



UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERIA ELECTRICA

DESARROLLO DE TÉCNICAS DE OPTIMIZACIÓN DE MÁQUINAS DE ESTADO
FINITO (FSM) PARA SU IMPLEMENTACIÓN BASADA EN UN FPGA

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELECTRICO

ANTONIO ENRIQUE RODRÍGUEZ GAJARDO

PROFESOR GUÍA:
FRANCISCO RIVERA SERRANO

MIEMBROS DE LA COMISIÓN:
ANDRÉS CABA RUTTE
MARCELO JIMÉNEZ DE FONSECA

SANTIAGO DE CHILE
2023

Resumen

DESARROLLO DE TÉCNICAS DE OPTIMIZACIÓN DE MÁQUINAS DE ESTADO FINITO (FSM) PARA SU IMPLEMENTACIÓN BASADA EN UN FPGA

En el ámbito de la electrónica digital, las FSM (*Finite State Machine*) juegan un papel importante en la implementación de sistemas de control para diferentes tipos de hardware. Este trabajo se centra en técnicas avanzadas de optimización de estas máquinas FSM y su aplicabilidad en dispositivos FPGA (*Field-Programmable Gate Array*). Se analiza la optimización desde tres perspectivas: eficiencia energética, rapidez en la ejecución de los algoritmos y optimización de recursos de hardware.

El objetivo principal de esta memoria de título es desarrollar estas técnicas de optimización de máquinas FSM, y finalmente, implementarlas en un software que optimice estas máquinas y las implemente en un FPGA real.

Para alcanzar este objetivo, se adoptó una metodología que comenzó con una revisión bibliográfica, abarcando técnicas de optimización de máquinas FSM. Según los algoritmos encontrados, se diseñó y desarrolló un algoritmo especializado que posteriormente se integró en una plataforma web. Esta herramienta web permite a los usuarios ingresar máquinas FSM, optimizarlas y, finalmente, generar archivos *SystemVerilog* listos para ser implementados en FPGA.

Se compilaron los archivos *SystemVerilog* utilizando software de optimización y se llevaron a cabo pruebas detalladas. Estas pruebas se centraron en evaluar la cantidad de recursos de hardware que la máquina FSM consumía antes y después del proceso de optimización. Los resultados, obtenidos mediante el uso de Quartus Prime, no solo confirmaron la eficacia de la herramienta desarrollada, sino que también resaltaron su potencial en la mejora de sistemas basados en FPGA.

En resumen, en este trabajo se desarrolló una herramienta innovadora que no solo optimiza máquinas FSM, sino que también permite su implementación en FPGA.

*Le dedico mi trabajo a mis padres Isabel Gajardo y Mauricio Rodríguez por todo este tiempo
apoyándome, gracias*

Agradecimientos

Agradezco al profesor Francisco Rivera por su paciencia y apoyo, y agradezco a mis padres por su guía incondicional.

Tabla de Contenido

| | |
|---|----------|
| 1. Introducción | 1 |
| 1.1. Formulación y Presentación del Problema | 1 |
| 1.2. Motivación | 2 |
| 1.3. Objetivos de este trabajo | 2 |
| 1.4. Organización del documento | 3 |
| 2. Marco Teórico y Estado del Arte | 5 |
| 2.1. Marco Teórico | 6 |
| 2.1.1. Circuitos Digitales y FPGA | 6 |
| 2.1.2. Máquinas de Estado | 7 |
| 2.1.3. Máquina de Mealy | 7 |
| 2.1.4. Máquina de Moore | 8 |
| 2.1.5. Diagrama de Flujo y Diagrama MDS | 9 |
| 2.2. Optimización de Máquinas de Estado | 10 |
| 2.2.1. Optimización de un Diagrama MDS y Conversión a HDL | 11 |
| 2.2.2. Codificación de Estados | 11 |
| 2.2.3. La Minimización de la Función de Transición | 12 |
| 2.2.4. Minimización Máquina de Estados Finitos | 12 |
| 2.3. Estado del Arte | 13 |
| 2.3.1. Vivado | 13 |
| 2.3.2. Intel Quartus Prime | 14 |

| | | |
|-----------|---|-----------|
| 2.3.3. | Latice Diamond | 14 |
| 2.3.4. | Herramientas de diseño de software | 15 |
| 3. | Implementación del Software de Optimización de Máquinas de Estado | 16 |
| 3.1. | Metodología de Trabajo | 16 |
| 3.2. | Algoritmo de Optimización | 18 |
| 3.2.1. | Minimización de Estados | 18 |
| 3.2.2. | Minimización de Funciones de Transición. | 19 |
| 3.2.3. | Representación en Códigos Binarios | 20 |
| 3.3. | Implementación de la Solución o Herramienta | 20 |
| 3.3.1. | Interfaz Gráfica | 21 |
| 3.3.2. | Diseño e Implementación de la API Flask | 26 |
| 4. | Pruebas y Resultados del Software de Optimización de Máquinas FSM | 28 |
| 4.1. | Software usado para validar la máquina FSM optimizada y FPGA utilizada . | 29 |
| 4.2. | Optimización de máquina FSM de tres nodos | 30 |
| 4.2.1. | Compilación de Códigos para máquina FSM de tres nodos y su optimización | 30 |
| 4.2.2. | Resultados | 31 |
| 4.3. | Ejemplo de una Máquina Expendedora | 33 |
| 4.3.1. | Compilación de Códigos | 36 |
| 4.3.2. | Resultados | 38 |
| 4.4. | Análisis de los Resultados | 40 |
| 4.4.1. | Máquina FSM de 3 nodos | 40 |
| 4.4.2. | Máquina expendedora | 40 |
| 5. | Conclusiones y Trabajo a Futuro | 41 |
| 5.1. | Trabajo a Futuro | 42 |
| | Bibliografía | 44 |

Índice de Ilustraciones

| | |
|---|----|
| 2.1. Arquitectura básica de un FPGA | 7 |
| 2.2. Máquina de Mealy | 8 |
| 2.3. Máquina de Moore | 9 |
| 2.4. Ejemplo de un Diagrama de Flujo | 10 |
| 2.5. Ejemplo de dos Diagramas equivalentes. Diagrama de Flujo y MDS. | 10 |
| 2.6. Software State Editor | 14 |
| 2.7. StateFlow Matlab/Simulink | 15 |
| 3.1. Diagrama de metodología de trabajo. | 18 |
| 3.2. Representación gráfica de distancias de Hamming de 3 bits. | 20 |
| 3.3. Desarrollo software | 21 |
| 3.4. Página de inicio | 22 |
| 3.5. Página de inicio - Sub-página | 22 |
| 3.6. Ventana "Guía de uso". | 23 |
| 3.7. Ventana "Guía de uso Tutorial". | 24 |
| 3.8. Ventana "Optimizador". | 25 |
| 3.9. Pestaña "Forma" de la ventana "Optimizador". | 25 |
| 3.10. Descripción de los nodos en JSON. | 26 |
| 3.11. Ejemplo de dos Diagramas equivalentes. Diagrama de Flujo y MDS. | 26 |
| 4.1. FPGA Cyclon V, Tarjeta de desarrollo DE1-SoC | 29 |
| 4.2. Máquina FSM de una máquina expendedora | 30 |

| | |
|---|----|
| 4.3. Diagrama MDS de la máquina FSM de 3 nodos y su optimización. | 30 |
| 4.4. Circuito RTL de máquina FSM de 3 nodos | 31 |
| 4.5. Circuito RTL de máquina FSM de 3 nodos optimizada | 31 |
| 4.6. Resumen de los resultados del código compilado de la maquina FSM de 3 nodos. | 32 |
| 4.7. Resumen de los resultados del código compilado de la maquina FSM de 3 nodos optimizada. | 32 |
| 4.8. Máquina FSM de una máquina expendedora | 34 |
| 4.9. Máquina FSM de una máquina expendedora | 35 |
| 4.10. Máquina FSM de una máquina expendedora | 36 |
| 4.11. Circuito equivalente RTL de una máquina expendedora | 37 |
| 4.12. Circuito equivalente RTL de una máquina expendedora optimizada | 38 |
| 4.13. Resumen de los resultados del código compilado de la máquina expendedora | 39 |
| 4.14. Resumen de los resultados del código compilado de la máquina expendedora optimizada. | 39 |

Capítulo 1

Introducción

1.1. Formulación y Presentación del Problema

En la era actual de la tecnología digital, la creciente demanda de sistemas electrónicos eficientes y de alto rendimiento ha impulsado el desarrollo de diversas técnicas de diseño y optimización. Entre estas técnicas, las FSM (*Finite State Machines*) han demostrado ser fundamentales en la construcción de sistemas digitales que responden a una amplia gama de aplicaciones, desde controladores simples hasta sistemas complejos de comunicación y procesamiento.

Las máquinas FSM, como modelos matemáticos de sistemas que cambian de estado en respuesta a eventos externos, forman una parte esencial en el diseño electrónico y la lógica secuencial, por lo tanto, la implementación eficiente de máquinas FSM es esencial para lograr un desempeño óptimo y un uso eficiente de los recursos en sistemas digitales. En este contexto, los FPGA (*Field Programmable Gate Arrays*) se destacan como plataformas para implementar lógica digital y sistemas basados en máquinas FSM, ofreciendo paralelismo y flexibilidad en su configuración.

