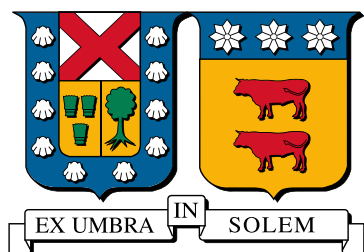


UNIVERSIDAD TÉCNICA FEDERICO SANTA
MARÍA

DEPARTAMENTO DE ELECTRÓNICA

VALPARAÍSO - CHILE



“DISEÑO E IMPLEMENTACIÓN EN FPGA
DE PROCESADOR BASADO EN
AUTÓMATAS CELULARES ENFOCADO EN
PROCESAMIENTO DE IMÁGENES”

RICARDO ESTEBAN MARDONES TABILO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL
ELECTRÓNICO.

PROFESOR GUIA: DR. IOANNIS VOURKAS

PROFESOR CORREFERENTE: DR. GONZALO CARVAJAL

MARZO 2023

Agradecimientos

En primer lugar, quisiera agradecer a mi profesor guía, Ioannis Vourkas, por su orientación experta y dedicada. Su conocimiento, experiencia y paciencia fueron vitales. Agradezco sinceramente su tiempo y compromiso para apoyarme en la elaboración de esta memoria.

A mis padres Clara y Joviel, quienes junto a mis hermanos supieron apoyarme en todo lo que pudieron desde el inicio hasta el fin de este camino, además de nunca perder su fé y orgullo hacia mi a pesar de los tropiezos.

A mis compañeros, no puedo imaginar haber llegado hasta este punto sin su amistad y compañía a lo largo de la carrera. Gracias por los gratos momentos, las conversaciones tanto las banales como profundas y serias. Gracias también por los carritos a los que en ocasiones nos tocó subir durante estos años.

A los cuarentena fitness, fueron la familia en la que me acogí y quienes me ayudaron a sobrellevar los días con alegría y energía durante los últimos años.

A Anelore, por la paciencia que tuvo al otorgarme su compañía desde el inicio de la investigación y al acudir cuando requerí de su ayuda, además de ser un gran apoyo y fuente de motivación hasta el final de este trabajo.

A los muchachos, estoy y estaré infinitamente agradecido por poder mantener su amistad a día de hoy a pesar de la distancia y la diferencia de nuestros caminos. Confío que a pesar de que hoy estamos más dispersos que nunca podremos continuar sumando anécdotas para rememorar, hay un plan.

A cada uno de ustedes, gracias de corazón. Su apoyo, amistad y amor han sido esenciales en mi camino hacia este punto. Sin ustedes, este viaje no hubiera sido tan significativo ni memorable.

"Esas dulces caricias refrescantes devolvieron a la flor su lozanía, y en el fondo del valle solitario se le vio sonreír con alegría.

Por desgracia, no abundan en el mundo las blancas mariposas condolidas, ni las fuentes tranquilas y serenas capaces de curar las heridas.

Oh, vosotros los hombres de fortuna que apuráis el placer hasta el delirio, acordaos de que hay prójimos que sufren en perpetua agonía, como el lirio." - Adriana Valdivia

Diseño e implementación en FPGA de procesador basado en Automatas Celulares enfocado en procesamiento de imágenes

Ricaro Esteban Mardones Tabilo

Memoria para optar al título de Ingeniero Civil Electrónico, mención Computadores,
submención Telecomunicaciones.

Universidad Técnica Federico Santa María

Profesor Guía: Dr. Ioannis Vourkas

Profesor Correferente: Dr. Gonzalo Carvajal

Marzo 2023

Resumen

En este proyecto se diseñó e implementó un procesador basado en *Cellular Automata (CA)* para el procesamiento de imágenes. Se desarrolló una versión de software para la detección de bordes y erosión, que se replicó en hardware utilizando una tarjeta Nexys4DDR. Los resultados obtenidos fueron contrastados con métodos estándar de procesamiento de imágenes y se encontró que la implementación basada en *CA* obtuvo resultados comparables en cuanto a resultados de procesamiento, además de presentar tiempos de ejecución más rápidos, pero con una alta utilización de recursos. Además, se propone un posible uso del sistema en la industria agrícola para la detección de defectos en frutas. Los resultados sugieren que el procesador basado en autómatas celulares podría ser una solución efectiva y para aplicaciones de procesamiento de imágenes en tiempo real en entornos industriales.

Keywords: Automata Celular; FPGA; Procesamiento de imágenes; Detección de bordes; Erosión

Design and implementation in FPGA of a processor based on Cellular Automata focused on image processing

Ricardo Esteban Mardones Tabilo

Thesis for the fulfillment of the B.S. in Electronic Engineering, major in Computer
Electronics, minor in Telecommunications (6 year program).

Universidad Técnica Federico Santa María

Advisor: Dr. Ioannis Vourkas

Co-Advisor: Dr. Gonzalo Carvajal

March 2023

Abstract

In this project, a processor based on Cellular Automata (CA) was designed and implemented for image processing. A software version was developed for edge detection and erosion, which was replicated in hardware using a Nexys4DDR board. The obtained results were compared with standard image processing methods, and it was found that the CA-based implementation achieved comparable processing results with faster execution times, but with high resource utilization. Furthermore, a potential use of the system in the agrifood industry for the detection of defects on fruits is proposed. The results suggest that the CA-based processor could be an effective solution for real-time image processing applications in industrial environments.

Keywords: Cellular automata; FPGA; Image processing; Edge detection; Erosion

Acrónimos

CA Cellular Automata. iii, viii, ix, 1–5, 7–27, 29–31, 37–57, 59–63, 68, 69, 71–73, 76, 77

CGoL Conway’s Game of Life. 13

CLB Configurable Logic Block. 6

FPGA Field-Programmable Gate Array. viii, 1, 2, 5, 6, 14, 15, 21, 31, 32, 34

FSM Finite State Machine. ix, 7, 25–28, 31, 62, 68

HD Distancia de Hamming. 37, 38, 48, 52, 56

HDL Hardware Description Language. 5, 6

LUT Look Up Table. 6, 47

MAE Error Medio Absoluto. 37, 38, 43, 48

RMSE Error Cuadrático Medio. 37, 38, 43, 48

SM Switching Matrix. 6

UART Universal Asynchronous Receiver-Transmitter. ix, 5, 31–33, 62, 77

Índice de contenidos

1. Introducción	1
1.1. Motivación y contexto	2
1.2. Alcances y Contribuciones	2
1.3. Objetivos del trabajo	3
1.4. Metodología	4
1.4.1. Metodología de investigación	4
1.4.2. Metodología de diseño	4
2. Antecedentes	6
2.1. FPGA	6
2.2. Autómata Celular	7
2.2.1. Conceptos básicos	7
2.2.2. Tipos de Autómatas Celulares	10
2.3. Autómatas celulares y procesamiento de imágenes	12
2.4. Estado del arte	13
3. Modelo de Autómata Celular y módulo diseñado para su implementación	15
3.1. Reglas de evolución	15
3.1.1. Detección de bordes	16
3.1.2. Erosión	16
3.2. Diseño de un Autómata Celular en Software	17
3.2.1. Funciones implementadas	17
3.3. Diseño de Autómata Celular en Hardware	21
3.3.1. Módulos de Celda	21
3.3.2. Módulos de control	25
3.3.3. Módulos de comunicación	31
3.3.4. Sistema completo	34
4. Resultados	37
4.1. Resultados de la implementación del CA en software	39

4.2. Resultados del diseño en hardware	46
4.3. Contraste de resultados entre software y hardware	56
4.4. Tiempos de ejecución	60
5. Potencial aplicación del diseño en la industria	63
5.1. Contexto y motivación	63
5.2. Acondicionamiento del sistema	64
5.3. Resultados Experimentales	69
6. Conclusiones	76
7. Trabajo Futuro	77
Referencias	79
8. Anexos	81
8.1. Funciones para implementación del CA en python	81
8.2. Módulo de Celda	84

Índice de tablas

4.1. Métricas de comparación para la erosión de la Imagen <i>Leopardo</i>	41
4.2. Métricas de comparación para la erosión de la Imagen <i>Círculos</i>	43
4.3. Métricas de comparación para la detección de bordes en imagen <i>Leopardo</i>	44
4.4. Métricas de comparación para la detección de bordes en imagen <i>Círculos</i>	46
4.5. Recursos físicos disponibles en la tarjeta de desarrollo Nexys 4 DDR [1].	46
4.6. Utilización de recursos para implementación de erosión en una malla de 125x125 celdas.	48
4.7. Utilización de recursos para erosión respecto al total disponible en la tarjeta. .	48
4.8. Métricas de comparación para la erosión de imagen <i>Leopardo</i>	49
4.9. Métricas de comparación para la erosión de imagen <i>Círculos</i>	51
4.10. Utilización de recursos para implementación de detección de bordes en una malla de 107x107 celdas.	51

4.11. Utilización de recursos para detección de bordes respecto al total disponible en la tarjeta.	52
4.12. Métricas de comparación para la detección de bordes en imagen <i>Círculos</i>	53
4.13. Métricas de comparación para la detección de bordes en imagen <i>Formas</i>	55
4.14. Contraste de resultados de la aplicación de erosión a través de su implementación software vs hardware.	58
4.15. Contraste de resultados de la aplicación de detección de bordes a través de su implementación software vs hardware.	60
4.16. Resultados de tiempos de ejecución de detección de bordes por el <i>CA</i> en software y para el método <i>Canny</i>	61
4.17. Resultados de tiempos de ejecución de erosión por el <i>CA</i> en software y para el metodo <i>Erode()</i>	61
4.18. Resultados de tiempos de ejecución de procesamiento por el <i>CA</i> para cada regla de evolución implementada.	62
4.19. Resumen de latencia para el procesamiento de imágenes con cada método analizado.	63
5.20. Utilización de recursos para el diseño enfocado en la potencial aplicación en la industria.	71
5.21. Utilización de recursos para erosión respecto al total disponible en la tarjeta. .	71

Índice de figuras

2.1. Estructura descriptiva de una <i>Field-Programmable Gate Array (FPGA)</i>	6
2.2. Vecindades más utilizadas en modelos de <i>CA</i> bidimensionales [2].	9
2.3. Ejemplo gráfico de condiciones de frontera más utilizadas en <i>CA</i> unidimensional. .	10
3.4. Operador lógico OR y operador lógico NOR como técnica de erosión.	17
3.5. Diagrama de malla de <i>CA</i> y diagrama del módulo <i>CELDA</i>	22
3.6. Ejemplo gráfico del proceso de carga/descarga de datos a una malla de <i>CA</i> de tamaño 3x3.	23
3.7. Malla genérica de celdas para un <i>CA</i>	24

3.8. Diagrama de alto nivel del módulo <code>FSM_OPERATION_MODES</code> y diagrama de estados de la <i>Finite State Machine (FSM)</i> asociada.	26
3.9. Diagrama de flujo de los casos posibles al ocurrir una transición en los estados <code>LOAD_WAIT</code> y <code>DOWN_WAIT</code>	29
3.10. Simulación de la <i>FSM</i> que controla la operación del <i>CA</i>	31
3.11. Diagrama de alto nivel de dispositivos conectados a través de interfaz <i>Universal Asynchronous Receiver-Transmitter (UART)</i>	32
3.12. Diagrama del módulo de interfaz <i>UART</i> implementado en el <i>device</i>	33
3.13. Diagrama de bloques del sistema diseñado.	36
4.14. Imágenes utilizadas para obtención y análisis de resultados.	39
4.15. Resultados de erosión de la Imagen <i>Leopardo</i> para el diseño en software.	41
4.16. Resultados de erosión de la Imagen <i>Círculos</i> para el diseño en software.	42
4.17. Resultados de detección de bordes en imagen <i>Leopardo</i>	44
4.18. Resultados de detección de bordes en imagen <i>Círculos</i>	45
4.19. <code>width=1</code>	47
4.20. Resultados de erosión de imagen <i>Círculos</i> con el <i>CA</i> implementado en hardware.	49
4.21. Resultados de erosión de imagen <i>Círculos</i>	50
4.22. Resultados de detección de bordes de imagen <i>Círculos</i>	53
4.23. Resultados de detección de bordes en imagen <i>Formas</i>	55
4.24. Resultados de erosión mediante <i>CA</i> en su versión en software y en hardware.	57
4.25. Resultados de detección de bordes mediante <i>CA</i> en su versión en software y en hardware.	59
5.26. Imágenes del contexto de uso de la detección de bordes en la industria agroalimentaria adaptadas de [3].	64
5.27. Histogramas para cada canal R, G y B del dataset de naranjas del grupo CO-FILAB.	65
5.28. Resultados de preprocesamiento en imagen de naranja.	67
5.29. Resultados de preprocesamiento en imagen de naranja.	67
5.30. Diagrama de estados del módulo <code>FSM_OPERATION</code> con erosión y detección de bordes conjunta.	69

5.31. Diagrama de bloques del sistema modificado, agregando una opción más al MUX 1 y extendiendo la cantidad de bits para representar la señal OPERATION.	70
5.32. Resultados del procesamiento de imágenes de frutas.	72
5.33. Promedios de cantidad de pixeles blancos en imagen junto a su desviación estándar.	74

1. Introducción

El procesamiento de imágenes es un campo en constante evolución, en el que continuamente se buscan alternativas para acelerar el procesamiento y mejorar la eficiencia. En este contexto, los Autómatas Celulares, en inglés *Cellular Automata (CA)*, han surgido como una herramienta prometedora debido a sus características de cómputo masivamente paralelo y su filosofía de integrar almacenamiento y procesamiento en una misma unidad. En este trabajo se aborda el diseño de un módulo de procesamiento de imágenes basado en *CA* y su implementación en *Field-Programmable Gate Array (FPGA)*, con el objetivo de explorar una alternativa para el procesamiento de imágenes y validar las ventajas que presentan los *CA* para este tipo de aplicaciones. Para ello, se comparan los resultados obtenidos mediante el uso de métodos tradicionales de procesamiento de imágenes con los resultados obtenidos mediante el procesamiento de imágenes a través de *CA*. Se espera demostrar que la implementación de *CA* en *FPGA* puede ofrecer una mayor eficiencia en términos de tiempo de procesamiento, a pesar del costo en términos de recursos físicos. Además, se discuten las implicaciones de la necesidad de una interfaz de comunicación de alta velocidad para lograr un procesamiento eficiente de imágenes utilizando *CA* implementados en *FPGA*.

El trabajo presentado se organiza en capítulos, en el Capítulo 2 se encuentra un marco teórico de los conceptos necesarios para la comprensión del trabajo, más adelante, en el Capítulo 3 se explica y describe en detalle el sistema propuesto, en qué consiste y cómo se logró implementar cada una de sus funcionalidades. En el Capítulo 4 encuentra un análisis de los resultados obtenidos a través del sistema propuesto. En el Capítulo 5, se menciona una posible aplicación del sistema en la industria, junto a modificaciones que se le aplican al diseño para poder satisfacer las necesidades de esta aplicación. Para finalizar, en el Capítulo 6 se exponen las conclusiones obtenidas gracias al trabajo desarrollado y en el Capítulo 7 se mencionan posibles mejoras que se le podrían aplicar al diseño, para mejorar los resultados entregados. Las referencias en las que se sostiene el proyecto se pueden encontrar en la sección de referencias al final de este manuscrito.

1.1. Motivación y contexto

El uso de modelos basados en autómatas celulares para el desarrollo de tareas de procesamiento de imágenes ha aumentado considerablemente en los últimos años gracias a la aparición de nuevas tecnologías que permiten su implementación en tiempo real. Estos modelos permiten una evolución global del sistema en el tiempo a través de reglas de complejidad arbitrariamente simples, lo que puede llegar incluso a sencillas operaciones discretas, y la obtención de resultados cercanos a los que se obtienen con métodos estándar para el mismo tipo de tareas. Dada la gran cantidad de aplicaciones para los modelos de celular automata, se espera que estos mismos puedan encaminar importantes proyectos en el futuro.

La implementación en hardware de este tipo de modelos puede aumentar la velocidad de ejecución del procesamiento que estos pretenden desarrollar, dadas las características de los circuitos combinatorios en comparación con su contraparte, la lógica secuencial, que es la base del procesamiento por software.

Se espera que el proyecto propuesto en esta memoria dé como resultado una unidad de procesamiento en hardware especializado que pueda acelerar la ejecución de tareas de procesamiento de imágenes con base en computación masivamente paralela. Además, se espera que esta unidad de procesamiento pueda usarse en conjunto con cualquier procesador afín y permita el desarrollo de nuevos trabajos sobre la evolución y adaptación de los autómatas celulares, permitiendo mejoras y nuevas implementaciones en diferentes áreas de la ingeniería.

1.2. Alcances y Contribuciones

Se limitan los alcances del presente trabajo bajo las siguientes condiciones:

- El trabajo corresponde a una prueba de concepto para comprobar la ganancia en términos de velocidad de procesamiento de una implementación en hardware, específicamente en *FPGA*, de un modelo de *CA*, dado que de esta manera, se pueden explotar las características de esta clase de sistemas, principalmente el cómputo masivamente paralelo, que en software no se puede lograr en su totalidad dada la forma secuencial en que se ejecutan las instrucciones de un programa que implementa los mismos modelos.

- El trabajo se realiza utilizando tarjetas de desarrollo enfocadas en la educación y evaluación de sistemas, por lo que el diseño del sistema propuesto corresponde a una prueba a pequeña escala acotada a las capacidades que las mismas herramientas utilizadas dispongan, es decir, recursos físicos y periféricos para la interacción con el sistema a implementar.
- El trabajo tiene un enfoque principalmente práctico, con el fin de poner a prueba los conceptos teóricos en los que se basa el supuesto de la factibilidad de lograr aceleración por hardware de modelos de *CA*, más que ahondar en lo investigativo y la innovación, sin embargo, estos aspectos no se descartan en el desarrollo del mismo.

A partir de los resultados obtenidos se puede reconocer las siguientes contribuciones:

- Una implementación en Lenguaje Python modular de un modelo genérico *CA*, fácilmente modificable para poder evaluar distintas reglas de evolución, tipos de vecindades y condiciones de frontera para malla de distinto tamaño.¹
- Módulos en lenguaje SystemVerilog, para la implementación de un *CA* en hardware, entre los que destaca el módulo de celda genérica para *CA*, disponible en [4]
- Validación del comportamiento de reglas de evolución capaces de efectuar erosión y detección de bordes sobre imágenes binarias mediante *CA*
- Propuesta de aplicación en la industria agrícola para sistemas digitales que contribuyan a la aceleración de procesamiento a través de hardware especializado, junto a un prototipo base funcional a pequeña escala.

1.3. Objetivos del trabajo

El objetivo principal de este trabajo consiste en lograr una implementación en hardware, específicamente en una tarjeta de desarrollo con FPGA, de un procesador con soporte para la ejecución de algoritmos de procesamiento digital de imágenes basado en un modelo de autómatas celulares, buscando explorar y analizar las ventajas que presentan los Autómatas Celulares

¹Los conceptos que se mencionan y que son referidos a un Autómata Celular se detallan en la sección 2.2, disponible en [4]

en este ámbito y explotar sus características para acelerar el proceso mediante la computación masivamente paralela inherente a este tipo de sistemas. Dado este objetivo primario se desprenden los siguientes objetivos específicos:

- Realizar una investigación sobre estado del arte sobre autómatas celulares y su uso en procesamiento de imágenes.
- Obtener reglas de evolución que permitan desarrollar tareas de procesamiento digital de imágenes.
- Implementación de modelo de Autómata Celular en Software capaz de desarrollar tareas de procesamiento de imágenes.
- Implementación en hardware del modelo de Autómata Celular diseñado.
- Implementación de protocolo de comunicación serial para transferencia de imágenes desde PC a FPGA y viceversa.

1.4. Metodología

1.4.1. Metodología de investigación

Los pasos necesarios para la realización de este trabajo se exponen a continuación:

- Estudiar el estado del arte, mediante publicaciones, revistas y tesis especializadas en el área de los *CA*.
- Simular modelos de *CA* conocidos como primer acercamiento a la implementación de estos sistemas.

1.4.2. Metodología de diseño

Los pasos para la implementación que se propone en este trabajo son:

- Implementación en Software de un *CA* usando lenguaje **Python** 3.10.7.
- Simular el modelo utilizando reglas enfocadas al procesamiento de imágenes.

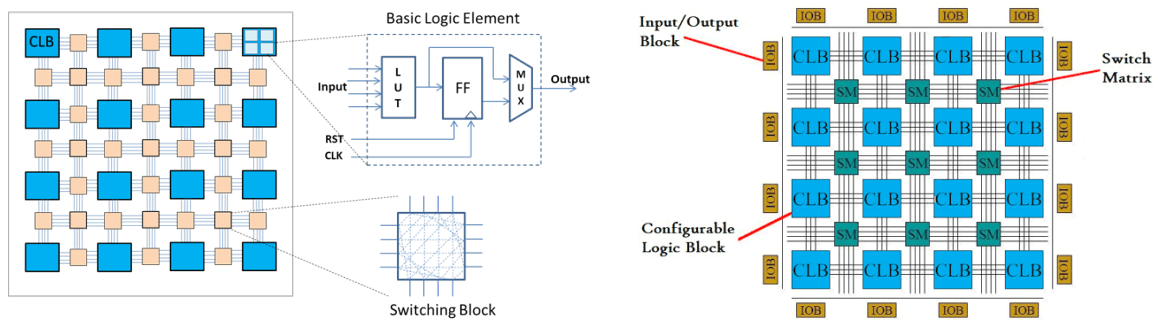
- Implementación en Hardware mediante de un modelo de *CA* mediante *Hardware Description Language (HDL)*.
- Simulación del modelo implementado en Hardware para verificar su correcto funcionamiento usando la herramienta **Vivado 2022.1**.
- Implementación de protocolo de comunicación *Universal Asynchronous Receiver-Transmitter (UART)* para interacción entre PC y la tarjeta de desarrollo con *FPGA*.
- Verificar el correcto funcionamiento de la implementación física del modelo en la *FPGA* para el procesamiento de imágenes.

2. Antecedentes

2.1. FPGA

Las *FPGA* son una clase de circuitos integrados que nacen como una evolución de los dispositivos lógicos programables existentes en la década de los 80, diseñadas por los cofundadores de Xilinx, Ross Freeman y Bernard Von Der Schmitt, surgen como una alternativa para el diseño de dispositivos lógicos, que logran incrementar la velocidad de ejecución, a la vez que se reduce el consumo energético y permiten una mayor densidad de componentes en un tamaño reducido.

Estos dispositivos están compuestos por una matriz de bloques lógicos programables o *Configurable Logic Block (CLB)*, lo que equivale a millones de puertas lógicas discretas encapsuladas en un pequeño chip. Cada *CLB* contiene ciertos elementos básicos: *Look Up Table (LUT)*, para implementar lógica combinatoria; *Flip-Flops*, que se utilizan como registros y multiplexores. Siendo todos estos elementos configurables. Los *CLB* que componen la matriz se interconectan entre sí a través de *Switching Matrix (SM)* o *matriz de conmutación*, también programables por el usuario. Además, estos dispositivos son capaces de interactuar con el exterior mediante bloques de *Entrada/Salida* permitiendo transmitir señales desde o hacia la *FPGA*. Las Figuras 2.1a y 2.1b muestran gráficamente lo descrito.



(a) Estructura de la matriz de bloques de una *FPGA*

(b) Esquema general de una *FPGA*

Figura 2.1: Estructura descriptiva de una *FPGA*.

Estos dispositivos se programan usando lenguajes de descripción *HDL* como VHDL o SystemVerilog a través de software especializado como Vivado, que puede ser ejecutado desde un

computador personal.

2.2. Autómata Celular

Los Automatas Celulares (CA) surgen en la década de 1940, originalmente como un medio para representar sistemas biológicos autoreproductivos, pero que a día de hoy se han utilizado como una forma de modelar sistemas dinámicos complejos en tiempo discreto. Debido a que representan una manera sencilla de estudiar distintos tipos de sistemas complejos, los CA han sido objeto de interés en diversas áreas, si bien, su aplicación se ha desatado principalmente en campos como la matemática y la computación, se ha extendido a otros campos como la física, biología y química [5, 6]. Algunos ejemplos de aplicación de CA han sido modelamiento de comportamiento de fluidos, crecimiento de cristales, propagación de enfermedades infecciosas, propagación de incendios forestales, patrones de actividad eléctrica en neuronales, entre otras.

