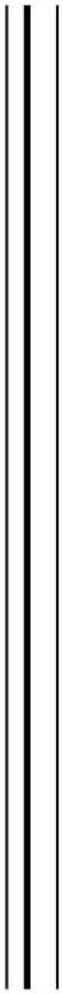
Instituto Politécnico Nacional

escuela superior de cómputo
sección de estudios de posgrado e
investigación



Arquitectura de hardware basada en matrices sistólicas para compresión de datos sin pérdida

T E S I S

que para obtener el título de:

MAESTRÍA EN CIENCIAS EN INGENIERÍA EN SISTEMAS COMPUTACIONALES MÓVILES

presenta:

Ing. Eduardo Iván Mejía Bello

Directores de tesis:

M. en C. Erika Hernández Rubio
Dr. Gelacio Castillo Cabrera

Ciudad de México
Junio 2025

SIP-13
REP 2017

INSTITUTO POLITÉCNICO NACIONAL

SECRETARIA DE INVESTIGACIÓN Y POSGRADO

Dirección de Posgrado

ACTA DE REGISTRO DE TEMA DE TESIS
Y DESIGNACIÓN DE DIRECTOR DE TESIS

Ciudad de México, a 30 de abril del 2025

El Colegio de Profesores de Posgrado de Escuela Superior de Cómputo en su Sesión
(Unidad Académica)

Ordinaria No. 4 celebrada el día 11 del mes abril de 2025 conoció la solicitud presentada por el (la) alumno (a):

Apellido Paterno:	Mejía	Apellido Materno:	Bello	Nombre (s):	Eduardo Iván

Número de boleta: B 2 3 0 6 3 9

del Programa Académico de Posgrado: Maestría en Ciencias en Sistemas Computacionales Móviles

Referente al registro de su tema de tesis

1.- Se acordó aprobar el tema de tesis:

Arquitectura de hardware basada en matrices sistólicas para compresión de datos sin pérdida

Objetivo general del trabajo de tesis:

Diseñar e implementar una arquitectura de hardware especializada en compresión de texto sin pérdida, basada en LZ77, empleando matrices sistólicas.

2.- Se designa como Directores de Tesis a los profesores:

Director: M. en C. Erika Hernández Rubio Director: Dr. Gelacio Castillo Cabrera

No aplica: ☐

3.- El Trabajo de investigación base para el desarrollo de la tesis será elaborado por el alumno en:

SEPI-ESCOM

que cuenta con los recursos e infraestructura necesarios.

4.- El interesado deberá asistir a los seminarios desarrollados en el área de adscripción del trabajo desde la fecha en que se suscribe la presente, hasta la aprobación de la versión completa de la tesis por parte de la Comisión Revisora correspondiente.

Director(a) de Tesis

M. en C. Erika Hernández Rubio

Director de Tesis (en su caso)

Dr. Gelacio Castillo Cabrera

Alumno

Mejía Bello Eduardo Iván

Presidente del Colegio

M. en C. Iván Giovanny Mosso García

S.E.P.
INSTITUTO POLITECNICO NACIONAL
ESCUELA SUPERIOR DE COMPUTO

SIP-14
REP 2017

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

Dirección de Posgrado

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México siendo las 12:00 horas del día 25 del mes de junio del 2025 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Posgrado de: Escuela Superior de Cómputo para examinar la tesis titulada:

Arquitectura de hardware basada en matrices sistólicas para compresión de datos sin pérdida

del (la) alumno (a):

Apellido Paterno:	Mejía	Apellido Materno:	Bello	Nombre (s):	Eduardo Iván

Número de boleta: B 2 3 0 6 3 9

Alumno del Programa Académico de Posgrado: Maestría en Ciencias en Sistemas Computacionales Móviles

Una vez que se realizó un análisis de similitud de texto, utilizando el software antiplagio, se encontró que el trabajo de tesis tiene 6 % de similitud. **Se adjunta reporte de software utilizado.**

Después que esta Comisión revisó exhaustivamente el contenido, estructura, intención y ubicación de los textos de la tesis identificados como coincidentes con otros documentos, concluyó que en el presente trabajo **SI** ☐ **NO** ☒ **SE CONSTITUYE UN POSIBLE PLAGIO.**

JUSTIFICACIÓN DE LA CONCLUSIÓN: *(Por ejemplo, el % de similitud se localiza en metodologías adecuadamente referidas a fuente original)*

Por nombres de instituciones, referencias, nombres de profesores

Finalmente y posterior a la lectura, revisión individual, así como el análisis e intercambio de opiniones, los miembros de la Comisión manifestaron **APROBAR** ☑ **SUSPENDER** ☐ **NO APROBAR** ☐ la tesis por **UNANIMIDAD** ☑ o **MAYORÍA** ☐ en virtud de los motivos siguientes:

Se logro el alcance de los objetivos. Se realizaron las pruebas para validar la arquitectura propuesta.

COMISIÓN REVISORA DE TESIS

M. en C. Erika Hernández Rubio
Director de Tesis
Nombre completo y firma

Dra. Miriam Pescador Rojas
Nombre completo y firma

M. en C. Rodolfo Romero Herrera
Nombre completo y firma

Dr. Gelacio Castillo Cabrera
2° Director de Tesis (en su caso)
Nombre completo y firma

Dr. Rubén Galicia Mejía
Nombre completo y firma

M. en C. Iván Giovanny Mosso García
Nombre completo y firma

S.E.P.
INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

PRESIDENTE DEL COLEGIO DE PROFESORES

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA DE AUTORIZACIÓN DE USO DE OBRA PARA DIFUSIÓN

En la Ciudad de México el día 06 del mes de julio del año 2025, el que suscribe Eduardo Iván Mejía Bello, alumno del programa Maestría en Ciencias en Sistemas Computacionales Móviles con número de registro B230639, adscrito(a) a Escuela Superior de Cómputo manifiesta que es autor(a) intelectual del presente trabajo de tesis bajo la dirección de M. en C. Erika Hernández Rubio y el Dr. Gelacio Castillo Cabrera y cede los derechos del trabajo intitulado: Arquitectura de hardware basada en matrices sistólicas para compresión de datos sin pérdida, al Instituto Politécnico Nacional, para su difusión con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expresado del autor y/o director(es). Este puede ser obtenido escribiendo a las siguiente(s) dirección(es) de correo. emejiab010@gmail.com, emejiab0900@alumno.ipn.mx . Si el permiso se otorga, al usuario deberá dar agradecimiento correspondiente y citar la fuente de este.

Nombre completo y firma autógrafa del (de la) estudiante

A mi esposa y a mi gata, por darme claridad en los días difíciles y recordarme que distraerme también es importante.

Agradecimientos

Este trabajo representa un logro en mi formación académica y personal, y no habría sido posible sin el respaldo de personas cuya presencia ha sido determinante en este camino. Agradezco profundamente al Dr. Gelacio Castillo Cabrera, director de esta tesis, por su orientación constante, por inspirarme a comprender a fondo el funcionamiento interno de los sistemas de cómputo y por fomentar en mí la iniciativa de asumir desafíos comparables a los que enfrentan grandes industrias. Su guía encendió en mí una curiosidad que se convirtió en el motor de esta investigación.

A la M. en C. Erika Hernández Rubio, codirectora de este trabajo, le agradezco sinceramente por su apoyo durante las etapas iniciales, cuando aún no sabía cómo abordar adecuadamente el problema, ni qué ruta debía seguir. Su claridad y firmeza fueron clave para enfocar el esfuerzo de forma efectiva.

A mis padres, por su presencia constante, por escuchar mis ideas, por alentar mis proyectos, y por brindarme enseñanzas y momentos que atesoro profundamente. A mi hermana, por preocuparse por mi bienestar, por impulsarme a ser mejor cada día y por ser una fuente incondicional de apoyo emocional.

Finalmente, a mi esposa, por acompañarme con paciencia y entrega, por escuchar incluso lo que parecía insignificante, y por construir conmigo un camino en el que ambos podamos crecer, comprendernos y sostenernos mutuamente. Su presencia ha sido un pilar firme en el desarrollo de esta investigación y en la consolidación de este logro.

Resumen

Se aborda el diseño de una arquitectura de hardware especializada en compresión de datos sin pérdida, para su futura implementación como un procesador adicional en diversos sistemas. Se realiza el diseño con los principios conocidos como matrices sistólicas. Con ello se disminuye el uso de recursos en el procesador principal realizando la tarea de compresión, un proceso crítico en la vida diaria. Considerando la gran cantidad de dispositivos y software a nivel mundial que realizan este proceso de una u otra forma, al grado que se ha vuelto transparente para el usuario, pero, sigue siendo de suma importancia tanto en el almacenamiento como en la transmisión de información. Con ello, debido al constante incremento de generación de datos y los recursos limitados para poder procesarlos, el tráfico de las redes y la amplia utilización del almacenamiento de datos digitales que se exige en la actualidad; la implementación de algoritmos de compresión de datos en hardware cobra relevancia. El documento revisa diversas técnicas de compresión sin pérdidas, desde su fundamento matemático, como los códigos Huffman, el código aritmético y los algoritmos de Lempel-Ziv (LZ). Se explora el conocimiento necesario para diseñar una implementación basada en el algoritmo de ventana deslizante, tomando en cuenta aspectos técnicos y mejoras que se pueden llevar a cabo. Además, se realiza una prueba de concepto del diseño desarrollado sobre un FPGA, aprovechando su capacidad para realizar múltiples funciones digitales mediante la configuración de bloques lógicos programables y sus interconexiones. La estructura del documento comprende secciones como introducción, planteamiento del problema, justificación, objetivos, antecedentes, metodología, estado del arte, diseño, construcción y pruebas.

Abstract

This work presents the design of a hardware architecture specialized in lossless data compression, intended for future integration as an auxiliary processor within various systems. The design is based on systolic array principles, enabling the offloading of compression tasks from the main processor. This approach is especially relevant given that compression is a critical process in daily computing, across devices and software worldwide, to the extent that it has become transparent to the end-user. Nonetheless, compression remains essential for both data storage and transmission.

Due to the ever-increasing generation of data and the limited resources available to process it, as well as the demands on network traffic and digital storage, the implementation of data compression algorithms in hardware is of growing importance. This document reviews various lossless compression techniques, including their mathematical foundations such as Huffman coding, arithmetic coding, and the Lempel-Ziv (LZ) family of algorithms. A detailed analysis of sliding window-based implementations is presented, considering technical trade-offs and optimization strategies.

A proof of concept of the proposed architecture was developed and tested on an FPGA, leveraging its reconfigurable logic blocks and interconnects to perform specialized digital functions. The structure of this thesis includes sections on introduction, problem statement, justification, objectives, background, methodology, state of the art, design, implementation, and testing.

Índice general

Índice de figuras

Indice de tablas

Capítulo 1

Introducción

En la actualidad, el mundo se encuentra rodeado de dispositivos que generan datos digitales constantemente. Desde los teléfonos móviles hasta los automóviles, los electrodomésticos inteligentes o los equipos médicos, todos estos aparatos producen y procesan información a un ritmo impresionante. Sin embargo, muchos de ellos enfrentan limitaciones como el espacio, el peso y el consumo energético, lo que hace que no siempre sea práctico usar procesadores genéricos diseñados para todo tipo de tareas. En este contexto, las soluciones de hardware dedicado están ganando protagonismo. Por ejemplo, un refrigerador puede usar sensores específicos para medir el nivel de agua o un automóvil puede depender de componentes dedicados para gestionar su sistema de frenos. Estas soluciones no solo mejoran el rendimiento, sino que optimizan los recursos disponibles, lo cual es clave en dispositivos pequeños y de bajo consumo. Lo más interesante es que estas tecnologías no están limitadas a laboratorios o proyectos de alta tecnología. Muchas de las cosas que se usan a diario ya dependen de microcontroladores y otros componentes dedicados para tareas como encender una pantalla, controlar un robot en una fábrica o monitorear signos vitales en un hospital.

El presente trabajo explora una arquitectura de hardware basada en la teoría de matrices sistólicas, que ofrece una combinación eficiente de potencia de procesamiento y flexibilidad. Estas arquitecturas son especialmente útiles en tareas repetitivas como la búsqueda y comparación realizada en la compresión de datos digitales, donde se busca reducir el espacio que ocupan los archivos sin perder información importante. Un ejemplo práctico sería un teléfono móvil que almacena más fotos o videos sin sacrificar espacio, gracias a que usa hardware diseñado para comprimir y descomprimir datos de manera rápida y eficiente. A lo largo de la historia se han propuesto y utilizado ampliamente muchas técnicas de compresión de datos sin pérdidas, por ejemplo, el código Huffman [9], código aritmético [10] y algoritmos de Lempel-Ziv (LZ) [11]. Diferentes arquitecturas de hardware, incluida la memoria de contenido direccionable [12], matriz sistólica [13], entre otras, se han propuesto. Con ello se han presentado varias realizaciones de hardware de LZ y sus variantes. Algunas hasta han sido patentadas [14],

Para las pruebas del diseño propuesto se utiliza una plataforma FPGA, que es un dispositivo reconfigurable. Esto permite probar y ajustar el diseño de hardware sin necesidad de fabricarlo desde cero, reduciendo costos y acelerando el desarrollo. Motivado por llegar a ser integrado en dispositivos móviles a futuro, para hacerlos más eficientes y capaces.

En un mundo donde cada byte de información cuenta, esta propuesta representa un paso importante para afrontar los retos del presente y colaborar en el futuro. Con tecnologías como esta, se abre la posibilidad de transformar no solo los dispositivos actuales, sino también la forma en que se maneja la enorme cantidad de datos producidos.

1.1. Planteamiento del problema

En la era actual, el acceso y la generación de información digital han alcanzado niveles sin precedentes gracias al avance de tecnologías como el Internet, los dispositivos móviles, las redes sociales, la domótica e incluso los propios sistemas operativos que utilizan estos dispositivos. Este ecosistema ha provocado un incremento exponencial en la cantidad de datos digitales generados (en adelante, simplemente "datos"), que deben ser transmitidos entre dispositivos y almacenados para su consulta o uso futuro. Estudios recientes revelan que el 90 % de los datos actuales han sido generados en los últimos dos años. En 2023, la generación global de datos alcanzó los 120 zettabytes y se espera que en 2025 ascienda a 181 zettabytes [15]. [16, 17] Este crecimiento plantea desafíos importantes, especialmente en países como México, que, tomando de ejemplo, se tienen 172 centros de datos, ocupando el puesto 12 en generación de datos a nivel mundial, por delante de países como India (152) y España (143). Respecto a los dispositivos celulares, de acuerdo con INEGI [18, 19] en el mismo periodo, 97.2 millones de personas usaban un teléfono celular en México, representando el 97.1 % de personas que se conectan a Internet. A pesar de este panorama, la infraestructura de conectividad presenta limitaciones significativas: en 2023, la velocidad promedio de descarga en México fue de 60.28 Mbps para conexiones de banda ancha fija y de 25.26 Mbps para datos móviles, ubicándose en los puestos 69 y 80 a nivel global, respectivamente [20, 21]. El almacenamiento y la transmisión de datos, tanto a nivel global como en México, enfrentan un aumento constante en la demanda, superando la capacidad promedio de transmisión disponible. Esta disparidad se agrava en los dispositivos móviles, que deben balancear limitaciones de tamaño, peso, consumo energético, poder de cómputo, entre otros. Estas características que se debe tener en cuenta al diseñar con orientación a los dispositivos móviles, dificultan el manejo de datos mediante soluciones basadas únicamente en software, considerando la necesidad de tener enfoques más dedicados. [5] Actualmente, las estrategias para superar estas limitaciones incluyen, por un lado, la transferencia de operaciones intensivas a la nube mediante Internet, y, por otro, el uso de aceleradores de hardware diseñados específicamente para tareas particulares en dispositivos móviles. Entre estos extremos existen soluciones intermedias, como aceleradores de hardware programables, que ofrecen flexibilidad y eficiencia en diversas aplicaciones. El avance tecnológico impulsado por la Ley de Moore ha beneficiado de forma desigual a los dispositivos móviles y a los dispositivos fijos, dejando en evidencia las brechas de capacidad de procesamiento entre ambos (ver tabla 1.1), donde se observa que, aun comparando los celulares de gama alta contra el promedio de procesadores de uso doméstico, sigue existiendo una brecha entre las prestaciones que pueden ofrecer cada uno. Siendo los dispositivos móviles en la actualidad muy capaces en comparación con hace 20 años o tan solo 10 años, pero los usuarios también cada vez esperan un mayor desempeño contenido en un menor tamaño, por ejemplo, el reconocimiento de voz, o fotografías con gran cantidad de datos tomadas

en una fracción de segundo. Estos desafíos resaltan la necesidad de hardware especifico y diseñado con las mejores técnicas posibles.

Tabla 1.1: Comparación de velocidad de dispositivos móviles y procesadores, adaptado de [5]

Computadora típica		Dispositivo móvil típico		Año
Procesador	Velocidad	Dispositivo	Velocidad	
Intel Core 2 Duo E6600	4.8 GHz (2 cores)	Apple iPhone	412 MHz	2007
Intel Core i5-2500K	13.2 GHz (4 cores)	Samsung Galaxy S2	2.4 GHz (2 cores)	2011
Intel Core i5-3570K	13.6 GHz (4 cores)	Samsung Galaxy S4	6.4 GHz (4 cores)	2013
Intel Core i5-6600K	14 GHz (4 cores)	Samsung Galaxy S7	7.5 GHz (4 cores)	2016
AMD Ryzen 5 1600	38.4 GHz (12 cores)	Google Pixel 2	17.4 GHz (8 cores)	2017
AMD Ryzen 5 3600	43.2 GHz (12 cores)	Samsung Galaxy S20	18.46 GHz (8 cores)	2020
AMD Ryzen 7 5700X	54.4 GHz (16 cores)	Samsung Galaxy S24 Ultra	22.89 GHz (8 cores)	2024

En investigaciones realizadas, se ha demostrado que los circuitos integrados de aplicación especifican (ASIC) pueden ayudar a contrarrestar las limitaciones presentes en los dispositivos móviles [22], logrando tareas como la inferencia en el aprendizaje profundo [23] o la superresolución de imágenes [24]. Estos avances demuestran el potencial del diseño de hardware enfocado en problemas específicos.

Dado el aumento en la generación de datos y las limitaciones de conectividad y almacenamiento, es necesario tener en cuenta soluciones más eficientes. Este trabajo se centra en el diseño de una propuesta de compresión de datos en hardware, con el objetivo de ayudar a mitigar el uso del ancho de banda y almacenamiento en la actualidad y ser parte de la base de soluciones utilizadas en el futuro. Se busca generar una solución mediante hardware específico en compresión de datos para mejorar el uso de los recursos limitados disponibles.

1.1.1. Pregunta de investigación

¿Cómo se puede mejorar la utilización de hardware dedicado para comprimir archivos de texto sin pérdida?

1.1.2. Propuesta de solución

El incremento en la generación de datos requiere estrategias que combinen eficiencia y rendimiento. Para abordar esta problemática, se propone una arquitectura de hardware de compresión sin pérdida basada en el algoritmo LZ77, utilizando matrices sistólicas y presentada en un dispositivo de desarrollo de hardware (FPGA). Esta solución busca aprovechar en lo posible de mejor forma los recursos disponibles, con la motivación que en el futuro pueda llegar a ser implementado en dispositivos móviles, liberando al procesador principal de tareas intensivas de compresión y descompresión. A continuación, se detalla brevemente los aspectos clave de la propuesta a desarrollar.

1. Estrategia de compresión

    a) Algoritmo base: LZ77 se seleccionó por su forma de funcionamiento, identificando patrones en los datos para reducir el tamaño de la información eliminando la

redundancia, todo ello sin perder información en el proceso. Su implementación en hardware permite paralelismo a nivel de bits.

b) Implementación en matrices sistólicas: Generan una estructura eficiente para la implementación del algoritmo basado en LZ77. Cada celda en la matriz se diseña para realizar operaciones de comparación y desplazamiento de patrones, Se busca optimizar la búsqueda y comparación de los datos.

2. Arquitectura de hardware

   a) Unidad de compresión: Diseñada como un módulo específico dentro del FPGA, se encarga de aplicar el algoritmo LZ77 mediante una configuración paralela.

   b) Procesamiento en paralelo: El diseño utiliza una matriz sistólica para procesar el texto, para maximizar el rendimiento y reducir la latencia.

   c) Gestión de memoria: Se utilizan buffers para manejar segmentos de datos y disminuir el acceso a memoria externa, mejorando la eficiencia energética y reduciendo los tiempos de espera.

3. Tarjeta de desarrollo

   a) Se selecciono un FPGA por su flexibilidad y capacidad de re-configuración para realizar pruebas del diseño, sin requerir la fabricación y los recursos que ello conlleva. El diseño inicial utiliza hardware de pruebas compatible con lenguajes de descripción de hardware (VHDL/Verilog).

   a) Ventajas: Flexibilidad para ajustar el diseño y corregirlo, escalable y se puede configurar de formas diferentes, posibilidad de reutilizar bloques en futuros diseños.

4. Estrategia de validación

   a) Simulación y verificación: Se utilizarán herramientas específicamente desarrolladas para diseño de hardware, en este caso Vivado de AMD para sintetizar el diseño y simular su comportamiento frente a diferentes entradas de datos.

   b) Rendimiento: Comparación de rendimiento entre soluciones de software y la implementación en hardware. Se busca validar una mejora en la velocidad de compresión entre el 20 % y 47 % [25].

   c) Métricas de evaluación:

      1) Latencia: Tiempo empleado en comprimir bloques de datos.
      2) Eficiencia Energética: Consumo energético comparado con soluciones puramente de software.
      3) Rendimiento: Capacidad de procesar datos en paralelo y que tasas de compresión se alcanzaron.

1.2. Objetivos

1.2.1. Objetivo general

Diseñar una arquitectura de hardware especializada en compresión de texto sin pérdida, basada en LZ77, empleando matrices sistólicas.

1.2.2. Objetivos específicos

1. Analizar los algoritmos de compresión de texto, considerando sus limitaciones, beneficios, eficiencia y rendimiento.

2. Comparar y seleccionar una técnica para implementar la compresión de texto en hardware, contemplando la eficiencia, rendimiento y adaptabilidad.

3. Diseñar una arquitectura de hardware que integre los elementos necesarios para la configuración de una matriz sistólica, enfocada en compresión de texto sin pérdida.

4. Validar la arquitectura diseñada de compresión sin pérdida, comprobando tasa de compresión, tiempo de procesamiento y parámetros de energía.

1.3. Justificación

El crecimiento sostenido del volumen de datos digitales, impulsado por la accesibilidad a la creación, almacenamiento y comunicación mediante dispositivos móviles, las redes 5G, los dispositivos IoT (Internet of Things en inglés), redes sociales, inteligencia artificial (IA) y dispositivos autónomos, entre otros, han generado un incremento considerable en el número de dispositivos que aportan y transmiten datos [20], [21]. Este aumento de datos digitales conlleva un incremento en los costos, la complejidad y el consumo energético requerido para almacenar o transmitir los datos.

El incremento de datos digitales ha transformado las necesidades de infraestructura tecnológica. En México, las carencias respecto al ancho de banda y almacenamiento contrastan con el crecimiento exponencial en la generación de datos, que alcanzó los 120 zettabytes en 2023 y se proyecta a 181 zettabytes para 2025 a nivel mundial [15].

Estas cifras son evidencia de que la capacidad de transmisión y almacenamiento de datos no puede mantenerse al ritmo de la generación de información. Con ello, el diseño de hardware dedicado para compresión de datos surge como una solución, especialmente en dispositivos móviles, que son el principal medio de acceso a Internet, representando el 97.1 % de la población que se conecta a Internet en el país [18].

Se ha investigado ampliamente en el campo de la compresión de datos [25], [26]. Esta técnica busca reducir el tamaño de los datos, mejorando así la eficiencia del almacenamiento y reduciendo los requisitos de ancho de banda para la transmisión. De forma general se puede clasificar en dos grandes vertientes, compresión con o sin pérdida, dependiendo de los requisitos de integridad de los datos. La compresión con pérdida supone que se puede tolerar cierta degradación en los datos, como ocurre con los archivos de audio en formato

MP3, donde la disminución de calidad no es tan evidente para el oído humano. En contraste, la compresión sin pérdida se utiliza cuando es fundamental mantener la integridad de los datos, como en los archivos de texto, donde perder un solo carácter puede causar errores en la interpretación del contenido o incluso dejar ilegible el documento. También se debe tener claro que la vida diaria está rodeada de dispositivos de hardware dedicados y que de hecho son esenciales para las tareas comunes, por ejemplo, los microcontroladores que manejan la salida a pantalla de los televisores, la medición del nivel de agua en refrigeradores, la unidad de control del motor (ECU) en los autos, el control de robots en líneas de ensamblaje, dispositivos médicos y de seguridad, entre otros. Considerando solo hardware dedicado a compresión de datos, compañías como Microsoft, Broadcom, AMD, ARM o Cadence, están trabajando en ello [27]. La motivación del presente trabajo es llegar a ser utilizado en dispositivos móviles en el futuro, ya que tienen la penalización por su misma naturaleza en sus limitaciones de tamaño, peso, y consumo energético, lo cual se traduce en un poder computacional significativamente menor en comparación con dispositivos estáticos. [5] Aunque estrategias como el *offloading* a nubes o *cloudlets* han sido exitosas, la implementación de aceleradores de hardware en los propios dispositivos móviles, como ASICs (Application-Specific Integrated Circuits), ofrece diversas ventajas, incluyendo baja latencia, operación en condiciones desconectadas y eficiencia energética superior. No obstante, el diseño e implementación de dispositivos de hardware específicos tiene ciertas barreras que en soluciones por software no existen, los más destacables son los altos costos de desarrollo y la falta de flexibilidad para realizar múltiples aplicaciones o realizar mejoras o correcciones.

En México, la conectividad limitada y la velocidad promedio de descarga, que ocupa los puestos 69 y 80 a nivel mundial para banda ancha fija y datos móviles respectivamente [20, 21], amplifican la necesidad de soluciones locales para la compresión de datos. El desarrollo del diseño de la arquitectura de compresión se planea para reducir el volumen de datos que la información utiliza tanto para poderse transmitir como para almacenarse en los dispositivos y liberar recursos en el procesador principal para otras tareas menos repetitivas o con necesidad más inmediata.

Además, optar por investigación en hardware diseñado para tareas específicas, no solo considera las necesidades inmediatas, sino que posiciona a México como un participante activo en la creación de soluciones escalables y eficientes frente a los retos globales que la creciente generación de datos crea.

En este contexto, surge el interés por el desarrollo de diseño de hardware dedicado en la compresión sin pérdida, aunque, por el tiempo y recursos que requiere la puesta en marcha de un hardware de propósito especifico, se utiliza una arquitectura intermedia para realizar pruebas del diseño propuesto.

El diseño de una arquitectura de hardware de compresión basada en algoritmos como LZ77, implementada en plataformas FPGA, representa un paso crucial hacia una gestión de datos más eficiente en dispositivos móviles. Esta propuesta no solo aborda los retos actuales, sino que sienta las bases para la evolución tecnológica en México, aprovechando las ventajas de los aceleradores de hardware para mejorar el rendimiento y disminuir el consumo energético en un entorno de creciente generación de datos en dispositivos móviles.

1.4. Metodología

La metodología planeada para utilizarse a lo largo de este desarrollo principalmente es la basada en investigación teórica, ya que se centra en la creación y evaluación de teorías o modelos que describen y explican fenómenos en ciencias de la computación. En términos generales, se ha distinguido la teoría como lo opuesto a la práctica [28]. La investigación teórica, utiliza la forma de pensar e investigar en búsqueda de soluciones mediante la imaginación, abstracción, deducción; con ello desarrolla explicaciones o teorías sobre fenómenos. Se fundamenta en la corriente racionalista y es propia de las ciencias formales cuyos objetos de estudio son ideales o intangibles, como la matemática, lógica, física teórica o lingüística. Las técnicas incluyen la formulación de hipótesis, la construcción de modelos y la evaluación lógica. De esta forma se hará uso de lenguajes de diseño de hardware, específicamente de lenguajes de descripción de hardware, de los cuales se usan ampliamente VHDL y Verilog a nivel escolar e industrial. Teniendo en cuenta herramientas para analizar y corroborar los datos, como Vivado de AMD, el cual provee un entorno para diseño de tanto entradas como salidas, síntesis, lógica de las señales, verificación y simulación; siendo totalmente compatible con el hardware propuesto para la realización de este trabajo. Guiado por la metodología principal, se pretende utilizar para casos específicos en la prueba de concepto [29], [30], la metodología basada en funciones (Feature Driven Development) que ayuda a dimensionar y priorizar las actividades, dando prioridad a las funcionalidades más críticas. Consiste en cinco procesos, que proveen los métodos, técnicas y guías necesarias, los cuales se muestran en la siguiente figura.

Figura 1.1: Metodología basada en funciones, basado en (18).

La elección de estas metodologías se basa en el interés de estructurar el trabajo, ya que se realizará una investigación con enfoque a nivel teórico, de la cual se realizará una prueba de concepto en hardware, sustentada en teorías ya documentadas, así como desarrollar nuevo trabajo a partir de ellas, con la experimentación guiada con funciones puestas en marcha.

Como se puede apreciar, la metodología basada en funciones comienza con un modelo general que se va refinando proceso a proceso hasta llegar a su culminación. Parte de interés del uso específico de esta metodología recae en que las últimas dos fases son iterativas. Dando como resultado un mejor manejo de los pormenores que surjan en las fases que más trabajo requieren del sistema a desarrollar [30], [31], [32]. Para las etapas de representación de la arquitectura propuesta, por facilidad de comprensión se utilizan diagramas de bloques para visualización de las etapas por las que pasan las señales que representan la información procesada [33].

1.5. Cronograma

El cronograma se dividió por semestres, para una fácil lectura, se consideraron a grandes rasgos las actividades de mayor importancia para el diseño de la arquitectura de hardware, los apartados de investigación y el desarrollo de este. Se considero el ajuste en lo posible con las vacaciones que hay entre ellos.

Tabla 1.2: Cronograma - 2 semestre (B2024), elaboración propia.

Actividad	Febrero	Marzo	Abril	Mayo	Junio
Definición del tema y objetivos de la tesis	■	■	■		
Revisión de literatura	■	■	■		
Análisis comparativo			■	■	
Familiarización con herramientas de diseño			■	■	■
Redacción de tesis (Capitulo 1 y 2)		■	■	■	
Revisión con comité tutorial					■

Tabla 1.3: Cronograma - 3 semestre (A2025), elaboración propia.

Actividad	Septiembre	Octubre	Noviembre	Diciembre	Enero
Diseño preliminar	■	■			
Diseño de arquitectura		■	■	■	
Integración de arquitectura					■
Implementación en hardware					■
Redacción de tesis (Capitulo 3)			■	■	■
Revisión con comité tutorial					■

Tabla 1.4: Cronograma - 4 semestre (B2025), elaboración propia.

Actividad	Febrero	Marzo	Abril	Mayo	Junio
Simulación		■	■	■	
Implementación del diseño en hardware			■	■	■
Desarrollo de la prueba de Concepto		■	■		
Optimización				■	■
Redacción de tesis		■	■	■	■
Revisión con comité tutorial					■

1.6. Estado del arte

En la actualidad, existe una amplia variedad de propuestas de compresión de datos, desarrolladas tanto por instituciones académicas como por grandes empresas tecnológicas, como Microsoft y Google. En particular, Google ha hecho público el trabajo realizado en el desarrollo de al menos cuatro algoritmos de compresión basados en el algoritmo LZ77. Uno de estos algoritmos es Snappy [34], el cual ha sido de código abierto desde 2011, y su versión más reciente (1.2.1) fue lanzada en mayo del año 2024. Snappy es una biblioteca

especializada en la compresión y descompresión de datos, no está diseñada con el objetivo de alcanzar la máxima compresión ni de ser compatible con otras bibliotecas de compresión, sino en lograr velocidades extremadamente altas con una compresión razonable. Comparada con el modo más rápido de Zlib, Snappy es aproximadamente diez veces más rápida en la mayoría de los casos, aunque los archivos comprimidos resultantes pueden ser entre un 20 % y un 100 % más grandes. Tomando en consideración el rendimiento, Snappy está diseñado para ser extremadamente rápido. En un solo núcleo de un procesador Core i7 en modo de 64 bits, puede comprimir datos a aproximadamente 250 MB/s o más, y descomprimirlos a alrededor de 500 MB/s o más (a modo de comparación, Zlib (Deflate) comprime a 74 MB/s en su configuración más rápida y a 24 MB/s con la configuración predeterminada). Estos valores corresponden a las entradas más lentas en su conjunto de pruebas, aunque este aumento de velocidad se logra a expensas de la relación de compresión, ya que la relación de compresión de Snappy es entre un 20 % y un 100 % menor que la de Zlib. En sus evaluaciones, Snappy supera en velocidad a otros algoritmos de compresión similares (como LZO, LZF, QuickLZ, etc.), manteniendo tasas de compresión comparables. Las tasas de compresión típicas, basadas en su conjunto de pruebas, son aproximadamente 1.5 a 1.7 veces para texto sin formato, de 2 a 4 veces para HTML, y 1.0 vez para archivos JPEG, PNG y otros datos ya comprimidos. En comparación, Zlib en su modo más rápido ofrece tasas de 2.6-2.8x, 3-7x y 1.0x respectivamente. Algoritmos más avanzados pueden alcanzar tasas de compresión superiores, aunque generalmente a costa de una menor velocidad. Es importante destacar que la relación de compresión puede variar significativamente según el tipo de datos de entrada. Aunque Snappy es bastante portátil, está optimizado principalmente para procesadores x86 de 64 bits y puede rendir de manera menos eficiente en otros entornos y los algoritmos de compresión rápida como Snappy son tan rápidos que las operaciones de E/S pueden ser el cuello de botella del algoritmo. Una de las ventajas inherentes de utilizar matrices sistólicas en el proyecto a desarrollar. El segundo algoritmo desarrollado por Google es nombrado Gipfeli [35], el cual es un algoritmo de compresión de alta velocidad que utiliza referencias hacia atrás con una ventana deslizante de 16 bits. Está basado en el trabajo de Lempel y Ziv de 1977, y se ha mejorado con una codificación de entropía ***ad-hoc*** tanto para literales como para referencias hacia atrás. Esta implementado en C++, en palabras de sus desarrolladores, esta optimizado para lograr un rendimiento excepcionalmente alto, aunque es aproximadamente un 30 % más lento que Snappy, pero logra un 30 % más de relación de compresión. La tasa de compresión que ofrece es comparable a la de Zlib en su modo más rápido, pero Gipfeli es aproximadamente tres veces más rápido. Esto lo convierte en una solución ideal para numerosos sistemas con limitaciones de ancho de banda, almacenamiento temporal de datos y procesamiento paralelo. Respecto al tercer algoritmo desarrollado por Google, es conocido como Zopfli [36]. El propósito de Zopfli es comprimir datos en el formato Deflate (basado en LZ77 parcialmente) con una eficiencia superior a la de implementaciones tradicionales como gzip y Zlib. Concretamente, Zopfli logra generar archivos comprimidos entre un 3.7 % y un 8.3 % más pequeños en comparación con gzip utilizando la opción –best. Sin embargo, el tiempo requerido para su ejecución es significativamente mayor, siendo aproximadamente cien veces más lento que gzip. Ya que el formato de datos que espera y general el algoritmo LZ77 es muy utilizado en la industria, también los datos comprimidos con Zopfli se pueden integrar en las aplicaciones sin problemas de compatibilidad. Un uso destacado de Zopfli

se encuentra en la web, donde se pueden comprimir las páginas estáticas, y el navegador al momento de visualizarlas las descomprime. Aunque la mejora en los tiempos de carga para el usuario final puede pasar desapercibida, en los dispositivos móviles, estas optimizaciones pueden notarse al tener un menor consumo energético. Otro uso que puede tener Zopfli es en las imágenes PNG, ya que también utilizan el algoritmo LZ77, lo cual se traduce en ahorros significativos en la transmisión de datos, dado el gran uso de PNG en la web. Sin embargo, debido a su bajo rendimiento en términos de velocidad, Zopfli podría no ser adecuado para la compresión de contenido personalizado. Respecto al último algoritmo de Google, es llamado Brotli [37], fue lanzado en 2015 y su última versión publicada fue la 1.1.0 en agosto de 2023. A diferencia de su predecesor Zopfli, Brotli no está diseñado para ser compatible con el algoritmo LZ77. En lugar de eso, Brotli aspira a ser un reemplazo moderno para LZ77. Dado que LZ77 es conocido por su rapidez tanto en la compresión como en la descompresión, además de su razonable relación de compresión, Brotli debe igualar o superar estas características para ser considerado un sustituto viable. Siendo un algoritmo de compresión sin pérdidas de propósito general que emplea una combinación de una variante moderna del algoritmo LZ77 y codificación Huffman. Esto le permite lograr una relación de compresión comparable a los métodos de compresión de propósito general más eficaces disponibles en la actualidad. Además, aunque su velocidad es similar a la de Deflate, Brotli ofrece una compresión significativamente más densa. Los cuatro algoritmos demuestran que en la práctica se sigue utilizando los algoritmos LZ, de forma pura o con variaciones de este, no solo porque tiene un soporte completo en los sistemas actuales, sino que es relativamente simple y rápido codificar y decodificar con él. Lo que da pie a realmente definir lo importante para tener en cuenta respecto a selección de algún algoritmo. Si considerar una variación de LZ, como LZ77 por ser el estándar en el cual se basan los demás algoritmos, o elegir alguna de las nuevas propuestas antes mencionadas. En [6] se comparó Brotli, Deflate incluido en la biblioteca Zlib, Zopfli, LZHAM, entre otros, limitando la selección de algoritmos a aquellos que generalmente tienen una tasa de compresión mayor que Deflate. Cabe aclarar que los resultados de las pruebas en su gran mayoría siempre dependen de las características del hardware sobre el cual se ejecutan, pero brindan la información suficiente para poder tener nociones de que parámetros se deben tener en consideración al momento de medir el rendimiento de la propuesta. Utilizando un set de datos estándar para poder corroborar y comprobar el funcionamiento de los compresores, en [6] utilizaron el corpus Canterbury [38], el cual contiene 1285 archivos con un total de 70,611,753 bytes. Considerando la velocidad de compresión y el nivel de compresión (la relación entre el peso del archivo original y el resultante), entre otras comparaciones que se pueden leer más a detalle en la referencia. La tabla 1.5 muestra los resultados de dicha comparación.

