UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Ciencias e Ingenierías

Diseño de un procesador RISC-V con soporte especializado para algoritmos de hash

Andrés Sebastián Orozco Chuquín

Ingeniería en Electrónica y Automatización

Trabajo de fin de carrera presentado como requisito
para la obtención del título de
Ingeniero Electrónico

Quito, 13 de mayo de 2025

UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Ciencias e Ingenierías

HOJA DE CALIFICACIÓN
DE TRABAJO DE FIN DE CARRERA

Diseño de un procesador RISC-V con soporte especializado para algoritmos de hash

Andrés Sebastián Orozco Chuquín

Nombre del profesor, Título académico **Eduardo Holguín, PhD**

Quito, 13 de mayo de 2025

© DERECHOS DE AUTOR

Nombres y apellidos: Andrés Sebastián Orozco Chuquín

Código: 00320595

Cédula de identidad: 1003730262

Lugar y fecha: Quito, 13 de mayo de 2025

ACLARACIÓN PARA PUBLICACIÓN

Nota: El presente trabajo, en su totalidad o cualquiera de sus partes, no debe ser considerado como una publicación, incluso a pesar de estar disponible sin restricciones a través de un repositorio institucional. Esta declaración se alinea con las prácticas y recomendaciones presentadas por el Committee on Publication Ethics COPE descritas por Barbour et al. (2017) Discussion document on best practice for issues around theses publishing, disponible en http://bit.ly/COPETheses.

UNPUBLISHED DOCUMENT

Note: The following capstone project is available through Universidad San Francisco de Quito USFQ institutional repository. Nonetheless, this project – in whole or in part – should not be considered a publication. This statement follows the recommendations presented by the Committee on Publication Ethics COPE described by Barbour et al. (2017) Discussion document on best practice for issues around theses publishing available on http://bit.ly/COPETheses.

RESUMEN

Este trabajo presenta el diseño y la implementación de un procesador RISC-V de ciclo único ampliado con soporte especializado para algoritmos hash, centrándose en el algoritmo MD5. La investigación se fundamenta en la creciente demanda de sistemas embebidos con capacidades criptográficas que optimicen tanto el rendimiento como la seguridad en entornos de recursos limitados. Se desarrolló una microarquitectura completa basada en el conjunto de instrucciones RV32I, incorporando instrucciones personalizadas para operaciones hash, lectura de memoria externa y rotación (ROL). El algoritmo MD5 se implementó a través de un módulo especializado compuesto por una unidad de relleno/memoria y un controlador de máquina de estados finitos (FSM). La validación se llevó a cabo mediante simulación en Vivado e implementación hardware real en una FPGA Zybo Z7, logrando la verificación funcional, el análisis del uso de recursos, la medición del consumo de energía y el análisis de temporización. El análisis realizado confirma la factibilidad y eficiencia del diseño para aplicaciones criptográficas en sistemas embebidos, estableciendo un fundamento sólido para futuras mejoras. Se prevé la incorporación de funciones hash más resistentes, con la posibilidad de implementar estructuras de hashes anidados para fortalecer la seguridad y optimizar el rendimiento.

Palabras clave: RISC-V, RV32I, MD5, procesador embebido, hash en hardware, extensión ISA, funciones criptográficas.

ABSTRACT

This work presents the design and implementation of an extended single-cycle RISC-V processor with specialized support for hash algorithms, focusing on the MD5 algorithm. The research is based on the growing demand for embedded systems with cryptographic capabilities that optimize both performance and security in resource-constrained environments. A complete microarchitecture was developed based on the RV32I instruction set, incorporating custom instructions for hash operations, external memory reading and rotation (ROL). The MD5 algorithm was implemented through a specialized module consisting of a padding/memory unit and a finite state machine (FSM) controller. Validation was carried out through simulation in Vivado and real hardware implementation on a Zybo Z7 FPGA, achieving functional verification, resource usage analysis, power consumption measurement and timing analysis. The analysis performed confirms the feasibility and efficiency of the design for cryptographic applications in embedded systems, establishing a solid foundation for future improvements. The incorporation of more robust hash functions is envisioned, with the possibility of implementing nested hash structures to strengthen security and optimize performance.

Key words: RISC-V, RV32I, MD5, embedded processor, hardware hashing, ISA extension, cryptographic functions.

TABLA DE CONTENIDO

Introducción **10**

Marco Teórico **12**

Arquitectura RISC-V **12**

Extensión RV32I Registros 14

Extension RV32I ISA 15

Algoritmo de hash **16**

Hash MD5 17

Padding 17

Instancias preliminares 18

Flujo del algoritmo 19

Implementación de la Arquitectura RV32I **22**

Diseño de la microarquitectura RISC-V RV32I **22**

Unidad de Control 24

Módulo Main Decoder 25

Módulo ALU Decoder 26

Extensión del conjuto de instrucciones **27**

Instrucción para lectura desde una ROM externa 27

Instrucción ROL (Rotate on Left) 28

Instrucción para realización de hash 29

Implementación del Algoritmo MD5 ..... **30**

Módulo Padding/Memory **30**

Módulo FSM **31**

Resultados **33**

Simulación en Vivado **33**

Implementación en FPGA **35**

Recursos utilizados de la FPGA 36

Consumo de potencia 37

Análisis de tiempo 38

Conclusiones **39**

Referencias bibliográficas **41**

Anexo A: Estructura de instrucciones RV32I **42**

Anexo B: Tabla de constantes del algoritmo MD5 ..... **43**

Anexo C: Diagramas de flujo del procesador **44**

ÍNDICE DE TABLAS

Tabla 1. Conjunto de registros RISC-V [5] 14
Tabla 2. Inicialización de registros 19
Tabla 3. Parámetros por ronda del MD5 21
Tabla 4. Valores de rotación circular 21
Tabla 5. Tabla de verdad del Main Decoder 26
Tabla 6. Tabla de verdad del ALU Decoder 27
Tabla 7. Modificación de la tabla ALU Decoder - ROL 28

ÍNDICE DE FIGURAS

Figura 1. Extensiones RV32IMAC....14
Figura 2. Formato de instrucciones RV32I [8]. 15
Figura 3. Generación del mesaje usado en MD5 18
Figura 4. Flujo de procesamiento del hash MD5....20
Figura 5. Diagrama de bloques de un procesador RV32I (single-cycle)....23
Figura 6. Diagrama de bloques del Controlador....25
Figura 7. Diagrama de bloques modificado para una ROM externa 28
Figura 8. Instrucción tipo H....29
Figura 9. Diagrama de bloques de la extensión Hash....30
Figura 10. Diagrama de flujo de la FSM 32
Figura 11. Resultado del Hash - MD5 Hash Generator....34
Figura 12. Resultado del Hash - Simulación en Vivado....34
Figura 13. Bloques IP - Implementación del procesador....35
Figura 14. Simulación en implementación -ILA 36
Figura 15. Recursos de la FPGA utilizados - Implementación....37
Figura 16. Consumo de potencia - Implementación 38
Figura 17. Análisis de tiempo - Implementación....38

INTRODUCCIÓN

En la era de la Internet de las Cosas (IoT) y los sistemas embebidos, la criptografía en hardware resulta esencial para garantizar la seguridad de los dispositivos conectados. La proliferación de sistemas en red ha convertido la protección de datos en una prioridad frente a la creciente amenaza de ataques cibernéticos[11]. En este contexto, implementar funciones criptográficas críticas directamente en hardware aumenta significativamente la robustez del sistema. Kocher *et al.* (1999) demostraron que la ejecución de algoritmos criptográficos en software puede filtrar información a través del análisis de consumo eléctrico, revelando claves privadas[7]. En contraste, las implementaciones hardware dedicadas reducen estas vulnerabilidades: Stallings (2017) señala que las variaciones de consumo en hardware criptográfico son tan pequeñas que el análisis de potencia simple (SPA) difícilmente compromete la información confidencial[10].

La elección de RISC-V como plataforma base se fundamenta en sus ventajas inherentes. RISC-V es una arquitectura con un Instruction Set Architecure (ISA) abierta y libre de regalías [9]. Además, RISC-V está diseñado de forma modular,lo que significa que se puede personalizar el procesador agregando sólo las extensiones necesarias y manteniendo la simplicidad cuando no lo requieran. La extensión RV32I por sí sola permite un compilador completo, y los subtotales de instrucciones opcionales facilitan cumplir requerimientos específicos sin penalizar el diseño base[3].

Esta tesis propone diseñar una extensión del procesador RISC-V RV32I que integre soporte especializado para cálculos hash (en particular, MD5) sin recurrir a coprocesadores criptográficos externos. En él, se implementa un mecanismo de excepción/trampa personalizado que permite desviar la ejecución hacia una rutina de hashing predefinida. Es decir, cuando el procesador ejecuta una instrucción especial de disparo (trigger instruction),

se interrumpe el flujo normal de ejecución y se guarda el estado de la CPU (program counter y registros) en memoria interna [12]. Tras completar el cómputo hash, el sistema restaura el estado de los registros salvados y retoma la ejecución original desde donde se había detenido.

De esta forma, la rutina MD5 queda “incrustada” en el procesador de forma transparente. En esencia, es una integración hardware/software que mejora la seguridad: como el hash se calcula en el propio core de manera aislada durante la trampa, se minimizan los riesgos de filtraciones laterales comparado con una implementación puramente en software corriente.

La investigación integra perspectivas internacionales y necesidades locales. Globalmente, la arquitectura RISC-V gana relevancia en aplicaciones seguras y eficientes energéticamente (IoT), con extensiones criptográficas recientemente estandarizadas para optimizar funciones hash y cifrado [9]. En el contexto latinoamericano y ecuatoriano, crece el interés por sistemas embebidos seguros y de bajo consumo, impulsado por iniciativas académicas e industriales en IoT y ciberseguridad. Instituciones como ESPOL contribuyen con investigaciones relevantes en estos campos [11]. Así, el desarrollo de un procesador RISC-V con soporte nativo para funciones hash como MD5 y capacidad de integrar extensiones futuras satisface tanto las tendencias tecnológicas globales como los requerimientos específicos de seguridad y eficiencia en Ecuador.