Los CA consisten en una malla regular N-dimensional de celdas o células, donde cada una de ellas puede encontrarse en uno de un conjunto finito de estados posibles. El estado de cada celda se actualiza en función del tiempo de acuerdo a un conjunto definido de reglas que considera el estado actual de la celda junto al estado de las celdas vecinas. Estas reglas son, esencialmente una *Finite State Machine (FSM)*, generalmente especificada en forma de una tabla de reglas con una entrada para cada posible configuración de los estados de una vecindad de celdas [7]. Las reglas para actualizar el estado de las celdas se aplica simultáneamente a toda la malla y de manera sincrónica, de esta forma el estado de la malla (o del CA) avanza en pasos de tiempo discreto obteniendo una dinámica global en base a interacciones locales. Considerando la forma en la que evolucionan las celdas y como se relacionan entre sí, el paralelismo resulta ser una característica inherente a los Autómatas Celulares y puede ser explotada, ya que contribuye a la aceleración de procesos y la ejecución de algoritmos.

2.2.1. Conceptos básicos

Existe una gran variedad de sistemas basados en CA específicos, diseñados para cumplir con requisitos específicos de su modelo particular, sin embargo, todos deben cumplir con las siguientes características:

Espacio Regular

En un sistema de CA , las celdas o células se distribuyen de manera uniforme y homogénea sobre una rejilla o malla de dimensión finita, en donde estas pueden interactuar entre sí de manera local. Las dimensiones ampliamente más utilizadas son la unidimensional que pueden verse como un arreglo o vector fila, y la bidimensional, que se puede estudiar como una matriz de tamaño $M \times N$, que permite desarrollar tareas de procesamiento o simulación de fenómenos espaciales, como también la ejecución de algoritmos que dan solución a problemas de planificación y sincronización de tareas.

Conjunto de estados

El conjunto de posibles estados es finito y cada elemento o celda del espacio regular toma un valor de este conjunto de estados, asociados a cada punto r de la malla, indicando el estado de cada celda en el instante de tiempo t .

$$C(\vec{r}, t) = \{C_1(\vec{r}, t), C_2(\vec{r}, t), \dots, C_m(\vec{r}, t)\} \quad (2.1)$$

Vecindad

Corresponde al conjunto de celdas o células adyacentes a una celda dada, así como su posición relativa respecto a esta. Puede definirse como:

$$\mathcal{N}(\vec{r}) = \{\vec{r}, \vec{r} + \vec{\delta}_1, \vec{r} + \vec{\delta}_2, \dots, \vec{r} + \vec{\delta}_q\} \quad (2.2)$$

Donde $\vec{r} + \vec{\delta}_k$ denota las celdas que se consideran dentro de la vecindad que influye en la evolución de la celda en $\vec{r}$

Las vecindades para las celdas dependen siempre de la topología del espacio del modelo y se pueden definir de distintas maneras, las dos más comunes para CA bidimensionales son la vecindad de Von Neumann (Figura 2.2a) que consiste en la celda y sus cuatro vecinas inmediatas sin considerar sus diagonales, y la vecindad de Moore (Figura 2.2b), que consiste en la celda y sus 8 vecinas inmediatas.

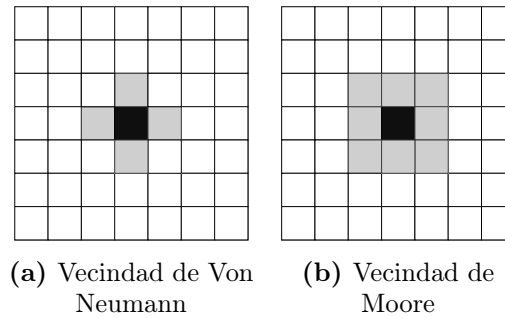


Figura 2.2: Vecindades más utilizadas en modelos de *CA* bidimensionales [2].

Regla de evolución

Las reglas de evolución, son un conjunto de reglas $R = \{R_1, R_2, \dots, R_m\}$, de complejidad arbitraria, que pueden ir desde una simple operación lógica hasta una función matemática no lineal. Estas reglas caracterizan la evolución en el tiempo de los estados de las celdas $C(\vec{r}, t)$ a partir de su interacción con los estados de las celdas de la vecindad de la siguiente manera:

$$C_j(\vec{r}, t+1) = R_j\{C(\vec{r}, t), C(\vec{r} + \vec{\delta}_1, t), \dots, C(\vec{r} + \vec{\delta}_q, t)\} \quad (2.3)$$

Existen diversas reglas de evolución que se pueden aplicar a modelos de *CA*, y estas también se pueden clasificar en categorías, sin embargo una de las más utilizadas y que justamente trabajan en este proyecto son las reglas totalísticas:

- **Reglas totalísticas:** Son reglas en donde la evolución de las celdas depende directamente del resultado de la suma de los estados de todas las celdas de la vecindad.

Condición inicial

Corresponde a la asignación inicial de los estados de cada una de las celdas de la malla.

Condiciones de frontera

Si bien, teóricamente los *CA* pueden extenderse infinitamente en un espacio, en la práctica, al implementarlos tanto en Software como en Hardware existen limitaciones, como la disponibilidad de memoria o de recursos físicos, por lo que se acotan a espacios donde la malla

tiene una dimensión finita, al hacer esto los elementos que se encuentran en los extremos de la malla presentan vecindades incompletas, por lo que se deben definir condiciones de frontera que suplan las vecinas carentes. Existen diversos tipos de condiciones de frontera, entre las más utilizadas se encuentran las siguientes [8]:

- **Frontera fija o abierta:** Considera que existen celdas de valor fijo (generalmente 0) fuera del espacio finito de la malla, este valor debe ser parte del conjunto de estados posibles de las celdas. Es el tipo de condición de borde más usado, pues reduce el número de posibles combinaciones a los que pueden evolucionar las celdas ubicadas en los límites de la malla.
- **Frontera reflectora:** Reemplaza los vecinos faltantes por copias de las celdas existentes en la vecindad, específicamente la celda ubicada al extremo opuesto de la celda central.
- **Frontera periódica o circular:** Las celdas interactúan con sus vecinos inmediatos y con las celdas que se encuentran en el extremo opuesto de la malla.
- **Frontera adiabática:** La vecindad se completa con celdas ficticias que son una replicas de la celda analizada o celda central de la vecindad.

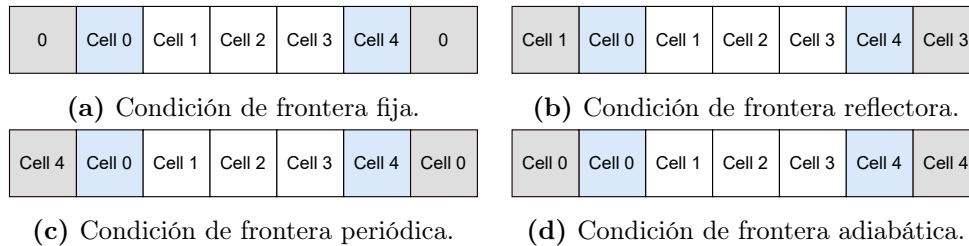


Figura 2.3: Ejemplo gráfico de condiciones de frontera más utilizadas en *CA* unidimensional.

2.2.2. Tipos de Autómatas Celulares

Las reglas de evolución que rigen el comportamiento de los *CA* en el tiempo, suelen ser homogéneas, es decir, se aplican indistintamente a todas las celdas de la malla sin importar su posición $\vec{r}$, sin embargo, la manera en que estas se aplican a cada modelo de *CA* puede determinarse según el problema específico que se desea atacar. Esto ha dado paso a que distintos autores

propongan variaciones del concepto de *CA*. En [2], se detallan algunas de estas variaciones, entre las que destacan las siguientes:

- **Autómatas celulares Uniformes:** Cumplen con la definición inicial del concepto de *CA*, aplican la misma regla de evolución a cada una de las celdas de la malla de manera simultánea generando una dinámica global. También se conocen como Autómatas celulares Homogéneos.
- **Autómatas celulares híbridos:** Un *CA* se dice Híbrido, si las reglas de evolución que determinan el próximo valor de cada elemento puede ser distinto a lo largo de la malla. Suele debatirse si esta estructura es clasificable como un *CA* dada la carencia del concepto de regla de evolución global, sin embargo, puede ser útil para representar sistemas en que interactúan elementos de distinta naturaleza, como la simulación de propagación de fenómenos en el espacio, donde cada celda puede representar distintas características del espacio en cuestión y así requerir una regla de evolución distinta.
- **Autómatas celulares asíncronos:** Una clase de *CA* que permiten evolucionar una célula a la vez. Al cambiar el valor de una celda, se alteran los vecindarios de todos los elementos pertenecientes a la malla. Al igual que en el caso de los Autómatas Celulares híbridos, suelen clasificarlos como una técnica distinta de los *CA* tradicionales, debido a que no cumplen con la condición de evolución simultánea, sin embargo, la dinámica de funcionamiento y la estructura son similares.
- **Autómatas celulares programables:** Son *CA* que además de considerar para su evolución los estados de la vecindad, emplean también algunas señales de control, cuyo valor en tiempo de ejecución permitirá implementar o modificar la regla de evolución vigente de manera dinámica.
- **Autómatas celulares probabilísticos:** Son una clase de *CA* en donde las reglas de evolución corresponden a reglas basadas en probabilidades de los valores presentes en las células. También se conocen como Autómatas celulares estocásticos.

2.3. Autómatas celulares y procesamiento de imágenes

Una imagen puede describirse matemáticamente como una función bidimensional I .

$$I = f(x, y) \quad (2.4)$$

Donde x e y son coordenadas espaciales en un plano, y f es cualquier par de coordenadas presentes en este. La amplitud de cualquier par de coordenadas (x, y) se conoce como intensidad I . Si las coordenadas espaciales y los valores de intensidad son cantidades discretas y finitas, la imagen se categoriza como una imagen digital [9]. Una imagen digital se compone de un número finito de elementos, cada uno con una posición y valor específicos, conocidos como *pixel*, estos se agrupan en una matriz bidimensional de $M \times N$ píxeles cuando se trata de imágenes en escala de grises o un conjunto de matrices bidimensionales de $M \times N$ cuando se trata de imágenes a color, donde cada una de estas corresponde a un canal de color, dependiendo del sistema utilizado, como por ejemplo el RGB, HSV, entre otros.

El procesamiento digital de imágenes, tiene como objetivo el poder, mediante uso de algoritmos y técnicas matemáticas aplicadas, lograr mejorar, analizar, interpretar y/o extraer información de imágenes digitales. Esto puede incluir tareas como la corrección de color, eliminación de ruido, detección de objetos, segmentación de imágenes, etc.

La mayoría de los algoritmos de procesamiento de imágenes se desarrollan mediante Software y de manera secuencial, lo que significa que para procesar una imagen se realiza un barrido pixel a pixel aplicando una fórmula matemática conocida como *kernel*, que considera el valor actual del pixel y los valores de los pixeles adyacentes (generalmente) y de esta manera calcular el nuevo valor del pixel posterior al procesamiento.

Como una imagen digital es una matriz bidimensional de tamaño $M \times N$, esta puede estudiarse como un *CA* bidimensional del mismo tamaño, en donde cada celda representa un pixel de la imagen y la intensidad de este está representado por el estado de la celda. Los valores de los pixeles se actualizan sincrónicamente en un paso de tiempo discreto, ya que los *CA* evolucionan de manera global y simultánea, por lo tanto, se pueden desarrollar tareas de procesamiento de imágenes en un tiempo reducido al explotar el paralelismo inherente a los *CA*.

2.4. Estado del arte

En esta sección se resumen brevemente algunos de los artículos, de entre los que fueron estudiados durante el proceso de investigación del presente trabajo, que resultaron de utilidad para el desarrollo del sistema propuesto.

- **An edge detection method using outer Totalistic Cellular Automata**

En [10], los autores presentan un método para detectar bordes en imágenes utilizando autómatas celulares totalísticos externos. El método se basa en la transformación de la imagen en una matriz binaria y la aplicación de reglas específicas de autómatas celulares para detectar bordes, entre ellas la regla 832 que se utiliza en este trabajo. Los autores compararon su método con otros métodos de detección de bordes en términos de precisión y velocidad, encontraron que su método proporciona resultados comparables o mejores en términos de precisión y es más rápido que algunos de los métodos existentes como el algoritmo de Canny. Concluyen que su método puede ser una herramienta útil para aplicaciones de detección de bordes en tiempo real.

- **Training Cellular Automata for Image Processing** Dentro de los trabajos de Rosin [11], se ha demostrado que es posible automatizar el aprendizaje de reglas para los CA, de forma que puedan realizar tareas de procesamiento de imágenes de manera eficiente. Concluyó que algunos de los beneficios que entrega el enfoque de procesamiento de imágenes basado en CA es su flexibilidad y capacidad de reutilización de la misma arquitectura para realización de diferentes tareas, donde solo se necesita aprender nuevas reglas para los CA y de esta forma adaptarse a una nueva tarea con nuevos datos. Evaluó los resultados de las reglas aprendidas para algoritmos de detección de bordes, morfología, *convex hull*, entre otros. Sin embargo, trabajos posteriores encontraron que **para ciertos algoritmos** de procesamiento las reglas propuestas por Rosin no lograron resultados adecuados [12].

- **Design and Implementation of Cellular Automata on FPGA for Hardware Acceleration**

En [13] describe la implementación de un acelerador de hardware de autómatas celulares en una FPGA. Los autores diseñaron una versión del clásico *CA Conway's Game*

of Life (*CGoL*) en lenguaje VHDL y lo implementaron en una FPGA buscando lograr una aceleración por hardware. Realizaron pruebas de rendimiento comparando la implementación en FPGA con la ejecución de software en una CPU, y encontraron que la implementación en FPGA otorgaba una velocidad de procesamiento de 36.7 veces la velocidad conseguida por en [14], en donde se realizaba una implementación similar del *CA* en una GPU GeForce GTX Titán X. También analizaron los resultados en términos de consumo de energía y encontraron que la implementación en FPGA consume menos energía que la ejecución de software en una CPU. Los autores concluyen que su diseño y método de implementación pueden ser útiles para acelerar la ejecución de algoritmos de autómatas celulares y otros algoritmos en aplicaciones que requieren procesamiento de datos en tiempo real.

Tras revisar el estado del arte en el procesamiento de imágenes basado en *CA*, se obtuvieron conclusiones que respaldan la decisión de utilizar un modelo de *CA* implementado en hardware, específicamente en una *FPGA*, en este trabajo. Estos hallazgos destacan la efectividad y las ventajas que los *CA* ofrecen en el campo del procesamiento de imágenes.

En primer lugar, se ha demostrado que los *CA* son una herramienta prometedora para técnicas de procesamiento de imágenes, como la detección de bordes. La aplicación de reglas específicas de *CA* ha demostrado proporcionar resultados comparables o incluso superiores en precisión en comparación con métodos tradicionales, al tiempo que se logra una mayor velocidad de procesamiento. Esto sugiere que los *CA* son una alternativa viable para aplicaciones de detección de bordes en tiempo real.

Además, la implementación de *CA* en hardware, como en una *FPGA*, ha demostrado proporcionar mejoras significativas en el rendimiento de procesamiento en comparación con la ejecución de software en una CPU. Esta implementación en hardware también ofrece ventajas en términos de consumo de energía, lo que la convierte en una opción atractiva para acelerar la ejecución de algoritmos de *CA* y otros algoritmos utilizados en aplicaciones que requieren un procesamiento eficiente de datos en tiempo real.

3. Modelo de Autómata Celular y módulo diseñado para su implementación

En el capítulo previo se abordó el tema de los autómatas celulares, donde se describieron sus características fundamentales como la célula, dimensión, reglas de evolución y vecindario. Además, se mencionaron algunas reglas de evolución tales como las totalísticas y las lógicas, las cuales determinan el estado de cada celda. En términos de imagen, esta se compone de un grupo de píxeles con valores finitos que se organizan en una matriz $M \times N$, lo cual se asemeja a las propiedades de un autómata celular. Es posible establecer un conjunto de reglas de evolución para el *CA* que emulen la ejecución de algoritmos de procesamiento de imágenes.

En este capítulo se propone el desarrollo de un modelo de *CA* que permita la ejecución de tareas de procesamiento digital de imágenes binarias, en particular, detección de bordes y erosión. Para esto, primero se plantean una serie de propiedades y consideraciones que caracterizarán el modelo de *CA* a implementar, luego se explican y definen las reglas de evolución que permitirán cumplir el objetivo de realizar procesamiento de imágenes binarias para finalmente pasar a la descripción del diseño e implementación del modelo en una versión en software y un equivalente implementable en hardware, específicamente en una *FPGA*.

En base a los conceptos expuestos en la sección 2.2, para la definición del *CA* se tienen las siguientes consideraciones del modelo:

- Vecindad de Moore de radio $r = 1$.
- Reglas totalísticas.
- Espacio de valores binario: $\{0, 1\}$
- Condiciones de borde Reflexiva

3.1. Reglas de evolución

Para poder lograr la ejecución de las tareas de procesamiento de imágenes de interés, se escogieron dos reglas de evolución en particular, una para la detección de borde y otra para la erosión.

3.1.1. Detección de bordes

Un borde se define como el conjunto de pixeles que se ubican en la frontera entre dos regiones que presentan un cambio brusco en su color o intensidad dentro de una imagen. Estos cambios pueden deberse a variaciones en la iluminación, textura, sombras, profundidad, entre otros factores.

Para poder realizar detección de bordes mediante *CA* se utilizó una leve adaptación de la regla 832 que se menciona en [10]. Esta consiste en:

- **Pixel de borde:** Si la suma de la vecindad es 3 o 4 el nuevo valor del pixel/celda será 1.
- **Pixel de fondo:** Si la suma de la vecindad es distinta de 3 y 4 el nuevo valor del pixel/celda será 0.

3.1.2. Erosión

La erosión es una operación morfológica que se utiliza en el procesamiento digital de imágenes para reducir el tamaño de los objetos en una imagen, eliminando pequeñas irregularidades y puntos aislados en el objeto. Consiste en contraer la imagen, haciendo una comparación de cada pixel de la imagen con su entorno inmediato. Para esto se escogió una regla simple que actúa de manera complementaria a un operador OR en la vecindad, es decir, si la suma de los pixeles de la vecindad es menor o igual a 8, el valor siguiente del pixel/celda será 0, en caso contrario mantendrá su valor actual. Esto se muestra gráficamente en la Figura 3.4

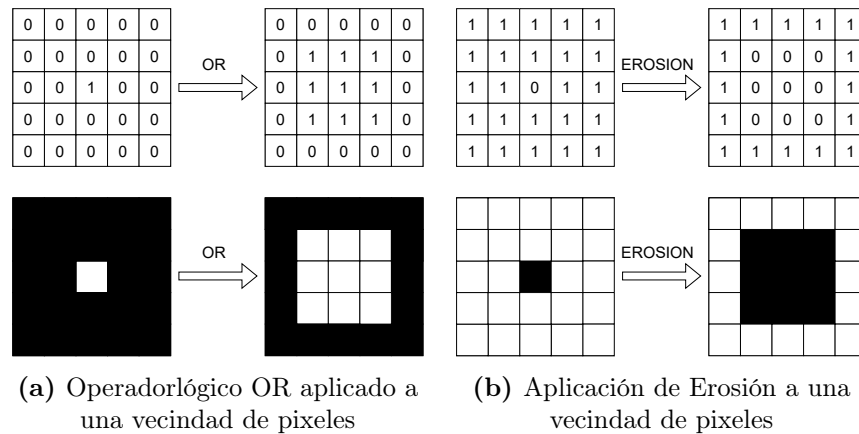


Figura 3.4: Operador lógico OR y operador lógico NOR como técnica de erosión.

3.2. Diseño de un Autómata Celular en Software

En el capítulo se detalla el proceso de desarrollo en software, a través del lenguaje de programación Python en su versión 3.10.7, de un autómata celular capaz de realizar erosión y detección de bordes en una imagen binaria. Se expone el diseño desde la creación de la malla hasta la implementación de la regla de evolución. Se explicará cada una de las funciones y procedimientos necesarios para el funcionamiento del autómata celular creado.

3.2.1. Funciones implementadas

Para la implementación del *CA* se crearon 6 funciones, con el fin de crear, inicializar, definir la regla de evolución y ejecutar la evolución del *CA*.

- `CA_init(heigth ,width, image)`
- `read_cell_state(col,row, CA, dimensions)`
- `find_neighborhood(col,row,CA, dimensions)`
- `rule(neighborhood_state, n_rule)`
- `save_CA(CA, CA_buff, dimensions)`
- `update_CA(CA, CA_buff, dimensions, n_rule)`

Los códigos de dichas funciones se encuentran en la sección 8, referida a los Anexos. Sin embargo, a continuación se detalla y explica la lógica y el funcionamiento de cada una de ellas.

1. `CA_init(heigth ,width, image)`

Esta función se encarga de crear la malla inicial del *CA*, para esto recibe como entrada una imagen, junto a sus dimensiones `heigth` y `width`, con las que inicializa una matriz del mismo tamaño de la imagen, a la vez que recorre cada uno de los píxeles de la propia imagen para copiarlos en la matriz creada. De esta forma, setea el valor inicial de las celdas de la malla del *CA*.

Finalmente, la función devuelve una cuadrícula con los valores de píxel correspondientes a la imagen de entrada.

2. `Read_cell_state(col,row, CA, dimensions)` Esta función se utiliza para leer el estado actual de una celda en particular, para esto recibe como parámetros las coordenadas de la celda, `col` y `row`, correspondientes a la columna y fila en que se encuentra dentro de la malla del *CA*, a su vez también recibe la matriz correspondiente al propio *CA* junto a sus dimensiones, que son necesarias, puesto que esta función también ajusta las coordenadas según las condiciones de frontera que rigen el modelo de *CA* implementado. Por ejemplo, en el caso de la implementación propuesta, se considera condición de frontera reflexiva, por lo que si la columna escapa de las dimensiones de la malla por la izquierda, la función analizará la celda ubicada en la segunda columna del *CA*. Finalmente, la función devuelve el valor del píxel correspondiente a la celda.

3. `Find_neighborhood(col,row,CA, dimensions)` Esta función se utiliza para encontrar la vecindad de una celda dada en el *CA*, recibe como parámetros coordenadas de la celda, `col` y `row`, junto a la matriz del *CA* y sus dimensiones. Esta función llama a la función antes descrita, `read_cell_state` 9 veces, una por cada elemento de la vecindad asociada a la celda actual que se está analizando, por ejemplo, para obtener el estado de la celda se llama de la forma:

```
1      Main_state = find_neighborhood(col,row,CA, dimensions)
```

Para obtener el estado de la celda ubicada al NO se llama de la forma:

```
NO = find_neighborhood(col-1,row -1 ,CA, dimensions)
```

Finalmente, devuelve una lista de los valores de todas las celdas que conforman la vecindad asociada a la celda dada.

4. **Rule(neighborhood_state, n_rule)** Esta función define la regla de evolución que regirá el comportamiento del *CA* en el tiempo, dado que el *CA* propuesto utiliza reglas totalísticas, es decir, el nuevo estado de las celdas dependerá únicamente del valor de la suma de los estados de las celdas de la vecindad, la función recibe como entrada la lista que entrega la función antes descrita `find_neighborhood(col, row,CA, dimensions)` y dado que el *CA* implementa dos reglas de evolución, erosión y detección de bordes, la función recibe también un entero `n_rule`, que indicará que regla se debe ejecutar en el instante de la evolución: 0 para ejecutar erosión y 1 para ejecutar detección de bordes.

Dado que los estados de las celdas se extraen directamente de una imagen binaria, los estados posibles son únicamente 0 y 255, por lo que la función se encarga de normalizar los valores a 0, 1, luego calcula la suma de la lista de estados, determina el nuevo estado de la celda según la regla que corresponda y retorna este valor.

5. **Save_CA(CA, CA_buff, dimensions)**

Esta función se utiliza para guardar el estado actual de la matriz del *CA* en una nueva matriz de respaldo, por lo que toma como entrada la matriz original *CA* y copia cada uno los estados de las celdas en la nueva matriz *CA_buff*. Esto es necesario, ya que, teóricamente todas las celdas de un Autómata Celular evolucionan simultáneamente de forma paralela, sin embargo, al ser una implementación en software esto no es posible y las celdas evolucionan de manera secuencial una a una, por lo que, al evolucionar una celda, las celdas siguientes tomarían en consideración este nuevo valor de celda dentro de su vecindad provocando una evolución incorrecta en cadena a lo largo de toda la malla. Esto se evita con la matriz de respaldo, pues se utilizan los estados almacenados en ella para aplicar la regla de evolución a las celdas, mientras que el estado resultante luego de su ejecución es el que se almacena en la nueva matriz, sin afectar la evolución de la malla con los nuevos estados.

Para poder realizar este proceso de respaldo correctamente la función también recibe las dimensiones de la malla a través del argumento `dimensions`, que es una tupla que contiene el ancho y alto de la malla.

6. `Update_CA(CA, CA_buff, dimensions, n_rule)`

Esta función se utiliza para actualizar el estado del *CA*. La función toma como entrada: la matriz original del *CA*, *CA*; la matriz de respaldo, *CA_buff*; las dimensiones de la malla, *dimensions* y el número asociado a una regla de evolución específica, *n_rule*.