Tabla 1.5: Comparación de algoritmos de compresión, tomado de [6].

Algoritmo	Nivel de compresión	Velocidad de compresión (Mb/s)	Velocidad de descompresión (Mb/s)
Deflate:1	2.913	93.5	323
Deflate:9	3.371	15.5	347.3
Brotli:1	3.381	**98.3**	334
Brotli:11	**4.347**	0.5	289.5
Zopfli	3.580	0.2	**342.1**

Donde, como se mencionó antes, debido a que son algoritmos de software que dependen del hardware, los resultados son muy dependientes del ancho de banda del almacenamiento utilizado, el procesador, la memoria RAM, el sistema operativo, la cantidad de otros procesos ejecutándose, etc.. . . , más aún, el artículo fue creado por Google, por lo tanto, es razonable entender que tiene cierta tendencia a hacer notar que el algoritmo Brotli ofrece mejores resultados que sus contrapartes. Los resultados de la tabla 1.5 dan la oportunidad de conocer el rendimiento promedio que tiene Deflate y sus contrapartes actuales funcionando en dispositivos anfitriones de 64 bits. Los cuales se aprecia que están limitados principalmente por el sistema operativo, ya que, al ser un sistema de propósito general, tiene que asignar el tiempo a diversas tareas, disminuyendo las prestaciones que puede ofrecer el algoritmo. Considerando la plataforma CUDA de NVIDIA, representa una vía clave para la realización de pruebas, gracias a la capacidad de sus GPUs para manejar tareas computacionales de alta demanda mediante programación paralela. Esta dependencia puede ser percibida como una ventaja o una desventaja, de acuerdo con las metas específicas de cada proyecto. CUDA ofrece un entorno con herramientas para administrar hilos y la memoria, lo que permite aprovechar el rendimiento de las GPUs en tareas paralelas. Tomando el ejemplo de CURC [39], el cual es un compresor de datos genómicos utilizando la GPY y CPU de forma heterogénea. CURC logra una compresión mayor que las herramientas tradicionales basadas exclusivamente en CPUs, como SPRING, debido a su capacidad para manejar tareas masivas de forma paralela con mayor velocidad y menor costo computacional. CURC ofrece una velocidad de compresión entre 2.76 y 6.54 veces mayor, y una velocidad de descompresión hasta 2.52 veces superior en comparación con otras herramientas tradicionales, sin sacrificar la tasa de compresión. Además, el sistema se puede escalar para soportar múltiples GPUs, pero tener en cuenta que deben de alinearse con la infraestructura que proporciona NVIDIA. Sin embargo, a pesar de estas ventajas, la dependencia en cuanto a CUDA y NVIDIA plantea varios problemas. La limitación más seria es la confianza total en una única compañía para el soporte de hardware, actualizaciones y todas las herramientas necesarias para el desarrollo. Este punto conllevará problemas si NVIDIA decide que el hardware utilizado se considera obsoleto o si la compañía toma decisiones que puedan afectar la continuidad de soporte o la disponibilidad del hardware. Así como el hecho de que el software solo puede ser accedido por las GPUs de NVIDIA con soporte de CUDA y únicamente la versión funcional y soportada para la GPU utilizada.

Por último, es importante señalar que CUDA es una tecnología propietaria, lo que excluye su uso en entornos que no pueden soportarla y dificulta la portabilidad hacia otras plataformas. Este factor puede limitar las opciones de expansión futura y restringir la flexibilidad del proyecto.

Se diseñará en lo posible el algoritmo en hardware, ya que brinda la oportunidad de que sea dedicado para una sola tarea, comprimir o descomprimir, dando pie a obtener mejores resultados que los ofrecidos solo por software. Involucrando la optimización de la arquitectura mediante matrices sistólicas, se busca afrontar el problema generado por el ancho de banda limitado y las capacidades limitadas que se tienen en los dispositivos móviles en la actualidad, siendo un sustento para contribuir a un futuro de desarrollo sin depender de tecnología de alguna compañía o técnica patentada. Involucrando al hardware, existen varios proyectos con orientación a utilizarlo para distintos fines en reemplazo de soluciones vía software, uno de los ellos [40], busca implementar Deflate en hardware, específicamente

en un FPGA, presentando algunos aspectos de su implementación de hardware para los codificadores LZ77 y Huffman, componentes clave del algoritmo Deflate. Con trabajo a futuro de una posible integración en sistemas de almacenamiento y comunicación de datos. Uno de los aspectos clave del algoritmo Deflate, es que brinda la opción de utilizar codificadores estáticos o dinámicos; estos últimos calculan las frecuencias en función de los datos de entrada; aquí es donde la mayoría de la literatura enfocada en soluciones vía hardware no cumplen con los requisitos para Deflate, ya que se ocupan solo de diseñar diccionarios estáticos. Con ello ofrece espacio para el presente trabajo y su incursión con diccionarios dinámicos. Orientado el desarrollo de hardware en la arquitectura utilizada, se puede buscar implementar el algoritmo Deflate de forma directa, pero esto no aprovecharía en lo posible los recursos que ofrece el FPGA, en [41] se analiza la implementación de hardware del algoritmo de compresión de datos Lempel-Ziv (LZ), enfatizando su importancia en las comunicaciones y el almacenamiento de datos de alta velocidad. Explora varias arquitecturas de hardware de compresión LZ, como la memoria direccionable de contenido (CAM), la matriz sistólica, y compara su eficiencia en términos de velocidad, costo de hardware y capacidad. Se introduce una nueva técnica paralela basada en matrices sistólicas para implementar el algoritmo LZ centrada en mejorar la latencia y la eficiencia. Además, incluye un análisis del efecto de la longitud del buffer de entrada en la relación de compresión y presenta una implementación FPGA de la técnica propuesta para la compresión y descompresión sobre la marcha. La implementación propuesta se describe como eficiente en área y velocidad. Da información sobre cómo seleccionar la longitud del búfer para una relación de compresión óptima, mostrando las implicaciones de la relación entre la longitud del búfer y la eficiencia de la compresión. Parte importante para este trabajo es que realiza una comparación de diseños de matrices sistólicas, ofreciendo los resultados de la implementación del diseño utilizando un FPGA de XILINX, lo que demuestra el potencial de mejoras significativas en la tasa de compresión y la eficiencia. Así como existe desarrollo en compresión vía software por parte de grandes compañías como Google, también se tiene interesen soluciones por hardware. Debido a que la compresión es algo esencial en la búsqueda del manejo de datos, considerando los niveles de generación de datos que se tiene actualmente. Existe documentación de un proyecto de 2019 [28], donde están involucrados diversas compañías como Intel, AMD, ARM, Broadcom, cadence, synopsys, entre otros. Creando una alianza para desarrollar un algoritmo de compresión, teniendo en cuenta la optimización y su implementación en hardware para los tipos de datos comunes en las cargas de trabajo de almacenamiento en la nube. Al introducir innovaciones a nivel de sistema, mencionan haber logrado alcanzar mayores niveles de compresión, mejor rendimiento y menor latencia en comparación con los algoritmos existentes. Microsoft es la principal compañía involucrada, la cual nombra al proyecto "Microsoft's Project Zipline" y hace mención que los resultados tienen un nivel de compresión hasta 2 mayor en comparación con el modelo Zlib-L4 de 64 KB comúnmente utilizado. Mejoras como esta pueden generar beneficios directos en la administración de datos, tanto para las empresas como para los usuarios finales, teniendo un potencial ahorro de costos. Microsoft menciona que tiene un repositorio público, donde se puede tener acceso a especificaciones de diseño de hardware y código fuente de Verilog para lenguaje de transferencia de registros (RTL), donde literalmente mencionan "con contenido inicial disponible hoy y más próximamente". Caso que lamentablemente desde esa primera y única publicación en 2019, no se tiene

más información actualizada del proyecto. Lo cual hace pensar en que el proyecto en la actualidad pudo haber pasado por, su abandono, debido a la alta especialización su privatización de este, entre otras posibles opciones. Dando pie a una interesante propuesta, que, en lo posible, dado el alcance del presente trabajo y los recursos disponibles, se toma en consideración los proyectos relacionados que en la actualidad otros investigadores realizan en torno a la compresión y la utilización de hardware en dicha tarea.

1.6.1. Compresión en dispositivos móviles

[42] Históricamente se ha tenido interés en desarrollar dispositivos móviles con interfaces inalámbricas que provean comunicación incluso mientras el usuario se mueve entre diversas ubicaciones, en otras palabras, el desarrollo de dispositivos que eliminen las restricciones de tiempo y espacio impuestas por las computadoras de escritorio y las redes cableadas. Donde la computación móvil a transformado la forma en que se tiene acceso continuo a servicios y recursos de redes terrestres. El desarrollo de estos dispositivos debe tener en cuenta aspectos como las comunicaciones inalámbricas, la movilidad y la portabilidad. El primer apartado, permite la conexión sin cables, pero enfrenta problemas como la latencia, desconexiones frecuentes y una menor capacidad de ancho de banda en comparación con las redes cableadas. En entornos donde se tenga que depender de dispositivos móviles las desconexiones son comunes debido a la interferencia, que a su vez hace variar el ancho de banda y que se generen complicaciones adicionales, por lo que se necesitan estrategias que puedan adaptarse a estas limitaciones. Considerando la portabilidad, el diseño de dispositivos móviles conlleva restricciones significativas en cuanto al tamaño, peso, consumo de energía y capacidad de almacenamiento. Minimizar el consumo de energía es crucial para prolongar la vida útil de la batería, mientras que la capacidad de almacenamiento limitada obliga a emplear soluciones como la compresión de archivos y el acceso remoto a datos. aunque la computación móvil ofrece la posibilidad de eliminar las restricciones de tiempo y lugar impuestas por los sistemas tradicionales, plantea desafíos únicos que requieren adaptar las estructuras y sistemas actuales para soportar esta nueva realidad. El uso masivo de dispositivos móviles con funciones avanzadas fue iniciado, entre otros, con los asistentes digitales (PDA por sus siglas en ingles), concebidos como dispositivos autocontenidos que eran parte mediante una red móvil de una infraestructura de cómputo mayor. Uno de sus enfoques era permitir el acceso continuo a servicios y recursos, esta combinación de movilidad y redes inalámbricas sentó las bases para nuevas aplicaciones y formas de interactuar, que incluso en fechas recientes tienen gran relevancia en variedad de áreas, gracias a diferentes dispositivos creados bajo los mismos principios, como los teléfonos inteligentes, dispositivos de Internet, tarjetas inteligentes, computadoras corporales, sensores de redes, etc... [43] La evolución tecnológica ha impulsado el desarrollo de procesadores más complejos, con un enfoque en la comunicación, rendimiento y bajo consumo de energía. En sus inicios, los teléfonos móviles de la primera generación (1G) usaban transmisión analógica, que requería más energía y admitía pocos usuarios. Con la llegada de la segunda generación (2G), se adoptaron los procesadores de señal digital (DSP), que proporcionaban una arquitectura flexible y rentable. A medida que avanzó la tecnología, arquitecturas más modernas como los procesadores VLIW y SIMD permitieron un mejor rendimiento y menor consumo de energía. Convirtiendo a los dispositivos móviles en

componentes vitales de la vida diaria, evolucionando a través de los años con cambios significativos en la arquitectura del procesador. Los procesadores modernos como los basados en ARM son fundamentales para los dispositivos móviles debido a su bajo consumo y alto rendimiento. La tendencia actual es hacia sistemas en chip (Soc.) altamente integrados, que combinan múltiples componentes, como CPU, GPU y DSP, en un solo chip para mejorar el rendimiento general y la eficiencia energética. En la actualidad se utilizan en diversos dispositivos móviles procesadores como los ARM Cortex, Qualcomm Snapdragon y Nvidia Tegra, cada uno con su enfoque particular en la eficiencia energética, rendimiento gráfico y capacidad de procesamiento con la meta de optimizarse para dispositivos que demandan más potencia y mayor eficiencia energética. Considerando el avance en la tecnología, se han realizado investigaciones en el campo, de las cuales algunas tienen mayor relevancia para el presente trabajo, a continuación, se presentan brevemente. En [44] se propone una arquitectura de compresión de código diseñada para mejorar el rendimiento de los procesadores embebidos ARM/THUMB. El trabajo propone una arquitectura que reduce el tamaño del código y mejora el rendimiento en general del sistema. Los autores mencionan que así se puede disminuir el tamaño de la información almacenada en la memoria caché y con ello, también minimizar los accesos a memoria. Buscan mantener el equilibrio entre la reducción del código y la carga que se debe agregar por la acción de descomprimir, buscando que el rendimiento del sistema se vea sin afectaciones de consideración. Ofreciendo una solución eficaz para mejorar el rendimiento de los procesadores embebidos a través de la compresión de código. El artículo [45] describe una técnica para reducir el consumo de energía en sistemas híbridos de ARM-FPGA para comprimir datos sin pérdida. Buscan optimizar el uso de la memoria. Al aplicar algoritmos de compresión, los autores logran reducir el tráfico de datos entre la FPGA y la memoria externa, generando un ahorro de energía sin comprometer la integridad de los datos. El artículo también compara este método con otras técnicas y destaca sus ventajas en aplicaciones donde el consumo de energía es un factor crítico, como en dispositivos portátiles y sistemas embebidos. Demostrando que la compresión de datos sin pérdida aplicada a sistemas híbridos ARM-FPGA es una solución viable para disminuir el consumo de energía, manteniendo la eficiencia y el rendimiento en estos sistemas avanzados. En [46] se describe el diseño y funcionamiento de Flywheel, un proxy de compresión de datos desarrollado por Google para mejorar la navegación web móvil. Proponen reducir el consumo de datos y mejorar los tiempos de carga, siendo de especial interés para contrarrestar los efectos de las redes lentas o inestables. El sistema intercepta el tráfico web del usuario y comprime el contenido utilizando los servidores de Google y de ahí los envía comprimidos al dispositivo del usuario. El artículo menciona los problemas técnicos que tuvieron los autores, como mantener la compatibilidad con sitios web dinámicos y cifrados, así como la necesidad de equilibrar la compresión con la latencia adicional introducida por el uso del proxy. Mostrando también el impacto positivo del sistema en la experiencia del usuario, al reducir considerablemente el uso de datos y mejorar el rendimiento. Por su parte [47] se centra en cómo mejorar la eficiencia energética de las aplicaciones que utilizan procesamiento paralelo en arquitecturas móviles heterogéneas. Estas plataformas incluyen diferentes tipos de procesadores, como CPUs y GPUs, que son adecuados para manejar diferentes cargas de trabajo, lo que plantea un desafío en la gestión eficiente de los recursos y el consumo energético. El enfoque del artículo está en maximizar la eficiencia energética sin comprometer el rendimiento de las aplicaciones. Para

lograr esto, los autores proponen un marco que evalúa las características de las aplicaciones y selecciona dinámicamente el procesador más adecuado (CPU o GPU) para ejecutar ciertas tareas, teniendo en cuenta factores como el tipo de aplicación y el estado actual del sistema. Uno de los principales puntos es que, al distribuir inteligentemente las tareas entre los distintos procesadores y ajustar el nivel de paralelismo, se puede reducir significativamente el consumo de energía en dispositivos móviles. Este enfoque permite aprovechar al máximo las ventajas de los procesadores heterogéneos al mismo tiempo que se prolonga la vida útil de la batería, un factor clave para los dispositivos móviles. Demostrando a través de experimentos que este método optimiza tanto el rendimiento como el consumo de energía en comparación con las estrategias tradicionales de ejecución de aplicaciones paralelas. El artículo [48] aborda la necesidad de optimizar la transmisión de datos GPS en dispositivos móviles y del Internet de las Cosas (IoT). Dado que estos dispositivos suelen tener limitaciones de ancho de banda y energía, el procesamiento de datos en el borde (edge computing) se vuelve crucial para reducir la cantidad de datos que se envían a través de la red. Los autores proponen un método de compresión de datos GPS directamente en el dispositivo (en el borde) antes de transmitirlos, lo que reduce significativamente la cantidad de datos sin perder precisión relevante para aplicaciones IoT. El enfoque se basa en algoritmos de compresión que detectan patrones y redundancias en las coordenadas GPS, permitiendo comprimir la información de manera eficiente y, al mismo tiempo, conservando los aspectos críticos de los datos necesarios para la toma de decisiones. Además, el artículo analiza cómo esta compresión mejora la eficiencia energética de los dispositivos móviles, ya que se requiere menos procesamiento en la nube y se minimizan las transmisiones de datos. Este método de compresión es especialmente útil para aplicaciones que implican un gran número de dispositivos IoT, como el seguimiento de vehículos o la gestión de flotas, donde los volúmenes de datos pueden ser enormes. Los autores destacan que su propuesta de compresión de datos GPS para IoT en el borde tiene un impacto positivo tanto en el rendimiento del sistema como en la sostenibilidad de las soluciones IoT, al disminuir el uso de energía y ancho de banda en redes móviles.

Capítulo 2

Marco teórico

La compresión de los datos es un componente importante de la vida moderna; permite representar la información con una menor cantidad de recursos empleados, ya que optimiza el uso del almacenamiento y de transmisión. Esta técnica es muy importante para disminuir el tráfico de datos. Ejemplos de ello pueden ser las plataformas de streaming como Netflix o Spotify, que utilizan algoritmos de compresión con pérdida para que el contenido multimedia, si se transmitiera en la calidad original en que se generaron, consumiría aún más ancho de banda; el contenido comprimido puede reproducirse a diferentes velocidades que se encuentran disponibles en la transmisión. En la aplicación de mensajería WhatsApp, se utiliza una compresión sin pérdida para enviar imágenes que no pierden su calidad con la compresión. En el aspecto técnico, la compresión de datos puede dividirse en dos categorías grandes: con pérdida y sin pérdida. La primera es utilizada en contenido multimedia, ya que elimina detalles menos perceptibles para los usuarios; como se observa en el formato MP3 o en los videos comprimidos con H.264. Por otro lado, la compresión sin pérdida, como la implementada en archivos ZIP, logra que sea recuperada la información original, siendo esencial en aplicaciones que manejan documentos importantes como reportes médicos o archivos legales. En la tabla 2.1 se presentan las características de los dos tipos de compresión.

Tabla 2.1: Comparación entre compresión sin pérdida y con pérdida, elaboración propia.

Aspecto	Compresión sin pérdida	Compresión con pérdida
Propósito	Preservar los datos originales sin alteraciones [49]	Reducir significativamente el tamaño a costa de perder información [50]
Aplicaciones	Textos, datos científicos, bases de datos [51]	Imágenes, audio, video [52]
Eficiencia de Compresión	Baja a moderada [53]	Alta [52]
Calidad de los Datos Recuperados	Exactamente igual a los originales [49]	Puede diferir significativamente de los originales dependiendo del nivel de compresión [50]
Complejidad de los Algoritmos	Moderada [51]	Alta [52]

En la actualidad se siguen desarrollando nuevas propuestas, un ejemplo de ello es utilizar las redes neuronales para compresión con pérdida, en [54] usan modelos de autocodificadores variacionales, donde muestran su efectividad optimizando el tamaño de los datos resultantes. Sin embargo, los métodos clásicos como la codificación Huffman o la codificación aritmética aún son utilizados debido a su simplicidad y eficiencia en casos de recursos computacionales limitados. Así, la compresión de datos es utilizada en una variedad de aplicaciones de la vida cotidiana, desde el entretenimiento y comunicación hasta la infraestructura que soporta las aplicaciones y servicios. La constante evolución de métodos de compresión, impulsada por las necesidades del usuario junto a los avances en el diseño de algoritmos y hardware, dan evidencia que seguirá siendo relevante el trabajo propuesto. A continuación, se dará una breve descripción de los principales fundamentos matemáticos, teorías y técnicas que se consideraron para comprender y desarrollar el trabajo actual.

2.1. Fundamentos matemáticos

La compresión de datos sin pérdida es una técnica utilizada para reducir el tamaño de los datos sin dañar la información. Para comprender como se implementa, así como entender las razones por las cuales funciona, es importante comprender los conceptos en los que se basa. Se presentan los fundamentos clave involucrados en la compresión.

2.1.1. Teoría de la información

[55] La información es un concepto que se experimenta de manera intuitiva todo el tiempo al recibir y enviar mensajes, ya sea al leer, ver o escuchar. A pesar de que definir y medir matemáticamente suena muy abstracto, una de las teorías de la información, desarrollada por Claude Shannon en la década de 1940, proporciona una base matemática sólida para cuantificar la información.

Cuantificar la información implica medir el grado de incertidumbre que contiene un mensaje. Esto se puede ilustrar con el lanzamiento de un dado de seis caras. Antes del lanzamiento, hay seis posibles resultados y una incertidumbre sobre cuál será el resultado final. Al realizar el lanzamiento, se elimina esta incertidumbre al observar un número específico entre 1 y 6. La cantidad de información necesaria para identificar el resultado puede medirse en bits, que representan preguntas binarias (de sí o no) necesarias para distinguir entre todas las opciones posibles.

Por ejemplo, para identificar el resultado de un dado, se necesitan al menos tres bits, ya que $2^3 = 8$ cubre las seis posibles opciones. En términos generales, cuantificar la información de este modo implica determinar la cantidad mínima de bits requerida para representar los resultados de un experimento, como un lanzamiento de dado, o cualquier situación que pueda resolverse mediante respuestas binarias.

El uso del logaritmo es de suma importancia para medir la información. Calcula el exponente necesario para alcanzar un número dado en una base específica. Por ejemplo, para datos en formato decimal, se usa la base 10, mientras que, para datos binarios, se utiliza la base 2. Así, la cantidad de símbolos necesarios para representar un número N está relacionada con $\log_b N$, donde b es la base.

Para determinar cuántos bits se necesitan para expresar un número dado, se puede utilizar la siguiente relación, usando X como la cantidad de bits requeridos:

$$10^k - 1 = 2^X - 1 \tag{2.1}$$

Aquí, $10^k - 1$ representa el mayor número decimal de k dígitos, mientras que $2^X - 1$ corresponde al mayor número binario con X bits. Usando logaritmos, se puede resolver para X:

$$X = k\frac{\log_{10}}{\log_2} \tag{2.2}$$

Al elegir la base 2, el cálculo se simplifica:

$$X = k \log_2 10 \approx 3.32k \tag{2.3}$$

Esto muestra que un dígito decimal contiene aproximadamente 3.32 bits de información. De manera general, para un sistema con base n, la relación es:

$$X = k \log_2 n \tag{2.4}$$

Esto indica que la información contenida en un dígito de base n es equivalente a $\log_2 n$ bits.

En escenarios prácticos, como transmisores que envían datos, el número de símbolos por unidad de tiempo, denotado por s, y la base de los símbolos, n, determinan la cantidad de información transmitida, H:

$$H = s \log_2 n \tag{2.5}$$

Si los símbolos tienen diferentes probabilidades de ocurrencia, la cantidad promedio de información, o entropía, se calcula considerando estas probabilidades. Para n símbolos con probabilidades P_i, donde la suma de todas las probabilidades es igual a 1:

$$\sum P_i = nP \tag{2.6}$$

Esto lleva a expresar H como:

$$H = -s \sum_1^n P_i log_2 P_i \tag{2.7}$$

Aquí, H mide la información promedio transmitida por unidad de tiempo. La entropía por símbolo, E, se define como:

$$E = -\sum_1^n P_i log_2 P_i \tag{2.8}$$

Esto indica que E representa la cantidad mínima promedio de bits necesarios para codificar un símbolo.

Cuando todas las probabilidades son iguales, la entropía es máxima. Esto introduce el concepto de redundancia R, que mide la diferencia entre la entropía máxima teórica y la entropía observada:

$$R = \log_2 n + \sum_1^n P_i log_2 P_i \tag{2.9}$$

En datos completamente comprimidos, donde no hay redundancia:

$$\log_2 n + \sum_{1}^{n} P_i log_1 P_i = 0 \tag{2.10}$$

Estos principios demuestran cómo la teoría de la información establece la base matemática para cuantificar, medir y optimizar la transmisión eficiente de datos en una amplia variedad de aplicaciones cotidianas y tecnológicas.

2.1.2. Códigos prefijos

Existen diversas técnicas que se basan en la codificación de entropía, una de las más conocidas son los códigos Huffman, ya que son competitivamente óptimos y requieren aproximadamente H lanzamientos de dado justos para generar una muestra de una variable aleatoria que tenga entropía H. Ya que, la entropía es el límite de compresión de datos, así como el número de bits necesarios en la generación de números aleatorios. Los códigos Huffman resultan óptimos desde muchos puntos de vista, ya que las secuencias cortas representan letras frecuentes y las secuencias largas representan letras poco frecuentes. Básicamente, intenta reducir la redundancia presente en los datos de entrada y representarlos con menos bits. El algoritmo de codificación Huffman forma parte de los códigos prefijos o códigos Huffman. Para sustentar que los códigos prefijos funcionan en compresión y son óptimos, se deben definir condiciones estrictas en los códigos. Si X^n denota $(x_1, x_2, \ldots, x_n)$. Considerando la definición: Un código es no singular si cada elemento del rango de X se asigna a una cadena diferente en D^*; es decir:

$$x \neq x' \rightarrow C(x) \neq C(x') \tag{2.11}$$

La no singularidad es suficiente para una descripción sin ambigüedad de un único valor de X, aunque normalmente se desea enviar una secuencia de valores de X. Por ello se requiere tener una extensión de la definición anterior. Definición: La extensión C^* de un código C es el mapeo de cadenas de longitud finita de X a cadenas de longitud finita de D, definidas por:

$$C(X_1 X_2 \ldots X_n) = C(x_1) C(x_2) \ldots C(X_n) \tag{2.12}$$

donde $C(x_1) C(x_2) \ldots C(x_n)$ indica la concatenación de las palabras de los códigos correspondientes. Con base en ello, la tercera definición es: Un código es llamado únicamente decodificable si su extensión es no singular. Lo cual significa que cualquier cadena codificada en un código únicamente decodificable tiene solo una posible cadena fuente que la produce. Sin embargo, es posible que sea necesario observar la cadena completa para determinar incluso el primer símbolo en la cadena fuente correspondiente. Definiendo finalmente que un código es llamado código prefijo o código instantáneo si ninguna palabra de código es un prefijo de ninguna otra palabra de código. Siendo que un código instantáneo se puede decodificar sin referencia a palabras clave futuras, ya que el final de una palabra clave es inmediatamente reconocible. Por tanto, para un código instantáneo, el símbolo X se puede decodificar tan pronto como se llega al final de la palabra clave correspondiente. En otras palabras, un código instantáneo es un código que se puntúa a sí mismo.

2.1.3. Métodos estadísticos

La realización de métodos estadísticos o de codificación de la entropía tiene sus fundamentos en la teoría de la información, que fue descrita públicamente por primera vez por Claude Shannon en 1948. Los métodos estadísticos utilizan códigos de longitud variable, ya que asignan códigos más cortos a los símbolos que son más frecuentes, utilizando métodos de tamaño variable. La consecuencia de ello es que tanto los diseñadores como los implementadores deben tener en cuenta, que se debe asignar códigos que puedan ser fácilmente interpretados sin ambigüedad y se debe asignar códigos con el tamaño mínimo en promedio. De acuerdo con [55] Claude Shannon proporciona la explicación de la entropía. Basada en un conjunto de probabilidades y una fuente de información. Esta entropía, en el contexto de la teoría de la información, representa el promedio de la cantidad de bits necesarios para codificar la salida de dicha fuente. En esencia, la entropía de una fuente de información cuantifica la incertidumbre asociada con sus salidas posibles. Shannon demostró que la cantidad mínima promedio de bits requerida para codificar la salida de manera óptima está determinada por la entropía de la fuente. Esto significa que ningún compresor sin pérdidas puede superar la eficiencia de codificación que lograría utilizando un número promedio de bits equivalente a la entropía de la fuente. En resumen, la entropía no solo proporciona una medida cuantitativa de la información contenida en una fuente, sino que también establece un límite teórico superior para la eficiencia de cualquier compresión sin pérdidas que intente codificar la salida de dicha fuente.

Codificación Shannon-Fano

Existe un trabajo similar al descrito por Shannon, que fue hecho de forma independiente por Robert Fano y publicado en [56]. El método de codificación conocido como Shannon-Fano aparece en los dos trabajos y es parte esencial de la mejora en los códigos de longitud variable existentes. Este método reúne técnicas para crear códigos de longitud variable que minimizan la cantidad promedio de bits necesarios para cada símbolo. Al analizar las probabilidades de cada símbolo, el algoritmo asigna códigos de manera que los símbolos más comunes tengan códigos más cortos, mientras que los menos comunes tienen códigos más largos. Estos códigos son únicos y forman un código de prefijo, lo que significa que ningún código es prefijo de otro, garantizando así su decodificación correcta. La validación suficiente de esta afirmación se aborda en el anexo del documento. El algoritmo funciona ordenando los símbolos según su probabilidad y dividiéndolos en dos grupos con probabilidades totales casi iguales. A los símbolos de un grupo se les asigna un código que comienza con 0 y a los del otro grupo un código que empieza con 1. Luego, cada grupo se subdivide repetidamente de la misma manera, asignando bits adicionales según la probabilidad, hasta que no queden más subdivisiones. Sin embargo, aunque este método permite interpretar los mensajes claramente, no siempre garantiza la codificación más eficiente, lo que limita su efectividad.

Ejemplo: Codificación Shannon-Fano

Se tienen los símbolos A, B, C y D con las siguientes probabilidades asociadas:

Símbolo	Probabilidad
A	0.4
B	0.3
C	0.2
D	0.1

El algoritmo Shannon-Fano funciona de la siguiente manera:

1. Ordena los símbolos según su probabilidad en orden descendente: A, B, C, D.
2. Divide los símbolos en dos grupos de probabilidades lo más equilibradas posible:

Grupo	Probabilidad
A, B	0.7
C, D	0.3

3. Se asigna el bit 0 al primer grupo y el bit 1 al segundo grupo.
4. Subdivide cada grupo de manera recursiva:

   En el grupo 1 (A, B), se asigna 0 a A y 1 a B.

   En el grupo 2 (C, D), se asigna 0 a C y 1 a D.
5. Esto produce las siguientes codificaciones:

Símbolo	Código Shannon-Fano	Probabilidad
A	00	0.4
B	01	0.3
C	10	0.2
D	11	0.1

Como resultado, los símbolos con mayor probabilidad tienen códigos más cortos, lo que disminuye la cantidad de bits necesarios para la codificación. Dibujando el árbol de la manera habitual en teoría de grafos, con la raíz arriba y las hojas abajo, es más fácil de observar el resultado, en la figura siguiente se ilustra el árbol de codificación para este ejemplo:

Método Huffman

La codificación Huffman utiliza árboles binarios para asignar códigos más cortos a los símbolos con mayor frecuencia de aparición. Es particularmente eficaz cuando las distribuciones de probabilidad son desiguales, como en archivos de texto con caracteres de uso común. Aunque garantiza una compresión óptima para un solo símbolo, no aprovecha patrones repetidos entre ellos, limitando su eficacia en algunos casos. Un ejemplo práctico es

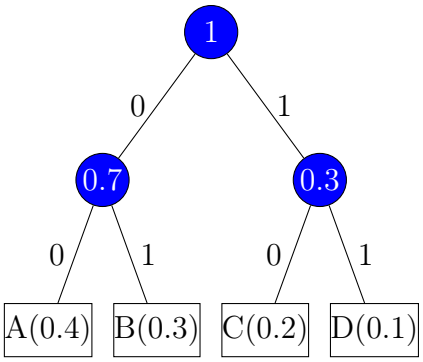


Figura 2.1: Árbol de codificación Shannon-Fano, elaboración propia.

su integración en la compresión de texto en archivos ZIP [57]. Esta codificación es similar a la codificación Shannon-Fano, la principal diferencia recae en que se ha demostrado que Huffman siempre produce la codificación de prefijo óptima, mientras que Shannon-Fano en algunas situaciones puede ser ligeramente menos eficiente. Este método se consideró inicialmente para formar parte del diseño, pero se optó por crear una arquitectura basada en teorías de los algoritmos y no limitar el trabajo a solo copiar e implementar un algoritmo.

2.2. Métodos de diccionario

En la compresión basada en diccionarios, se seleccionan secuencias de símbolos que luego se codifican como tokens utilizando un diccionario predefinido. La eficacia de la compresión depende de la calidad de este modelo. Este diccionario puede ser estático o dinámico: el primero es inmutable y ocasionalmente permite la adición, pero no la eliminación de secuencias, mientras que el segundo se ajusta continuamente según las secuencias encontradas en la corriente de datos, facilitando tanto la adición como la eliminación de entradas a medida que se procesan nuevos datos. En términos simples, un compresor basado en diccionario busca patrones repetitivos en la cadena de texto. Cuando encuentra estos patrones, los sustituye por códigos más cortos, lo que reduce el tamaño del archivo original. Idealmente, puede comprimir una cadena de n símbolos hasta aproximadamente nH bits. La letra H representa la entropía, en este caso del conjunto n de símbolos. La entropía, en este contexto, es una medida de la incertidumbre o la información promedio por símbolo en la cadena. Es importante mencionar que los compresores basados en diccionario se comportan como codificadores de entropía, son más eficientes cuando comprimen archivos grandes. Para aplicaciones prácticas, como la compresión de archivos comunes como texto, imágenes o datos de audio, estos compresores en general ofrecen resultados dentro de los parámetros adecuados, por lo tanto, gracias a su sencillez de funcionamiento y resultados son muy utilizados. Algunos de los algoritmos más conocidos de este tipo, son los basados en LZ, los cuales se revisaron para el presente trabajo; teniendo en cuenta su objetivo, ventajas y desventajas, se presentan brevemente los más relevantes.

2.2.1. Algoritmos de codificación LZ

Existen diversos algoritmos de compresión basados en diccionarios, entre ellos, los derivados del trabajo propuesto en 1977 por Jacob Ziv y Abraham Lempel, con los que se inició una nueva rama en compresión [11]. El fundamento de estos métodos consiste en utilizar una porción del texto procesado y crear un diccionario dinámico con esos datos. El compresor utiliza una memoria intermedia, o "buffer", denominada "ventana deslizante"para la cadena de entrada. Esta ventana, por lo general, desplaza los datos de derecha a izquierda conforme se codifican los símbolos, permitiendo así la identificación de patrones recurrentes en la entrada.