MARCO TEÓRICO

El desarrollo de un procesador dedicado a una aplicación específica tal y como es la rama de la criptografía conocida como hash, requiere una comprensión de los fundamentos teóricos que permite el diseño de la arquitectura y el algoritmo involucrado en hardware. Durante este capítulo, se presentan conceptos clave que permitan contextualizar las decisiones de implementación tomadas durante la realización del proyecto, incluyento la arquitectura RISC-V, el algoritmo de hash MD5, las características del procesador de un solo ciclo y las ventajas en el diseño de microarquitectura personalizadas para realizar la aplicación específica dentro de sistemas embebidos.

Arquitectura RISC-V

Los procesadores Reduce Instruction Set Computer (RISC) presentan una dualidad interesante al ser simples y complejos a la vez. Su simplicidad se debe a la estructura del diseño, centrada en un conjunto reducido de intrucciones, lo que implica menos intrucciones especiales a comparación de otras variantes como las arquitecturas Complex Instruction Set Computing (CISC). Sin embargo, la complejidad en la microarquitectura radica en la optimización del uso de recursos limitados, garantizando un rendimiento eficiente y permitiendo la expansión del conjunto de instrucciones sin comprometer la integridad del procesador.

RISC-V es una arquitectura que nació en 2010 en la Universidad de California, Berkeley, como un proyecto de investigación liderado por Krste Asanović, Yunsup Lee y Andrew Waterman, con la colaboración de David Patterson. En 2015, RISC-V se consolidó como un estándar dentro de la industria del hardware y el software, bajo el respaldo de la organización RISC-V International [9].

El diseño de RISC-V se sustenta en dos principios fundamentales: apertura y flexibilidad. A diferencia de otras arquitecturas con instrucciones complejas y poco modificables, RISC-V se organiza de forma modular. Su ISA se compone de un conjunto base mínimo y diversas extensiones opcionales. Esta estructura le permite adaptarse tanto a aplicaciones simples, como en la educación o sistemas embebidos, como a escenarios exigentes, como los entornos de alto rendimiento y multiproceso[13]. Para lograr una colaboración global, acelerar la innovación y obtener un alto impacto, este proyecto se constituyó con una ISA de código abierto, lo que significa que su especificación está disponible públicamente, permitiendo su uso, modificación y distribución sin necesidad de licencias ni pagos de regalías. Permitiendo que desarrolladores de todo el mundo pueden contribuir activamente a su evolución y adaptarla a diferentes necesidades tecnológicas.

Esta arquitectura ha incorporado diversos espacios de direcciones, incluyendo versiones de 32 bits, 64 bits y una adaptación reciente de 128 bits. Sin embargo, es importante destacar que la variante de 32 bits continúa siendo la más utilizada debido a las tendencias tecnológicas actuales. Dentro de este contexto, existen varias extensiones que forman parte del esquema modular de la arquitectura, conocidas como RV32IMAC. La extensión M incorpora operaciones específicas para multiplicación y división de números enteros, optimizando el rendimiento en cálculos complejos; la extensión A proporciona soporte para operaciones atómicas, facilitando la sincronización en sistemas multiprocesador; la extensión C implementa instrucciones comprimidas que mejoran la eficiencia del flujo de ejecución; y la extensión I constituye el conjunto de instrucciones fundamentales necesario para cualquier implementación de RISC-V.

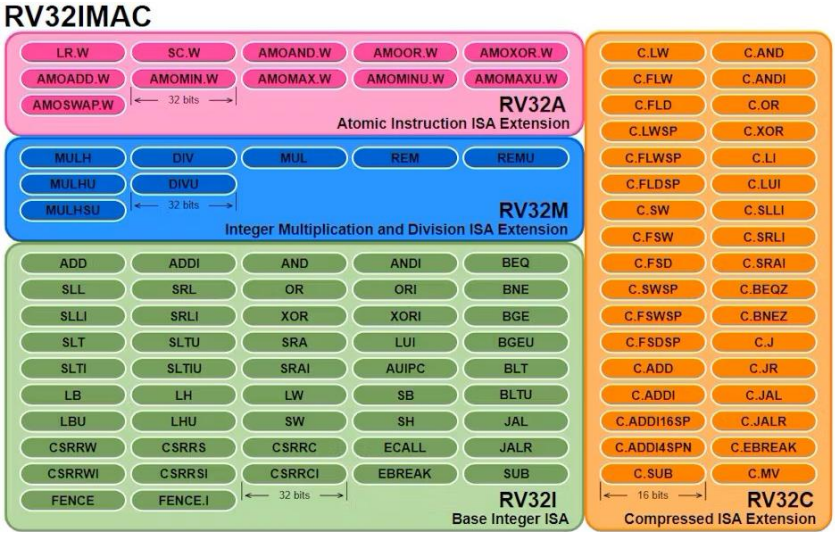

Figura 1. Extensiones RV32IMAC

Extensión RV32I Registros

El diseño de la extensión base de 32 bits incluye 37 instrucciones esenciales que permitan cumplir con requisitos de los sistemas operativos modernos. Al igual que en numerosas arquitecturas, las instrucciones se clasifican por procesamiento de datos, control de flujo y acceso a memoria. Por otro lado, al ser una arquitectura del tipo load-store, las operaciones realizadas por la unidad aritmética solo trabajan sobre el banco de registros, mientras existen instrucciones específicas que gestionan el intercambio de información con la memoria.

Registro	**Nombre**	**Descripción**
x0	zero	Valor constante 0
x1	ra	Dirección de retorno
x2	sp	Puntero de pila
x3	gp	Puntero global
x4	tp	Puntero de hilo
x5-7	t0-2	Registros temporales
x8	s0/fp	Registro guardado/Puntero de marco
x9	s1	Registro guardado
x10-11	a0-1	Argumentos de función/Valores de retorno
x12-17	a2-7	Argumentos de función
x18-27	s2-11	Registros guardados
x28-31	t3-6	Registros temporales

Tabla 1. Conjunto de registros RISC-V [5]

El conjunto de registros de RISC-V, detallado en la Tabla 1, comprende 32 registros de 32 bits (x0–x31), donde x0 (zero) está reservado para almacenar permanentemente el valor 0, descartando cualquier intento de escritura. Estos registros siguen una organización convencional: los registros s0–s11 (x8–x9, x18–x27) y t0–t6 (x5–x7, x28–x31) se destinan al almacenamiento de variables temporales, mientras que ra (x1) y a0–a7 (x10–x17) desempeñan funciones específicas durante las llamadas a procedimientos. Adicionalmente, los registros sp (x2), gp (x3) y tp (x4) cumplen roles especializados para la gestión de la pila, punteros globales y control de hilos, respectivamente. En su totalidad, el estado arquitectónico visible para el usuario constituye 1024 bits.

Extension RV32I ISA

La arquitectura RV32I define seis formatos de instrucciones, representados en la Figura 2, cada uno con un propósito específico. El formato R se usa exclusivamente para operaciones entre registros, mientras que el formato I abarca valores inmediatos, saltos incondicionales y cargas desde memoria. Para el almacenamiento, se emplea el formato S, complementado por el formato B para gestionar saltos condicionales. Finalmente, el formato U optimiza el manejo de valores inmediatos extensos y el formato J facilita la implementación eficiente de saltos incondicionales[8].

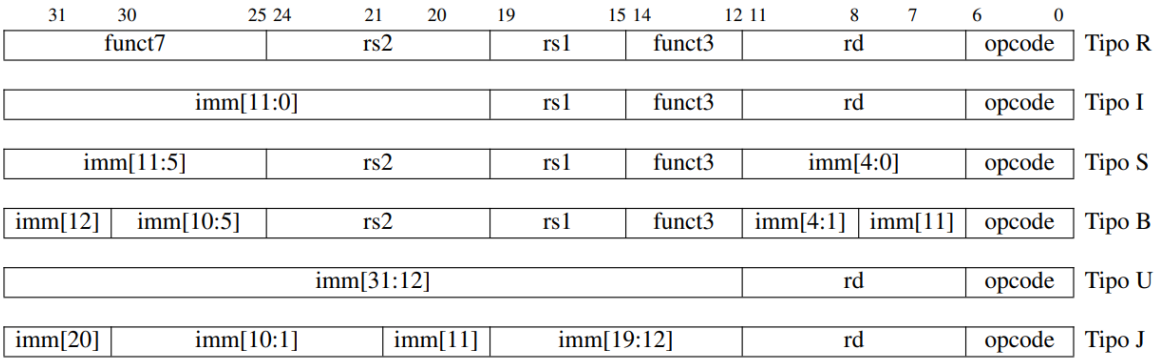

Figura 2. Formato de instrucciones RV32I [8].

Para complementar la descripción de los formatos de instrucciones, el Anexo A: presenta un mapa detallado con los opcodes utilizados en la arquitectura RV32I, omitiendo

las instrucciones de sistema. Esta ilustración muestra la estructura específica de cada instrucción, definiendo claramente el tipo de palabra asociado y el nombre correspondiente.

Algoritmo de hash

Las funciones de hash criptográficas son esenciales en la seguridad informática, ya que convierten datos de tamaño variable en un resumen o hash de longitud fija, garantizando la autenticidad e integridad. Estas funciones operan mediante funciones de carácter iterativo que aplican una función de compresión. Dentro de las propiedades se encuentra la unidireccionalidad, lo que significa que es inviable recuperar el mensaje original a partir del hash y la resistencia a colisiones, lo que dificulta encontrar dos mensajes distintos con el mismo hash. De esta manera, es posible detectar cualquier alteración en los datos pues con el mínimo cambio, el hash resultante será completamente diferente[10].

Dentro de este ámbito se ofrecen múltiples aplicaciones en el ámbito de la seguridad informática, destacando su uso en códigos de autenticación de mensajes (MAC), generación de firmas digitales y producción de valores pseudoaleatorios (PRNG) o claves simétricas. No obstante, una de sus implementaciones más críticas se encuentra en el almacenamiento seguro de contraseñas que constituye una defensa primaria contra ciberataques. Durante el proceso de autenticación, el sistema simplemente compara los valores hash para verificar la legitimidad del usuario, sin necesidad de almacenar o recuperar la contraseña original en texto plano. Esta técnica, conocida como "one-way hashing", garantiza que incluso si la base de datos de credenciales fuera comprometida, el atacante no podría determinar fácilmente las contraseñas originales debido a la irreversibilidad matemática del proceso hash. Este marco de aplicación fundamenta precisamente el propósito del presente proyecto: la implementación hardware de un algoritmo hash utilizando los recursos proporcionados por la arquitectura del

procesador RISC-V, buscando optimizar tanto el rendimiento como la seguridad en sistemas embebidos donde los recursos computacionales son limitados.