Los FPGA y los circuitos digitales son grandes actores en la electrónica moderna. El incremento en el uso de los circuitos digitales y los FPGA, es una consecuencia directa de la creciente demanda de procesamiento de datos y de la digitalización de diversas industrias como:

- En la electrónica de consumo, desde los teléfonos inteligentes y computadoras hasta televisores, sistemas de sonido y electrodomésticos inteligentes. Estos circuitos permiten la implementación de funciones complejas y la integración de múltiples dispositivos en un solo sistema.
- En la industria automotriz, los circuitos digitales se utilizan en sistemas de control de motores, sistemas de seguridad y entretenimiento, y en la interfaz hombre-máquina de los vehículos. En sistemas de procesamiento de señales para mejorar la calidad del sonido y la imagen en los sistemas de audio y video de los vehículos.

- Los FPGA son especialmente útiles en la industria aeroespacial y de defensa, ya que pueden reprogramarse para adaptarse a diferentes misiones y condiciones de operación.
- En la industria de las telecomunicaciones, los circuitos digitales y los FPGA se utilizan en la implementación de sistemas de conmutación, enrutamiento y procesamiento de señales de audio y video y datos en redes de telecomunicaciones.

1.2. Motivación

La necesidad de mejorar el desempeño de los circuitos digitales, es constante en la industria de la electrónica, ya que se busca mejorar la eficiencia y el rendimiento de los sistemas digitales y dispositivos eléctricos. Haciendo hincapié esencialmente en industrias que dependen directamente del rendimiento de sus dispositivos como, por ejemplo, la industria espacial, que si se desea hacer cualquier cambio de circuitos en un equipo en el espacio, es necesario hacer una importante inversión económica para enviar a una persona a modificar el instrumento o la industria automotriz que está transitando a la electromovilidad, sin embargo, uno de sus más grandes restricciones es el rendimiento de sus baterías.

En particular, tres parámetros son los más influyentes en el rendimiento de un circuito lógico. Estos son la velocidad de ejecución de una cadena de instrucciones, el gasto de energía generado por un circuito eléctrico, generalmente en forma de disipación de temperatura, y la cantidad de lógica utilizada para implementar las funcionalidades del circuito. Existen distintas técnicas de optimización que permiten reducir el gasto energético, la cantidad de lógica utilizada y, en algunos casos, la velocidad de ejecución de las cadenas de instrucciones:

- Reducción de la latencia: la latencia es el tiempo que tarda un sistema en responder al cambio de una entrada. La reducción de la latencia en los circuitos digitales puede mejorar significativamente la frecuencia de procesamiento y la eficiencia de los sistemas.
- Aumento de la frecuencia de reloj: el reloj es la señal que sincroniza el funcionamiento de los circuitos digitales. Aumentar la velocidad del reloj permite mejorar la velocidad de procesamiento de los sistemas.
- Optimización del diseño: un diseño bien optimizado puede mejorar significativamente el desempeño de los circuitos digitales, ya que permite la reducción del consumo de energía y la eliminación de cuellos de botella en el procesamiento de datos.
- Implementación en FPGA: la implementación de circuitos digitales en FPGA puede mejorar el desempeño de los sistemas al permitir la paralelización de tareas y la optimización del diseño.

1.3. Objetivos de este trabajo

El objetivo general de este trabajo consiste en crear un algoritmo para la optimización del diseño de máquinas de estado finito y desempeño de circuitos digitales en un FPGA. Para

lograr este objetivo, este trabajo se ha dividido en los siguientes objetivos específicos.

1. Analizar los diferentes métodos para la optimización del consumo de energía, el consumo de recursos de la FPGA y la velocidad de máquinas de estados, revisando el estado del arte de dichos métodos.
2. Crear un algoritmo que tenga como entrada una máquina de estados finitos, utilice dichos métodos de manera eficiente para optimizar la máquina de estados y retorne una nueva máquina de estados optimizada.
3. Diseñar e implementar una interfaz gráfica que permita que cualquier usuario, sin necesidad de tener experiencia en programación de software, utilice la herramienta.
4. Crear un algoritmo que utilice como entrada una máquina de estados y entregue un archivo en *System Verilog* de la máquina de estados optimizada.

1.4. Organización del documento

La presente memoria se estructura en cuatro capítulos esenciales. El primer capítulo, titulado "Introducción", aborda la presentación del problema, la motivación subyacente y los objetivos propuestos.

El segundo capítulo, bajo el título "Marco Teórico y Estado del Arte", se dedica a elucidar los conceptos de circuitos digitales y máquinas de estado. En él, se ofrece una panorámica detallada sobre las máquinas de estados, sus aplicaciones y su concreción en circuitos digitales. Asimismo, se discuten los diversos enfoques orientados a optimizar dichas máquinas, tanto en términos de velocidad de procesamiento como en eficiencia energética. Este capítulo culmina con una revisión del estado actual del arte en la materia.

El tercer capítulo, denominado "Implementación del Software de Optimización de Máquinas de Estado", inicia con la exposición de la metodología adoptada para el desarrollo del software de implementación de Máquinas FSM. Posteriormente, se desgrena el proceso de desarrollo del algoritmo y la interfaz gráfica destinada a la optimización de máquinas de estados. Se realiza un análisis meticuloso de las técnicas de optimización de estados finitos, poniendo especial énfasis en las técnicas seleccionadas para el diseño del algoritmo. Se detalla la concepción y operatividad del algoritmo de optimización de máquinas de estados, así como el algoritmo de compilación para código *System Verilog*. El capítulo concluye con la presentación de la interfaz gráfica diseñada específicamente para la implementación de dicho algoritmo.

El cuarto, titulado "Pruebas y Resultados del Software de Optimización de Máquinas FSM", presenta los resultados obtenidos al emplear máquinas redactadas en *System Verilog* y el software de prueba de Altera, Quartus Pro, para evaluar el algoritmo propuesto. Se exhiben pruebas realizadas con máquinas de Mealy y Moore. Como colofón, se ilustra una aplicación práctica que demuestra la eficacia del software al minimizar un algoritmo de correlación cruzada.

El capítulo 5 y final, titulado "Conclusión", sintetiza las reflexiones más relevantes y esboza las posibles direcciones para investigaciones y trabajos futuros derivados de este estudio.

Capítulo 2

Marco Teórico y Estado del Arte

Una metodología aceptada en diferentes industrias, como en IoT, sistemas de control de potencia y sistemas de transporte, que resulta eficiente y efectiva para el diseño de circuitos digitales, es utilizar un diagrama de flujo que permite visualizar el paso de las señales directamente desde la entrada a la salida [27] [25] [24].

Para convertir un diagrama de flujo en un esquema circuital se va a utilizar la técnica de diseño por paso de un diagrama MDS (*Mnemonic Documented State diagram*). Esta técnica permite redefinir el diagrama de flujo en una máquina de estado finito donde las transiciones dependen de una función de transición booleana. La ventaja del diagrama MDS es que su interpretación a un programa HDL resulta directo [21].

En el presente capítulo se establecen las bases fundamentales que proporcionarán un contexto sólido para la comprensión del contenido subsiguiente de esta memoria de título. Se procederá a examinar de manera detallada el funcionamiento de los circuitos digitales, junto con las estrategias clave que rigen su proceso de diseño. Se brindará una explicación clara de cómo se realiza la transición desde un simple diagrama de flujo hacia la representación en un esquema MDS y, finalmente, cómo esto se traduce en un circuito lógico concreto.

La sección final sino hace un análisis del estado actual en el ámbito de la optimización de estados finitos. Se llevará a cabo una exploración de las tendencias y los desarrollos más recientes en este campo en evolución. Asimismo, se examinará detenidamente las herramientas de diseño de software diseñadas específicamente para FPGA, evaluando su relevancia y contribución al progreso tecnológico contemporáneo.

A lo largo de este capítulo, se fusionarán de manera coherente los aspectos teóricos y las tendencias actuales, proporcionando un cimiento sólido para las investigaciones posteriores y la proposición de enfoques innovadores en el área de estudio.

2.1. Marco Teórico

2.1.1. Circuitos Digitales y FPGA

Los circuitos digitales se encargan de procesar señales discretas, es decir, aquellas que adoptan valores bien definidos y están separadas en el tiempo. Estas señales son de naturaleza binaria, representadas por los valores 0 o 1, y se utilizan para codificar información digital, como textos, imágenes, sonidos y otros datos.

Estos circuitos digitales están compuestos por dispositivos lógicos, tales como compuertas lógicas, *Flip-Flop* y *Latch*, que permiten la realización de operaciones lógicas y aritméticas en el hardware. Estos dispositivos se combinan para formar bloques funcionales más complejos llamados CLB (*Configurable Logic Block*).

El diseño y la implementación de circuitos digitales pueden ser realizados mediante software de diseño electrónico, como software de simulación, y pueden ser implementados en diferentes tecnologías, como ASIC (*Application Specific Integrated Circuits*), FPGA y microcontroladores.

Los FPGA permiten la implementación directa de bloques lógicos gracias a su arquitectura, que se basa en una matriz de bloques lógicos CLB con conexiones configurables de entradas y salidas por puertos IO (*Input and Output ports*).

Estos bloques lógicos contienen LUT (*Look Up Table*), *Flip-Flops* y multiplexores que pueden interconectarse y reconfigurarse mediante software HDL (*Hardware Description Language*) para formar circuitos digitales personalizados. La figura 2.1 muestra la arquitectura básica de un FPGA siendo CLB los bloques lógicos programables y los IO los puertos de entrada y salida del FPGA [9].