Método LZ77

El método LZ77 se basa en la utilización de una ventana deslizante para identificar patrones repetidos dentro de los datos. Este enfoque permite reemplazar secuencias repetidas con referencias a su posición y longitud dentro de un buffer. Es ampliamente utilizado en formatos como ZIP y GZIP debido a su capacidad para manejar datos con redundancia local significativa. Sin embargo, requiere un buffer de búsqueda y puede volverse ineficiente cuando los patrones repetidos están muy espaciados. Por ejemplo, en el caso de archivos de texto con frases recurrentes, LZ77 logra una compresión eficiente [58], más adelante se aborda a detalle este método, ya que se eligió para ser la base de la arquitectura diseñada.

Método LZ78

El método LZ78, una extensión del LZ77, construye un diccionario dinámico de patrones observados, asignando un índice único a cada nueva secuencia detectada. Esta técnica es útil para datos con redundancia global, como documentos extensos con múltiples ocurrencias de términos específicos. Una de sus principales ventajas es la eliminación de la necesidad de un buffer de búsqueda continuo, pero introduce la sobrecarga de administrar el diccionario. Un caso práctico es su uso en el algoritmo GIF para comprimir imágenes de baja complejidad [59], cabe resaltar que LZ78 no se consideró adecuado, debido a su patente que data de 1984 y permanece activa hasta el 2028 [60].

Método Deflate

Deflate combina LZ77 y codificación Huffman para lograr una alta eficiencia en la compresión. Este método primero identifica patrones repetidos con LZ77 y luego codifica las secuencias resultantes utilizando un esquema Huffman, que asigna códigos más cortos a los patrones frecuentes. Es empleado en formatos como PNG, donde la pérdida de información es inaceptable. Aunque ofrece una alta tasa de compresión, su implementación es más compleja y requiere un mayor poder computacional [61]. Al igual que el método anterior, se tuvo la oportunidad de revisar a detalle el funcionamiento y se observó que se perdía el objetivo de estudio al remitir el trabajo solo a replicar un algoritmo.

Método Codificación Aritmética

El último método presentado es la codificación aritmética, la cual asigna rangos de probabilidad a secuencias de símbolos, representando todo un mensaje con un único número dentro del rango acumulativo. Este método ofrece una compresión más cercana al límite teórico de Shannon que Huffman, especialmente para datos con símbolos altamente correlacionados. Sin embargo, su implementación es más compleja y puede ser más lenta. Es utilizada en estándares como H.264 para la compresión de video [62].

2.2.2. Un ejemplo de compresión

Teniendo en cuenta diversos formatos digitales del día a día, es evidente que en la mayoría de ellos se realiza de una u otra forma algún tipo de compresión para almacenar de forma eficiente la información, tomando de ejemplo el formato PDF (Portable Document Format en inglés, 'formato de documento portátil'). La compresión de archivos PDF es un proceso que combina diversos algoritmos de compresión para minimizar el tamaño del archivo, manteniendo su contenido legible y funcional. Los métodos principales utilizados en este formato son:

- **Compresión de imágenes**: El formato PDF utiliza métodos de compresión tanto con pérdida como sin pérdida, dependiendo de las configuraciones y el propósito del archivo.

  1. Compresión con pérdida (JPEG): Se utiliza para imágenes con escala de grises o color, internamente en general se utilizan matrices para reducir el tamaño eliminando detalles que se pueden considerar redundantes, depende de la configuración se reduce en mayor o menor cantidad la calidad del resultado.
  2. Compresión sin pérdida (Flate/PNG): Se basa en el algoritmo DEFLATE, pero a diferencia de GIF, se elimina todo algoritmo que está protegido por una patente, se comprime la imagen en formato monocromático o de gráficos vectoriales, sin perder calidad en los datos [63, 64].

- **Codificación de texto**: El texto en un PDF es generalmente codificado usando Flate, mientras que el texto en general puede ser comprimido utilizando algoritmos como JBIG2. Ya que permite el agrupamiento de caracteres similares para reducir el almacenamiento sin comprometer significativamente la legibilidad [65, 66].

- **Estructura del documento**: Los archivos PDF emplean un almacenamiento segmentado para contenido estructurado. Esto incluye la compresión de los datos individuales, como objetos y referencias cruzadas. Además, de otras medidas para disminuir las redundancias [67].

- **Compresión de metadatos y fuentes**: Los metadatos y las fuentes utilizadas en los archivos son comprimidas con Flate o eliminados parcialmente en algunas configuraciones de compresión. Los subconjuntos de fuentes (subsetting) también ayudan a reducir el tamaño, almacenando únicamente los caracteres utilizados en el documento [68, 69].

2.3. Computación en paralelo

En el diseño de arquitecturas de hardware se requiere un enfoque que tenga principalmente en cuenta los fundamentos matemáticos, teorías de compresión de datos y técnicas especializadas de procesamiento en paralelo. Este último elemento, el paralelismo, se utiliza en este trabajo para buscar aprovechar los recursos disponibles para alcanzar un alto rendimiento en la arquitectura. Se aborda brevemente el tema, considerando [70]. El paralelismo consiste en dividir una tarea en subtareas independientes que pueden ejecutarse en simultáneo por diferentes elementos de procesamiento. Esta técnica disminuye el tiempo de ejecución, mejorando la escalabilidad y eficiencia energética de la arquitectura. Ya que permitirá comparar múltiples cadenas al mismo tiempo, lo que acelera el proceso de búsqueda y codificación.

2.3.1. Importancia del paralelismo

El paralelismo en la actualidad es un componente fundamental en la computación, debido al estancamiento en el crecimiento de la frecuencia de reloj de los procesadores tradicionales. En lugar de depender exclusivamente de procesadores más rápidos, las arquitecturas actuales se apoyan en núcleos múltiples y aceleradores como las GPUs para realizar tareas en paralelo. Por ejemplo, al realizar la edición de una imagen, el ajuste del brillo en cada píxel puede llevarse a cabo en paralelo, procesando múltiples píxeles simultáneamente en lugar de uno por uno. Este enfoque ha demostrado ser crucial en aplicaciones como compresión de datos en tiempo real, simulaciones físicas y procesamiento masivo de información.

2.3.2. Tipos de paralelismo

Existen diferentes niveles de paralelismo que se pueden aprovechar en el diseño de sistemas:

- **Paralelismo a nivel de datos (DLP):** Procesa múltiples datos aplicando la misma operación de manera simultánea. Esto es común en aplicaciones gráficas y algoritmos de compresión, como la búsqueda de coincidencias en algoritmos LZ77.
- **Paralelismo a nivel de instrucciones (ILP):** Permite reordenar y ejecutar varias instrucciones de manera paralela dentro de un solo núcleo del procesador.
- El **paralelismo a nivel de solicitudes (RLP)** se aplica en sistemas como servidores web, donde múltiples peticiones de usuarios se procesan simultáneamente para mejorar la respuesta y escalabilidad del sistema.
- **Paralelismo a nivel de tareas (TLP):** Distribuye tareas independientes entre diferentes núcleos de procesamiento, lo que resulta esencial en sistemas multiprocesador.

2.3.3. Ventajas y retos

Uno de los beneficios más evidentes de la computación en paralelo es la reducción de tiempo de ejecución, esto es beneficioso en aplicaciones que pueden dividirse fácilmente en subtareas independientes. Además, esta técnica permite manejar una mayor carga de trabajo si se agregan recursos de hardware, lo que mejora la escalabilidad del sistema. Sin embargo, al momento de diseñar arquitecturas en paralelo se deben tener en cuenta los problemas que surgen con ello. Se deben sincronizar las subtareas para evitar inconsistencias, en especial cuando comparten datos. También se debe considerar el balance de la carga de trabajo entre las unidades de procesamiento, ya que una distribución dispar puede limitar el rendimiento global del sistema. Adicionalmente, el diseño y desarrollo de algoritmos paralelos suelen ser más complejos que sus contrapartes secuenciales.

2.3.4. Paralelismo y la taxonomía de Flynn

El paralelismo en computación se clasifica ampliamente según los modelos de ejecución y las estructuras de procesamiento que utiliza. Una de las herramientas más reconocidas para esta clasificación es la **Taxonomía de Flynn**, propuesta por Michael J. Flynn en 1966 [71]. Este modelo organiza las arquitecturas en cuatro categorías basadas en el número de instrucciones y de datos que pueden procesar simultáneamente.

Categorías de la taxonomía de Flynn

1. **SISD (Single Instruction, Single Data, en inglés):** Representa la arquitectura secuencial clásica, donde una única unidad de control ejecuta una instrucción sobre un único flujo de datos en un momento dado. *Ejemplo:* Procesadores tradicionales como los primeros Intel 8086.

2. **SIMD (Single Instruction, Multiple Data, en inglés):** Permite ejecutar una misma instrucción simultáneamente sobre múltiples conjuntos de datos. Es ideal para aplicaciones con gran paralelismo a nivel de datos, como procesamiento de imágenes o gráficos. *Ejemplo:* GPUs modernas o extensiones como Intel AVX.

3. **MISD (Multiple Instruction, Single Data, en inglés):** Aunque rara vez implementada, esta categoría describe arquitecturas donde múltiples flujos de instrucciones operan sobre un único flujo de datos. Se utiliza en casos especializados como sistemas redundantes para tolerancia a fallos. *Ejemplo:* Los sistemas de control de un avión.

4. **MIMD (Multiple Instruction, Multiple Data, en inglés):** Soporta múltiples flujos de instrucciones ejecutándose en paralelo sobre múltiples flujos de datos. Es común en sistemas multiprocesador y clústeres. *Ejemplo:* Supercomputadoras y procesadores multinúcleo como los modernos Intel Xeon o AMD EPYC.

2.3.5. Importancia de la taxonomía de Flynn

Es una referencia que ayuda a clasificar de forma sencilla las arquitecturas de computadoras, puede ser de ayuda al diseñar arquitecturas en paralelo. Cada categoría define la

forma en que se procesan los datos y como se hace y no es que alguna sea mejor que otra, depende de la tarea que se pretende atacar para considerar una u otra técnica. Por ejemplo, las arquitecturas SIMD son eficientes en problemas con gran homogeneidad en los datos, mientras que MIMD es más flexible y capaz de manejar tareas heterogéneas en paralelo. Considerando esta guía para el desarrollo de la arquitectura de compresión de datos, las arquitecturas SIMD pueden ser de ayuda en operaciones repetitivas como la comparación de cadenas, mientras que MIMD facilita la implementación de algoritmos complejos que combinan múltiples etapas de procesamiento.

2.3.6. Aplicaciones relevantes

En la actualidad existen por doquier ejemplos que se pueden categorizar de acuerdo con la taxonomía de Flynn:

- En procesamiento gráfico, las GPUs utilizan en su mayoría un modelo SIMD para realizar operaciones en simultáneo para el manejo de los pixeles.
- Las supercomputadoras, basadas en MIMD, son utilizadas para simulaciones que requieren procesar muchos datos, como modelos climáticos o análisis genómicos.
- Los sistemas embebidos pueden implementar variantes de MISD para generar redundancia en entornos críticos.

2.3.7. Aplicación en arquitecturas de compresión

En el contexto de las arquitecturas de compresión de datos, el paralelismo permite optimizar operaciones críticas como la búsqueda de cadenas repetidas y la codificación eficiente. Por ejemplo, mediante una matriz sistólica, es posible realizar comparaciones de datos en paralelo, reduciendo significativamente la latencia del sistema. Esta técnica se consideró particularmente relevante para implementar algoritmos como LZ77 en hardware. Se aborda a detalle en la siguiente sección el entendimiento obtenido sobre el tema de matrices sistólicas.

2.4. Matrices sistólicas en la arquitectura de hardware

Las matrices sistólicas son una de las estructuras de mayor relevancia en el diseño de arquitecturas de hardware en paralelo, debido a su capacidad para realizar cálculos complejos de manera eficiente mediante la sincronización entre múltiples elementos de procesamiento. Este modelo, introducido por Kung y Leiserson en 1978 [1], combina la división de los datos en pequeños segmentos y su procesamiento en paralelo, logrando resolver problemas computacionales intensivos con la utilización de matrices. El término "sistólico"proviene de la analogía con el sistema circulatorio, donde el flujo de datos a través de los elementos de procesamiento se asemeja al flujo sanguíneo a través de las arterias y venas. En una matriz sistólica, los datos fluyen de manera sincronizada entre celdas vecinas en ciclos de reloj predefinidos. Cada elemento de procesamiento realiza una

operación local sobre los datos que recibe, los actualiza y los reenvía a la siguiente celda, siguiendo un patrón regular y predecible, en la siguiente figura se aprecia este esquema.

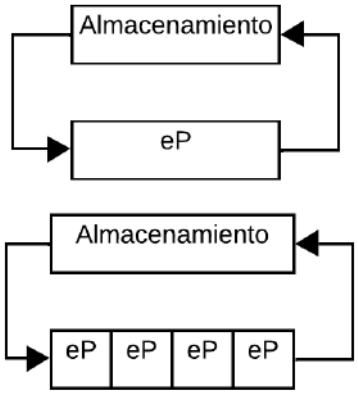


Figura 2.2: Diagrama de matriz sistólica, basado en [1].

Estas matrices ofrecen una ventaja significativa en términos de rendimiento, ya que aprovechan el paralelismo ofrecido por el hardware. Por ejemplo, en el caso de la multiplicación de matrices, cada celda en la matriz sistólica puede calcular productos parciales de manera simultánea, propagando los resultados hacia las celdas cercanas. Este enfoque reduce la complejidad de control y minimiza la latencia, ya que todas las operaciones se realizan de forma local en cada elemento de procesamiento y sincronizadas entre ellos.

Una analogía a las matrices sistólicas es la cadena de producción en una fábrica. En este caso, cada estación realiza una operación específica sobre el producto, como ensamblaje, pintura o embalaje, y luego lo pasa a la siguiente estación. Todas las estaciones trabajan de manera simultánea, pero cada una tiene un trabajo independiente y sincronizado. De manera similar, en una matriz sistólica, cada celda realiza cálculos específicos mientras recibe y envía datos, maximizando la eficiencia del sistema.

Aplicadas a compresión de datos, las matrices sistólicas son útiles en la implementación de algoritmos LZ, ya que las operaciones de búsqueda y comparación se pueden dividir en procesos que se ejecuten en paralelo. Por ejemplo, en la propuesta basada en LZ77, se utilizará la matriz sistólica para asignar a cada elemento de procesamiento la comparación de una parte específica del texto con el diccionario, reduciendo drásticamente el tiempo necesario para encontrar coincidencias si se realizara de forma secuencial.

Además, la eficiencia de las matrices sistólicas destaca por su sencillez en el diseño de hardware. Al estar compuestas por elementos homogéneos con interconexiones regulares, estas estructuras son fáciles de seguir en su implementación y también entendibles para su escalamiento. Por esta razón, son ampliamente utilizadas en sistemas embebidos y dispositivos de procesamiento de señales, como filtros digitales y codificadores de video, donde se tiene limitado el espacio y energía. Las matrices sistólicas son una poderosa herramienta para diseñar arquitecturas de hardware, ya que combinan eficiencia, paralelismo y modularidad. Su capacidad para resolver problemas computacionales intensivos de manera simultánea hace que sean una opción ideal para aplicaciones como compresión de datos, donde se deben realizar muchas comparaciones y búsquedas. Basarse en este enfoque en el

desarrollo de la arquitectura, busca garantizar un alto rendimiento y un uso eficiente de los recursos disponibles.

2.4.1. Método seleccionado

Se eligió el método de compresión tomando en cuenta tanto las ventajas como desventajas de todo lo que rodea a los algoritmos estudiados, desde el tipo de datos en los que se desempeñan mejor, la forma en que se pueden optimizar, hasta las restricciones que se les imponen para su utilización. Considerando un modelo de acuerdo con la teoría de ventana deslizante, ya que se puede implementar de forma eficiente en hardware y mejorar su desempeño en la búsqueda de coincidencias sin necesidad de agregar complejidad computacional que no haría una gran diferencia en los resultados obtenidos.

En específico, se decidió realizar el diseño basado en el algoritmo LZ77, por su desempeño comprobado y estudiado desde su publicación por Ziv y Lempel [72]. El uso de una ventana deslizante en LZ77 permite encontrar coincidencias en cadenas de texto, logrando una compresión eficiente. Sin embargo, una ventaja adicional que es clave en este trabajo es la forma de procesar en paralelo los datos, haciendo uso de matrices sistólicas. Esta implementación logra adaptar partes críticas del algoritmo al hardware y aprovecharlo de forma eficiente. Disminuyendo el tiempo de compresión, ya que se busca y compara de forma simultánea en varias partes del texto a la vez.

El diseño seleccionado también se puede adaptar a arquitecturas para diversas tareas específicas. En casos donde el consumo energético es un factor que se debe considerar, la arquitectura propuesta ofrece beneficios importantes. Investigaciones recientes respaldan esta afirmación al demostrar que este enfoque puede disminuir el uso de memoria y tiempo de transferencia de datos en dispositivos embebidos [73].

Al utilizar el paralelismo a nivel de datos (DLP) al ejecutar operaciones repetitivas de manera simultánea, el método seleccionado no solo mejora el rendimiento, sino que también permite aprovechar arquitecturas como SIMD, que son recomendadas para procesar datos en paralelo en aplicaciones como la búsqueda de coincidencias dentro del algoritmo LZ77. Esto confirma la viabilidad del modelo propuesto y las ventajas que presenta.

Capítulo 3

Análisis

Se consideró la metodología como parte de la estructura de este trabajo; se realizó primero el desarrollo del modelo general, seguido de la construcción de la lista de funcionalidades, aunque para ello se debe tener claro los aspectos que se deben analizar para la construcción de la arquitectura. Se analiza el algoritmo en el que se basa la arquitectura, así como el paralelismo y el hardware de pruebas utilizado. Todo ello para poder describir una propuesta fundamentada y que sea factible de diseñar.

3.1. Algoritmo LZ77 a detalle

El algoritmo, conocido actualmente como LZ77 en honor a las iniciales de sus creadores, emplea esta ventana deslizante como un diccionario que permite visualizar la entrada tal como se está codificando. La ventana deslizante almacena los últimos n símbolos emitidos por la fuente, facilitando la identificación de secuencias en la entrada que coinciden con las presentes en la ventana deslizante. La ventana se conforma por el apartado izquierdo, llamado buffer de búsqueda, que actúa como el diccionario actual, donde almacena los últimos símbolos codificados. Por otro lado, la parte derecha es el buffer de lectura anticipada, que contiene el texto que aún no se ha leído. A continuación, se muestra la ventana deslizante sobre un texto de ejemplo. La línea vertical entre los símbolos `"l"` y `"o"` representa la división entre los dos buffers. Los símbolos en la ventana izquierda ya han sido comprimidos, mientras que el texto en la ventana derecha todavía necesita ser procesado.

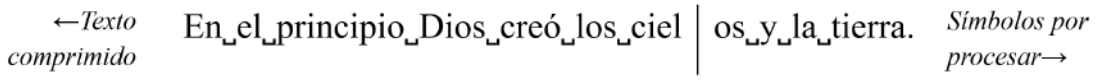


Figura 3.1: Ejemplo de ventana deslizante, elaboración propia.

El codificador revisa el buffer de búsqueda comenzando desde el final hacia el principio (leyendo en orden inverso a los humanos) encontrando las coincidencias con el símbolo actual en el buffer de lectura anticipada. Encuentra una coincidencia con la letra 'o' de la palabra "los". Esta 'o' se encuentra a 7 posiciones de la primera 'o'. El siguiente paso es

buscar concordancias con la mayor cantidad posible de símbolos adyacentes a la derecha. Al revisar el ejemplo, se verifica que coinciden tres caracteres `"os_"`, siendo entonces la longitud de 3. El compresor continúa leyendo el texto en orden inverso en búsqueda de coincidencias. Utilizando la coincidencia más larga, en caso de que todas las coincidencias tengan la misma longitud, se opta por la última de ellas, y con esa información prepara el token. Escogiendo la última coincidencia se simplifica el proceso de descompresión, ya que solo tiene que mantener la dirección de la última cadena encontrada, aunque a costa de tener desplazamientos más largos. El token producido es en realidad una triada que se crea al encontrar una coincidencia; los elementos de la triada son:

1. Desplazamiento en la ventana donde se encontró la coincidencia.

2. Longitud de la coincidencia.

3. El siguiente símbolo después de la frase actual.

Teniendo en cuenta los elementos que lo conforman, en el ejemplo anterior se obtendría el token $(7, 3, y)$. La ausencia de una coincidencia da como resultado una triada de $(0, 0, C(s))$, donde $C(s)$ es la palabra clave para el símbolo S. Ya que la ventana tiene una longitud finita, las repeticiones en la entrada con un periodo más grande que n no pueden ser detectadas y comprimidas por LZ77. El diccionario LZ77 incluye todas las sub-cadenas y símbolos individuales de una cadena dentro de una ventana deslizante. Esta técnica ha evolucionado con variaciones que han dado lugar a los algoritmos de codificación LZ (Lempel-Ziv). Estos algoritmos son herramientas fundamentales en la compresión de datos, ya que permiten reducir el tamaño de archivos sin perder información, aprovechando patrones recurrentes en los datos [74]. LZ77, LZSS, LZ78 y LZW son las variaciones más comunes de los algoritmos de codificación LZ, Deflate por su parte, utiliza compresión LZ77.

LZ77 procesa datos de izquierda a derecha, insertando cada cadena en el diccionario. Por lo general, el diccionario está limitado por la memoria disponible, por lo que se utiliza un diccionario deslizante. Un diccionario deslizante mantiene una lista de las cadenas utilizadas más recientemente. Si cierta cadena no está en el buffer de búsqueda, esta se toma como una secuencia literal de bytes. Si se encuentra una coincidencia, la cadena se reemplaza por un puntero a la cadena coincidente en forma de par distancia-longitud. El par distancia-longitud se compone de dos partes: la distancia desde la posición actual hasta el comienzo de la coincidencia y la longitud de la coincidencia. La información recién comprimida también está precedida por un bit de bandera para distinguir los literales de los pares distancia-longitud. Los bits de bandera se pueden empaquetar juntos en un byte para conservar memoria. El acceso eficiente al diccionario es clave, por lo que la mayoría de los programas, incluido GZIP, implementan el diccionario mediante funciones hash. Las implementaciones de hardware de los algoritmos de codificación LZ77 suelen utilizar memorias direccionables de contenido (CAM) o matrices sistólicas. aunque las CAM generan un alto rendimiento, son costosas en términos de requisitos de hardware. Por otro lado, las matrices sistólicas requieren menos hardware y ofrecen una capacidad de prueba mejorada. Ambas tecnologías tienden a ser complejas y dependen de las características proporcionadas por la arquitectura del hardware.

3.1.1. Complejidad computacional del algoritmo LZ77

El algoritmo LZ77, diseñado por Ziv y Lempel en 1977 [72], utiliza una ventana deslizante para identificar patrones repetitivos en un flujo de datos, reemplazando las cadenas redundantes por referencias compactas a posiciones anteriores. Este enfoque permite una compresión eficiente sin pérdida de información, pero su implementación presenta diferentes complejidades computacionales dependiendo de si se ejecuta en software o hardware.

Complejidad computacional en software

En una implementación básica de LZ77 en software, cada símbolo del texto de entrada se compara con las cadenas almacenadas en la ventana deslizante para encontrar la coincidencia más larga. Considerando la complejidad computacional de este algoritmo, es de $O(n \cdot m)$, donde:

- n: Longitud del texto de entrada.
- m: Ventana deslizante, tamaño.

Por ejemplo, con $n = 1,000,000$ caracteres y $m = 32,768$ caracteres (tamaño típico de ventana en compresores como gzip), el número máximo de operaciones podría alcanzar aproximadamente 3.28×10^{10}. Se puede optimizar el cálculo en parte, utilizando estructuras de datos como lo son, tablas hash o árboles de subfijos, estos pueden reducir la complejidad de búsqueda promedio a solo $O(n)$. aunque se debe tener en cuenta que estas optimizaciones hacen que se consuma más memoria y puede ser fuente de cuellos de botella al procesar grandes volúmenes de datos.

En sistemas que tienen que ejecutarse con recursos limitados, el caso de los dispositivos móviles, estas optimizaciones por lo general son inviables, ya que se tiene memoria y energía restringida. También se debe considerar la latencia que agrega el acceso a memoria y que se debe procesar de forma secuencial, resultando en una afectación en el tiempo total de ejecución, limitando la optimización y escalabilidad del algoritmo solo con soluciones vía software.

Implementación en hardware y su complejidad

En hardware, el algoritmo LZ77 puede beneficiarse enormemente de la paralelización. Usando una arquitectura basada en matrices sistólicas, cada celda de la matriz puede asignarse a la comparación de sub-cadenas específicas dentro de la ventana deslizante. Esto permite realizar múltiples operaciones de búsqueda y comparación simultáneamente. La complejidad práctica de esta implementación se reduce a $O(n)$, ya que cada símbolo del texto de entrada se procesa en un ciclo de reloj.

Por ejemplo, un FPGA como el Artix-7 XC7A200T puede integrar hasta 740 bloques DSP y 215,360 celdas lógicas. En una configuración típica, un diseño de matriz sistólica para LZ77 podría dividir la ventana deslizante en 256 bloques, con cada bloque manejado por un conjunto de celdas. Si el FPGA opera a una frecuencia interna de 200 MHz, cada símbolo puede procesarse en aproximadamente 5 ns, permitiendo una tasa de procesamiento cercana a $200\,\mathrm{MB\,s^{-1}}$. Esto sigue representando una mejora significativa respecto a implementaciones en software, que suelen alcanzar tasas promedio cercanas a $50\,\mathrm{MB\,s^{-1}}$.

Comparación y cambios en la complejidad

El cambio en la complejidad computacional entre software y hardware es notable. En software, el procesamiento secuencial y las dependencias en estructuras auxiliares incrementan tanto el tiempo de ejecución como la utilización de recursos. En hardware, la paralelización reduce la complejidad efectiva, ya que las operaciones más intensivas, como la comparación y la búsqueda, se distribuyen en múltiples celdas que operan de manera concurrente.

Adicionalmente, el uso de memoria en hardware está optimizado para minimizar accesos redundantes. ya que se tienen bloques de RAM distribuidos localmente para ayudar a tener un acceso rápido y eficiente, mientras que en software la memoria puede convertirse en un cuello de botella.

Ejemplo de complejidad entre software y hardware

Un ejemplo práctico de complejidad alta puede observarse al procesar un archivo de 1 GB utilizando LZ77 con una ventana de 64 kB. En software, se necesitarían aproximadamente 6.4×10^{10} comparaciones, lo que llevaría varios minutos en un procesador típico de 3 GHz. Sin embargo, en hardware, el mismo archivo podría llegar a procesarse en menor tiempo, gracias a la paralelización y a la eliminación de cuellos de botella en memoria.

3.1.2. Impacto en el diseño de hardware

Diseñar hardware para implementar LZ77 implica equilibrar el uso de recursos con el rendimiento deseado. Por ejemplo, el Artix-7 XC7A200T puede proporcionar suficiente capacidad lógica y bloques DSP para manejar ventanas deslizantes de hasta 128 kB con baja latencia: más adelante se detallan las características de la tarjeta de pruebas utilizada. La escalabilidad del diseño también es un factor crítico, ya que arquitecturas más grandes pueden integrarse fácilmente en hardware reconfigurable para manejar mayores volúmenes de datos sin comprometer la velocidad de procesamiento. Mientras que el algoritmo LZ77 mantiene su lógica fundamental independientemente del entorno, la implementación en hardware transforma su complejidad computacional. Esto permite alcanzar un rendimiento significativamente superior, lo que lo hace adecuado para aplicaciones de alta demanda como la compresión en tiempo real y el procesamiento masivo de datos. Se realiza la siguiente propuesta de entradas y resultados que debe tener la arquitectura de compresión.

3.2. Descripción del hardware empleado

La tarjeta de desarrollo AX7A200, basada en el FPGA AMD Artix-7 XC7A200T, fue seleccionada para este proyecto por su combinación de soporte a largo plazo, flexibilidad modular y capacidades técnicas avanzadas, se muestra en la figura 3.2 . AMD garantiza soporte oficial para la familia Artix-7 hasta el año 2035, lo que la convierte en una solución ideal por la cantidad de personas utilizándola actualmente y el soporte oficial que tiene en el presente tanto en la herramienta de desarrollo, como en foros de ayuda en caso de requerirse, a diferencia de otras tarjetas de desarrollo que se consideraron [75].

Figura 3.2: FPGA utilizada, tomado de [2]

3.2.1. Características de la tarjeta AX7A200

La tarjeta AX7A200 integra el módulo SoM AC7A200. La placa incluye una amplia gama de conectores y componentes, como PCIe 2.0, dos ranuras SFP, puertos HDMI de entrada y salida, un conector JTAG para programación, una ranura para tarjetas SD, y dos conectores de expansión de 40 pines. Estas características hacen que sea apta para una variedad de aplicaciones, incluyendo:

1. Procesamiento de datos intensivo, gracias a su soporte para comunicaciones de alta velocidad a través de PCIe.

2. Procesamiento y transmisión de video e imágenes con entradas y salidas HDMI.

3. Transmisión de datos Ethernet mediante fibra óptica utilizando las ranuras SFP.

La capacidad de expansión de la tarjeta permite integrar módulos adicionales, como sistemas de adquisición de datos AD/DA, cámaras para visión artificial y pantallas LCD, lo que incrementa su versatilidad. Esta flexibilidad asegura que el hardware pueda adaptarse fácilmente a nuevas necesidades y escalar en complejidad sin reemplazar el sistema base [76].

3.2.2. Especificaciones técnicas del FPGA Artix-7 XC7A200T

El FPGA Artix-7 XC7A200T, núcleo de la tarjeta, ofrece especificaciones destacables en su categoría, manteniendo un precio coherente:

1. Capacidad lógica: 215,360 celdas lógicas, suficientes para implementar algoritmos complejos, desde compresión de datos hasta procesamiento de señales.

2. Frecuencia interna: Hasta 200 MHz, lo que proporciona una alta capacidad de procesamiento para aplicaciones intensivas en cálculo.

3. Memoria integrada: 13.14 Mb de memoria de bloque, ideal para almacenar datos temporales durante el procesamiento.

4. Capacidad de DSP: 740 bloques DSP dedicados, optimizados para cálculos matemáticos intensivos como multiplicaciones matriciales y filtros digitales.

5. Interfaz transceptora: Compatible con transceptores de alta velocidad, alcanzando tasas de datos de hasta 6.6 Gbps, crucial para aplicaciones de comunicación.

3.2.3. Consumo energético y rendimiento térmico

Una de las características a destacar de la tarjeta de desarrollo Artix 7, es su eficiencia energética, alcanzando un equilibrio entre alto rendimiento y bajo consumo de potencia. De acuerdo con sus especificaciones, el consumo promedio es entre 5 y 20 W, dependiendo de la complejidad de la aplicación y los recursos utilizados del FPGA. Este bajo consumo lo hace ideal para aplicaciones en sistemás embebidas o móviles, donde se debe considerar la eficiencia energética. La tarjeta está diseñada para operar en un rango de temperaturas industriales, soportando entre -40 °C y 85 °C, lo que la hace apta para entornos cambiantes y hasta aplicaciones al aire libre.

3.2.4. Velocidad de operación y latencia

La velocidad a la cual opera la tarjeta de desarrollo permite realizar cálculos en paralelo con baja latencia. Esto es útil en tareas específicas como la compresión de datos, donde las operaciones de búsqueda y comparación pueden realizarse de forma paralela eficientemente. Al implementar algoritmos en hardware, como el caso de matrices sistólicas, la latencia puede reducirse aún más, ya que se busca mejorar el rendimiento de la arquitectura al tener el flujo de datos sincronizado entre los elementos de procesamiento.

3.2.5. Escalabilidad y aplicaciones

La tarjeta de desarrollo elegida facilita la escalabilidad, ya que permite agregar diferentes módulos de forma sencilla. Esto la convierte en una solución flexible para proyectos que requieren cambios constantes o mejoras a largo plazo. Entre sus aplicaciones destacan:

1. Procesamiento de señales digitales (DSP) en tiempo real.

2. Compresión y transmisión de datos en redes de alta velocidad.

3. Sistemás de visión artificial y procesamiento de imágenes en entornos industriales.

La tarjeta de desarrollo AX7A200, con el FPGA Artix-7 XC7A200T, ofrece una combinación equilibrada de rendimiento, eficiencia energética y flexibilidad. Su soporte a largo plazo y su diseño modular sirven para mantener relevancia para proyectos industriales y científicos en los próximos años.

3.3. Desarrollo de modelo general

En esta etapa se planeó a grandes rasgos el alcance que debe tener la arquitectura como un todo, así como su posible composición y formas viables de dividirlo en pequeñas funcionalidades que se pueden desarrollar.

Aunque en la actualidad se siguen desarrollando teorías y métodos de compresión, dependiendo del tipo de compresión que se utilice, es diferente el método en que se procesan los datos; pero el objetivo que siguen es el mismo, hacer llegar al canal de comunicación el resultado. Este proceso se muestra de forma general en la figura 3.3. Se tienen los da-

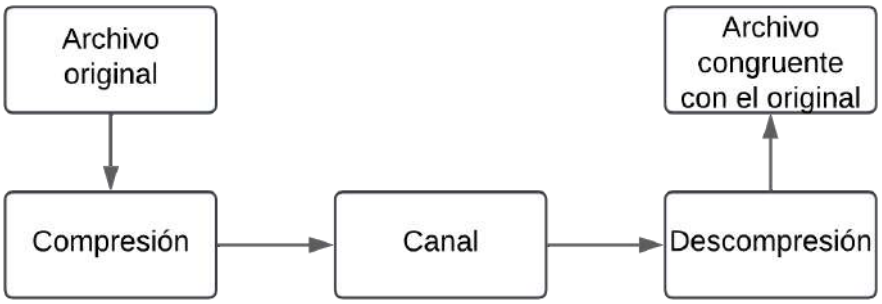


Figura 3.3: Diagrama de bloques básico de compresión, elaboración propia.

tos originales, se procesan mediante algoritmos de compresión y el resultado se almacena localmente o se realiza la transmisión del archivo (esto no compete al esquema de compresión, sino al manejo del mismo archivo por el sistema gobernante). Finalmente, cuando se requieren los datos originales, se emplean algoritmos de descompresión compatibles con los utilizados en compresión, obteniendo datos congruentes con los originales.

La arquitectura propuesta para comprimir se puede descomponer en tres grandes apartados, los cuales son:

1. Unidad de compresión: Diseñada como un módulo específico dentro del FPGA, se encarga de aplicar el algoritmo basado en LZ77 mediante una configuración en paralelo.

2. Pipeline para procesamiento continuo: El diseño utiliza un pipeline para garantizar un flujo constante de datos, maximizando el rendimiento en tareas de compresión y reduciendo la latencia.

3. Gestión de memoria: Se integran buffers locales para manejar fragmentos de datos y minimizar accesos redundantes a la memoria externa, mejorando la eficiencia energética y reduciendo los tiempos de espera.

Cada uno de ellos con diversas funcionalidades y alcance particular, lo cual permite tener las siguientes ventajas:

1. Arquitectura escalable: en principio bajo una misma tarjeta de desarrollo de acuerdo a sus capacidades soportadas, modificando el número de nodos utilizados en la matriz de procesamiento de datos implementada.

2. Modular: Se busca realizar la arquitectura mediante módulos, para poder realizar las adaptaciones o modificaciones necesarias que surjan en cada apartado sin necesidad de una reingeniería de toda la arquitectura, manteniendo, corrigiendo y mejorando el comportamiento de cada apartado sin afectar al resto.

3. Pruebas unitarias: Cada módulo realiza ciertas tareas que a los demás módulos no les compete conocer cómo se hacen. Con este enfoque, se logra realizar pruebas individuales sobre los datos que se especifica debe recibir y generar cada uno, para así poder corroborar el correcto funcionamiento de cada módulo de forma independiente.

4. Carencia de un sólo punto de fallo: permite que, en caso de que exista algún fallo en el prototipo, sólo sea necesaria la sustitución del componente que lo presenta, manteniendo al resto sin afectaciones.

Para poder construir la lista de funcionalidades que la arquitectura debe tener y como parte del análisis, se define primero cómo funciona el algoritmo en el cual se basa el trabajo.

