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

Durante el proceso de desarrollo del presente trabajo de grado, es impensable no resaltar a las personas que hiceron parte del proceso. Inicialmente queremos agradecer a nuestros directores de tesis, Fransisco Fernando Viveros y Jose Luis Uribe, cuya orientación y dirección nos permitió alcanzar los objetivos planteados para el mismo.

También queremos dar a conocer nuestro sentimiento de gratitud hacia los ingenieros Francisco Calderón y Eduardo Gerlein, quienes nos brindaron su apoyo de forma desinteresada. Al laboratorio de electrónica, por brindarnos las herramientas necesarias fuera de los horarios estipulados.

A nuestros amigos Santiago Salamanca, Carolina Mercado, Juan Manuel Dominguez y Diego Varela, quienes contribuyeron de alguna forma en el desarrollo de este trabajo.

Muchas gracias a todos ustedes.

En este libro quiero manifestar mis sentimientos de gratitud por su incondicional y permanente apoyo a mis padres, Guillermo Saúl y Clara Inés, quienes estuvieron presentes en cada momento y decisión de mi vida, y me dieron la posibilidad de estudiar esta carrera universitaria, Ingeniería Electrónica. También quiero agradecer a mis hermanas Natalia Alexandra y Sandra Ximena, quienes me brindaron su amor incondicional y su voz de aliento en todo momento. A mis compañeras de tesis, Silvana y Adriana, por su amistad y compañía durante toda la carrera, y sin quienes no hubiese sido posible lograr este resultado.

Maria Paola Fonseca Páez

Gracias a mi madre María Cristina Angel Díaz, a mi padre José Octavio Aguirre Cruz y a mi novio Javier Franciso Moreno Mora, por siempre estar ahí brindándome su apoyo y compañía, por darme un consejo cuando lo necesitaba y sus mensajes de aliento para seguir con el proceso. A mis amigas Maria Paola Fonseca Páez y Silvana López Cuellar sin las cuales no habría sido posible la culminación de este trabajo de grado, pues, aunque estuviéramos cansadas y estresadas, estábamos para apoyarnos mutuamente y así poder continuar cada día con la motivación para llegar a este exitoso resultado.

Adriana Andrea Aguirre Angel

Quiero agradecer a mis compañeras por seguir este proceso conmigo, pues el trabajo en grupo fue de vital importancia para culminar el trabajo propuesto, además del compañerismo provisto. De igual forma, quiero agradecer a mi familia por brindarme el apoyo requerido para alcanzar este punto de mi carrera.

Silvana López Cuéllar

Tabla de Contenidos

Glosario

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	*Serie de operaciones matemáticas cuyo objetivo es cifrar un mensaje.*
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*
IP	*Intelectual Property, hace referencia a un sistema o invención perteneciente a un tercero.*
GUI	*Programa informático de interfaz con un usuario, que usa imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz*
Testbench	*Entorno utilizado para verificar el desempeño de un diseño o modelo*
DUT	*Producción fabricada que se somete a pruebas funcionales que establecen cuando la producción está funcionando de acuerdo con sus lineamientos*
GPU	*Coprocesador dedicado al procesamiento de gráficos u operaciones*

1 Introducción

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 cifra 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 cifrar la información de forma que no pueda obtenerse nuevamente el mensaje original; el resultado se conoce como valor *hash* o solamente *hash* [4].

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. A continuación, se enuncian algunos estudios realizados en esta área junto con la evolución del *hardware* utilizado en el transcurso del tiempo.

La primera generación de *hardware* usado para una minería fue la implementación de la *CPU (Central Processing Unit o Unidad Central de Procesamiento)* [24]. Una de las más reconocidas fue la *CPU core i7-990x* de gama alta desarrollada por *Intel*® con 6 núcleos y una capacidad de realizar 33 *Mhash* por segundo gracias a la implementación de extensiones *SIMD (Single Instruction Multiple Data)* [24]. La desventaja de esta *CPU* era el desperdicio que se generaba con un *hardware* adicional que se utilizaba para optimizar cálculos menores [24].

Por otro lado, la *GPU (Graphics Processing Unit)* brindaba mayor capacidad de cómputo que las *CPU* y eran más accesibles que las *FPGA (Field-Programmable Gate Array* o matriz de puertas programables*)* en el año 2000 [24]. Un ejemplo es la capacidad de cómputo que brindaba la *GPU*, como la *Nvidia GTX 570* alcanzando 155 *MHash/s* siendo una *GPU* accesible a diferencia de la *AMD 7970* alcanzando 0,675 *GHash/s* [24].

Posteriormente la tecnología evolucionó con la implementación de *FPGA* operando los mismos *hash* pero a 60 *W* en lugar de 200 *W* [24]. Con el auge de la *FPGA* también se implementó el *hardware* de propósito específico *ASIC (Application-Specific Integrated Circuit)* [24]. Actualmente, el *ASIC* usado por la compañía *Bitman (Antminer S9)* con poder de cómputo de 13,5 *T Hash/s* en 1,33 *W*, realiza el mayor porcentaje de generación de hash. De forma comercial, según la página oficial de *Blockchain* en su sección de gráficos se muestra que un bloque de tamaño 1.09 *MBytes* tarda aproximadamente 10 minutos para validarse [5]. Cada bloque está compuesto en promedio por 2206 transacciones.

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 como se enunció anteriormente, una GPU realizaba alrededor de 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, de aproximadamente 10 minutos como se expresó anteriormente.

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 reparte 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 página citada [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.

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.

2 Marco Teórico

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 cifra 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. A 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[12]. 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 [12].

El mensaje que se va a cifrar, 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
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000018

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.

En *Blockchain*, para agregar los llamados 'bloques' a la cadena de cada nodo, se debe 'demostrar' que se ha realizado una cierta cantidad de trabajo, también conocido como Prueba de Trabajo (PoW)[13]. 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 [13]. 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 [13].

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 del cifrado 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.

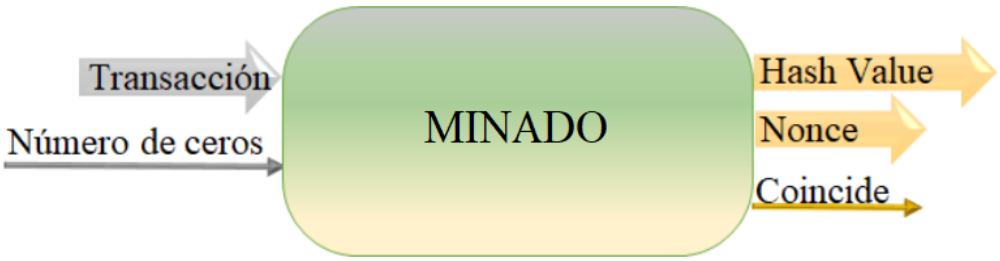

Figura 1. Esquema de entradas y salidas de un bloque de minado

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.

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)[14]. 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[14].

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 que si las condiciones de entrada o semilla se replican, se pueda obtener la misma secuencia de números.

Existen diversos métodos para la generación de secuencias pseudo-aleatorias, sin embargo, el método que se implementó para la generación del *nonce* en el sistema diseñado, fue el de los cuadrados medios.

Este es uno de los métodos más populares que fue 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 $x0^2$. A continuación x2 se genera a partir de x1 del mismo modo[15], y así continúa la secuencia.

Los conceptos anteriormente presentados de manera breve conforman la totalidad de las bases teóricas necesarias para la comprensión del proyecto. Estos serán nuevamente abordados a lo largo del desarrollo del mismo, donde compete para cada caso.

3 Objetivo del Proyecto

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.

3.1 Objetivo General

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.*

3.2 Objetivos Específicos

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.

4 Desarrollo

En este capítulo se expone 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:

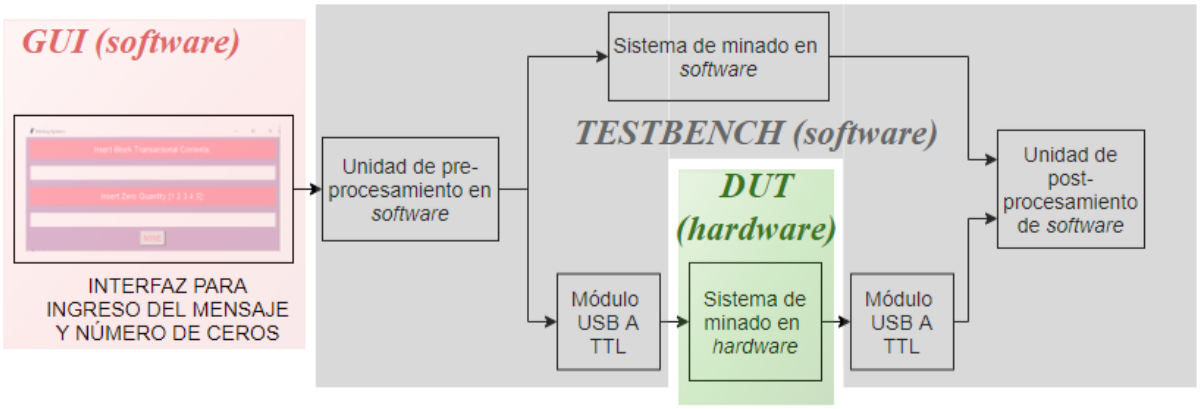

Figura 2. Diagrama general del proyecto

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 que 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 elaboración de 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 de desempeño del sistema de minado *hardware*.

El sistema total desarrollado se compone de una interfaz, implementada en *software*, en la que se ingresan las entradas del sistema, las cuales son la información y el número de ceros que se desea minar. 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 para 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. Estos datos, junto con los *nonce* que produjeron los correspondientes valores *hash* y 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* modificada para las mismas, 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 sobre el sistema operativo por medio del editor de código *Visual Studio Code*.
- Unidad de pre-procesamiento en *software* desde la 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*, utilizando 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 cifrar. 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.

4.1 Sistema de Minado en *Hardware*

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.

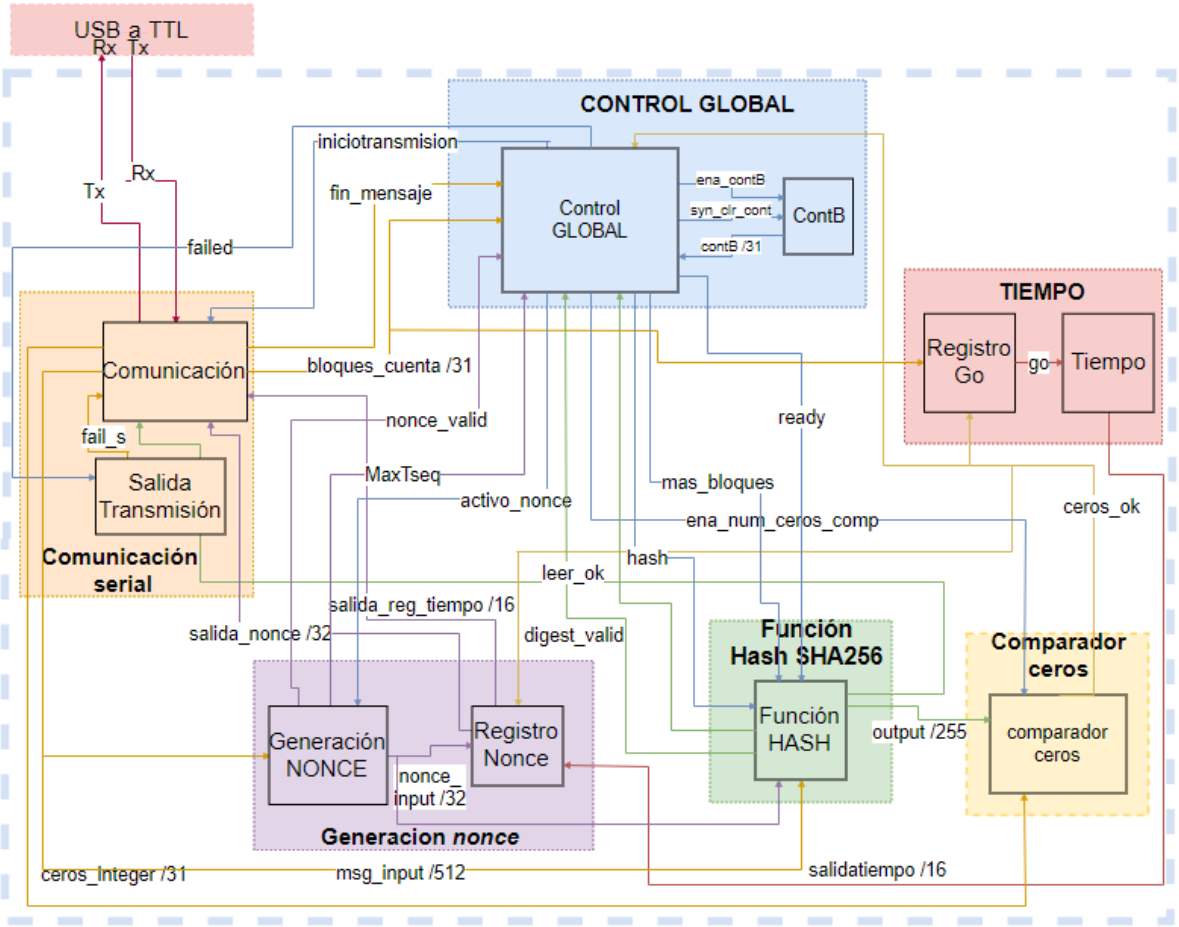

Figura 3. Diagrama de bloques del sistema de minado de hardware

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 al 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.