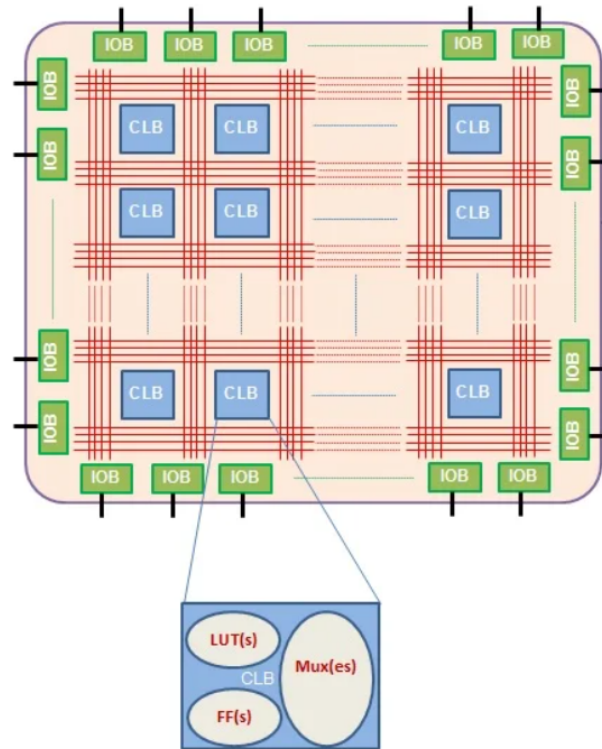


Figura 2.1: Arquitectura básica de un FPGA

2.1.2. Máquinas de Estado

Una máquina de estados finitos es un modelo computacional que se utiliza para representar el comportamiento de un algoritmo particular. Se compone de un conjunto finito de estados, un conjunto de funciones de transición de estado y una función de salida. El comportamiento de la máquina depende del estado actual y de la entrada, y genera una salida en función de estos. Las máquinas de Mealy y Moore son dos tipos de máquinas FSM utilizadas en el diseño de circuitos digitales donde las funciones de transición son simplemente valores booleanos. Se diferencian en que en la primera, la salida depende del estado actual y su entrada, mientras que la segunda depende exclusivamente del estado actual. Este concepto se puede aplicar a las máquinas FSM para definir sus salidas. En general, las máquinas de Moore se utilizan en aplicaciones donde la salida debe mantenerse constante durante un ciclo de reloj, mientras que las máquinas de Mealy se utilizan en aplicaciones donde la salida debe ser actualizada más frecuentemente, en función de las entradas.

2.1.3. Máquina de Mealy

En una máquina de Mealy, la función de salida depende tanto del estado actual como de la entrada actual, es decir, la salida se determina a partir del estado actual y de la entrada actual. En este caso, la salida puede cambiar en cualquier momento, incluso durante un ciclo de reloj[17].

La figura número 2.2 representa una máquina de Mealy donde cada nodo funciona de la manera siguiente:

- Nodo A: es el primer nodo de la máquina de Mealy. Si la entrada es un 1 lógico la salida es 0 y la máquina pasa al estado B. Si la entrada es un 0 lógico la salida es 0 y la máquina pasa al estado D.
- Nodo B: si la entrada es un 1 lógico la salida es 1 y la máquina pasa al estado C. Si la entrada es un 0 lógico la salida es 1 y la máquina pasa al estado A.
- Nodo C: si la entrada es un 1 lógico la salida es 0 y la máquina pasa al estado D. Si la entrada es un 0 lógico la salida es 0 y la máquina pasa al estado B.
- Nodo D: si la entrada es un 1 lógico la salida es 1 y la máquina pasa al estado A. Si la entrada es un 0 lógico la salida es 1 y la máquina pasa al estado C.

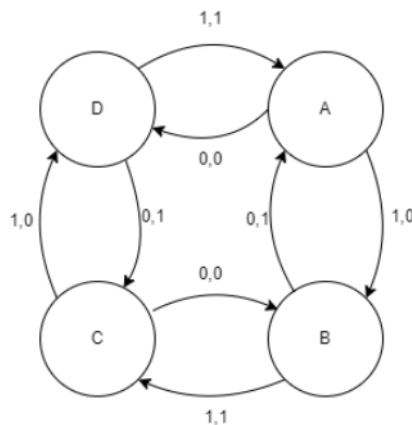


Figura 2.2: Máquina de Mealy

2.1.4. Máquina de Moore

En una máquina de Moore, la función de salida depende únicamente del estado actual de la máquina, es decir, la salida se determina directamente a partir del estado en el que se encuentra la máquina. En este caso, la salida se mantiene constante durante todo el ciclo de reloj [18].

La figura número 2.3 representa una máquina de Moore. Donde cada nodo funciona de la manera siguiente:

- Nodo A: es el primer nodo de la máquina de Moore. Si la entrada es un 1 lógico la máquina pasa al estado B. Si la entrada es un 0 la máquina pasa al estado D. La salida siempre es 0.
- Nodo B: si la entrada es un 1 lógico la máquina pasa al estado C. Si la entrada es un 0 la máquina pasa al estado A. La salida siempre es 1.

- Nodo C: si la entrada es un 1 lógico la máquina pasa al estado D. Si la entrada es un 0 lógico la máquina pasa al estado B. La salida siempre es 0.
- Nodo D: si la entrada es un 1 lógico la máquina pasa al estado A. Si la entrada es un 0 lógico la máquina pasa al estado B. La salida siempre es 1.

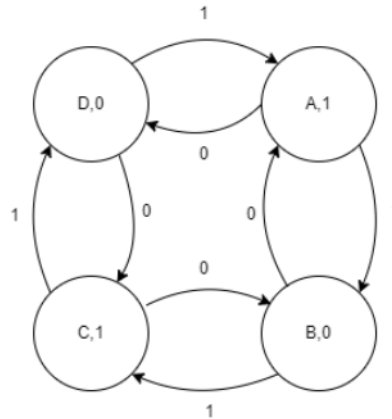


Figura 2.3: Máquina de Moore

2.1.5. Diagrama de Flujo y Diagrama MDS

Diagrama de Flujo

Un diagrama de flujo es una representación gráfica de un proceso o algoritmo. Se utiliza para visualizar las etapas que se siguen para completar una tarea o resolver un problema. En un diagrama de flujo, se utilizan símbolos gráficos para el proceso. Estos símbolos están conectados por flechas que indican la secuencia en la que se realizan las acciones. Los símbolos principales que se utilizan son los estados que son representados por rectángulos y las funciones de transición por un rombo [23]. La figura 2.4 muestra un ejemplo concreto de esto. El diagrama de flujo comienza en el Estado 1, para pasar al estado dos depende de la función de transición representada por el rombo. Si esta función de transición arroja un uno lógico, entonces pasa al Estado 2, si no se mantiene en el Estado 1.

Diagrama MDS

El diagrama MDS (*Mnemonic Documented State diagram*) es una manera de representar estados y sus transiciones sin necesidad de definir sus funciones directamente. Mnemotécnico se refiere a que las funciones de transición no son especificadas, sino que se utiliza un nombre para representarlas, sin embargo, son expresiones booleanas. El Diagrama MDS es similar a lo que se conoce en los textos como diagrama ASM[3] (*Algorithmic State Machine*).

La figura 3.5 representa un diagrama MDS y un diagrama de flujo equivalente a su izquierda. Se puede ver que ambas poseen los mismos estados (Estado 1 y Estado 2) que transicionan por uso de una misma función.



Figura 2.4: Ejemplo de un Diagrama de Flujo

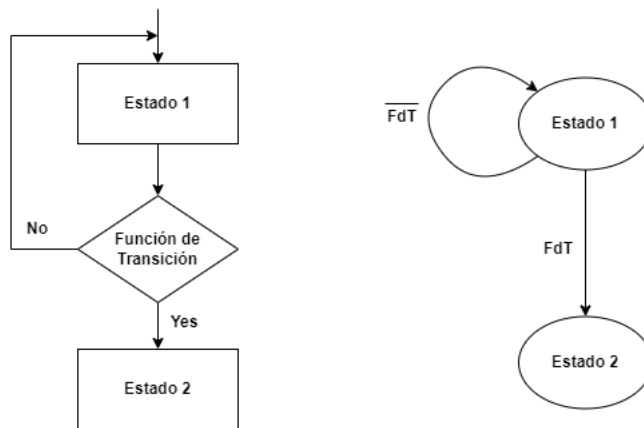


Figura 2.5: Ejemplo de dos Diagramas equivalentes. Diagrama de Flujo y MDS.

Este es un ejemplo directo de como convertir un diagrama de Flujo a un diagrama MDS. Lo siguiente sería convertir el diagrama MDS en un circuito lógico o en un programa HDL. Para esto basta con seguir la siguiente fórmula:

1. Utiliza binario para asignar los estados. En este ejemplo, en particular, como hay dos estados, basta solamente usar un símbolo de 0 o 1.
2. Utilizar *Flip-Flops* para representar los estados del sistema.
3. La función de transición al ser booleana se puede crear un circuito usando simples compuertas lógicas que sean capaz de representarlo.

2.2. Optimización de Máquinas de Estado

Existen diversos enfoques para optimizar máquinas FSM. Esta sección explica las distintas técnicas existentes para la optimización de estas máquinas y pone énfasis en las técnicas seleccionadas para diseñar un software especializado en su optimización. Estas técnicas son las siguientes:

- La simplificación del diagrama de estado: esta técnica implica reducir el número de estados necesarios para representar el comportamiento de la máquina, lo que a su vez puede disminuir la complejidad del diseño y el número de componentes requeridos.
- La codificación de estados consiste en asignar códigos únicos a cada estado de la máquina. La lógica utilizada varía según el formato que se emplee para representar los estados individuales.
- La minimización de funciones de transición: implica simplificar la función de transición de la máquina para reducir la complejidad y el número de componentes necesarios.
- Uso de herramientas de software: existen herramientas de software disponibles que permiten optimizar el diseño de máquinas de estados finitos. Estas herramientas incluyen síntesis de alto nivel y herramientas de simulación. Su utilización facilita el proceso de diseño y ayuda a obtener una implementación más eficiente y efectiva de las máquinas de estados finitos.