Hash MD5

El MD5 (Message-Digest Algorithm 5), desarrollado por Ronald Rivest en 1991, es un algoritmo de hash criptográfico que genera resúmenes de 128 bits mediante un proceso iterativo que divide mensajes en bloques de 512 bits y aplica operaciones lógicas no lineales. Aunque fue ampliamente implementado, actualmente se considera inseguro para aplicaciones críticas debido a vulnerabilidades demostradas en ataques de colisión que permiten generar diferentes mensajes con idénticos valores hash con un esfuerzo computacional de 2^{64}, factible con la tecnología moderna [10].

Aunque MD5 ya no se recomienda para aplicaciones críticas de seguridad debido a sus vulnerabilidades conocidas, su amplia difusión histórica y eficiencia computacional lo mantienen vigente en escenarios de integridad no críticos, como la verificación de descargas de archivos, entre otras aplicaciones.

Padding

El algoritmo MD5 inicia su procesamiento recibiendo un mensaje de entrada con longitud K arbitraria. Sin embargo, para mantener la uniformidad en el procesamiento, cada bloque operativo requiere exactamente 512 bits. Para adaptarse a esta exigencia estructural, se implementa un procedimiento conocido como "padding" o relleno. Este mecanismo de relleno está diseñado meticulosamente para garantizar que la longitud resultante del mensaje sea congruente con 488 módulo 512. Esta especificación reserva estratégicamente los últimos 64 bits de cada bloque para almacenar la información referente a la longitud original del mensaje [6].

Como se ilustra en la Figura 3, el proceso de preparación del mensaje comienza con la transferencia exacta bit a bit del mensaje original, preservando íntegramente sus K bits iniciales. Posteriormente, se introduce estratégicamente un único bit con valor 1, seguido por una secuencia de bits 0 que se extiende hasta alcanzar precisamente 448 bits de longitud total. Esta estructura reserva un segmento final de 64 bits con propósito específico: los primeros 32 bits contienen la representación numérica de la longitud original del mensaje, mientras que los 32 bits restantes se establecen uniformemente a cero. Esta organización culmina en la formación de un bloque completo de 512 bits, que posteriormente se fragmentará en 16 unidades procesables de 32 bits cada una.

Un aspecto técnico fundamental a considerar es la arquitectura de ordenamiento de bytes predominante en los sistemas computacionales modernos. Dado que la mayoría de computadoras implementa el formato little-endian para la organización de datos en memoria, el mensaje final también adoptará esta convención, donde el byte menos significativo ocupa la posición de memoria más baja, garantizando así la compatibilidad con las operaciones subsecuentes del algoritmo.

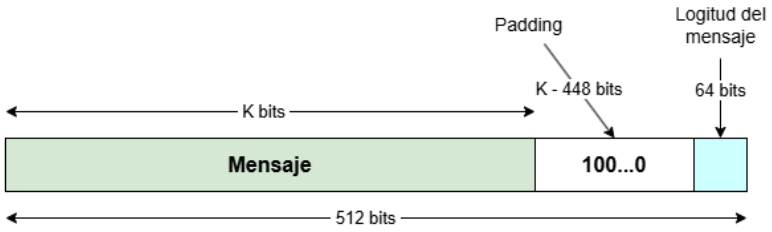

Figura 3. Generación del mesaje usado en MD5

Instancias preliminares

MD5 opera fundamentalmente sobre un conjunto de cuatro buffers, denominados A, B, C y D de 32 bits de longitud cada uno. Estos registros temporales funcionan como acumuladores durante todo el proceso iterativo y se encuentran previamente inicializados en hexadecimal de la manera que se encuentran en la Tabla 2.

Palabra A	67 45 23 01
Palabra B	ef cd ab 89
Palabra C	98 ba dc fe
Palabra D	10 32 54 76

Tabla 2. Inicialización de registros

Con el objetivo de cumplir con la no linealidad del algoritmo, durante una parte del proceso es necesario obtener un valor que proviene de una función matemática tracendental como es el seno, de la siguiente manera:

$$K[i] = 2^{32} \cdot |\sin(i+1)| \quad (1)$$

No obstante, el cálculo de este parámetro puede resultar extenso, ya que debe realizarse en cada iteración. Por ello, se ha definido una tabla de constantes K que contiene todos los valores previamente calculados. Estos valores se encuentran disponibles en el Anexo B.

Flujo del algoritmo

El procesamiento del mensaje de 512 bits generados por el padding es donde encuentra el enfoque las funciones criptográficas. Cada uno de los 16 bloques de 32 bits, identificados secuencialmente desde M_0 hasta M_{15}, se somete a tratamiento individualizado mediante una sofisticada combinación de operaciones aritméticas y transformaciones lógicas. Este procedimiento se estructura en 64 iteraciones meticulosamente organizadas en cuatro secciones simétricas de 16 iteraciones cada una. Esta arquitectura algorítmica no es arbitraria, sino deliberadamente diseñada para maximizar la dispersión criptográfica necesaria en la generación del hash resultante [1].

Tras cada ciclo de operaciones, se efectúa una actualización estratégica de variables temporales que replican el estado actualizado de los registros principales. Siguiendo el esquema ilustrado en la Figura 4, se implementan las asignaciones B = C', C = D' y D = A', donde la notación prima (') indica el valor actualizado del registro correspondiente. Sin embargo, el tratamiento de B' presenta una particularidad significativa: a esta variable se le

asigna específicamente el resultado procedente de un bloque combinatorio complejo que implementa la función matemática descrita en la ecuación 2, introduciendo así un elemento adicional de no linealidad en el sistema.

$$B' = B + left\_rotate(A + Funct(B, C, D) + M[j] + T[j], shift) \qquad (2)$$

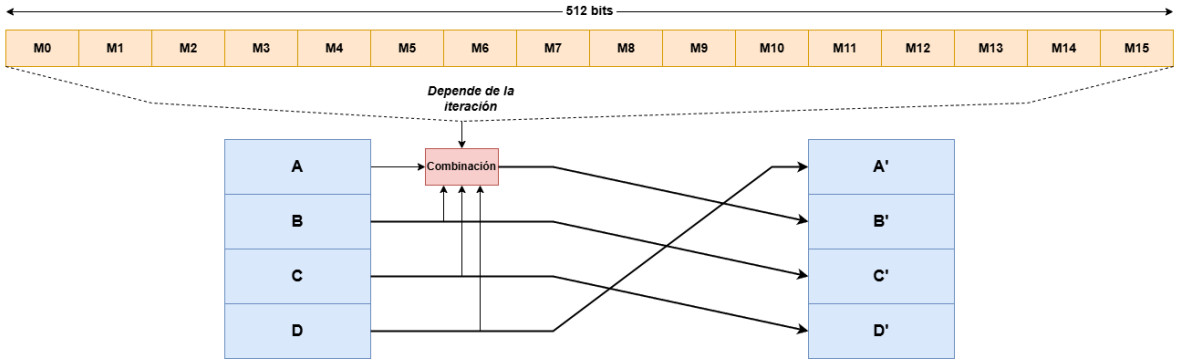

Figura 4. Flujo de procesamiento del hash MD5

El comportamiento del registro B' constituye un elemento determinante en el algoritmo MD5, caracterizado por una dependencia estructurada de cuatro componentes variables que evolucionan metódicamente durante la ejecución. Primero, el algoritmo alterna secuencialmente entre cuatro funciones distintivas (F, G, H e I) que están descritas en las funciones 3, 4, 5 y 6 respectivamente, cada una implementando diferentes operaciones lógicas donde &, |, ~ y ^ representan una compuerta and, or, not y xor respectivamente.

Segundo, la indexación del bloque de mensaje M(i) sigue un patrón determinístico que varía según la ronda de procesamiento en curso. Tercero, se incorporan valores constantes predefinidos T(i) que aportan entropía adicional al proceso. Finalmente, la magnitud del desplazamiento circular (shift) aplicado a los resultados intermedios se modifica estratégicamente dependiendo tanto de la función activa como de la iteración actual. La Tabla 3 proporciona una visualización estructurada de estas interdependencias, detallando metódicamente cómo cada parámetro evoluciona a través de las diferentes rondas e iteraciones.

$$F(B,C,D) \rightarrow (B\&C)|\big((\sim B\&D)\big) \quad (3)$$

$$G(B,C,D) \rightarrow (D\&B)|\big((\sim D)\&C\big) \quad (4)$$

$$H(B,C,D) \rightarrow B\hat{}C\hat{}D \quad (5)$$

$$I(B,C,D) \rightarrow C\hat{}\big(B\big|(\sim D)\big) \quad (6)$$

	Función	$\boldsymbol{M[i]}$	$\boldsymbol{T[i]}$	$\boldsymbol{Shift}$
$0 \leq i < 16$	F(B,C,D)	i	i	$sh\_F(i\%4)$
$16 \leq i < 32$	G(B,C,D)	$(5i+1)\%16$		$sh\_G(i\%4)$
$32 \leq i < 48$	H(B,C,D)	$(3i+5)\%16$		$sh\_H(i\%4)$
$48 \leq i < 64$	I(B,C,D)	$7i\%16$		$sh\_i(i\%4)$

Tabla 3. Parámetros por ronda del MD5

Los valores de desplazamiento circulares están organizados en cuatro palabras de 32 bits cada una correspondientes a las funciones criptográficas. La operación módulo 4 sobre el número de iteración produce un resultado entre 0 y 3, que actúa como índice para seleccionar precisamente el byte que almacena el valor de desplazamiento en cada palabra. De esta manera, obtenemos 16 magnitudes de rotación para las 64 iteraciones del proceso. En la Tabla 4 se muestra las palabras con los valores correspondientes en hexadecimal.