4.1.1 Diseño e implementación del subsistema 'función *hash sha256*' .

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.

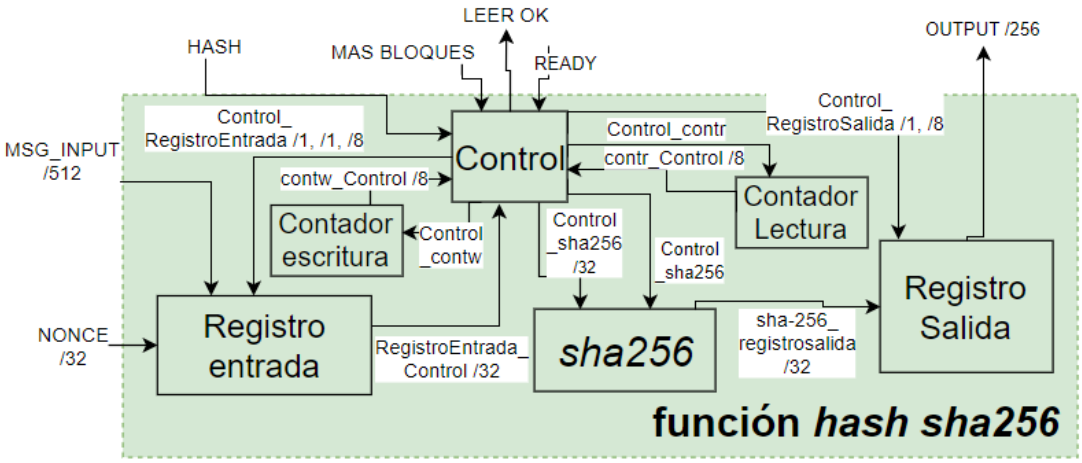

Figura 4. Diagrama de bloques de la unidad que implementa la función hash

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 en 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 *hash*.

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. De acuerdo con 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 en 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*, 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 esta restricción 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 tomar 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 tomar 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, es decir, 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 en 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, que 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*.

Líneas del código		Descripción
Desde	Hasta	
1	38	Comentarios de derechos de autor
39	40	Definición del módulo de *testbench* para el sha256
41	91	Definición de las constantes y parámetos
92	111	Definición de los registros y los cables (señales)
112	129	Instanciación del bloque sha256_core como DUT (*Device Under Test*)
130	140	Función para generar el reloj
141	153	Función para contar los ciclos
154	197	Función para proyectar información sobre el DUT
198	212	Función para reiniciar
213	234	Función para inicializar los valores de las señales
235	254	Función para proyectar los resultados de las pruebas
255	276	Función para esperar la bandera que indica que la salida es válida
277	301	Función para escribir las palabras en la entrada de 32 bits indicando una direción de 8 bits
302	328	Función para escribir un bloque de 512 bits (16 palabras) llamando a la función de escribir palabras
329	353	Función para leer una palabra indicando una dirección de 8 bits
354	380	Función para leer el nombre y versión del DUT
381	408	Función para leer la salida de 256 bits (8 palabras) llamando a la función de leer palabras
409	452	Función para probar el DUT con una entrada sencilla de un solo bloque de 512 bits
453	530	Función para probar el DUT con una entrada doble de dos bloques de 512 bits
531	565	Función para probar el sistema llamando a la función de probar el DUT para una entrada sencilla
566	599	Función para probar el sistema llamando a la función de probar el DUT para una entrada doble
600	620	Contenido del código que llama las funciones anteriores
621	626	Finalización del módulo de *testbench*

Figura 5. Tabla del análisis del código de pruebas de la IP de Github, dividida en líneas de 17mplem

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. Se determinó la forma de ingresar entradas de tamaños mayores a los 512 bits de longitud, 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’.
4. 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’.

5. Esperar un periodo de reloj.
6. 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.
7. 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 el 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 *hash*. 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.

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

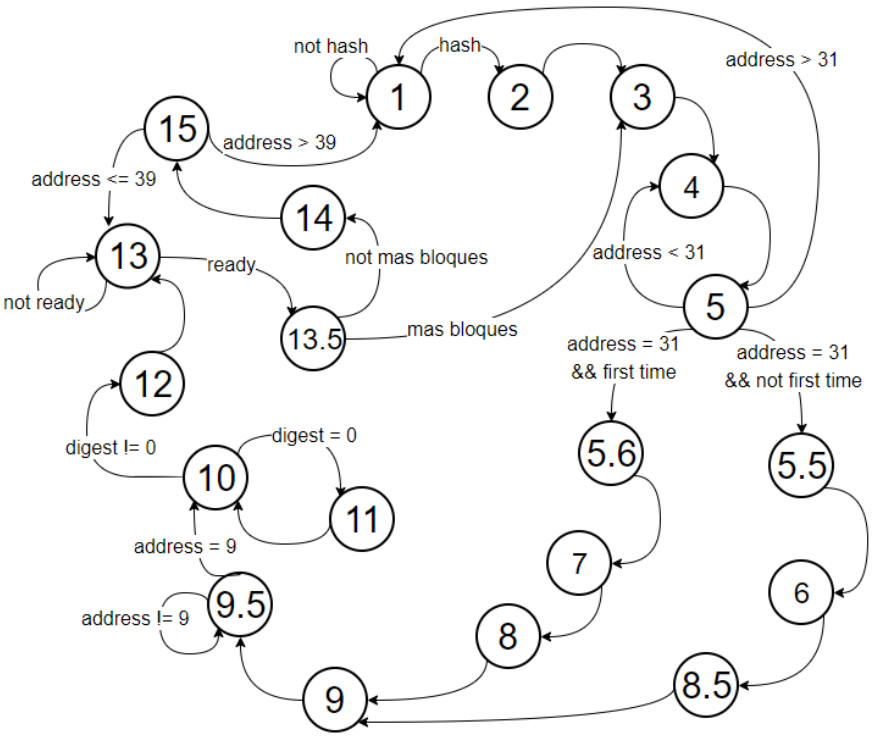

Figura 6. Máquina de estados finitos de la unidad que implementa la función hash

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 sea 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 *hash*.

4.1.2 Diseño e implementación del subsistema 'generación *nonce*'

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.

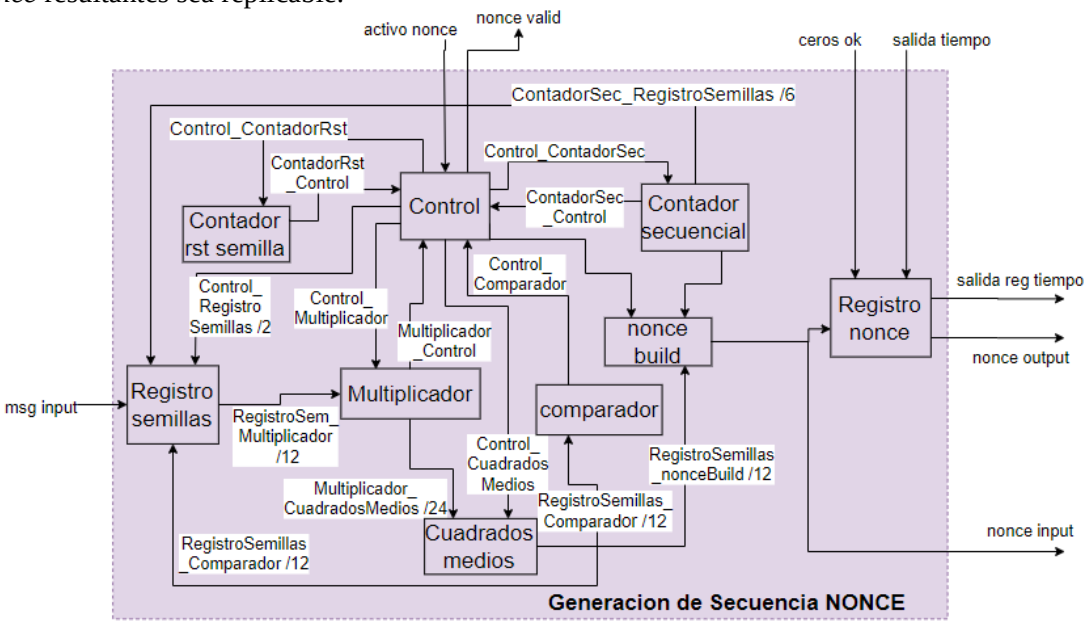

Figura 7. Diagrama de bloques de la unidad que implementa la generación de nonce

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 *nonce*.

Concepción y Diseño

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

En base a dicha investigación se encontró que es recomendable 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[22].

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[22].

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.

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. Esto con el fin de establecer el alcance al utilizar menos de un 10% de un bloque de información. Se estableció que se usaría 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 en esto, al momento del diseño se tomaron en cuenta estos inconvenientes y se concertaron 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 tomarían los 6 LSB de la porción secuencial y se concatenarían 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 2^{20}-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 x0 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.

***Nonce build*:** 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 2^n-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 *nonce*. 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*.

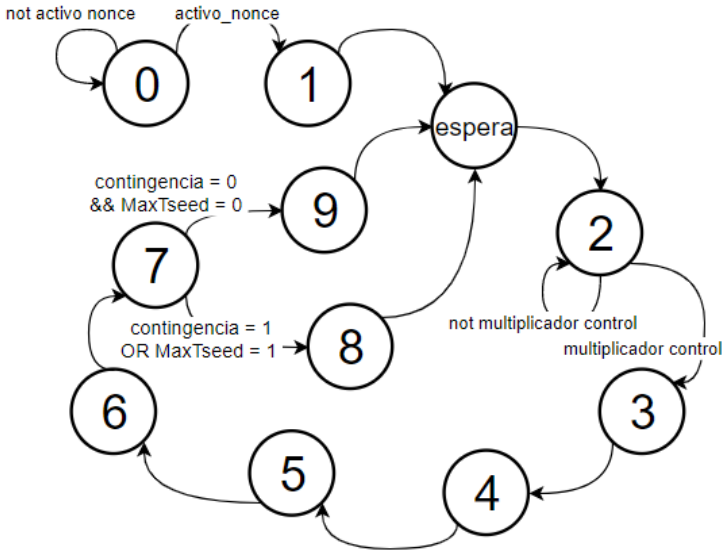

Figura 8. Máquina de estados finitos de la unidad que implementa la generación del nonce

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 el 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 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, la señal que indica que se ha alcanzado el máximo cnteo, 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 *nonce*.