2.2.1. Optimización de un Diagrama MDS y Conversión a HDL

Para simplificar las máquinas de estado, representadas en un MDS se puede optar por disminuir la cantidad de estados de la máquina, simplificar las funciones de transición o, crear un circuito o archivo HDL en su menor expresión posible. Las técnicas actuales más conocidas son las siguientes:

- Identificación de estados equivalentes: se buscan estados que tengan el mismo comportamiento en cuanto a la respuesta a las entradas y la transición a otros estados.
- Eliminación de estados inalcanzables: se identifican los estados que no pueden ser alcanzados desde el estado inicial de la máquina o que no conducen a ningún estado final.
- Simplificación de transiciones: se buscan transiciones redundantes o innecesarias entre estados y se eliminan para simplificar el diagrama MDS.
- Asignación de códigos binarios: se asignan códigos binarios únicos a cada estado de la máquina. El número de bits necesarios para cada código dependerá del número total de estados. Por ejemplo, si hay ocho estados, se necesitan tres bits para representar cada estado ($2^3 = 8$).

2.2.2. Codificación de Estados

La técnica de codificación de estados implica los siguientes pasos:

- Identificación del número de estados: el primer paso es identificar el número total de estados que se necesitan para representar el comportamiento de la máquina.

- Verificación de la unicidad de los códigos: es importante asegurarse de que cada código binario asignado a un estado sea único y no se repita en ningún otro estado.

Al emplear códigos binarios únicos para representar cada estado, se logra una reducción en la cantidad de *Flip-Flops* necesarios para mantener el estado actual de la máquina FSM. Esto a su vez conlleva una disminución en la cantidad de lógica requerida para su implementación.

2.2.3. La Minimización de la Función de Transición

La minimización de la función de transición es una técnica utilizada en el diseño de circuitos digitales y máquinas de estados finitos para reducir el número de elementos lógicos necesarios para implementar la función de transición. La idea detrás de esta técnica es simplificar la función de transición, eliminando elementos lógicos redundantes.

La técnica de minimización de la función de transición implica los siguientes pasos:

Obtención de la tabla de verdad de la función de transición: el primer paso es obtener la tabla de verdad de la función de transición a minimizar.

Simplificación de la tabla de verdad: a continuación, se simplifica la tabla de verdad utilizando el método de Quine-McCluskey. El objetivo es reducir el número de términos y elementos lógicos necesarios para implementar la función de transición.

Obtención de la función de transición simplificada: después de simplificar la tabla de verdad, se obtiene la función de transición simplificada. Esta función puede ser representada utilizando una expresión booleana, un mapa de Karnaugh, o una tabla de transición simplificada.

Verificación de la equivalencia funcional: es importante verificar que la función de transición simplificada es equivalente a la función de transición original, es decir, que produce las mismas salidas para todas las posibles combinaciones de entradas. Esto se puede hacer comparando las tablas de verdad de la función de transición original y simplificada.

2.2.4. Minimización Máquina de Estados Finitos

Para minimizar la cantidad de estados basta con poder diferenciar si dos estados son equivalentes. Dos (o más) estados son equivalentes si la salida y el estado siguiente son los mismos, o equivalentes, para todas las entradas posibles:

1. Consistencia en el tiempo de entrada: que ambos estados reciban el mismo tiempo de entrada garantiza que los estados se activen y reaccionen de manera idéntica ante las mismas condiciones de entrada. Esto se refiere a si el tipo de entrada es síncrona o asíncrona y a la función de transición entre dos estados que entregue un mismo resultado para idénticas entradas.

2. Igualdad en las transiciones: ambos estados deben tener la misma cantidad de estados siguientes. Esto asegura que, independientemente de las entradas, ambos estados proporcionarán el mismo número de rutas o transiciones a otros estados.
3. Tipo de salida: la salida generada por ambos estados debe ser coherente en términos de su condicionalidad. Es decir, la salida de ambos estados debe ser o bien condicional (dependiendo de ciertas condiciones o entradas) o incondicional (siempre la misma, independientemente de las entradas). Esto se refiere a si la máquina FSM sigue una arquitectura de una máquina de Moore o de Mealy.

2.3. Estado del Arte

En esta sección se presenta el estado del arte de tecnología dedicada a la implementación y optimización de máquinas FSM. Además, se hace una revisión de los métodos planteados en la literatura para la optimización de recursos de un FPGA habiéndose implementado una FSM. Las máquinas FSM son fundamentales en la electrónica moderna. Desempeñan un rol importante en el diseño de lógica secuencial para los FPGA. Existen diversos software que se utilizan para la simulación e implementación de máquinas de estado finito. Los más destacados son software dedicados a diseñar circuitos y lógica en FPGA que son entregados por los mismos fabricantes de FPGA. A continuación se listan los softwares más relevantes.

2.3.1. Vivado

Es una herramienta de diseño de hardware desarrollada por Xilinx. Vivado utiliza técnicas avanzadas de optimización para mejorar la utilización de recursos en los FPGA, permitiendo un diseño más eficiente y una mejor utilización de la capacidad de procesamiento del dispositivo [26]. Vivado utiliza algoritmos de síntesis lógica avanzada que buscan optimizar la estructura lógica del diseño para minimizar el número de puertas lógicas y rutas de interconexión necesarias. Esto conduce a un diseño más compacto y a una reducción del consumo de recursos en el FPGA [4].

State Editor antes StateCAD

State Editor es un software desarrollado por Xilinx que se puede incorporar como otra herramienta de diseño de Vivado. Esta permite diseñar y crear máquinas de estado finito utilizando una plataforma gráfica como la mostrada en la figura 2.6. Permite además crear código de la máquina FSM diseñada en la plataforma gráfica y optimizar a nivel de código los recursos utilizados por la máquina.

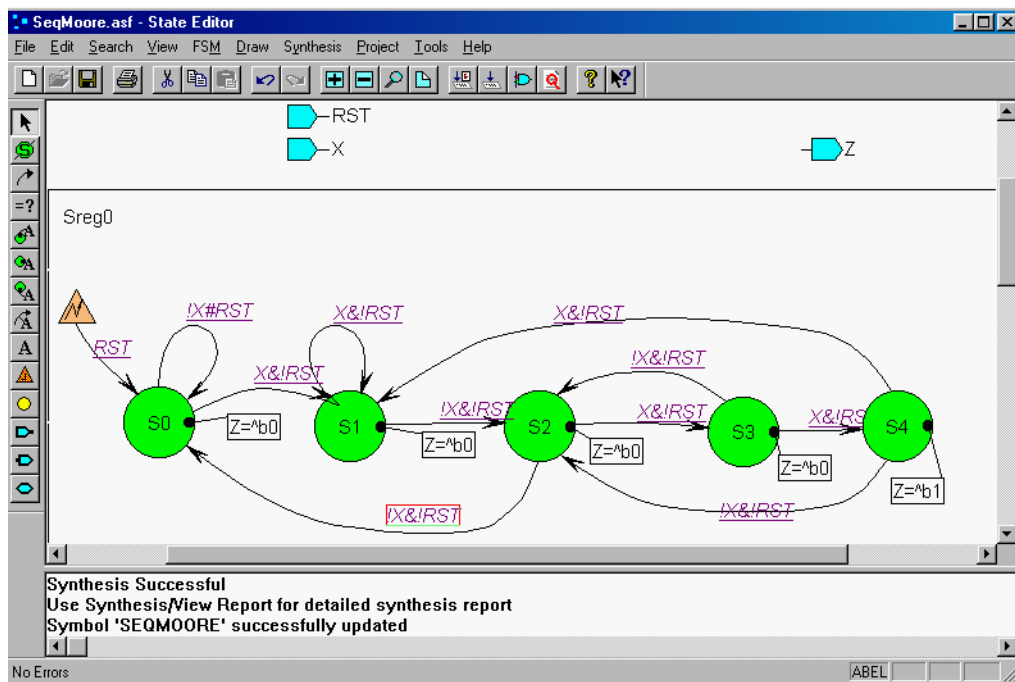


Figura 2.6: Software State Editor

2.3.2. Intel Quartus Prime

Es un software de diseño de hardware desarrollado por Intel que realiza la síntesis del diseño, es decir, traduce el código HDL o el esquema del circuito en una representación lógica y optimizada que puede ser implementada en el FPGA objetivo. La herramienta también entrega optimizaciones avanzadas para mejorar la utilización de recursos de hardware y el rendimiento del circuito. Realiza un mapeo del diseño lógico en los bloques específicos y elementos del FPGA que mejor se ajusten a las funciones lógicas utilizadas en el diseño. Si hay bloques lógicos que no están siendo utilizados, la herramienta los asigna a diferentes partes del diseño, lo que permite una mejor utilización de los recursos [12].

2.3.3. Lattice Diamond

Es un entorno de diseño integrado (IDE) desarrollado por Lattice Semiconductor. Diamond viene con un optimizador de energía y de lógica incorporado en su diseño [22].

Esta herramienta posee algoritmos de optimización de rendimiento para ayudar en el diseño y la implementación de sistemas basados en FPGA.