Lo primero que hace la función es guardar el respaldo de la matriz en *CA_buff* usando la función *Save_CA*, luego mediante la función *Find_neighborhood()* encuentra la vecindad de cada celda y aplica la regla de evolución al llamar a la función *Rule()* a cada celda, actualizando de esta forma el estado del *CA*.

Con estas funciones implementadas, el programa que crea y evoluciona el *CA* para procesamiento de imágenes binarias puede verse como el siguiente extracto de código:

```
#carga de la imagen
imagen = cv2.imread( <nombre_de_la_imagen>)
#obtencion de dimensiones
4 height,width,_ = imagen.shape

#inicializacion del CA
CA = CA_init(width,height, imagen)
buff = CA_init(width,height,imagen)
9

#evolucion del CA
update_CA(CA,buff, (height,width), 0) #para erosion
update_CA(CA,buff, (height,width), 1) #para deteccion de bordes

14 #actualizacion de la imagen a imagen procesada
Update_img(imagen, CA, (height,width))
```

3.3. Diseño de Autómata Celular en Hardware

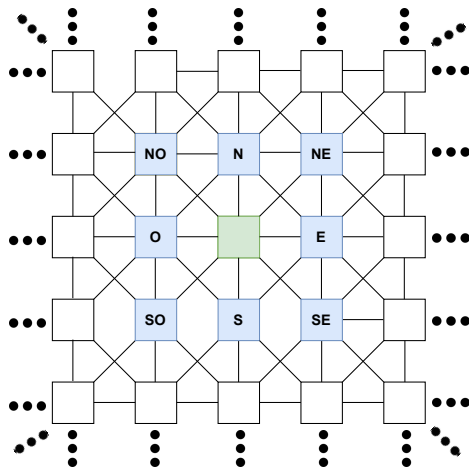
El diseño en hardware del modelo de *CA* representa una de las contribuciones más importantes de este trabajo. El objetivo principal de esta sección es presentar la descripción detallada del proceso de diseño del hardware utilizado para implementar el modelo, incluyendo la topología del circuito, los módulos diseñados y los detalles de su operación.

En el diseño en hardware del *CA*, se buscaron alternativas para llevar a cabo la implementación de la misma regla de evolución utilizada en el software, pero con un enfoque más eficaz en términos de tiempo de ejecución y eficiente en uso de recursos de la plataforma *FPGA* donde se implementará. Para ello, se utilizó una combinación de técnicas de diseño de circuitos digitales y programación en lenguaje *SystemVerilog* a través de la herramienta Vivado 2022.1.

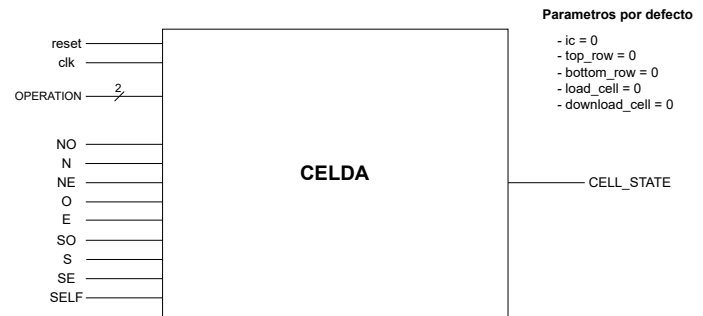
3.3.1. Módulos de Celda

Se diseña el módulo *CELDA.sv*, que corresponde a la descripción de Hardware asociado a las celdas que conforman la malla del *CA* y se puede encontrar en la sección 8 al final del documento. La Figura 3.5b muestra el diagrama de alto nivel correspondiente a dicho módulo.

El módulo consta de doce entradas, que corresponden a los 8 estados de las celdas vecinas NO, N, NO, O, E, SO, S, SE; el estado actual de la celda, *SELF* (para fines de inicialización); la señal de reloj y reset; todas estas de 1 bit. Además, tiene como entrada la señal *OPERATION*, proveniente del módulo de control que se detalla en la Sección 3.3.2, esta señal definirá el modo de operación en el que trabajarán las celdas. Por otro lado, la única salida del módulo es el estado siguiente de la celda, la señal *CELL_STATE*.



(a) Malla de celdas del CA



(b) Diagrama de alto nivel de la Celda

Figura 3.5: Diagrama de malla de CA y diagrama del módulo CELDA.

Las celdas tienen cuatro modos posibles de operación:

- **Ocioso:** Modo en el que el sistema no hace nada, todas las celdas mantienen su estado.
- **Carga de datos:** Modo en el que se cargan los pixeles de una imagen de entrada en las celdas de la malla del CA.
- **Descarga de datos:** Modo en el que se descargan estados de las celdas luego de haber evolucionado (o resultado del procesamiento de los pixeles) de la malla del CA.
- **Evolución Normal:** Modo en el que todas las celdas del CA evolucionan según la regla de transición definida.

La carga de los valores de los pixeles de la imagen a procesar se realiza de manera semi-paralela, de forma similar a la que se propone en [15], debido a la cantidad limitada de pines de Entrada/Salida que presenta la tarjeta de desarrollo utilizada. Esta explota el hecho de que cada celda está conectada a sus vecinas, por la naturaleza de los CA y consiste en ingresar de forma secuencial un pixel de la imagen a la vez, utilizando como entrada de datos únicamente la celda ubicada en el extremo superior izquierdo de la malla, en cuanto se recibe un nuevo pixel, este valor se almacena como estado de la celda de entrada y el valor anterior de esta se copia en la celda ubicada a su derecha. Así sucesivamente hasta haber cargado un nuevo

estado a todas las celdas ubicadas en la fila superior de la malla (fila 0), en este instante, al recibir un nuevo valor de pixel, en lugar de hacer la copia horizontal que se hacía hasta ahora, se realiza una copia vertical entre las filas de la malla, a la vez que la celda de entrada almacena el nuevo valor ingresado. El proceso descrito se muestra gráficamente en la Figura 3.6a.

De forma similar, la descarga de datos se realiza mediante una única celda de salida, correspondiente a la celda ubicada en la esquina inferior derecha de la malla. Los valores de las celdas de la última fila (fila N) se propagan a lo largo de esta copiando el valor de la celda que se encuentra a su izquierda, de esta forma, la celda de salida toma eventualmente cada uno de los valores almacenados en la fila, cuando todos estos han sido descargados, todas las filas de la malla copian los valores almacenados en la fila anterior (la que se encuentra por encima), y se repite el proceso hasta haber descargado todos los estados de las celdas de todas las filas del CA. El proceso descrito se muestra gráficamente en la Figura 3.6b.

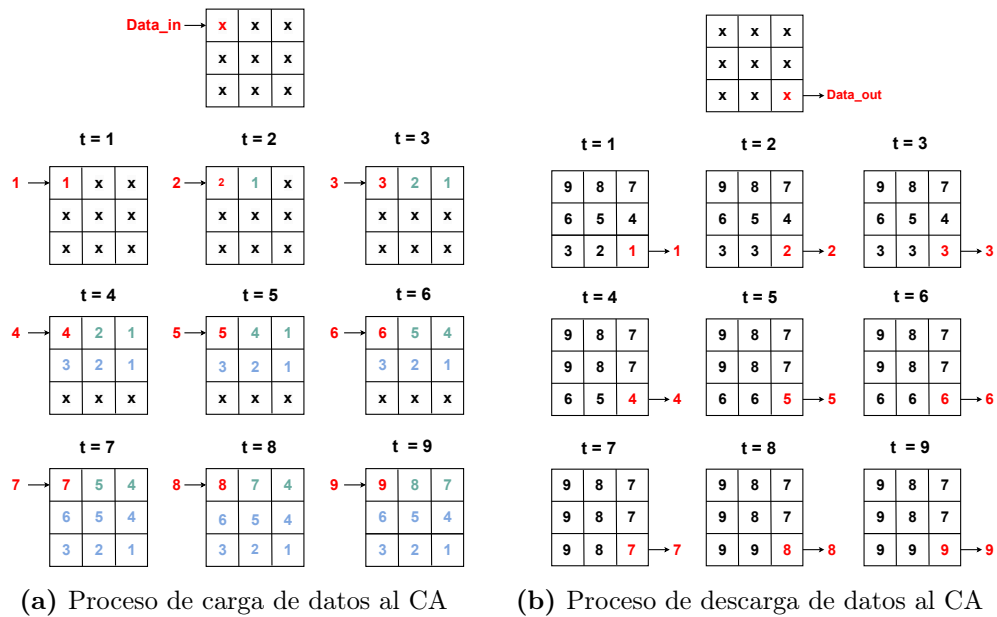


Figura 3.6: Ejemplo gráfico del proceso de carga/descarga de datos a una malla de CA de tamaño 3x3.

Para la implementación de este mecanismo de carga y descarga, se definen una serie de parámetros que se setean al momento de efectuar la instancia la las celdas, estos son:

- **TOP_ROW:** Un 1 indica si se trata de una celda presente en la primera fila de la malla (fila 0), y un 0 en caso contrario.
- **BOTTOM_ROW:** Un 1 indica si se trata de una celda presente en la última fila de la malla (fila N), y un 0 en caso contrario.
- **LOAD_CELL:** Un 1 indica que se trata de la celda de entrada, a través de la cual se cargaran los datos a la malla, 0 en caso contrario.
- **DOWNLOAD_CELL:** Un 1 indica que se trata de la celda de entrada, a través de la cual se cargaran los datos a la malla, 0 en caso contrario.

De esta forma, se pueden diferenciar 5 tipos de celdas: Celda corriente, Celda de entrada, Celda de la fila superior, Celda de salida y Celda de la fila inferior. La Figura 3.7b expone de manera gráfica lo descrito anteriormente, las celdas en color azul corresponden a las Celdas de la fila superior; las celdas en blanco son las Celdas corrientes, y las celdas en color verde son las Celdas de la fila inferior. Notar que la Celda de entrada y Celda de salida son un caso particular de la fila superior y fila inferior respectivamente, estas son las únicas celdas con dos parámetros en 1, **TOP_ROW** y **LOAD_CELL** en el caso de la Celda de entrada y **BOTTOM_ROW** a la vez que **DOWNLOAD_CELL** en el caso de la Celda de salida.

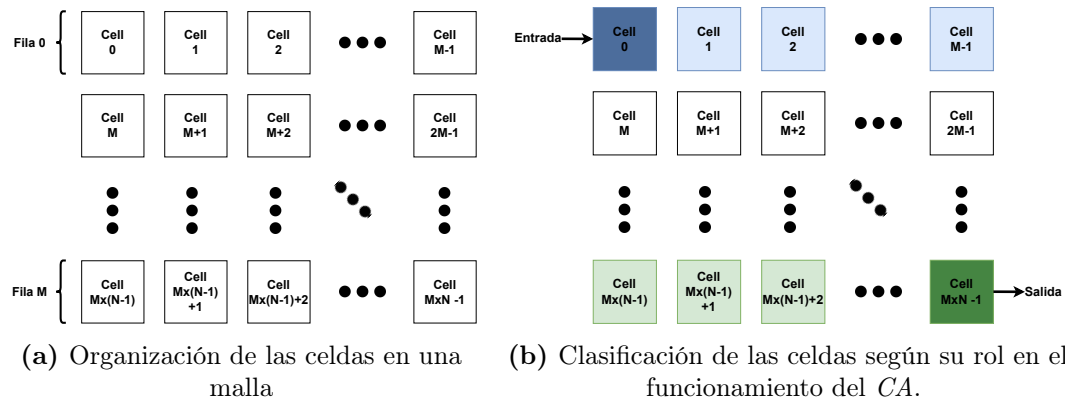


Figura 3.7: Malla genérica de celdas para un CA.

Para la malla de celdas que conforman el CA, se utiliza el módulo **GRID_SCRIPT**, que se genera automáticamente con instancias para las $M \times N$ celdas con todas las interconexiones necesarias

entre ellas mediante un script de Python. Este módulo consta de dos entradas: `DATA_IN` que corresponde al dato de se carga a la malla del *CA* y la señal `OPERATION`, que se le entregará como entrada también a todas las celdas instanciadas en este módulo.

3.3.2. Módulos de control

En esta sección, se presenta el módulo `FSM_OPERATION`, que corresponde a una *FSM*, la cual controla la operación del *CA*. El diagrama de estados de dicho módulo se muestra en la Figura 3.8. El módulo presenta un estado de espera y tres procesos posibles asociados a 6 estados. Estos se describen a continuación:

Estado en espera de instrucciones Este estado considera un único estado de la *FSM*, este es:

- **IDLE**: estado en el que el sistema no hace nada y el *CA* mantiene su estado actual.

Proceso de carga de datos

Este proceso considera dos estados de la *FSM*, estos son:

- **LOAD_WAIT**: estado en que el sistema está a espera de un nuevo valor de pixel para ser cargado al *CA* a través de la Celda de entrada.
- **LOAD**: estado en que el valor de pixel recibido es almacenado.

Proceso de descarga de datos

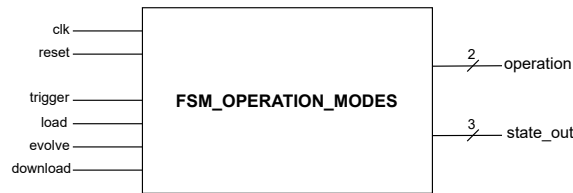
Este proceso considera dos estados de la *FSM*, estos son:

- **DOWN_WAIT**: estado en que el sistema está a espera de un nuevo valor de pixel para ser descargado desde el *CA* a través de la Celda de entrada.
- **DOWN**: estado en que el valor de pixel recibido es almacenado.

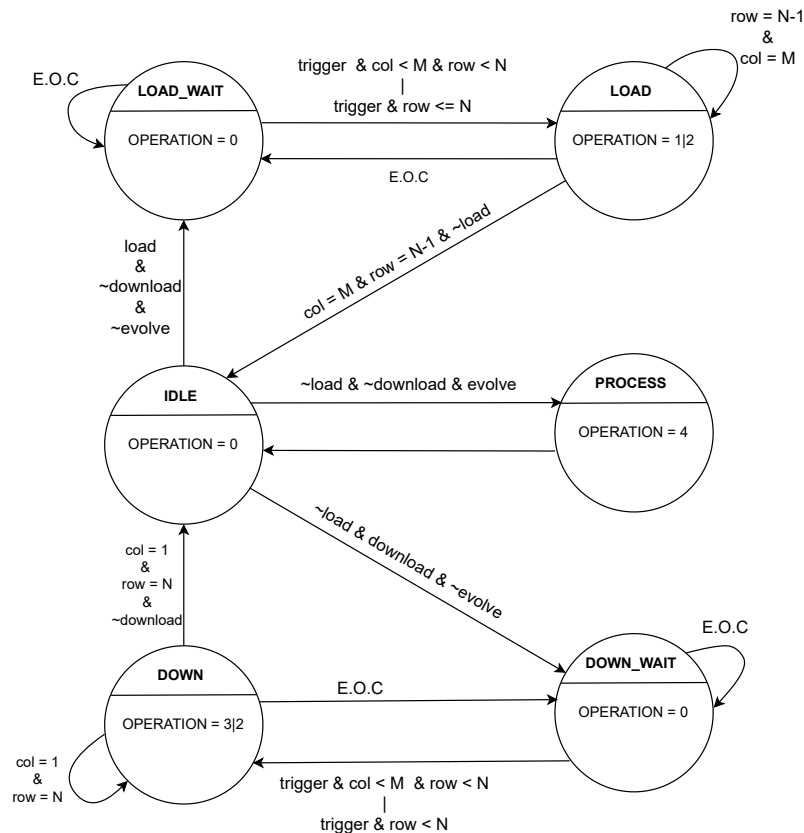
Evolución del CA

Este proceso se conforma de un único estado de la *FSM*, este es:

- **PROCESS**: estado en que se ejecuta la regla de evolución del *CA*, en este trabajo se implementaron dos reglas por separado, por lo que dependiendo del caso este estado ejecutará la regla de erosión o la de detección de bordes según corresponda.



(a) Diagrama de alto nivel de la *FSM* que gobierna el sistema.



(b) Diagrama de estados de la *FSM* que gobierna el sistema.

Figura 3.8: Diagrama de alto nivel del módulo **FSM_OPERATION_MODES** y diagrama de estados de la *FSM* asociada.

El módulo consta de 6 entradas de 1 bit cada una: las señales de reloj y reset; 4 señales de control, **LOAD**, **DOWNLOAD** y **EVOLVE**, que indicaran el tipo de proceso que se debe ejecutar (Carga, Descarga, o Evolución), además de la señal **TRIGGER** que indica la recepción de un nuevo valor de pixel que se debe cargar al *CA* o bien que se ha transmitido correctamente un valor de pixel descargado desde el *CA*, según el proceso que se esté desarrollando.

Por otro lado, el módulo cuenta con dos salidas de 3 bits cada una: la señal **OPERATION**, que se conectará a cada una de las celdas de la malla e indicará cómo deben operar estas en el ciclo actual de reloj (los valores posibles de esta señal y las operaciones asociadas se detallan más adelante en esta misma sección); y la señal **STATE_OUT**, que se utiliza para indicar al resto del sistema en que estado se encuentra actualmente.

Adicionalmente, para tener control de las celdas que se han cargado o descargado del *CA*, el módulo consta de 2 contadores internos **COL** y **ROW**, asociados a las *M* columnas y *N* filas que de una malla de tamaño $M \times N$.

La señal principal de este módulo es **OPERATION**, ya que define el comportamiento de todas las celdas del *CA*, de la siguiente manera:

- **OPERATION** = 0: El *CA* Mantiene su estado actual.
- **OPERATION** = 1: Se realiza un corrimiento horizontal únicamente en las celdas de la primera fila (Fila 0).
- **OPERATION** = 2: Se realiza un corrimiento vertical en toda la malla.
- **OPERATION** = 3: Se realiza un corrimiento horizontal únicamente en las celdas de la última fila (Fila *N*).
- **OPERATION** = 4: Se ejecuta la regla de evolución en toda la malla.

Lógica de la *FSM*

El contador de columnas, **COL**, está asociado a la cantidad de pixeles que se han cargado satisfactoriamente en la primera fila de la malla (Fila 0), una vez este contador llega al límite *M* (que viene de las dimensiones de la malla $M \times N$), se reinicia a 0 mientras que el contador

de filas, **ROW**, se incrementa en 1. Este proceso se repite hasta que el contador de filas haya llegado a su límite **N**.

El estado inicial de la *FSM* es **IDLE**, a su vez, los contadores **COL** y **ROW** se inicializan en 0 al igual que la señal de salida **OPERATION** se mantiene en este estado hasta que se presente un cambio en alguna de las señales de control de procesos **load**, **DOWNLOAD** o **EVOLVE**. Estas señales son excluyentes entre sí, por lo que solo una de ellas debe estar en alto para poder hacer una transición de estado, en caso contrario el sistema se mantiene en **IDLE**.

Si la señal **LOAD**, que habilita la carga de una nueva imagen, toma un valor en alto, se pasa al estado **LOAD_WAIT**, en donde se queda a la espera de la recepción de un nuevo valor de pixel. En este punto existen 3 casos posibles que se detallan a continuación y se muestran de forma gráfica en la Figura 3.9a:

- Recepción de un nuevo valor de pixel (**trigger** = 1), mientras que los contadores de columnas y filas aún no llegan a las dimensiones de la malla **M** y **N**: En este caso, aún no se han cargado valores a todas las celdas de la Fila 0, por lo que **col** se incrementa en 1, **OPERATION** toma el valor de 1 y se pasa al estado **LOAD**.
- Recepción de un nuevo valor de pixel (**trigger** = 1) mientras que el contador de columnas ha llegado a su límite: En este caso, ya se han cargado valores a todas las celdas de la Fila 0, por lo que se debe realizar un corrimiento vertical en la malla a la vez que se carga un valor en la Celda de entrada, se setea **col** en 1, ya que se ha cargado una celda, **OPERATION** toma el valor de 2, **row** se incrementa en 1, ya que se ha completado la carga de una fila y se pasa al estado **LOAD**.
- No se ha recibido ningún nuevo valor de pixel: En este caso, no se reconoce ninguna recepción de un nuevo valor de pixel a cargar, por lo que el sistema se mantiene en el mismo estado, **LOAD_WAIT**.

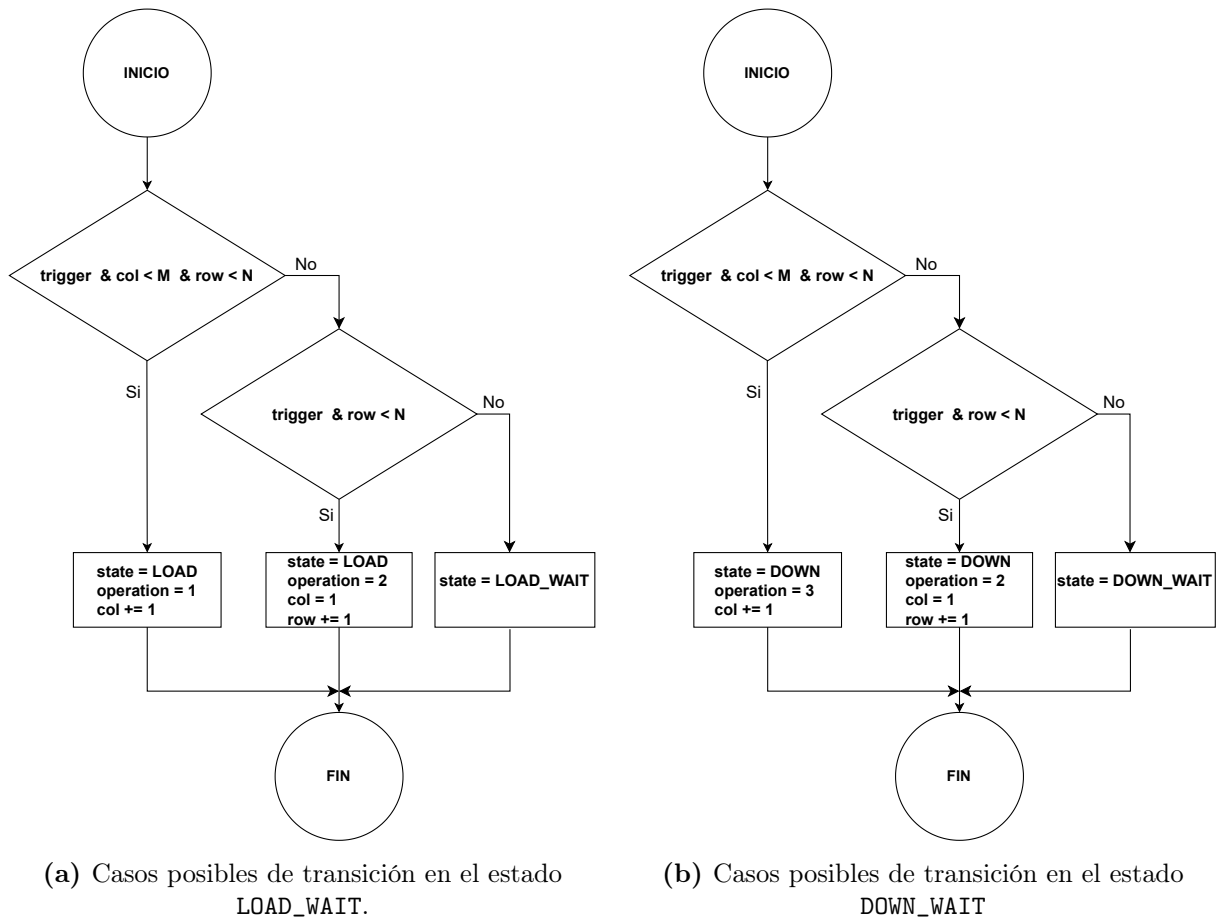


Figura 3.9: Diagrama de flujo de los casos posibles al ocurrir una transición en los estados LOAD_WAIT y DOWN_WAIT.

En el estado LOAD se realiza la carga del valor de pixel recibido en la Celda de entrada, tomando esta operación un ciclo de reloj para luego volver al estado LOAD_WAIT si es que la malla no se ha cargado completamente o al estado IDLE en caso de que todas las celdas hayan sido cargadas con un valor de pixel (COL y ROW alcanzaron los límites de las dimensiones de la malla), si esto sucede, ambos contadores se reinician y se setean en 0. En cualquiera de los casos mencionados la señal OPERATION se setea en 0.

Estando en el estado IDLE, si la señal EVOLVE, que habilita la evolución del CA, toma un valor en alto, se pasa al estado PROCESS, a la vez que la señal OPERATION se setea en un valor de 4. El sistema se queda en este estado por solo un ciclo de reloj, para finalmente volver al estado IDLE, en donde OPERATION tiene como valor 0 y el CA se mantiene ocioso.