4.1.3 Diseño e implementación del subsistema 'comunicación serial'

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 de la salida del sistema desde la *FPGA* hacia el computador.

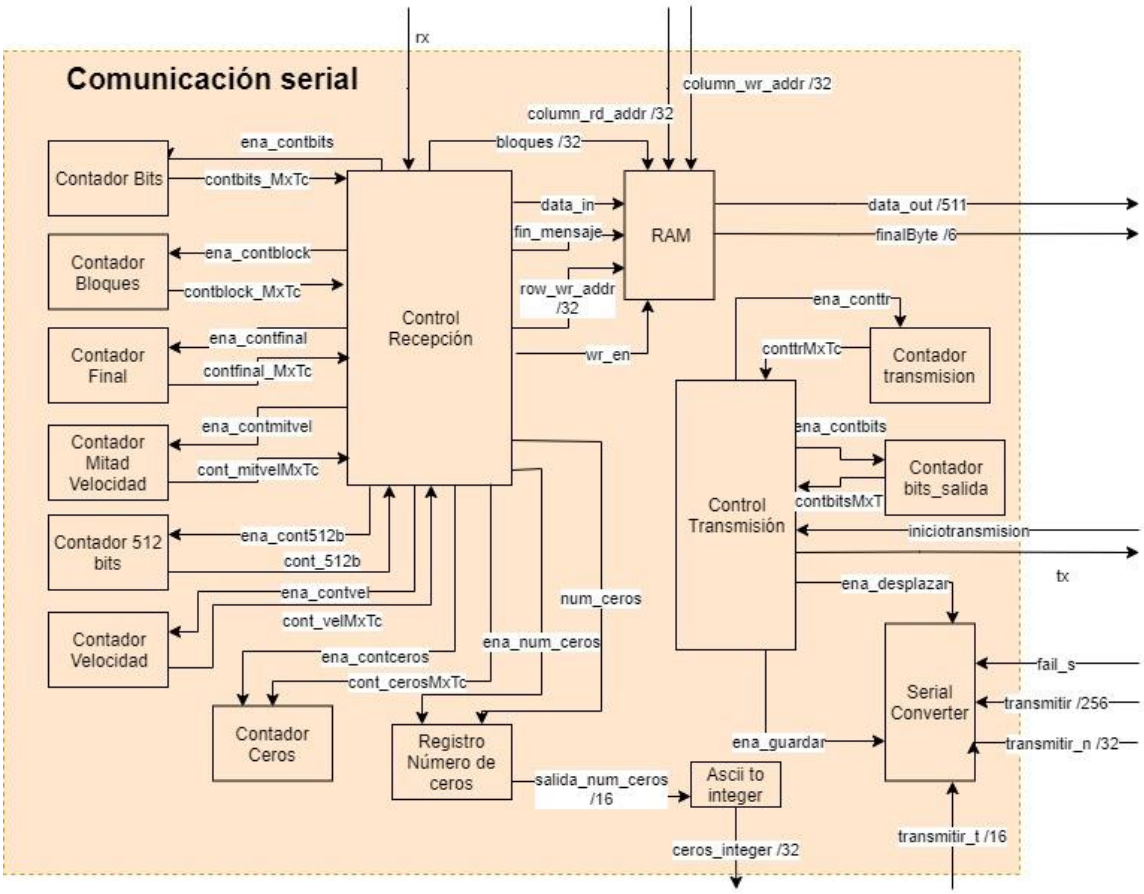

Figura 9. Diagrama de bloques del subsistema que implementa la comunicación serial.

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.

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 de 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 este tiempo, 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, al ser intuitivo para el usuario. 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*.

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**, **contador de 512 bits**, **contador de bloques** y **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 almacenarlos 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.

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

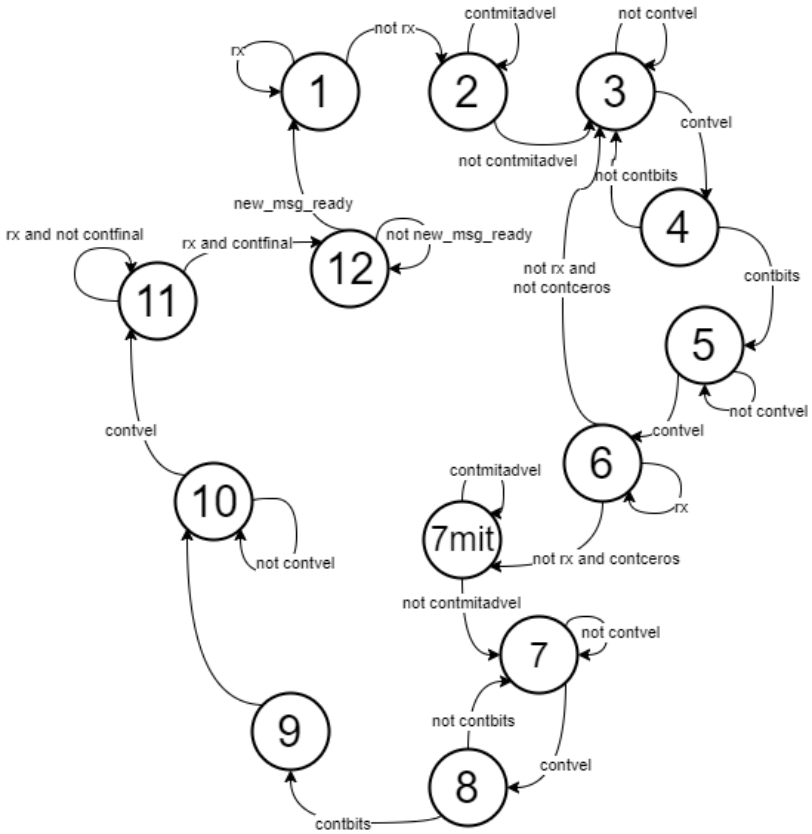

Figura 10. Máquina de estados finitos que implementa la recepción de la comunicación serial.

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 de 16 bloques por medio del contador de 512 bits que señala 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.

2. 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. Adicionalmente, 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 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 hacer efectiva 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.

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

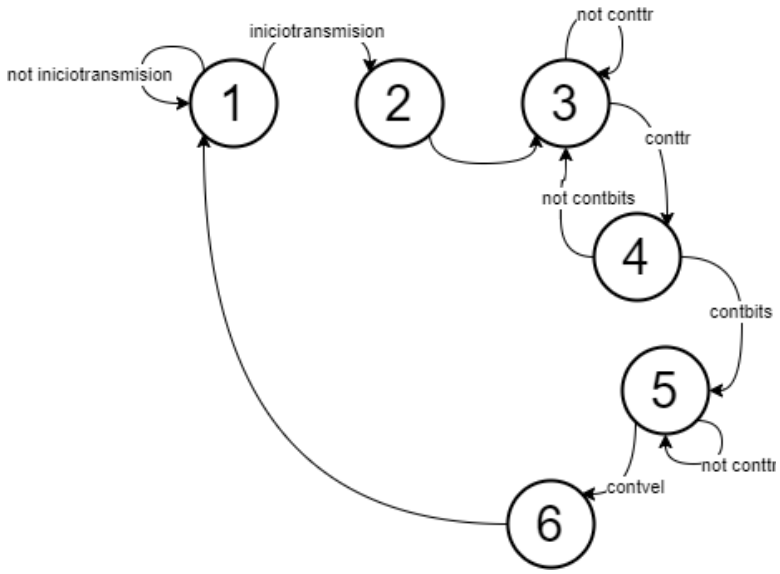

Figura 11. Máquina de estados finitos que implementa la transmisión de la comunicación serial.

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 desplazamiento de los bits mientras la cuenta de bits transmitidos sea menor a 390. Primero transmiten los 256 bits del valor *hash*, 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.

4.1.4 Diseño e implementación del subsistema 'unidad de tiempo'

En esta 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.

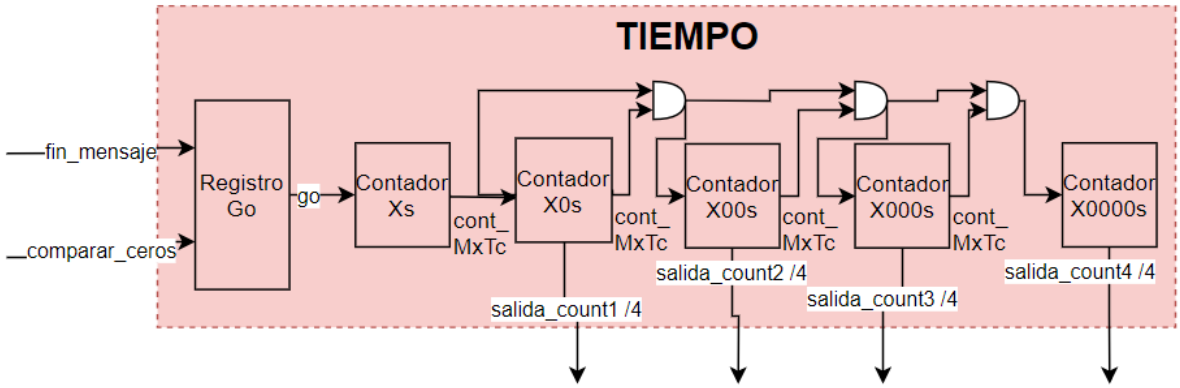

Figura 12. Diagrama de bloques de la unidad de tiempo.

.

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 en cuenta 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, esté 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, las cuales serán explicadas más adelante.

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:

Número de ceros requerido	Número de conteos de reloj	Tiempo de cuenta Xs	Tiempo de cuenta X0s	Tiempo de cuenta X00s	Tiempo de cuenta X00s	Tiempo de cuenta X000s	Tiempo de cuenta total
1	100	1us	9us	90us	900us	9000us	9999us
2	100	1us	9us	90us	900us	9000us	9999us
3	10000	100us	900us	9ms	90ms	900ms	999ms
4	100000	1ms	9ms	90ms	900ms	9s	9.999s

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

En la *Figura* 13 se muestra el cambio realizado en el primer contador (Xs) para cada número de ceros requeridos. Pues a medida que aumenta el número de ceros, aumenta el tiempo en el sistema se demora

en encontrarlos. Por lo tanto, después de tres ceros requeridos se sobrepasa el tiempo dispuesto inicialmente, por lo que se aumentó la escala de dicho contador para tres y cuatro ceros requeridos. Con lo anterior el tiempo de cuenta total para el número de los ceros mencionados, se obtiene un tiempo de cuenta total de 999ms y 9.999s respectivamente.

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 habilitador 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.

4.1.5 Diseño e implementación del subsistema ‘comparar ceros’

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. 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.

4.1.6 Diseño e implementación del subsistema ‘control global’

En esta 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 está meramente compuesto por el control y el contador de apoyo, se describirá únicamente el funcionamiento de la máquina de estados finitos.