Herramientas de síntesis para convertir el código HDL en una implementación lógica en el hardware. Capacidad para optimizar el diseño, asignar recursos en el FPGA y realizar el enrutamiento de señales para una implementación exitosa. Cuenta soporte para la programación y configuración de dispositivos FPGA para su funcionamiento en tiempo real.

2.3.4. Herramientas de diseño de software

Además de las herramientas de diseño HDL entregadas por los fabricantes de FPGA, existen herramientas de diseño de programación que incorporan paquetes para la optimización de máquinas FSM. La más importante es Matlab Simulink, desarrollado por MathWorks. Es una herramienta utilizada para el modelado de sistemas en ingeniería. Su enfoque gráfico y basado en bloques proporciona una visualización intuitiva de los algoritmos, lo que facilita la creación de modelos complejos. Una de las características destacadas de Simulink es su capacidad de compilación interna, que permite transformar los modelos basados en bloques en código HDL para su posterior implementación en FPGA. Esta habilidad de generar código optimizado para la FPGA se traduce en una eficiente utilización de recursos lógicos y una mejora en los tiempos de respuesta del sistema. De esta manera, Matlab/Simulink se convierte en una valiosa herramienta para ingenieros que deseen diseñar y optimizar sistemas complejos en plataformas FPGA [16]. Matlab/Simulink acepta paquetes llamados Toolbox que son diseñados por empresas o por particulares con el objetivo de proporcionar diseños de código que agilicen la simulación e implementación de sistemas. El Toolbox por defecto para el diseño e implementación de máquinas FSM es StateFlow. Este Toolbox permite diseñar máquinas de estados finitos de manera gráfica, crear simulaciones y tests combinándolo con otros sistemas para así verificar la calidad de la máquina [15]. Además, permite compilar la máquina FSM a código C o HDL. La figura 2.7 muestra una aplicación básica de máquina FSM utilizando StateFlow.

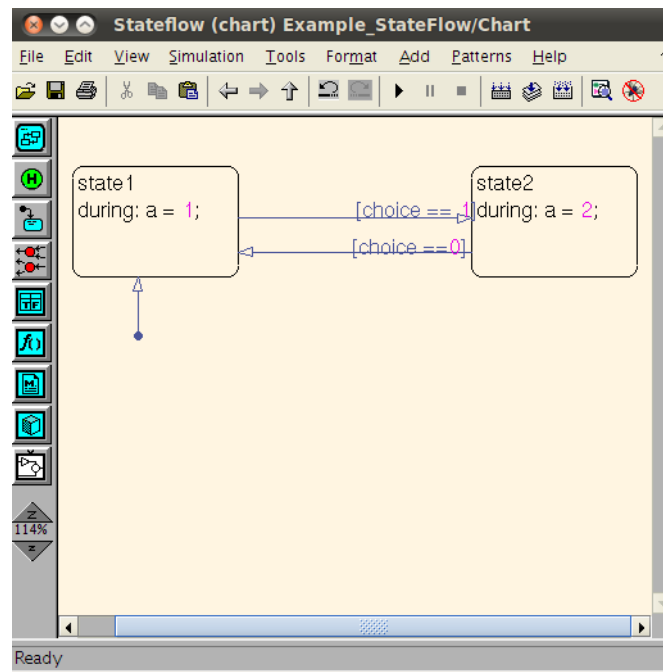


Figura 2.7: StateFlow Matlab/Simulink

Capítulo 3

Implementación del Software de Optimización de Máquinas de Estado

En este capítulo, se explora las máquinas FSM y las técnicas avanzadas de optimización, con un enfoque particular en el desarrollo de software para automatizar este proceso. El objetivo es proporcionar una visión completa de cómo se pueden mejorar la eficiencia de las máquinas FSM.

Además de abordar el enfoque mencionado, este capítulo también incluye una exposición detallada de la metodología de trabajo empleada. Aquí, se presenta el desarrollo lógico que condujo al diseño integral de todo el proceso. Este capítulo ofrece un marco práctico que guía el diseño, desarrollo e implementación de un software que automatiza la optimización de las máquinas FSM.

El marco práctico presentado no solo permitirá a los usuarios y programadores mejorar la eficiencia de sus sistemas, sino también reducir el tiempo y los recursos necesarios para llevar a cabo ajustes y mejoras.

Se detallan las etapas para desarrollo del software de optimización y se muestra la metodología detrás de cada elección y enfoque adoptado en el proceso de diseño. Desde la selección de una interfaz web implementada en JavaScript hasta la utilización de una API (*Application Programming Interface*) Flask para la comunicación con la página web, cada aspecto se integró para lograr un sistema efectivo.

3.1. Metodología de Trabajo

En esta sección, se proporciona el paso a paso y el desarrollo lógico que se utilizó para esta memoria. También se explica la relevancia de la optimización de estas máquinas para mejorar su eficiencia y rendimiento en aplicaciones prácticas.

Como se muestra en la figura 3.1 la metodología de este trabajo es estructurada linealmente.

- Comienza con una definición del problema donde se plantean los objetivos y los posibles resultados que se desean obtener. Esto se muestra en la introducción de esta memoria de título, sección 1.3.
- Una revisión bibliográfica donde se analiza el marco teórico necesario para comprender las máquinas de estado finito y sus diferentes técnicas de optimización. Además, un análisis del estado del arte de software usados para la optimización de máquinas FSM para la utilización en FPGA.
- Se seleccionan las técnicas de optimización que son utilizadas para la implementación de un software de optimización de máquinas FSM. Esto se muestra en la sección 3.2.
- Se desarrolla un primer software de optimización de máquinas FSM. Este software debe ser capaz de utilizar las técnicas de optimización definidas en la sección 3.2.
- Se hacen pruebas para validar el código desarrollado donde se demuestre que ambas máquinas son equivalentes. Estas pruebas buscan demostrar que para cada igual combinación de entradas aplicadas a ambas máquinas, las salidas son iguales.
- Se desarrolla un software capaz de traducir una máquina de estados finitos en un archivo de código con el formato de *SystemVerilog*.
- Una vez implementado y demostrado el software básico de optimización de máquinas FSM, se desarrolla una interfaz gráfica que permita a cualquier usuario, sin importar su nivel técnico.
- Finalmente, se muestran las pruebas hechas en un FPGA real.

Esta metodología proporciona un enfoque sistemático y estructurado para el desarrollo y evaluación de técnicas de optimización de FSM y su implementación en un FPGA a través de una API y una interfaz gráfica.

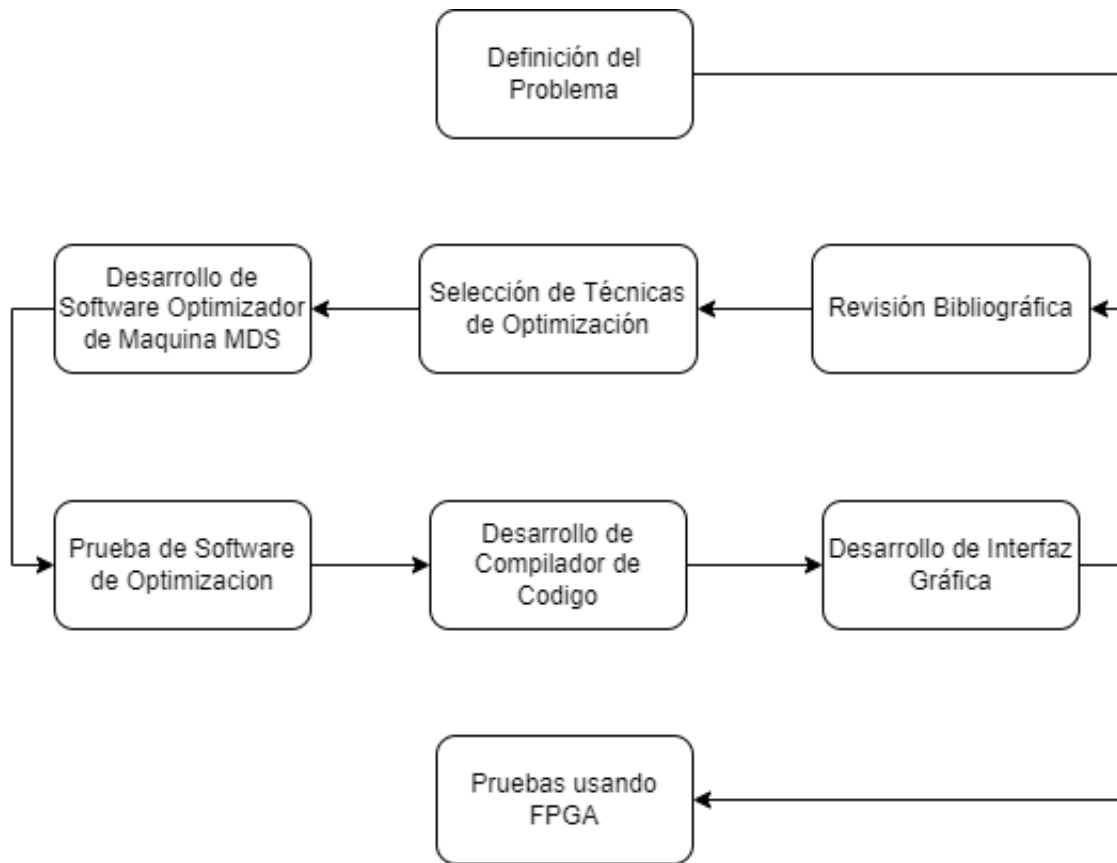


Figura 3.1: Diagrama de metodología de trabajo.