Shift F	16 11 0C 07
Shift G	14 0E 09 05
Shift H	17 10 0B 04
Shift I	15 0F 0A 06

Tabla 4. Valores de rotación circular

Finalmente, tras completar las 64 iteraciones del algoritmo, se realiza una operación de acumulación: los cuatro registros temporales (A, B, C, D) son sumados a sus respectivos valores iniciales. La concatenación secuencial de los cuatro registros de 32 bits genera el hash final de 128 bits, que constituye a la una huella única del mensaje procesado.

IMPLEMENTACIÓN DE LA ARQUITECTURA RV32I

La microarquitectura RV32I constituye el núcleo funcional sobre el cual se construye la estructura lógica completa del procesador. Esta sección detalla un diseño rigurosamente alineado con las especificaciones del conjunto de instrucciones (ISA) presentadas en el manual de referencia desarrollado por Walterman y colaboradores [12]. Adicionalmente, se describen las extensiones específicas requeridas para la implementación eficiente del algoritmo MD5, así como la metodología de integración que garantiza su correcto acoplamiento con la infraestructura base de la microarquitectura previamente establecida.

Diseño de la microarquitectura RISC-V RV32I

La arquitectura RISC-V destaca en el panorama computacional actual por tres características fundamentales: su simplicidad conceptual, su estructura modular y su notable eficiencia operativa, todo ello respaldado por una documentación técnica exhaustiva y accesible. La implementación microarquitectónica desarrollada en este trabajo adopta específicamente el modelo de ciclo único (single-cycle), donde cada instrucción completa su ejecución dentro de un único ciclo de reloj, procediendo a actualizar registros o posiciones de memoria en el ciclo inmediatamente posterior. Esta organización temporal establece una relación directa y optimizada con el flujo de datos del sistema, permitiendo un control preciso de las dependencias entre instrucciones y aumenta la certeza en el orden de ejecución.

El esquema de ciclo único se articula a través de un datapath integrando diversas unidades hardware especializadas que controlan el flujo de datos durante la ejecución del programa. En este entorno convergen múltiples categorías de señales de procesamiento de información: códigos de operación, operandos, contenidos, direcciones de memoria, destinos de salto y constantes que aportan valores predefinidos al sistema. Todos estos elementos informativos son sistemáticamente manipulados, encaminados y seleccionados mediante un

conjunto de señales de control, generado por un módulo controlador, que establecen la secuencia precisa de operaciones para cada instrucción procesada[4].

La implementación incorpora todos los componentes fundamentales necesarios para un procesador plenamente operativo: banco de registros, Unidad Aritmético-Lógica (ALU), unidad de control principal, memorias de instrucciones y datos, circuitos extensores, contador de programa (PC), y módulos auxiliares como multiplexores y sumadores. La arquitectura se ha diseñado con una rigurosa modularización jerárquica que facilita tanto la futura expansión del conjunto de instrucciones como el diagnóstico de errores durante el desarrollo. La Figura 5 ilustra esquemáticamente la estructura fundamental del procesador base con sus interconexiones críticas.

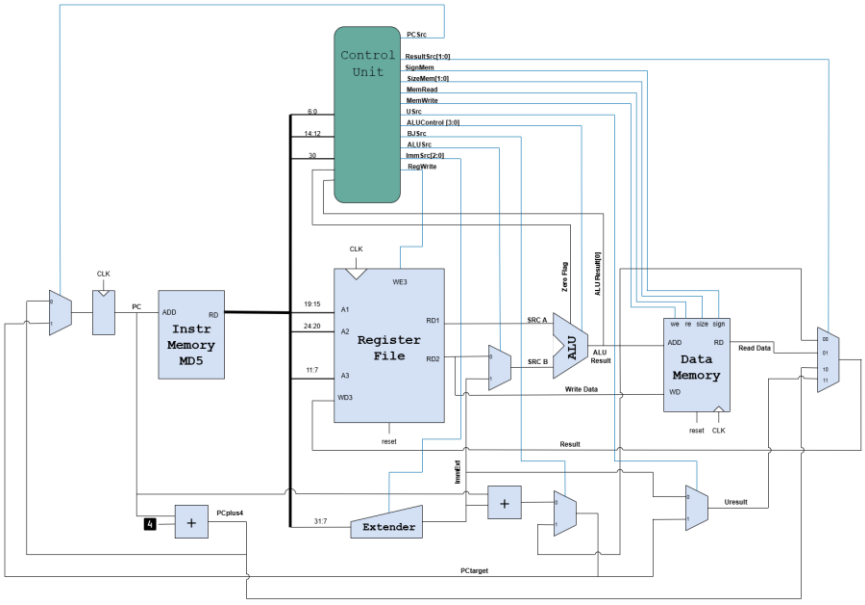

Figura 5. Diagrama de bloques de un procesador RV32I (single-cycle)

El procesador está diseñado para soportar seis tipos de instrucciones, los cuales se muestran en la Figura 2. En este contexto, la distribución del hardware debe ser capaz de manejar el flujo de datos correspondiente a cada tipo. En el apartado Anexo C, se incluyen imágenes que ilustran el flujo de datos para cada una de estas instrucciones.

Unidad de Control

El módulo que organiza y dirige el funcionamiento del procesador es la unidad de control, la cual se encarga de gobernar todas las operaciones computacionales mediante la generación precisa de señales de control que coordinan la interacción entre los distintos bloques funcionales. Su arquitectura interna adopta un enfoque jerárquico, compuesto por cuatro decodificadores especializados: el Main Decoder, el ALU Decoder, el Memory Decoder y el Branch Decoder, tal como se muestra en la Figura 6.

El Memory Decoder se encarga exclusivamente de interpretar el campo funct3 de la instrucción para determinar el tamaño (byte, half-word, word) y el signo (con/sin signo) de la palabra que será accedida en la memoria, ya sea para operaciones de lectura o escritura.
Por su parte, el Branch Decoder evalúa si se deben realizar saltos condicionales, verificando que se cumplan las condiciones especificadas en la instrucción. Esta evaluación depende principalmente del bit zero y del bit más significativo del resultado de la ALU, además del campo funct3, que define el tipo de comparación.

Si bien estos módulos contribuyen al comportamiento correcto del procesador, los componentes más complejos y críticos en la toma de decisiones son el Main Decoder y el ALU Decoder, los cuales analizan el opcode y campos adicionales de la instrucción para generar múltiples señales que afectan directamente la selección de rutas de datos, operaciones

de la ALU, control de registros y acceso a memoria. La descripción detallada de estos módulos se presenta en las secciones posteriores.

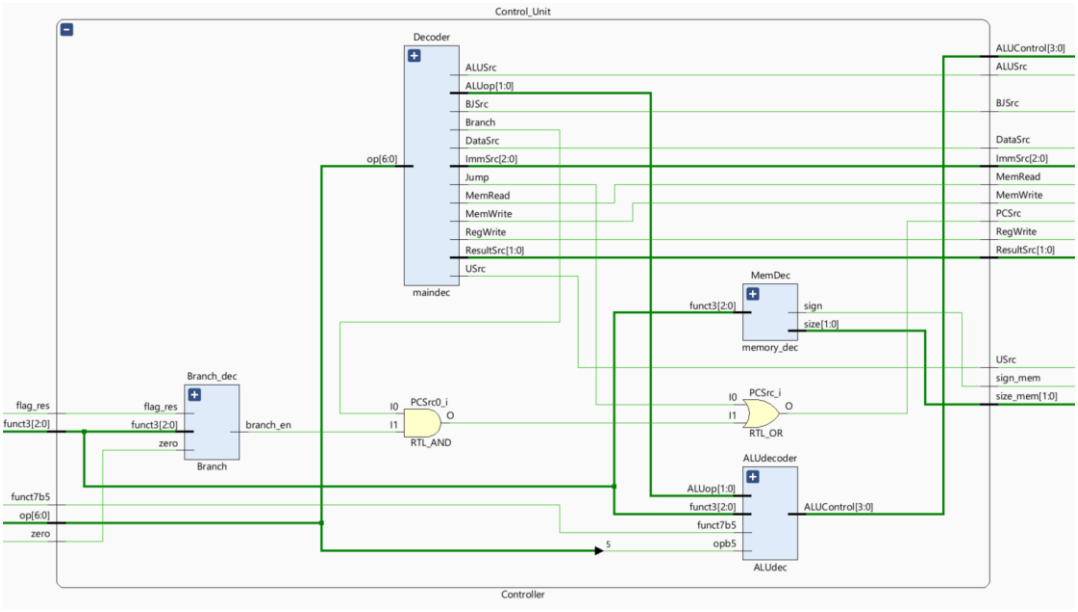

Figura 6. Diagrama de bloques del Controlador

Módulo Main Decoder

El Main Decoder identifica el tipo de instrucción a partir del campo opcode, generando las señales de control correspondientes para gobernar el comportamiento del datapath. Estas señales incluyen el control de escritura en registros, selección de fuentes para la unidad aritmético-lógica (ALU, habilitación de memoria y direccionamiento de salto o bifurcación, entre otr. Asimismo, el Main Decoder produce la señal ALUop, la cual es utilizada por el ALU Decoder para determinar la operación aritmético-lógica específica a ejecutar. Señales como Branch y Jump también se generan en este módulo y se emplean para modificar el flujo de control del programa mediante decisiones lógicas evaluadas en etapas posteriores del procesamiento[5]. En la Tabla 5 se detallan todas las señales de control emitidas por este módulo según el tipo de instrucción.

Instrucción	Opcode	Reg Wr	IMm Src	ALU Src	Mem Wr	Result Src	PC Src	Branch	ALUop	Mem Re	Jump	BJ Src	U Src
Load I-type	0000011	1	000	1	0	01	0	0	00	1	0	x	x
S-type	0100011	0	001	1	1	xx	0	0	00	0	0	x	x
R-type	0110011	1	xxx	0	0	00	0	0	10	0	0	x	x
B-type	1100011	0	010	0	0	xx	1	1	01	0	0	0	x
I-type	0010011	1	000	1	0	00	0	0	10	0	0	x	x
JAL	1101111	1	011	x	0	10	1	0	xx	0	1	0	x
JALR	1100111	1	000	1	0	10	1	0	00	0	1	1	x
LUI	0110111	1	100	x	0	11	0	0	xx	0	0	x	0
AUIPC	0010111	1	100	x	0	11	0	0	xx	0	0	0	1

Tabla 5. Tabla de verdad del Main Decoder

Módulo ALU Decoder

El módulo ALU Decoder genera una señal de control de 4 bits denominada ALUControl, la cual define la operación que ejecutará la ALU. Esta señal se deriva a partir de múltiples campos de la instrucción y del valor de ALUop. En particular, intervienen los bits [14:12] (campo funct3), el bit 30 (parte de funct7), y el bit 5 del campo opcode.