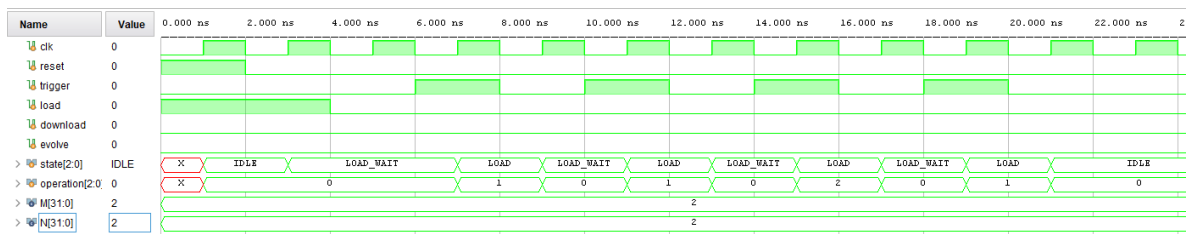
En el estado **IDLE**, también puede tomar un valor alto la señal **DOWNLOAD**, la que indica que se inicia el proceso de descarga de los estados de las celdas del *CA* pasando al estado **DOWN_WAIT**, en donde se presentan 3 casos posibles al igual que en el proceso de carga y también se muestran de manera gráfica en la Figura 3.9b, estos son:

- Transmisión de un nuevo valor de pixel (**TRIGGER** = 1), mientras que los contadores de columnas y filas aún no llegan a las dimensiones de la malla *M* y *N*: En este caso, aún no se han descargado los valores de todas las celdas de la Fila *N*, por lo que **COL** se incrementa en 1, **OPERATION** toma el valor de 3 y se pasa al estado **DOWN**
- Transmisión de un nuevo valor de pixel (**TRIGGER** = 1) mientras que el contador de columnas ha llegado a su límite: En este caso, ya se han descargado los estados de todas las celdas de la Fila *N*, por lo que se debe realizar un corrimiento vertical en la malla a la vez que se descarga un valor desde la Celda de salida, se setea **COL** en 1, ya que se ha descargado una celda, **OPERATION** toma el valor de 2, **ROW** se incrementa en 1, pues se ha completado la descarga de una fila y se pasa al estado **DOWN**.
- No se ha transmitido ningún nuevo valor de pixel: En este caso, no se reconoce ninguna transmisión completa de un nuevo valor de pixel a descargar, por lo que el sistema se mantiene en el mismo estado, **DOWN_WAIT**.

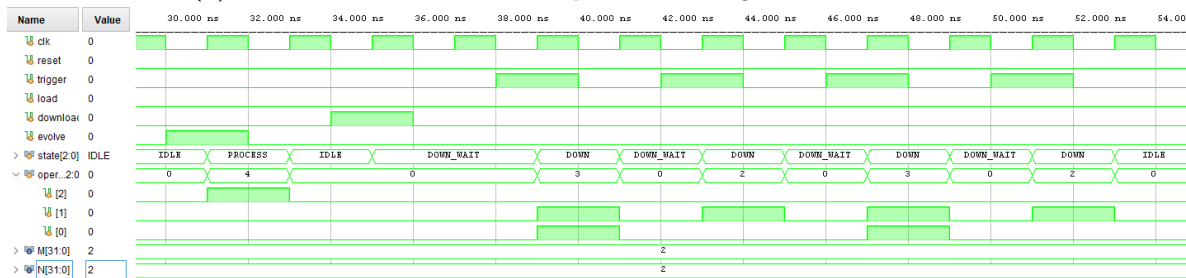
En el estado **DOWN** se realiza la descarga del valor del estado actual en la Celda de salida, tomando esta operación un ciclo de reloj para luego volver al estado **DOWN_WAIT** si es que la malla no se ha descargado completamente o al estado **IDLE** en caso de que el estado de todas las celdas hayan sido descargadas (**COL** y **ROW** alcanzaron los límites de las dimensiones de la malla). Al igual que en el caso del estado **LOAD**, si esto sucede, ambos contadores se reinician y se setean en 0. En cualquiera de los casos mencionados la señal **OPERATION** se setea en 0.

La Figura 3.10a muestra una simulación de como trabajan las señales de este módulo en el proceso de carga datos de una malla de $M \times N = 2 \times 2$. En ella se ve que para cargar 4 datos primero es necesario un pulso en la señal **load**, para pasar al estado **LOAD_WAIT**, luego con 2 pulsos de la señal **TRIGGER** se cargan los 2 datos gatillando un corrimiento horizontal en la malla a través del valor de la señal **OPERATION**, en un nuevo pulso de **TRIGGER** se carga

un nuevo dato a la vez que se provoca un corrimiento vertical en la malla mediante la señal **OPERATION**, para luego con un último pulso de **TRIGGER** se carga el último dato, se realiza un último corrimiento y se vuelve al estado **IDLE**. De forma similar La Figura 3.10b, muestra la el comportamiento de las señales de la *FSM* en el proceso de evolución del *CA* y la descarga de los datos luego de esta.



(a) Simulación de la *FSM* en el proceso de carga datos en el sistema.



(b) Simulación de la *FSM* en el proceso de evolución del *CA* y descarga datos desde el sistema.

Figura 3.10: Simulación de la *FSM* que controla la operación del *CA*.

3.3.3. Módulos de comunicación

En esta sección se describe el diseño del mecanismo de comunicación que utiliza el sistema para poder efectuar la carga y descarga de imágenes hacia y desde la *FPGA*.

El sistema consta de dos dispositivos o *endpoints*, en una configuración *host/device*, donde un computador cumple el rol de *host* y es el encargado de transmitir las imágenes que se pretende procesar mediante el *CA* implementado en la *FPGA*, que toma el rol de *device*. Ambos dispositivos se comunican mediante una interfaz de comunicación serial *UART*. La Figura 3.11 muestra un diagrama de alto nivel del sistema.

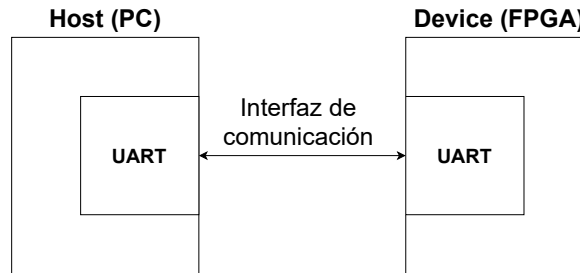


Figura 3.11: Diagrama de alto nivel de dispositivos conectados a través de interfaz *UART*.

Para efectuar la comunicación entre ambos dispositivos *endpoint* se utilizan los siguientes parámetros en la interfaz *UART*:

- *baudrate*: 115200
- Bits de stop: 1
- Paridad: Sin paridad

A continuación se detalla la implementación de la interfaz de comunicación en cada uno de los *endpoints*, el *device* y el *host*.

UART en el *device*

Para la implementación de la interfaz *UART* del lado del *device* se utilizaron como base los códigos de referencia los de la experiencia asociada al uso de protocolo *UART* del curso *ELO212* disponibles en el repositorio GitHub [16], gracias a los cuales se implementó el módulo *UART_LOGIC*, encargado tanto de la recepción de datos por parte de la *FPGA* así como de la transmisión de datos hacia el *host*. La Figura 3.12 muestra el diagrama de bloques de dicho módulo junto a las entradas y salidas que lo componen.

Internamente, este módulo cuenta con dos módulos principales: *UART_RX*, asociada a la recepción de datos por parte de la interfaz. Este módulo tiene como entrada la señal *rx*, la cual corresponde a cada bit del frame que se está recibiendo y los almacena en el buffer *rx_data* de 8 bits, una de sus salidas. En cuanto el buffer haya almacenado todos los bits de información del frame que se está recibiendo, la salida *rx_ready* pasa a estado alto, indicando que se ha completado la recepción de datos. Por otro lado, de manera análoga el módulo asociado a la

transmisión de la interfaz, **UART_TX** recibe como entradas dos señales, **tx_start** que indica que se debe comenzar la transmisión de la segunda entrada, **tx_data**, señal de 8 bits que corresponde a la información que se debe transmitir. En cuanto a las salidas, se tiene la señal **tx**, correspondiente a cada bit individual que conforman la señal **tx_data** y que se transmitirán uno a uno hasta haber enviado todos los bits de información, y la señal **tx_busy**, que indica si la interfaz de comunicación se encuentra en plena transmisión de un frame o está disponible para comenzar la transmisión de uno nuevo. En los módulos **UART_RX** y **UART_TX** se omitieron las entradas **clk** y **reset** para facilitar la comprensión del diagrama.

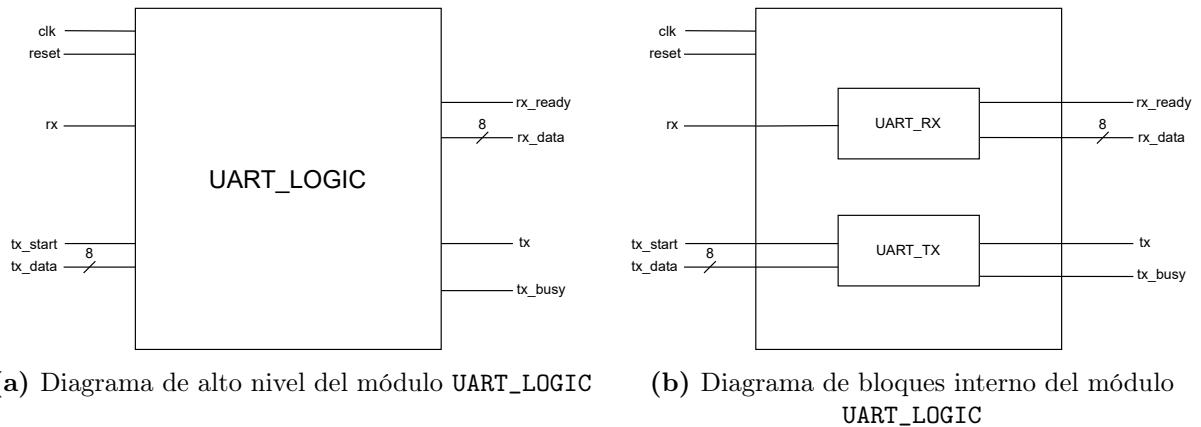


Figura 3.12: Diagrama del módulo de interfaz *UART* implementado en el *device*.

UART en el *host*

Para la implementación de interfaz de comunicación *UART* del lado del *host*, se desarrolló un script en lenguaje Python, en su versión 3.10.7, el cual permite hacer uso del puerto serial del computador a través de la biblioteca **serial**.

Se configura el puerto serial de forma que los parámetros coincidan con los que presenta la interfaz *UART* por parte del *device*, agregando además una ventana de 4 segundos de *timeout*, tal como se muestra a continuación:

```

serial_port = serial.Serial()
serial_port.port = 'COM<puerto_com_asignado>'
serial_port.baudrate = 115200
4 serial_port.parity = serial.PARITY_NONE

serial_port.timeout = 4
  
```

Una vez configurado el puerto serial para la comunicación, se utilizan los métodos de transmisión y recepción de datos según corresponda:

```
serial_port.write(data) #para transmitir informacion  
serial_port.read(data_size) #para recibir informacinn
```

Donde los argumentos de los métodos `data` y `data_size` corresponden a la información a enviar, en este caso los pixeles de la imagen a procesar y la cantidad de pixeles o frames que se recibirán por parte del *device* respectivamente.

En cuanto a los pixeles, dado que el formato de imágenes que se trabaja es binario, es decir, las intensidades de sus pixeles pertenecen al conjunto $\{0, 255\}$, estos se normalizan dividiendo por 255 antes de la transmisión de los mismos, llevándolos al conjunto $\{0, 1\}$, de esta forma los pixeles transmitidos serán de tipo binario de un solo bit, permitiendo optimizar el uso de recursos de la *FPGA* al almacenarlos dado que el estado de cada celda será representable por un bit. El proceso inverso se realiza al recibir los pixeles de la imagen ya procesada, ya que estos se multiplican por 255 para poder reconstruir una imagen binaria en el *host*. Si bien, con esta técnica se optimiza el uso de recursos físicos utilizados para almacenar el estado de los pixeles, este no es un método eficiente a la hora de transmitir la información, ya que, a pesar de que los frames enviados por la interfaz son de 8 bits, de estos uno solo corresponde a información útil (el valor del pixel normalizado) mientras que los bits restantes del frame se descartan. Este aspecto es un evidente espacio de mejora para versiones futuras del diseño.

3.3.4. Sistema completo

El sistema diseñado utiliza los módulos descritos en secciones anteriores, la Figura 3.13 muestra el diagrama de bloques que describe de manera gráfica las conexiones y componentes necesarios para que el diseño funcione correctamente. El sistema opera de manera sincrónica, por lo que todos los módulos que lo componen reciben como entrada la señal de reloj y la señal reset, sin embargo, estas se omitieron en el diagrama para facilitar la comprensión.

En el diagrama se encuentran los bloques **SPLIT** y **CMP**, estos módulos no están diseñados como tal, sino que se agregaron al diagrama para representar la extracción del bit menos significativo de la señal `RX_DATA` proveniente del módulo de comunicación, y para resumir comparaciones

entre señales respectivamente.

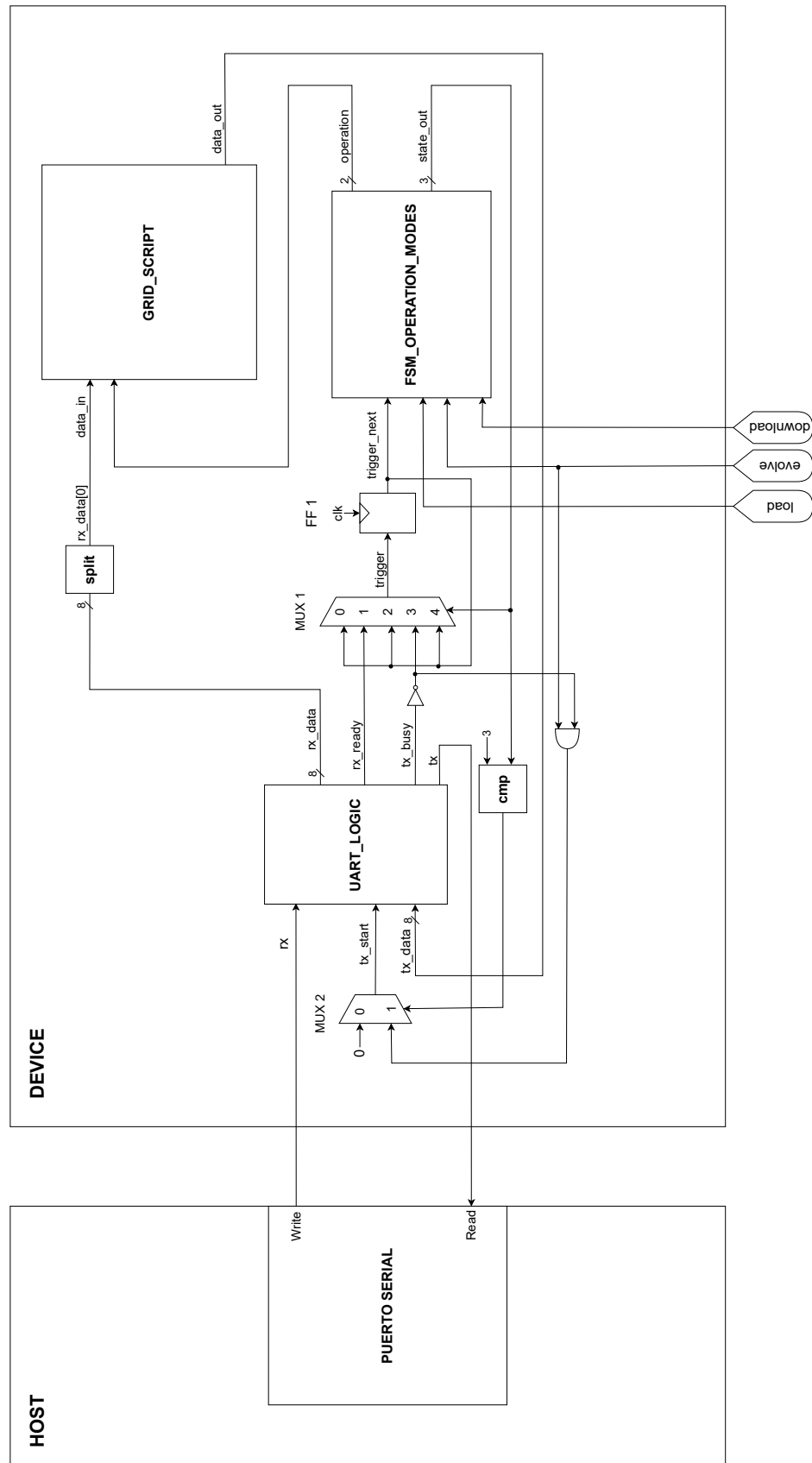


Figura 3.13: Diagrama de bloques del sistema diseñado.

4. Resultados

En este capítulo, se mostrarán y compararán los resultados obtenidos del procesamiento de imágenes utilizando el diseño en software y en hardware de un autómata celular. Además, se verificará la calidad de estos resultados en contraste con los resultados que se obtienen para las mismas tareas de procesamiento a través de algoritmos estándar, en particular se compararán con los métodos y `erode()` que provee la biblioteca `OpenCV`. También se toma en cuenta como métrica relevante el tiempo de ejecución que requiere cada una de las alternativas. Se discutirán los hallazgos relevantes, incluyendo la calidad de la imagen, el tiempo de procesamiento y la eficiencia del algoritmo. Se mostrarán los resultados de manera gráfica para una mejor comprensión.

En primer lugar, se presentarán los resultados del procesamiento de imágenes utilizando el diseño en software del *CA* junto a los que se obtienen para la misma tarea mediante los métodos de la biblioteca `OpenCV`. Ambos resultados se compararán con la imagen base binaria y entre sí. Luego, se hará lo mismo para el caso del diseño en hardware. Se destacarán las ventajas y desventajas de cada alternativa. Se compararán también los resultados de ambas implementaciones, el diseño en software y el diseño en hardware entre sí.

Se seleccionaron tres métricas comunes de comparación para el procesamiento de imágenes que se adaptan bien al contexto de imágenes binarias, estas son: *Error Cuadrático Medio (RMSE)*, *Error Medio Absoluto (MAE)* y *Distancia de Hamming (HD)*. Para el uso de estas métricas se consideran las siguientes definiciones:

Error cuadrático medio

$$R.M.S.E = \sqrt{\frac{1}{M \cdot N} \sum_{i=1}^M \sum_{j=1}^N (X_{1(i,j)} - X_{2(i,j)})^2} \quad (4.5)$$

Error medio absoluto

$$M.A.E = \frac{1}{M \cdot N} \sum_{i=1}^M \sum_{j=1}^N |X_{1(i,j)} - X_{2(i,j)}| \quad (4.6)$$

Distancia de Hamming

$$H.D = \frac{1}{M \cdot N} \sum_{i=1}^M \sum_{j=1}^N X_{1(i,j)} \oplus X_{2(i,j)} \quad (4.7)$$

Se escogieron estas métricas, ya que tanto el *RMSE* como el *MAE* proporcionan una medida cuantitativa de la calidad de los resultados en función de los errores o diferencias que presentan en cuanto a valor de píxeles, por lo que estas métricas se encuentran en el rango $[0,255]$, ya que son los valores mínimos y máximo posible en los píxeles con los que se trabaja. Por otro lado, la *HD* mide la similitud entre dos imágenes binarias en términos de la cantidad de píxeles diferentes entre ellas, por lo que esta métrica entrega resultados entre 0 y 1, el cual es interpretable como un porcentaje. En cuanto a *RMSE* y *MAE*, se espera que los resultados de implementación se acerquen lo más posible a las correspondientes a los métodos de la biblioteca *OpenCV*, ya que estas técnicas se encuentran dentro del estado del arte del procesamiento de imágenes, mientras que en cuanto a *HD*, lo ideal es la obtención del menor valor posible, dado que esto sugiere que el resultado es más cercano al que se obtiene por técnicas convencionales.

Para realizar el análisis de resultados que se obtienen a través de las versiones del *CA* diseñado, se utilizan como imágenes de *benchmark* las que se muestran en las Figuras 4.14a, 4.14b y 4.14c, que serán referidas de aquí en más como *Imagen Leopardo*, *Imagen Círculos* e *Imagen Formas* respectivamente.

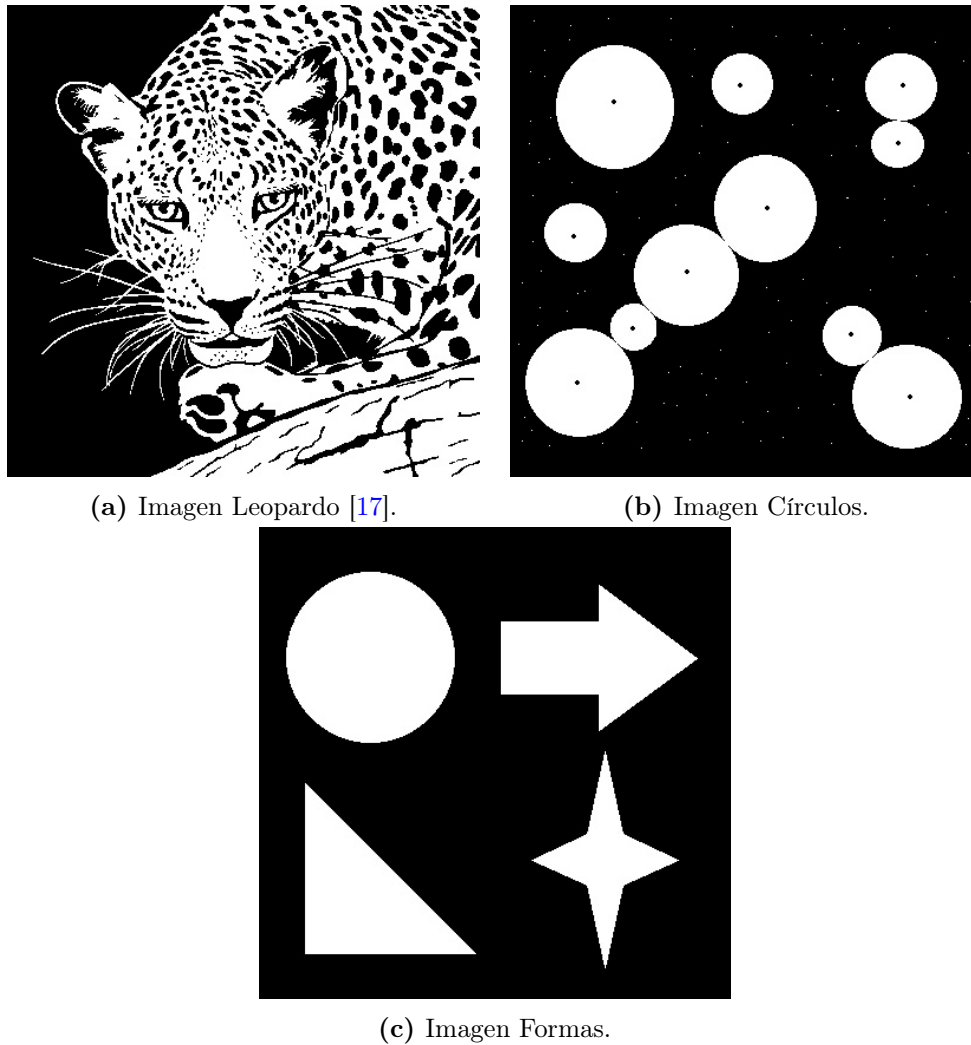


Figura 4.14: Imágenes utilizadas para obtención y análisis de resultados.

4.1. Resultados de la implementación del CA en software

El objetivo principal de la evaluación de los resultados del diseño mediante software, es validar la factibilidad del uso de las reglas de evolución escogidas para el desarrollo de tareas de procesamiento de imágenes, es por ello que en esta sección se comparan los resultados obtenidos del procesamiento de imagen con el CA diseñado en software con respecto a la imagen base, esta última también se compara con el resultado obtenido a través de los métodos `erode()` con un kernel 3x3 e incluidos en la biblioteca OpenCV.

Erosion

Al aplicar erosión de la Figura 4.15a, mediante el método `erode()`, perteneciente a la biblioteca `OpenCV`, con un kernel de 3×3 y con el *CA* diseñado en software, los resultados de este proceso se muestran en la Figura 4.15b y 4.15c respectivamente. La Tabla 4.1 resume los resultados obtenidos en el procesamiento de esta imagen para cada una de las métricas propuestas.

En este caso, se observa como los patrones de la piel del leopardo y las secciones oscuras de la imagen en general, se han engrosado, mientras que detalles finos de la imagen, como los bigotes del animal, se eliminan casi por completo. De los resultados de las métricas, se puede destacar que la comparación entre las imágenes resultantes del procesamiento con el método `erode()` y por el *CA* arroja cero para cada una de ellas, es decir que la regla de evolución propuesta genera un resultado idéntico al método convencional de erosión.