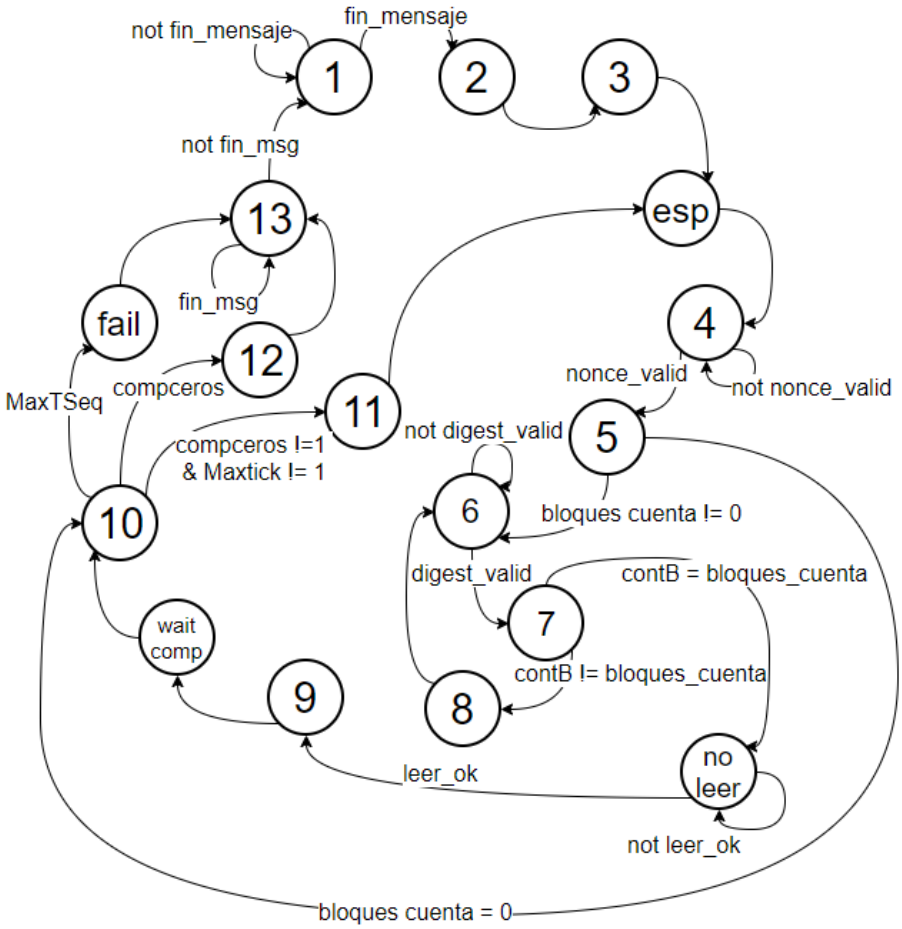

Figura 14. Máquina de estados finitos del Control Global.

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 de un 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. 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.

4.2 Sistema de Minado en *Hardware* (Variación)

Con base en el objetivo específico número 2, "Desarrollar al menos dos variaciones en la arquitectura para realizar una comparación entre éstas", se desarrolló una segunda arquitectura, en la cual se abarcan diferentes variaciones con respecto a utilización de los recursos lógicos y paralelización de procesos, utilizando como referencia la arquitectura original o primera arquitectura, descrita en el capítulo anterior: Sistema de minado de *hardware*.

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.

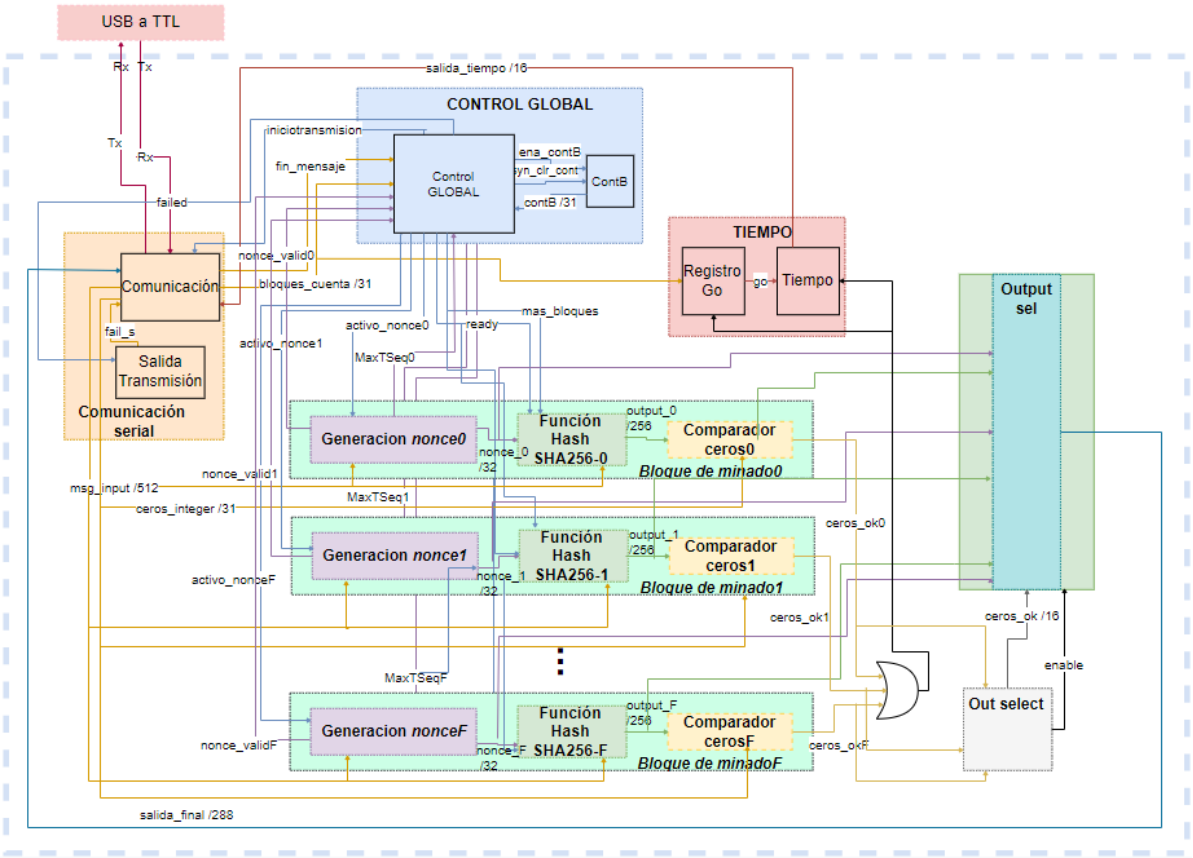

Figura 15. Diagrama de bloques para la variación del hardware.

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 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 la variación de la arquitectura.

4.2.1 Planeamiento y variaciones para la implementación de las unidades de minado

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 **función *hash*** y **comparador ceros**, solo es necesario replicarlos para cada unidad de minado.

El cambio principal que se realizó a los subsistemas de generación *nonce*, fue a sus configuraciones mas 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 efectuar al definir que con un número *m* de unidades de minado, cada contador debía contar un total de $2^k/m$ veces, siendo *k* el número de bits del contador. En este orden de ideas, el número de unidades de minado a implementar debía ser un múltiplo de 2^n. Con base al espacio de almacenamiento ocupado por la primera arquitectura se inició implementando 8

unidades de minado seguido 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.

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.

Unidad de minado	Valor inicial	Valor Final
0	0	65535
1	65536	131071
2	131072	196607
3	196608	262143
4	262144	327679
5	327680	393215
6	393216	458751
7	458752	524287
8	524288	589823
9	589824	655359
A	655360	720895
B	720896	786431
C	786432	851967
D	851968	917503
E	917504	983039
F	983040	1048575

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

Las configuraciones iniciales para cada contador secuencial en cada subsistema de generación *nonce* para 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.

4.2.2 Diseño e implementación del subsistema ‘out select’

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.

4.2.3 Diseño e implementación del subsistema 'output sel'

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

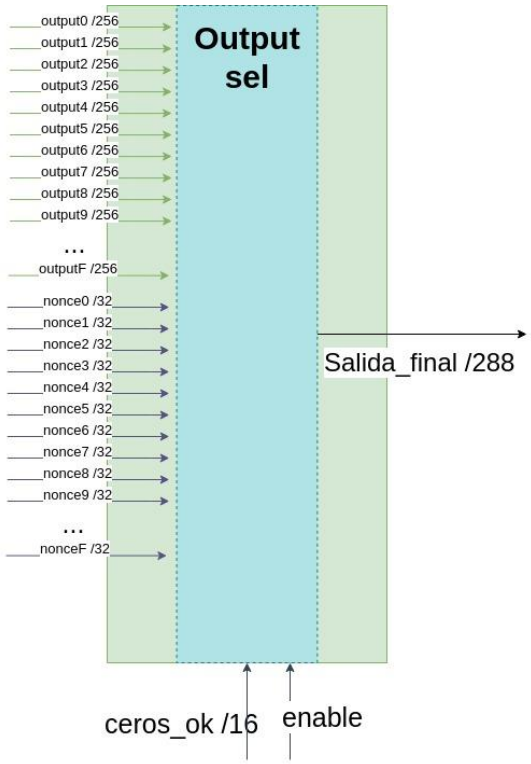

Figura 17. Diagrama de bloques del subsistema Output sel.

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, así como 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 almacenan 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.

4.2.4 Diseño e implementación del subsistema 'control global' para la variación

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 está meramente compuesto por el control y el contador de apoyo, se abordará únicamente el funcionamiento de la máquina de estados finitos.

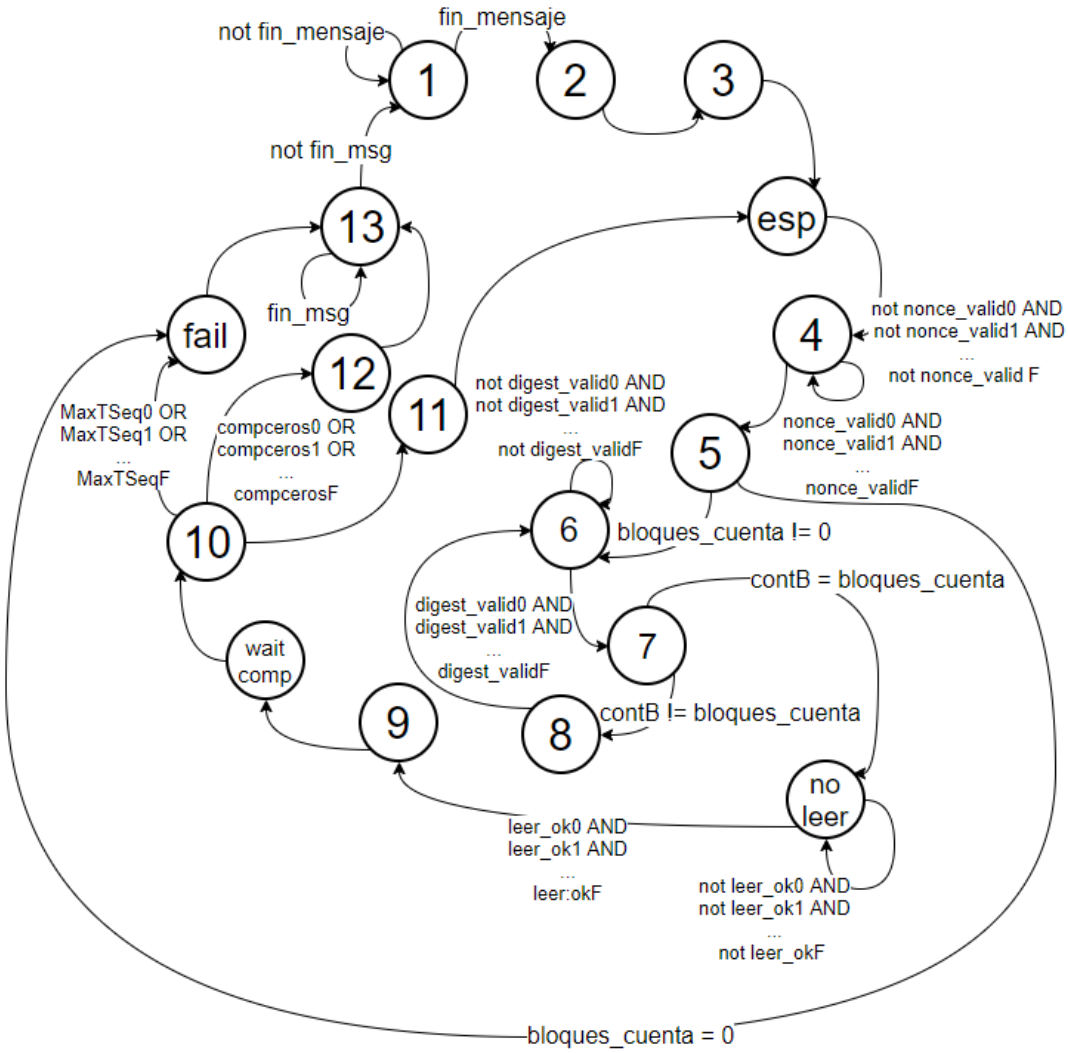

Figura 18. Máquina de estados finitos del Control Global para la variación.