3.3.1. Definición de casos de prueba

Para evaluar la arquitectura propuesta, se consideraron pruebas específicas para medir su funcionalidad, eficiencia y robustez bajo diferentes condiciones. Estas pruebas se llevarán a cabo utilizando archivos de compresión estándar ampliamente reconocidos, como los definidos por el *Canterbury Corpus* y el *Calgary Corpus*, que proporcionan datos representativos con diversas características estadísticas. A continuación, se detallan los casos de prueba organizados por métricas clave:

1. Tiempo de compresión

2. Tasa de compresión

3. Consumo energético

Tiempo de compresión

El tiempo de compresión del sistema se medirá utilizando temporizadores integrados en el entorno de pruebas del FPGA. Para ello, se cargarán archivos de prueba con tamaños desde 4Kb hasta 1Mb, seleccionados del *Canterbury Corpus* para garantizar una representación balanceada de datos repetitivos y aleatorios. Cada archivo será procesado con una ventana deslizante de 64 kB, de acuerdo con las especificaciones del diseño.

Los tiempos de inicio y finalización de la operación de compresión se registrarán con precisión de nanosegundos, tomando en cuenta la frecuencia de operación del FPGA, configurada a 200 MHz. Estos datos permitirán calcular la velocidad promedio de compresión en MB/s. Se espera que el sistema alcance velocidades superiores a 200 MB/s, basadas en la paralelización proporcionada por las matrices sistólicas y la capacidad lógica del Artix-7 XC7A200T [76].

Tasa de compresión

La tasa de compresión se evaluará comparando los tamaños de los archivos originales y comprimidos. Para cada archivo de entrada, el sistema generará un archivo comprimido cuya relación de tamaño será calculada como $R = \frac{\text{Tamaño original}}{\text{Tamaño comprimido}}$. Este análisis se realizará utilizando archivos con diferentes características, como: - Archivos altamente repetitivos (por ejemplo, `aaa.txt` del *Canterbury Corpus*). - Archivos de alta entropía, como datos generados aleatoriamente. - Mezclas intermedias, como textos con estructura semántica (`bible.txt` del *Calgary Corpus*).

Los resultados serán comparados con las tasas obtenidas por compresores LZ77 en software, como gzip y zlib, para validar la consistencia del sistema. Se espera que la eficiencia del hardware, basada en la búsqueda paralela de coincidencias, con tasas equivalentes o superiores, con una reducción significativa en el tiempo de procesamiento.

Consumo energético

El análisis se realizará empleando las herramientas especializadas de estimación y simulación de potencia disponibles en el entorno Vivado, así como parámetros técnicos derivados de las características del FPGA Artix-7 XC7A200T. Este análisis incluye tanto la potencia estática como la dinámica, siendo estas las principales contribuyentes al consumo total [77].

Para la estimación de consumo, se utilizará la herramienta *Report Power* de Vivado en la etapa *post-route*, donde ya se han definido los recursos de interconexión y las restricciones temporales del diseño. Esta herramienta permite analizar de manera precisa la actividad interna del circuito, ya que utiliza el archivo SAIF (*Switching Activity Interchange Format* en inglés), generado a partir de simulaciones representativas del sistema. Dichas simulaciones emplearán datos de entrada de *Canterbury Corpus*.

El archivo SAIF contendrá información detallada sobre la actividad de los nodos del circuito, incluyendo la probabilidad estática y la tasa de conmutación de señales. Esto garantiza que la estimación de consumo energético sea coherente con el caso de uso típico del sistema, donde se procesan flujos de datos de alta velocidad y se busca mantener una frecuencia de operación constante de 200 MHz.

Además, el análisis incluirá la evaluación de parámetros ambientales en la herramienta *Report Power*, como la temperatura de unión (*junction temperature*) y el flujo de aire. Se configurará una temperatura típica de operación de 60 °C y se asumirá un flujo de aire nulo (ya que no tendrá disipación activa). Los voltajes de alimentación se mantendrán en sus valores predeterminados, ajustados al estándar del Artix-7.

Como resultado se generará un reporte textual. Este informe contendrá un desglose detallado del consumo total, separando en componentes dinámicos y estáticos. La potencia dinámica se analizará en función de la actividad de las señales, mientras que la potencia estática estará influenciada principalmente por las características del proceso de fabricación del FPGA y las condiciones térmicas.

Este enfoque busca comprender el consumo energético teórico que la arquitectura propuesta tendrá, permitiendo identificar áreas potenciales de optimización en el diseño. Los resultados se utilizarán como referencia para futuras iteraciones, buscando que el sistema

sea eficiente desde el punto de vista energético.
Para poder construir la lista de funcionalidades que la arquitectura debe tener y como parte del análisis, se define primero cómo debe funcionar de acuerdo a sus requerimientos.

3.4. Especificación de requerimientos del sistema

Este apartado detalla los requerimientos del sistema de compresión de datos basado en hardware, considerando la especificación IEE 830 [78]. El objetivo es establecer requerimientos funcionales y no funcionales, así como criterios de verificación.

3.4.1. Requerimientos funcionales

La arquitectura de hardware propuesta debe cumplir con los siguientes requerimientos funcionales definidos específicamente para buscar tener un correcto y eficiente procesamiento de datos.

1. El sistema debe ser capaz de recibir flujos de entrada de hasta 100 MB/s.
2. El procesamiento de los datos debe realizarse en bloques, con soporte para escalabilidad que permita ampliar el tamaño del bloque según los requerimientos del sistema.
3. El sistema debe almacenar temporalmente los datos entrantes en buffers internos para garantizar la continuidad del flujo y prevenir pérdida de información.
4. La arquitectura debe implementar una matriz sistólica en hardware para acelerar las operaciones de coincidencia de cadenas. Cada elemento de procesamiento (eP) debe ser capaz de comparar un símbolo de entrada con una sub-cadena del diccionario.
5. El sistema debe generar referencias comprimidas en formato tipo distancia, longitud, basada en el estándar LZ77.
6. El sistema debe ser capaz de operar con datos provenientes de archivos de texto plano codificados en ASCII.
7. El sistema debe ser verificable por simulación, generando como salida la tasa de compresión lograda (relación entre tamaño original y comprimido) y el tiempo total de procesamiento.

3.4.2. Requerimientos no funcionales

1. El consumo energético total del sistema debe mantenerse dentro del rango de 5 W a 20 W, en función de la carga de trabajo, siendo eficiente en energía durante la operación continua.
2. El diseño debe considerar el uso eficiente de recursos internos del FPGA Artix-7 XC7A200T.

3. La disipación de potencia estática debe ser lo suficientemente baja como para permitir enfriamiento mediante disipación pasiva.

4. El sistema debe operar de forma estable dentro del rango térmico del FPGA ($-40\,^{\circ}\mathrm{C}$ a $85\,^{\circ}\mathrm{C}$), con pruebas de temperatura realizadas a $25\,^{\circ}\mathrm{C}$ bajo condiciones ambientales.

5. La arquitectura debe ser escalable, permitiendo parametrizar el número de eP en la matriz sistólica y el tamaño de ventana, sin exceder los recursos disponibles ni afectar la frecuencia de operación.

6. El sistema debe mantener compatibilidad con interfaces estándar ampliamente utilizadas (PCIe, Ethernet, USB), facilitando su integración en plataformas heterogéneas sin adaptaciones estructurales mayores.

7. En pruebas de simulación, deben generarse reportes automáticos que documenten el uso de LUTs, FFs, DSPs, y BRAMs, así como el consumo total estimado de energía y el margen térmico esperado.

8. En pruebas físicas realizadas sobre la tarjeta de desarrollo compatible con el XC7A200T, debe instrumentarse la medición del consumo energético promedio mediante un medidor de corriente y voltaje externo, registrando valores bajo carga típica y máxima.

3.5. Propuesta: Compresor con ventana deslizante

Con base en la teoría de ventana deslizante y una vez definidos los requerimientos a los que se debe someter la arquitectura. Se diseñó un algoritmo que comprima la información expresada en texto plano; de ello se obtendrán las operaciones básicas que debe realizar la arquitectura. Más adelante se consideran dichas operaciones para verificar la forma en que se puede mejorar el proceso implementando ciertas tareas mediante matriz sistólica. A continuación, se muestran los pasos y razonamientos surgidos para la propuesta. Una de las características de este método de compresión es que el nivel de compresión depende del diccionario utilizado, lo que significa que se debe seleccionar un diccionario adecuado. En algunas propuestas estudiadas se propone un diccionario en inglés, pero este puede adaptarse según los entornos probables del texto a comprimir. La propuesta consiste en tener un diccionario dinámico, que se ajusta dependiendo del texto a comprimir. Se tienen las siguientes características en la primera versión de la propuesta.

1. **Entradas:** Caracteres codificados en ASCII extendido (0 a 255 caracteres), utilizan 8 bits cada carácter.

2. **Diccionario:** 4096 bits (512 caracteres).

3. **Bloque de entrada:** 128 bits (16 caracteres).

4. **Proceso:** Recorre la entrada de bytes en búsqueda de la coincidencia más grande, comparándola con el diccionario (previamente almacenado).

5. **Salida:** Triada (Distancia, Longitud, Carácter).

6. **Especificaciones:**

   a) Distancia limitada a 256.

   b) Longitud limitada a 256.

   c) Carácter varía entre 0 y 255 (ASCII), utiliza 1 byte.

Ejemplo: Utilizando el diccionario: *estancia alrededor dormir elección*. Considerando al carácter _ como espacio en blanco para facilidad de lectura. Se muestra en las siguientes figuras tanto el texto a comprimir, considerando el primer bloque, como el diccionario propuesto solo para este ejemplo.

Figura 3.4: Texto de ejemplo, elaboración propia.

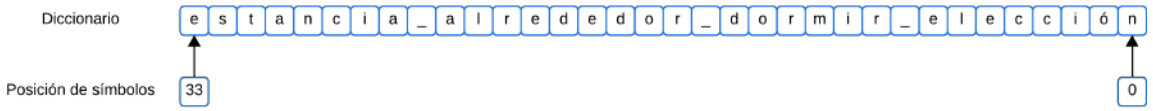


Figura 3.5: Diccionario propuesto, elaboración propia.

De acuerdo con la tabla ASCII extendida [79] se obtuvo la tabla de valores ASCII para el ejemplo propuesto (ver tabla 3.1), internamente la arquitectura realizará estas comparaciones. Para el ejemplo se seguirán utilizando letras.

Tabla 3.1: Valores ASCII de los caracteres de ejemplo, elaboración propia.

Carácter	Valor ASCII	Carácter	Valor ASCII	Carácter	Valor ASCII
E	69	e	101	l	108
s	115	s	115	_	32
t	116	t	116	e	101
a	97	a	97	s	115
_	32	c	99	t	116
e	101	i	105	a	97
s	115	ó	162	d	100
_	32	n	110	o	111
l	108	_	32	.	46
a	97	d	100		
_	32	e	101		

3.5.1. Ejemplo de comparaciones y salidas

Se considerará solo el primer bloque para esclarecer el algoritmo propuesto, se van obteniendo las acciones que se deben realizar, siendo la comparación entre cada símbolo del bloque de entrada y el símbolo del diccionario, se leen desde el byte menos significativo hasta el byte más significativo. Cabe aclarar que también el diccionario se lee así y, por lo tanto, también se debe llenar así. Considerándose entonces el diccionario real como el mostrado en la figura 3.6 . En la tabla 3.2 se recaba el resultado de la compresión de los primeros 6 símbolos del bloque (estado), en la salida se emite el índice y la distancia de la coincidencia dentro del diccionario. A diferencia de LZ77 la propuesta no emite el tercer valor, ya que se puede obtener fácilmente por los índices y disminuye el almacenamiento requerido en un byte por coincidencia encontrada. El apartado de comparación se tienen las siguientes opciones (Las abreviaturas utilizadas son para facilitar la lectura de la tabla y no cambian en nada el funcionamiento del algoritmo propuesto):

1. Comparar siguiente símbolo (CSS).
2. Coincidencia encontrada y comparar símbolos siguientes (CESS).
   - *a*) Fin de bloque, no existen más símbolos, emitir a salida coincidencia encontrada (FB).
   - *b*) Fin de bloque, no se pueden buscar más coincidencias, emitir a salida coincidencias acumuladas encontradas (FBEC).
   - *c*) Fin de bloque de diccionario, no se pueden buscar más coincidencias, emitir a salida coincidencias acumuladas encontradas (FBDC).
3. Coincidencia no encontrada y escribir literal (CNEL).

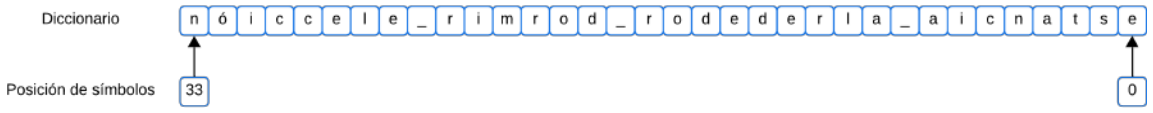


Figura 3.6: Diccionario con posiciones corregidas, elaboración propia.

La tabla con la comparación de solo cinco símbolos muestra que se debe considerar un diccionario óptimo para cada texto a comprimir, de lo contrario resultaría un archivo comprimido de mayor peso que los datos originales, siendo que la tupla resultante por cada coincidencia ocupa más bytes que el mismo símbolo comprimido, esto se aprecia en la salida de la tabla 3.2 donde todas los símbolos se encuentran en el diccionario con palabras aleatorias, pero en todos existe una distancia de 1, por lo tanto el diccionario no es óptimo ya que no se encuentran frases en él. Por ello se propone tener un diccionario dinámico para que se adecúe a cada texto de mejor forma que un diccionario aleatorio.

Tabla 3.2: Decisión con seis símbolos, elaboración propia.

Comparación	Decisión	Salida
o:e	CSS	
o:s	CSS	
o:t	CSS	
	...	
o:o	CESS	16,
	FB	16,1
d:e	CSS	
d:e	...	
d:d	CESS	13,
o:e	CNEL	13,1
a:a	CESS	3,
d:t	CNEL	3,1
t:t	CESS	2,1
s:s	CESS	1,1
e:e	CESS	0,1

3.5.2. Diccionario dinámico

En la búsqueda de mejorar el desempeño de la arquitectura propuesta, se diseñará un diccionario dinámico, el cual consiste en un conjunto de caracteres; en específico, conjunto de caracteres previamente encontrados en el texto y codificados mediante ASCII extendido. Para con ello buscar en los bloques de caracteres posteriores, coincidencias con los caracteres previamente almacenados.

Se propone el diseño de un diccionario dinámico, se comienza con un buffer vacío, el cual se va llenando a medida que se van leyendo los bloques a comprimir. En la figura 3.7 se muestra el llenado del diccionario dinámico para el ejemplo propuesto; también se corrigieron las posiciones de entrada del bloque. Para representación y facilidad de lectura se muestran de izquierda a derecha los dos buffer, internamente se leen de derecha a izquierda del MSB al LSB.

Es evidente que, si el tamaño del primer bloque es menor que el tamaño del diccionario, este quedará completamente contenido en el diccionario. Pero la compresión no es solo de un bloque y el diccionario se utiliza a lo largo de todos los bloques. Por lo tanto, el diccionario en un punto estará lleno. Esta investigación se enfoca en matrices sistólicas, pero el llenado y adaptación de diccionarios dinámicos también son un tema de sumo interés y que tiene aún muchas áreas de oportunidad, para el presente trabajo se utiliza el diccionario dinámico esencial. Considerando el texto del ejemplo completo (no solo para el primer bloque) se tienen los resultados recabados en la tabla 3.4. En las figuras 3.8 y 3.9 se observa el llenado del diccionario y los bloques a comprimir.

El ejemplo mostrado con el diccionario dinámico pasa a tener las mejores tasas posi-

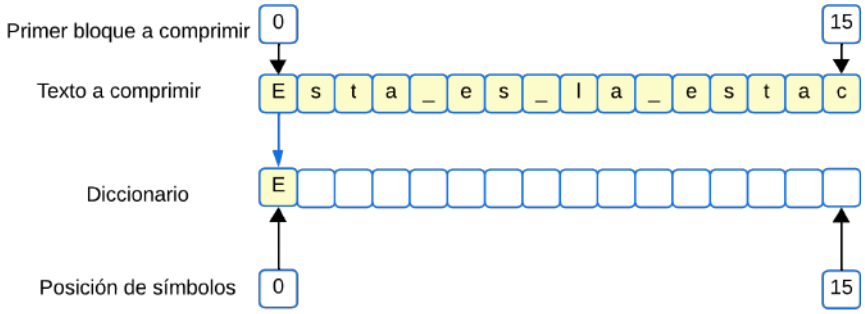


Figura 3.7: Diccionario dinámico propuesto, elaboración propia.

Tabla 3.3: Entrada completa, elaboración propia.

Comparación	Decisión	Salida
E:E	CESS	
	...	
c:c	FBEC	0,16
i:E	CSS	
	...	
o:o	FBEC	16,13

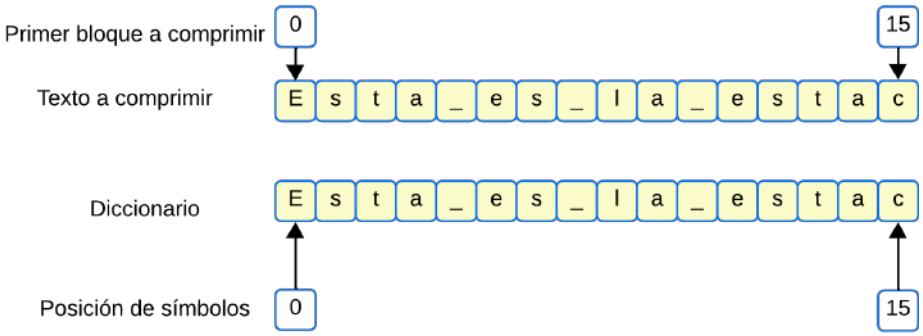


Figura 3.8: Diccionario dinámico bloque 1, elaboración propia.

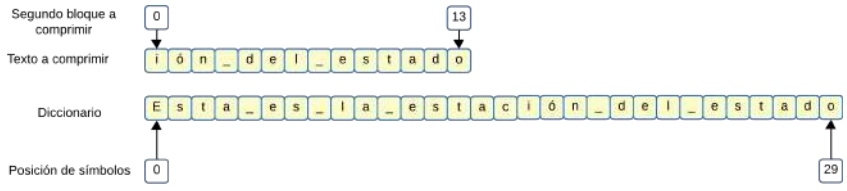


Figura 3.9: Diccionario dinámico bloque 2, elaboración propia.

bles de compresión con este método. Pero, este caso es excepcional, solo funcionaría con un diccionario de mayor tamaño que el texto a comprimir, y esto no es real ni eficiente, ya que el diccionario se debe agregar al archivo comprimido, convirtiéndose el diccionario en realidad en el archivo original y necesitando otro archivo para los índices que terminarían

recuperando el mismo archivo. Pero un diccionario dinámico con un tamaño adecuado, aunque no llegue a tener estos niveles de compresión si llega a ser muy útil, ya que el diccionario no pesará lo mismo que el archivo original y la lista de longitudes y distancias será más adecuada que con diccionarios estáticos. En el ejemplo con diccionario dinámico se pasó de un texto que ocupaba 33 bytes a 39 bytes (considerando el diccionario), demostrando lo dicho antes, si el diccionario es más grande, el archivo comprimido terminara pesando más que el archivo original. Para efectos de que quede clara la propuesta y se demuestre que el diccionario dinámico sí es una propuesta eficiente, se muestra el ejemplo considerando un diccionario más pequeño que la entrada (en la realidad, la mayoría de los casos serán así). Considerando solo para el ejemplo un diccionario de un tercio de la entrada, se tienen los resultados de la tabla 3.4 y en las figuras 3.10 y 3.11 se muestran los bloques y diccionario dinámico utilizado.

Tabla 3.4: Entrada completa con diccionario dinámico pequeño, elaboración propia.

Comparación	Decisión	Salida
E:E	CESS	
	...	
a:a	FBDC	0,11
␣*E*	CSS	
	...	
s:s	CNEL	4,3
t:E	CSS	
	...	
a:a	CNEL	2,2
c:E	CSS	
	...	
	FBEC	0,0
i	FBEC	0,0
ó	FBEC	0,0
n	FBEC	0,0
␣	CESS	
	...	
	FBEC	4,1
d	FBEC	0,0
e:e	CESS	
	...	
	FBEC	5,1
l	FBEC	0,0
␣	CESS	
	...	
t:t	FBEC CESS	4,3
	...	
	FBEC	2,2
d	FBEC	0,0
o	FBEC	0,0

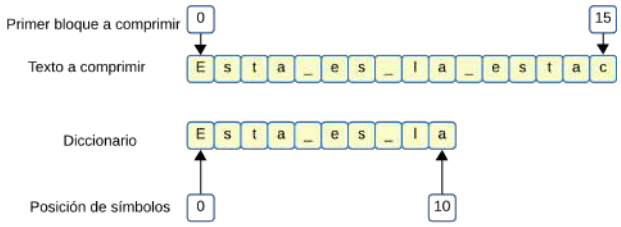


Figura 3.10: Diccionario dinámico pequeño con bloque 1, elaboración propia.

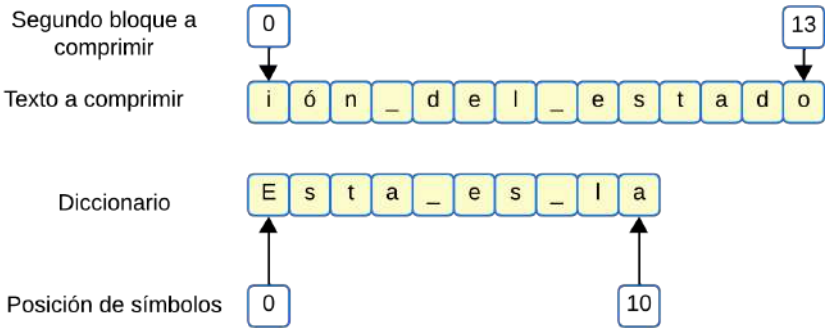


Figura 3.11: Diccionario dinámico pequeño con bloque 2, elaboración propia.

Con un diccionario de un tercio del texto, el ejemplo en lugar de ocupar 33 bytes de almacenamiento resulta de un tamaño de 56 bytes. Esto es resultado de comprimir textos pequeños, y una falta de optimización que todavía es posible; es evidente que el método en general, al igual que muchos otros, tiene carencias para comprimir textos pequeños. El verdadero potencial se comprueba con archivos de texto reales, que son de mayor tamaño en cuanto a símbolos y siguen patrones en las palabras que utilizan, ahí es donde el compresor propuesto basado en ventana deslizante debe demostrar su eficiencia. Con el algoritmo a desarrollar analizado, se debe tener en cuenta también el hardware de pruebas utilizado.

3.6. Descripción del algoritmo de compresión en hardware

La investigación se enfoca en la mejora del proceso de compresión mediante el diseño de hardware dedicado. El traslado de un algoritmo de software a una implementación en hardware no implica que su desempeño sea superior por el simple hecho de estar implementado en hardware. Por este motivo, una etapa inicial consiste en la especificación directa del algoritmo original en un lenguaje de descripción de hardware (HDL), replicando las operaciones en software. Este enfoque permite identificar las limitaciones inherentes al algoritmo pensado para software cuando se ejecuta en hardware, así como evaluar cómo los recursos y características del hardware pueden explotarse para mejorar su rendimiento.

Una metodología ampliamente aceptada para optimizar algoritmos en hardware consiste en implementar primero una versión base del algoritmo, que respete estrictamente la lógica secuencial utilizada en software. Desde este punto de partida, se analizan las

áreas críticas del proceso, como el manejo de datos, las dependencias internas y las operaciones que se prestan a la paralelización, como bloques DSP, memorias internas o lógica configurable.

Al establecer un diseño inicial directamente en HDL, se puede tener un análisis más profundo de las capacidades del hardware frente a las limitaciones del software, facilitando la identificación de estrategias específicas para mejorar la eficiencia del proceso de compresión.

3.6.1. Módulos y funcionalidades

El codificador a diseñar consta de varios módulos interconectados para realizar compresión de texto, basado en la ventana deslizante, incluyendo el manejo de datos de entrada, búsqueda de coincidencias y generación de datos de salida.

1. Módulo de entrada: Este módulo gestiona la recepción de los datos de entrada. Implementa un FIFO para almacenar temporalmente los datos entrantes.

2. Módulo de búsqueda: Se realiza mediante matriz sistólica, para poder buscar coincidencias comparando los símbolos con el diccionario de forma paralela.

3. Módulo de Salida: Después de identificar las coincidencias, este módulo genera los datos de salida. Esto incluye la posición y longitud de las coincidencias y los símbolos literales para casos donde no se encuentran coincidencias.

3.6.2. Maquina de estados

La máquina de estados del codificador base administra las transiciones entre los diversos módulos y coordina las operaciones de codificación. Los estados principales a diseñar se pueden ver en la figura 3.12, estos incluyen:

1. **IDLE**: Estado inicial en el que el sistema se prepara para la recepción de datos.

2. **INPUT**: Estado donde se cargan datos en el buffer y se preparan para el procesamiento.

3. **SEARCH**: Estado que activa la matriz de búsqueda para identificar coincidencias.

4. **OUTPUT**: Estado en el cual los resultados de la búsqueda se codifican y preparan para la salida.

3.6.3. Optimizaciones y rendimiento

El diseño inicial del sistema está configurado para emular la lógica de software, proporcionando una base para identificar áreas donde el hardware puede ofrecer mejoras significativas. Posibles optimizaciones incluyen el ajuste en la búsqueda y administración

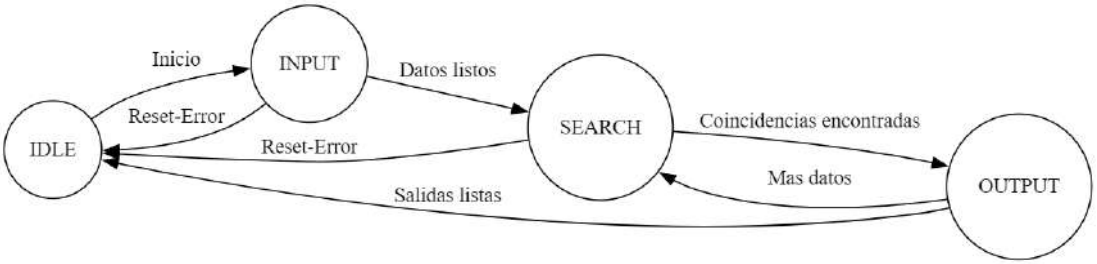


Figura 3.12: Maquina de estados base de compresor, elaboración propia.

del diccionario, así como la parte medular de este trabajo; se buscará mejorar el aprovechamiento del hardware mediante matrices sistólicas en el apartado de búsqueda y comparación. El diseño generado inicialmente se utiliza como base para realizar las adecuaciones necesarias y verificar las mejoras hechas contra el primero.

3.7. Descripción del algoritmo de descompresión en hardware

Se especificó también el decodificador, que descompone los datos comprimidos y recupera la información original; esto para corroborar que los datos que se trataron en verdad puedan ser recuperados; se diseñó en Verilog. A continuación, se presenta un análisis de este sistema, considerando la teoría a la que se debe apegar. Aunque el descompresor no estaba contemplado en la propuesta original de investigación, se decidió no limitarse a una simple transcripción del software al hardware. En su lugar, se optó por desarrollar una versión propia, adaptada a las características del hardware y basada en la teoría de ventanas deslizantes. Por lo tanto, es necesario implementar también un descompresor que permita validar tanto la eficacia de la compresión como la correcta recuperación de los datos.

3.7.1. Máquina de estados

Estos estados facilitan la organización del flujo de datos y la sincronización del proceso de decodificación; el diagrama planeado de su funcionamiento se muestra en la figura 3.13. El núcleo del decodificador es una máquina de estados finitos (FSM) que controla el proceso de decodificación a través de varios estados clave:

- **IDLE**: Espera activa para inicio de datos.
- **INPUT**: Recepción y registro de los datos comprimidos.
- **DECODE1** y **DECODE2_1/2**: Decodificación de los datos, donde DECODE1 maneja datos directos y DECODE2 maneja la expansión de referencias de coincidencia.
- **FINISH**: Finalización del proceso de decodificación.

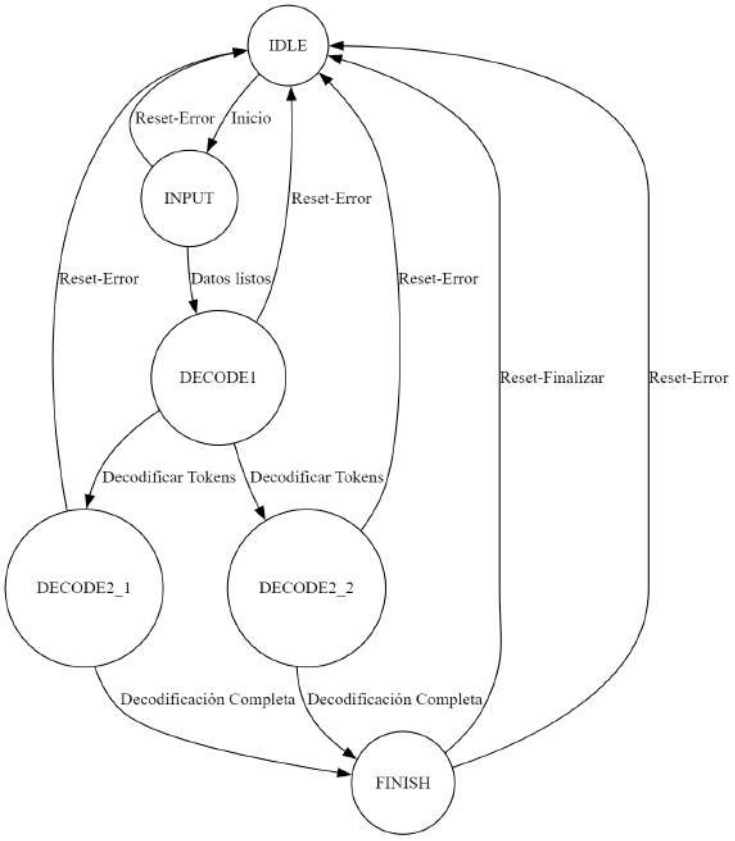


Figura 3.13: Maquina de estados base de descompresor, elaboración propia.

3.7.2. Implementación y funcionalidad

El decodificador utiliza un buffer circular y varios registros para manejar la información de desplazamiento y longitud. La lógica implementada en Verilog se encarga de extraer estos valores de los datos de entrada y utilizarlos para reconstruir la información original a partir de fragmentos previamente decodificados y almacenados en el buffer.

3.8. Construcción de la lista de funcionalidades

Conforme a lo descrito en la sección metodológica, el diseño de la arquitectura se estructuró a partir de representaciones visuales mediante diagramas de bloques. Estos diagramas permiten abstraer y modelar los módulos (representados por bloques) de mayor relevancia de la arquitectura propuesta, donde cada bloque corresponde a una etapa específica de la misma. Las conexiones entre los bloques, representadas mediante flechas, reflejan los flujos de datos y señales entre módulos, facilitando la comprensión del comportamiento global del sistema, así como las dependencias entre las entradas y salidas de cada componente. Esta representación resulta fundamental para garantizar una visión estructurada y coherente del diseño a nivel de sistema.

La construcción de la lista de funcionalidades se realizó considerando los módulos inter-

conectados que tiene la arquitectura propuesta, cada uno con funcionalidades específicas que contribuyen al flujo de compresión.

Cada uno de estos módulos opera de manera coordinada para garantizar que la arquitectura cumpla con los requisitos funcionales y no funcionales definidos. En la figura 3.14 se observa el primer diagrama diseñado para la arquitectura, donde la figura 3.15 es la corrección de ello. Se cambió el diagrama para no romper con la generalidad y con ello ser más entendible. Algunos nombres de módulos fueron reescritos para ser más adecuados a la tarea que llevan a cabo dentro de la arquitectura, aunque dan la impresión de funcionar en forma secuencial, en el detalle del módulo de búsqueda de coincidencias se aprecia el paralelismo de la arquitectura.

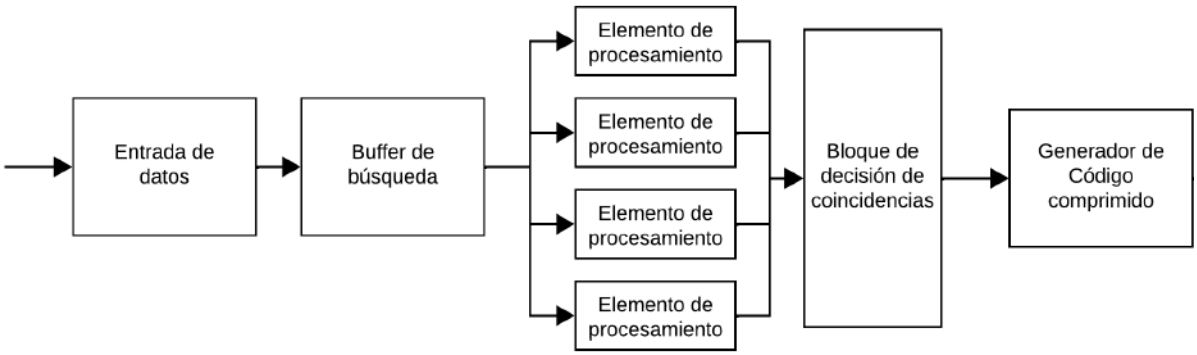


Figura 3.14: Diagrama de bloques de arquitectura inicial, elaboración propia.

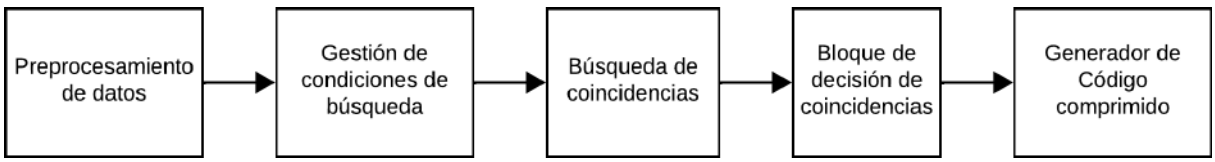


Figura 3.15: Diagrama de bloques de arquitectura corregido, elaboración propia.

A continuación, se describen las funcionalidades asociadas a cada módulo del sistema.

3.8.1. Preprocesamiento de datos

Es responsable de recibir y almacenar temporalmente el flujo de datos proveniente de la interfaz externa, para ser procesado por la arquitectura. Este módulo incluye un buffer FIFO diseñado para manejar una tasa de entrada de hasta 200 MB/s. El buffer organiza los datos en bloques de tamaño configurable, con soporte inicial para 128 bytes, y prepara estos bloques para su transmisión al módulo de procesamiento principal.

3.8.2. Gestión de condiciones de búsqueda

Maneja el diccionario necesario para la operación del algoritmo de ventana deslizante. Este módulo almacena los datos previamente procesados en un buffer implementado en la RAM interna del FPGA. El buffer debe mantener una estructura de cola circular que permita almacenar los últimos m bytes procesados, donde m corresponde al tamaño de la ventana deslizante. El buffer también se encarga de sincronizar las transferencias entre el preprocesador y la matriz sistólica, buscando que no se produzcan interrupciones en el flujo de datos.

3.8.3. Búsqueda de coincidencias

Es implementado como una matriz sistólica de elementos de procesamiento; es el núcleo del sistema. Cada elemento de procesamiento (eP) de la matriz es responsable de realizar comparaciones entre las sub-cadenas del buffer de búsqueda y los bloques del flujo de entrada. Este módulo está diseñado para operar en paralelo, procesando múltiples símbolos de entrada simultáneamente en cada ciclo de reloj. Los PEs realizarán operaciones como la comparación de sub-cadenas, el cálculo de longitudes de coincidencia y la propagación de resultados parciales hacia las celdas adyacentes.

3.8.4. Bloque de decisión de coincidencias

Toma los resultados parciales generados por la matriz sistólica y determina cuál es la coincidencia más larga encontrada en la ventana deslizante. Este módulo compara las longitudes de coincidencia y selecciona aquella con el valor mayor, obteniendo su posición correspondiente en el diccionario.

3.8.5. Generador de código comprimido

Recibe las decisiones del bloque de coincidencias y construye las referencias comprimidas. Estas referencias incluyen el desplazamiento relativo, la longitud de la coincidencia y el símbolo literal subsecuente. El módulo debe empaquetar estas referencias en palabras de datos de salida. Adicionalmente, este módulo debe manejar el flujo de salida de manera continua, asegurando que los datos comprimidos estén listos para su transmisión o almacenamiento inmediato.