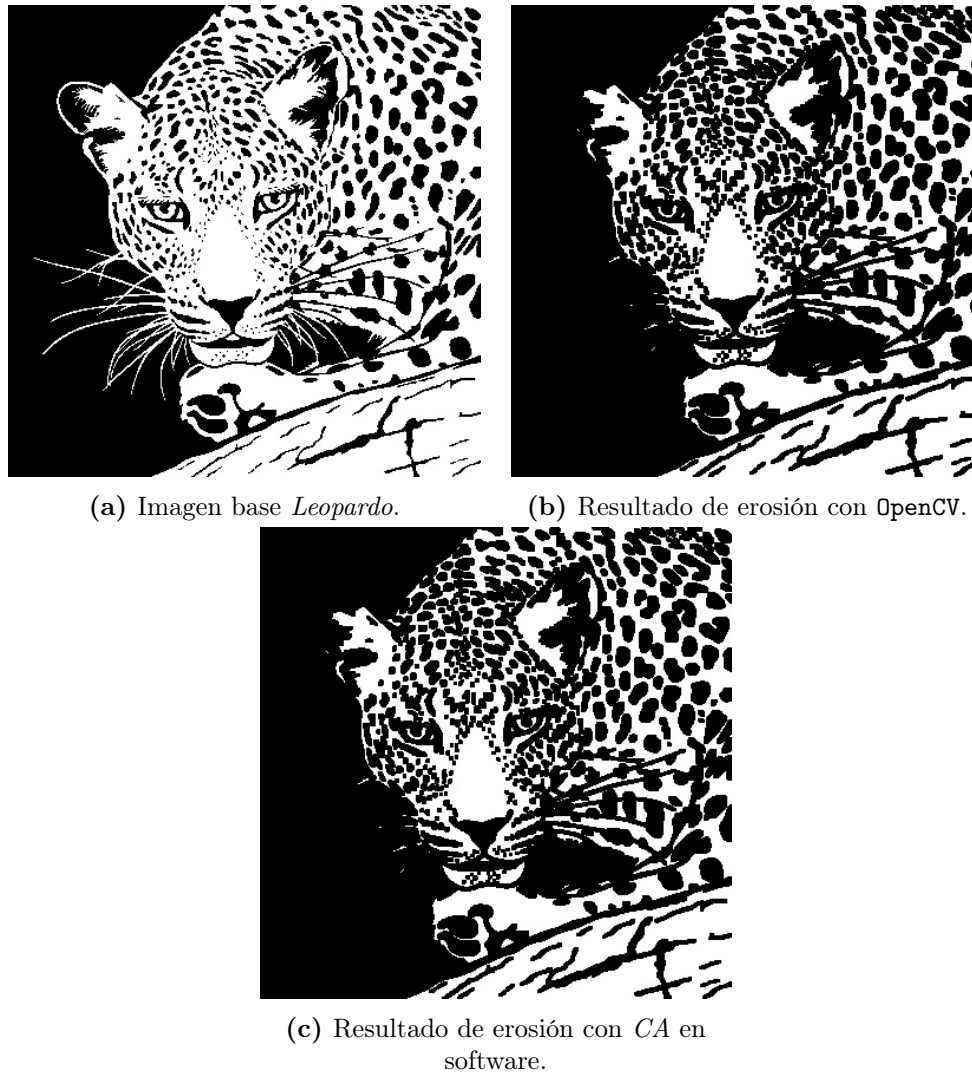


Figura 4.15: Resultados de erosión de la Imagen *Leopardo* para el diseño en software.

Tabla 4.1: Métricas de comparación para la erosión de la Imagen *Leopardo*.

Comparación	RMSE	MAE	HD
CA en software vs Base	0.1619	0.1619	0.1619
OpenCV vs Base	0.1619	0.1619	0.1619
CA en software vs OpenCV	0	0	0

Al aplicar erosión a la Figura 4.16a, en las mismas condiciones que en el caso anterior, a través del método `erode` y mediante *CA*. Las Figuras 4.16b y 4.16c muestran la imagen resultante

para cada caso. Al igual que en el caso anterior, se destaca la equivalencia exacta entre ambos resultados. Además, esta vez se expone otra de las características de la técnica de erosión, que es la eliminación de detalles no deseados, como lo son en este caso los pequeños píxeles blancos presentes en la imagen base.

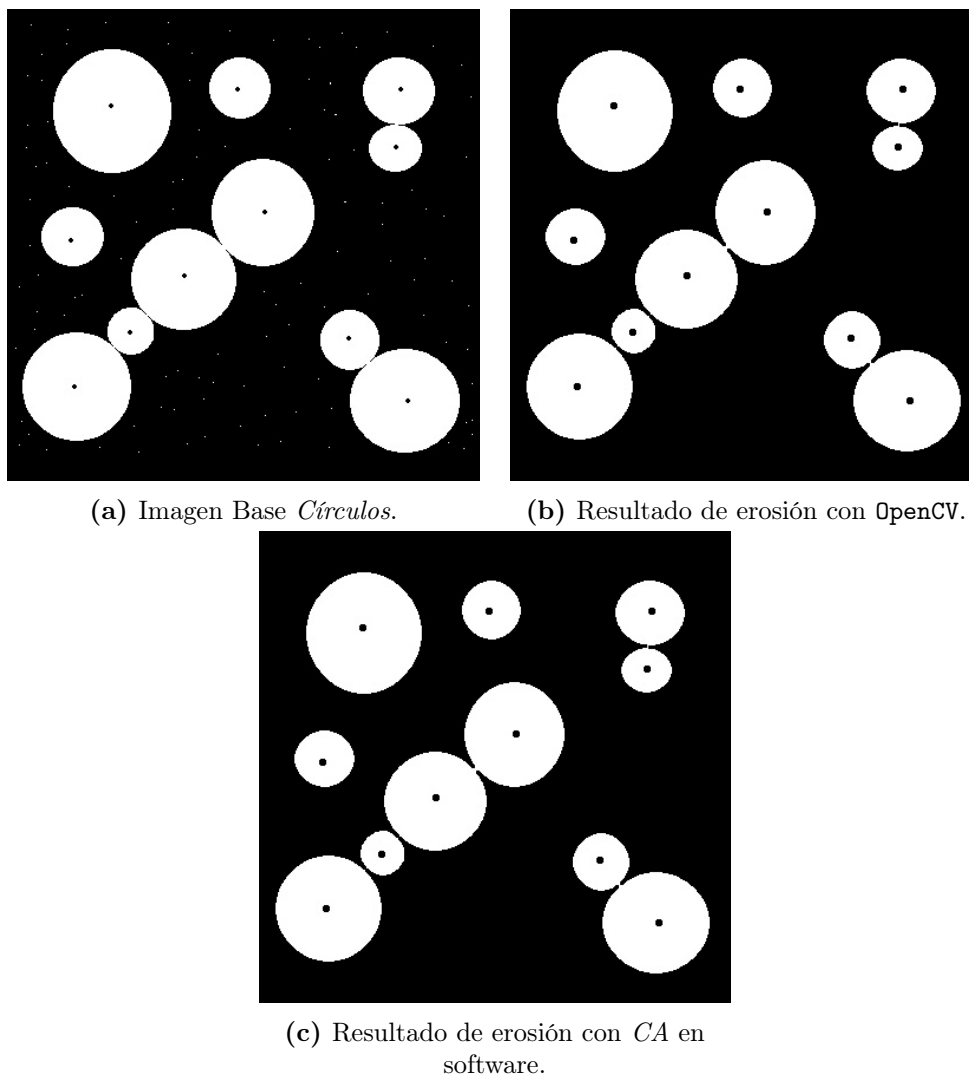


Figura 4.16: Resultados de erosión de la Imagen *Círculos* para el diseño en software.

Tabla 4.2: Métricas de comparación para la erosión de la Imagen *Círculos*.

Comparación	RMSE	MAE	HD
CA en software vs Base	0.14348	0.02058	0.02058
OpenCV vs Base	0.14348	0.02058	0.02058
CA en software vs OpenCV	0	0	0

Detección de bordes

En cuanto a la detección de bordes, al aplicar esta técnica a la Imagen *Leopardo*, si bien, se ve en la Figura 4.17 que mediante el método **Canny()** y el *CA* se detectan prácticamente los mismos bordes, los detectados por el primero de estos se perciben ligeramente más notorios y gruesos. Esto se apoya con los resultados de las métricas de comparación en donde si bien, los valores obtenidos para el *CA* no se alejan más allá de un 9 % en cuanto a diferencia de píxeles. Por su parte, las métricas *RMSE* y *MAE* sí muestran resultados más alejados con respecto a la imagen base. Estos valores se encuentran en la Tabla 4.3

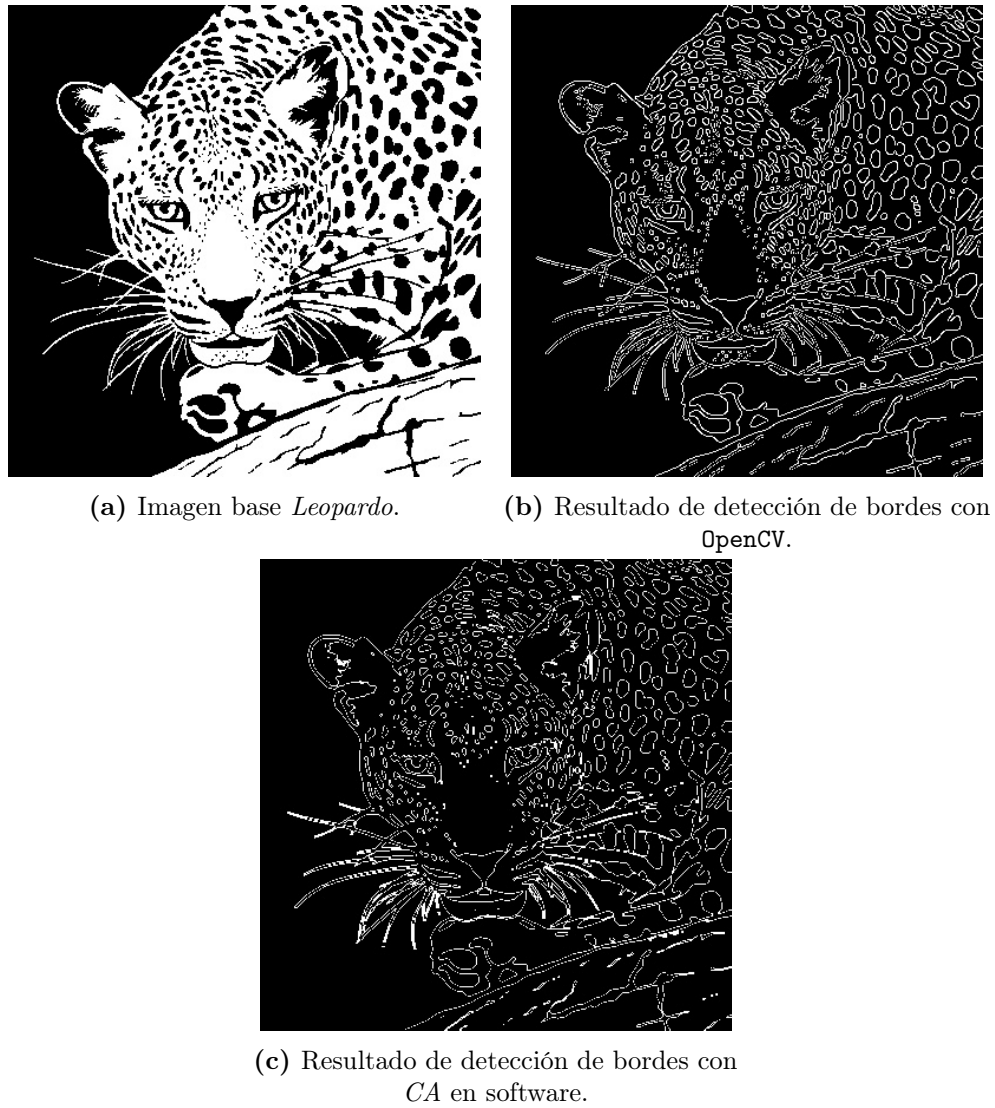


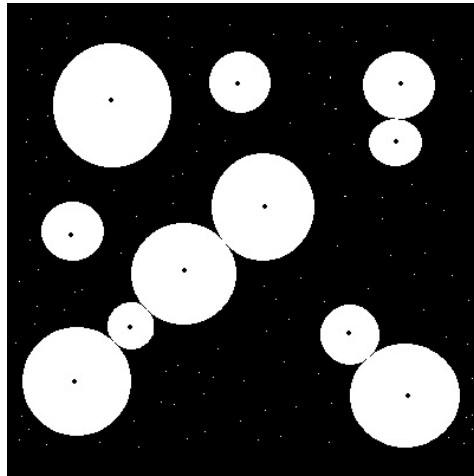
Figura 4.17: Resultados de detección de bordes en imagen *Leopardo*.

Tabla 4.3: Métricas de comparación para la detección de bordes en imagen *Leopardo*.

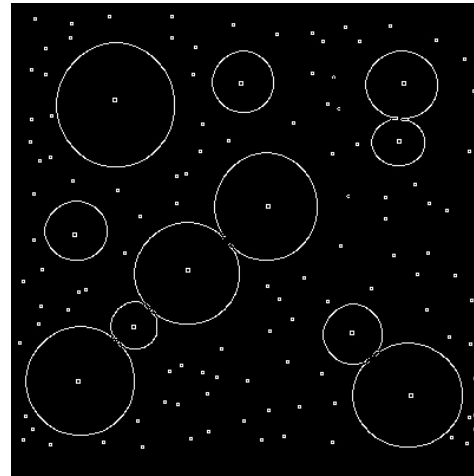
Comparación	RMSE	MAE	HD
CA en software vs Base	0.7452	19.4562	0.5554
OpenCV vs Base	0.70380	15.7242	0.4953
CA en software vs OpenCV	0.2984	7.4391	0.0890

La Figura 4.18 muestra los resultados obtenidos al aplicar la detección de bordes a la Imagen

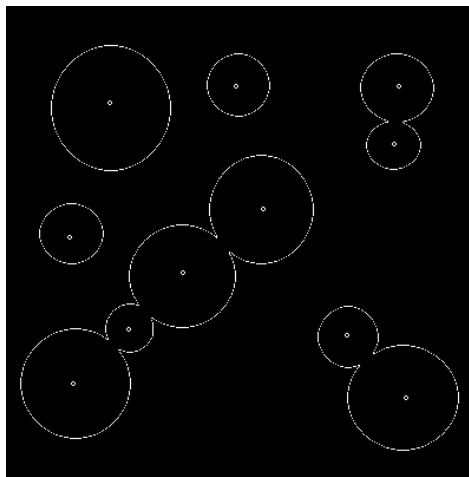
Círculos, a través del método *Canny()* y el *CA*, se puede ver que nuevamente los bordes de *Canny* se perciben más gruesos. También destaca la detección de los puntos que simulan ruido en la imagen, que son ignorados en el procesamiento a través del *CA*. El considerar estos artefactos efectivamente como bordes, o bien, como falso positivo, depende netamente del contexto de aplicación de la técnica de procesamiento.



(a) Imagen base *Círculos*.



(b) Resultado de detección de bordes con **OpenCV**.



(c) Resultado de detección de bordes con *CA* en software.

Figura 4.18: Resultados de detección de bordes en imagen *Círculos*.

Tabla 4.4: Métricas de comparación para la detección de bordes en imagen *Círculos*.

Comparación	RMSE	MAE	HD
CA en software vs Base	0.5389	2.7971	0.2904375
OpenCV vs Base	0.5342	3.6017	0.2854
CA en software vs OpenCV	0.1258	0.5857	0.0158

4.2. Resultados del diseño en hardware

En esta sección, se comparan los resultados obtenidos del procesamiento de imagen con el CA diseñado en hardware con respecto la Imagen Base, esta última también se compara con el resultado obtenido a través de los métodos `erode()` con un kernel 3x3 y `canny()` incluidos en la biblioteca OpenCV.

Se implementó el diseño del CA descrito en la sección 3.3 en la tarjeta de desarrollo Digilent Nexys4 DDR, basada en el chip Artix-7™ de Xilinx®, esta se muestra en la Figura 4.19.

La cantidad de recursos físicos de la tarjeta utilizada, de los cuales los más relevantes para esta aplicación se muestran en la Tabla 4.5, significó una limitante para el tamaño máximo de malla que se logró implementar para cada una de las reglas de evolución escogidas. En el caso de la detección de bordes se logró una malla de tamaño 107x107 celdas, mientras que para la erosión se logró un tamaño de 125x125 celdas. Las tablas 4.10 y 4.6 muestran el uso de recursos de la tarjeta que se requirieron para cada implementación.

Tabla 4.5: Recursos físicos disponibles en la tarjeta de desarrollo Nexys 4 DDR [1].

Tarjeta	LUT	Registros	Logic Slices
Nexys 4 DDR	63400	126800	15850

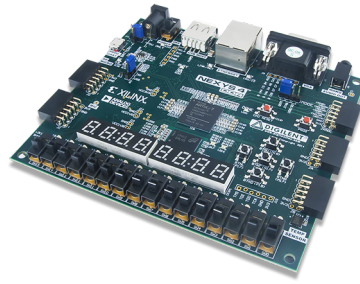


Figura 4.19: Tarjeta de desarrollo Nexys 4 DDR.

Debido al limitado tamaño del *CA* implementado, las imágenes con las que se realizaron las pruebas debieron ser redimensionadas para que se ajusten a la capacidad del modelo. A continuación se presentan los resultados obtenidos para cada una de las técnicas de procesamiento implementadas, erosión y detección de bordes, cuyos resultados fueron comparados de la misma manera que en la sección 4.1, es decir, se comparan los resultados obtenidos del procesamiento de imagen con el *CA* con respecto la imagen Base, esta última también se compara con el resultado obtenido a través de los métodos `erode()` con un kernel 3x3 e incluidos en la biblioteca OpenCV. Además, se hace una comparación entre los resultados de ambas técnicas y se miden los tiempos de ejecución de cada uno de los procesamientos.

Erosión

Al implementar la regla evolución para aplicar erosión de una imagen, se logró un tamaño máximo de malla cuadrada de *CA* de 125x125 celdas o pixeles, la Tabla 4.6 muestra la utilización de recursos de la tarjeta Nexys4 DDR que se requirieron para lograr dicha malla, en desglose por módulos mientras que la Tabla 4.7 muestra la utilización de recursos respecto a los disponibles en la tarjeta. Entre los tipos de recursos más utilizados se ven las *LUT* y las *Logic Slices*. Se utilizó cerca de un 86 % de los *LUT* disponibles en la tarjeta, mientras que la principal limitante para las dimensiones logradas fueron las *Logic Slices*, viéndose utilizadas casi la totalidad de las disponibles, llegando a un 99.66 %

Si bien, la imagen de la Figura 4.20a al sufrir el redimensionamiento para adecuarse al tamaño de la malla del *CA*, se ve altamente distorsionada, esto se puede pasar por alto para llevar el foco únicamente a los resultados del procesamiento y métricas aplicadas. Las Figuras 4.20b y

Tabla 4.6: Utilización de recursos para implementación de erosión en una malla de 125x125 celdas.

Módulo	LUT	Registros	Logic Slices
GRID_SCRIPT	23394	15625	15757
FSM_OPERATION_MODES	30912	25	15687
UART_LOGIC	29	67	23
BOTONES	37	30	17
TOTAL	54371	15763	15796

Tabla 4.7: Utilización de recursos para erosión respecto al total disponible en la tarjeta.

Tipo de recurso	Utilizado	Disponible	% Utilizado
LUT	54371	63499	85.76 %
Registros	15763	126800	12.43 %
Logic Slices	15796	15850	99.66 %

4.20c muestran las imágenes resultantes mediante el método `erode()` de `OpenCV` y el mediante la evolución del *CA* implementado en hardware respectivamente. Se puede apreciar cualitativamente que las imágenes obtenidas son similares. Esto se condice con lo que se desprende de las métricas de comparación utilizadas, *RMSE*, *MAE* y *HD*, las que se presentan en la Tabla 4.8, donde nuevamente se destaca la coincidencia exacta pixel a pixel entre ambos procesamientos.

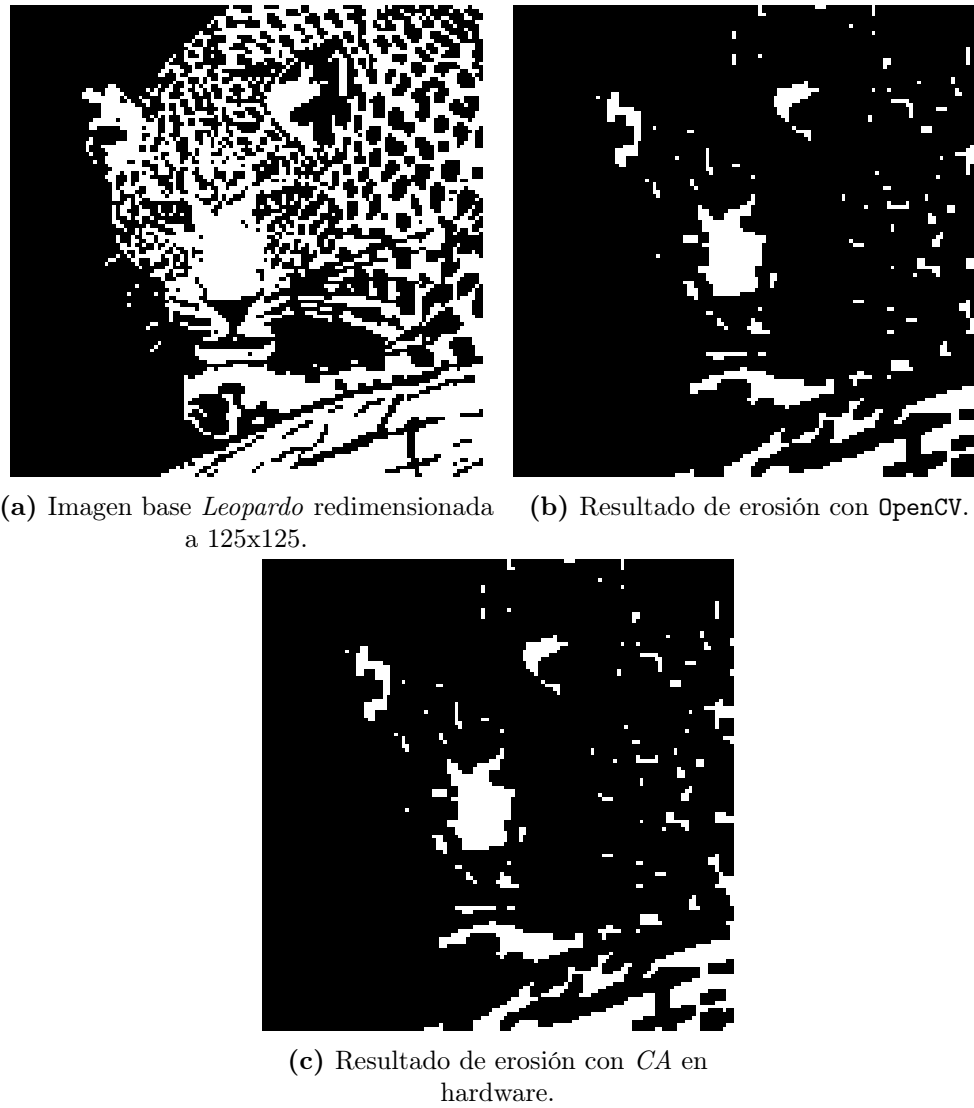


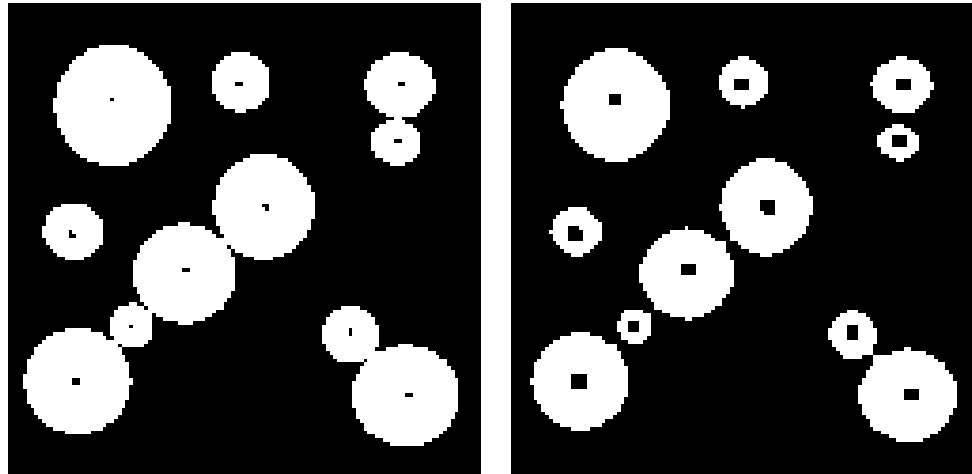
Figura 4.20: Resultados de erosión de imagen *Círculos* con el *CA* implementado en hardware.

