Parte 1 del proyecto: Modelo de alto nivel

Isaac F. Fonseca Segura
Escuela de Ingeniería Eléctrica
Universidad de Costa Rica
San José, Costa Rica
Email: isaac.fonsecasegura@ucr.ac.cr

***Resumen*—En este trabajo se plantean dos diseños, uno optimizado en área y otro en desempeño, para implementar un sistema que calcula hashes establecido un target. Se profundiza en el diseño en bloques de las implementaciones para lograr describir mejor el funcionamiento del sistema.**

INTRODUCCIÓN

Las funciones de hash consisten en un tipo de función que toma cualquier array de datos de cualquier tamaño y lo convierte en un array de una longitud definida, asegurando que ante una entrada con cualquier cambio la salida será completamente distinta, y además debe ser casi imposible descifrar la entrada a partir de la salida [1] [2].

En el presente trabajo se plantea el posible diseño para un módulo que calcula el hash de un bloque de datos concatenado a un nonce, dado un target inicial. Los dos primeros bytes del hash deben ser menores al target para ser considerados como bounty. Una representación del sistema se muestra en la Figura 1

Para esto primero se planteó un diseño que siguiera exactamente el algoritmo, este diseño se considera como el optimizado en área, ya que tendrá sólo los bloques necesarios. Luego se tomó este diseño y se intentó de paralelizar, añadiendo una entrada más y un módulo de control que pueda manejar las salidas. Se verá adelante como se propuso la paralelización para que esta fuera eficiente y así el desempeño pueda aumentar.

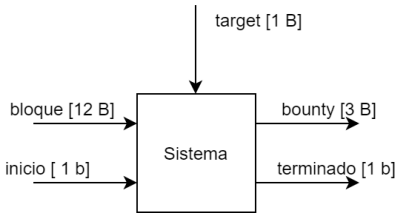

Figura 1. Representación del sistema con sus entradas y salidas. Observar que cada entrada tiene entre paréntesis cuadrados el tamaño del bus.

6 de junio, 2021

I. DISEÑO OPTIMIZADO EN ÁREA

Para el diseño optimizado en área decidí seguir el algoritmo dado para *micro_ucr_hash* y modularizarlo en bloques que me parecieran convenientes. A continuación presento una breve explicación de la arquitectura.

I-A. Cargar datos

Este es el primer bloque de la arquitectura, este recibe las entradas de bloque de datos y la condición de inicio. Su función consiste en cargar los datos del bloque de datos una vez dada la condición de inicio y concatenar el nonce a bloque. El nonce está inicializado en [0x00, 0x00, 0x00, 0x00] dentro del bloque y siempre debe ser así para operar correctamente. Cabe notar que el nonce en esta implementación es interno a la arquitectura y no se puede modificar desde el mundo exterior.

A la salida del bloque se obtiene el bloque de datos con el nonce concatenado, es decir W[0:16], al cual dentro de la arquitectura se le llamará la parte baja de W o *W_l*.

I-B. Generar W

Una vez que el bloque anterior carga la parte baja de W es posible llenar la parte alta (W[16:31]) a partir de esta aplicando operaciones lógicas (ver bloque en Anexo 4) secuencialmente y así obtener W mediante la concatenación de la parte alta y baja. Este bloque a la salida retorna W[0:31].

I-C. Algoritmo de Hashing

Este bloque toma W y aplica el algoritmo de hashing dado (ver bloque en Anexo 4), el cual consiste en realizar operaciones lógicas bajo el conjunto de datos de W. Con cada dato de W se genera la entrada para la siguiente iteración del algoritmo de hashing. Una vez utilizados todos los datos de W se procede a retornar el hash H.

I-D. Comparador target-hash

Como su nombre lo indica este comparar el target con el hash. Para ello recibe una entrada principal del exterior, que es el target, y una entrada interna que es el hash calculado. Se revisa si los dos primeros bytes del hash son menores al target, de ser así se ha encontrado un bounty, por lo que se debe poner la condición de terminado en alto y retornar el bounty. De no cumplir con el target la condición de terminado sigue en bajo y se retorna un bounty no válido.