El funcionamiento en la máquina de estados observada en la Figura 18, es bastante similar al anteriormente descrito en la sicción: Diseño e implementación del subsistema 'control global', 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.

4.2.5 Herramientas y dispositivos programables utilizados

Con respecto a cumplimiento del tercer objetivo específico: "Evaluar las herramientas y dispositivos programables que permitan realizar una evaluación objetiva de las arquitecturas y sus variaciones." Y teniendo en cuenta el requerimiento de elementos lógicos necesarios para la 40mplementación de la IP *sha256*, especificación que se encuentra en la pagina de *GitHub*, donde se enuncian las *FPGA* de *Altera* propuestas para la implementación del módulo, se encontró que se utilizan aproximadamente 3882 elementos lógicos y 1813 registros.

Para nuestro alcance propuesto, inicialmente de 16 bloques de información se requerían 8192 registros de memoria *RAM*. Adicionalmente, para la variación paralelizada de la arquitectura planteada, la cual requiere de mayor cantidad de elementos lógicos (no pertenecientes a la memoria *RAM*), se propuso hacer uso de una *FPGA stratix IV dispositivo EP4SGX230KF40C2* en una tarjeta de desarrollo *ALTERA Terasic DE4*, debido a que la misma posee 182400 *ALU's* y 228000 elementos lógicos. Con ello, se garantizó suficiente espacio disponible para la implementación de las dos arquitecturas planteadas.

Para la implementación de la arquitectura original, se tuvo una utilización lógica del 11% de la *FPGA*, con un total de 15447 *ALUT's* y 13178 registros lógicos, como se muestra en la *Figura* 19 de la partición total del diseño.

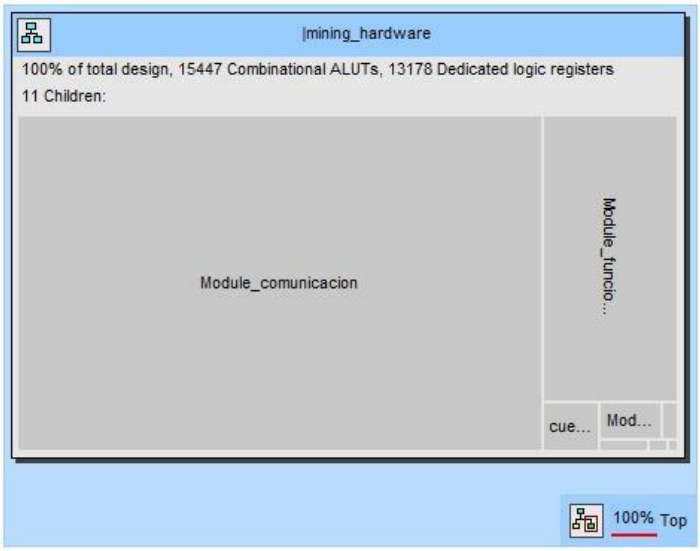

Figura 19. Partición lógica del diseño de la arquitectura original.

Para la implementación de la variación de la arquitectura, se obtuvo una utilización lógica del 43% con un total de 54165 ALUT's y 57158 registros lógicos, como se muestra en la *Figura* 20.

Figura 20. Partición lógica del diseño de la arquitectura paralelizada

El sistema desarrollado en *software* se ejecutó sobre el sistema operativo *Microsoft Windows 10* en un procesador *Intel Core i5* de sexta generación a 2.30 *GHz* con dos procesadores principales y cuatro procesadores lógicos.

4.3 Desarrollo en *Software*

Esta sección 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 efectuar 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*.

Para el ingreso de mensajes de forma manual, se implementó y adecuó un código, por medio de la librería *TKINTER*, con el fin de generar una interfaz gráfica para facilitar la interacción usuario-sistema, permitiendo el ingreso de la cantidad de ceros deseada en conjunto con el mensaje a minar. La interfaz gráfica se muestra en la *Figura* 21.

Figura 21. Interfaz gráfica.

La primera adecuación que se efectuó fue ingresar el mensaje de entrada desde un archivo externo con *padding* de n•64 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 hizo 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, haciendo referencia al mismo proceso realizado por el *hardware* (capitulo 5.1.2). 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 hacer el cálculo del *hash*, en la entrada se requiere un mensaje de longitud múltiplo de n•64 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 efectúa 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.

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 un menor

número de caracteres se requiere rellenar el mensaje hasta obtener 64 caracteres. 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 hace 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 el cálculo del número de ceros, se halló el tamaño de la cadena de mensaje de la línea y se hizo la operación que se muestra en la Ecuación 1.

$$zeroes = 64 - (size \,\%\, 64) - 1 - 8$$

Ecuación 1

A partir del resultado se generaron arreglos de *bytes* de ceros para ser almacenados antes de la transmisión a la *FPGA*. Posteriormente, con base en el tamaño de la cadena de línea multiplicado por 8 se halló el número de *bits* que componen el mensaje, y se utilizó específicamente la función *pack* del módulo *structure* 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 la siguiente forma: 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 22. 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. Cada 2 digitos hex se representa un carácter *ASCII*, como se muestra en la Figura 22. 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.

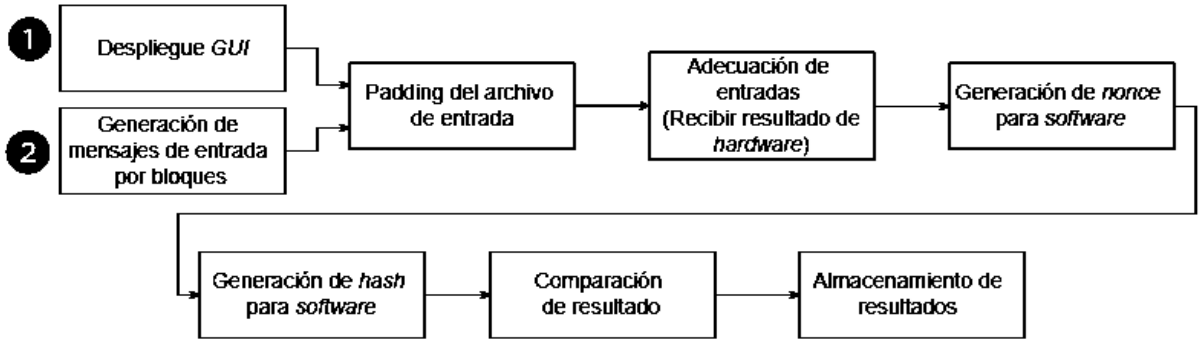

Figura 23. Secuencia de funcionamiento software.

En la Figura 23, se ilustra la secuencia de funcionamiento del script en *software*, ya sea para su primer uso en el cual se obtiene la información directamente el usuario desplegando una *GUI*, o para su segundo uso, que genera la información de entrada de manera aleatoria.

Los código implementados se pueden encontrar en los anexos, ingresando en el siguiente enlace: Códigos en python.

5 Protocolo de Pruebas

En el presente capítulo se presenta la descripción del protocolo de pruebas planteado para obtener las evidencias del cumplimiento tanto de los objetivos del trabajo de grado, 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 presente trabajo de grado nos presenta como fin la realización un sistema en *hardware*, el cual implementara 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 por medio de 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 de *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 *software*, las variables que se almacenan principalmente en los documentos de salida fueron el valor *hash*, el tiempo tanto en *hardware* como en *software*, el valor de *nonce* y si la prueba fue válida o no (*software* igual al *hardware*) como se muestra a continuación en la *Figura* 24.

```
Valid result --
Hash SW: 017fdd2be25e69cc0e336984dd691d91da288656aa1b4f235165cf304475796c
Hash HW: 017fdd2be25e69cc0e336984dd691d91da288656aa1b4f235165cf304475796c
Nonce SW: 80400005
Nonce HW: 80400005
Time SW: 0.02991199493408203
Time HW: 0007*2 us

Valid result --
Hash SW: 0b219ca90f7ab7e6854a4a8d6b5a2bb3ba9a256368aecb8722866d0abcd76ffa
Hash HW: 0b219ca90f7ab7e6854a4a8d6b5a2bb3ba9a256368aecb8722866d0abcd76ffa
Nonce SW: 10400009
Nonce HW: 10400009
Time SW: 0.04088187217712402
Time HW: 0013*2 us

Valid result --
Hash SW: 04c8acf9ea10de9ba79c2da238d25a1aaf13ef09561775eaeaa3d3e7d58c7f29
Hash HW: 04c8acf9ea10de9ba79c2da238d25a1aaf13ef09561775eaeaa3d3e7d58c7f29
Nonce SW: 7690001f
Nonce HW: 7690001f
Time SW: 0.07375526428222656
Time HW: 0047*2 us


Number of tests: 25, Successful tests: 25, Failed texts: 0
```

Figura 24. 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 a fin de corroborar el funcionamiento; si existía una prueba fallida se procedía a revisar los sistemas de *software-hardware* para verificar y arreglar la lógica errónea cuando se produce 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* 25. Con respecto a la variación de la arquitectura original, se tomaron los datos de tiempo con el propósito de comparar entre las dos arquitecturas, dando como resultado el cumplimiento objetivo específico número 4, basado en el protocolo de pruebas y las muestras de tiempo: "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". El resultado de *hardware* y *software* se fue obteniendo y almacenando por cada mensaje.

```
Time SW: 0.22738957405090332 Time HW: 0310
Time SW: 0.25331592559814453 Time HW: 0311
Time SW: 0.41590380668640137 Time HW: 0555
Time SW: 0.10869812965393066 Time HW: 0110
Time SW: 0.12270736694335938 Time HW: 0118
```

Figura 25. Fracción de archivo de salida de tiempo para hardware original.

En cuanto a 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 verificar que el sistema proporcionara un resultado válido para todas las posibilidades de cantidad de bloques en la entrada.