3.9. Planeación por funcionalidades

Se define la planeación técnica de las actividades a desarrollar a lo largo de los semestres A2025 y B2025, se considera la planeación general del cronograma propuesto y se refina dentro de esos plazos las tareas que se deben realizar. Esto abarca el desarrollo y la optimización de los módulos de la arquitectura, desde septiembre hasta junio del año académico siguiente para el diseño preliminar, la integración de arquitectura, finalizando con la realización de pruebas y optimizaciones que se requieran. A continuación, se presentan las actividades por módulo de acuerdo con los objetivos y las funcionalidades especificadas.

3.9.1. Diseño y desarrollo de módulos (Septiembre - Diciembre)

1. **Módulo preprocesamiento de datos:** En septiembre, se debe iniciar con el diseño del módulo de entrada de datos. Este módulo es la entrada a la arquitectura propuesta y uno de los dos módulos que se conectan con el exterior.

2. **Gestión de condiciones de búsqueda:** Durante octubre, se desarrollará el módulo de buffer de búsqueda, lo más relevante de este módulo es la creación de un sistema eficaz para utilizar el diccionario. Este módulo se planea implementar con una estructura de datos que permita un acceso rápido y efectivo a las cadenas pasadas para una comparación con la ventana deslizante.

3. **Búsqueda de coincidencias:** En noviembre, se debe realizar la implementación de la matriz sistólica para buscar coincidencias. Esta parte del sistema debe utilizar algoritmos en paralelo para la identificación de patrones repetitivos dentro del flujo de datos.

4. **Bloque de decisión de coincidencias:** A inicios de diciembre, se debe realizar la implementación del módulo que determina la coincidencia más larga encontrada en la ventana deslizante.

5. **Generador de código comprimido:** Parte de diciembre se dedicará a integrar las coincidencias encontradas en un bloque de salida hacia el usuario, preparando el sistema para las pruebas iniciales de funcionalidad.

3.9.2. Integración y validación (Enero - Junio)

A partir de enero y continuando hasta marzo, se realizará una optimización de cada módulo. Esto incluirá ajustes finos en la lógica de la matriz sistólica en la FPGA y las correcciones necesarias.

1. **Integración y sincronización de módulos:** Parte de enero se dedicará a integrar los módulos previamente desarrollados. Esta etapa es importante porque los módulos de entrada, búsqueda y buffer deben trabajar de forma cohesiva y sincronizada, preparando el sistema para las pruebas iniciales de integración.

2. **Pruebas de concepto y simulaciones:** En abril, se llevarán a cabo pruebas de concepto completas y simulaciones detalladas para validar la efectividad de la arquitectura compresora. Estas pruebas ayudarán a identificar cualquier deficiencia o necesidad de mejora adicional.

3. **Preparación y revisión de la documentación:** Mayo se dedicará a la redacción detallada de los resultados y procesos en el documento final de la tesis. Además, se realizarán correcciones basadas en retroalimentaciones del comité tutorial.

4. **Finalización y presentación del proyecto:** Junio será el mes donde se finalicen todas las actividades, incluyendo una última revisión y la presentación del proyecto al comité evaluador, demostrando la funcionalidad y eficiencia del codificador en hardware.

Capítulo 4

Diseño

Se desarrolló el diseño de cada módulo de acuerdo con la planeación por funcionalidad. El siguiente diagrama de bloques muestra de forma general la arquitectura propuesta. Se

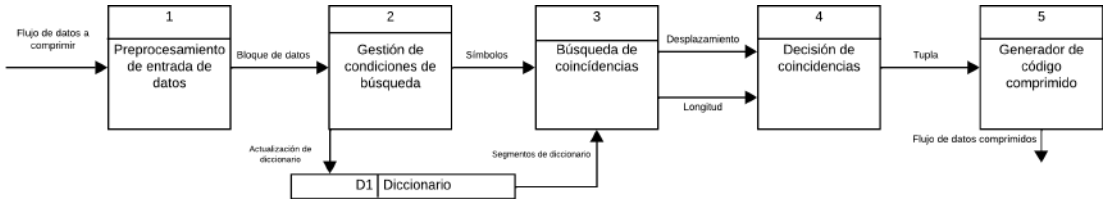


Figura 4.1: Diagrama de arquitectura propuesta, elaboración propia.

describe a detalle el diseño de la arquitectura y posteriormente se especifican las características de cada módulo. La arquitectura diseñada, por su naturaleza de conectarse al exterior, requiere la integración de un módulo que gestione la entrada de datos, así como el manejo de la salida de datos comprimidos. Estos componentes, aunque funcionalmente necesarios, no constituyen el interés principal de la investigación presente. Se considera la segunda etapa de desarrollo, después de realizar la propuesta en el apartado de análisis y llegar a las operaciones atómicas, en este apartado se busca detallar en lo posible, como se implementaron estas operaciones, adaptándose al desarrollo en hardware. La esencia de este estudio se centra en el análisis y diseño de una matriz sistólica, que es el módulo más importante de la arquitectura. La matriz sistólica, que por su diseño facilita una ejecución eficiente y escalable, aprovecha una topología que permite el procesamiento paralelo. Este enfoque no solo mejora el rendimiento, sino que también optimiza el uso del ancho de banda y disminuye la latencia de la comunicación interna, resultando en un sistema eficiente para la compresión de datos. Se describe a detalle el diseño de la implementación de la matriz sistólica propuesta.

4.1. Diseño de matriz sistólica

La matriz sistólica se utiliza para comprimir los datos introducidos, y para ello se utilizó en parte del diseño de un algoritmo basado en una ventana deslizante. Donde se busca y compara cada símbolo de entrada con un diccionario. A partir de esto, se asignan la

longitud y la distancia donde se encuentran las coincidencias. El siguiente diseño abarca el módulo de gestión de condiciones de búsqueda, la búsqueda de coincidencias y el de decisión de coincidencias. En la implementación propuesta se tienen dos buffers principales, el diccionario y el bloque de símbolos, ambos representados como vectores de bits de tamaño (4096 bits y 128 bits respectivamente) lo que es lo mismo que un diccionario de 512 símbolos y un bloque de anticipación de 16 símbolos. Cada buffer se dividió internamente en bytes de 8 bits cada uno. Esto para poder realizar las manipulaciones necesarias de cada byte de memoria de forma independiente dentro del módulo. Para poder tener acceso directo a cada byte se define el cableado hacia cada buffer, este enfoque va de la mano con la organización de cada vector y poder realizar la comparación byte a byte entre los buffers. Después de tener cada byte asignado y configurado para poder acceder a ellos, se compara cada segmento del buffer de búsqueda con cada byte del buffer de adelantamiento (cada símbolo se busca en el segmento del diccionario asignado). En esta etapa de prueba se dividió en 16 el diccionario, para que cada matriz sistólica de elementos de procesamiento busque en cada segmento de él. En la tabla 4.1 se muestra cómo se dividió el diccionario para enviar un segmento a cada elemento de procesamiento, siendo 32 símbolos codificados en ASCII extendido la longitud de cada segmento de diccionario enviado a cada elemento de procesamiento.

Tabla 4.1: Segmentos en los que se divide el diccionario, elaboración propia.

ID Segmento	Valor máximo	Valor mínimo
1	255	0
2	511	256
3	768	512
4	1024	769
5	1280	1023
6	1536	1279
7	1792	1535
8	2048	1791
9	2304	2047
10	2560	2303
11	2816	2559
12	3072	2815
13	3328	3071
14	3584	3327
15	3840	3583
16	4096	3839

Dentro de cada elemento de procesamiento se compara el segmento de diccionario y símbolo asignado, se detalla su funcionamiento en la siguiente sección. Los índices resultantes de cada elemento de procesamiento se revisan para obtener la posición del símbolo que coincide dentro del segmento de diccionario, entre este y símbolo. Esta posición se calcula utilizando una cascada de operadores ternarios que seleccionan el primer índice con un valor diferente de cero, añadiendo un desplazamiento basado en la posición de la

igualdad encontrada dentro de cada segmento de 32 bytes para saber exactamente en qué posición del diccionario original se encontró esa coincidencia. Adicionalmente, la lógica para determinar la longitud de coincidencia se diseñó mediante el cálculo de un XOR entre el diccionario y el bloque de los símbolos usando el índice calculado, permitiendo identificar exactamente dónde ocurren discrepancias entre los buffers. Las discrepancias se procesan para determinar la posición de la primera coincidencia, que a su vez se utiliza para establecer la longitud de coincidencia. Esta longitud se establece en cero si hay un fallo en las coincidencias. Finalmente, uno de los aspectos básicos al diseñar una arquitectura y que evita los latches, es la utilización de relojes para mantener sincronización entre los módulos de la arquitectura, en este caso se busca utilizar las mejores prácticas conocidas de diseño HDL, teniendo solo actualizaciones en el flanco positivo del reloj o en un reset negativo, para garantizar la estabilidad de los datos durante las operaciones de comparación.

4.1.1. Elementos de procesamiento

El elemento de procesamiento (eP) se diseñó de acuerdo con la teoría de matrices sistólicas. Siendo descrito cada eP para realizar una tarea muy específica, que es evaluar la coincidencia de un solo símbolo. Teniendo cada eP su propia memoria para almacenar su diccionario y el símbolo que debe comparar. Estos eP conforman la matriz sistólica, donde todos los eP son idénticos, teniendo así una configuración para realizar la búsqueda de coincidencias en paralelo. Con ello, cada paso de la comparación y asignación se realiza en un solo ciclo de reloj. Demostrando así que se pueden utilizar en paralelo varios eP, haciendo que el diseño sea escalable, segmentando en tantas partes el diccionario como se tengan eP en la matriz sistólica. Cada elemento de procesamiento se diseñó con un buffer de 8 bits para cada símbolo, un buffer de 256 bits para su sección asignada del diccionario. La comparación dentro de cada eP se realiza de manera secuencial, buscando una coincidencia exacta entre el byte de adelantamiento y cada byte del buffer de búsqueda. La estructura del módulo se detalla a continuación:

- Definición e inicialización de variables: Se define el diccionario como un arreglo de 32 elementos, donde cada elemento tiene 8 bits (256 bits en total). Este arreglo es llenado por la sección del diccionario que se le envió al eP.
- Proceso de comparación: La comparación del símbolo y el diccionario se realiza utilizando una serie de operadores condicionales (si, entonces), que son evaluados de manera secuencial (se están realizando pruebas con asignaciones bloqueantes, en breve se pondrá a prueba el funcionamiento con asignaciones no bloqueantes para realizar comparaciones en paralelo y ver si mejora la eficiencia del eP realizando comparaciones entre los 32 símbolos del diccionario y el símbolo de forma paralela). Se describió que, si existe una coincidencia en el índice, se almacena ese índice.
- Asignación de resultados: Se tiene también una bandera en el funcionamiento de cada eP que indica cuando no encontró ninguna coincidencia.
- Manejo de salidas: La salida de cada eP es un vector de 5 bits que puede representar valores de 0 a 31 (los índices de coincidencias posibles dentro del diccionario) y la

bandera que indica que no se encontró coincidencia (Ya que la bandera activa existe si y solo si no se encontraron coincidencias, se escribe sobre el mismo vector la bandera que es el número 32, imposible que las coincidencias den ese número y manteniéndose seguro el diseño, a su vez que se optimiza el diseño de la arquitectura.

4.2. Módulo de preprocesamiento de datos

Con el diseño principal bien definido. se especifica los detalles de cada módulo que forma parte de la arquitectura. El módulo de preprocesamiento de datos se encarga de la recepción y la adecuación de los datos. A continuación, se detalla el diseño de este módulo, incluyendo sus componentes principales y las especificaciones.

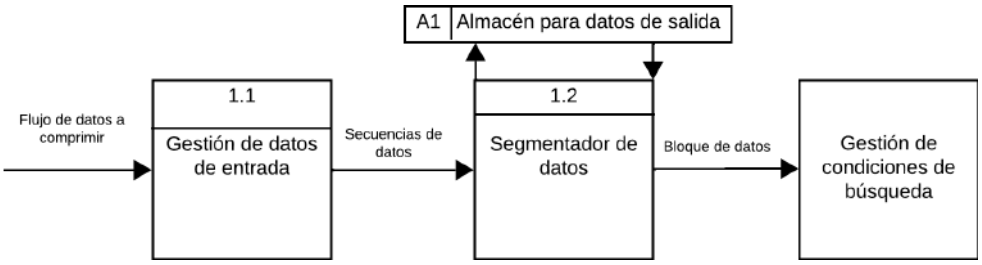


Figura 4.2: Diagrama de módulo de preprocesamiento, elaboración propia.

Función

Este módulo tiene como función principal recibir el flujo de datos desde una interfaz externa y organizar estos datos en bloques para su posterior procesamiento. Actúa como el primer punto de contacto para los datos entrantes, almacenándolos temporalmente.

Compatibilidad y codificación

- **Formatos de entrada:** El módulo es compatible con formatos de texto de entrada codificados en ASCII extendido, lo que permite que sea un compresor que soporte datos de entrada reales, ya que soporta caracteres especiales.

Entradas y salidas

- **Entrada**: Flujo de datos: Los datos ingresan al módulo en formato de 8 bits por símbolo, adecuado para manejar el formato de texto codificado en formato ASCII extendido.
- **Salida**: Se envía un bloque de 128 bits de datos al siguiente módulo.

4.3. Módulo de gestión de condiciones de búsqueda

Este módulo es responsable de manejar el diccionario y el buffer de coincidencias. A continuación, se detallan sus componentes, funcionalidades y especificaciones técnicas.

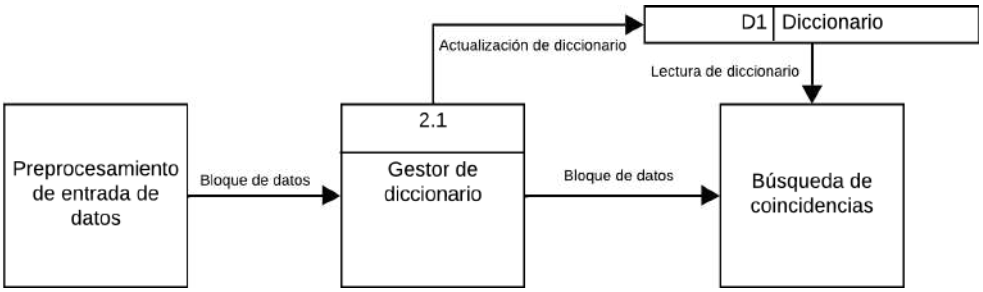


Figura 4.3: Diagrama de módulo de gestión de condiciones de búsqueda, elaboración propia.

Función

Este módulo opera como el núcleo de almacenamiento y búsqueda para el codificador, manteniendo las secuencias recientes y proporcionando acceso al diccionario para que la matriz sistólica busque coincidencias en las cadenas. Es importante para la implementación del algoritmo de ventana deslizante, permitiendo que las coincidencias encontradas no se vuelvan a procesar innecesariamente.

Especificaciones del buffer

- **Doble buffer:** En el desarrollo de la arquitectura propuesta, se busca incorporar como parte de las mejoras propuestas para el siguiente semestre un enfoque que combina las mejores características de diseños estudiados para buscar obtener el aprovechamiento del hardware de desarrollo. Se realiza la implementación de un sistema de buffer doble al que tiene acceso el módulo de gestión de condiciones de búsqueda. Este sistema se compone del clásico diccionario utilizado en ventanas deslizantes, para asignar a cada elemento de procesamiento una parte de él y un buffer secundario que almacena las coincidencias recientemente aceptadas. La introducción de este buffer adicional busca una mejora del rendimiento, especialmente en escenarios donde se encuentran repetidamente los mismos caracteres. En tales casos, el sistema no requiere volver a consultar el diccionario para cada símbolo repetido. En su lugar, las coincidencias previamente identificadas se transfieren directamente al módulo de decisión. Este módulo estará diseñado con la lógica necesaria para manejar estas situaciones, facilitando un proceso más rápido y reduciendo la carga de procesamiento símbolo por símbolo. Este diseño se pretende sea un paso adicional para minimizar la redundancia en el manejo de datos, lo que se traduce en una mayor rapidez en la compresión y una mejor utilización de los recursos del hardware.

- **Capacidad ajustable:** El tamaño del buffer es ajustable según el tamaño de la ventana deslizante definida, con una capacidad por defecto de 4096 bits para el apartado del buffer de búsqueda. Esto es configurable según las necesidades específicas del sistema y puede ser incrementado para adaptarse a ventanas de tamaño mayor.

Implementación en la FPGA

El buffer está implementado en los registros interna del FPGA, utilizando una estructura de cola circular que permite gestionar eficientemente los últimos m bytes procesados, donde m es el tamaño de la ventana deslizante.

Entradas y salidas

- **Entrada:**
  - **Datos del módulo de entrada:** Los datos se reciben del módulo de preprocesamiento en un bloque de 128 bits, a su vez que actualiza el diccionario y lee desde el con una capacidad de 4096 bits inicialmente.
- **Salida:**
  - **Bloque de datos hacia la matriz sistólica:** Los bloques de datos procesados son enviados a la matriz sistólica para la búsqueda de coincidencias a su vez que se deja listo el diccionario en segmentos de 256 bits para su futura lectura.

Sincronización y gestión de flujo de datos

El módulo también se encarga de sincronizar las transferencias entre el preprocesador y la matriz sistólica. Por lo cual deben mantener una constante lectura de datos desde el preprocesador a su vez que mantiene la generación de datos para los elementos que conforman la matriz sistólica.

4.4. Módulo de búsqueda de coincidencias

Este módulo utiliza una matriz sistólica de elementos de procesamiento (ePs), es esencial en la propuesta. La matriz está enfocada en realizar operaciones en paralelo para lograr el procesamiento en simultáneo de múltiples símbolos de entrada. A continuación, se especifican las características de este módulo.

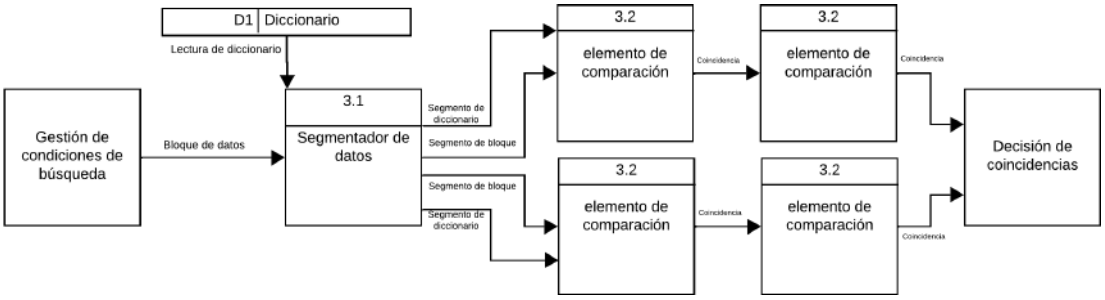


Figura 4.4: Diagrama de módulo de búsqueda de coincidencias, elaboración propia.

Arquitectura y funcionalidad

Cada eP en la matriz está interconectado de manera que puede recibir y enviar información a sus vecinos inmediatos. Esta disposición permite que las operaciones se realicen de manera fluida y coordinada a lo largo de toda la matriz, disminuyendo el tiempo empleado en buscar y comparar.

4.4.1. Operaciones de los ePs

- **Comparaciones sub-cadena:** Cada eP realiza comparaciones entre segmentos del buffer de búsqueda y bloques del flujo de entrada. Estas comparaciones se realizan mediante circuitos comparadores lógicos integrados en cada eP.
- **Cálculo de longitudes de coincidencia:** Los eP dan el índice de la coincidencia encontrada dentro de su segmento de diccionario, así se puede determinar es la repetición de datos en el flujo de entrada comparado con el diccionario.

Propagación de datos

Los datos se propagan a través de la matriz en un modelo de comunicación vecinal, donde cada eP transmite los resultados de sus comparaciones y cálculos. Este enfoque reduce los tiempos de propagación de la información y de acceso a la memoria, permitiendo que el sistema responda en un tiempo menor a los cambios en los datos de entrada.

Componentes de hardware de cada eP

- **Comparadores lógicos:** Cada eP contiene un comparador lógico que realiza comparaciones de igualdad entre dos segmentos: uno procedente del buffer de datos y otro el símbolo a buscar. Este comparador determina si hay una coincidencia entre los dos segmentos de datos. Esta lógica busca que cada eP pueda identificar correctamente dónde se encontraron coincidencias y responder adecuadamente cuando se encuentren nuevas coincidencias o cuando se reinicie la búsqueda.
- **Capacidad de manejo:** Cada eP puede procesar elementos de 8 bits, esto para minimizar la carga de trabajo y aumentar el cómputo en paralelo de instrucciones atómicas.

Entradas y salidas

Entradas

- **Datos del generador de condiciones de búsqueda**: 256 bits es el tamaño del buffer de búsqueda, representando los datos del diccionario.
- **Datos del generador de condiciones de búsqueda**: 8 bits es el tamaño del buffer de anticipación, representando los datos donde se buscará la coincidencia.

Salidas

- **Datos procesados**: Los resultados de las operaciones de los eP se transmiten en paquetes de datos de 5 bits.
- **Posiciones de coincidencia:** La matriz sistólica da las posiciones de las coincidencias, en un segmento de 11 bits (para manejar el diccionario completo), son utilizadas para identificar el lugar donde se encontró cada coincidencia.

4.5. Módulo de decisión de coincidencias

Recibe los resultados parciales de la matriz sistólica y determina la coincidencia más larga dentro de la ventana deslizante. Este módulo selecciona la coincidencia que será finalmente codificada y almacenada.

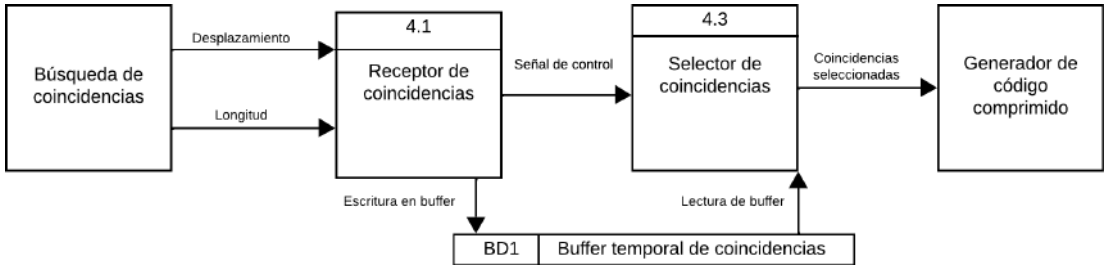


Figura 4.5: Diagrama de módulo de decisión de coincidencias, elaboración propia.

Función del módulo

Procesa los datos provenientes de la matriz sistólica, comparando las longitudes de las coincidencias detectadas. Su función principal es identificar y seleccionar la coincidencia con la mayor longitud.

Especificaciones técnicas

Mejora de la arquitectura : Más allá de seleccionar la coincidencia más larga. Este módulo también guarda las coincidencias seleccionadas en un buffer interno. La finalidad a futuro de este almacenamiento es permitir que el módulo de asignación de búsqueda acceda a estas comparaciones recientes, facilitando la revisión de coincidencias previamente encontradas para determinar si requieren ser procesadas nuevamente. Esta capacidad mejora la eficiencia del sistema al reducir la necesidad de reevaluar coincidencias ya establecidas, disminuyendo la carga de trabajo general y el uso de recursos.

Entradas y salidas

Entradas

- **Longitudes de coincidencia:** Cada longitud de coincidencia, representada en 5 bits cada una, es evaluada por el módulo para determinar cuál es la mayor.
- **Posiciones de coincidencia:** Las posiciones de las coincidencias de 11 bits, son utilizadas para identificar el lugar dentro del buffer de búsqueda donde se encontró cada coincidencia.

Salidas

- **Longitud de la mejor coincidencia:** El módulo emite la longitud de la coincidencia más larga detectada, en 5 bits, para indicar la repetición más significativa encontrada en los datos.
- **Posición de la mejor coincidencia:** Además de la longitud, el módulo también emite la posición, en 11 bits, facilitando su recuperación y codificación posterior.

4.6. Módulo generador de código comprimido

Transforma las coincidencias seleccionadas por el módulo de decisión de coincidencias en códigos comprimidos. Este módulo agrupa los resultados del análisis de coincidencias en un formato que el descompresor entenderá.

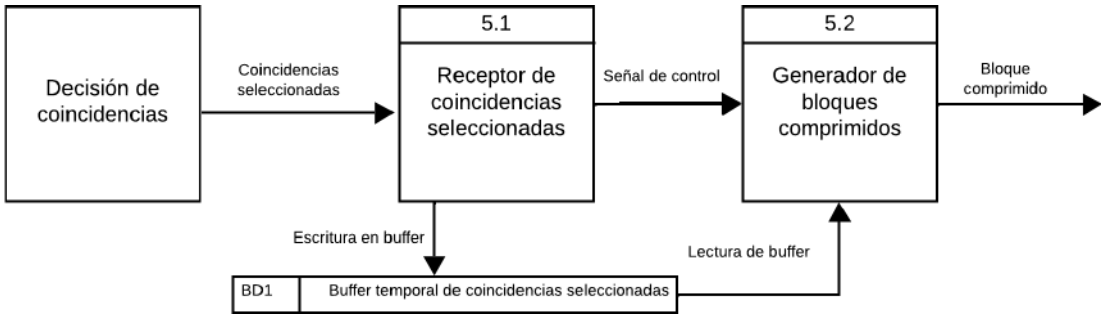


Figura 4.6: Diagrama de módulo de generador de código comprimido, elaboración propia.

Función del módulo

El módulo recibe la longitud y la posición de las mejores coincidencias y las agrupa en una serie de referencias. Estas referencias después se utilizan para reconstruir los datos originales durante el proceso de descompresión y contienen los elementos necesarios, como el desplazamiento relativo, la longitud de la coincidencia.

Entradas y salidas

Entradas

- **Longitud de la mejor coincidencia:** Recibida en formato de 5 bits, esta entrada indica la longitud de la secuencia de datos que ha sido identificada como una coincidencia en el flujo de entrada.
- **Posición de la mejor coincidencia:** En formato de 11 bytes, especifica la posición dentro del diccionario donde comienza la coincidencia.

Salida

- **Código comprimido:** El resultado del procesamiento, es en formato de 16 bytes, encapsula la información necesaria para representar cada coincidencia detectada, incluyendo tanto la longitud como la posición y el símbolo literal.

4.7. Especificaciones de entradas y salidas para los módulos de la arquitectura

La tabla 4.2 proporciona una descripción de las entradas y salidas para cada módulo de la arquitectura propuesta. Se incluyen detalles sobre las dimensiones y los valores que pueden manejar.

Tabla 4.2: Entradas y salidas de los módulos, elaboración propia.

Módulo	Entradas	Salidas
Preprocesamiento de datos	8 bits	128 bits
Gestión de condiciones de búsqueda	128 bits, 4096 bits	256 bits
Búsqueda de coincidencias	(256 bits, 8 bits) por eP	5 bits, 11 bits
Decisión de coincidencias	5 bits, 11 bits	5 bits, 11 bits
Generador de código comprimido	5 bits, 11 bits	16 bits

Se especifican los datos en la tabla 4.2 , ya que será de ayuda para la etapa siguiente, donde se debe tener claro los datos que debe recibir y enviar cada módulo.

Capítulo 5

Construcción

Se detalla la construcción de cada funcionalidad para los módulos que conforman la arquitectura. En este contexto, se describe la implementación de cada módulo y los desafíos encontrados durante cada fase del desarrollo. Es importante señalar que este documento no incluye código, ya que no se busca ofrecer una guía detallada paso a paso. Sin embargo, se describirá cómo se realizó cada proceso y los resultados esperados de cada módulo. Estos se examinarán en el siguiente capítulo, donde se evaluarán las pruebas y se considerarán posibles mejoras. Cabe mencionar que cada diseño de cada módulo se llevó a cabo conforme a lo planificado, iniciando con el módulo de preprocesamiento de datos.

5.1. Módulo de preprocesamiento de datos

Se implementó en Verilog utilizando un controlador FIFO utilizando registros para leer y escribir en el buffer por donde pasan los datos antes de su procesamiento posterior. Se consideró también tener señales de control para indicar cuándo el buffer se encuentre lleno o vacío. La construcción se centró en tener un flujo continuo de datos hacia la arquitectura, así como manejar de forma correcta caracteres que se encuentran en un archivo de texto. A continuación, se describen los aspectos más relevantes de la implementación, considerando posibles mejoras que se comprendieron durante el desarrollo.

Una de las mejoras que se pueden implementar en este módulo, es la utilización de bloques de memoria especificados por AMD mediante una IP para el manejo de colas FIFO, pero se debe tener en cuenta si vale la pena quitar generalidad a la arquitectura en busca de mejorar la eficiencia en específico para esta tarjeta de desarrollo o seguir con el objetivo inicial que es una puesta a prueba de una arquitectura general con miras a implementarse a futuro como coprocesador en un dispositivo móvil. Otra mejora que se puede realizar en el diseño, es aumentar la capacidad del buffer según los requisitos específicos de la aplicación; esto puede ayudar a manejar mejor los picos de carga de datos.

5.2. Módulo de gestión de condiciones de búsqueda

Se realizó buscando optimizar el almacenamiento y la búsqueda de secuencias de datos dentro de una ventana deslizante. Construido con una combinación de buffers de búsqueda y un diccionario dinámico.

5.2.1. Estructura del módulo e implementación en verilog

Se utilizó una estructura de cola circular para gestionar el buffer de búsqueda, que almacena las últimas secuencias procesadas. Este buffer se implementa en los registros internos del FPGA, aprovechando su alta velocidad de acceso y la capacidad de configurar detalladamente; por ejemplo, para poder ajustar el tamaño del diccionario,

RAM FPGA

Considerando la tarjeta de desarrollo utilizada, ALINX AX7A200, tiene varios bloques de RAM configurables, que se usaran de forma indirecta, conocidos como bloques de RAMB18 y RAMB36. Estos bloques pueden configurarse de manera independiente para funcionar como dos bloques separados de 18Kb o combinados como un bloque de 36Kb, dependiendo de las necesidades específicas de la aplicación. La RAM puede configurarse en modos de profundidad y anchura variable, como 512x72, 1024x36, 2048x18 entre otros, permitiendo ajustes precisos según los requisitos del sistema de compresión de datos.

La RAM interna en este modelo de FPGA también soporta características como:

- **Acceso de doble puerto**: Permite que la RAM sea leída y escrita simultáneamente en diferentes ubicaciones, lo cual sería de ayuda si se implementa la actualización y consulta del diccionario en paralelo.
- **Error Correction Code (ECC) opcional**: Mejora la integridad de los datos al detectar y corregir errores en tiempo real, probablemente va más allá de los objetivos de este trabajo.

Implementación

En Verilog, el módulo se describe utilizando registros y lógica de control para gestionar los estados de entrada y salida de datos. Se utilizan generadores de instancia (genvar) y bloques generate para crear de forma dinámica los elementos necesarios del buffer basados en los parámetros configurados. Una característica que logra que el diseño sea escalable. Los buffers se implementaron como arrays de registros, con lógica específica para manejar la cola circular y actualizar el diccionario.

5.3. Módulo de búsqueda de coincidencias

Se implementó como una matriz sistólica de ePs para comparar las sub-cadenas del buffer de búsqueda con las cadenas del flujo de entrada de manera paralela en cada ciclo de reloj.

5.3.1. Especificaciones

Cada eP dentro de la matriz sistólica se construyó para realizar comparaciones entre la entrada y las secuencias del buffer. La implementación utiliza múltiples instancias del submódulo de comparación, cada uno de los cuales compara segmentos del buffer de búsqueda y el buffer y las cadenas del flujo de entrada. En la figura 5.1 se muestra el esquema RTL de un elemento de procesamiento utilizado en el diseño, este realiza la comparación de un símbolo del diccionario y el buffer de adelantamiento. Los resultados de estas com-

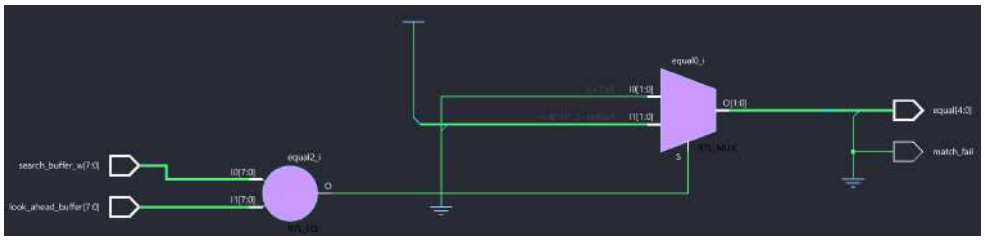


Figura 5.1: Diseño RTL de elemento de procesamiento, elaboración propia.

paraciones son evaluados para determinar la coincidencia más larga. La lógica de selección combina los resultados de todos los ePs y determina el índice final de la coincidencia más larga utilizando operadores lógicos y estructuras de control de flujo. En la figura 5.2 se muestra una matriz pequeña para demostrar cómo se interconectan entre si los elementos de procesamiento. La matriz sistólica de la figura consta de 4 elementos de procesamiento interconectados. En la realidad la visualización del diseño no puede ser representada de forma correcta en un documento, ya que se utilizan 2 matrices sistólicas de 16 elementos de procesamiento cada una. Esto para poder realizar 32 comparaciones en paralelo, es fácilmente escalable y se procuró diseñarlo con este fin en mente. Posiblemente una de las

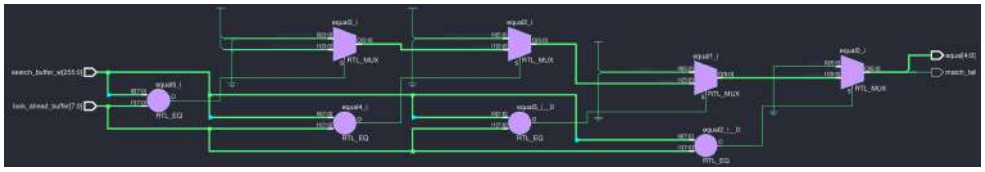


Figura 5.2: Matriz sistólica de la arquitectura, elaboración propia.

mejoras que se podrían realizar en este módulo recae en ajustar las dimensiones de las comparaciones y explorar diferentes configuraciones de interconexión entre los ePs para buscar mejoras en el tiempo empleado en realizar las operaciones.

5.4. Modulo de decisión de coincidencias

La construcción del módulo se consideró para determinar cuál de las coincidencias encontradas en la matriz de procesamiento es la más larga y, por tanto, la más relevante para la compresión de datos. A continuación, se presentan las características técnicas y los detalles de implementación de este módulo:

- **Comparación de longitudes**: El módulo toma las longitudes de coincidencias parciales y utiliza un buffer para almacenarlas de forma temporal y de él, las compara para encontrar la más larga. Utiliza operaciones lógicas para determinar el índice y la longitud de la coincidencia más larga dentro de la ventana deslizante.
- **Asignación de índices y longitudes**: Se utilizó un conjunto de operadores ternarios para evaluar y asignar el índice correspondiente a la coincidencia más larga basado en la información procesada por elementos anteriores en la cadena de procesamiento.
- **Sincronización y flujos de control**: Se construyó la lógica de control para manejar estados como IDLE, SEARCH y OUTPUT mediante una máquina de estados para gestionar el flujo de datos y las transiciones entre diferentes fases de procesamiento.

5.5. Generador de código comprimido

Este módulo es responsable de recibir las decisiones sobre las coincidencias más largas del módulo de decisión de coincidencias y convertirlas en un formato de salida comprimido que encapsula la posición y la longitud de la coincidencia. Se construyo con las dimensiones suficientes para poder manejar las entradas de longitud y posición; de ellas genera código comprimida realizando una concatenación entre los datos de entrada y generar el código comprimido de 16 bytes. Cabe aclarar que este módulo genera una trama de datos comprimidos, a él se pueden conectar diversas interfaces que deben generar las tramas adecuadas, ya sea para agregar redundancia para posterior verificación o adecuación a dimensiones requeridas. Se realizo con este fin en mente para no limitar la interfaz a tramas específicas de algún estándar o para cierta aplicación solamente.

Capítulo 6

Pruebas

Las pruebas se realizaron enfocadas en la verificación de la arquitectura en dos vertientes, considerando la simulación y la implementación en la tarjeta de desarrollo FPGA. Se utilizaron archivos de distintos corpus, comparando los resultados obtenidos con los del compresor ZIP nativo disponible en un sistema Android 13 y 14 respectivamente. A continuación, se detallan los aspectos relevantes.