La señal ALUop se utiliza para clasificar el tipo de operación general a realizar:

- 00: instrucciones de acceso a memoria (carga/almacenamiento),
- 01: instrucciones de salto condicional (branch),
- 10: operaciones aritméticas o lógicas (tipo inmediato o tipo registro).

La Tabla 6 resume las combinaciones de entrada y las correspondientes señales ALUControl generadas por este decodificador.

ALUop	funct3	op5, funct7_5	ALUControl	Instrucción
00	x	xx	0000(add)	LW,SW
01	000	xx	0001(sub)	BEQ
	001	xx	0001(sub)	BNE
	100	xx	0011(STL)	BLT
	101	xx	0011(SLT)	BGE
	110	xx	0100(SLTU)	BLTU
	111	xx	0100(SLTU)	BGEU
10	000	00, 01, 10	0000	ADD /ADDI
	000	11	0001	SUB

	001	xx	0010	SLL / SLLI
	010	xx	0011	SLT / SLTI
	011	xx	0100	SLTU /SLTIU
	100	xx	0101	XOR / XORI
	101	10,00	0110	SRL /SRLI
	101	11,01	0111	SRA /SRAI
	110	xx	1000	OR / ORI
	111	xx	1001	AND / ANDI

Tabla 6. Tabla de verdad del ALU Decoder

Extensión del conjuto de instrucciones

El ISA de la arquitectura RISC-V reserva ciertos espacios de opcode para permitir la instancia de extensiones personalizadas, conocidas como “non-estándar extensions”. Mediante estas extensiones se ofrece a los diseñadores de hardware la oportunidad de implementar instrucciones propias para aplicaciones específicas, sin que interfieran con las instrucciones estándar[12]. Bajo este concepto, fue necesario extender el conjunto de instrucciones con nuevas operaciones y mecanismos de acceso a datos de formas no contempladas por la estructura tradicional. Esta incorporación implica nuevas definiciones a nivel ensamblador y cambios en hardware en el datapath y señales de control. A continuación se presentan los principales cambios establecidos:

Instrucción para lectura desde una ROM externa

La primera función incorporada se debe debido a que el algoritmo MD5 necesita varias contantes tal y como se explico en el apartado de instancias preliminares. Esta instrucción incorpora un multiplexer con el módulo “data memory” basado en un cambio en el opcode que provee una ruta alterna para la lectura de datos. De esta manera evitamos ingresarlos manualmente ni instanciarlos dentro de la memoria RAM con el riesgo de que estos valores sean modificados.

En este contexto, se identificó que la instrucción load corresponde a un formato tipo I con el opcode 0000011. Tras analizar el espacio disponible en el mapa de opcodes de RISC-

V, se encontró una alternativa en el opcode 1000011, que difiere únicamente en el bit 6. Este bit fue aprovechado para generar la señal de control DataSrc, la cual es el selector del multiplexor de la fuente de datos. Esta modificación habilita la lectura desde una memoria ROM externa, como se ilustra en la Figura 7.

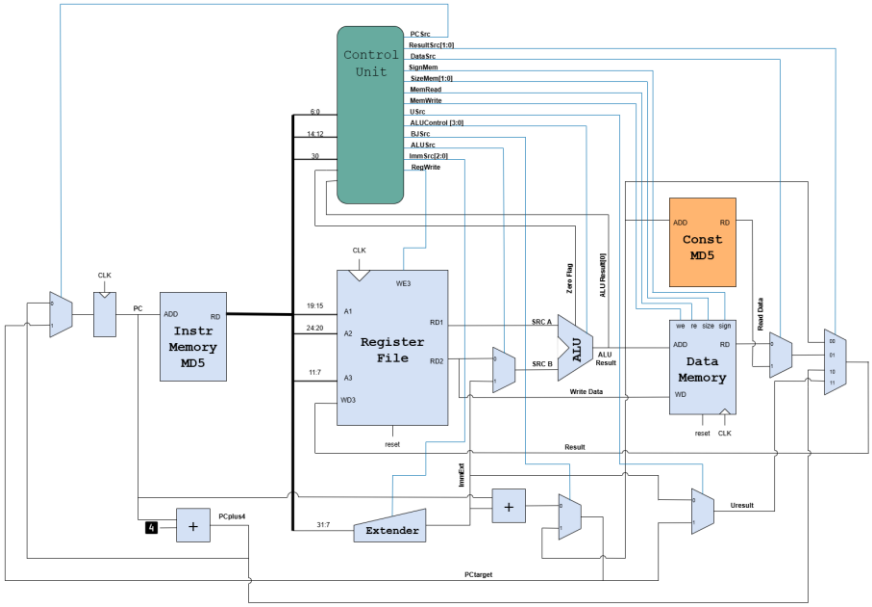

Figura 7. Diagrama de bloques modificado para una ROM externa

Instrucción ROL (Rotate on Left)

La instrucción ROL es fundamental en la implementación del algoritmo MD5, ya que este emplea múltiples rotaciones circulares a la izquierda como parte de su operación no lineal para mezclar los bits de los datos. En estas rotaciones, el bit que sale por el extremo más significativo se reinserta en el menos significativo, conservando su orden[2]. Para implementar la instrucción ROL, es necesario modificar la tabla del ALU Decoder. Dado que ROL comparte el mismo valor de func3 con la instrucción SLL (Shift Left Logical), se debe utilizar el bit 30 como mecanismo de diferenciación entre ambas.

ALUop	funct3	op5, funct7_5	ALUControl	Instruction
10	001	10,00	0010	SLL / SLLI
	001	11,01	1010	ROL/ROLI

Tabla 7. Modificación de la tabla ALU Decoder - ROL

Instrucción para realización de hash

Para llevar a cabo la implementación del algoritmo de hash, se diseñó una nueva instrucción tipo H (de hash), cuya distribución de pines se muestra en la ilustración correspondiente. Esta instrucción tiene dos propósitos principales: en primer lugar, iniciar una máquina de estados que permita la generación del hash, esta operación se identifica cuando el bit D está en 0; en segundo lugar, indicar que el proceso de hash ha finalizado (done), y se activa cuando el bit D está en 1. De este modo, al detectar la instrucción de inicio correspondiente al opcode 7'b0111011, el procesador detiene su flujo de ejecución regular para dar paso al flujo específico correspondiente al cálculo del hash.

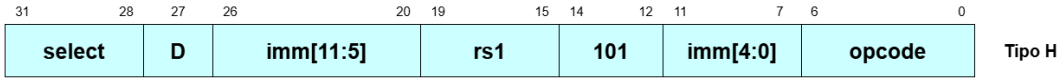

Figura 8. Instrucción tipo H

Asimismo, esta instrucción tiene como objetivo que el resultado del hash se almacene en una ubicación específica de memoria. Para ello, se utiliza el campo rs1, que indica el registro con la dirección base, y un conjunto de bits en el campo de inmediato que define el desplazamiento, siguiendo un formato propio de la instrucción tipo H. Esto implica una adaptación del bloque *extender* para satisfacer esta funcionalidad. Además, el campo func3 se encuentra predefinido con el valor de 3 bits en binario (101), lo que indica que se almacenará una palabra completa. Finalmente, el bloque *select* representa una proyección del desarrollo actual, que permitirá en el futuro la implementación de un algoritmo de selección entre distintos tipos de hash. Un ejemplo de la instrucción en assembly seria el siguiente:

$$hash \;\; offset(rs1)$$

IMPLEMENTACIÓN DEL ALGORITMO MD5

Para la implementación de la aplicación en el proyecto, se incorporaron dos bloques funcionales fundamentales: el bloque de Padding/Memory y el bloque de FSM (Máquina de estados finitos). Esta estructura modular permite organizar el procesamiento del hash en etapas definidas, lo que optimiza tanto la fase de preparación del mensaje como la ejecución secuencial del algoritmo criptográfico. La integración de estos componentes resultó en un diagrama de bloques funcional que opera como una extensión especializada para el procesador RISC-V, potenciando sus capacidades de seguridad sin comprometer su rendimiento base.

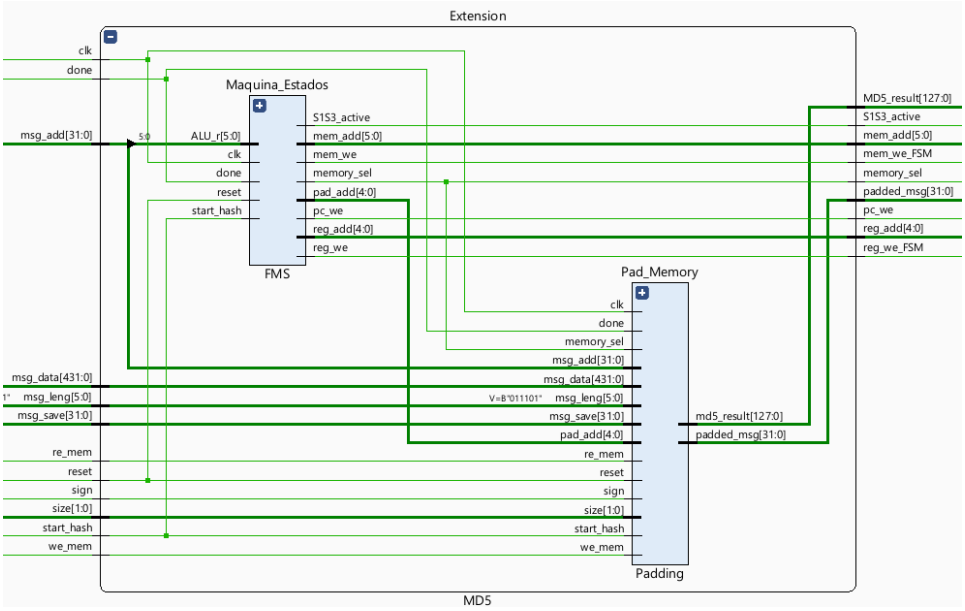

Figura 9. Diagrama de bloques de la extensión Hash

Módulo Padding/Memory

El módulo Padding/Memory cumple dos funciones principales dentro del proceso de generación del hash MD5. En primer lugar, se encarga de adaptar el mensaje original para cumplir con el formato requerido por el algoritmo, cuyo detalle se describe en la sección del marco teórico dedicada al funcionamiento del hash. En segundo lugar, administra una memoria interna de 32 palabras de 32 bits, donde se almacenan tanto el mensaje procesado

como otros datos relevantes. Las primeras 16 posiciones están destinadas al mensaje preparado, mientras que las últimas cuatro contienen las constantes de desplazamiento necesarias para la ejecución del algoritmo.