Tabla 4.8: Métricas de comparación para la erosión de imagen *Leopardo*.

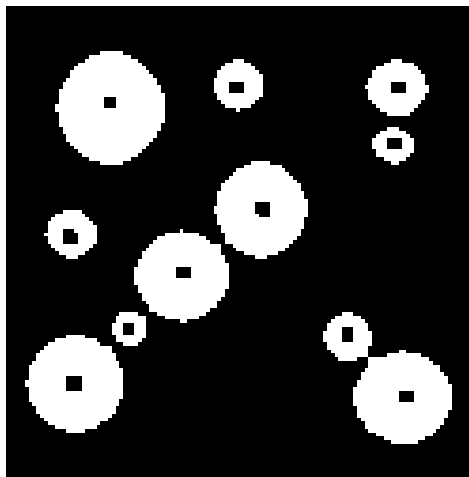
Comparación	RMSE	MAE	HD
CA en hardware vs Base	0.53041	0.28134	0.28134
OpenCV vs Base	0.53041	0.28134	0.28134
CA en hardware vs OpenCV	0	0	0

Para procesamiento de la Imagen *Círculos*, igual que en el caso anterior, al haber redimen-

sionado la imagen base de la Figura 4.21a, esta ha perdido algunos detalles. Sin embargo, concentrándose en los resultados de procesamiento, nuevamente se comprueba que la regla de evolución propuesta para el *CA* otorga exactamente los mismos resultados que el método `erode()` de la biblioteca `OpenCV`. Esto se observa en las Figuras 4.21b y 4.21c. Además, las métricas de comparación se muestran en la Tabla 4.9



(a) Imagen base *Círculos* redimensionada a 125x125. (b) Resultado de erosión con `OpenCV`.



(c) Resultado de erosión con *CA* en hardware.

Figura 4.21: Resultados de erosión de imagen *Círculos*.

Tabla 4.9: Métricas de comparación para la erosión de imagen *Círculos*.

Comparación	RMSE	MAE	HD
CA en hardware vs Base	0.25069	0.06284	0.06284
OpenCV vs Base	0.25069	0.06284	0.06284
CA en hardware vs OpenCV	0	0	0

Detección de bordes

Al implementar la regla evolución para aplicar detección de bordes de una imagen, se logró un tamaño máximo de malla cuadrada de *CA* de 107x107 celdas o pixeles, la Tabla 4.10 muestra la utilización de recursos de la tarjeta Nexys4 DDR que se requirieron para lograr dicha malla, desglosado por módulos. Al igual que en el caso de la erosión, los tipos de recursos más utilizados se ven las LUT y las Logic Slices. La Tabla 4.11 muestra la utilización de recursos en comparación con el total disponible en la tarjeta. Esta vez ambos recursos se encuentran usados prácticamente en su totalidad, llegando a ser usados 97.4 % y 99.95 %. Cabe señalar que, pese a que las dimensiones logradas en este caso, son menores a las conseguidas en la regla de erosión, esta vez se utilicen más recursos, y esto se debe a que la regla de detección de bordes es más compleja en cuanto a lógica requerida, ya que debe hacer dos comparaciones por cada celda para verificar que la suma de los estados de la vecindad sea 3 o 4.

Tabla 4.10: Utilización de recursos para implementación de detección de bordes en una malla de 107x107 celdas.

Módulo	LUT	Registros	Logic Slices
GRID_SCRIPT	44530	11449	15817
FSM_OPERATION_MODES	17161	96	12075
UART_LOGIC	29	67	25
BOTONES	36	45	20
Total	61757	11659	15842

Al aplicar detección de bordes en la imagen de la Figura 4.22a, se obtienen las imágenes que se muestran en las Figuras 4.22b y 4.22c correspondientes al método *Canny()* y al *CA*

Tabla 4.11: Utilización de recursos para detección de bordes respecto al total disponible en la tarjeta.

Tipo de recurso	Utilizado	Disponible	% Utilizado
LUT	61757	63499	97.4 %
Registros	11659	126800	9.19 %
Logic Slices	15842	15850	99.95 %

implementado en hardware respectivamente. Se puede observar que existen elementos como los píxeles negros al interior de los círculos que se consideran bordes con `Canny()`, pero que el *CA* no considera como tal, dado que la regla de evolución propuesta para este fin requiere de la existencia de al mínimo 3 y máximo 4 celdas/píxeles en blanco dentro de la vecindad para que la celda central se considere borde. Este resultado se ve directamente influenciado por las dimensiones de la imagen, pues al hacer el redimensionamiento de la imagen base, los elementos negros dentro de los círculos e representan únicamente 1 o 2 píxeles quedando fuera del criterio de la regla de evolución. Esta observación será relevante y retomada en la sección 5.

Otro aspecto relevante de este resultado es la notoriedad y continuidad de los bordes detectados mediante `Canny()`, en contraste con los que fueron detectados por el *CA*, en donde, si bien, se pueden diferenciar claramente la existencia de contornos, estos presentan discontinuidades a lo largo de los trazos.

En cuanto a las métricas de comparación, que se resumen en la Tabla 4.12, a pesar de lo mencionado anteriormente, las métricas arrojan que los resultados difieren en no más un 5 % de los píxeles (según *HD*).

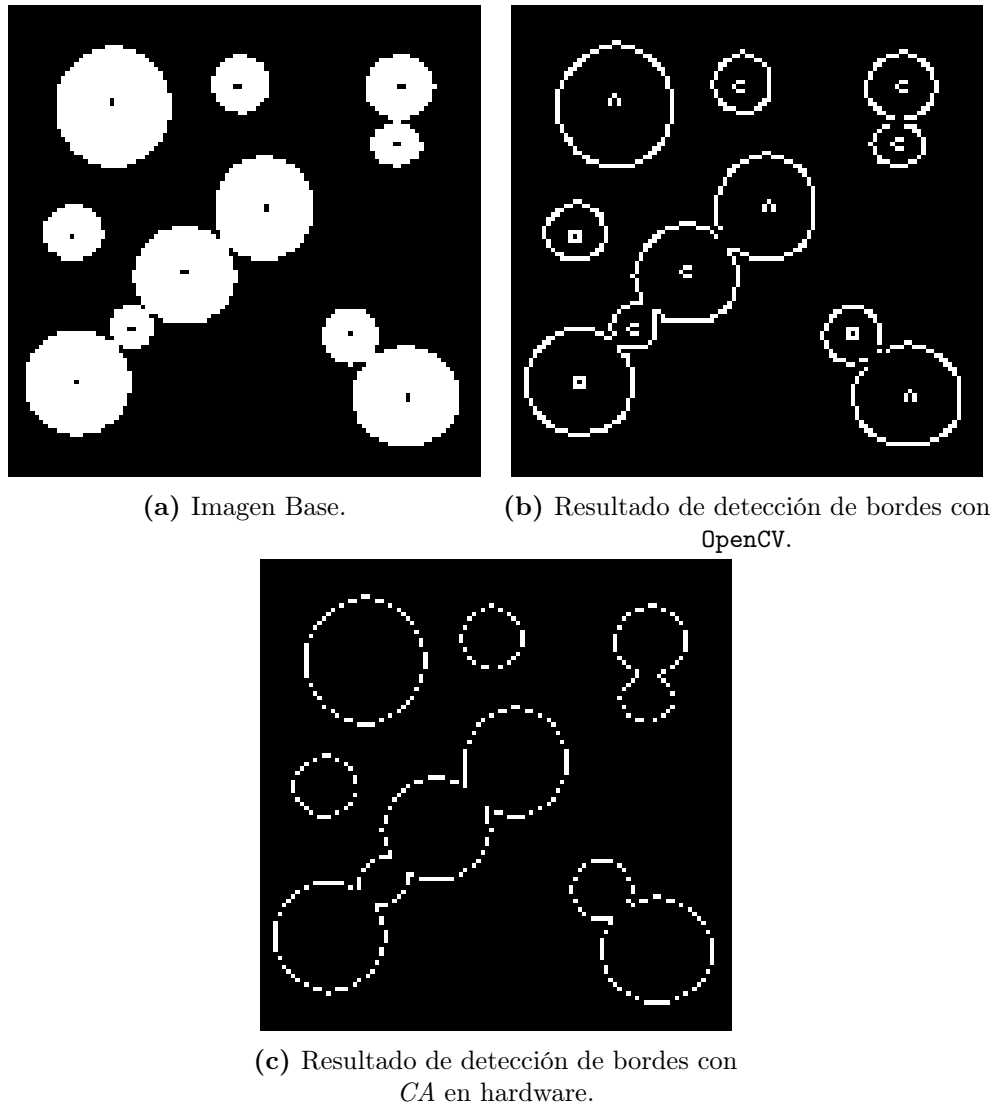


Figura 4.22: Resultados de detección de bordes de imagen *Círculos*.

Tabla 4.12: Métricas de comparación para la detección de bordes en imagen *Círculos*.

Comparación	RMSE	MAE	HD
CA en hardware vs Base	0.54205	8.19180	0.29382
OpenCV vs Base	0.50389	6.33269	0.25390
CA en hardware vs OpenCV	0.20196	1.97091	0.04078

Al aplicar la detección de bordes en la Figura 4.23a, se obtienen las imágenes que se muestran

en las Figuras 4.23b y 4.23c correspondientes al método `Canny()` y al *CA* implementado en hardware respectivamente. De estas imágenes se puede notar nuevamente que, si bien el *CA* logra detectar bordes de forma consistente con lo que detecta el método `Canny()` los obtenidos por el *CA* presentan discontinuidad en los trazos. Además, con esta imagen se puede notar que los bordes asociados a contornos rectos de los elementos de la imagen base, ya sea horizontal o verticales, se detectan sin problemas, es decir, logran resultar en un trazo continuo, sin embargo, cuando se trata de bordes curvos o esquinas, se observan las discontinuidades mencionadas. Pese a esto, según las métricas aplicadas, el procesamiento con `Canny()` y el procesamiento con *CA* difieren únicamente en un 3 % de los píxeles.

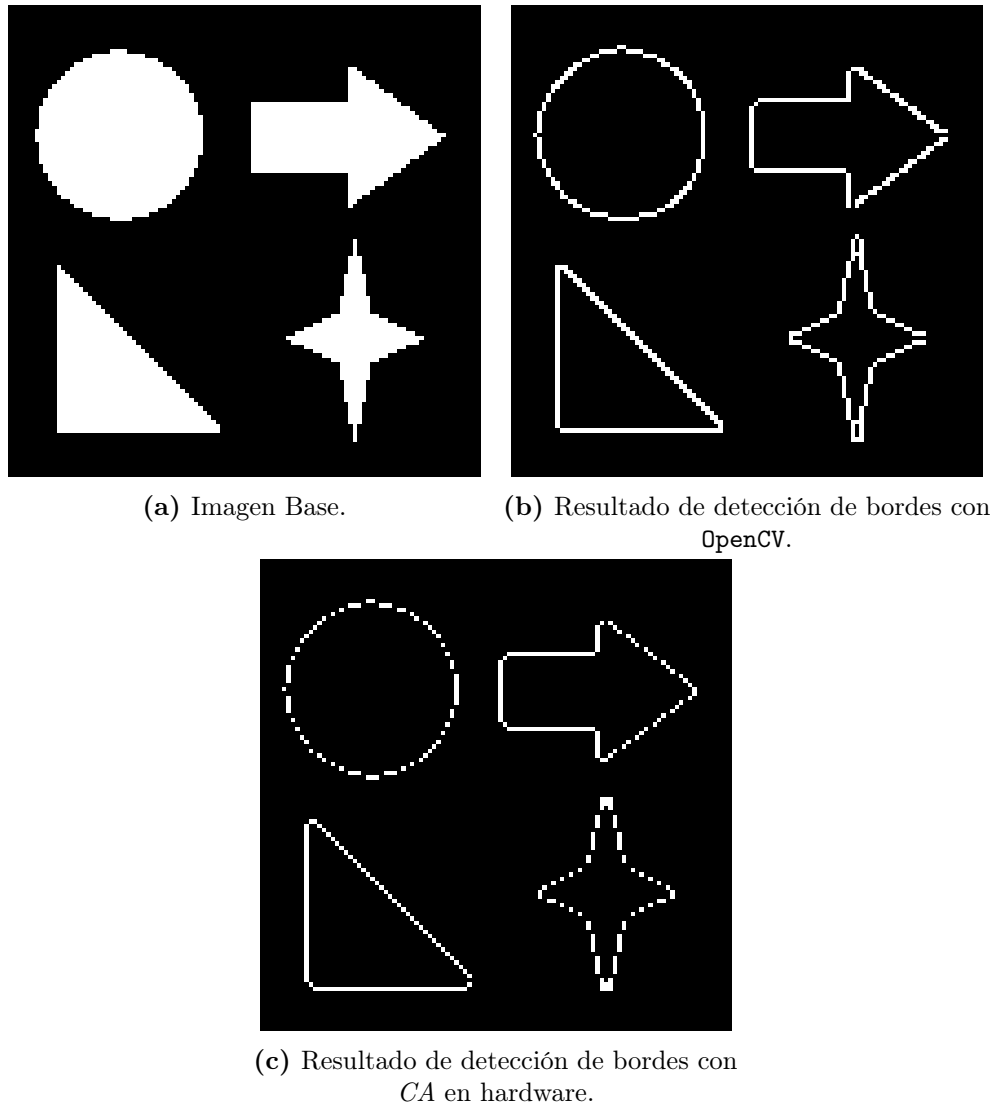


Figura 4.23: Resultados de detección de bordes en imagen *Formas*.

Tabla 4.13: Métricas de comparación para la detección de bordes en imagen *Formas*.

Comparación	RMSE	MAE	HD
CA en hardware vs Base	0.53557	8.07389	0.28683
OpenCV vs Base	0.51026	6.45008	0.26037
CA en hardware vs OpenCV	0.17459	2.13809	0.03048

4.3. Contraste de resultados entre software y hardware

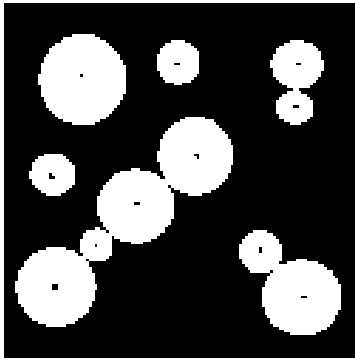
Además de comparar los resultados del procesamiento de imágenes obtenido a través del *CA* diseñado con respecto a técnicas del estado del arte como el algoritmo de Canny o el método de erosión disponibles en la biblioteca **OpenCV**, también resulta interesante comparar los resultados del *CA* en sus dos versiones, software y hardware, de forma que se pueda comprobar la validez del diseño en hardware.

Para esto se define en software una malla del mismo tamaño que la máxima implementable mediante hardware para cada regla y se comparan uno a uno los píxeles resultantes del procesamiento por software con los obtenidos mediante hardware.

Erosión

Para el caso de la erosión, se utiliza una imagen de dimensiones 125x125 para inicializar la malla en software, dado que fue el tamaño máximo que se logró implementar en la tarjeta Nexys4 DDR.

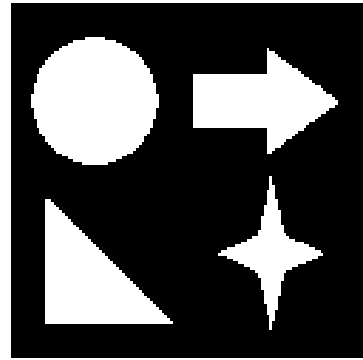
Se utilizaron las imágenes de *Círculos*, *Leopardo* y *Formas* para realizar la comparación. Los resultados obtenidos se muestran en la Figura 4.24, mientras que la tabla 4.14 expone la diferencia en términos de cantidad de píxeles diferidos entre ambos resultados, junto al porcentaje equivalente de coincidencia entre los píxeles de ambas imágenes. Se obtienen un 100 % de coincidencia entre la versión de software y hardware para las tres imágenes, lo que es consistente como se mostró en secciones anteriores, donde para la métrica de *HD* tanto software como hardware obtuvieron 0 como resultado en comparación a la referencia, el método `erode()` de la biblioteca **OpenCV**.



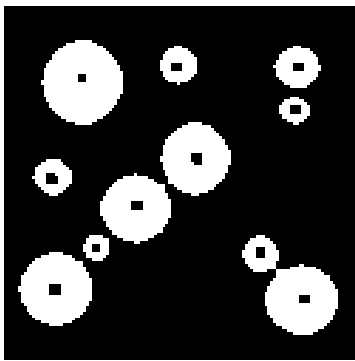
(a) Imagen base Círculos de dimensiones 125x125.



(b) Imagen base Leopardo de dimensiones 125x125.



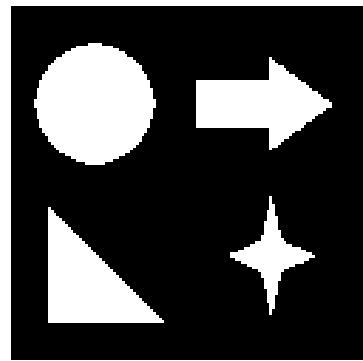
(c) Imagen base Formas de dimensiones 125x125.



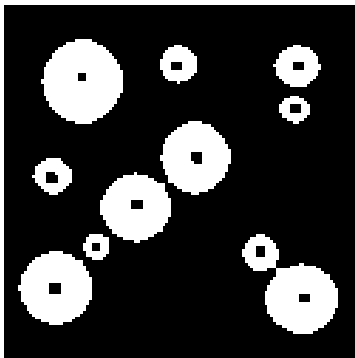
(d) Resultado mediante software de imagen Círculos.



(e) Resultado mediante software de imagen Leopardo.



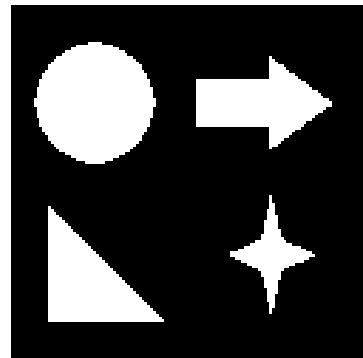
(f) Resultado mediante software de imagen Formas.



(g) Resultado mediante hardware de imagen Círculos.



(h) Resultado mediante hardware de imagen Leopardo.



(i) Resultado mediante hardware de imagen Formas.

Figura 4.24: Resultados de erosión mediante *CA* en su versión en software y en hardware.

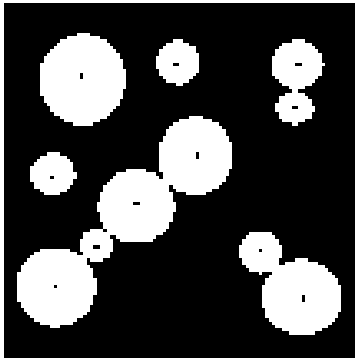
Tabla 4.14: Contraste de resultados de la aplicación de erosión a través de su implementación software vs hardware.

Imagen	Píxeles Distintos	% de coincidencia
Círculos	0	100 %
Leopardo	0	100 %
Formas	0	100 %

Detección de bordes

Para el caso de la detección de bordes, se utiliza una imagen de dimensiones 107x107 para inicializar la malla en software, dado que fue el tamaño máximo que se logró implementar en la tarjeta Nexys4 DDR.

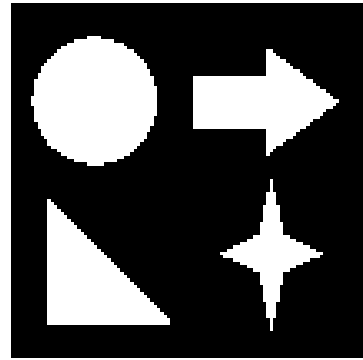
Se usaron las mismas imágenes que en el caso de erosión para realizar la comparación. Los resultados se muestran en la Figura 4.25, mientras que la tabla 4.15 expone la diferencia en términos de cantidad de píxeles diferidos el porcentaje equivalente con respecto al total de píxeles de la imagen. Al igual que en el caso de la erosión, en las tres imágenes se encontró una completa coincidencia entre el resultado obtenido a través de software con el obtenido mediante hardware.



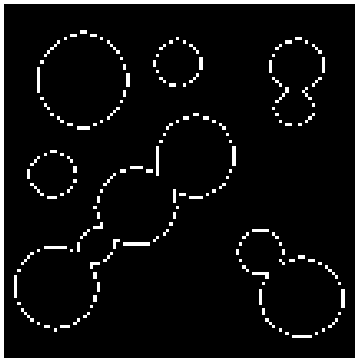
(a) Imagen base Círculos de dimensiones 107x107.



(b) Imagen base Leopardo de dimensiones 107x107.



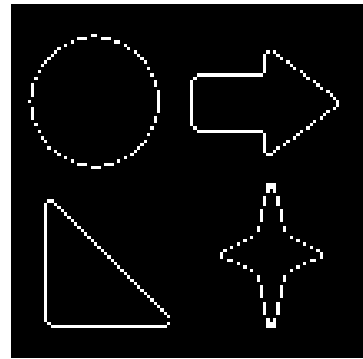
(c) Imagen base Formas de dimensiones 107x107.



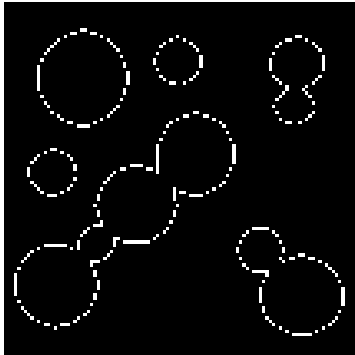
(d) Resultado mediante software de imagen Círculos.



(e) Resultado mediante software de imagen Leopardo.



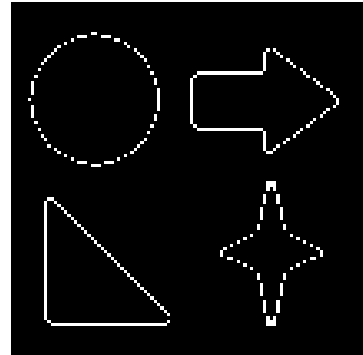
(f) Resultado mediante software de imagen Formas.



(g) Resultado mediante hardware de imagen Círculos.



(h) Resultado mediante hardware de imagen Leopardo.



(i) Resultado mediante hardware de imagen Formas.

Figura 4.25: Resultados de detección de bordes mediante *CA* en su versión en software y en hardware.

Tabla 4.15: Contraste de resultados de la aplicación de detección de bordes a través de su implementación software vs hardware.

Imagen	Píxeles Distintos	% de coincidencia
Círculos	0	100 %
Leopardo	0	100 %
Formas	0	100 %

Dado que los porcentajes de coincidencias entre los resultados obtenidos por la implementación del *CA* en software y su implementación en hardware corresponden al 100 % tanto para erosión como para detección de bordes en cada una de las imágenes, resulta factible usar la versión en software como un método de estimación de los resultados que se pueden obtener mediante hardware para mallas de *CA* de distintas dimensiones, incluso para la generación de prototipos previos antes de realizar la implementación en una tarjeta física, o para estimar el desempeño del *CA* en cuanto a resultados antes de adquirir una tarjeta con recursos que soporten el diseño.

4.4. Tiempos de ejecución

Uno de los principales objetivos de este trabajo es explorar la capacidad de los *CA* de otorgar aceleración a tareas de procesamiento al ser implementados en hardware, donde se pueden explotar realmente sus propiedades de cómputo masivamente paralelo, a diferencia de cuando se implementan en software. En esta sección se analizan los resultados obtenidos, en cuanto a tiempo de ejecución de los métodos propuestos para las tareas de procesamiento de imágenes que fueron implementadas, se comparan los resultados la implantación tanto en software como en hardware del *CA*, y además el tiempo que requieren estas tareas a través de la biblioteca *OpenCV*.

Implementación en software y *OpenCV*

Tanto la implementación del *CA* en software como el uso de la biblioteca *OpenCV* se desarrollan en el lenguaje de programación Python, y ambos se ejecutan en el mismo dispositivo, por lo que los tiempos de ejecución para ambos casos se miden con respecto a la capacidad de cómputo

Tabla 4.16: Resultados de tiempos de ejecución de detección de bordes por el *CA* en software y para el método *Canny*.

Método	Tiempo de procesamiento	Tiempo de carga	Tiempo de procesamiento y carga
Detección de bordes con OpenCV	0.00099642 s	0.00171725 s	0.00271367 s
Detección de bordes con <i>CA</i> software	0.08957690 s	0.00171725 s	0.09129415 s

Tabla 4.17: Resultados de tiempos de ejecución de erosión por el *CA* en software y para el metodo *Erode()*.

Método	Tiempo de procesamiento	Tiempo de carga	Tiempo de procesamiento y carga
Erosión con OpenCV	0.00098580 s	0.00171725 s	0.00270305 s
Erosión con <i>CA</i> software	0.08161981 s	0.00171725 s	0.08333706 s

soportada por el sistema utilizado, en este caso un Notebook con un procesador Intel(R) Core(TM) i5-9300H CPU @ 2.40 GHz y 12GB de memoria RAM.