Además si se encontró que no se cumple el target este bloque se encarga de generar un nuevo nonce para el bloque *Cargar datos*. Este nuevo nonce es simplemente sumarle 1 al nonce anterior. Si el byte ya está lleno, se le suma al siguiente y así sucesivamente hasta saturar los 4 bytes del nonce.

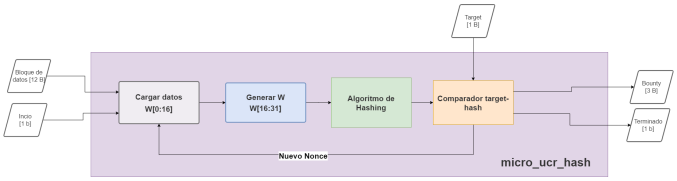

Figura 2. Diagrama de bloques para arquitectura optimizada en área.

II. Diseño optimizado en desempeño

Para optimizar el desempeño plantea paralelizar la búsqueda del bounty. Para esto es necesario modificar el bloque *cargar datos* por un *cargar datos_mod* y el bloque *comparador target-hash* por un *comparador target-hash_mod*. Además que agregar un nuevo bloque de control que esté pendiente cuál sub módulo termina para pasar el bounty a la salida. El detalle de estos nuevos bloques se puede observar en los Anexos 5 y 6.

El algoritmo para paralelizar la operación se explica a continuación. Imagine que se desean poner a trabajar 3 módulos en paralelo. Entonces el módulo 0 comienza con el nonce en [0x00, 0x00, 0x00, 0x00], el módulo 1 con el nonce [0x00, 0x00, 0x00, 0x01] y el módulo 2 con el nonce [0x00, 0x00, 0x00, 0x02]. Es decir cada módulo comienza con el nonce que corresponde a su índice de módulo. Luego cada módulo debe sumar el número de módulos para obtener el siguiente nonce. Así en cada iteración cada módulo revisará un nonce diferente a los otros y se puede mejorar el desempeño a la hora de encontrar el bounty. Un ejemplo para clarificar esto se encuentra en el Cuadro I.

Modulo	Nonce incial	Siguiente iteración	Siguiente iteración
0	Nonce	Nonce + 3	Nonce + 6
1	Nonce + 1	Nonce + 4	Nonce + 7
2	Nonce + 2	Nonce + 5	Nonce + 8

Cuadro I
Ejemplo del algoritmo para paralelizar la búsqueda del bounty.

II-A. Cargar datos_mod

Este bloque añade la funcionalidad de poder establecer en cuál nonce iniciar. Esto es importante debido a cómo se ha establecido la funcionalidad del módulo de desempeño. El resto de las funcionalidades se mantienen del módulo de área.

II-B. Comparador target-hash_mod

Se añade la funcionalidad de sumar el número de módulos al nonce en vez de sólo sumar 1. Esto para respetar la funcionalidad establecida. El resto de funcionalidades se mantienen. Esta suma es interna, por lo que no es modular. Es decir si se desean trabajar con 4 módulos, entonces siempre se suma 4. De querer añadir más módulos se puede añadir una entrada para determinar la suma.

II-C. Control de bounty

A este módulo se conectan todas las salidas de los módulos *Comparador target-hash_mod*. Este se encarga de determinar si alguno de los módulos alcanzó la condición de terminado para pasar a la salida el bounty dado por la entrada correspondiente y la señal de terminado.