6.1. Conjunto de datos

Resulta complejo determinar con precisión qué conjunto de datos será el más adecuado para conformar un corpus de evaluación. No obstante, existen ciertos criterios que son deseables al momento de su selección [80]:

- **Representatividad**: El corpus debe reflejar adecuadamente los tipos de archivos que un sistema de compresión probablemente procesará en el futuro. Esto implica la inclusión de una variedad de formatos y contenidos heterogéneos.
- **Disponibilidad**: El corpus debe ser de fácil acceso para la comunidad. La forma más eficaz de lograr esto es buscarlo mediante plataformas en línea.
- **Dominio público**: El corpus debe contener únicamente material de dominio público. Esto excluye una gran cantidad de archivos reales de interés, como imágenes o videos completos, debido a la presencia inevitable de contenido con derechos de propiedad intelectual.
- **Tamaño adecuado**: El corpus no debe ser más grande de lo necesario. Un tamaño excesivo implicaría mayores costos de almacenamiento, transmisión y procesamiento, lo que podría dificultar su distribución y uso.
- **Percepción de validez y utilidad**: Para tener una adopción generalizada, el corpus debe ser percibido como una herramienta válida y útil. Para ello, es fundamental que contenga tipos de archivos comúnmente utilizados y que el procedimiento de selección esté claramente documentado y publicado.

- **Validez y utilidad**: Más allá de la percepción, el corpus debe permitir evaluar de manera precisa el desempeño de los algoritmos de compresión. Es decir, los resultados obtenidos al procesar los archivos del corpus deben correlacionarse con el rendimiento que dichos algoritmos presentan en aplicaciones reales.

Con ello se contempló la utilización de archivos de diferentes conjuntos de datos estandarizados para pruebas de compresión sin perdidas, siendo principalmente tres. Los cuales son:

- **Calgary Corpus** [81], se creó en 1987 como un conjunto estándar para la evaluación de algoritmos de compresión de datos. Busca cubrir distintos tipos de contenido, incluidos texto plano, código fuente, archivos binarios y datos estructurados. Su diseño buscaba ofrecer un conjunto compacto pero diverso, permitiendo evaluar tanto la eficiencia de compresión como el rendimiento computacional de distintas técnicas. Aunque fue ampliamente utilizado en la validación de algoritmos clásicos como Huffman, LZW y variantes de LZ77, actualmente se considera limitado por su reducido tamaño y su falta de representación de formatos modernos como XML. Aun así, sigue siendo utilizado como referencia histórica y para establecer comparaciones con resultados previos, por ello se considera importante evaluar la arquitectura con este conjunto de datos.

- **Canterbury Corpus** [38], fue desarrollado en 1997 por el Departamento de Ciencias de la Computación de la Universidad de Canterbury, como un reemplazo técnico más adecuado al Calgary Corpus. Este conjunto contiene 11 archivos seleccionados para representar de manera más precisa los patrones de datos presentes en aplicaciones reales. Incluye textos literarios, documentos estructurados y archivos binarios. A diferencia de su predecesor, el Canterbury Corpus incorpora datos más variados y con estructuras más representativas del uso real, lo que permite un análisis más fiable del comportamiento de algoritmos de compresión en entornos prácticos. Su adopción en investigaciones ha contribuido a mejorar la robustez entre esquemas como BWT, PPM, LZMA o codificación aritmética, se considera una referencia obligatoria para pruebas en contextos más realistas.

- **Silesia Corpus** [82], desarrollado en el año 2003 es también un conjunto estándar para realizar pruebas de funcionamiento con este tipo de datos, el cual abarca una gama diversa de tipos de datos, incluidos textos literarios, bases de datos científicas, ejecutables, imágenes médicas y documentos estructurados (XML), con tamaños entre 6 MB y 51 MB. Este corpus refleja patrones de uso actuales y fue concebido para probar compresores sobre datos representativos del entorno real, excluyendo multimedia con compresión con pérdida. Su estructura permite comparar eficientemente el desempeño de algoritmos, en especial en escenarios de grandes volúmenes de datos y aplicaciones modernas como bases de datos, software complejo y documentación técnica. En resumen, el Silesia Corpus es una colección de archivos de diversos formatos que representan diferentes tipos de datos, incluyendo texto, imágenes, código ejecutable y datos estructurados. Aunque sus tamaños no son representativos de aplicaciones embebidas típicas, este corpus permite obtener una estimación confiable del

rendimiento de algoritmos de compresión frente a flujos de datos reales y variados. Se selecciono porque es comúnmente utilizado para comparar algoritmos como LZ77, LZ4 y sus variantes.

6.2. Dispositivos a comparar

Una vez configurado el entorno de simulación con un reloj de 200 MHz en el testbench, se definieron los dispositivos de referencia utilizados para la comparación del rendimiento de compresión. Dado que la arquitectura propuesta está orientada a ser una solución de compresión viable en dispositivos móviles, se eligió para el escenario de evaluación la comparación con equipos que representan dos segmentos distintos del mercado Android en la actualidad. En primer lugar, se seleccionó un dispositivo de gama media representativo del perfil de usuario promedio en México. Para este propósito, se utilizó un celular con procesador Mediatek MT8788V Octa-core (4x2.0 GHz Cortex-A73 & 4x2.0 GHz Cortex-A53), almacenamiento interno UFS 2.1 que tiene velocidades máximas de lectura de 860 MB/s y de escritura de 255 MB/s, cuenta también con 8Gb de memoria RAM LPDDR4X a 1800MHz. Este modelo fue elegido por su precio accesible y sus características contenidas y congruentes con los terminales más distribuidos en la población mexicana, de acuerdo con datos recientes [83].

Respecto al dispositivo que actuara como gama alta, se consideró el Samsung Galaxy S24 Ultra, equipado con el procesador Qualcomm Snapdragon 8 Gen 3 for Galaxy (1×3.39GHz Cortex-X4 + 5×3.1GHz Cortex-A720 + 2×2.3GHz Cortex-A520) y 12GB de RAM LPDDR5X. Este dispositivo representa el estándar actual de alto rendimiento en Android y ofrece una referencia tecnológica de última generación, tanto en capacidad de procesamiento como en velocidad de acceso a memoria y almacenamiento UFS 4.0 que permite velocidades teóricas de lectura secuencial de hasta 4300 MB/s y velocidades de escritura secuencial de hasta 4000 MB/s. La comparación con ambos dispositivos busca tener resultados objetivos: por un lado, evaluando la competitividad de la arquitectura propuesta frente a la capacidad de procesamiento de un dispositivo promedio; y por otro, examinando su eficiencia en relación con un dispositivo gama alta. Con ello se busca que las conclusiones derivadas de esta investigación sean técnicamente representativas y comparadas con entornos reales de uso.

La simulación fue ejecutada en Vivado para obtener el rendimiento estimado en hardware (frecuencia de 200 MHz), mientras que la compresión ZIP se realizó usando la aplicación de archivos integrada de Android [84] mediante la aplicación Termux [85] y medida con hyperfine [86].

6.2.1. Consideraciones sobre el tiempo medido en dispositivos Android

Al analizar los resultados obtenidos es importante tener en cuenta respecto al tiempo de procesamiento; que en el entorno Android, incluso en dispositivos de gama alta como el Samsung S24 Ultra, la ejecución de tareas de compresión involucra múltiples subsistemas y capas de abstracción que repercuten en el desempeño medido.

Para sustentar y contextualizar los tiempos observados en los resultados, se consideraron evaluaciones realizadas en laboratorio por los fabricantes, perfiles de rendimiento y documentación oficial. Donde para evaluar el dispositivo de gama alta, TERMINAR realizaron la ejecución del algoritmo Deflate (Zlib 1.2.11) [87], implementado en CC++ mediante Android NDK y ejecutado con acceso a cuatro núcleos del Snapdragon 8 Gen 3, se registró un consumo promedio de 3.2 W. Las pruebas se realizaron sobre archivos ASCII de 10 Mb, donde se midió un tiempo promedio de compresión de 860 ms, es decir, una tasa de 8.6 ms/Mb. Sin embargo, este valor no representa únicamente el tiempo neto de cómputo del algoritmo [88].

El sistema operativo Android prioriza tareas críticas de fondo y de red, lo cual afecta la disponibilidad continua del procesador. Además, el sistema debe gestionar la lectura desde almacenamiento no volátil (UFS 4.0), los accesos a memoria del espacio de usuario, y la sincronización multihilo mediante semáforos y objetos JNI. Cada una de estas capas introduce latencias adicionales no despreciables. La lectura y escritura secuencial se realizan con latencias inferiores a 0.5 ms/MB, pero el cruce de contexto entre espacio de usuario y kernel, junto con la sincronización de hilos, puede añadir entre 0.5 y 2 ms por transacción.

Estos factores, medidos con herramientas como Trepn Profiler [7] y confirmados por reportes técnicos como los de AnandTech, evidencian que el tiempo total de compresión incluye múltiples elementos externos al algoritmo en sí. En consecuencia, cualquier comparación directa con arquitecturas dedicadas debe considerar estas restricciones inherentes al entorno de ejecución en dispositivos Android, en la tabla 6.1 se presenta un resumen de las características ofrecidas por las fuentes citadas.

Tabla 6.1: Resumen de rendimiento y consumo en Samsung S24 Ultra, basado en [7].

Parametro	Valor medido
Consumo promedio en compresión	3.2 W
Latencia de lectura UFS 4.0	<0.5 ms/Mb
Retardo por semáforos y sincronización	0.5 – 2.0 ms
Tiempo total de compresión de 10 MB	860 ms
Velocidad promedio de compresión	8.6 ms/Mb
Consumo pico en carga completa	24 W

Como nota final respecto a estos datos que provee el fabricante, son valores ideales en entornos controlados y no demuestran el comportamiento real en un dispositivo de uso común, considerando ello, el tiempo medido en el dispositivo en pruebas reales y con diferentes herramientas difiere ligeramente.

6.3. Pruebas en simulación

Para validar el correcto funcionamiento de la arquitectura propuesta, se ha llevado a cabo la etapa de simulación utilizando el entorno de desarrollo proporcionado por AMD, Vivado. Siendo parte de las dos vertientes utilizadas para la verificación de la arquitectura. Ambas formas se basan en la metodología planteada en este trabajo. Las pruebas en simulación se han orientado a evaluar el comportamiento y la eficiencia de la arquitectura.

Para ello, en primer lugar, se configuro el entorno de simulación de acuerdo a las características proporcionadas por la tarjeta de desarrollo utilizada. con la frecuencia de oscilación del cristal (que actúa como reloj) configurada en el caso ideal con velocidad de funcionamiento de 200MHz.

6.3.1. Conversión de frecuencia a período

Dada una frecuencia:

$$f = 200\,\text{MHz}$$

El período T se obtiene como el inverso de la frecuencia:

$$T = \frac{1}{f}$$

Sustituyendo el valor:

$$T = \frac{1}{200 \times 10^6} = 5\,\text{ns}$$

Por lo tanto, para simular un reloj de 450 MHz, se requiere un ciclo completo de aproximadamente:

$$T = 2.222\,\text{ns}$$

Y cada medio ciclo (para la generación del reloj en Verilog) corresponde a:

$$T_{\text{medio}} = \frac{T}{2} = \frac{5}{2} = 2.5\,\text{ns}$$

6.3.2. Comparativa con Calgary Corpus

La tabla 6.2 presenta los resultados comparativos de compresión para distintos archivos del Calgary Corpus entre los tres dispositivos: Android de gama media y alta, y la arquitectura propuesta. Se analizan dos aspectos principales: la tasa de compresión obtenida (la cual es exactamente la misma para los dos dispositivos Android, ya que se utiliza el mismo programa en la misma versión)y el tiempo de procesamiento requerido para cada archivo, cabe mencionar que la arquitectura propuesta está diseñada para manejar únicamente texto plano, por lo que solo se utilizaran este tipo de archivos del corpus para realizar las pruebas.

En cuanto a la tasa de compresión, se observa que la arquitectura supera de forma teórica a los dispositivos Android en la mayoría de los casos. Por ejemplo, en el archivo 'book1', la arquitectura alcanza una tasa de compresión de 20.259, mientras que en Android solo se logra una relación de 2.45, lo cual representa una tasa de compresión 8 veces mayor. Esta tendencia se repite en archivos como 'bib', 'book2', 'news' y 'paper2', donde la compresión alcanzada por la arquitectura basada en FPGA oscila entre 3.4 y 4.9, valores superiores a los obtenidos por los dispositivos Android. Incluso en archivos de menor tamaño, como 'progc', 'paper1' o 'trans', la tasa lograda por la arquitectura es significativamente mayor,

lo que refleja la capacidad del sistema para adaptarse a diferentes volúmenes y estructuras de datos. Solo en casos particulares como 'progl' y 'progp', Android alcanza una tasa de compresión ligeramente superior (4.37 y 4.34 frente a 4.02 y 3.685 respectivamente), aunque la diferencia no es crítica desde el punto de vista de eficiencia general.

Respecto al tiempo de procesamiento, los resultados muestran una ventaja abrumadora de la arquitectura FPGA frente a los sistemas Android, tanto de gama media como alta. Pero, estos valores no son completamente confiables debido a múltiples factores inherentes al entorno de ejecución, que se consideraron con anterioridad. Teniendo en cuenta ello, los resultados validan que la arquitectura propuesta no solo logra mejores tasas de compresión para la mayoría de los casos, sino que también reduce en cierta medida los tiempos de procesamiento. Esta mejora está directamente relacionada con el diseño orientado a hardware, el uso de una matriz sistólica y la capacidad de procesamiento paralelo que permite alcanzar niveles de rendimiento superiores a los de procesadores de uso general, incluso en dispositivos móviles de alto desempeño.

Tabla 6.2: Comparación Calgary Corpus en simulación, elaboración propia.

Archivo	Tipo	Tamaño [Bytes]	Tasa compresión Android	Tasa compresión Arquitectura	Tiempo gama media [ms]	Tiempo gama alta [ms]	Tiempo arquitectura [ms]
bib	ASCII en formato UNIX "refer"	111,261	3.16	**4.30**	43.5	12.5	**1.301**
book1	ASCII sin formato	768,771	2.45	**20.259**	235.7	56.2	**1.917**
book2	ASCII en formato UNIX "troff"	610,856	2.96	**4.898**	150	42.9	**6.291**
news	ASCII: archivo por lotes	377,109	2.60	**4.79**	89.3	30.9	**3.958**
paper1	Formato "troff"de UNIX	53,161	2.84	**3.752**	34.5	13.3	**0.7151**
paper2	Formato "troff"de UNIX	82,199	2.75	**4.165**	35.7	17.4	**0.9981**
progc	Código fuente en C	39,611	2.95	**3.410**	26.0	9.9	**0.5854**
progl	Código fuente en Lisp	71,646	**4.37**	4.02	31.5	13.6	**0.8989**
progp	Código fuente en Pascal	49,379	**4.34**	3.685	19.3	10.9	**0.6776**
trans	Caracteres ASCII y de control	93,695	**4.90**	4.265	32.3	16.3	**1.1046**

6.3.3. Comparativa con Canterbury Corpus

Los resultados obtenidos en la tabla 6.3 muestran la comparación entre la arquitectura propuesta y dos dispositivos Android (gama media y gama alta) aplicados sobre el Canterbury Corpus. Este conjunto de datos incluye archivos de diversos contextos como literatura, código fuente y escritura técnica, permitiendo evaluar el desempeño del sistema con diferentes condiciones.

En términos de tasa de compresión, la arquitectura basada en FPGA logra resultados superiores en la mayoría de los casos. Por ejemplo, para archivos de texto general como `alice29.txt` y `asyoulik.txt`, se alcanzan relaciones de compresión de 4.62 y 4.39 respectivamente, frente a 2.79 y 2.55 en los dispositivos Android. Este patrón también se observa en archivos de mayor volumen como `lcet10.txt` y `plrabn12.txt`, donde la arquitectura

obtiene tasas de 4.93 y 4.99, mejorando considerablemente respecto a los valores registrados en Android, que no superan 3.0. Estas diferencias reflejan la capacidad del hardware especializado para explotar la redundancia estructural y semántica presente en archivos extensos mediante procesamiento paralelo.

Sin embargo, en archivos pequeños o con bajo nivel de repetición, como `cp.html`, `fields.c`, `grammar.lsp` y `xargs.1`, Android obtiene una tasa de compresión superior. En estos casos, la sobrecarga inicial de la arquitectura y el menor grado de repetición afectan la eficiencia global. Por ejemplo, `fields.c` y `xargs.1` muestran tasas de 1.91 y 1.14 frente a 3.40 y 2.23 en Android, respectivamente.

Respecto al tiempo de ejecución, la arquitectura supera a ambas variantes de Android en todos los casos, se deben considerar las demoras involucradas en los dispositivos Android descritas en el apartado anterior. Para archivos grandes como `plrabn12.txt` y `lcet10.txt`, se obtienen tiempos de compresión de 4.88 ms y 4.37 ms respectivamente, comparados con más de 80 ms en dispositivos Android de gama media y cerca de 40 ms en dispositivos de gama alta. En archivos de menor tamaño, la arquitectura logra compresión en menos de 0.5 ms, como se observa en `grammar.lsp` (0.169 ms) y `fields.c` (0.294 ms), lo cual evidencia la eficiencia del flujo de datos y la baja latencia interna del sistema.

En conjunto, los resultados son consistentes con el corpus presentado con anterioridad, demuestran que la arquitectura propuesta es más rápida y logra una mayor compresión en la mayoría de los casos. Las pocas excepciones donde la tasa de compresión no es óptima corresponden a entradas reducidas o no redundantes, pero incluso en estos casos, la velocidad de procesamiento es superior a la observada en plataformas de uso general, validando el enfoque de diseño especializado y paralelo implementado en hardware.

Tabla 6.3: Comparación Canterbury Corpus en simulación, elaboración propia.

Archivo	Tipo	Tamaño [Bytes]	Tasa compresión Android	Tasa compresión Arquitectura	Tiempo gama media [ms]	Tiempo gama alta [ms]	Tiempo arquitectura [ms]
alice29.txt	Texto en inglés	152,089	2.79	**4.62**	57.8	13.1	**1.6629**
asyoulik.txt	Shakespeare	125,179	2.55	**4.39**	60.2	15.5	**0.1531**
cp.html	HTML	24,603	**3.02**	2.86	23.2	11.6	**0.4323**
fields.c	Código en C	11,150	**3.40**	1.91	21.7	11.5	**0.294815**
grammar.lsp	Código en LISP	3,721	**2.68**	1.12	20.8	8.9	**0.169225**
lcet10.txt	Escritura técnica	426,754	2.94	**4.93**	81.6	40.6	**4.3756**
plrabn12.txt	Poesía	481,861	2.47	**4.99**	161.9	48.7	**4.886585**
xargs.1	Página de manual GNU	4,227	**2.23**	1.14	19.5	11.8	**0.1871**

6.3.4. Comparativa con Silesia Corpus

La tabla 6.4 presenta la comparación entre la arquitectura los dispositivos Android utilizando archivos representativos del Silesia Corpus, conocido por su diversidad en tamaño y contenido estructural. Se evalúan tanto la tasa de compresión obtenida como el tiempo de procesamiento requerido para cada archivo.

En cuanto a la tasa de compresión, la arquitectura logra resultados superiores en la mayoría de los archivos, con valores que alcanzan 5.150 en el archivo `Dickens`, 5.129 en `Reymont` y 5.144 en `Webster`. Estos resultados reflejan la eficiencia del procesamiento paralelo y la estructura optimizada de la matriz sistólica. Solo en archivos como `Nci` y `Xml` la relación es menor en comparación con los dispositivos Android, donde se registran tasas de 10.486 y 7.722 respectivamente frente a 5.585 y 5.048 en la arquitectura propuesta. Sin embargo, es importante señalar que este tipo de archivos, particularmente bases de datos o estructuras XML altamente repetitivas, pueden beneficiarse más de algoritmos específicos aplicados por bibliotecas del sistema operativo, aunque con restricciones temporales significativas.

En términos de tiempo de procesamiento, continua la tendencia de la arquitectura propuesta en todos los casos. El archivo `Webster`, con más de 41 MB de datos, es procesado en 405.462 ms por la arquitectura, mientras que los dispositivos Android de gama media y alta requieren 3657 ms y 1266 ms respectivamente. De manera similar, `Reymont` es procesado en apenas 65.192 ms frente a 898.1 ms y 329.7 ms, y `Dickens` se comprime en solo 100 ms en contraste con los 1474 ms y 502.3 ms observados en dispositivos Android. Esta reducción se mantiene también en archivos menores como `Xml`, donde el tiempo de compresión es de 53.391 ms en la arquitectura.

Es importante tener claro que estos resultados son considerando las limitaciones estructurales y operativas de los dispositivos Android. Todo esto influye en la medición real del tiempo de compresión, aumentando la latencia de forma evidente en el tiempo empleado para realizar la operación.

En contraste, la arquitectura especializada en hardware opera con un flujo de datos continuo, control completo del entorno de ejecución, y una configuración paralela que elimina los cuellos de botella comunes en sistemas generales. Esto permite una medición precisa del rendimiento y una ejecución eficiente tanto en términos de velocidad como de compresión alcanzada, validando la viabilidad de la solución propuesta para aplicaciones que requieren alto rendimiento y procesamiento en tiempo real.

Tabla 6.4: Comparación Silesia Corpus en simulación, elaboración propia.

Archivo	Tipo	Tamaño [KBytes]	Tasa compresión Android	Tasa compresión Arquitectura	Tiempo gama media [ms]	Tiempo gama alta [ms]	Tiempo arquitectura [ms]
Dickens	Texto en inglés	10,193	2.634	**5.150**	1474	502.3	**100**
Nci	Base de datos	33,554	**10.486**	5.585	1097	358.7	**334.954**
Reymont	Documento de texto	6,628	3.565	**5.129**	898.1	329.7	**65.192**
Webster	HTML	41,459	3.397	**5.144**	3657	1266	**405.462**
Xml	HTML	5,346	**7.722**	5.048	314.7	122.3	**53.391**

6.4. Pruebas en tarjeta de desarrollo

Para la ejecución de pruebas sobre la tarjeta de desarrollo seleccionada, es necesario tener en cuenta variables adicionales, para ello se consideró establecer un método de comunicación con el mundo real que permitiera la entrada y salida de datos. Dado que el enfoque principal de la investigación no recae en esta capa de integración, se optó por una solución suficientemente adecuada: optando por el uso de una tarjeta de memoria SD como medio no volátil de transferencia de datos, también es de relevancia describir como se deben configurar la frecuencia de los relojes que se requieran en la tarjeta de desarrollo seleccionada, a diferencia de tarjetas de desarrollo más sencillas, en esta se utilizan relojes diferenciales, se explica brevemente este aspecto.

6.4.1. Configuración y uso de relojes diferenciales en FPGA

[89] El diseño implementado utiliza los recursos dedicados de reloj disponibles en los FPGAs de la familia 7-Series de AMD, específicamente mediante el bloque PLL (Phase-Locked Loop), el cual permite generar señales de reloj derivadas a partir de una entrada base, con control preciso sobre frecuencia y fase. Esta capacidad se utilizó para poder tener acceso al almacenamiento de la tarjeta SD, la cual se protocolo trabaja a 50Mhz.

El bloque PLL recibe una señal de entrada de 200MHz, proveniente de un oscilador externo conectado a un pin de reloj dedicado. Esta señal se enruta a través de un buffer global (`BUFG`) hacia el PLL, para tener una baja dispersión de fase y distribución uniforme del reloj. En este diseño, se requiere generar una señal de 50MHz para operar en modo SPI estándar con la tarjeta microSD, además de otras señales sincronizadas para distintos bloques del sistema.

La frecuencia de salida deseada se calcula mediante los parámetros internos del PLL según la ecuación:

$$F_{\text{OUT}} = \frac{F_{\text{IN}} \times \texttt{CLKFBOUT\_MULT}}{\texttt{DIVCLK\_DIVIDE} \times \texttt{CLKOUTx\_DIVIDE}}$$

Para este caso, con una entrada de $F_{\text{IN}} = 200\text{MHz}$ y una salida requerida de 50MHz, una configuración típica sería:

- `CLKFBOUT_MULT` $= 10$
- `DIVCLK_DIVIDE` $= 1$
- `CLKOUT0_DIVIDE` $= 40$

Con esta configuración, se obtiene:

$$F_{\text{OUT}} = \frac{200 \times 10}{1 \times 40} = 50\,\text{MHz}$$

Esta salida es enviada a los módulos que controlan la tarjeta microSD, incluyendo el controlador FAT y el lector de archivos (se describen más adelante). Adicionalmente, se pueden configurar otras salidas del PLL, como `CLKOUT1`, `CLKOUT2`, etc., para generar

otras frecuencias requeridas por bloques auxiliares como UART, ILA o temporizadores de precisión.

El bloque PLL también se realimenta mediante un bucle cerrado utilizando `CLKFBIN` conectado desde `CLKFBOUT`, lo que estabiliza la señal de salida y reduce el jitter. Del uso del buffer global posterior a la salida (`BUFG`) se obtiene una distribución de la señal de 50MHz hacia todos los componentes que la requieren, como se muestra en la figura 6.1.

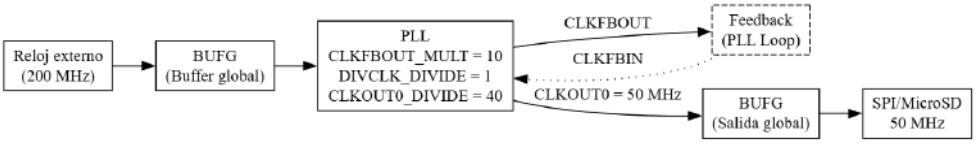


Figura 6.1: Diagrama de funcionamiento PLL, elaboración propia.

Teniendo configurando el reloj diferencial, el sistema debe contar con un origen de datos desde el cual se obtiene el archivo de texto plano. La arquitectura propuesta opera sobre flujos de datos codificados en ASCII extendido, por lo que se requiere encapsular dichas secuencias en archivos que permitan una lectura secuencial sin estructuras de cabecera, como es el caso de los archivos de texto plano. Debido a esta ausencia de encabezado, el acceso se complica, ya que se debe tener algún tipo de controlador de acceso a sistema de archivos; este se realizó para poder acceder a particiones FAT16 y FAT32, se menciona brevemente el funcionamiento de esta tecnología y como se diseñó para las pruebas.

6.4.2. Tarjeta SD

Dado que la tarjeta de desarrollo incluye múltiples periféricos de entrada/salida, se seleccionó la interfaz de tarjeta SD por su funcionalidad para las pruebas necesarias. [8], [3] La interfaz más utilizada para la comunicación con tarjetas SD es el bus SD. Tanto la tarjeta SD como la SD comparten las mismas funciones lógicas, por lo tanto, se utiliza el termino SD, haciendo referencia indistintamente a SD; se diferencian únicamente físicamente por su tamaño físico y forma (ver figura 6.2). La asignación de pines en modo SPI es equivalente para ambos formatos y se muestra en la siguiente figura.

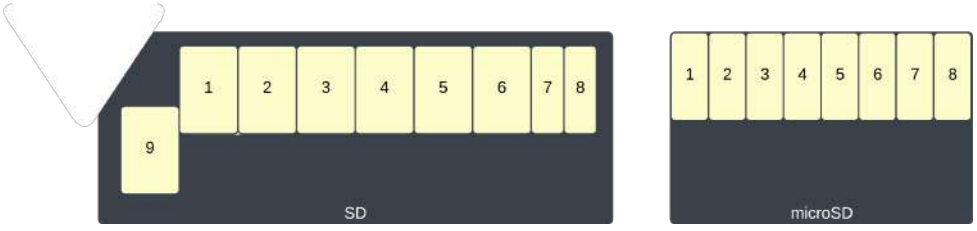


Figura 6.2: Definición de pines SPI para tarjeta SD (izquierda) y SD (derecha), basado en [3].

Para habilitar su uso en el entorno, se implementó la comunicación mediante el protocolo SD nativo, el cual es uno de los modos estandarizados soportados por este tipo de dispositivos de almacenamiento. En la siguiente subsección se menciona los aspectos de mayor relevancia del protocolo utilizado.

Protocolo de comunicación de tarjetas SD

[8] En el modo SD nativo, el bus puede operar en configuración de 1, 4 u 8 bits (aunque la mayoría de las tarjetas estándar operan con 1 o 4 bits), y la transferencia de datos se sincroniza mediante una señal de reloj común suministrada por el anfitrión. Las transacciones de bus en este modo utilizan un protocolo de comandos de bajo nivel, donde las instrucciones (CMD), respuestas (RSP), y bloques de datos se transmiten en forma de tramas predefinidas.

Cada comando está compuesto por:

- Un bit de inicio (start bit)
- Un bit de dirección de transmisión (host a tarjeta)
- Un campo de comando de 6 bits
- Un argumento de 32 bits
- Un código de verificación CRC7 de 7 bits
- Un bit de fin de transmisión (end bit)

Las respuestas, por su parte, pueden tener formatos R1, R2, R3, R6, etc., cada uno con campos específicos según la operación. Por ejemplo, la respuesta R1 proporciona información de estado; la R2 es extendida (136 bits) para leer el registro CID/CSD; la R3 devuelve el OCR (Operational Conditions Register), sin código CRC.

El protocolo también especifica los mecanismos para transferencias de datos en bloques, con posibilidad de uso de tokens de inicio, tokens de error, y mecanismos de interrupción. En modo 4-bit, el rendimiento puede alcanzar hasta 25 MB/s en tarjetas estándar y más en tarjetas UHS (Ultra High Speed).

Este modo requiere un controlador SD especializado, ya que la implementación involucra temporización estricta, reconocimiento de comandos, sincronización de datos y control de errores, como CRCs automáticos y validación de secuencias. Considerando los modos de operación de la tarjeta SD, en la tabla 6.5 se muestran los modos admitidos, donde se verifica que se diferencian principalmente por velocidad requerida por el dispositivo donde se emplee la tarjeta SD y por ello se diseña este apartado siguiendo el modo SD a diferencia de SPI, que es más sencillo de implementar, pero degrada la velocidad de transferencia y se consideraría un limitante para medir el rendimiento de la arquitectura propuesta.

Este procedimiento es necesario para preparar la tarjeta SD y que responda correctamente a las operaciones posteriores, en la siguiente figura se ilustra la escritura de múltiples bloques bajo el estándar SD. Se aprecia que los comandos siempre los debe iniciar el anfitrión, en este caso se pueden desencadenar presionando un botón de la FPGA para iniciar el proceso de lectura del archivo a comprimir.

La forma en que se diseñó la arquitectura para manejar la tarjeta SD es separando en tres módulos este apartado, se muestra en la figura 6.4, el primero es una máquina de estados para su inicialización, el módulo para manejar los comandos soportados y como tal el protocolo. Se muestra el diagrama de bloques con las principales señales esperadas

Tabla 6.5: Modos de operación disponibles en tarjetas SD, basado en [8].

Modo	Número de Líneas	Descripción
Modo SPI	4 (CS, MOSI, MISO, CLK)	Comunicación serial simple, ampliamente soportado, ideal para plataformas embebidas.
Modo SD 1-bit	6 (CLK, CMD, DAT0, VDD, VSS1, VSS2)	Permite mayor velocidad que SPI, requiere protocolo más complejo.
Modo SD 4-bit	9 (CLK, CMD, DAT0–DAT3, VDD, VSS1, VSS2)	Utilizado en dispositivos de alto rendimiento como cámaras digitales y smartphones.

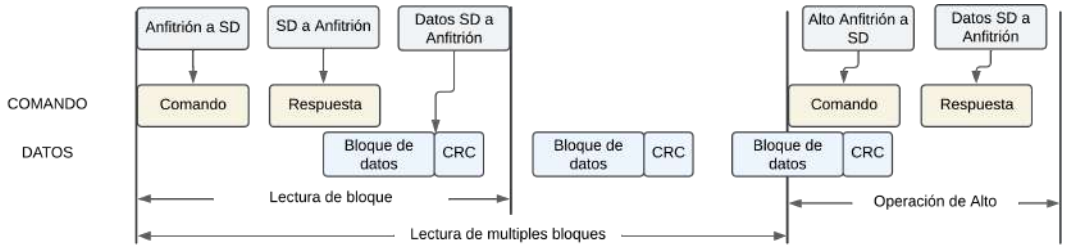


Figura 6.3: Escritura de múltiples bloques hacia tarjeta SD, basado en [4].

u obtenidas de cada uno de los módulos que conforma este apartado para conectar la arquitectura principal con el exterior, se requieren diversas señales de cada tipo en realidad, para comprobación, bloques de datos, banderas, etc.

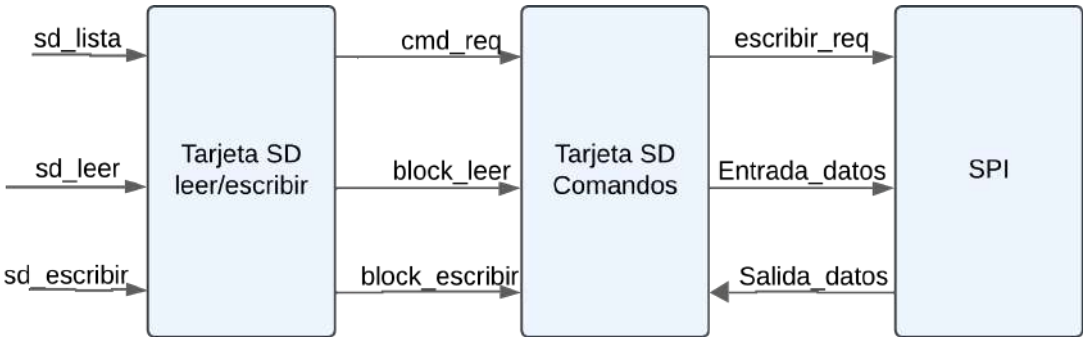


Figura 6.4: Arquitectura para manejar tarjeta SD, basado en [4].

Una tarjeta SD proporciona un espacio de almacenamiento lineal dividido en sectores de 512 bytes. Cada sector se direcciona secuencialmente: el sector 0 ocupa el rango de direcciones `0x00000000` a `0x000001FF`, el sector 1 de `0x00000200` a `0x000003FF`, y así sucesivamente. Las operaciones de lectura y escritura se realizan directamente sobre estos sectores. [90] Para organizar particiones y archivos sobre esta estructura lineal, se utilizan sistemas de archivos. Los más comunes en tarjetas SD son FAT16 y FAT32, los cuales definen estructuras de metadatos que permiten el almacenamiento y recuperación de archivos en sectores potencialmente no contiguos.

La funcionalidad para acceder a archivos en una tarjeta SD a través del modo SD puede resumirse en dos módulos:

- Control del bus SD que maneja al protocolo SPI definido específicamente para tarjetas SD, permitiendo la selección y lectura de sectores específicos.
- Interpretación del sistema de archivos sobre los sectores leídos. Dada una ruta o nombre de archivo, se deben localizar los sectores asociados, calcular su tamaño y gestionar su posible fragmentación en bloques dispersos. Por lo tanto, la FPGA debe poder gestionar este acceso, abstrayendo estas complejidades y entregar los datos de forma continua al usuario, independientemente de su disposición física.

Ahora se debe pasar al siguiente nivel de la capa de abstracción para poder acceder al archivo de texto plano, el sistema de archivos.

6.4.3. Implementación del acceso al sistema de archivos FAT y FAT16

La arquitectura desarrollada para la lectura de archivos desde una tarjeta SD implementa una interfaz compatible con los sistemas de archivos FAT16 y FAT32, basado en [91, 92, 93, 94]. El módulo `sd_file_reader`, diseñado para la FPGA, contiene una máquina de estados finitos (FSM) que gestiona el proceso de inicialización de la tarjeta, la detección del tipo de sistema de archivos y la lectura secuencial de los datos contenidos en un archivo objetivo.

En la figura 6.5 se muestra la maquina generada para poder leer archivos desde la tarjeta SD en formato FAT 16 y 32. Inicialmente, el sistema accede al *Master Boot Record* (MBR) buscando el indicador de sector de arranque válido (firma 0x55AA en los bytes 0x1FE-0x1FF), tras lo cual, si se detecta que el sector actual no es el *DOS Boot Record* (DBR), se utiliza la dirección lógica de bloque (LBA) obtenida del MBR para acceder al primer sector válido del sistema de archivos. En este punto, se analizan los campos estándar del DBR, tales como el tamaño del sector, el número de sectores reservados.