El módulo permite operaciones de lectura y escritura controladas mediante señales (re_mem, we_mem, memory_sel) y soporta accesos de diferentes tamaños. Al finalizar el proceso (done), el resultado del hash se obtiene concatenando las palabras ubicadas en las posiciones 16 a 19 de memoria, que contienen los 128 bits correspondientes a la salida del hash.

Módulo FSM

El módulo FSM es el encargado de controlar el flujo del sistema generador de hash, asegurando una transición ordenada entre el funcionamiento normal del procesador y el proceso de cálculo del hash. En su estado inicial o de inactividad, el sistema opera de forma convencional, sin limitar las funcionalidades del procesador. Cuando se detecta la instrucción de hash y el controlador emite la señal start_hash, la FSM transita al estado 1. En esta fase, se realiza el resguardo de los registros del procesador en una sección reservada de la memoria de datos, utilizando señales de control que redirigen los valores desde el datapath hacia el puerto de entrada de dicha memoria.

Luego, en el estado 2, se reinicia el program counter (PC) y se realiza el intercambio de la memoria de instrucciones por un conjunto compacto de instrucciones dedicadas al cálculo del hash MD5. Finalizado este proceso mediante la señal done, el sistema entra en el estado 3, donde los registros son restaurados desde la memoria hacia el register file, y el resultado del hash es almacenado en la memoria de datos. La Figura 10 se muestra el diagrama correspondiente a esta máquina de estados.

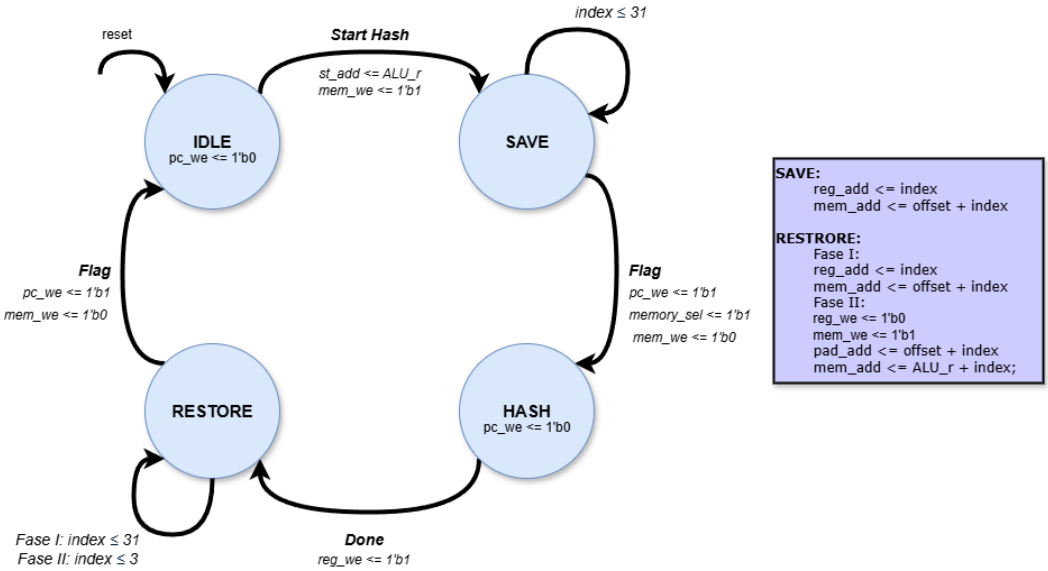

Figura 10. Diagrama de flujo de la FSM

RESULTADOS

Dentro de este capítulo se presentan los resultados obtenidos tras la implementación del sistema generador de hash MD5 en el procesador RISC-V. Estos resultados se dividen en dos secciones principales que permiten validar tanto la lógica funcional como el comportamiento físico del diseño. La primera sección corresponde a la simulación en Vivado, donde se evalúa el funcionamiento del sistema en un entorno controlado. La segunda sección muestra la implementación en FPGA, donde se comprueba el rendimiento y la operatividad del sistema en hardware real.

Simulación en Vivado

Para efectuar la simulación dentro del entorno de desarrollo Vivado, fue necesario implementar un conjunto de 22 archivos Verilog que constituyen, de manera estructurada, el procesador RISC-V, el modelo de flujo hash y las memorias externas (RAM y ROMs). Esta simulación representa la primera fase de validación del funcionamiento integral del sistema. El proceso de validación siguió una metodología ascendente, comenzando con pruebas individuales de cada módulo y progresando gradualmente hacia niveles de abstracción superiores hasta alcanzar la integración completa del sistema procesador.

La arquitectura del módulo principal (top) comprende diversos bloques funcionales estratégicamente integrados:

1. El procesador RISC-V, que incorpora el controlador y datapath con sus respectivas adaptaciones específicas para esta implementación.
2. Un módulo de memoria de datos (data memory) que implementa la RAM, estructurada en dos segmentos diferenciados:
    - Un segmento accesible por el usuario para operaciones convencionales.

- Un segmento reservado que funciona como espacio de respaldo para los registros del sistema.

3. El módulo de extensión diseñado específicamente para gestionar el flujo del algoritmo hash, que complementa las funcionalidades del procesador principal.
4. Tres memorias ROM dedicadas que almacenan las instrucciones del procesador y las constantes necesarias para la ejecución del algoritmo hash.

Adicionalmente, se efectuó una validación cruzada de los resultados generados mediante su comparación con valores de referencia obtenidos a través de herramientas de software especializadas, garantizando así la precisión en la implementación del algoritmo MD5. La Figura 11 ilustra el resultado esperado, generado mediante la herramienta en línea "MD5 Hash Generator", mientras que la Figura 12 exhibe la simulación correspondiente ejecutada en el entorno de desarrollo Vivado.

Your String	Universidad San Francisco de Quito
MD5 Hash	d19990b74e8ae7f6f89378f5cbcf81f1

Figura 11. Resultado del Hash - MD5 Hash Generator

Figura 12. Resultado del Hash - Simulación en Vivado

El recuadro azul ilustra las instrucciones de carga (load) desde la memoria de datos (data memory) hacia una dirección específica, considerando que la instrucción de entrada fue "Hash 0x8(t6)", donde el registro t6 contenía el valor 0x8. En consecuencia, el resultado del

hash se almacena a partir de la dirección 0x10, mientras que la señal del estado confirma el retorno al flujo normal de ejecución del procesador.

En el recuadro rojo se verifica que los valores del hash fueron almacenados correctamente en memoria. La diferencia en el formato se debe al orden de bytes Little-endian utilizado internamente. Al realizar la conversión adecuada de formato, se obtiene un resultado que coincide exactamente con el valor esperado, validando así la correcta implementación del algoritmo.

$$Big\ Endian \rightarrow 128'hd19990b74e8ae7f6f89378f5cbcf81f1$$

Implementación en FPGA

La implementación en FPGA se llevó a cabo para validar el sistema en un entorno físico, en este caso se utilizó un dispositivo de Diligent – Zybo Z7. El diseño fue sintetizado y programado en el dispositivo, y se realizaron pruebas de funcionamiento en tiempo real. Debido a la limitación de puertos físicos fue necesario configurar el procesador en bloques IP (Intellectual Property). Con lo cual, mediante módulos VIO(Virtual Input/Output) y ILA(Integrated Logic Analyzer) se puedo realizar la debida implementación. Otro punto a considerar es la incorporación de un bloque Clocking Wizard que nos permite regular la frecuencia del reloj, transformando de 125MHz que nos brinda la tarjeta a 40MHz que es la velocidad que el procesador maneja. A continuación se observa el esquema de implementación.

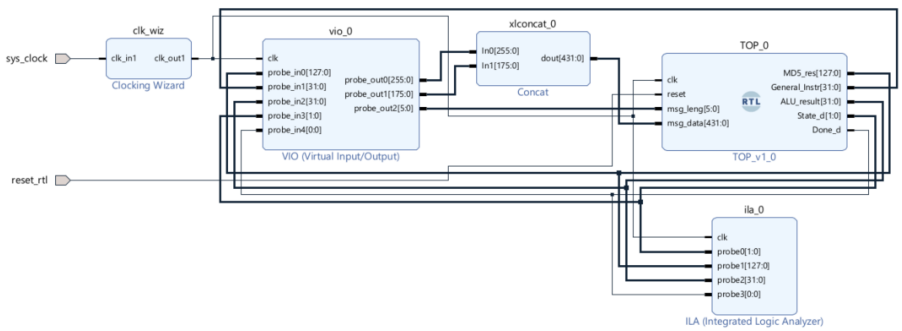

Figura 13. Bloques IP - Implementación del procesador

Se implementó un entorno de pruebas comprehensivo mediante el cual se cargaron diversos mensajes de entrada con complejidad variable, monitorizando meticulosamente los resultados del algoritmo hash obtenidos a través de los puertos de salida específicamente configurados para este propósito. Adicionalmente, se verificó el comportamiento dinámico del sistema mediante señales de diagnóstico, lo cual permitió confirmar la correcta transición a través de todos los estados críticos del flujo de ejecución establecido en la máquina de estados finitos. En la Figura 14 se puede observar el valor de hash generado demostrando la coincidencia con el resultado obtenido en simulación validando así tanto la integridad de la lógica algorítmica implementada como la robustez de la integración completa del diseño.