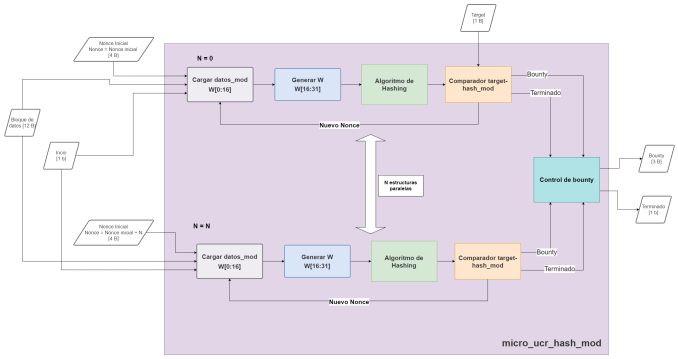

Figura 3. Diagrama de bloques para arquitectura optimizada en desempeño.

III. Resultados

Sólo se implementó en *python* el módulo optimizado en área. Un resultado se muestra a continuación.

```
Se ha encontrado un bounty!
Input:  [0x39, 0x7d, 0x9f, 0x2f, 0x40,
0xca, 0x9e, 0x6c, 0x6b, 0x1f, 0x33, 0x24,
0x3d, 0xff, 0xff, 0xff]
Target:  0xa
Hash:  [0x9, 0x9, 0x10]
Tiempo de ejecución: 96.559525
microsegundos

Segunda prueba: Prueba de inicio = False
Resultado:
Esperando inicio
```

Se observa que una vez que se encuentra el bounty la simulación se detiene, se informa de la entrada (con el nonce utilizado concatenado al final), del target y el hash (bounty) que se obtuvo bajo la premisa del target. Además incluí el tiempo de ejecución ya que es una las principales preocupaciones en este tipo de sistema.

Ahora se obtiene una "*prueba de trabajo* [2], que corresponde al nonce que se utilizó para encontrar el hash que cumplía con el target.

También incluí una pequeña segunda prueba donde se coloca la señal de inicio en bajo y el sistema simplemente retorna una advertencia de que aún no llega una señal de inicio. Esto realmente está demás y simplemente es para observar el comportamiento conductual del módulo.

IV. Aplicación: Criptomonedas

Este sistema permite encontrar pruebas de trabajo que pueden funcionar para crear nuevos bloques en una cadena

de bloques o *blockchain* los cuales se usan para validar transacciones de criptomonedas en una red de transacciones decentralizada como el *bitcoin* y así obtener ganancias cada vez que se genere la prueba de trabajo [2].

Entre los objetivos más importantes se tiene que entradas distintas nunca generen la misma salida y que la función de hash sea segura. Entre los objetivos de optimización más importante es que se puedan calcular los hashes rápidamente [1]. Para calcular más hashes también es importante entonces poder calcular varios al mismo tiempo.

Debido a una variedad de factores hoy en día hay una escasez de chips de silicio [3] lo cual dificulta la obtención y fabricación del hardware que se usa para calcular las pruebas de trabajo (minar criptomonedas), lo cual ha llevado a situaciones como la inflación de los precios de las tarjetas de vídeo para computadoras de escritorio [4], causando que los consumidores de estos productos no puedan encontrarlos en el mercado o tengan que pagar varias veces su precio original.

V. Conclusión

Esta primera parte del proyecto ha sido una gran herramienta de aprendizaje para lograr profundizar en el diseño de hardware y cubrir así la primera etapa del desarrollo.

El modelo implementado, en este caso el optimizado en área, funcionó como se esperaba, por lo que se espera que en la siguiente etapa del desarrollo se pueda implementar con menos o nulos problemas.

El modelo optimizado en área no se implementó por dificultades con programación paralela de alto nivel, pero espero haber presentado la idea de una manera clara.

Además este proyecto deja una parte importante sobre la información de las criptomonedas, ya que es una tecnología emergente con la cual puede que tengamos que convivir en el futuro cercano. Y creo que esta primera parte del proyecto deja claro el funcionamiento básico detrás de las criptomonedas.

Referencias

[1] *What is Hashing? Hash Functions Explained Simply*, ago. de 2018. dirección: https://www.youtube.com/watch?v=2BldESGZKB8.