Para obtener una muestra representativa del tiempo empleado en cada caso para realizar el procesamiento, se repitió el proceso de carga y procesamiento 100.000 veces para cada regla de evolución, tomando dos mediciones distintas en cada ejecución, el tiempo que se tarda en cargar la imagen al código y el tiempo que se tarda el código en ejecutar el procesamiento como tal. En el caso de la erosión se utilizaron imágenes de tamaño 125x125 y en el caso de la detección de bordes se usaron imágenes de tamaño 107x107, de esta forma se puede lograr una comparación justa con el desempeño logrado por la implementación en hardware.

La Tabla 4.17 resume los tiempos asociados tanto al método *erode()* como al *CA* en software, considerando el tiempo de carga de imagen en promedio, el mismo para ambos casos. Por otro lado, la Tabla 4.16 expone la misma información pero referente a la ejecución de detección de bordes por *Canny()* y por el *CA*.

Implementación en hardware

En la implementación en hardware, se trabajó a una frecuencia de reloj de 100 *MHz*, correspondiente a la señal de reloj incluida en la tarjeta Nexys4 DDR. Para esta versión del *CA*, tiene que el tiempo de procesamiento es fijo, pues este se realiza en un solo ciclo de reloj al

Tabla 4.18: Resultados de tiempos de ejecución de procesamiento por el *CA* para cada regla de evolución implementada.

Regla de evolución	Tiempo de procesamiento	Tiempo de carga y descarga	Tiempo de procesamiento, carga y descarga
Erosión	0.00000001 s	0.00031250 s	0.00031252 s
Detección de bordes	0.00000001 s	0.00022898 s	0.00022899 s

pasar la *FSM* por el estado asociado a la evolución del *CA*, *PROCESS*, por lo que el tiempo de procesamiento depende únicamente de la frecuencia a la que trabaja el sistema. Sin embargo, se debe considerar como factor relevante en esta métrica el tiempo requerido para la comunicación para carga y descarga datos desde el *CA*, dado que el sistema de comunicación implementado corresponde a una interfaz *UART*, este tiempo de carga y descarga incrementa linealmente con la cantidad de pixeles de la imagen a procesar, es decir, la latencia del sistema o el tiempo requerido por el sistema para entregar como resultado una imagen procesada se puede caracterizar de la siguiente manera:

$$latencia = \frac{2(M \cdot N) + 1}{f} \quad (4.8)$$

Donde M y N corresponden a las dimensiones de la imagen y f es la frecuencia de la señal de reloj a la que opera el sistema.

La Tabla 4.18 resume los tiempos asociados a las implementaciones de cada una de las reglas de evolución propuestas, considerando en cada caso la dimensión máxima lograda, 107×107 y 125×125 para erosión y detección de bordes respectivamente.

Comparación

Los resultados obtenidos antes en esta sección, se resumen en la Tabla 4.19, donde se expone la ventaja, en términos de latencia, que presenta la implementación en hardware del *CA*, teniendo en contraste con técnicas estándar como los métodos incluidos en la biblioteca *OpenCV*. Se logra una aceleración de 8.6 veces el tiempo requerido para completar el proceso de procesamiento de una imagen, en el caso de la erosión y cerca de 4.5 veces menos para la detección de bordes. Teniendo en cuenta que más del 99% del tiempo que emplea el *CA* corresponde a *overhead*

asociado a la comunicación por puerto serial, se comprueba que la implementación de modelos de *CA* en hardware puede explotar la propiedad de procesamiento paralelo de estos sistemas y otorgar una gran aceleración de cómputo, sin embargo, para el diseño implementado, esta ventaja solo es válida para mallas o imágenes relativamente pequeñas del orden de los 100000 pixeles, ya que como se indica en la Ecuación 4.8, las dimensiones de la imagen a procesar son un factor significativo en la latencia de operación del sistema, por lo que si las dimensiones de la imagen aumentan, llegado un punto la ventaja que presenta la implementación en hardware del *CA* desaparece, a menos que se introduzca una mejora en la implementación de la comunicación y la carga-descarga de información.

Tabla 4.19: Resumen de latencia para el procesamiento de imágenes con cada método analizado.

Técnica de procesamiento	Latencia con OpenCV	Latencia con CA en software	Latencia con CA en hardware
Erosión	0.00271367 s	0.09129415 s	0.00031252 s
Detección de bordes	0.00099642 s	0.08957690 s	0.00022899 s

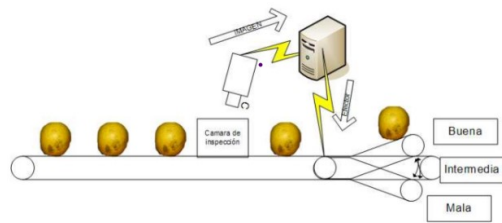
5. Potencial aplicación del diseño en la industria

En este capítulo se expone una posible aplicación del diseño implementado en este trabajo dentro de la industria. Se explica el contexto del uso del sistema y la motivación que lleva a plantearlo como solución a problemáticas dentro del área de uso. Se muestran también resultados experimentales obtenidos con el diseño en el contexto de la aplicación.

5.1. Contexto y motivación

El procesamiento de imágenes ve utilidad en diversas áreas, entre las que se encuentran la industria agroalimentaria, campo en que se emplean diversas herramientas tecnológicas para las etapas de producción, procesamiento y control de calidad de productos. Es por esto que el uso de algoritmos y técnicas que empleen procesamiento paralelo son muy importantes para disminuir los tiempos de ejecución de tareas computacionalmente intensas.

En trabajos anteriores como [3] mencionan la posibilidad de clasificar el estado de madurez o descomposición de las frutas basándose en características visuales de estas, de esto surge la necesidad de integrar hardware especializado que sea capaz de ejecutar tareas de procesamiento de imágenes y que a la vez pueda otorgar resultados en tiempo real para una línea de producción. En este contexto, la detección de bordes puede ser de ayuda para descartar frutas y vegetales defectuosos de manera automática.



(a) Diagrama del sistema donde se capturan, analizan y clasifican las imágenes de las naranjas.

	Good orange	Defective orange
Original image		
Border detection		

(b) Clasificación de naranjas defectuosas mediante detección de bordes.

Figura 5.26: Imágenes del contexto de uso de la detección de bordes en la industria agroalimentaria adaptadas de [3].

Se plantea como posible solución el sistema diseñado: un sistema que otorga aceleración por hardware a tareas de procesamiento de imágenes, en particular, erosión y detección de bordes basado en un modelo de autómata celular.

5.2. Acondicionamiento del sistema

Para poder aplicar el sistema diseñado en el contexto de clasificación de frutas defectuosas y sanas, se debieron realizar una serie de modificaciones al sistema para que fuera compatible y útil. Estas se explican y describen a continuación:

1. Preprocesamiento de las imágenes de entrada:

El sistema trabaja con imágenes binarias, por lo que es necesario convertir las imágenes de frutas a este formato, pero sin perder demasiada información o distorsionar la fruta. Para esto se desarrolla mediante software un preprocesamiento de las imágenes, que consiste en la separación de los canales RGB para convertirlos en imágenes binarias

de manera individual con umbrales específicos para cada uno de los canales. Dichos umbrales determinaron de manera heurística, para ello se realizó un análisis de los histogramas de cada canal para imágenes de naranjas que se encuentran en el dataset otorgado por el equipo de COFILAB [18]. Los canales que entregan más información sobre las naranjas en las imágenes son los canales rojo y verde, mientras que el canal azul entrega principalmente información sobre el fondo. La Figura 5.27 muestra los gráficos de los histogramas obtenidos para las imágenes del dataset utilizado.

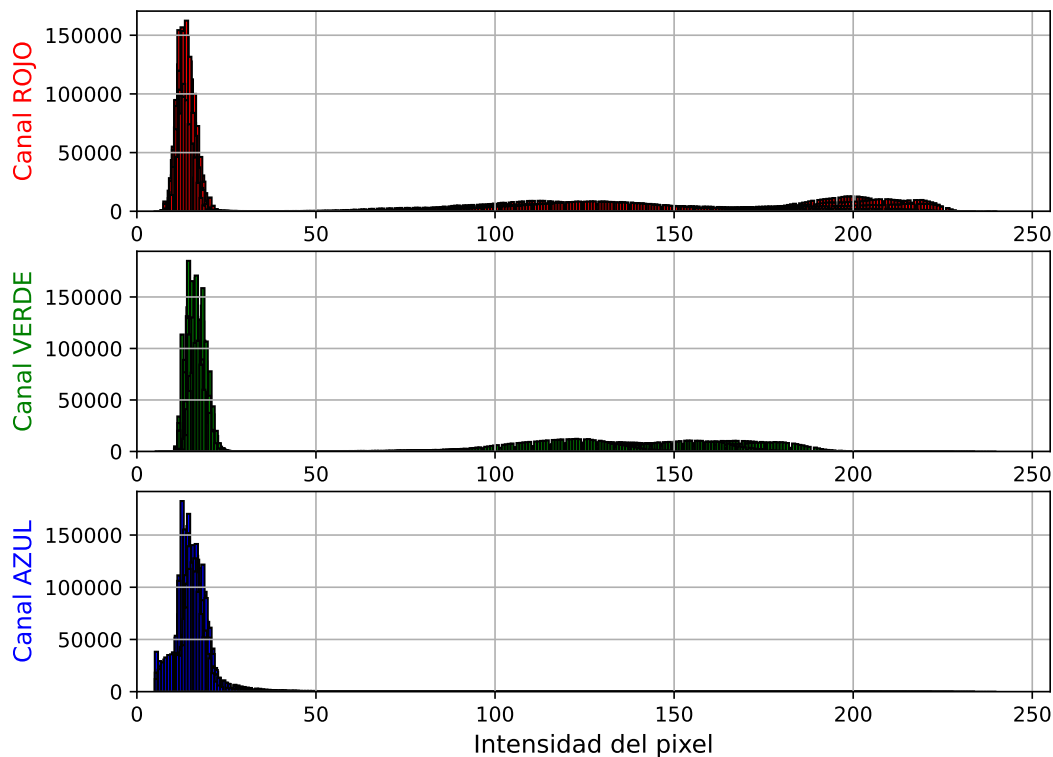


Figura 5.27: Histogramas para cada canal R, G y B del dataset de naranjas del grupo COFILAB.

Para obtener la imagen binarizada, se encontró de manera heurística, con base en lo mencionado anteriormente acerca de la información que entregan los canales RGB, una relación entre estos mismos que logra el objetivo. Esta relación es la siguiente:

$$Pix_result = (R \text{ or } G) - B \quad (5.9)$$

Con la relación anterior, se logra conservar la información de los canales que entregan aspectos característicos de las frutas en cuestión, sin importar su estado de madurez, mientras que al restar el canal asociado al fondo y colores oscuros, la información obtenida se mantiene intacta mientras que se logran mantener dentro del área que abarca la fruta, los elementos que indican defectos en ellas. Con esto en consideración, a partir de las gráficas de los histogramas, por inspección cualitativa, se definen como umbrales de los canales rojo y verde las intensidades de 175 y 100 respectivamente. Si bien existen pixeles con intensidades inferiores a 175 para el canal rojo, estas intensidades coinciden con las que se presentan con mayor densidad en el canal verde y dado que en el método de preprocesamiento de la imagen se aplica un operador lógico OR entre estos canales, basta con que esta intensidad sea considerada en el canal Verde así conservar para el canal Rojo únicamente los valores de intensidad más alta. Para el canal azul se considera un umbral arbitrariamente bajo de 20. En las Figuras 5.28 y 5.29 se muestran dos ejemplos de los pasos del preprocesamiento, en donde se logra apreciar la información que entrega cada canal a la imagen resultante según el método propuesto. Se puede ver como el resultado final conserva la forma y tamaño de la fruta mientras que no se pierde la información asociada a los defectos que esta presenta, los cuales permiten realizar la clasificación en producción.

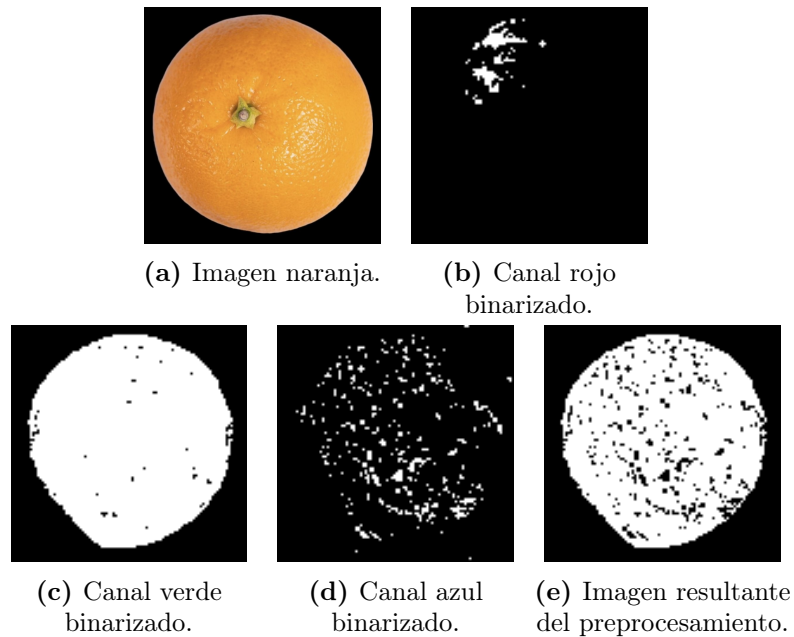


Figura 5.28: Resultados de preprocesamiento en imagen de naranja.

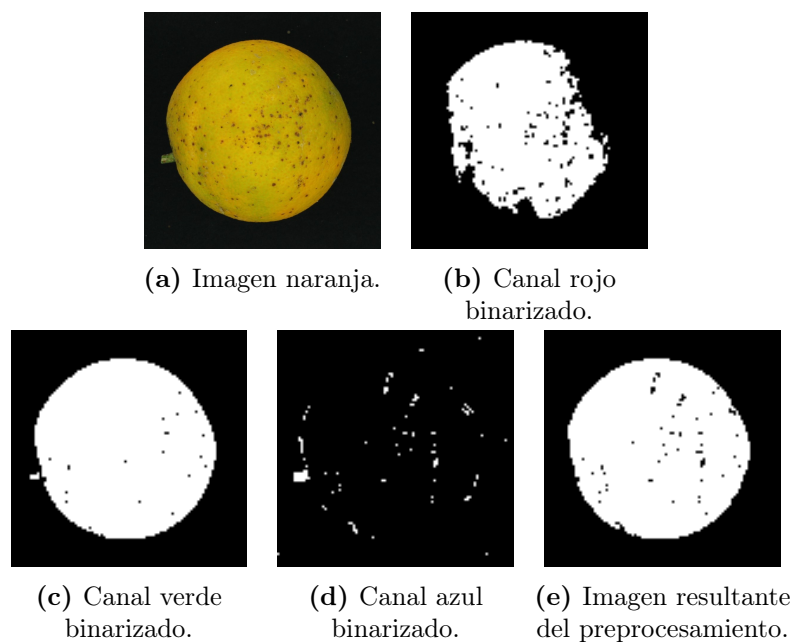


Figura 5.29: Resultados de preprocesamiento en imagen de naranja.

2. Combinación de las reglas de evolución:

Debido a que el tamaño máximo que soporta la tarjeta de desarrollo utilizada para implementar el *CA* diseñado, es necesario hacer un redimensionamiento de las imágenes, como se mencionó en la sección 4.2, sin embargo, con esto, los defectos existentes en las frutas quedan representados por pocos píxeles, por lo que la regla de evolución que permite detección de bordes no los podría identificar, tal como se concluyó en la misma sección. Esto puede solucionarse aplicando ambas reglas de evolución implementadas de manera consecutiva, de esta forma se logra que los defectos abarquen una mayor cantidad de píxeles y puedan sean considerados por la regla de detección de bordes.

Esto tiene implicancias en la definición de la máquina de estados del diseño en hardware, ya que la evolución del *CA* ahora es de dos pasos, teniendo que separar el estado **PROCESS**, que se describe en la sección 3.3.2 en dos estados consecutivos: **ERODE** y **EDGES**. Del mismo modo, este cambio afecta en la cantidad de bits necesarios para representar las señales **OPERATION** y por consiguiente, también al **MUX 1** de la Figura 3.13. La Figura 5.31 muestra el diagrama del sistema que resulta tras esta modificación.

- **ERODE**: estado en el que el *CA* evoluciona ejecutando la regla de erosión.
- **EDGES**: estado en el que el *CA* evoluciona ejecutando la regla de detección de bordes.

El diagrama de estados de la *FSM* resultante de esta modificación se muestra en la Figura 5.30.

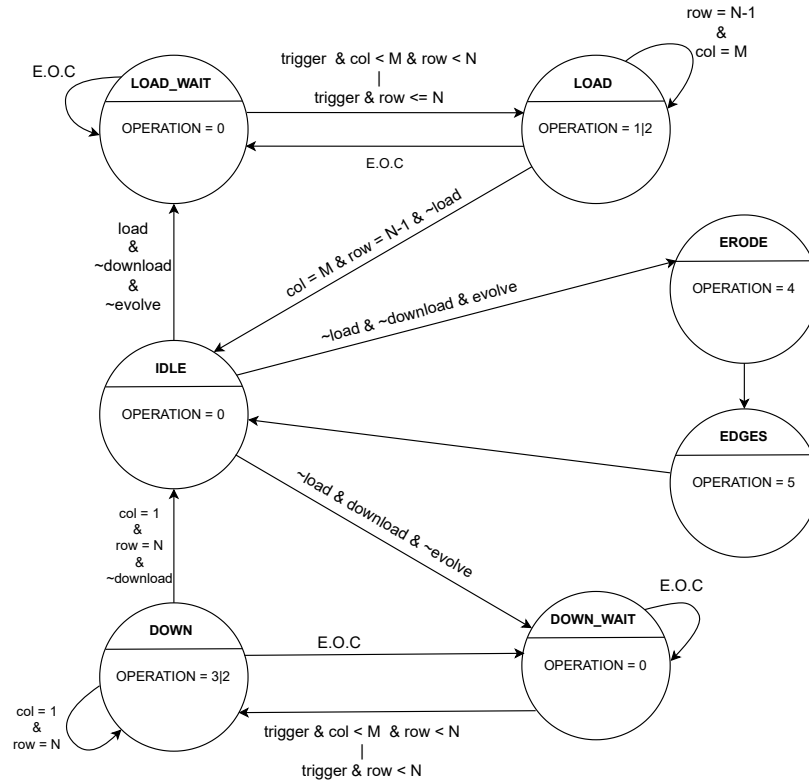


Figura 5.30: Diagrama de estados del módulo FSM_OPERATION con erosión y detección de bordes conjunta.

5.3. Resultados Experimentales

En esta sección se exponen los resultados del procesamiento de imágenes de naranjas en distintos niveles de madurez o putrefacción obtenidos mediante *CA* en software, *CA* en hardware y el método `Canny()` de la biblioteca OpenCV. También se muestra el uso de recursos para la implementación del diseño con el acondicionamiento mencionado en la sección anterior.

Con las nuevas funcionalidades se logró implementar una malla de *CA* de tamaño 99x99, el cual realiza el procesamiento de la imagen en dos pasos consecutivos, primero aplica la erosión y posteriormente la detección de bordes. Tabla 5.20 muestra la cantidad de recursos utilizados en este diseño, donde nuevamente se logró llegar prácticamente al máximo de recursos disponibles en la tarjeta Nexys4 DDR.

La Figura 5.32 muestra los resultados obtenidos con el sistema acondicionado para el procesa-

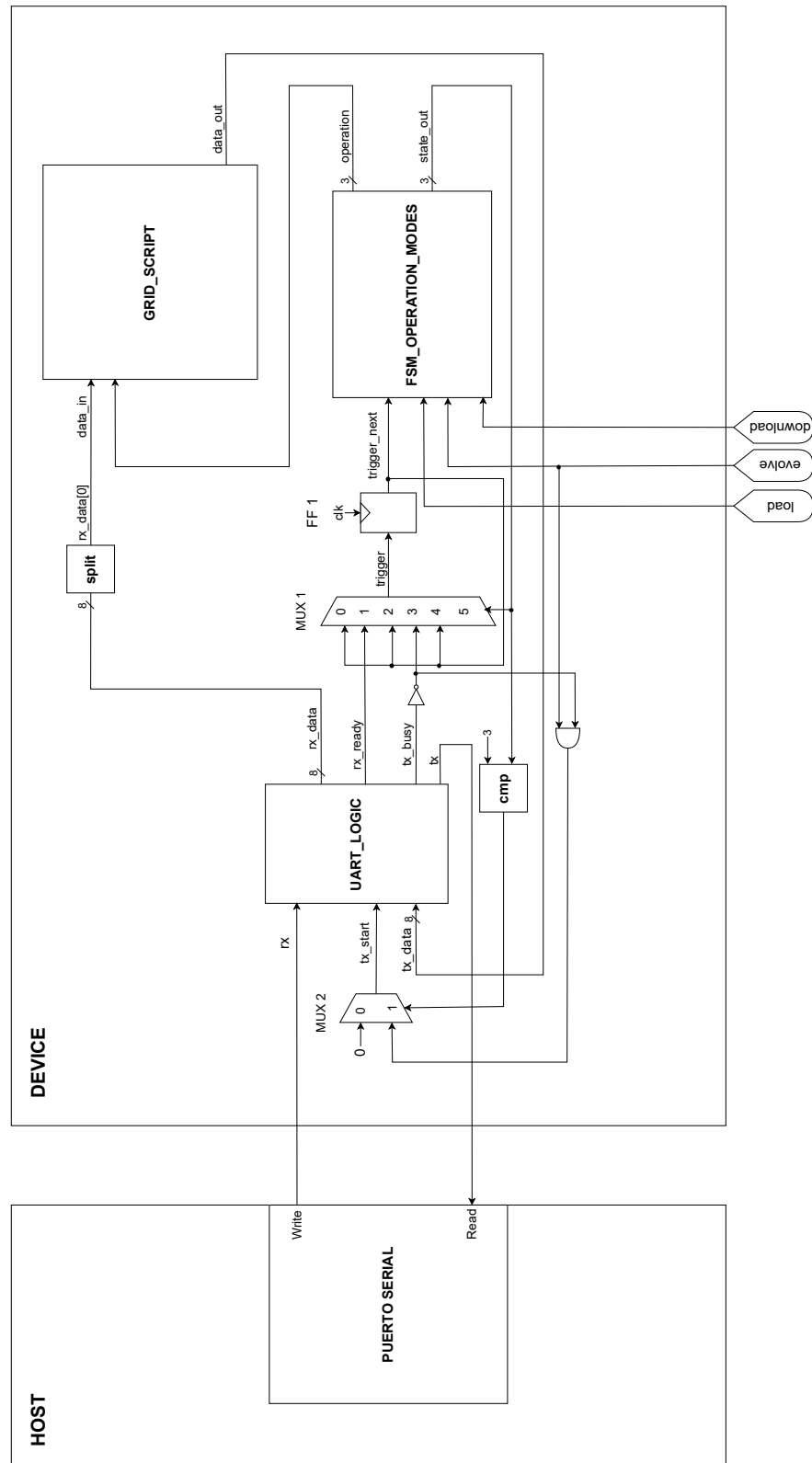


Figura 5.31: Diagrama de bloques del sistema modificado, agregando una opción más al MUX 1 y extendiendo la cantidad de bits para representar la señal **OPERATION**.

miento de imágenes de naranjas, en donde se puede notar que el uso de detección de bordes, tanto mediante el algoritmo de Canny, como a través del *CA* es posible reconocer el estado de las naranjas procesadas, en la naranja sana se reconoce el contorno de la fruta y pequeños bordes en el interior correspondientes al tallo de la misma, mientras que en las demás se detectan bordes al interior de la fruta, lo que implica que la naranja debe ser descartada en el contexto de una línea de producción.

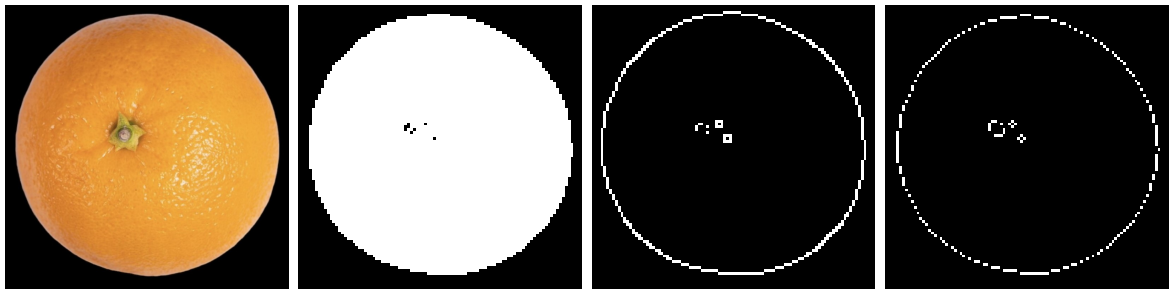
Las Tablas 5.20 muestran la cantidad de recursos utilizados en la implementación del sistema para la aplicación en la industria agroalimentaria, en desglose por módulos, mientras que la Tabla 5.21 muestra la utilización total de recursos en comparación con los disponibles en la tarjeta utilizada, donde se puede ver que nuevamente se han utilizado casi el máximo de los recursos disponibles en la tarjeta para poder procesar una imagen relativamente pequeña, de 99x99 píxeles.