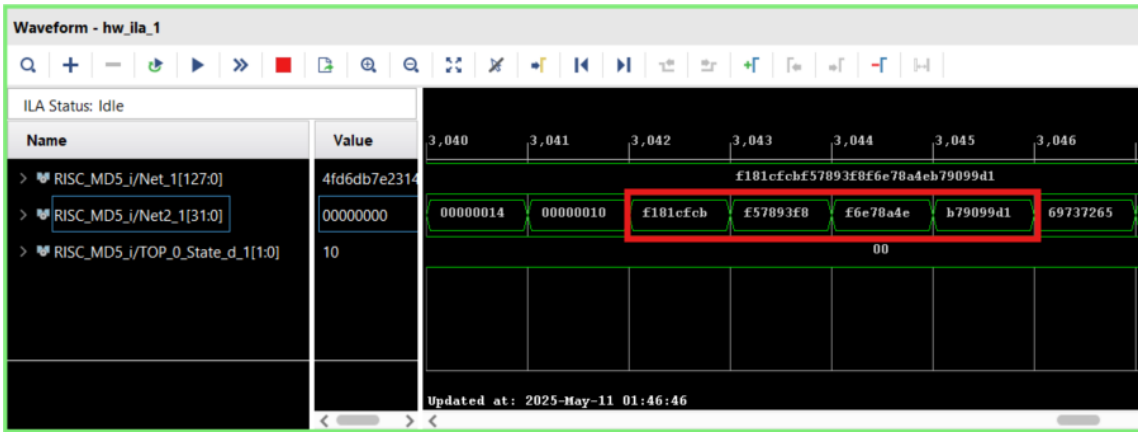

Figura 14. Simulación en implementación -ILA

A continuación, se presenta un análisis de los recursos utilizados y del desempeño del procesador RISC-V con el módulo generador de hash MD5. De esta forma se permite evaluar la eficiencia del diseño en términos de uso de lógica programable, consumo de potencia y velocidad de procesamiento.

Recursos utilizados de la FPGA

Durante la implementación encontramos un uso de 6736 LUTs (38,3 %) y 9099 flip-flops (25,9 %) refleja un diseño equilibrado: la lógica de control y datos ocupa una fracción moderada, lo que deja espacio para futuras extensiones. El consumo de 18,5 bloques de

BRAM (30,8 %) indica la importancia del almacenamiento interno para el buffering del mensaje y los resultados intermedios del hash MD5, mientras que el bajo uso de LUTRAM (4,2 %) indica que la mayoría de las operaciones de memoria se realizan en BRAM dedicados. Utilizamos en I/O(2%) debido a los pines virtuales de medición y el uso de un bloque MMCM (Multi-Mode Clock Manager) se debe a la adaptación de la frecuencia del reloj.

Resource	Utilization	Available	Utilization %
LUT	6736	17600	38.27
LUTRAM	254	6000	4.23
FF	9099	35200	25.85
BRAM	18.50	60	30.83
IO	2	100	2.00
MMCM	1	2	50.00

Figura 15. Recursos de la FPGA utilizados - Implementación

Consumo de potencia

El sistema implementado presenta un consumo total de 0,262 W, distribuido entre consumo estático (0,093 W, 35%) y dinámico (0,169 W, 65%). En el componente dinámico, el MMCM destaca como el mayor consumidor con 0,106 W (62%), seguido por las señales internas de interconexión con 0,033 W (19%). Los subsistemas de reloj (6%), lógica (9%) y memoria BRAM (3%) muestran consumos relativamente menores.

Esta distribución energética evidencia que la infraestructura de temporización, particularmente el MMCM responsable de la conversión de frecuencia de 125 MHz a 40 MHz, constituye el factor dominante en el presupuesto energético. Mientras tanto, los módulos dedicados a la lógica de hashing y almacenamiento mantienen un impacto secundario (0,156 W combinados) siendo así un procesador de baja potencia.

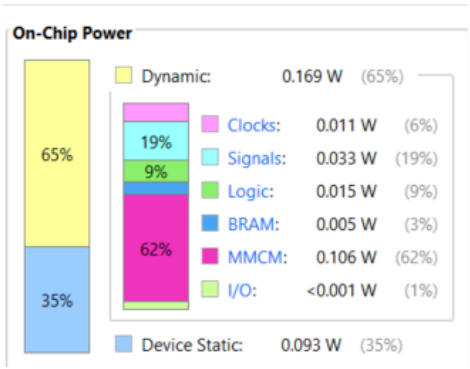

Figura 16. Consumo de potencia - Implementación

Análisis de tiempo

Un análisis temporal verifica que las señales en el circuito lleguen a tiempo y sin errores. El margen de configuración (WNS) de 2.157 ns indica que las señales llegan con suficiente anticipación para ser capturadas correctamente. El margen de retención (WHS) de 0.031 ns, aunque ajustado, sigue asegurando que los datos se mantengan estables el tiempo necesario después del reloj. Además, el margen de ancho de pulso (WPWS) de 2.000 ns garantiza que las señales de reloj tengan la duración adecuada. La Figura 17 muestra que no hay fallos en ninguno de estos parámetros, lo que asegura un funcionamiento confiable y estable del sistema.

Figura 17. Análisis de tiempo - Implementación

CONCLUSIONES

El desarrollo de un procesador RISC-V con capacidades especializadas para la ejecución de algoritmos de hash representa un aporte relevante tanto a nivel académico como práctico dentro del campo de los sistemas embebidos y la seguridad informática. La implementación de extensiones personalizadas a la arquitectura RV32I permitió incorporar de manera eficiente operaciones específicas para el cálculo del hash MD5, evidenciando la flexibilidad y adaptabilidad de la arquitectura RISC-V para aplicaciones de propósito específico.

Una de las principales diferencias respecto a trabajos tradicionales radica en la implementación de instrucciones personalizadas (non-standard extensions), lo que permitió optimizar la ejecución del hash directamente en hardware, reduciendo la carga de procesamiento que tradicionalmente recaería sobre el software y obteniendo un buen desempeño en potencia debido a que al ser instrucciones de carácter constante se reduce la perdida de enegía por conmutación. Este enfoque representa una ventaja significativa para aplicaciones de seguridad embebida en dispositivos de recursos limitados brindando un porcentaje mayor de certeza en el hash generado.

Entre las principales dificultades enfrentadas se destaca la integración de las extensiones personalizadas sin comprometer el funcionamiento del procesador base, así como la gestión de las dependencias entre módulos dentro de la FSM. La utilización de herramientas como Vivado y la validación cruzada con plataformas de referencia permitieron superar estas limitaciones y garantizar la robustez del diseño.

Entre las principales dificultades enfrentadas se destaca la integración de las extensiones personalizadas sin comprometer el funcionamiento del procesador base, así como la gestión de las dependencias entre módulos dentro de la FSM sin comprometer la veracidad

de la información almacenada en las memorias. Es por lo cual se requirió un rediseño iterativo para asegurar la coherencia entre las fases normales de ejecución y el proceso de hashing.

Como líneas de trabajo futuro, se sugiere la implementación de algoritmos de hash más seguros como SHA-3 o SHA-256, es de considerar que la instrucción tipo H creada tiene un campo de selección lo que nos permite insertar más tipologías de hash. Aparte de eso, se puede incrementar una lógica de selección que permita realizar hashes anidados dependiendo del mensaje de entrada para aumentar la robuztes. Por otro lado, sería valiosa la exploración sobre la generación del layout con ayuda de herramientas especializadas que permitan llevar este trabajo a un diseño top-down apto para fabricación.

REFERENCIAS BIBLIOGRÁFICAS

[1] Ahmed, F. (2016). *Cryptographic Hashing Functions -MD5.* Indiana State University. URL: https://cs.indstate.edu/~fsagar/doc/paper.pdf

[2] Fernández, G. (2015). Elementos de sistemas operativos, de representación de la información y de procesadores hardware y software. Universidad Politécnica de Madrid.

[3] Five EmbedDev. (2021). *The RISC-V privileged architecture specification version 1.12*. https://five-embeddev.com/riscv-user-isa-manual/Priv-v1.12/rv32.html

[4] Golze, U. (1996). VLSI Chip Design with the Hardware Description Language VERILOG. An Introduction Based on a Large RISC Processor Design. Springer.

[5] Harris, S. & Harris, D. (2022). Digital Design and Computer Architecture – RISC-V Edition. ELSEVIER.

[6] Kioon, M., Wang, Z. & Das, S. (2013). *Security Analysis of MD5 Algorithm in Password Storage.* DOI: 10.2991/isccca.2013.177

[7] Kocher, P., Jaffe, J., & Jun, B. (1999). Differential power analysis. *Advances in Cryptology – CRYPTO '99*, 388–397. https://doi.org/10.1007/3-540-48405-1_25

[8] Patterson, D. & Waterman, A. (2018). *Guía Práctica de RISC-V: El Atlas de una Arquitectura Abierta – Primera Edición*. Berkeley, California: Strawberry Canyon LLC.

[9] RISC-V International. *About RISC-V.* URL: https://riscv.org/about/

[10] Stallings, W. (2017). Cryptography and Network Security. Principles and Practice – Seventh edition. Pearson.

[11] Vera, C. (2024). *Aplicación de Ciberseguridad cuántica en la seguridad de puertos de comunicación de la IoT*. Revista Tecnológica - ESPOL (RTE), *36*(2), 135-157. https://doi.org/10.37815/rte.v36n2.1188

[12] Waterman, A., Lee, Y., Patterson, D.& Asanovic, K. (2016). *The RISC-V Instruction Set Manual. Volume I: User-Level ISA – Version 2.1.* CS Division, EECS Department, University of California, Berkeley.

[13] Waterman, A. (2016). *Design of the RISC-V Instruction Set Architecture.* University of California, Berkeley.

ANEXO A: ESTRUCTURA DE INSTRUCCIONES RV32I

Se presenta un compilado de 37 instrucciónes donde se puede apreciar la distribución de bits y el tipo de palabra correspondiente, estas instrucciones corresponden al ISA del procesador RISC-V, específicamente a la extensión RV32I.