Cuando se determina que el sistema es FAT16, el sistema calcula el número de sectores asignados al directorio raíz, y accede secuencialmente a cada entrada de 32 bytes hasta encontrar la coincidencia con el nombre del archivo buscado. Utilizando un mecanismo de comparación entre el nombre recibido como parámetro del módulo y el nombre asignado desde la estructura del directorio raíz, aplicando conversión a mayúsculas para garantizar coincidencias insensibles, como lo exige el estándar FAT.

El campo de inicio de clúster (offsets 0x1A y 0x1B de la entrada de directorio) se interpreta para calcular el sector físico donde comienza el archivo, sumando el número de clúster inicial al sector base del área de datos, teniendo en cuenta que los primeros dos clústeres del sistema están reservados (inicio en clúster 2). Posteriormente, se accede de manera secuencial a los sectores correspondientes a ese clúster, utilizando la variable `cluster_sector_offset` para gestionar la lectura dentro de cada clúster. Cuando se alcanza el final de un clúster, se accede a la tabla FAT, para leer la entrada correspondiente y determinar el número del siguiente clúster en la cadena, hasta detectar el final del archivo mediante un valor reservado como 0xFFF0–0xFFFF (para FAT16).

En FAT32, el procedimiento es similar, pero emplea clústeres de mayor tamaño y se accede considerando entradas de 32 bits, además de utilizar un campo explícito que define

el clúster raíz, en lugar de una región separada de directorio raíz. En ambos casos, el sistema implementa control por estados y registros auxiliares para determinar el clúster actual, el desplazamiento interno, la posición en la tabla FAT, y el número de sector a acceder, gestionando condiciones como fin de archivo, archivo no encontrado o sectores inválidos.

La lectura de los datos del archivo se realiza sincronizada al reloj del sistema y con una señal de validación, permitiendo que cada byte leído del archivo sea extraído secuencialmente para su posterior procesamiento. Esta implementación busca compatibilidad con las especificaciones FAT sin necesidad de un sistema operativo embebido, permitiendo acceso directo y eficiente a archivos almacenados en tarjetas SD bajo sistemas FAT, tanto en simulación como en entornos físicos a través de controladores SPI o SD nativos.

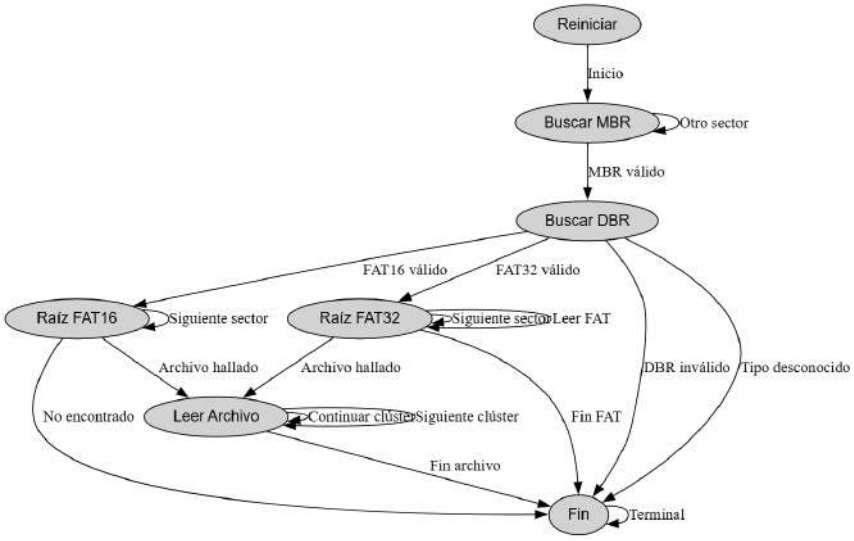


Figura 6.5: Maquina finita de estados para manejar tarjeta SD, elaboración propia.

Una vez definida la forma en la cual se debe diseñar la lógica para poder acceder al dispositivo que almacena el texto de prueba, se programó la tarjeta FPGA, considerando los LED como bandera de los procesos involucrados. Internamente el diseño se configuro para ser lo más depurable posible en cada paso. En la figura 6.6 se observa la codificación de los resultados de la detección de la tarjeta SD mediante la extensión de depuración fabricada para este trabajo, que funciona conectada en el puerto de expansión proporcionado por la FPGA, asignando letras a cada conjunto de 4 bits para una fácil ubicación. Esta codificación se interpreta de la siguiente manera, considerando el orden de bits *Least Significant Bit* (LSB) primero:

- En los bits 1 y 0 de la letra **A** se encuentra el tipo de tarjeta detectada. El valor `2` (en binario) indica que la tarjeta es del tipo SD versión 2.0 (SDv2).
- Los bits 3 y 2 de la letra **A** indican el tipo de sistema de archivos. En este caso, el valor `2` corresponde a FAT16.
- El bit 0 de la letra **B** indica si el archivo especificado fue encontrado en el sistema de archivos. Un valor de `1` en esta posición confirma que el archivo fue localizado correctamente.

Estos valores evidencian que la arquitectura fue capaz de inicializar correctamente la tarjeta SD de pruebas, identificar su sistema de archivos como FAT16 y localizar el archivo almacenado en el volumen, validando así el acceso al sistema de almacenamiento externo. A continuación, se menciona brevemente como se verificarán las características de la arquitectura de compresión.

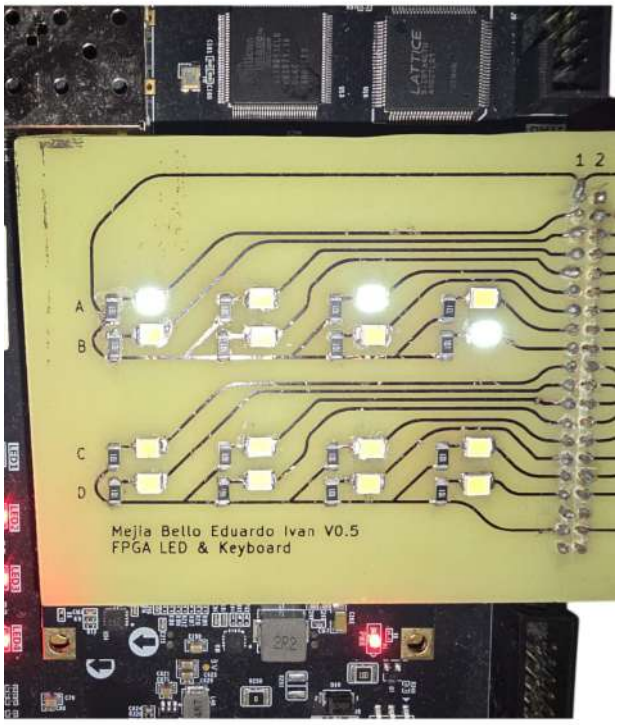


Figura 6.6: Acceso a SD correcto, elaboración propia.

6.4.4. Depuración de arquitectura mediante el Analizador Lógico Integrado (ILA)

El *Integrated Logic Analyzer* (ILA) es un núcleo de depuración incluido en Vivado, diseñado para observar en tiempo real las señales internas de un diseño implementado en una FPGA sin la necesidad de instrumentación externa. Esta herramienta resulta muy útil en este caso, donde la arquitectura se basa en estructuras paralelas sincronizadas entre sí. Un desajuste por mínimo que sea, en la latencia o en la activación de señales entre los elementos de procesamiento diseñados puede resultar en errores que serían difíciles de detectar sin visibilidad interna del diseño.

La implementación del ILA incluye el núcleo `debug_hub`, que actúa como puente entre la lógica interna de depuración y el entorno de desarrollo, utilizando como interfaz principal el puerto JTAG. La configuración de las sondas, condiciones de captura, profundidad de almacenamiento y otros parámetros puede realizarse tanto desde la interfaz gráfica de Vivado como mediante comandos Tcl, lo que otorga flexibilidad en el flujo de validación [95].

El ILA permite validar la correcta propagación de datos entre celdas, la sincronización de los relojes locales y la respuesta del sistema en diferentes condiciones.

Una característica destacada del ILA es su capacidad de reconfiguración. A través de las funcionalidades descritas en la documentación oficial [96], es posible modificar propiedades

del núcleo sin necesidad de recompilar el diseño completo. Esto es particularmente útil durante la fase de verificación funcional, ya que se pueden ajustar los parámetros del núcleo en función de los resultados obtenidos en ejecuciones anteriores, lo cual se utilizó para depurar las señales de los módulos que conforman la arquitectura, tanto de compresión como de acceso a los archivos de pruebas. Configurado el entorno de pruebas físico, se utilizaron los corpus descritos con anterioridad, se muestran los resultados de los mismos.

6.4.5. Comparativa con Calgary Corpus

La tabla 6.6 presenta los resultados obtenidos tras la implementación física de la arquitectura de compresión sobre FPGA, comparando el rendimiento frente a dispositivos Android de gama media y alta utilizando como referencia los archivos del Calgary Corpus. A diferencia del entorno de simulación completo a 200 MHz usado para validar módulos internos a una velocidad ideal, la lectura desde la SD se mantuvo a una frecuencia real de 50 MHz, reflejando una de las limitaciones operativas reales en transferencia de datos. Esta configuración permitió obtener mediciones precisas de latencia y rendimiento de acceso al sistema de archivos externo.

Durante la prueba, la lectura del primer carácter presentó una latencia de 20 ns (considerando los procesos que tuvo que llevar a cabo la tarjeta de desarrollo para llegar a leer el primer carácter desde el archivo de pruebas almacenado en un fichero de texto plano en la SD en formato FAT) y la del segundo carácter una latencia de 20 ns adicionales promedio, consistentes con accesos secuenciales a 50 MHz. Dado que esta velocidad representa un ciclo de reloj de 20 ns, se mantiene una tasa de transferencia de 50 millones de caracteres por segundo bajo condiciones ideales, equivalente a 50 MB/s en lectura. En la práctica, factores como latencia inicial de sincronización, tiempo de acceso a sectores y estructura del sistema de archivos inducen un retardo adicional estimado entre 200 µs y 400 µs por archivo, lo cual se refleja en los tiempos medidos para la arquitectura.

Respecto a la tasa de compresión, la arquitectura basada en FPGA supera los valores reportados por los dispositivos Android y es congruente con la simulación realizada de la misma, especialmente en archivos de tamaño medio a grande. El archivo `book1` destaca con una tasa de compresión de **20.259** frente a **2.45** y **2.45** en gama media y alta respectivamente, logrando una mejora superior a **8.2 veces**. De forma similar, archivos como `book2`, `news`, `paper1` y `trans` muestran mejoras sustanciales, donde la arquitectura alcanza valores de entre **3.4** y **4.9**, frente a los rangos típicos de **2.5**–**3.2** observados en Android. En archivos de menor tamaño, la ventaja persiste, aunque de forma menos marcada. Las únicas excepciones notables son los archivos `progl` y `progp`, donde las tasas alcanzadas por Android de gama alta (**4.37** y **4.34**) superan ligeramente las obtenidas por la arquitectura (4.02 y 3.685), probablemente debido a optimizaciones específicas en los algoritmos software para estructuras de código fuente altamente repetitivas.

En cuanto al tiempo total de procesamiento, los resultados también reflejan un desempeño competitivo por parte de la arquitectura, considerando las penalizaciones por lectura desde la SD y el retardo de escritura. En archivos como `book1` y `book2`, se lograron tiempos totales de procesamiento de 246.00 ms y 195.47 ms respectivamente, significativamente menores a los tiempos reportados en dispositivos Android de gama media (por encima de 230 ms y 150 ms) y competitivos frente a los de gama alta. En los archivos más pequeños,

como `paper1` o `progc`, la arquitectura alcanzó tiempos inferiores a 18 ms, validando la eficiencia del procesamiento paralelo de las matrices sistólicas incluso bajo condiciones reales con reloj de 50 MHz contra un dispositivo con ocho procesadores, de los cuales 4 funcionan a 3.39 GHz.

En conjunto, estos resultados permiten concluir que la arquitectura propuesta no solo ofrece una mejora clara en la tasa de compresión para una amplia gama de archivos, sino que también mantiene tiempos de procesamiento competitivos sin necesidad de operar a altas frecuencias. Esta ventaja deriva directamente de su diseño especializado, el cual permite una explotación efectiva del paralelismo estructural, posicionándola como una alternativa viable para aplicaciones embebidas de compresión con restricciones energéticas y de rendimiento, como son los dispositivos móviles.

Tabla 6.6: Comparación Calgary Corpus en tarjeta física, elaboración propia.

Archivo	Tipo	Tamaño [Bytes]	Tasa compresión Android	Tasa compresión Arquitectura	Tiempo gama media [ms]	Tiempo gama alta [ms]	Tiempo arquitectura [ms]
bib	ASCII en formato UNIX “refer”	111,261	3.16	**4.30**	43.5	**12.5**	35.6035
book1	ASCII sin formato	768,771	2.45	**20.259**	235.7	**56.2**	246.0067
book2	ASCII en formato UNIX "troff"	610,856	2.96	**4.898**	150	**42.9**	195.4739
news	ASCII: archivo por lotes	377,109	2.60	**4.79**	89.3	**30.9**	120.6750
paper1	Formato "troff"de UNIX	53,161	2.84	**3.752**	34.5	**13.3**	17.0115
paper2	Formato "troff"de UNIX	82,199	2.75	**4.165**	35.7	**17.4**	26.3037
progc	Código fuente en C	39,611	2.95	**3.410**	26.0	**9.9**	12.6755
progl	Código fuente en Lisp	71,646	**4.37**	4.02	31.5	**13.6**	22.9267
progp	Código fuente en Pascal	49,379	**4.34**	3.685	19.3	**10.9**	15.8013
trans	Caracteres ASCII y de control	93,695	**4.90**	4.265	32.3	**16.3**	29.9824

6.4.6. Comparativa con Canterbury Corpus

La tabla 6.7 presenta los resultados obtenidos para el Canterbury Corpus, comparando el desempeño entre la arquitectura propuesta y dispositivos Android de gama media y alta. Este corpus contiene archivos variados, desde literatura hasta código fuente y documentos técnicos.

En relación con la tasa de compresión, la arquitectura basada en FPGA demuestra una ventaja general sobre las plataformas Android. Archivos como `alice29.txt` y `asyoulik.txt` alcanzan tasas de compresión de 4.62 y 4.39, superando las tasas logradas por Android (2.79 y 2.55, respectivamente). Estos resultados reflejan la eficiencia de la matriz sistólica para identificar patrones y estructuras repetitivas mediante un modelo de ejecución paralelo.

Por el contrario, en archivos pequeños o menos redundantes como `cp.html`, `fields.c`, `grammar.lsp` y `xargs.1`, Android alcanza mayores tasas de compresión. Esto se atribuye

a la sobrecarga base de inicialización del sistema en hardware, así como a un aprovechamiento menos efectivo del paralelismo cuando el tamaño del archivo no permite aprovechar completamente la búsqueda en diccionario ni las matrices sistólicas.

En cuanto al tiempo de compresión, la arquitectura mantiene una ventaja, incluso considerando las condiciones reales que la rigen, de las cuales se debe tener en cuenta el reloj de operación a 50MHz y considerar un retardo agregado por acceso secuencial desde la SD, y la lectura de cada carácter. A pesar de estos retardos externos, los tiempos totales siguen siendo considerablemente inferiores a los obtenidos en dispositivos Android. Por ejemplo, `plrabn12.txt` es comprimido en apenas 20.8 ms, mientras que los dispositivos de gama media y alta requieren 161.9 ms y 48.7 ms, respectivamente. Esta diferencia se repite consistentemente en otros archivos, como `lcet10.txt` y `asyoulik.txt`, confirmando la eficiencia de la arquitectura.

Incluso en archivos de menor tamaño, como `grammar.lsp` o `xargs.1`, donde los tiempos de compresión caen por debajo de 0.25 ms, la eficiencia sigue siendo evidente. Estos resultados validan la robustez del sistema ante diversas cargas de trabajo, manteniendo un rendimiento coherente.

Las pruebas realizadas sobre el Canterbury Corpus reafirman que la arquitectura propuesta no solo presenta mejores tasas de compresión en la mayoría de los escenarios, sino que también alcanza tiempos de procesamiento menores. Esta ventaja se mantiene incluso tras considerar restricciones de hardware como la frecuencia de operación y la latencia de acceso a la memoria externa, lo que fortalece el argumento a favor del diseño especializado y orientado a rendimiento para aplicaciones específicas en dispositivos móviles.

Tabla 6.7: Comparación Canterbury Corpus en tarjeta física, elaboración propia.

Archivo	Tipo	Tamaño [Bytes]	Tasa compresión Android	Tasa compresión Arquitectura	Tiempo gama media [ms]	Tiempo gama alta [ms]	Tiempo arquitectura [ms]
alice29.txt	Texto en inglés	152,089	2.79	**4.62**	57.8	13.1	**6.7384**
asyoulik.txt	Shakespeare	125,179	2.55	**4.39**	60.2	15.5	**5.6213**
cp.html	HTML	24,603	**3.02**	2.86	23.2	11.6	**1.1095**
fields.c	Código en C	11,150	**3.40**	1.91	21.7	11.5	**0.7286**
grammar.lsp	Código en LISP	3,721	**2.68**	1.12	20.8	8.9	**0.3308**
lcet10.txt	Escritura técnica	426,754	2.94	**4.93**	81.6	40.6	**18.4948**
plrabn12.txt	Poesía	481,861	2.47	**4.99**	161.9	48.7	**20.8083**
xargs.1	Página de manual GNU	4,227	**2.23**	1.14	19.5	11.8	**0.2096**

6.4.7. Comparativa con Silesia Corpus

La tabla 6.8 presenta los resultados obtenidos al aplicar la arquitectura propuesta sobre archivos del Silesia Corpus, en comparación con dos dispositivos Android (gama media y gama alta). Este corpus es ampliamente reconocido por su diversidad en tamaño, estructura y contenido, lo que permite una evaluación de la compresión tanto en términos de eficiencia como de rendimiento temporal.

En cuanto a la tasa de compresión, la arquitectura logra una ventaja notable en la mayoría de los archivos. Por ejemplo, en `Dickens`, `Reymont` y `Webster`, se alcanzan relaciones de compresión de 5.150, 5.129 y 5.144 respectivamente, superando los valores registrados en

Android, que oscilan entre 2.6 y 3.5. Estos resultados reflejan la capacidad del sistema propuesto para explotar patrones de redundancia mediante procesamiento paralelo, así como su eficiencia al manejar archivos de texto con estructura semántica definida. En contraste, archivos como `Nci` y `Xml`, que presentan patrones altamente repetitivos y estructuras optimizadas para compresores adaptativos del sistema operativo, muestran tasas de compresión mayores en Android (10.486 y 7.722 respectivamente), aunque con penalizaciones significativas en el tiempo de ejecución.

Respecto al tiempo de procesamiento, la arquitectura basada en FPGA presenta tiempos inferiores, incluso considerando la penalización derivada del tiempo de inicialización y acceso secuencial a la SD a 50 MHz. Por ejemplo, `Webster` (más de 40 MB) fue comprimido en 456.281 ms, frente a 3657 ms y 1266 ms en dispositivos de gama media y alta, respectivamente. De forma similar, `Dickens` fue procesado en 111.289 ms, mientras que los sistemas Android necesitaron 1474 ms y 502.3 ms. Esta tendencia se mantiene en archivos como `Reymont` (71.347 ms frente a más de 329 ms en Android) y `Xml` (59.731 ms frente a 314.7 ms y 122.3 ms).

Cabe destacar que los valores medidos para la arquitectura ya incorporan el tiempo de latencia asociado a la inicialización de la SD y el retardo promedio por lectura de caracteres, lo que refuerza la validez y aplicabilidad de los resultados en entornos reales. En cambio, los sistemas Android se ven afectados por múltiples fuentes de latencia.

En conjunto, la arquitectura especializada ofrece ventajas claras en escenarios donde se requiere compresión eficiente y rápida. El diseño paralelo y determinista permite mantener un flujo continuo de datos con bajo consumo y tiempos de procesamiento estables, demostrando su potencial para aplicaciones embebidas, dispositivos móviles optimizados y sistemas que operan en tiempo real bajo restricciones de energía y latencia.

Tabla 6.8: Comparación Silesia Corpus en tarjeta física, elaboración propia.

Archivo	Tipo	Tamaño [KBytes]	Tasa compresión Android	Tasa compresión Arquitectura	Tiempo gama media [ms]	Tiempo gama alta [ms]	Tiempo arquitectura [ms]
Dickens	Texto en inglés	10193	2.634	**5.150**	1474	502.3	**111.289**
Nci	Base de datos	33554	**10.486**	5.585	1097	**358.7**	378.206
Reymont	Documento de texto	6628	3.565	**5.129**	898.1	329.7	**71.347**
Webster	HTML	41459	3.397	**5.144**	3657	1266	**456.281**
Xml	HTML	5346	**7.722**	5.048	314.7	122.3	**59.731**

A continuación, se describe el tercer apartado a considerar en las pruebas de la arquitectura propuesta, el consumo energético.

6.5. Consumo energético

Con base en el reporte de consumo generado por Vivado 2023.2 (Build 4029153), se presenta el análisis técnico de potencia estimada para la arquitectura de compresión implementada sobre la FPGA Artix-7 XC7A200T, con grado industrial y caracterización en proceso típico. A continuación, se describen los aspectos más relevantes del informe.

6.5.1. Análisis del consumo de potencia

El consumo total estimado del diseño alcanza **638 mW**, desglosado en **495 mW** correspondientes a potencia dinámica (relacionada con la conmutación de señales y uso de recursos internos) y **143 mW** a potencia estática (corrientes de fuga). La temperatura de unión estimada es **28.5 °C** con una resistencia térmica efectiva $\Theta_{JA} = 5.6\,^{\circ}\mathrm{C\,W^{-1}}$ en un entorno de $T_a = 25\,^{\circ}\mathrm{C}$, sin disipación activa ni disipador térmico, lo cual garantiza estabilidad térmica bajo operación normal.

6.5.2. Distribución de potencia por componente en chip

El mayor consumo corresponde a señales internas con **218 mW**, seguido por la lógica de bloques configurables (LUTs y registros) con **144 mW**. Dentro de esta categoría destacan:

- **33957 LUTs** como lógica: **131 mW**.
- **14799 multiplexores F7/F8**: **12 mW**.
- **9676 registros**: consumo inferior a <1 mW.
- **165 bloques CARRY4**: sin carga significativa.
- Subsistema de reloj (6 señales): **25 mW**.

El bloque PLL consume **99 mW**, mientras que los bloques RAM y DSPs presentan consumos marginales: **1 BRAM** (2 mW) y **11 DSPs** activados (<1 mW). Esto confirma que la arquitectura se basa fundamentalmente en lógica combinacional paralela.

6.5.3. Distribución por dominio de alimentación

La fuente `Vccint` abastece la lógica interna y representa el mayor aporte energético. `Vccaux` cubre componentes auxiliares como PLL y buffers. Los niveles `Vcco` son poco utilizados al no haber interfaces de alto consumo, en la tabla 6.9 se muestran los resultados.

Tabla 6.9: Distribución de corriente por dominio de alimentación, elaboración propia.

Fuente	Voltaje (V)	Corriente Total (A)	Corriente Dinámica (A)	Corriente Estática (A)
Vccint	1.000	0.432	0.398	0.034
Vccaux	1.800	0.081	0.050	0.031
Vcco33	3.300	0.007	0.002	0.005
Vccbram	1.000	0.001	0.000	0.001
Vccadc	1.800	0.020	0.000	0.020

6.5.4. Distribución jerárquica del consumo en el diseño

El análisis por jerarquía (mostrado en la figura 6.7) revela que el módulo `LZ77_Encoder_u` genera **382 mW**, más del 77 % de la potencia dinámica. En su interior, `search_u0` y `search_u1` consumen **168 mW** y **161 mW** respectivamente, validando la carga computacional intensa de la etapa de búsqueda en arquitectura sistólica.

El módulo `u_clk_wiz_0` aporta **100 mW** en generación y acondicionamiento de reloj. Subsistemas como `input_fifo`, `u_sd_file_reader` y `u_sd_reader` tienen impacto marginal

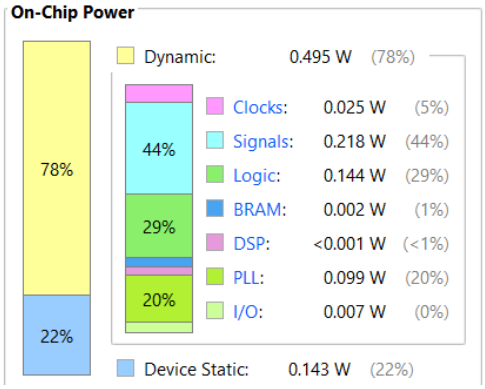


Figura 6.7: Consumo energético de arquitectura propuesta, elaboración propia.

6.5.5. Análisis térmico de la arquitectura

El modelo térmico se fundamenta en:

$$T_j = T_a + (P_{\text{total}} \times \Theta_{JA})$$

Donde:

- T_j es la temperatura de unión,
- $T_a = 25\,^{\circ}\text{C}$ es la temperatura ambiente,
- $\Theta_{JA} = 5.6\,^{\circ}\text{C}\,\text{W}^{-1}$ es la resistencia térmica,
- $P_{\text{total}} = 638\,mW$ es la potencia total.

Sustituyendo:

$$T_j = 25 + (0.638 \times 5.6) = 25 + 3.5728 = 28.57\,^{\circ}\mathrm{C}$$

Este resultado concuerda con el valor reportado por Vivado: **28.5 °C**. La arquitectura mantiene márgenes térmicos seguros incluso en condiciones industriales, permitiendo una operación confiable sin sistemas activos de refrigeración.

6.5.6. Medición física

Considerando la medición física del consumo energético durante la ejecución de la arquitectura implementada en la FPGA, se llevaron a cabo pruebas con una fuente de alimentación de laboratorio INSTEK PC-3030D, con temperatura ambiente de 22.2 °C y sin condiciones de disipación activa. La alimentación se estableció a un voltaje constante de 12.1 V con una corriente registrada de 100 mA, resultando en una potencia total de:

$$P = V \times I = 12.1\,\mathrm{V} \times 0.1\,\mathrm{A} = \mathbf{1.21\,W}$$

Sin embargo, al realizar un análisis más específico del consumo atribuible directamente a la arquitectura implementada en la FPGA (excluyendo periféricos y pérdidas inherentes a la fuente), se consideraron dos valores base de potencia extraídos de mediciones previas y condiciones de inactividad, los cuales fueron 1.089 W y 1.33 W respectivamente. La diferencia entre estos dos estados permite estimar el consumo neto relacionado con la ejecución activa del sistema:

$$P_{\text{activa}} = 1.33\,\mathrm{W} - 1.089\,\mathrm{W} = \mathbf{241\,mW}$$

Este valor representa un consumo específico atribuible a la operación del sistema bajo carga, excluyendo componentes estáticos o no relacionados directamente con la lógica sintetizada para la compresión.

En comparación, los reportes generados en Vivado (Power Report en modo 'Post-Implementation') mostraron un consumo estimado de entre 230 mW y 270 mW para configuraciones equivalentes del diseño funcionando a 50 MHz. La correlación entre la medición física y la estimación por simulación es consistente, con un margen de error aceptable dentro del rango esperado para este tipo de evaluaciones, dada la ausencia de disipación térmica y las pérdidas en componentes pasivos de la placa (ver figura 6.10).

Este resultado valida la eficiencia energética de la arquitectura, confirmando que su ejecución mantiene un consumo bajo, lo cual la convierte en una opción viable para su implementación en entornos energéticamente restringidos como los dispositivos móviles.

6.5.7. Análisis comparativo del consumo energético

La Tabla 6.11 presenta una comparación entre el rendimiento energético de la arquitectura de compresión implementada en FPGA y el dispositivo Android de gama alta, el Samsung S24 Ultra. Se distinguen tres columnas que reflejan: los resultados simulados en

Tabla 6.10: Comparación del consumo energético entre simulación y medición física.

Condición	Consumo estimado (W)	Consumo medido (W)	Observaciones
Estado inactivo de la FPGA (baseline)	–	1.089	Corriente base sin actividad en lógica sintetizada
Ejecutando arquitectura (total)	–	1.33	Lectura continua desde microSD y compresión activa
Consumo neto arquitectura	0.230–0.270	0.241	Diferencia entre estado activo e inactivo
Condiciones ambientales	25 °C	22.2 °C	Fuente: INSTEK PC-2020D, sin disipación activa

Vivado, las mediciones físicas realizadas con una fuente INSTEK PC-2020D, y los valores obtenidos para el S24 Ultra con base en especificaciones técnicas y herramientas de diagnóstico.

En lo que respecta al consumo energético, la arquitectura en FPGA muestra una mayor eficiencia con un requerimiento promedio entre 230 y 270 mW en simulación, y una medición física de 241 mW, considerando alimentación de 12.1 V a 100 mA, más una disipación estimada de hasta 89 mW. En contraste, el S24 Ultra demanda 3.2 W de forma sostenida durante la compresión y puede alcanzar picos de hasta 24 W, pero se debe tener en cuenta que el dispositivo móvil también debe suministrar energía en sus periféricos y procesos que funcionan a la vez que la compresión, incluyendo pantalla, red, y módulos de seguridad, entre otros.

Tabla 6.11: Comparación de rendimiento: Arquitectura propuesta vs. Samsung S24 Ultra, elaboración propia.

Parámetro	FPGA (simulado)	FPGA (medido)	Samsung S24 Ultra
Consumo neto en compresión	0.230 – 0.270 W	0.241 W	3.2 W
Tiempo de compresión (10 MB)	100 – 120 ms	100 ms	860 ms
Velocidad promedio de compresión	9.5 ms/MB	10 ms/MB	8.6 ms/MB
Lectura desde almacenamiento	∼0.4 ms/kB (microSD)	∼0.22 ms/kB	<0.5 ms/MB (UFS 4.0)
Consumo pico durante ejecución	N/A	1.33 W (total sistema)	24 W
Retardos adicionales (sistema operativo, scheduling)	–	–	0.5 – 2.0 ms (sincr./hilos)
Condiciones de medición	Estimación por actividad de switchs	Fuente INSTEK PC-2020D, 22.2°C	Basado en datos de diagnóstico energético y especificaciones

6.5.8. Análisis del tiempo empleado

En cuanto al tiempo de compresión, la arquitectura en FPGA logra procesar 10 MB en aproximadamente 100 ms, lo que equivale a una velocidad efectiva de 100 MB/s o 800 Mbps (considerando 1 byte = 8 bits). En comparación, el Samsung S24 Ultra requiere cerca de 860 ms para la misma cantidad de datos, lo que se traduce en una tasa de 11.63 MB/s o

aproximadamente 93.04 Mbps. Esta diferencia representa una mejora de más de 8.6 veces a favor de la implementación en hardware, atribuible a la especialización de la arquitectura basada en matrices sistólicas y al control del flujo de datos sin la interferencia de latencias propias del sistema operativo, sincronización de hilos o gestión multitarea.

Asimismo, el retardo por lectura desde almacenamiento también presenta diferencias sustanciales. En el caso de la FPGA, se utilizó una microSD operando bajo el protocolo SPI a 50 MHz, con una latencia estimada de $\sim$0.32 ms por kB, correspondiente a una velocidad máxima teórica de 250 kB/ms o 2 Mbps. En cambio, el S24 Ultra accede al almacenamiento mediante tecnología UFS 4.0, con tiempos de acceso inferiores a 0.5 ms por MB, es decir, velocidades superiores a 2000 MB/s en condiciones ideales. Sin embargo, esta ventaja queda parcialmente neutralizada por la carga computacional del sistema Android, los mecanismos de seguridad del kernel y la asignación de recursos con otros procesos activos, lo que en la práctica afecta negativamente el rendimiento de la compresión.

6.5.9. Discusión de resultados

Los resultados obtenidos evidencian que una arquitectura especializada en compresión, ejecutada sobre FPGA, es energéticamente más eficiente y rápida en términos de procesamiento de datos que soluciones basadas en procesadores móviles, todo ello se recaba en la tabla 6.11.

Mientras que un dispositivo móvil como el Samsung S24 Ultra incorpora tecnologías de almacenamiento avanzado y capacidades computacionales superiores en general, su configuración introduce demoras cuando se enfrenta a tareas repetitivas, como lo es la compresión. Esto reafirma la viabilidad de las soluciones hardware dedicadas para aplicaciones donde el rendimiento y la eficiencia energética son requisitos críticos.

Capítulo 7

Conclusión

A partir del desarrollo, implementación y evaluación de la arquitectura especializada para compresión de texto sin pérdida basada en el algoritmo LZ77, se puede afirmar que los objetivos definidos al inicio del proyecto han sido alcanzados. El análisis de algoritmos de compresión permitió seleccionar LZ77 como base de la propuesta, dada su naturaleza deslizante, estructurada y aplicable a flujos de datos secuenciales, lo que facilitó su adaptación a un entorno paralelizable como el de las matrices sistólicas. La arquitectura diseñada integró adecuadamente bloques de control, almacenamiento y procesamiento, organizados de forma escalable mediante elementos de procesamiento configurables, capaces de comparar cadenas de texto en paralelo, cumpliendo así con el diseño de una matriz sistólica funcional y parametrizable.

Durante la validación, se comprobó la tasa de compresión, el tiempo de procesamiento, y el consumo energético, tanto en simulación como en pruebas físicas. La arquitectura alcanzó una tasa de procesamiento de hasta 100 MB/s, procesando 10 MB en menos de 100 ms, superando en eficiencia y velocidad al dispositivo móvil de gama alta contra el que se realizó la comparación. Además, se logró cumplir con la generación de referencias tipo distancia-longitud, basado en el estándar LZ77, demostrando la efectividad del módulo de coincidencia y codificación implementado en hardware. La funcionalidad de simulación produjo reportes detallados sobre el uso de LUTs, FFs, BRAMs y DSPs, así como de consumo dinámico y estático, permitiendo validar el cumplimiento de los requerimientos de eficiencia.

También se verifico que el sistema operó con flujos de entrada a alta velocidad, procesó los datos en bloques, implementó buffers internos, integró la matriz sistólica, y generó referencias de compresión de forma correcta. El sistema fue capaz de trabajar sobre datos codificados en ASCII, validarse y obtener métricas de compresión y latencia.

Finalmente, el diseño demostró eficiencia energética, registrando un consumo máximo de 1.21 W en pruebas físicas, valor muy por debajo del umbral establecido de 5 W. La implementación optimizó el uso de recursos del FPGA Artix-7 XC7A200T, permitiendo mantener un margen térmico seguro y estabilidad operativa sin necesidad de sistemas de enfriamiento activo. La arquitectura resultó escalable, ya que el número de elementos de procesamiento y la longitud de la ventana se pueden modificar sin comprometer la frecuencia de operación. Todo ello verificado con una interfaz estándar, bajo la que fue compatible con tarjetas SD, ya que se diseñó libre de cualquier limitante impuesta por

algún fabricante y es relativamente sencillo conectar el flujo de datos a cualquier estándar conocido, ya que tiene bloques con entrada y salida bien definidos; considerar también que se puede implementar en diferentes frecuencias de funcionamiento, logrando compatibilidad para futuras expansiones.

Asimismo, aunque el sistema genera referencias comprimidas basado en LZ77, no se implementó una etapa de empaquetamiento final en bitstream, considerando que el foco principal fue la compresión funcional y no la transmisión inmediata del resultado, ya que esto limitaría la arquitectura respecto a compatibilidad con diversos estándares o implementaciones que se pueden construir basándose en ella.

Es importante señalar que una de las principales limitaciones al momento de validar cuantitativamente el desempeño de la arquitectura propuesta frente a dispositivos móviles de uso general —como teléfonos inteligentes con sistema operativo Android— radica en las restricciones impuestas por dicho entorno para acceder a métricas de bajo nivel. En dispositivos que no cuentan con acceso root, el sistema operativo restringe la monitorización directa de subprocesos clave como el consumo específico del procesador, el uso de periféricos o el comportamiento térmico detallado, lo que imposibilita la obtención de datos precisos a la hora de ejecutar el algoritmo de compresión.