3.2. Algoritmo de Optimización

Como se explicó en el capítulo dos de esta memoria, la optimización de una máquina de estados finitos se va a abordar de tres maneras diferentes. La primera, y la más importante, es la minimización de estados. La segunda, es minimización de funciones de transición y la tercera, es la codificación de estados en códigos binarios.

3.2.1. Minimización de Estados

El primer algoritmo tomado en cuenta es la Minimización de Estados. Para esto, se va a considerar que las funciones de transición equivalentes vienen representadas por un código único. Es decir, a cada función se le otorga un nombre sin considerar el verdadero valor de la función. Dos funciones equivalentes deben recibir el mismo nombre. El algoritmo implementado está inspirado en el algoritmo de minimización de Hopcroft.

El algoritmo de Hopcroft es un algoritmo utilizado para minimizar máquinas FSM, lo que significa que toma una máquina FSM como entrada y produce una máquina FSM equivalente. El algoritmo de Hopcroft busca la equivalencia de estados, es decir, dos estados son equivalentes si y solo si conducen a estados finales o no finales con las mismas transiciones.

El algoritmo de Hopcroft funciona de la siguiente manera[10]:

1. Inicialización: divide los estados de la máquina FSM en dos grupos: estados finales y estados no finales. Agrega estos dos grupos a una cola de particiones iniciales.
2. Bucle principal: mientras haya una partición que pueda dividirse, se extrae una partición P de la cola. Para cada símbolo del alfabeto de entrada se divide P en subconjuntos de estados que tienen la misma transición en el símbolo dado. Si alguno de los subconjuntos resultantes tiene al menos un estado de ambos grupos (final y no final), agregar esos subconjuntos a la cola.
3. Construcción de máquina FSM mínima: una vez que la cola esté vacía y no haya más particiones para dividir, la partición final representa el conjunto de estados mínimos. Construye una nueva máquina FSM con las clases de equivalencia como estados. Actualiza las transiciones de la máquina FSM mínima utilizando las transiciones de la original entre los estados de cada clase de equivalencia.
4. Asignación de estados finales: un estado mínimo será final si contiene al menos un estado final de la máquina FSM original. Para hacer el software, se utiliza una nueva estructura de datos llamada "State". Para crear esta estructura se debe precisar el estado actual, los siguientes, los valores de entradas y las salidas que tiene.

Para implementar el algoritmo de Hopcroft se utiliza una estructura de datos capaz de representar cada nodo. Esta estructura de datos se llama "Nodo", y posee "Estado Actual", "Estado Siguiente", "Outputs" y "Estados Iguales".

Considerando lo anterior, en términos simples, lo que hace el algoritmo es:

1. Separar los estados si tienen salidas condicional o incondicional.
2. Separar los estados según la cantidad de estados siguientes que tengan.
3. Separar los estados según la función de transición que tengan.
4. Revisar cada uno de estos subsistemas si, según las entradas, los estados siguientes son equivalentes.
5. Guardar los estados iguales y crear un nuevo estado como la combinación de los nombres de los estados equivalentes.

3.2.2. Minimización de Funciones de Transición.

Las funciones de transición son funciones booleanas que representan las condiciones que se tienen que dar para que un estado A pase a un estado B. Las funciones booleanas se pueden minimizar, reduciendo así el tiempo de ejecución y la energía gastada por el circuito. Para hacer esta reducción, esta memoria utiliza el método de Quine-McCluskey[14]. Sin embargo el método de Quine-McCluskey permite minimizar minterminos considerando solo los valores que se encuentran en la ecuación lógica entregada, es decir, que si la cantidad de entradas

es mayor a los minitérminos que se entregaron, la optimización no es correcta. Por esto mismo, se entrega una recomendación de las funciones de transferencia en el código, pero no se implementa directamente.

3.2.3. Representación en Códigos Binarios

La representación en códigos binarios se refiere a nombrar cada estado con un número binario, así poder representar los estados usando *Flip-Flops*. Esto indicaría que para representar 2^a estados solo se requieren a *Flip-Flops*.

Para minimizar el gasto energético hay que minimizar las distancias de Hamming entre dos estados adyacentes[19].

La distancia de Hamming es una métrica utilizada para medir la diferencia entre dos cadenas de igual longitud.

La distancia de Hamming se calcula comparando posición por posición los caracteres de las dos cadenas y contando cuántas posiciones tienen caracteres diferentes. Es importante destacar que la distancia de Hamming solo es válida para cadenas de igual longitud.

La figura 3.2 muestra la distancia respectiva de los 8 números de 3 bits representados en un cubo [13].

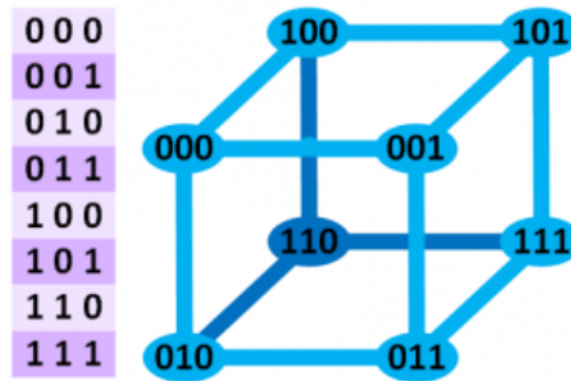


Figura 3.2: Representación gráfica de distancias de Hamming de 3 bits.

3.3. Implementación de la Solución o Herramienta

Para el software de optimización se desarrolló una página web capaz de recibir la máquina FSM como un archivo de texto o diseñarla directamente usando la interfaz. Esta máquina debe ser mostrada en forma de grafo[2] indicando entradas, salidas y estados. Un botón ubicado en la parte inferior del grafo permite optimizarlo. Al optimizar el grafo se puede ver el grafo optimizado y el anterior, además de un botón para descargar el archivo en *System Verilog*. El *Back-End* se hace con una aplicación Flask API que recibe la información

proveniente de Javascript en formato JSON. Este devuelve el nodo actualizado en formato JSON y un archivo de texto en formato *SystemVerilog*.

La arquitectura del sistema se puede observar en la figura 3.3. El *Front-End* que viene siendo solamente la interfaz gráfica, realiza pedidos HTTP a la API que posee 4 diferentes funciones. Cada una de esas funciones se encarga de un fragmento de la optimización o el funcionamiento de la API.

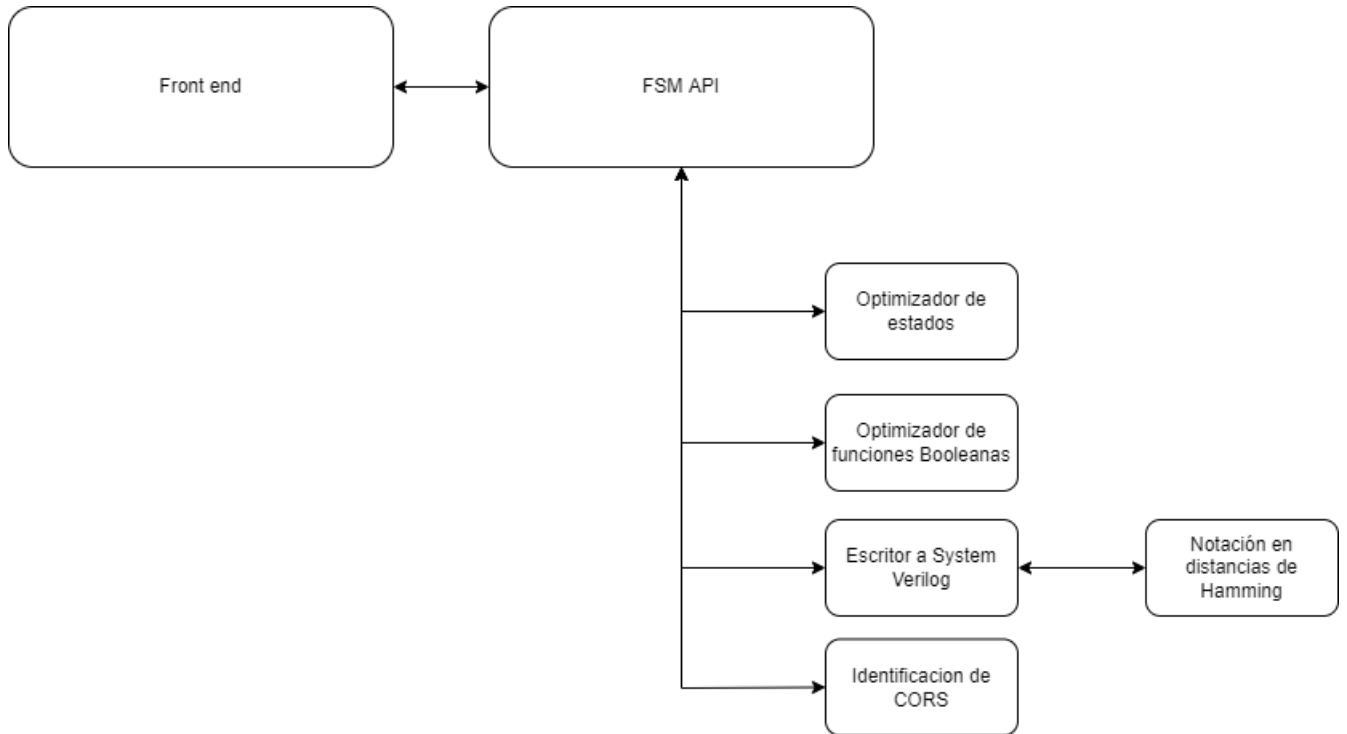


Figura 3.3: Desarrollo software

3.3.1. Interfaz Gráfica

La página web se caracteriza por una barra de navegación con 3 diferentes ventanas. La primera es "Inicio", la segunda es "Guia de uso" y la tercera es "Optimizador".