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 documento. Cada documento contiene 25 líneas, es decir 25 mensajes de entrada, y cada mensaje se divide por medio de un fin de línea como se muestra en la *Figura* 26. 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.

```
96Ecz2OpqMkD3tlsJi1bEtYv1DYNwE8brdhUbNuCNyavULNUzOLTdWD
abLKuIuNTQU0tSWpa7g5Ipt5kphjFQU9TqTfwbrVBq8JaGH3AwqR54g
on128ley7f2M1obAbcUxPbRINUkblKjgNl4Dgc8bYT57TKEblncFpc7
IztILjbf7XKD0W0pOw3QFUA99q36GqbOMWv2Oq9hTjWTui0kDgFG9Jg
58Uum359KyxYdRDLSGH36CpSq76V2aUrdKWh5rqwfAci0Ac7BoqSQVc
7QDThmL3YA7c5dWOQQg2AU3sVlm4Q7QwCVpneoPIGSdLYnsXbHkN1Lf
Il08B1lE79pjbNOKR1xxRx5iyxsHSkiih1w8oDci05SFhkvygZl8GPi
8uw13SUuyxpN8XUptCvPy5vxzrEOpRUgVP1b7x27iZ0Bwcl6P3zHg4n
d8wHTmeruc1hGBkhxDnaWZu76QTYuorKnH3lJ2Vkl2wi6yxn8EyYuVh
H0Sd8AqMPlbNfCa3GexmJhH3YjEOd9KpRKAwPPmetbgmxaFHRL14Wqf
JAjRllAzoY11H7KdxbdBxDCJAJmn2A3jWKw1Hy4UwjHtBAATE0u8gah
uUU6JE5e0OM73ts6YD4xhEiH3qu5Jjzp0x7WbCa3tq4liNT8oDG5wJ1
eVZbabo6VqmoluXOBAhRY7Tnyy8fW2kpU0Vi6G5vTBfNL8U82Jk64Jv
To2CG9p5l4s7z0nGGCGNGzhhHu1lG5cusrFaLMQffCgmNNqiAy33z8s
s7oybUXQqVBlfIMFeC7W1SgQDYLL5utKC2jZksICw5XjeczdErvzLq8
Onm9c3aNgaMUJq6QTDSaII8YvnoMPzPZD5jaYU0dhTHXGWLhTemgeMl
IssBQc5GX0ZKnwTbMjzWf9VDZH1kqOCXqOt7K5JlknLywlu8LJQh5PA
TcPVcb4ExSTgIUNkbUsqatnniWjmSWoexizmjufWFQW7YnX38EOCYAs
aqfNQCPEE8YjifOtMcevb5MO43RtdWFebyBOsNWFvxMcH1laYcoVc1I
ivZhWCI5ELxleE4BOenc7DOr7YKLChAYgujS7UToaZC8mE4ibZ5kYBV
ZwMRYxCERWUBaYkjHe182FCZcjJsNkwozksrilHzDB33b8Bld4J48gi
89k4X7rXvQ6qXTa7EfIRxWe9703Q24P95pmIQBbMWsKVkNSiAD1spzX
TJZc12IE5cKOxW2j1J82GUq08ysB0n6HJ0z5W6OMGlaixJvmqWIuAVF
WlvKi96fJZGsECpdol7kociph9JL7bF2IKZYeGtZWMa124hMAu0uO4A
XMY3wzzJoa1EUEDReSCcpk6JbftfA46iwXSGCCIZNFDCDVg8LXHAzta
```

```
8ZEd9OsILdMZ2x1y0RvtS2VCHYVY4G4HqK3JInO8RbdyKNGDL7BJFtiNqh5oLQQEr1jzhJOOuP7zWIjGSgJjQeZcGrudRGMyOUg8coDu4NUfWx
DkqhfDzdX4SrBANwO24ZEowA5G6SPva1cg026TSmDYizmL9O6WPQ3vujPjb2jvqDUzIcCmMGKNPgp6o0LFJGBdhGJI7kcgRivdj36oJB5b5F0k
foxCSVMHx6ShP7CgebivThIBlk930QV4Hv0IzOYQZaVjLfSk1h11ApMYjFeb4HCPS3dfNfrcXxXLcmpvC5YiDm1MKgErgixwCjWM3IrUmXHMKJ
qp9FIMcAsqnwRCkN7dQkOhyS9NWvv3ZN8uAhGQECphRIHKEuEjLUbJwL0DQfatIPxEH0eSrEjZk1fB4AtdJIjrF9fyyXs9bd9Jv1A5hNehZEcd
8WVWt439Rx05sMV4QP2yebrPyzZwXckmCfgrU5hxE2YstRQOxX9YYZNseRv9bT1xdDUfLgOjxzvvaL4pv93gIZPXJcD47SLnbDrcV4cUH8Tqgo
1Xjq923L3z3FvaqNh9p0pipLzSzCAsXCSiiO4T9gixOOV9PU3saBg7Dyf97kr4nVPqdcSTLe4eBfvq1yWSW1Ts2RRY088tGVIxmjBaKzYm2IdG
sGHujqghcaPJt97RccgG9dFCn6p9CVwIfbJE0Ym4ZXku82NIlJOZemyDlP1zPLGfwMYdKquod42VI8MTU0TQEMVpIsYZLsWSvGYxG7jiwTfn1v
JOiHan3lKhoWHJyTorRjR1wHPzHYgarA9qaZso1dFMVrsSjTYZzyoGdKUkSHUhVXVbQVSUSfyDZMKy2wkjyW3IlKaV3PqAkCW6I08Orwh6CCK
vlYWB6IPFrLboTSmdGqxeD4Ac54WbyAgcMpcR5NPydX2xY5Crp88qbPHkbJh1CzvCSWEVHvo9c3a2Y0ia3AN9AQ0Ty4gOZFNvoBNXxumEBrdku
4e5kji0NFoWdxA4GOA6tqUnycHqqbH3WOez9yaTzj5b5dLWY2Vx42URfnjVlJzP3ixQ3QZGWRU8vlStwtByanb5OZHEKRzIXa5rbu6oqvgq53r
sGHujqghcaPJt97RccgG9dFCn6p9CVwIfbJE0Ym4ZXku82NIlJOZemyDlP1zPLGfwMYdKquod42VI8MTU0TQEMVpIsYZLsWSvGYxG7jiwTfn1v
vlYWB6IPFrLboTSmdGqxeD4Ac54WbyAgcMpcR5NPydX2xY5Crp88qbPHkbJh1CzvCSWEVHvo9c3a2Y0ia3AN9AQ0Ty4gOZFNvoBNXxumEBrdku
JZqCwTemREnsm9VPsxalf8jlaYDPPTkORTPwweL5PzLwcJU06kRbZQDOOzTjFwyJx63MhEcqxnFsruMctdxaUG2Fm0bU7y9UpuLWtqPEhstaf2
dBA3Jhpr62yDwmfn8332NIYfKMZWfuxpLDCVGBNdCrejbPgrrOZimBZnziDfTZQj76PN3WnLTtqufdNHozYhBSjW1XpYo55nXMlzUgO8iBb095
xC3NQj1X7f0brQdKz3W1CLKZ7dcYNLiIOyyB1OHjecEnRLjefd0CkOzkX9Ri0bWgwPBIy7mZ24I5upAdVwfr87IFD7maqh4NUufYGkeoxoxwBo
DEzx06QF69TzQNyS4KKlRMwgN7u7nhChZsJo8Tr98vwKkoAGzUl5ZP7t6EVBLhM4tbsZamVIE1UaAqkyNUrxhcxfs9Jw03ljNE8yyFFDR0lRhP
qR0FzDCvXc0HDSepKjht8aA5RuOC0tUPPodYD6p99a7eCDA7kXxil3ZjmCqOZCoiZnyvhd1EBNnA61VMs5MDdotSHMYIMzhF6HGak0FDqZ5rr
CTItSW2Y7haktIMSU0YeVU7n8TmuTr762SSnCfuyckzNMNFMwCjT6uRVTVOaXORargZq7qM0Ya0sh1E5Y1sRw0Wei10XbTFXXt6xYjlNUPXlnb
psqLvfYjadaJxn6Jfv2K41t4LQROL1r1NFQNpcGQsduhsFmRY2rkUgdXP18yF6zm3kynu9V6oY4hycfM0EmAOd28oV9t5kAZ3OnJVULOqk7AWt
re9OnffjTjOSTGSX6Lu8WypAXW3OoMr5lyAxhLRWSCnwfv5FWzWIF5xB28znqfY2d00bIsIzblwwYmtCY5fE0Jenk6l3luYYqj6SehZPMyS6hZ
8KrioGtV1xeVuoa9nzDHudOM1S7Jh3YUVU6lJftcnOZDHVzyEPQ7lydtagino83AmxWlwQsxKcaB3QKEY7ukhpRgZyqpUxiqWQSIPZfPfU8l37
VTYthbYZZx4UvpVH4QsOeuptK6iZ73vgwolsae9zCQU3j9aFsTi0Hh6CoyLWTPeZE7pQgbylUC6v7JSlYiX16GZZtW29sjcHbIwZ8AekjLMmse
hfqR32WVVw3AotkKBFCG4kbsIGlfW6pJTsovFejsYT1q76TZ7beQLrTlBUd5g0u1zPR3Fmj3sf1C1mqAnNTVlxUMFEnSW4gcAUQgsVufR6GXAX
uBKdyhdngLXGwUbOC5AR3yp9MtQBUpojKrK8r3buQVLqn6hsVveDXCzU64ois9PtDvzGBMlnryu0NOhO83iBdt2mSqT0BsSuuKRV2ilf0VNpJy
zEVVVDOB02VlddFnZSjTo7tsBSOxBn8ZZgZfoJ82E1Y1kSAv1GK1QZJMnVaH4LULTffrbVepPeIjntfg8DaEfp4uo9Ken1kOg5Xr3yKF8FKcGs
RXpxTaRJ1Dg1AY1xwErA5GYT5n5r4KVN0IFoW5x5PLFrHpQQOCfvmQkWjuhQP7Xc63ufQzlGdVbUelBJngZAsWp1KhlgJkjjxV0aqMMSXTh174
fbw9MAJVYsAto72NAvZpLoo8CRsgGW1jJdbFvtQDHjf7Cd7g74X3eSH5udvGsTVqo4EZeGY10xZSACa30fW7oZenFMjKp8decpoO1CCxveB5rK
```

Figura 26. Ejemplo archivos generados por el entorno de pruebas como entrada (mensajes) para 1 y dos 2 bloques.

Con el objetivo de comprender 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 bloques y un ciclo interno para el manejo del número de ceros por bloque. El ciclo del número de ceros inicia en un 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 evidenciaron diferencias entre la salida del *hash* de los dos sistemas de minado, para una entrada de más de 8 bloques.

Así pues, al evidenciar este resultado de *hash* que difería entre *software* y *hardware*, se verificaron las posibles variaciones en el módulo para generar el valor de *nonce* de cada sistema. 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 valor *hash*. Se visualizaron los resultados de valor *hash* para cada iteración por bloque, es decir, si la entrada tiene 10 bloques, existen 9 resultados parciales de hash y un último resultado, el cual es válido para la totalidad del mensaje.

Al correr esta prueba (más de 8 bloques visualizando la salida para cada resultado parcial), se evidenció que dicha variable era igual para todos los bloques a excepción del último, y con base en este hecho se decidió observar la variable que actualiza su valor inicial para cada *hash* dentro de la IP del *hardware*, y se obtuvo que este resultado erróneo proviene directamente de la IP. Al realizar el análisis descrito anteriormente se decidió contactar por medio de *GitHub* al autor de la IP para informar del problema en la generación del *hash* en la IP, que genera discrepancias entre el *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 propuesto, pues realizar la comparación entre resultado de *software* y este sistema en *hardware* es innecesario debido a la forma en que se obtiene el resultado, es decir, se obtiene un distinto resultado entre los dos sistemas. Esto es debido a que la implementación en *hardware* es más eficiente al no realizar el proceso de manera secuencial como el código en *software*.

Para la arquitectura paralelizada no se realizó la comparación de los resultados obtenidos con el *software*, debido a que se asumió su apropiado funcionamiento, con base en los resultados de comparación obtenidos de la arquitectura base con el mismo.

A partir de esto se decidió descartar la opción de realizar la comparación entre resultados y la toma de muestras del *software* al mismo tiempo con el *hardware* de la variación, a diferencia de la arquitectura original. Para este sistema se generaron dos documentos, el primero solo con los parámetros obtenidos del *hardware* como el valor *hash*, el tiempo y el *nonce*, como se muestra en la *Figura* 27. Adicionalmente se produjo un documento que solo contiene el tiempo, con el fin de realizar diferentes análisis sobre esta variable.

```
01 Hash HW: 029ad2874c25ee34016a4cd1811d4e1e3aeca6de0bd06e02e6dd484ab88b1a0f
Nonce HW: 71070001
Time HW: 0009 us

02 Hash HW: 0433a5126ab6417c5d481e4c5d7d13fefd0631cee0496371a5dd732018586b56
Nonce HW: 710f0001
Time HW: 0009 us

03 Hash HW: 01116cfa1039d08f922d5d1cb9cad9d9b8aebe7caa4fb5e0d50b6ebe4781441d
Nonce HW: 78400002
Time HW: 0018 us
```

Figura 27. 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 con cada sistema, para un total de pruebas de 1280. La mitad de estas pruebas proveniente de arquitectura original y la otra mitad de las pruebas provenientes de la arquitectura paralelizada. La cantidad de pruebas se planteó como se muestra a continuación: 20 mensajes con 1, 2, 3, y 4 ceros, con bloques de tamaño de entrada entre 1 y 8 bloques de longitud, 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 valor *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*. Para aclarar el funcionamiento del entorno de pruebas, en la *Figura* 28 se ilustra el flujo de alto nivel para el manejo de las mismas.