Tabla 5.20: Utilización de recursos para el diseño enfocado en la potencial aplicación en la industria.

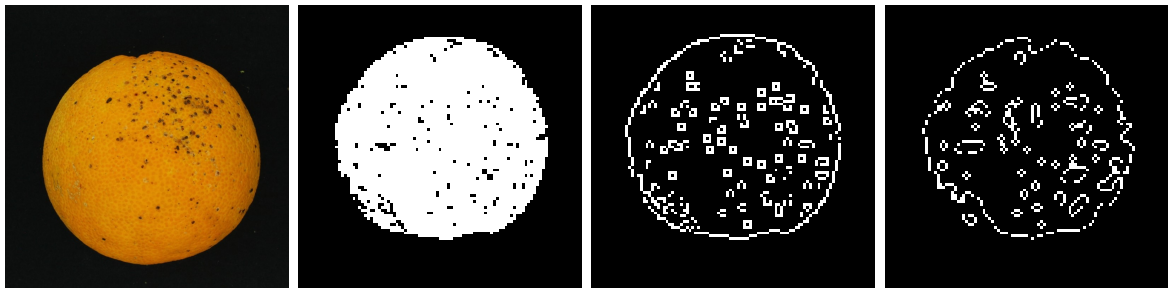
Módulo	LUT	Registros	Logic Slices
GRID_SCRIPT	57634	9801	15825
FSM_OPERATION_MODES	5031	97	4815
UART_LOGIC	28	67	23
BOTONES	36	45	19
TOTAL	62728	10011	15849

Tabla 5.21: Utilización de recursos para erosión respecto al total disponible en la tarjeta.

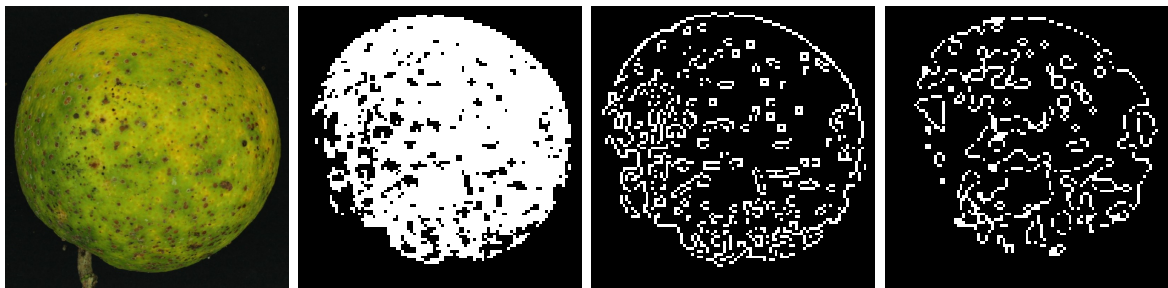
Tipo de recurso	Utilizado	Disponible	% Utilizado
LUT	62728	63400	98.94 %
Registros	10011	126800	7.90 %
Logic Slices	15849	15850	99.99 %



(a) Imagen base de Naranja sana. (b) Resultado de preprocesamiento. (c) Resultado de procesamiento con OpenCV. (d) Resultado de procesamiento con CA.



(e) Imagen base de Naranja defectuosa. (f) Resultado de preprocesamiento. (g) Resultado de procesamiento con OpenCV. (h) Resultado de procesamiento con CA.



(i) Imagen base de Naranja defectuosa. (j) Resultado de preprocesamiento. (k) Resultado de procesamiento con OpenCV. (l) Resultado de procesamiento con CA.

Figura 5.32: Resultados del procesamiento de imágenes de frutas.

La Figura 5.32 muestran los resultados del procesamiento de imágenes de naranjas en tres estados, en primer lugar, para una naranja sana (5.32a - 5.32d), luego para una naranja medianamente defectuosa (5.32e - 5.32h) y finalmente una naranja defectuosa (5.32i - 5.32l). En esta Figura se puede ver como los resultados del procesamiento de imágenes de naranjas defectuosas presentan claramente una mayor densidad de pixeles blancos, obtenidos a partir de la detección de bordes, mientras que en el resultado del procesamiento de la imagen con una naranja sana, prácticamente el único sector en donde se encuentran pixeles blancos son el contorno de la naranja y la zona del tallo. Esto hace suponer que es posible realizar clasificación de naranjas en base a su estado (sano o defectuoso) a partir de realizar detección de bordes.

En base a esto se propone un método de clasificación de naranjas entre dos categorías, sanas o defectuosas, que consiste en realizar el procesamiento de las imágenes con el sistema diseñado, para posteriormente hacer un conteo de la cantidad de pixeles blancos resultantes.

Para evaluar la factibilidad del método que se propone, se hizo una prueba preliminar, en la que se procesaron un conjunto de 10 naranjas sanas y 10 naranjas defectuosas, encontradas en el mismo dataset utilizado anteriormente [18], todas de dimensiones 800x800. Además, el procesamiento de las imágenes se efectuó a través de la versión de software del *CA*, dado que la versión actual en hardware no es capaz de realizar el procesamiento a imágenes con estas dimensiones, y según lo expuesto en la sección 4.3, el resultado obtenido a través de las dos versiones (software y hardware) es idéntico, por lo que es válido usar la versión de software para esta evaluación del método.

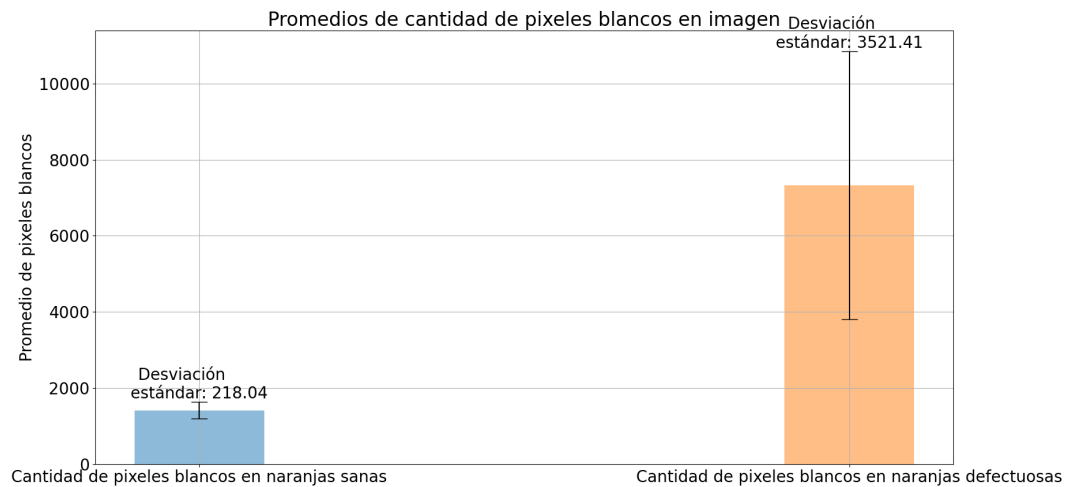


Figura 5.33: Promedios de cantidad de pixeles blancos en imagen junto a su desviación estándar.

La Figura 5.33 muestra el promedio de pixeles encontrados en las imágenes resultantes del procesamiento tanto para naranjas sanas como defectuosas. Se puede ver cómo en el caso de las naranjas defectuosas, el promedio de pixeles blancos encontrados en ellas es cercano a los 7300, mientras que en el caso de las naranjas sanas es cercano a los 1400, respecto a un total de 640000 que conforman cada imagen, siendo notoria la diferencia promedio en densidad de pixeles blancos encontrados. Además, se deben considerar más factores que pueden influir en la cantidad de pixeles blancos encontrados, como el tamaño de la naranja presente en la imagen y el nivel de defección de la naranja, por lo que también se considera en el análisis la desviación estándar de los resultados obtenidos, el cual se muestra también en la gráfica de la Figura, sobre la barra asociada al promedio en cada caso.

Se estudia entonces el caso más crítico en este contexto, que es cuando al promedio mayor se le resta su desviación estandar y al promedio menor se le suma su respectiva desviación estandar, de esta manera, cabe la posibilidad de que se solape la cantidad de pixeles blancos encontrados tanto para naranjas sanas como defectuosas. Este caso se verifica con la siguiente ecuación:

$$d = |(P_1 + \sigma_1) - (P_2 - \sigma_2)| \quad (5.10)$$

Donde P_1 corresponde al promedio de pixeles blancos encontrados en imágenes de naranjas sanas procesadas, σ_1 su desviación estándar, P_2 corresponde al promedio de pixeles blancos encontrados en imágenes de naranjas defectuosas procesadas, σ_2 y d es la diferencia encontrada en el caso crítico.

Al reemplazar los datos que se tienen en la ecuación 5.10:

$$d = | (1412 + 218.04) - (7328.2 - 3521.41) | \approx 2176$$

Se obtiene entonces una diferencia de aproximadamente 2176 pixeles lo que permite sospechar que el método si puede llegar a ser válido, ya que existe una diferencia significativa entre los dos promedios obtenidos considerando la variabilidad de los datos utilizados para su calculo, representada por las desviaciones estándar, lo que también se puede concluir de manera gráfica observando la Figura 5.33. Con ello se podría definir un umbral para la cantidad de pixeles blancos que se encuentren en una imagen procesada, que caracterice el estado de las naranjas, es decir, si la cantidad de pixeles blancos resultantes es inferior al umbral, la naranja se clasifica como sana y en caso contrario se clasifica como defectuosa.

Este análisis constituye un buen punto de partida para la definición de un método confiable. Sin embargo, para establecer este método de manera definitiva, sería necesario realizar pruebas más rigurosas y detalladas. Es importante ampliar el tamaño del dataset utilizado y llevar a cabo un análisis más completo para obtener conclusiones sólidas y respaldar aún más la validez del método propuesto. Este método se aleja de los alcances y objetivos del trabajo de memoria expuesto en este manuscrito, pero se plantea como posible extensión del proyecto en versiones futuras.

6. Conclusiones

En este trabajo se aborda el diseño e implementación en *CA* de un procesador basado en *CA* enfocado en procesamiento de imágenes. El procesamiento de imágenes es un campo en constante evolución y las técnicas utilizadas para ello están en permanente desarrollo. Se buscó explorar una alternativa mediante la implementación de autómatas celulares en FPGA, a la vez que se logra realizar una prueba práctica de las ventajas que presentan estos sistemas en el ámbito de la aceleración de procesamiento dadas sus características inherentes, el cómputo masivamente paralelo. Los principales objetivos del trabajo fueron diseñar y poner a prueba un procesador de imágenes basado en autómatas celulares en *CA* y comparar los resultados con los obtenidos mediante el uso de métodos tradicionales de procesamiento de imágenes para validarlos, además de comprobar la eficiencia de procesamiento que teóricamente debiese existir al trabajar con dispositivos que permitan explotar el procesamiento paralelo asociado a la evolución global de los modelos de *CA*.

En cuanto al procesamiento de imágenes como tal, los resultados son satisfactorios, puesto que en el caso de la regla erosión mediante *CA* estos resultan idénticos a los que se logran a través del método de erosión de la biblioteca OpenCV. Mientras que en el caso de la detección de bordes, si bien los resultados no son perfectos, son similares a los obtenidos por métodos tradicionales y ampliamente utilizados como el algoritmo de Canny, teniendo una diferencia en menos del 5 % de los píxeles resultantes del procesamiento. Si bien esto puede considerarse un error no despreciable, dependiendo de la aplicación, se debe tener en cuenta que Canny es un algoritmo de varios pasos mientras que el procesamiento a través de *CA* requiere un paso únicamente y se completa en un ciclo de reloj, por lo que se debe tener en consideración este *trade off* entre precisión y tiempo de ejecución antes de optar por el método propuesto como solución a la aplicación.

La idea principal del concepto de *CA*, eliminar la diferencia entre unidad de cómputo y unidad de almacenamiento, tal como se espera, resulta en una reducción del tiempo de procesamiento, sin embargo, este beneficio viene de la mano con un gran costo en lo que a recursos físicos se refiere, ya que en lugar de existir una unidad especializada para cómputo, cada celda que conforma la malla del *CA* corresponde a un procesador a pequeña escala, por lo que mientras

más celdas tiene la malla, se tienen más procesadores. Esto se comprobó con los reportes de utilización de recursos físicos de la tarjeta de desarrollo en la que se implementó el sistema, logrando en cada una de las pruebas realizadas estar a menos del 1 % de usar la totalidad de los recursos disponibles. Este nivel de utilización además, se obtuvo en implementación capaz de realizar las tareas de procesamiento en imágenes relativamente pequeñas, del orden de los 10000 pixeles, por lo que para implementaciones que soporten procesamiento para imágenes de mayor resolución se debe tener en cuenta este factor.

Para lograr un procesamiento eficiente de imágenes utilizando *CA* implementados en FPGA, resulta crucial contar con una interfaz de comunicación de alta velocidad, ya que los procesos de transmisión y recepción de datos pueden convertirse en un cuello de botella para el tiempo de operación del sistema. En particular, durante las pruebas realizadas, se observó que el módulo de comunicación representó más del 99 % de la latencia total, lo que opacó la potente ventaja en tiempo de cómputo, que se logró con respecto al procesamiento por software.

Dados los resultados obtenidos en el trabajo realizado, fue posible proponer una versión del sistema como base para un potencial uso en la industria agroalimentaria, enfocado en la clasificación de naranjas en líneas de producción, en donde se ve la necesidad de implementar sistemas de procesamiento de imágenes que permitan satisfacer los requerimientos de velocidad propias de la industria.

7. Trabajo Futuro

Con base en los resultados obtenidos a lo largo del trabajo, se proponen las siguientes posibles mejoras del trabajo presentado en el trabajo de memoria, los cuales pueden llevarse a cabo en un futuro:

- Optimización de la interfaz de comunicación, actualmente además del cuello de botella que se tiene por la interfaz de comunicación *UART*, esta se está sub utilizando, dado que este protocolo comúnmente transmite frames de 8 bits, sin embargo, la implementación actual utiliza esta interfaz para enviar un solo bit con información dentro de este frame, cuando se podrían enviar hasta 8 simultáneamente en el mismo frame y reducir de esta

manera el *overhead* que impone el módulo de comunicación a la latencia de operación del sistema.

- Cambiar la interfaz de comunicación a una que admita mayor tasa de transmisión de datos, como ETHERNET o HDMI.
- Migrar el diseño a una tarjeta de desarrollo con mayor cantidad de recursos físicos disponibles, para poder implementar un sistema capaz de desarrollar el procesamiento a imágenes de mejor resolución.
- Para la aplicación en la industria, implementar el preprocesamiento de las imágenes de entrada en el mismo hardware, de este modo el sistema resulta más compacto y cobra más sentido utilizarlo como coprocesador, ya que no habría ningún tipo de procesamiento por software.
- Para la aplicación en la industria, integrar un método de clasificación automática en base a la cantidad de píxeles blancos resultantes, dado que actualmente el sistema únicamente desarrolla el proceso de detección de bordes a las imágenes, pero no entrega ningún tipo de información con ese resultado.

Referencias

- [1] Digilent, “Nexys4 ddr™ fpga board reference manual,” 2016, [Web; accedido el 11-03-2023]. [Online]. Available: {https://digilent.com/reference/_media/reference/programmable-logic/nexys-4-ddr/nexys4ddr_rm.pdf}
- [2] D. R. Nayak, P. K. Patra, and A. Mahapatra, “A survey on two dimensional cellular automata and its application in image processing,” *arXiv preprint arXiv:1407.7626*, 2014.
- [3] J. P. Mercol, M. J. Gambini, and J. M. Santos, “Automatic classification of oranges using image processing and data mining techniques,” in *XIV Congreso Argentino de Ciencias de la Computación*, 2008.
- [4] R. M. Mardones Tabilo, “Rmardonest/ca_imageprocessingonfpga: Design and implementation in fpga of a processor based on cellular automata focused on image processing.” [Online]. Available: https://github.com/RMardonesT/CA_imageProcessingOnFPGA
- [5] B. Chopard and M. Droz, “„cellular automata modeling of physical systems, 2005.”
- [6] C. Mizas, G. C. Sirakoulis, V. Mardiris, I. Karafyllidis, N. Glykos, and R. Sandaltzopoulos, “Reconstruction of dna sequences using genetic algorithms and cellular automata: Towards mutation prediction?” *Biosystems*, vol. 92, no. 1, pp. 61–68, 2008.
- [7] P. L. Rosin, “Training cellular automata for image processing,” *IEEE transactions on image processing*, vol. 15, no. 7, pp. 2076–2087, 2006.
- [8] J. Mohammed and D. R. Nayak, “An efficient edge detection technique by two dimensional rectangular cellular automata,” in *International Conference on Information Communication and Embedded Systems (ICICES2014)*. IEEE, 2014, pp. 1–4.
- [9] J. R. M. Vilet, “Procesamiento digital de imagenes,” *Facultad de Ingeniería UASLP, Documento PDF, disponible en la página< http://read.pudn.com/downloads159/ebook/711796/Procesamiento_Digital_de_Imagenes.pdf>*[Citado 22 de septiembre de 2012], 2005.
- [10] S. Amrogowicz, Y. Zhao, and Y. Zhao, “An edge detection method using outer totalistic cellular automata,” *Neurocomputing*, vol. 214, pp. 643–653, 2016.
- [11] P. Rosin, “Training cellular automata for image processing,” *IEEE Transactions on Image Processing*, vol. 15, no. 7, pp. 2076–2087, 2006.
- [12] I. Katis and G. C. Sirakoulis, “Cellular automata on fpgas for image processing,” in *2012 16th Panhellenic Conference on Informatics*, 2012, pp. 308–313.
- [13] R. Bakhteri, J. Cheng, and A. Semmelhack, “Design and implementation of cellular automata on fpga for hardware acceleration,” *Procedia Computer Science*, vol. 171, pp. 1999–2007, 2020.
- [14] E. N. Millán, N. Wolovick, M. F. Piccoli, C. G. Garino, and E. M. Bringa, “Performance analysis and comparison of cellular automata gpu implementations,” *Cluster Computing*, vol. 20, no. 3, pp. 2763–2777, 2017.

- [15] I. Vourkas and G. C. Sirakoulis, “Fpga based cellular automata for environmental modeling,” in *2012 19th IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2012)*, 2012, pp. 93–96.
- [16] G. Carvajal B, “gcarvajalb/elo212-reference-modules.” [Online]. Available: https://github.com/gcarvajalb/ELO212-reference-modules/tree/master/UART_reference_files/UART_ME_reference/UART_master.srscs/sources/uart
- [17] [Online]. Available: https://st.depositphotos.com/1018782/3858/v/450/depositphotos_38580397-stock-illustration-leopard.jpg
- [18] COFILAB, “Oranges-infested-with-scale.” [Online]. Available: <http://www.cofilab.com/portfolio/orangesdb/>

8. Anexos

8.1. Funciones para implementación del CA en python

```

3 #####
##                                CREATE GRID
#####
def CA_init(heigth ,width, image):
    grid = []
8
    for y in range(heigth): #rows
        row = []
        for x in range(width): #columns
            row.append(image[y][x])
13    grid.append(row)

    return grid

18 #####
##                                READ CELL STATE
#####
def read_cell_state(col,row, CA, dimensions):
23
    N,M = dimensions
    #print('entran', col,row, 'Compara con' , M, N)
    if col >= M:
        col = col-2
    elif col < 0:
28        col = 1

    if row >= N:
        row = row -2
    elif row < 0:
33        row = 1

    return CA[row][col]
#####
##                                FIND NEIGHBORHOOD
38 #####

#encuentra la vecindad de la celda correspondiente a la coordenada (col,row)

```

```
def find_neighborhood(col,row,CA, dimensions):

43     MAIN = read_cell_state(col,row,CA, dimensions)

        N = read_cell_state(col,row-1,CA, dimensions)
        S = read_cell_state(col,row+1,CA, dimensions)

48     E = read_cell_state(col+1,row,CA, dimensions)
        O = read_cell_state(col-1,row,CA, dimensions)

        NE = read_cell_state(col+1,row-1,CA, dimensions)
        NO = read_cell_state(col-1,row-1,CA, dimensions)

53     SE = read_cell_state(col+1,row+1,CA, dimensions)
        SO = read_cell_state(col-1,row+1,CA, dimensions)

        return [NO,N,NE, O,MAIN,E, SO,S,SE]

58

#####
##                                REGLA DE EVOLUCION
#####
63 def rule(neighborhood_state, n_rule):

        neighborhood_state = np.array(neighborhood_state)/255

68     if n_rule == 0:

            if (sum(neighborhood_state) <= 8):
                return 0

73     else:
                return neighborhood_state[4]*255

    else:

83     if (sum(neighborhood_state) ==3 or sum(neighborhood_state) == 4):
                return 255

        else:
            return 0

#####
```

```
##                                GUARDADO DE ESTADOS ACTUALES
#####

def save_CA(CA,CA_buff, dimensions):
88     N,M = dimensions
        for row in range(N):
            for col in range(M):
                CA_buff[row][col] = CA[row][col]

93 #####

##                                EVOLUCION DEL CA
#####

def update_CA(CA, CA_buff, dimensions, n_rule):

98     N,M = dimensions
        save_CA(CA, CA_buff, dimensions)

        for row in range(N):

103         for col in range(M):

                neighborhood = find_neighborhood(col,row,CA_buff, dimensions)

                CA[row][col] = rule(neighborhood, n_rule)

108

#####

113 def show(CA, buff):
        for i in range(len(CA)):
            print(CA[i], "    ", buff[i])
        print()

#####
```

8.2. Módulo de Celda

```

timescale 1ns / 1ps

module CELDA

#(parameter
    ic = 0,
    top_row = 0,
    bottom_row = 0,

    load_cell = 0,
    download_cell = 0)
(

    input logic clk, reset,

    input logic [2:0] operation, //0: no operation i.e normal
                                //1: horizontal operation (top row only), 2:
                                //vertical operation

    input logic [0:0] NO, N, NE, //top neighbors
    input logic [0:0] O, E,      //side neighbor
    input logic [0:0] SO, S, SE, //bottom neighbor
    input logic [0:0] SELF, //main state of cell

    output logic [0:0] cell_state
);

```



```
logic [0:0] next_cell_state;
logic [7:0] sum_neighborhood;
always_ff @(posedge clk) begin

    if (reset)
        cell_state <= ic;

    else
        cell_state <= next_cell_state;
end

assign sum_neighborhood = NO + N + NE + O + E + SO + S + SE +
    SELF;

always_comb begin

    case(operation)

        //0 KEEP CURRENT STATE
        'd0: begin
            next_cell_state = SELF;
        end

        //1 HORIZONTAL SHIFT ON TOP ROW (load)
        'd1: begin
            if(top_row)
                next_cell_state = O;
        end
    endcase
end
```

```
        else
            next_cell_state = SELF;
        end

//2 VERTICAL SHIFT
'd2:    begin
        //copy state from northern neighbor
        if (top_row == 0)
            next_cell_state = N;

        else
            //corner cell gets new data_in from uart
            if (load_cell)
                next_cell_state = O;

            //cell manteins its current state
            else
                next_cell_state = SELF;
            end
        end

//3 HORIZONTAL SHIFT ON BOTTOM ROW (download)
'd3:    begin
        if (bottom_row)
            next_cell_state = O;
        else
            next_cell_state = SELF;
        end
    end
```

//4 NORMAL EVOLUTION

```
'd4:    begin /*
        if ( (sum_neighborhod == 'd3) | (
            sum_neighborhod == 'd4)) // ((
            sum_neighborhod == 'd4) & (SELF == 'd1)) |
            next_cell_state = 'd1;

        else
            next_cell_state = 0;

        */

        if (sum_neighborhod <= 8)
            next_cell_state = 0;
        else
            next_cell_state = cell_state;

    end

//5 NORMAL EVOLUTION edge detection
'd5: begin
    if ( (sum_neighborhod == 'd3) | (sum_neighborhod
        == 'd4)) // ((sum_neighborhod == 'd4) & (SELF ==
        'd1)) |
        next_cell_state = 'd1;

    else
```



```
                                next_cell_state = 0;

                                end

                                default:    next_cell_state = cell_state;

                                endcase

                                end

                                endmodule
```