En la ventana de inicio, denominada "Inicio", los usuarios encontrarán una presentación de la aplicación. Esta ventana los invita a explorar y probar todas las funcionalidades y características que ofrece la aplicación. Además de brindar una visión general de la aplicación, también destaca los beneficios que puede proporcionar a los usuarios, simplificando y optimizando sus tareas diarias. A través de enlaces y botones, los usuarios podrán acceder a otras secciones y explorar más a fondo la aplicación.

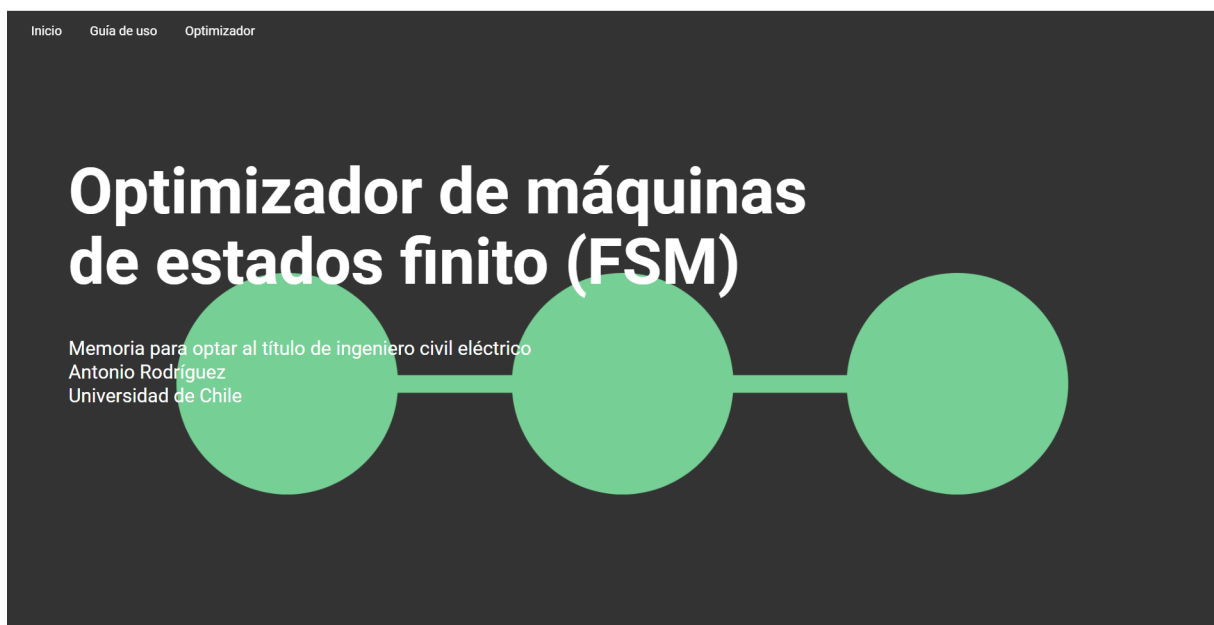


Figura 3.4: Página de inicio

La segunda página de la ventana de inicio tiene como objetivo principal invitar cordialmente a hacer uso del algoritmo de optimización. Presenta un título llamativo que busca captar la atención del usuario, acompañado de un botón que redirige de manera directa al mencionado algoritmo.

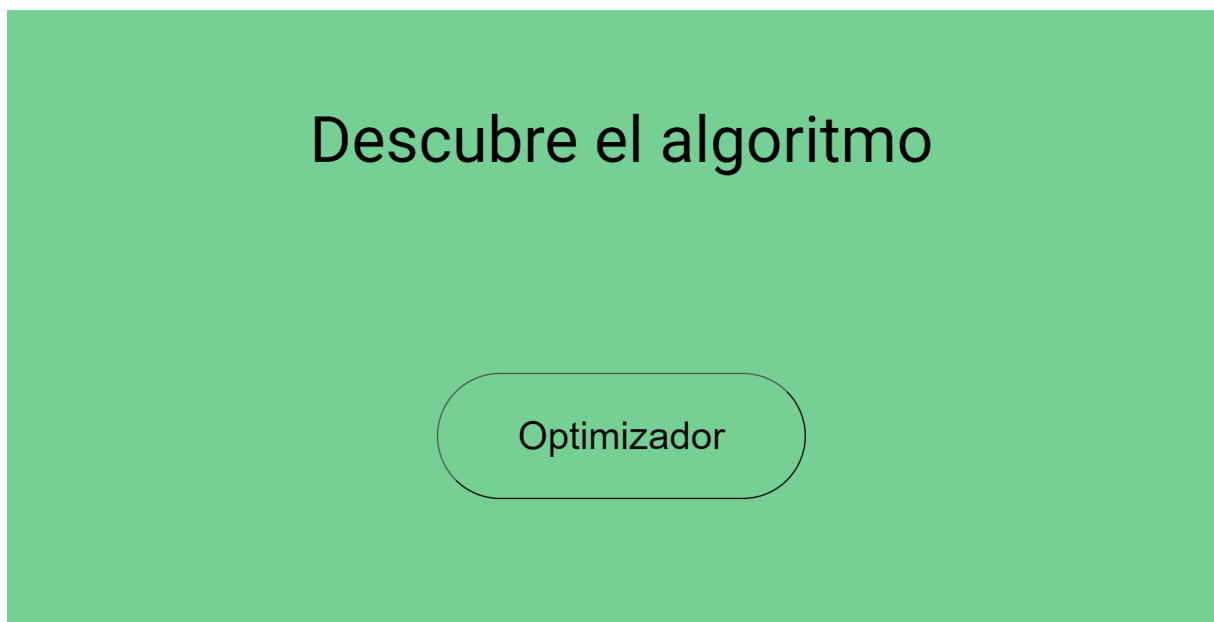


Figura 3.5: Página de inicio - Sub-página

La siguiente ventana, "Guía de uso", ofrece a los usuarios un tutorial completo de la aplicación. Aquí, los usuarios encontrarán una guía paso a paso que los acompañará a lo largo de todas las funcionalidades y características disponibles. Este tutorial detallado les

permitirá familiarizarse rápidamente con la aplicación y comprender cómo aprovechar al máximo todas las opciones de optimización que se ofrecen. Desde la configuración inicial hasta el uso avanzado de las herramientas, los usuarios encontrarán respuestas y orientación en esta sección.

La figura 3.6 es la primera página que se ve al hacer clic en "Guía de uso". Esta página explica qué hace el software y presenta el concepto de estado mnemotécnico documentado.

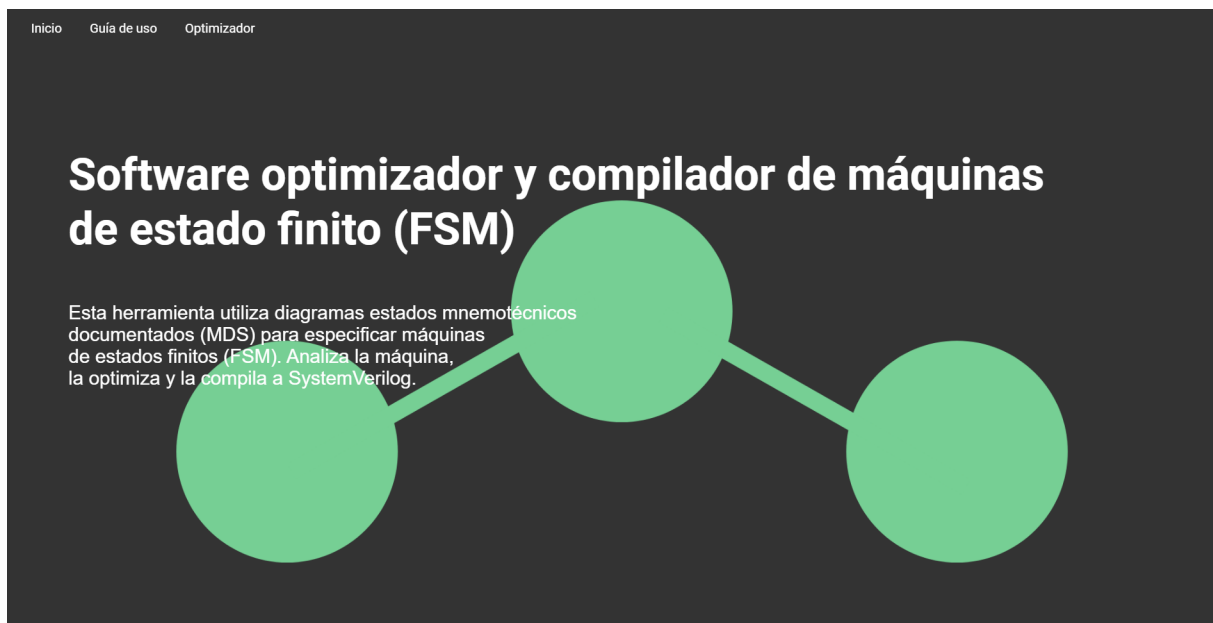


Figura 3.6: Ventana "Guía de uso".