[2] *But how does bitcoin actually work?* Jul. de 2017. dirección: https://www.youtube.com/watch?v=bBC-nXj3Ng4.

[3] *How a Chip Shortage Snarled Everything From Phones to Cars*. dirección: https://www.bloomberg.com/graphics/2021-semiconductors-chips-shortage/.

[4] V. Vicente, *Why Is It So Hard to Buy a Graphics Card in 2021?* Mayo de 2021. dirección: https://www.howtogeek.com/726236/why-is-it-so-hard-to-buy-a-graphics-card-in-2021/.

APÉNDICE A
DETALLE DE MICRO_HASH_UCR

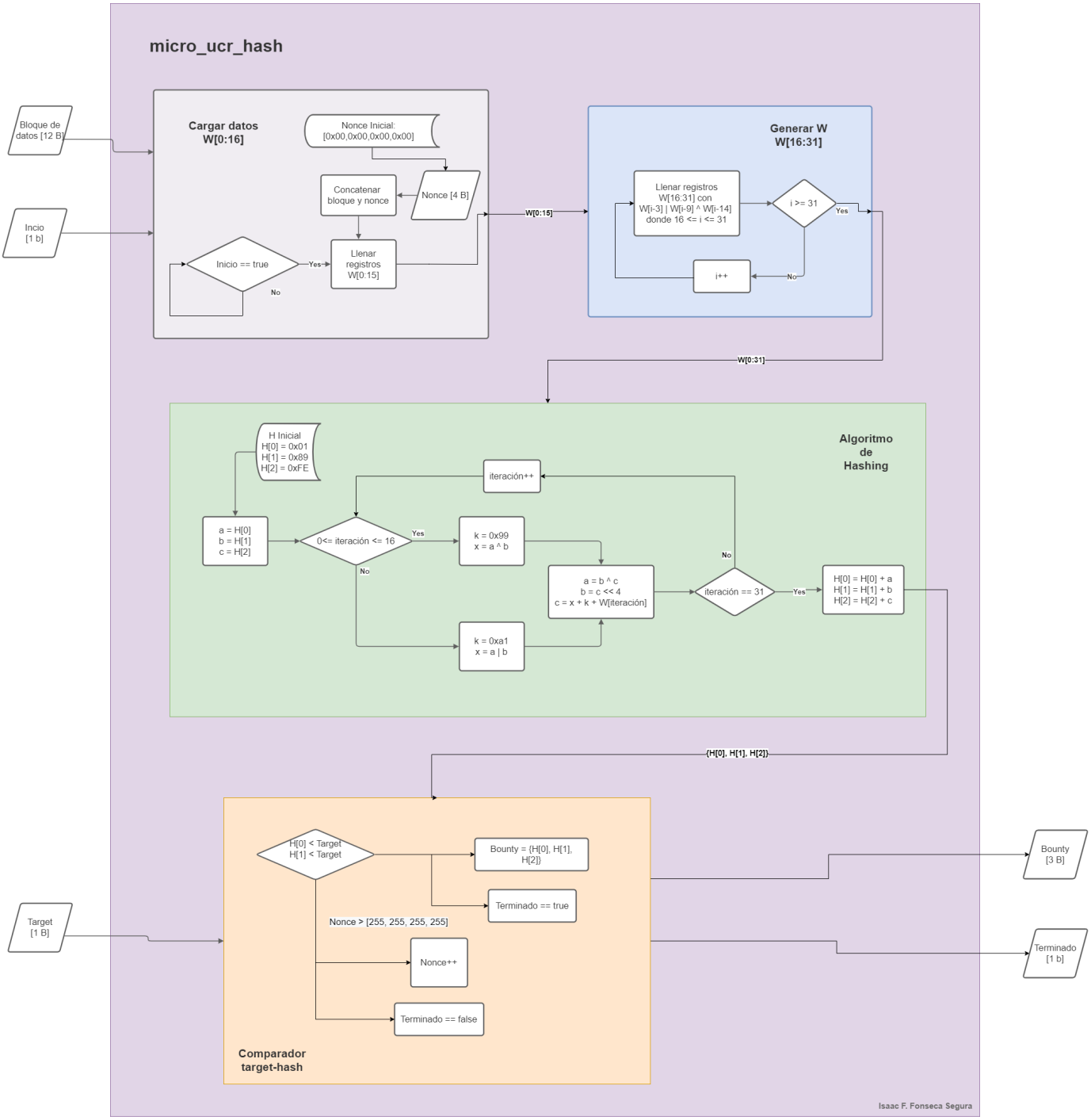

Figura 4. Especificación a nivel de algoritmo de los bloques del diagrama de bloques optimizado para área.