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	R sltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and

Figura A. ISA de la extensión RV32I [8]

ANEXO B: TABLA DE CONSTANTES DEL ALGORITMO MD5

En este anexo se presentan las 64 constantes dependientes de la ecuación (1). Estos valores han sido instanciados en la memoria ROM y han contribuido a la adaptación de una nueva instrucción para el ISA.

K[0.. 3]	0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee
K[4.. 7]	0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501
K[8..11]	0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be
K[12..15]	0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821
K[16..19]	0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa
K[20..23]	0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8
K[24..27]	0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed
K[28..31]	0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a
K[32..35]	0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c
K[36..39]	0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70
K[40..43]	0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05
K[44..47]	0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665
K[48..51]	0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039
K[52..55]	0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1
K[56..59]	0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1
K[60..63]	0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391

Figura B. Constantes del algoritmo MD5 [1]

ANEXO C: DIAGRAMAS DE FLUJO DEL PROCESADOR

Se presenta diagramas de flujo generado en esta microarquitecta en específico para cada tipo de palabra que soporta la arquitectura RISC-V (RV32I).

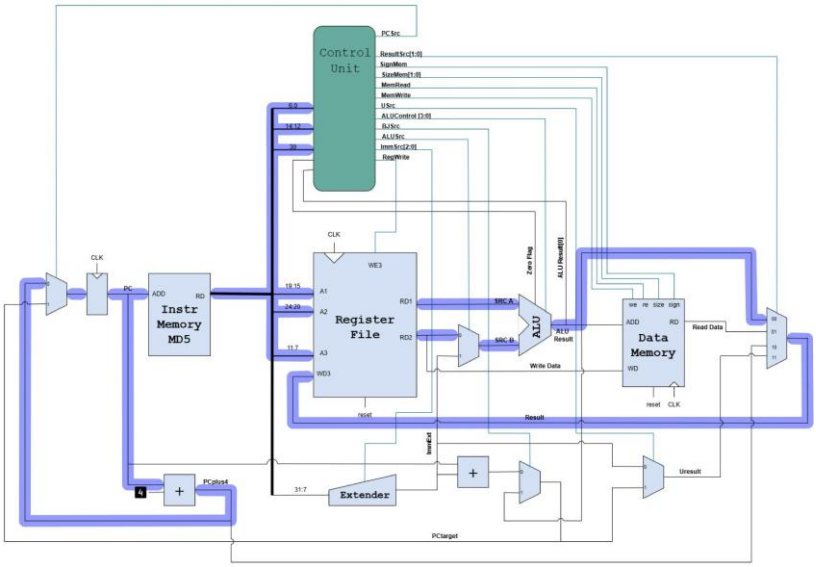

Figura C. Instrucción tipo R - Diagrama de flujo

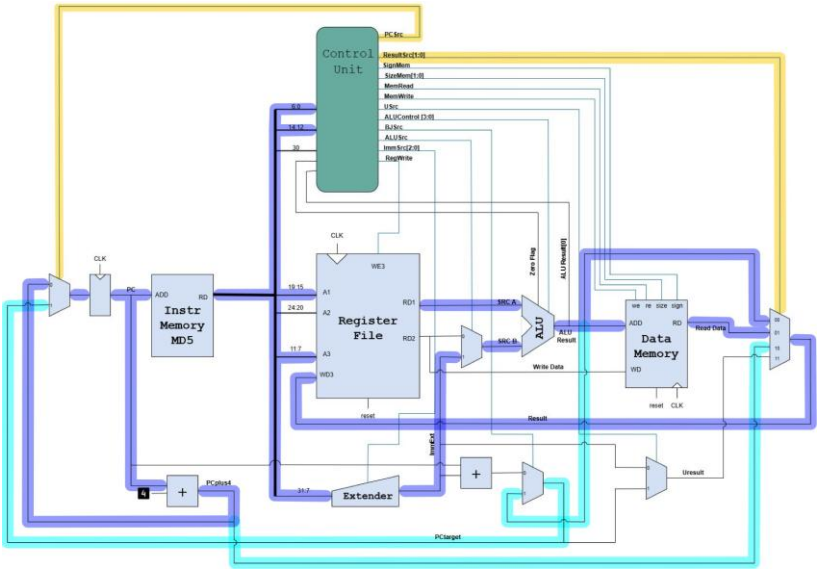

Figura D. Instrucción tipo I - Diagrama de flujo

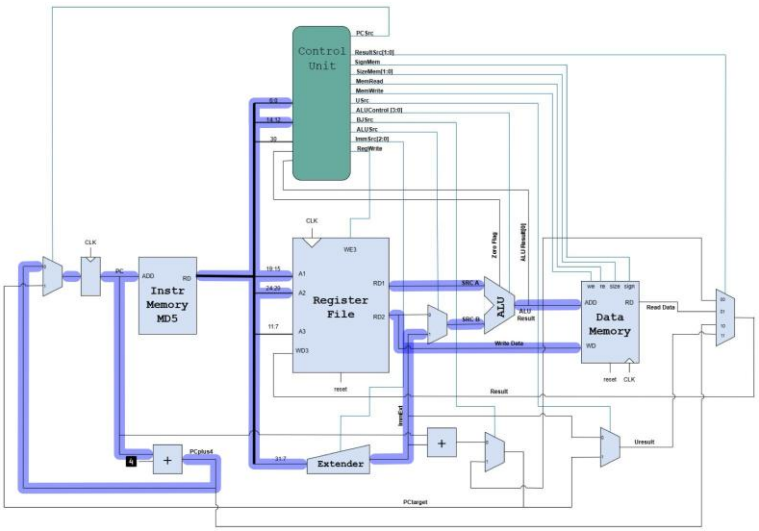

Figura E. Instrucción tipo S - Diagrama de flujo

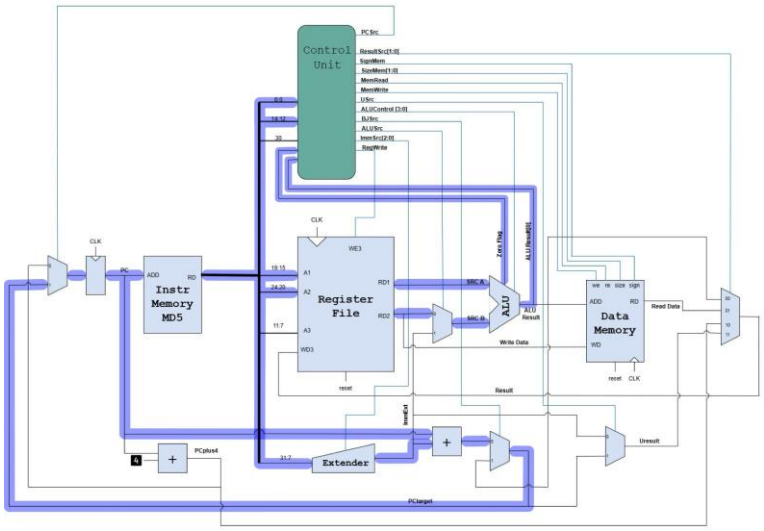

Figura F. Instrucción tipo B - Diagrama de flujo

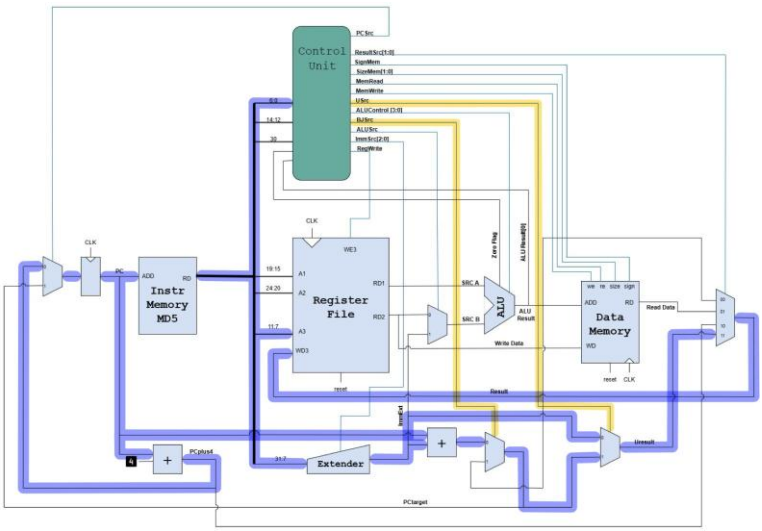

Figura G. Instrucción tipo U - Diagrama de flujo

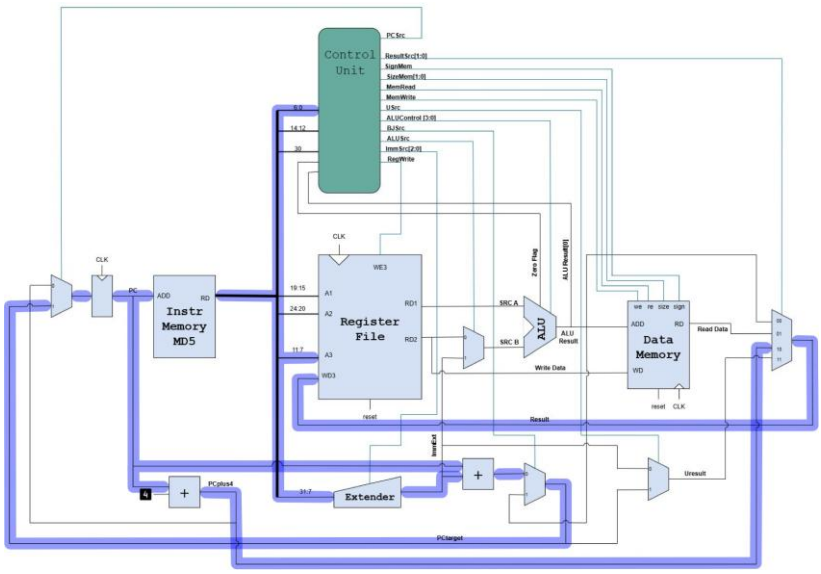

Figura H. Instrucción tipo J - Diagrama de flujo