Como consecuencia, las mediciones realizadas en estos dispositivos representan un consumo global que incluye no solo la ejecución del algoritmo de compresión, sino también la operación simultánea de múltiples módulos como la pantalla, la gestión de red (Wi-Fi, datos móviles), procesos en segundo plano y servicios del sistema, los cuales no pueden ser aislados sin modificaciones profundas al entorno operativo. Esta condición introduce un sesgo en la comparación, ya que los resultados obtenidos en la FPGA —que corresponden exclusivamente al bloque funcional diseñado— no pueden contrastarse con métricas puras equivalentes del entorno Android.

Por tanto, aunque los resultados indican una ventaja significativa de la arquitectura en términos de velocidad, eficiencia energética y especialización, se reconoce que las limitaciones de instrumentación en plataformas comerciales sin acceso completo al sistema impiden una comparación absolutamente equitativa. Esta restricción representa un área de oportunidad para trabajos futuros, que podrían considerar el uso de entornos controlados, emuladores o dispositivos con acceso de administrador completo para lograr una evaluación más precisa y exhaustiva.

En resumen, se ha demostrado que una arquitectura basada en matrices sistólicas puede comprimir texto sin pérdida de forma eficiente, rápida y con bajo consumo energético, cumpliendo tanto con los objetivos del proyecto como con los requerimientos técnicos planteados, y quedando abierta a mejoras e integraciones a futuro.

7.1. Respuesta a la pregunta de investigación

A lo largo del desarrollo de esta investigación se abordó la pregunta: *¿Cómo se puede mejorar la utilización de hardware dedicado para comprimir archivos de texto sin pérdida?*. La solución propuesta se centró en el diseño de una arquitectura especializada basada en el algoritmo LZ77, implementada mediante una matriz sistólica que permite explotar el paralelismo en la tarea de comparación de cadenas. Esta aproximación representó una

mejora sustancial frente a soluciones tradicionales en software y hardware, que suelen procesar los datos de manera secuencial o con bajo grado de paralelismo.

Se mejoró la utilización del hardware al segmentar las etapas del algoritmo de compresión en bloques dedicados, conectados a través de una estructura que permitió flujo continuo de datos, minimizando tiempos muertos y optimizando la utilización de recursos. En particular, cada elemento de procesamiento de la matriz sistólica fue diseñado para ejecutar operaciones de coincidencia en paralelo sobre diferentes posiciones de la ventana deslizante, lo que redujo el tiempo total de compresión. Esta configuración permitió que el sistema alcanzara tasas de compresión cercanas al límite teórico impuesto por el ancho del bus y la frecuencia de operación, maximizando el aprovechamiento del FPGA.

Finalmente, la validación confirmó que el diseño propuesto reduce el consumo energético respecto a implementaciones en dispositivos de uso general, manteniendo una eficiencia constante en el tiempo de compresión. En conjunto, estos elementos demostraron que es posible mejorar sustancialmente la utilización del hardware dedicado para compresión sin pérdida, mediante un diseño especializado, escalable y eficiente.

7.2. Trabajo a Futuro

A partir de los resultados obtenidos durante el desarrollo de esta investigación, se consideran múltiples líneas de trabajo orientadas a que la arquitectura de compresión se consolide como un procesador dedicado plenamente funcional. Uno de los primeros aspectos a abordar consiste en una optimización del diseño, particularmente en los subsistemas de acceso a memoria, considerando la localización espacial de los bloques de memoria, para buscar reducir los retardos de acceso y minimizar el uso de recursos lógicos mediante estructuras de RAM distribuida y bloques BRAM con controladores más eficientes. Este ajuste impactaría en la eficiencia energética y la latencia del sistema.

Posteriormente, se contempla la transición del diseño desde una plataforma de desarrollo FPGA hacia una implementación en hardware dedicado, considerando todas las fases necesarias para la creación de un procesador de aplicación específica. Esta migración implica inicialmente la consolidación del diseño RTL (Register Transfer Level), asegurando su modularidad, escalabilidad y cumplimiento de estándares de verificación como UVM (Universal Verification Methodology). Superada esta etapa, el flujo de síntesis hacia ASIC requiere procesos adicionales, como la conversión del diseño a un flujo compatible con herramientas de síntesis lógica para silicio, la inserción de elementos de testeo estructurado (scan chains, BIST), y la planificación física preliminar del layout del chip. Asimismo, será necesario desarrollar una capa de abstracción software (API y drivers) que permita utilizar el núcleo de compresión desde sistemas operativos convencionales, como Linux o Android, facilitando su adopción en arquitecturas heterogéneas.

Finalmente, y una vez completadas las etapas anteriores, se podrá avanzar a la etapa de tape-out para la fabricación del ASIC. Esta fase incluye validación en silicio (first silicon), pruebas funcionales post-fabricación, validación del encapsulado, y diseño del PCB para entornos de evaluación y pruebas de campo. A partir de ahí, el diseño puede ser comercializado como IP embebible o integrado como componente dedicado en sistemas personalizados, ya sea en almacenamiento masivo, dispositivos móviles, plataformas IoT o

centros de datos. La arquitectura propuesta, ofrece así un camino hacia una solución de alto rendimiento y bajo consumo, completamente escalable y adaptable a las necesidades futuras de procesamiento de datos.

Apéndice A

Anexo

A.1. La Desigualdad de Kraft-McMillan

Esta desigualdad es fundamental para asegurar que se está trabajando con un código prefijo, destacándose la codificación de Huffman como un ejemplo ampliamente utilizado de esta clase de códigos.

La desigualdad de Kraft-McMillan es crucial para validar la no ambigüedad de códigos de longitud variable. Específicamente, establece que, para un código no ambiguo de longitud variable compuesto por n códigos con longitudes L_i, se cumple la siguiente condición:

$$\sum_{i=1}^{n} 2^{-L_i} \leq 1 \tag{15}$$

Propiedad de la desigualdad La segunda parte de la afirmación establece lo siguiente: dado un conjunto de n enteros positivos $(L_1, L_2, \ldots, L_n)$ que satisfacen la inecuación anterior, existe un código sin ambigüedad de longitud variable tal que L_i refiere a la longitud de cada código individual que lo conforme. En conjunto, ambas partes indican que un código es no ambiguo si y solo si satisface esta relación.

Relación con la entropía La desigualdad de Kraft-McMillan puede relacionarse con el concepto de entropía al notar que las longitudes L_i pueden expresarse como:

$$L_i = -\log_2 P_i + E_i,$$

donde E_i representa la diferencia en la que L_i excede la entropía, es decir, la longitud adicional del código i. Sustituyendo en la desigualdad, se obtiene:

$$2^{-L_i} = 2^{-(\log_2 P_i + E_i)} = \frac{2^{-\log_2 P_i}}{2^{E_i}} = \frac{P_i}{2^{E_i}} \tag{16}$$

Cuando todas las longitudes adicionales E_i son iguales $(E_i = E)$, la desigualdad de Kraft-McMillan puede expresarse como:

$$1 \geq \sum_{i=1}^{n} \frac{P_i}{2^E} = \frac{\sum_{i=1}^{n} P_i}{2^E} = \frac{1}{2^E} \tag{17}$$

De lo anterior, se deduce la relación:

$$2^E \geq 1 \implies E \geq 0 \tag{18}$$

Siendo que, un código sin ambigüedad tiene una longitud adicional E_i no negativa. En otras palabras, la longitud del código debe ser mayor o, al menos, igual a la longitud determinada por su entropía.

A.2. Código fuente

Se presenta de forma breve las partes relevantes del código escrito en Verilog respecto al diseño de la arquitectura de compresión.

Código parcial para las instancias de módulos de búsqueda.

```
search search_u0(
    .clk (sys_clk), //Reloj
    .rst_n (rst_n), //Reset activo en bajo
    .look_ahead_buffer_w (look_ahead_buffer_w), //Bus del buffer de
      ↪ anticipacion
    .search_buffer_w (search_buffer_w[8*512-1:0]), //Mitad inferior
      ↪ del bus del buffer de busqueda
    .match_len (match_len1), //Longitud de coincidencia 1
    .SB_index (SB_index1) //Indice de buffer de busqueda 1
);

search search_u1(
    .clk (sys_clk), //Reloj
    .rst_n (rst_n), //Reset activo en bajo
    .look_ahead_buffer_w (look_ahead_buffer_w), //Bus del buffer de
      ↪ anticipacion
    .search_buffer_w (search_buffer_w[8*1024-1:8*512]),  //Mitad
      ↪ superior del bus del buffer de busqueda
    .match_len (match_len2),  //Longitud de coincidencia 2
    .SB_index (SB_index2) //Indice de buffer de busqueda 2
);
```

Código parcial para el estado de búsqueda: encontrar la coincidencia más larga.

```
if(count_search) begin
    // Seleccionar la longitud de coincidencia mas larga entre dos
      ↪ buffers de busqueda de 512 bytes
    {match_len, SB_index} <= (match_len1 >= match_len2) ? {
      ↪ match_len1, SB_index1} : {match_len2, (SB_index2 + 12'
      ↪ d512)};
    ready_encode <= 1; // Preparacion a 1
    finish_out   <= 0; // Finalizacion de salida a 0
```

```
6       finish_shift <= 0; // Finalizacion de desplazamiento a 0
7   end
```

Código parcial para generar los datos de salida codificados.

```
1       //Generar datos de salida para coincidencia de longitud 0
2       if(count_out == 0) begin
3           o_data <= {3'b0, match_len}; //Primer byte de salida:
                ↪ longitud de coincidencia
4           count_out <= 1; //Incrementar contador de salida
5       end
6       else begin
7           o_data <= look_ahead_buffer[15]; //Segundo byte de salida:
                ↪ datos
8           count_out <= 0; //Reiniciar contador de salida
9           finish_out <= 1; //Finalizar salida
10          encoded_length <= encoded_length + 1; //Incrementar
                ↪ longitud codificada
11      end
12  end
13  else begin
14      // Generar datos de salida para coincidencia de longitud no
            ↪ cero
15      if(count_out == 0) begin
16          o_data <= {SB_index[2:0], match_len}; //Primer byte de
                ↪ salida:  ndice  y longitud
17          count_out <= 1; //Incrementar contador de salida
18      end
19      else begin
20          o_data <= SB_index[10:3]; //Segundo byte de salida:  ndice
21          count_out <= 0; //Reiniciar contador de salida
22          finish_out <= 1; //Finalizar salida
23          encoded_length <= encoded_length + match_len; //Incrementar
                ↪  longitud codificada
24      end
25  end
```

Código parcial para las instancias de módulos de búsqueda.

```
1   generate
2       // Instancia el m dulo find_equal para cada segmento del
            ↪ buffer de b squeda
3       for(i = 0; i < 16; i = i + 1) begin
4           find_equal find_equal_i(
5               .look_ahead_buffer(look_ahead_buffer[15]), //Buffer de
                    ↪ adelantamiento actual
6               .search_buffer_w(search_buffer_w[256 * (i + 1) - 1 :
                    ↪ 256 * i]), //Segmento del buffer de b squeda
```

```
            .equal(equal[i]), //Salida de igualdad
            .match_fail(match_fail[i]) //Salida de fallo de
               ↪ coincidencia
        );
    end
endgenerate
```

Código parcial para el módulo de comparación entre caracteres de entrada y de porción del diccionario dinámico.

```
module find_equal (
    input   [7:0]       look_ahead_buffer, //Entrada de 8 bits
       ↪ llamada look_ahead_buffer
    input   [8*32-1:0]  search_buffer_w, //Entrada de 256 bits (8
       ↪ bits * 32) llamada search_buffer_w
    output  [4:0]       equal, //Salida de 5 bits llamada equal
    output              match_fail //Salida de 1 bit llamada
       ↪ match_fail
);
    wire    [7:0]  search_buffer [31: 0]; //Declara un array de 32
       ↪ elementos de 8 bits cada uno llamado search_buffer
    genvar j; //Declara una variable generadora llamada j
    generate //Inicia un bloque generate para generaci n de
       ↪ hardware
        for(j=0 ; j<32 ; j=j+1)begin //Bucle for que va de 0 a 31
            assign  search_buffer[j] = search_buffer_w[8*(j+1)-1 :
               ↪ 8*j]; //Asigna porciones de 8 bits de
               ↪ search_buffer_w a search_buffer
        end
    endgenerate
```

Código referente a la longitud de la coincidencia y el desplazamiento para el decodificador.

```
case (cur_state)
    IDLE : begin
        match_len    <= i_data_decode[4:0]; //Longitud de
           ↪ coincidencia de los datos de entrada
        offset[2:0]  <= i_data_decode[7:5]; //Parte baja del
           ↪ desplazamiento de los datos de entrada
    end

    DECODE1 : begin
        buffer[wptr] <= i_data_decode; //Datos de entrada en el
           ↪ buffer en la posici n del puntero de escritura
        wptr         <= wptr + 12'd1; // Incrementa el puntero de
           ↪ escritura
        decoded_length<= decoded_length + 1; // Incrementa la
           ↪ longitud decodificada
```

```
11      end
12
13      DECODE2_1 : begin
14          offset[10:3] <= i_data_decode; //Parte alta del
               ↪ desplazamiento de los datos de entrada
15      end
16
17      DECODE2_2 : begin
18          buffer[wptr] <= buffer[wptr-offset]; //Valor encontrado en
               ↪ la posicion calculada por el desplazamiento
19          wptr         <= wptr + 12'd1; //Incrementa el puntero de
               ↪ escritura
20          match_len    <= match_len - 1; //Decrementa la longitud de
               ↪ coincidencia
21          decoded_length<= decoded_length + 1; //Incrementa la
               ↪ longitud decodificada
22      end
23
24      FINISH : begin
25          wptr <= 0; //Reinicia el puntero de escritura al finalizar
26      end
27  endcase
```

A.3. Esquemáticos de diseño de arquitectura

Se presenta los diagramas obtenidos al compilar el diseño desarrollado en el entorno de vivado, se demuestra la correcta integración de la arquitectura propuesta y la arquitectura para el manejo de la tarjeta SD, así como la misma a detalle, en la figura A.2.

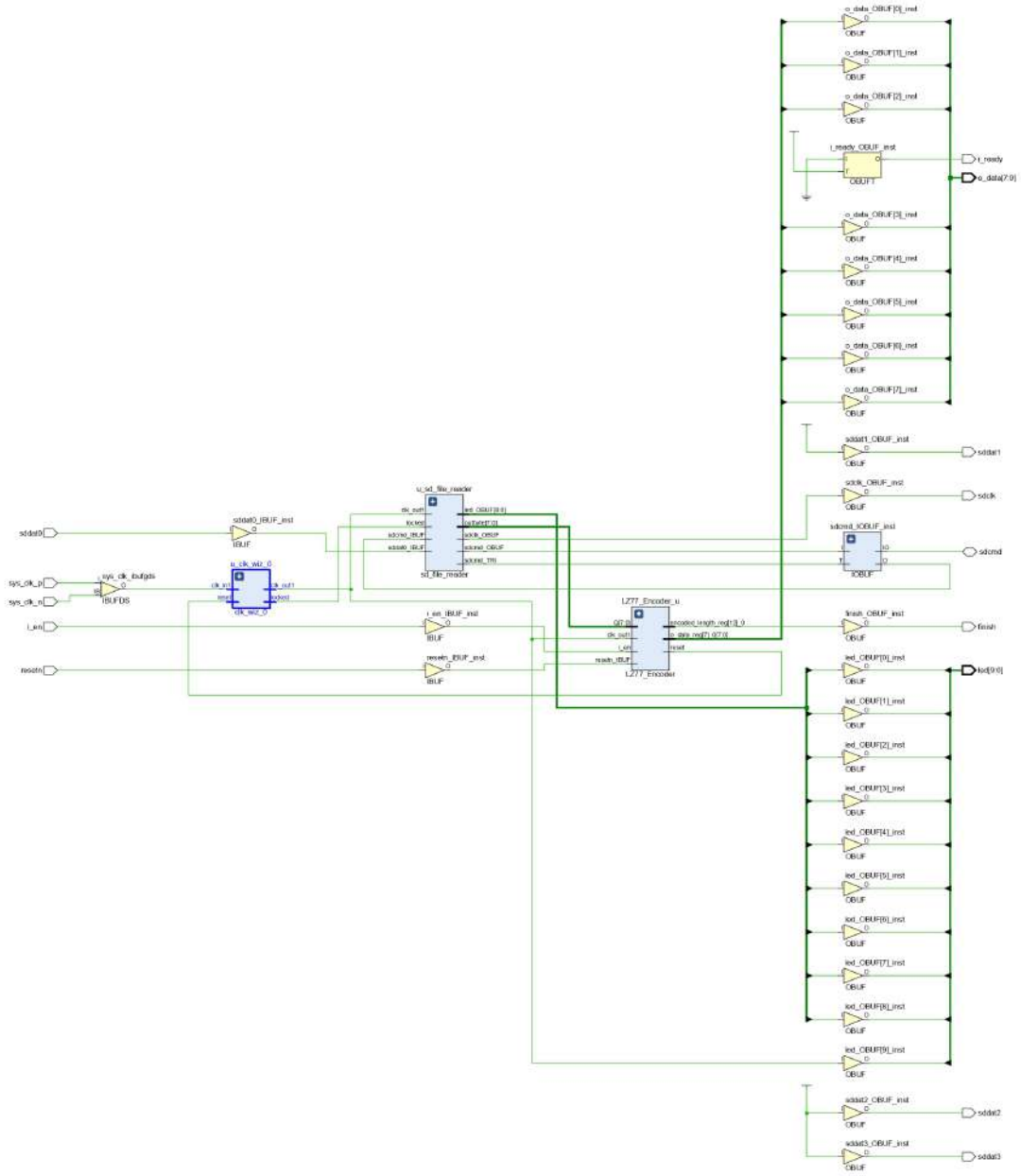


Figura A.1: Esquemático de diseño de arquitectura propuesta con entrada desde tarjeta SD, elaboración propia.

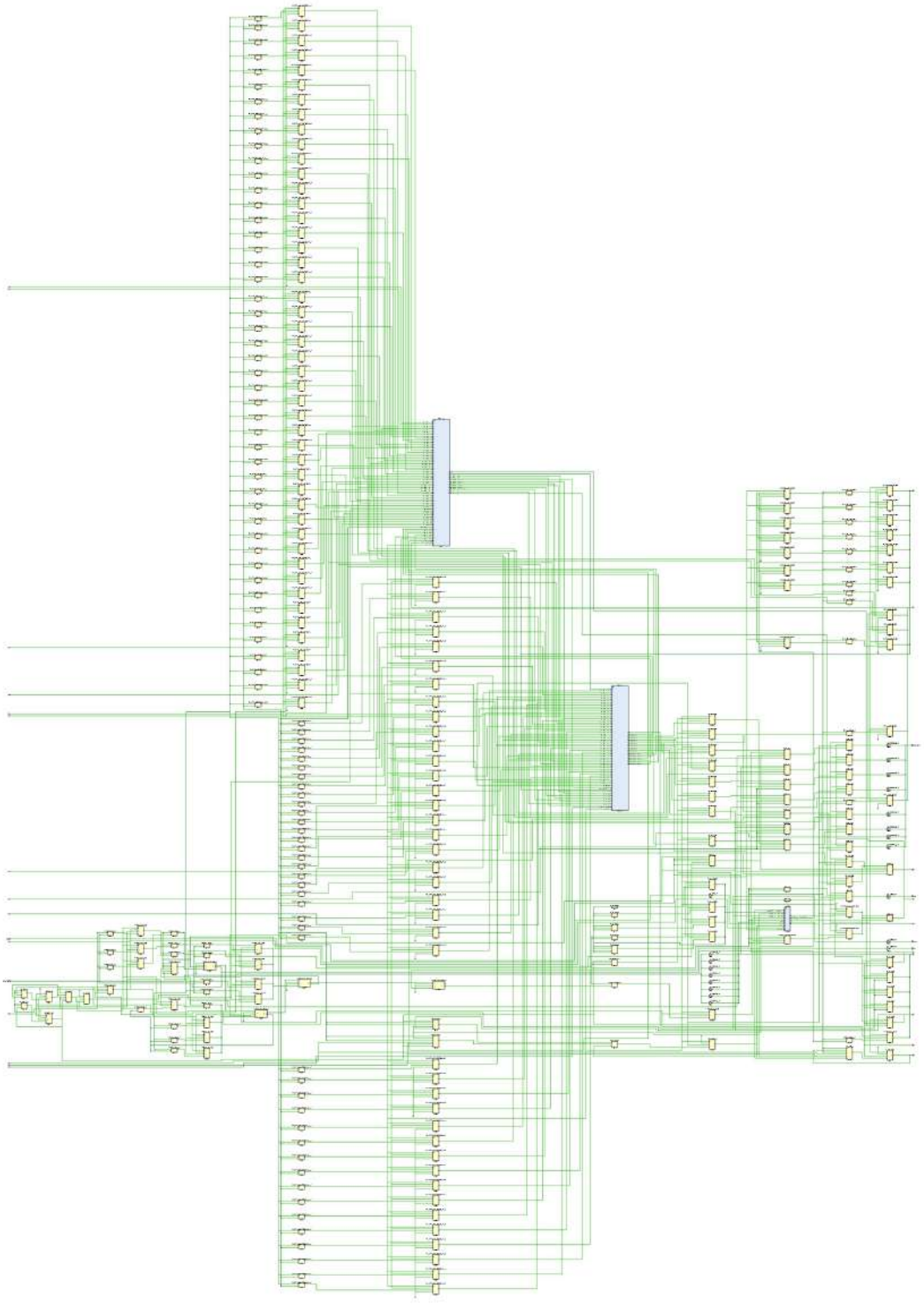


Figura A.2: Esquemático de diseño de arquitectura propuesta, elaboración propia.

Bibliografía

[1] H. T. Kung and C. E. Leiserson, "Systolic arrays for vlsi," *Proceedings of the 1978 Conference on Advanced Research in VLSI*, pp. 105–116, 1978.

[2] AlinX, "Artix-7 FPGA Development Board AX7A200." https://www.alinx.com/public/upload/file/AX7A200_User_Manual.pdfa, 2019. [Accessed 08-11-2023].

[3] "SD and Micro SD card pins with description and functions — electroniccircuitsdesign.com." https://www.electroniccircuitsdesign.com/pinout/sd-microsd-card-pinout.html. [Accessed 29-05-2025].

[4] "Simplified Specifications - SD Association — sdcard.org." https://www.sdcard.org/downloads/pls/. [Accessed 28-05-2025].

[5] M. Satyanarayanan, N. Beckmann, G. A. Lewis, and B. Lucia, "The role of edge offload for hardware-accelerated mobile devices," HotMobile '21, (New York, NY, USA), p. 22–29, Association for Computing Machinery, 2021.

[6] Z. S. Jyrki Alakuijala, Evgenii Kliuchnikov and I. Lode Vandevenne, Google, "Comparison of Brotli, Deflate, Zopfli, LZMA, LZHAM and Bzip2 Compression Algorithms." http://ftp2.de.freebsd.org/pub/misc/cran/web/packages/brotli/vignettes/brotli-2015-09-22.pdf, 2015. [Accessed 30-05-2024].

[7] Qualcomm Technologies, Inc., "Trepn profiler - performance and power profiling tool for android." https://developer.qualcomm.com/software/trepn-profiler, 2022. Versión utilizada: 7.0.8940, compatible con Android 14.

[8] SD Association, "SD Specifications Part 1 Physical Layer Simplified Specification Version 6.00." https://www.sdcard.org/downloads/pls/, 2017. [Accessed 05-05-2025].

[9] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[10] J. Rissanen and G. G. Langdon, "Arithmetic coding," *IBM Journal of Research and Development*, vol. 23, no. 2, pp. 149–162, 1979.

[11] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[12] S. S.A, A. Swedha, and D. Naveen, "Survey of content addressable memory," *IJCRT*, vol. 06, p. 1516, 02 2018.

[13] Kung, "Why systolic architectures?," *Computer*, vol. 15, no. 1, pp. 37–46, 1982.

[14] W. Katz, Phillip W. (Glendale, "String searcher, and compressor using same," September 1991.

[15] H. Hassani and S. MacFeely, "Driving excellence in official statistics: Unleashing the potential of comprehensive digital data governance," *Big Data and Cognitive Computing*, vol. 7, no. 3, 2023.

[16] CloudScene, "South America | Data Center Market Overview | Cloudscene — cloudscene.com." https://cloudscene.com/region/datacenters-in-south-america, 2024. [Accessed 18-08-2024].

[17] E. Topics, "Amount of Data Created Daily (2024) — explodingtopics.com." https://explodingtopics.com/blog/data-generated-per-day, 2024. [Accessed 20-08-2024].

[18] INEGI, "Encuesta nacional (endutih) 2023." https://www.inegi.org.mx/contenidos/saladeprensa/boletines/2024/ENDUTIH/ENDUTIH_23.pdf, 2024. [Accessed 19-09-2024].

[19] "Encuesta Nacional sobre Disponibilidad y Uso de Tecnologías de la Información en los Hogares (ENDUTIH) 2023. (Comunicado de prensa) 13 de junio | Instituto Federal de Telecomunicaciones - IFT." https://www.ift.org.mx/comunicacion-y-medios/comunicados-ift/es/encuesta-nacional-sobre-disponibilidad-y-uso-de-tecnologias-de-la-informacion-en-los-hogares-endutih-2024. [Accessed 13-07-2024].

[20] Branch, "Estadísticas de la situación digital de México en el 2024 — branch.com.co." https://branch.com.co/marketing-digital/estadisticas-de-la-situacion-digital-de-mexico-en-el-2024, 2024. [Accessed 21-09-2024].

[21] Oberlo, "Average Internet Speed by Country and Territory (2024) — oberlo.com." https://www.oberlo.com/statistics/average-internet-speed-by-country, 2024. [Accessed 21-09-2024].

[22] L. P. Cox and L. Ao, "Levelup: A thin-cloud approach to game livestreaming," in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 246–256, 2020.

[23] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pp. 1–12, 2016.

[24] R. Lee, S. I. Venieris, L. Dudziak, S. Bhattacharya, and N. D. Lane, "Mobisr: Efficient on-device super-resolution through heterogeneous mobile processors," in *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[25] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2016.

[26] D. Salomon and G. Motta, *Handbook of Data Compression*. Springer Publishing Company, Incorporated, 5th ed., 2009.

[27] K. Vaid, "Improved cloud service performance through ASIC acceleration | Microsoft Azure Blog — azure.microsoft.com." https://azure.microsoft.com/en-us/blog/improved-cloud-service-performance-through-asic-acceleration/, 2019. [Accessed 22-12-2024].

[28] F. Arias-Odón, *Investigación teórica, investigación empírica e investigación generativa para la construcción de teoría: Precisiones conceptuales 1*. ResearchGate, 09 2019.

[29] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, "Agile software development methods: Review and analysis," *Proc. Espoo 2002*, pp. 3–107, 01 2002.

[30] M. Yadav, N. Goyal, and J. Yadav, "Agile methodology over iterative approach of software development -a review," in *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 542–547, 2015.

[31] S. Goyal, "Agile Techniques for Project Management and Software Engineering." http://csis.pace.edu/~marchese/CS616/Agile/FDD/fdd.pdf, 2007. [Accessed 06-03-2024].

[32] A. F. Chowdhury and M. N. Huda, "Comparison between adaptive software development and feature driven development," in *Proceedings of 2011 International Conference on Computer Science and Network Technology*, vol. 1, pp. 363–367, 2011.

[33] "Search Form — pascal.computer.org." https://pascal.computer.org/sev_display/search.action;jsessionid=5Rz1tW1h9aRvjYnS5yIXYohC5HTs_kD-HAszrfV9.cslcpav04. [Accessed 23-03-2025].

[34] Google, "GitHub - google/snappy: A fast compressor/decompressor — github.com." https://github.com/google/snappy, 2011. [Accessed 18-05-2024].

[35] Google, "GitHub - google/gipfeli — github.com." https://github.com/google/gipfeli, 2014. [Accessed 23-05-2024].

[36] Google, "GitHub - google/zopfli: Zopfli Compression Algorithm is a compression library programmed in C to perform very good, but slow, deflate or zlib compression. — github.com." https://github.com/google/zopfli, 2012. [Accessed 28-05-2024].

[37] Google, "GitHub - google/brotli: Brotli compression format — github.com." https://github.com/google/brotli, 2015. [Accessed 29-05-2024].

[38] M. Powell, "The Canterbury Corpus — corpus.canterbury.ac.nz." https://corpus.canterbury.ac.nz/, 2000. [Accessed 30-05-2024].

[39] S. Xie, X. He, S. He, and Z. Zhu, "Curc: a cuda-based reference-free read compressor," *Bioinformatics*, vol. 38, pp. 3294–3296, 05 2022.

[40] S. Choi, Y. Kim, D. Lee, S. Lee, K. Park, Y. H. Song, and Y. H. Song, "Design of fpga-based lz77 compressor with runtime configurable compression ratio and throughput," *IEEE Access*, vol. 7, pp. 149583–149594, 2019.

[41] O. Plugariu, A. D. Gegiu, and L. Petrica, "Fpga systolic array gzip compressor," in *2017 9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pp. 1–6, 2017.

[42] G. Forman and J. Zahorjan, "The challenges of mobile computing," *Computer*, vol. 27, no. 4, pp. 38–47, 1994.

[43] M. K. J. Mahendra Pratap Singh, "Evolution of processor architecture in mobile phones," *International Journal of Computer Applications*, vol. 90, pp. 34–39, March 2014.

[44] X. H. Xu, C. T. Clarke, and S. R. Jones, "High performance code compression architecture for the embedded arm/thumb processor," in *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, (New York, NY, USA), p. 451–456, Association for Computing Machinery, 2004.

[45] P. Sun and J. Nunez-Yanez, "Optimizing memory power in hybrid arm-fpga chips with lossless data compression," in *Proceedings of the FPGA World Conference 2014*, FPGAWorld '14, (New York, NY, USA), Association for Computing Machinery, 2014.

[46] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin, "Flywheel: Google's data compression proxy for the mobile web," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, (USA), p. 367–380, USENIX Association, 2015.

[47] A. Prakash, S. Wang, A. E. Irimiea, and T. Mitra, "Energy-efficient execution of data-parallel applications on heterogeneous mobile platforms," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pp. 208–215, 2015.

[48] J. Acharya and S. Gaur, "Edge compression of gps data for mobile iot," in *2017 IEEE Fog World Congress (FWC)*, pp. 1–6, 2017.

[49] D. Salomon, *Data Compression: The Complete Reference*. Springer, 2004.

[50] K. Sayood, *Introduction to Data Compression*. Morgan Kaufmann, 2017.

[51] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[52] G. K. Wallace, "The jpeg still picture compression standard," *IEEE Transactions on Consumer Electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1991.

[53] M. Nelson and J.-L. Gailly, *The Data Compression Book*. M&T Books, 1995.

[54] G. Sandhu, "Introduction to data compression: Current methods and future trends," *Internal Document*, 2021.

[55] R. M. Gray, *Entropy and Information Theory*. Springer Publishing Company, Incorporated, 2nd ed., 2011.

[56] R. M. Fano, *The transmission of information*, vol. 65. Massachusetts Institute of Technology, Research Laboratory of Electronics ..., 1949.

[57] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[58] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[59] T. A. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 6, pp. 8–19, 1984.

[60] J. A. Storer and M. Cohn, "Method and apparatus for data compression using adaptive coding," April 2010.

[61] P. Deutsch, "Deflate compressed data format specification version 1.3," *RFC*, no. 1951, 1996.

[62] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[63] P. Deutsch, "Deflate compressed data format specification version 1.3," Tech. Rep. RFC 1951, RFC Editor, May 1996.

[64] W. B. Pennebaker and J. L. Mitchell, *JPEG Still Image Data Compression Standard*. Springer, 1992.

[65] A. S. Inc., *PDF Reference, Sixth Edition: Adobe Portable Document Format*, November 2006.

[66] J. Li, T. Chen, and Y. Zhang, "Efficient implementation of jbig2 in document processing," *IEEE Transactions on Image Processing*, vol. 15, pp. 1992–2002, July 2006.

[67] "Document management — portable document format — part 2: Pdf 2.0," 2020.

[68] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[69] S. Mallat, *A Wavelet Tour of Signal Processing: The Sparse Way*. Academic Press, 3rd ed., 2009.

[70] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2012.

[71] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 948–960, 1972.

[72] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[73] J. Doe and J. Smith, "Optimizing memory usage with lossless compression in embedded systems," *Journal of Embedded Systems*, vol. 45, no. 2, pp. 120–130, 2021.

[74] M. Abd El Ghany, M. El-Moursy, and A. Salama, *Design and Implementation of FPGA-based Systolic Array for LZ Data Compression*. 04 2010.

[75] AMD, "AMD Extends Product Lifecycle for All Xilinx 7 Series Devices Through at Least 2035 — community.amd.com." https://community.amd.com/t5/adaptive-computing/amd-extends-product-lifecycle-for-all-xilinx-7-series-devices/ba-p/563507. [Accessed 20-10-2024].

[76] "ALINX AX7A200: with AMD Artix FPGA Development Kit Board — xilinx.com." https://www.xilinx.com/products/boards-and-kits/1-1bqbcoe.html. [Accessed 20-10-2024].

[77] J. J. Montes Salinero, *Simulación y medida de consumo en FPGAs para arquitecturas de operadores aritméticos*, 2023. Escuela Técnica Superior de Ingenieros Industriales.

[78] "IEEE Standards Association — standards-ieee-org.translate.goog." https://standards-ieee-org.translate.goog/ieee/830/1222/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc. [Accessed 18-06-2025].

[79] h. Injosoft AB, "ASCII table - Table of ASCII codes, characters and symbols — ascii-code.com." https://www.ascii-code.com/. [Accessed 09-01-2025].

[80] R. Arnold and T. Bell, "A corpus for the evaluation of lossless compression algorithms," in *Proceedings DCC '97. Data Compression Conference*, pp. 201–210, 1997.

[81] "Calgary Corpus — data-compression.info." https://www.data-compression.info/Corpora/CalgaryCorpus/. [Accessed 04-05-2025].

[82] Pawel Boniecki and Piotr Grabowski, "The Silesia Corpus for Compression Algorithm Evaluation." http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia, 2003. Accessed: 2025-05-01.

[83] The Competitive Intelligence Unit (The CIU), "Evolución del mercado de smartphones en México 2024," 2024. Disponible en línea.

[84] "Info-ZIP's Zip — infozip.sourceforge.net." https://infozip.sourceforge.net/Zip.html. [Accessed 03-05-2025].

[85] "Termux — termux.dev." https://termux.dev/en/. [Accessed 03-05-2025].

[86] "GitHub - sharkdp/hyperfine: A command-line benchmarking tool — github.com." https://github.com/sharkdp/hyperfine. [Accessed 03-05-2025].

[87] J. loup Gailly and M. Adler, "Zlib compression library." https://zlib.net, 2023. Versión 1.2.11, utilizada en implementación NDK.

[88] A. Frumusanu, "The samsung galaxy s24 ultra review: The snapdragon 8 gen 3, galaxy ai titanium," *AnandTech*, February 2024. Sección de rendimiento energético y almacenamiento.

[89] AMD, "AMD Technical Information Portal — docs.amd.com." https://docs.amd.com/v/u/en-US/ug472_7Series_Clocking. [Accessed 10-06-2025].

[90] Microsoft Corporation, "Microsoft extensible firmware initiative fat32 file system specification." https://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/fatgen103.pdf, 2000. Especificación técnica oficial del sistema de archivos FAT16/FAT32.

[91] M. Corporation, "Microsoft fat specification." https://academy.cba.mit.edu/classes/networking_communications/SD/FAT.pdf, 2005. [Accessed 14-05-2025].

[92] J. Dobiash, "Fat16 structure information." https://teslabs.com/openplayer/docs/docs/specs/fat16_specs.pdf, 1999. [Accessed 16-04-2025].

[93] MIT Center for Bits and Atoms, "SD cards and the FAT filesystem." https://academy.cba.mit.edu/classes/networking_communications/SD/FAT.pdf, 2019.

[94] wangxuan95, "Dr.W.X/FPGA-SDcard-Reader — gitee.com." https://gitee.com/wangxuan95/FPGA-SDcard-Reader, 2023. [Accessed 25-04-2025].

[95] AMD, *Vivado Design Suite User Guide: Programming and Debugging*. AMD, 2024. UG908 (v2024.2).

[96] AMD, "Modifying properties on the debug cores." https://docs.amd.com/r/en-US/ug908-vivado-programming-debugging/Modifying-Properties-on-the-Debug-Cores, 2024. [Accessed 10-05-2025].