APÉNDICE B
DETALLE DE MICRO_HASH_UCR_MOD

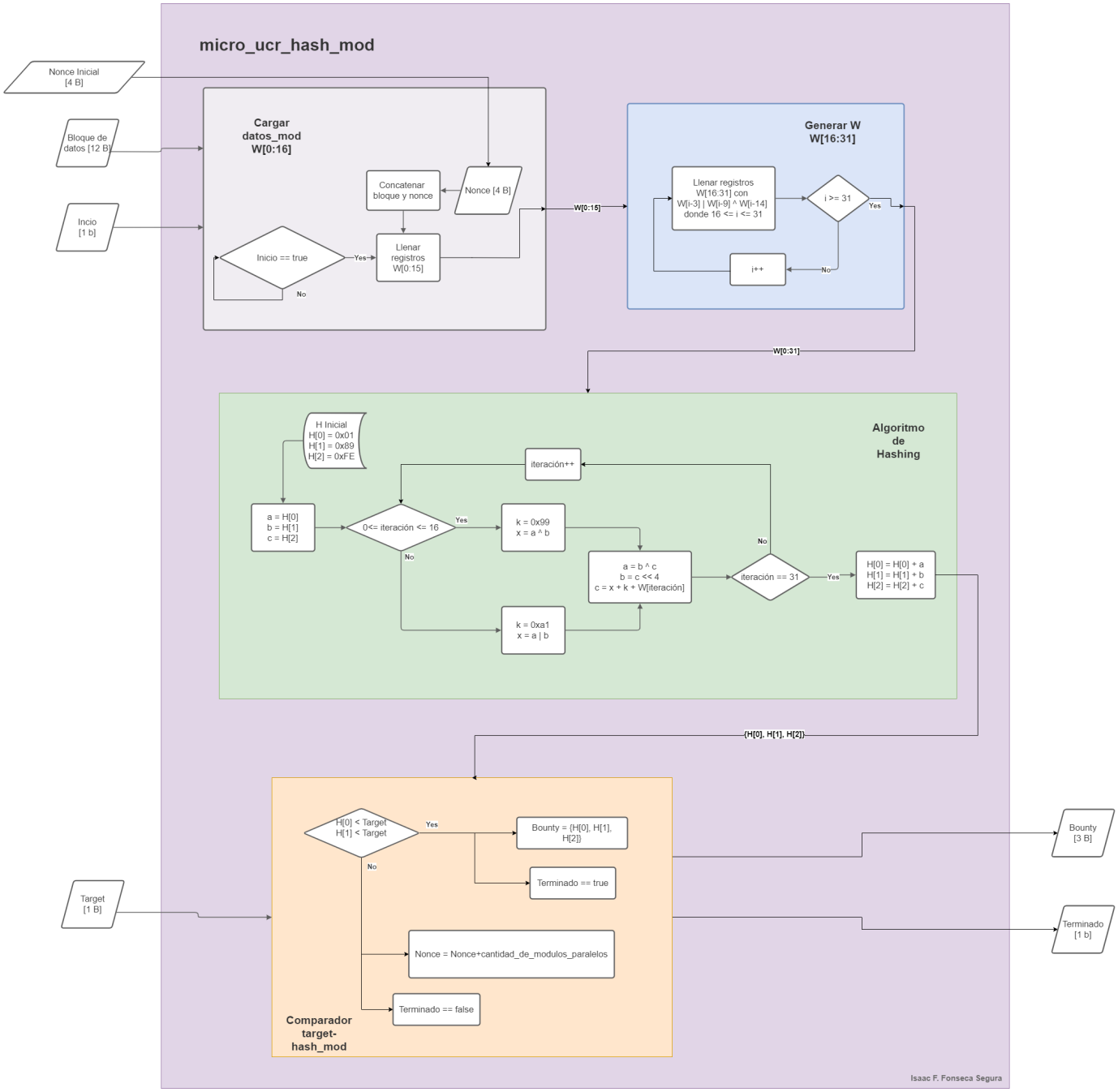

Figura 5. Especificación a nivel de algoritmo de los bloques del diagrama de bloques optimizado para desempeño.