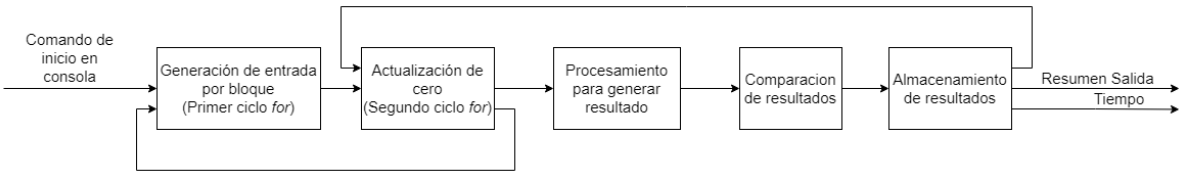

Figura 28. Flujo de funcionamiento de pruebas.

Asimismo, se realizó una prueba de funcionamiento adicional, la cual se encuentra fuera del alcance del protocolo de pruebas, pues su fin es realizar una demostración de aplicabilidad para el proyecto. La prueba consiste en minar un mensaje el cual simula una transacción para *bitcoin*. El mensaje se adecuó e ingresó como se muestra en la *Figura 29*.

Hash 52af66c2df5a62eccbfa450d259cd105fce01e4cbeed5efb20e2a22b83cf4367>>Status Confirmada>>Hora de Recepcion 2019-11-19 21:35>>Tamano 225 bytes>>Peso 900>>Incluido en el bloque 604577>>Confirmaciones 127>>Entrada total 0.00912257 BTC>>Salida total 0.00905901 BTC>>Comisiones 0.00006356 BTC>>Tarifa por byte 28.249 sat/B>>Tarifa por unidad de peso 7.062 sat/WU>>Value when transacted 73,79 US$

Figura 29. Mensaje de ingreso para simular una transacción.

A continuación, en la *Figura 30**Figura 29* se muestra el resultado para la prueba de la transacción con arquitectura original. En la *Figura 31* se muestra la salida para la arquitectura paralelizada.

```
Valid result --
Hash SW: 0000c395d26894782c4d951c6cc71ffcbedb0b2cf7817f016e6826f57322a249
Hash HW: 0000c395d26894782c4d951c6cc71ffcbedb0b2cf7817f016e6826f57322a249
Nonce SW: b0300eeb
Nonce HW: b0300eeb
Time SW: 35.31930422782898
Time HW: 1043us

Number of tests: 1, Successful tests: 1, Failed texts: 0
```

Figura 30. Resultado para mensaje de ingreso para simular una transacción con arquitectura original.

```
Invalid result --
Hash SW: 0000c395d26894782c4d951c6cc71ffcbedb0b2cf7817f016e6826f57322a249
Hash HW: 0000e927e74efdad753e1be3d3a340c6353cc586dfd558b110796bfc29e031ce
Nonce SW: b0300eeb
Nonce HW: 4a8f029d
Time SW: 36.945446491241455
Time HW: 5438us

Number of tests: 1, Successful tests: 0, Failed texts: 1
```

Figura 31. Resultado para mensaje de ingreso para simular una transacción con arquitectura paralelizada.

La Figura 30 y Figura 31, muestran el resultado de la transacción para la arquitectura original y la variación de la arquitectura respectivamente. Como se mencionó anteriormente la arquitectura original tiene como sistema de referencia el *software*; sin embargo, la arquitectura paralelizada da un resultado diferente por la forma en que se obtiene el resultado, es decir que existen diferentes valores del *nonce* que suplen el requerimiento del número de ceros en el valor *hash*, los cuales pueden ser encontrados de forma más rápida con la paralelización de módulos del sistema.

6 Análisis de Resultados

En este capítulo se presenta el análisis e interpretación de los resultados obtenidos al seguir el protocolo de pruebas desarrollado en el capítulo anterior, explicando en cada instancia la información que es presentada.

Los resultados obtenidos en el protocolo de pruebas almacenados en archivos de texto (.txt), se exportaron y procesaron en *Microsoft Excel*®, con el fin de realizar un análisis basado del desempeño en tiempo con respecto al número de ceros y el número de bloques. Es decir, se buscó satisfacer el objetivo número 5: "Analizar el desempeño de las arquitecturas propuestas a partir del entorno planteado".

Se buscó identificar las diferencias en el comportamiento del tiempo por número de ceros entre los sistemas: la arquitectura base en *hardware* (*HW*), la variación de arquitectura en *hardware* (*HW VAR*) y el sistema en *software* (*SW*). Para realizar este análisis se propuso realizar un gráfico de barras tipo columna agrupada para visualizar los tres sistemas, donde se expone en el eje Y (escala logarítmica) la variación del tiempo contra cada sistema agrupado por número de ceros en el eje X, como se muestra en las figuras: Figura 32, Figura 33, Figura 34, Figura 35, Figura 36, Figura 37, Figura 38 y Figura 39.

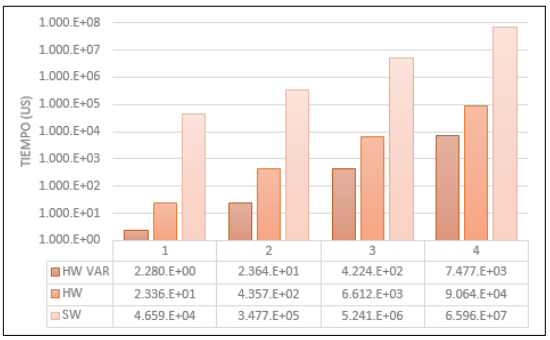

Figura 32. Gráfico para 1 bloque - análisis de tiempo cuando se varía el número de ceros.

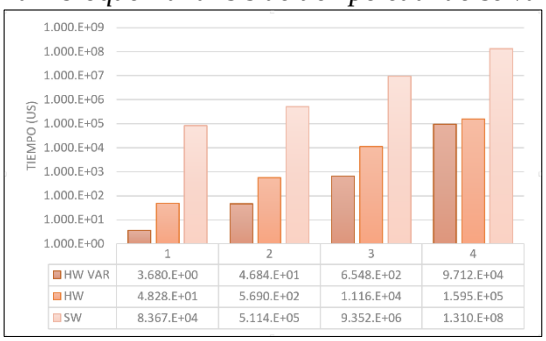

Figura 33. Gráfico para 2 bloques - análisis de tiempo cuando se varía el número de ceros.

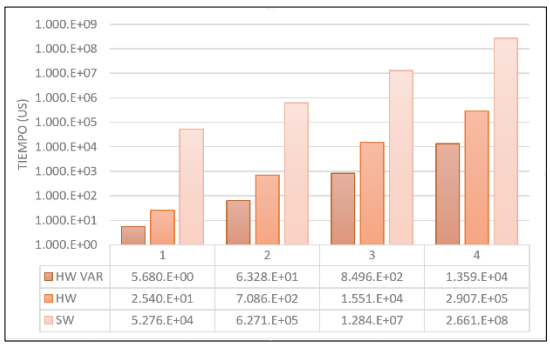

Figura 34. Gráfico para 3 bloques - análisis de tiempo cuando se varía el número de ceros.

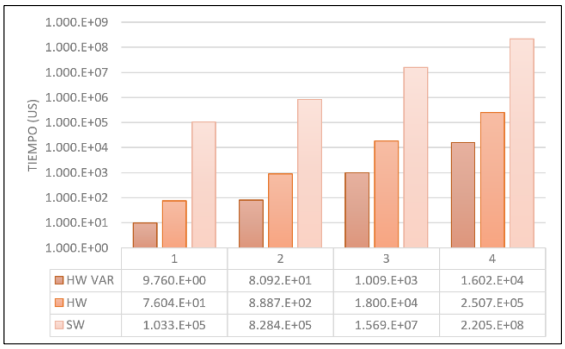

Figura 35. Gráfico para 4 bloques - análisis de tiempo cuando se varía el número de ceros

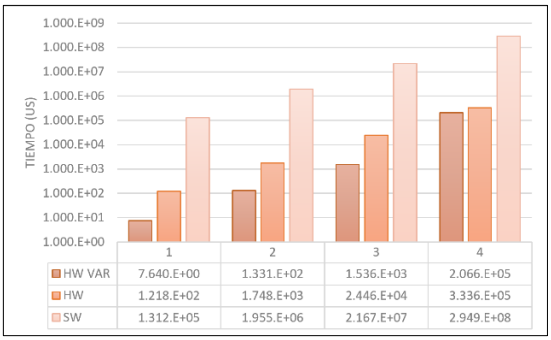

Figura 36. Gráfico para 5 bloques - análisis de tiempo cuando se varía el número de ceros.

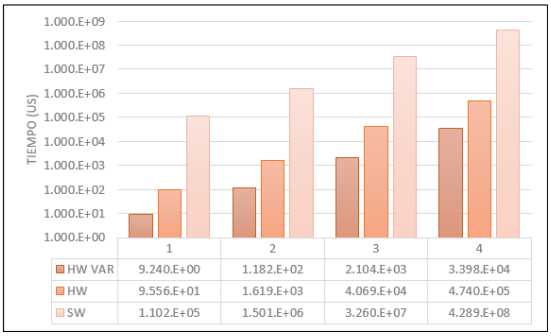

Figura 37. Gráfico para 6 bloques - análisis de tiempo cuando se varía el número de ceros.

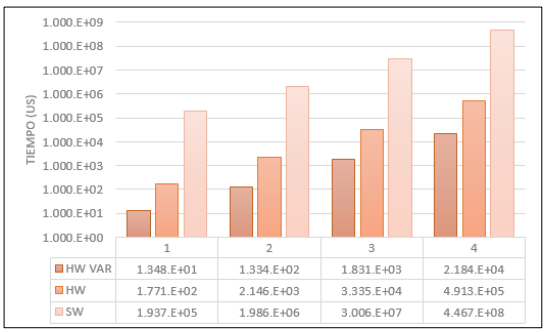

Figura 38. Gráfico para 7 bloques - análisis de tiempo cuando se varía el número de ceros.

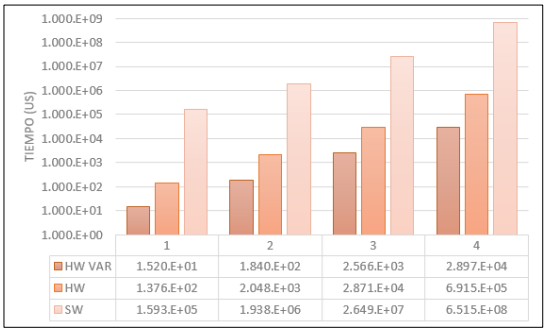

Figura 39. Gráfico para 8 bloques - análisis de tiempo cuando se varía el número de ceros.

Como se puede evidenciar en las figuras: *Figura* 32, *Figura* 33, Figura 34, *Figura* 35, *Figura* 36, *Figura* 37, Figura 38 y *Figura* 39. La diferencia entre el tiempo medido del sistema de minado en *software* con respecto al tiempo medido en ambos sistemas de *hardware* es abismalmente notoria. Con base a esto, se realizaron las gráficas utilizando una escala logarítmica en el eje del tiempo. Esto permitió evidenciar un comportamiento exponencial para una base de tiempo lineal, teniendo en cuenta que la tendencia de los gráficos presentados es lineal creciente con base de tiempo logarítmica. De igual manera, se evidencia la variación del tiempo medido con respecto al número de ceros, de forma que con cada aumento de esta variable, se incrementa 1000 veces el tiempo que tarda el sistema en encontrar la solución (valor *hash*).

