mit.edu. Available at: http://Blockchain.mit.edu/block/

[Accessed 26 Sep. 2018].

[18] PREUKSCHAT, A. (2015). ¿Qué es una colisión en criptografía? – Criptografía aplicada a Bitcoin. *Oro Y Finanzas Diario Digital Del Dinero.* [online] Available at://https://www.oroyfinanzas.com/2015/02/que-es- colision-criptografia-bitcoin/. [Accessed 27 Sep. 2018].

# **[ANEXOS](#_ÍNDICE)**

En esta sección se presentan los planos del *hardware*, códigos de *VHDL* y *verilog*, imágenes tomadas del *RTL* de Quartus Prime para diagramas de bloques y controles, los proyectos realizados en Quartus Prime y la totalidad de los programas fuente desarrollados. Los anexos del presente trabajo de grado se encuentran disponibles en el OneDrive de la Pontificia Universidad Javeriana, en la carpeta TG1826, disponible en el siguiente hipervínculo: [TG1826](https://livejaverianaedu-my.sharepoint.com/:f:/g/personal/fonseca_maria_javeriana_edu_co/ErzEdEaXXU5Ll8dKUsNBmmoB9Z6WoSxgODbQjjhYHEk7sQ?e=CKxS6x).