APÉNDICE C
DETALLE DE CONTROL DE BOUNTY

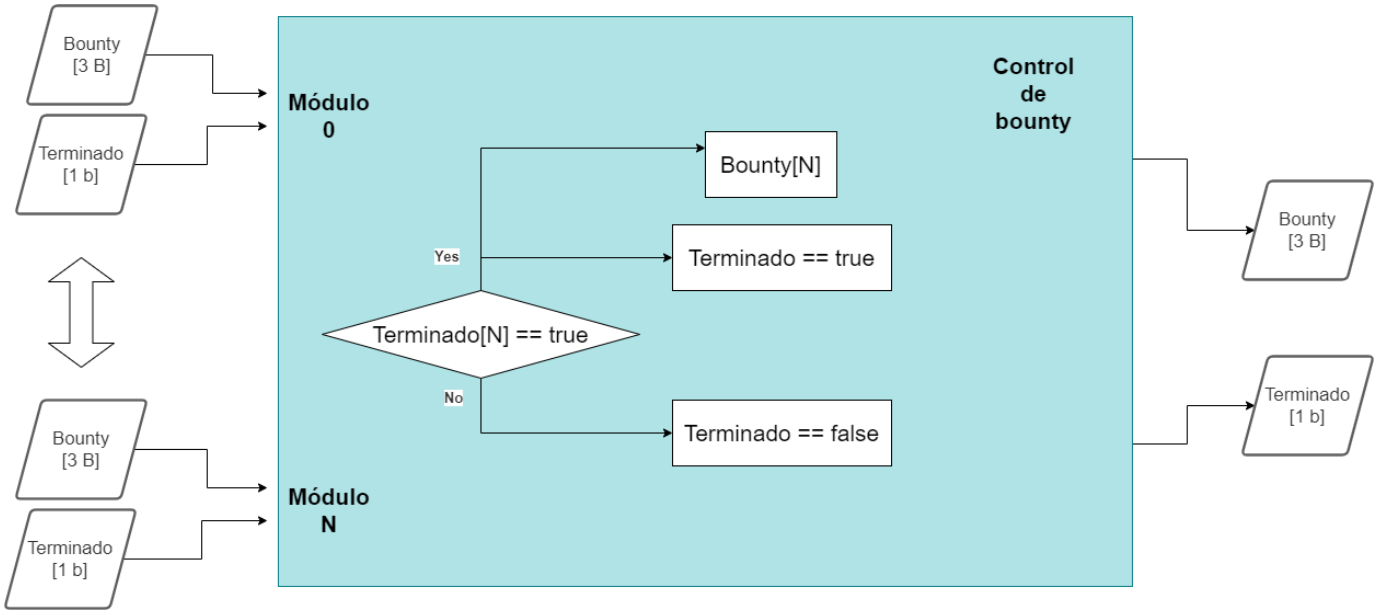

Figura 6. Especificación a nivel de algoritmo del bloque de control de bounty para el diseño optimizado en desempeño.