Igualmente, se realizó el análisis para la variación del tiempo medido entre los sistemas de minado, por tamaño del mensaje de entrada como se muestra en las figuras: *Figura* 40, *Figura* 41 y Figura 42. En esta sección se evidencia cómo afecta al tiempo medido, los cambios en las variables de número de ceros y cantidad bloques de información en la entrada, presentando un crecimiento exponencial con el aumento de tamaño de la información en la entrada para todas las cantidades de ceros usadas. En estas figuras cada línea hace referencia a una cantidad de ceros diferente, siendo un cero la línea azul, dos ceros la roja, tres la gris y cuatro la naranja. Para cada una de estas, las líneas punteadas son los resultados obtenidos graficados, mientras que las líneas gruesas representan las tendencias seguidas en cada caso.

A partir de estos resultados se evidencia que entre la cantidad de ceros y la longitud del mensaje, es la cantidad de ceros la variable que más repercute en el rendimiento en tiempo de los sistemas.

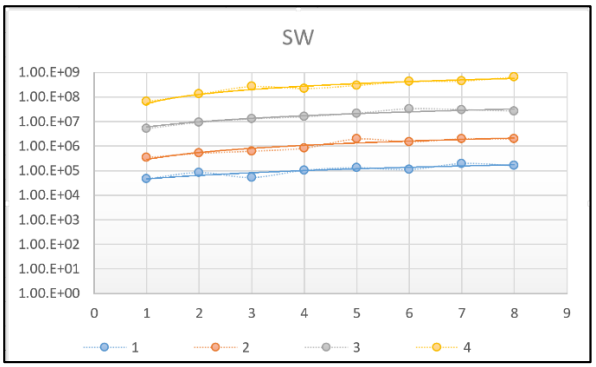

Figura 40. Análisis de tiempo por longitud en bloques para software.

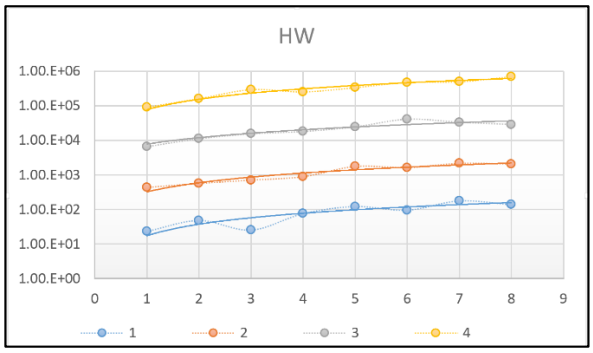

Figura 41. Análisis de tiempo por longitud en bloques para hardware.

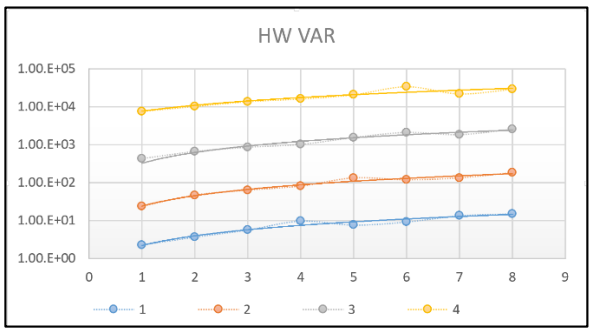

Figura 42. Análisis de tiempo por longitud en bloques para la variación del hardware.

En la *Figura* 43 se muestra el porcentaje de tiempo que tarda la variación de arquitectura con respecto al tiempo que tarda la arquitectura original, en realizar el proceso de minado, por cantidad de bloques de información de entrada. La tabla X permite evidenciar que el rendimiento de la arquitectura paralelizada es aproximadamente 10 veces más rápida.

Número de bloques	%HW VAR
1	7.456
2	6.953
3	4.908
4	8.484
5	6.589
6	7.329
7	5.940
8	8.289

Figura 43. Tabla de porcentaje de tiempo de la segunda arquitectura con respecto a la arquitectura original por cantidad de bloques.

Al analizar el funcionamiento por cada sistema se puede afirmar que el sistema en *software* es el que más tiempo tarda para realizar el proceso de minado sin importar el número de ceros o tamaño del mensaje, pues tarda aproximadamente 994 veces más que el proceso de minado de la arquitectura base y 14316 veces más que el proceso de minado de la arquitectura con unidades de minado paralelizadas. Luego del *software*, se tiene la arquitectura base, y por último, se tiene como sistema más rápido la variación de arquitectura la cual solo tarda aproximadamente un 10% del tiempo le toma a la arquitectura base, como se muestra la *Figura* 43. El resultado es el esperado, pues para el sistema en *software*, no existe un *hardware* dedicado a este proceso, sino que se realizan más funciones al tiempo, al usar un *hardware* de propósito general en el computador externo.

La arquitectura original, para la generación de *nonce* hace uso de una sección secuencial, que es dependiente de un contador, el cual hace un barrido en todos los posibles valores del mismo, mientras que en la variación de arquitectura, al paralelizar este proceso, se tiene más de un contador (cada uno iniciando desde diferentes valores). Esto permite un barrido múltiple en paralelo, que en conjunto, resulta en probar diferentes valores del barrido de la arquitectura original. Cabe resaltar que pueden existir diferentes valores del *nonce* que cumplan con el número de ceros requeridos para el valor *hash*, los cuales pueden ser encontrados de forma más rápida con el barrido segmentado.

7 Conclusiones y Recomendaciones

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.

El proyecto cumple con la función principal de generar el cifrado de una transacción, enunciada en el objetivo general "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*.", supliendo los requerimientos mínimos para una aplicacion real de criptomonedas. Es esta información generada en el proceso de minado, la que se almacena en un bloque dentro de una red *blockchain*.

Con el fin de suplir el objetivo específico número uno: "Diseñar una arquitectura del sistema que supla los requerimientos funcionales del algoritmo de consenso PoW", en el diseño de los sistemas realizados (*software* y *hardware*) se implementó la posibilidad de minar (obtener un valor *hash* que supla con características especificadas) para un número variable de ceros, es decir que el valor *hash* provisto por los sistemas, puede cumplir con un acertijo cumpliendo con el proceso de minado propuesto por el algoritmo de consenso *Proof of Work* específicamente para obtener entre 4 y 32 ceros binarios al inicio del valor *hash*.

El protocolo de pruebas implementado permitió evidenciar el comportamiento anormal entre las IP originales de *hash* en *hardware* y *software.* Al realizar un considerable número de pruebas e identificar las discrepancias entre los resultados realizando un riguroso seguimiento de la raíz del problema, se encontró que dichas discrepancias siempre son presentadas a partir del 8vo bloque de información en una entrada.

A pesar de esta dificultad evidenciada en la etapa final de pruebas del sistema, se considera satisfactorio el resultado obtenido del presente trabajo de grado, pues la arquitectura diseñada, además de cumplir con el objetivo funcional del proyecto, permite entradas de hasta 8 bloques de información (es decir una entrada de 4096 bits). Esta, con las pertinentes correcciones a la IP del *hardware*, representa para este proyecto el escalamiento y evolución del mismo, permitiendo el ingreso de entradas de tamaño incluso superior.

Otro de los motivos, es que durante el desarrollo se implementó un método alternativo para la generación del valor *nonce,* involucrando la concatenación de un valor generado de forma pseudo-aleatoria, con un valor generado de forma secuencial, en vez de solo usar un valor meramente generado de manera secuencial. Esto propone para futuros trabajos de investigación, tanto la utilización de otros métodos alternativos para la generación del *nonce,* como el desarrollo de un entorno de *blockchain*, en el cual sea posible generar bloques a partir de transacciones validadas por un mismo sistema.

En adición a esto, la estrategia que se usó para implementar el entorno de comparación entre los resultados obtenidos de los sistemas de minado en *hardware* y *software*, resultó completamente funcional, dando como resultado un porcentaje de error entre las salidas igual al 0%.

Un punto a mejorar del presente proyecto de trabajo de grado, es el de la medición del tiempo en el sistema de minado en *hardware*, pues en la etapa de pruebas se evidenció la necesidad de usar diferentes

escalas de tiempo, dependiendo de la cantidad de ceros y la longitud de la información a la entrada del sistema.

Con base a los objetivos específicos, se encontraron resultados que se consideran satisfactorios con respecto a la segunda arquitectura, desarrollada a partir de la diseñada e implementada originalmente, de forma que se logró una disminución significativa en el tiempo de minado, la cual se considera proporcional al aumento en la cantidad de unidades de minado implementadas internamente en la arquitectura.

Esta segunda arquitectura, tuvo un desempeño del proceso de minado, en promedio 10 veces más rápida que la primera arquitectura, con un aumento en la utilización de los recursos lógicos de la *FPGA*, de aproximadamente 4 veces.

Con respecto al análisis de los resultados de las pruebas que se realizaron, se puede concluir que el factor más influyente sobre el tiempo de minado de los sitemas diseñados, fue el número de ceros requeridos por encima de la longitud de la información. Esto con base en las tendencias de aumento del tiempo evidenciadas en los gráficos. También se evidenció que los sistemas en *harware* son considerablemente más rápidos que el sistema en *software*. Sin embargo, se debe tener en cuenta que no es posible realizar una comparación objetiva con respecto al tiempo de minado del sistema de *software*, debido a que el sistema operativo reparte los recursos del procesador entre diferentes tareas.

8 Bibliografía

[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] Descriptions of SHA-256, SHA-384, and SHA-512 . [online] Available at: http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf

[13] Dr. Baliga A. (2007). Corporate CTO Office. Understanding Blockchain Consensus Models [online] Available at: https://pdfs.semanticscholar.org/da8a/37b10bc1521a4d3de925d7ebc44bb606d740.pdf?_ga=2.21200635.1919538867.1522092864-1798624458.1520283070&source=post_page----------

[14] Rogaway, P. (2003). Nonce-Based Symmetric Encryption. [online] Web.cs.ucdavis.edu. Available at: http://www.cs.ucdavis.edu/~rogaway/papers/nonce.pdf [Accessed 30 Nov. 2019].

[15] Webs.um.es. (n.d.). Generación de números aleatorios. [online] Available at: https://webs.um.es/mpulido/miwiki/lib/exe/fetch.php?id=amio&cache=cache&media=wiki:simt1b.pdf [Accessed 30 Nov. 2019

[16] GitHub. (2019). *Build software better, together*. [online] Available at: https://github.com/WaveShapePlay/ArduinoPyserialComConnect[Accessed 2 Dec. 2019].

[17] Strömbergson, J. (2019). secworks/sha256. [online] GitHub. Available at: https://github.com/secworks/sha256 [Accessed 2 Jun. 2019].

[18] Strömbergson, J. (2019). Secure Hash Algorithm. [online] Csrc.nist.gov. Available at: https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/examples/sha256.pdf [Accessed 2 Jul. 2019].

[19] Pythonhosted.org. Welcome to pySerial's documentation — pySerial 3.0 documentation. [online] Available at: https://pythonhosted.org/pyserial/[Accessed 17 Oct. 2019].

[20] GitHub. WaveShapePlay/ArduinoPyserialComConnect. [online] Available at: https://github.com/WaveShapePlay/ArduinoPyserialComConnect [Accessed 17 Oct. 2019].

[21] Xilinx.com. (2019). Instantiating a Verilog Module in a VHDL Design Unit. [online] Available at: https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ism_p_instantiating_verilog_module_mixedlang.htm [Accessed 23 Sep. 2019].

[22] 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 [Accessed 5 Ago. 2019].

[23] Youtube.com. (2019). TkInter para Python: Programar los botones, suma de dos números (Básico). [online] Available at: https://www.youtube.com/watch?v=D4_SQawLwQs [Accessed 2 Nov. 2019].

[24] 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- Sep2018].

9 Anexos

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.