La figura 3.7 es la segunda página que se ve al hacer clic en "Guía de uso". Esta página muestra un tutorial para entender la teoría necesaria para utilizar el algoritmo de optimización. El primer rectángulo muestra un diagrama de flujo que inicia en el "Estado 1". Si la función de transición entrega un "Sí" pasa al "Estado 2" mientras que si entrega un "No" se mantiene en el "Estado 1". El Segundo rectángulo muestra el diagrama MDS equivalente al diagrama de flujo del primer rectángulo. Pasa del "Estado 1" al "Estado 2" pero mostrando la función de transición en las flechas del diagrama. El tercer rectángulo muestra como describir el diagrama MDS de forma que el computador pueda leerlo. Este utiliza la notación en formato de archivo JSON. El primer nodo se denomina "A". Este tiene una transición al nodo "B" con la función de transición "f1" y se mantiene en "A" con la función de transición "f2". El segundo nodo "B" no tiene transiciones.

Diagrama mnemotécnico documentado y máquina FSM

Instrucciones:

1. Escribir el Diagrama de Flujo como se muestra en el primer cuadrado
2. Escribir el Diagrama usando Diagramas mnemotécnicos como se muestra en el segundo cuadrado
3. Utilizar la version escrita del diagrama mnemotécnico como se muestra en el tercer cuadrado directamente en el software

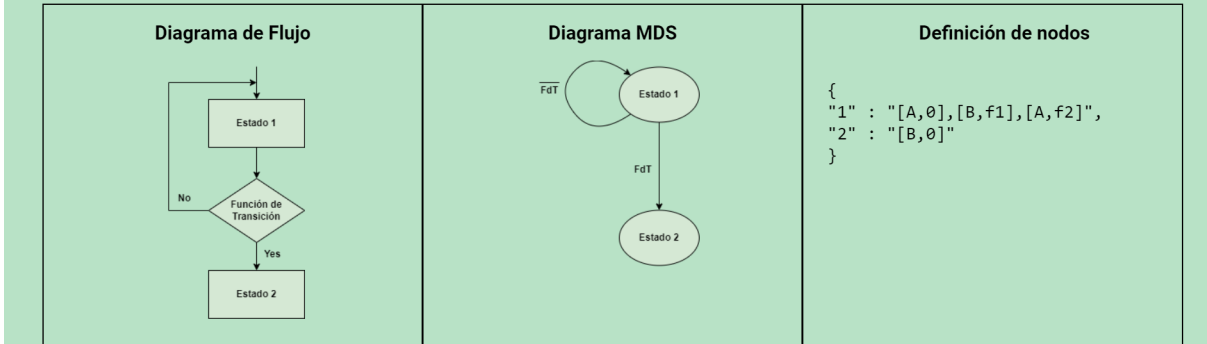


Figura 3.7: Ventana "Guía de uso Tutorial".

La tercera ventana, "Optimizador", es el núcleo de la aplicación. Aquí, los usuarios encontrarán una amplia gama de herramientas y funciones diseñadas para optimizar máquinas FSM. Con una interfaz intuitiva y poderosas herramientas, el "Optimizador" les proporciona las soluciones necesarias para mejorar su eficiencia y alcanzar sus objetivos de manera más efectiva.

La página de "Optimizador" mostrado en la figura 3.8. Presenta dos botones para subir el diagrama MDS. El primer botón permite abrir la pestaña de "Forma", la cual se puede observar en la figura 3.9. El segundo botón permite abrir la pestaña "File", la cual se observa en la figura 3.11.

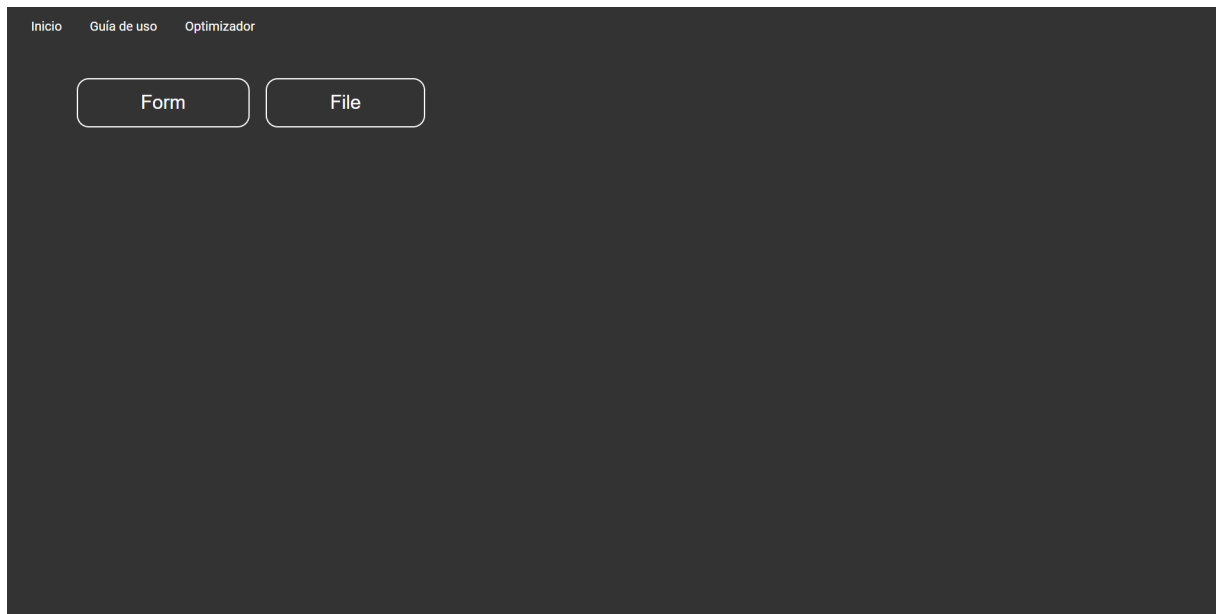


Figura 3.8: Ventana "Optimizador".

La figura número 3.9 muestra la pestaña "Forma". Esta permite agregar nodos uno por uno y dibujar una máquina MDS en la pantalla. Se ve la primera forma que permite agregar dos valores. La de la izquierda permite agregar el nombre del nodo actual, mientras la de la derecha permite agregar los nodos siguientes. La segunda forma también tiene dos entradas. El primero permite agregar el nombre de la función de transición y el segundo los mini-términos.

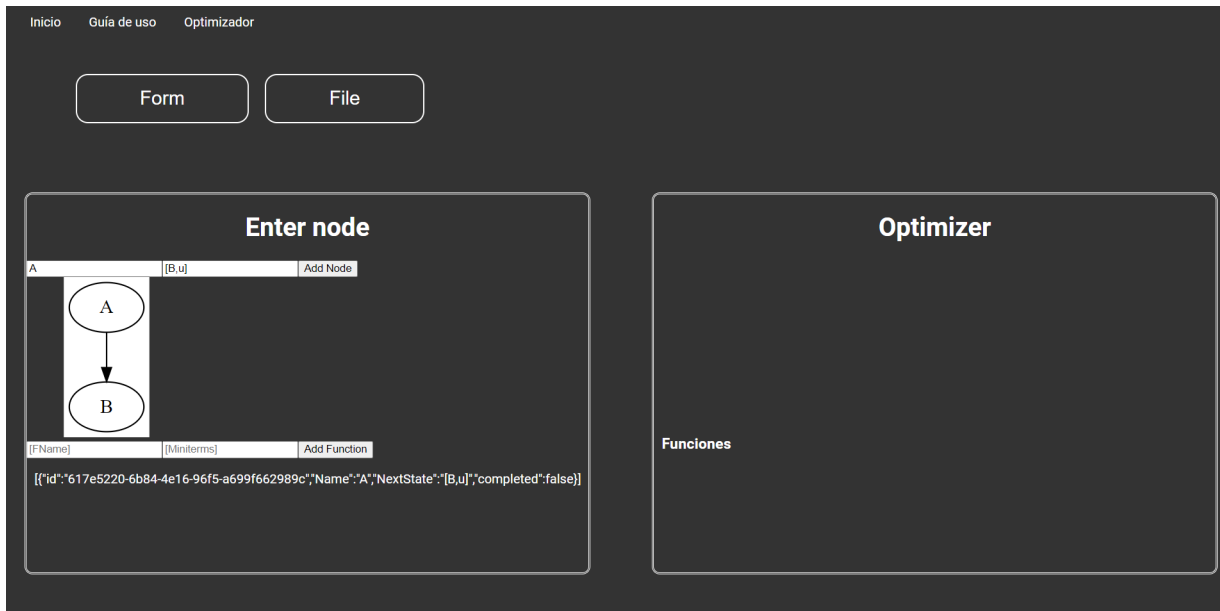


Figura 3.9: Pestaña "Forma" de la ventana "Optimizador".

La figura número 3.11 muestra la pestaña "File". Esta permite agregar la máquina MDS directamente desde un archivo JSON [20]. Al archivo JSON debe tener formato de la figura

3.10:

```
[{
  "1" : "[Nodo1,Salida],[estadoSiguiete1,funcionTransferencia1],[estadoSiguiete2,
  ....],[.....]"
  "2" : "[Nodo2,Salida],[estadoSiguiete1,funcionTransferencia1],[estadoSiguiete2,
  ....],[.....]"
}]
{
  funcion: [miniterminos]
}
```

Figura 3.10: Descripción de los nodos en JSON.

En la figura 3.11 se puede observar un archivo ya cargado que muestra un diagrama MDS de una máquina FSM de 3 nodos. Las funciones son *uz*, *zu* y *zz*, con mini-términos "01", "12" y "23", respectivamente. A la derecha, se puede apreciar el grafo optimizado con las funciones booleanas recomendadas expresadas.

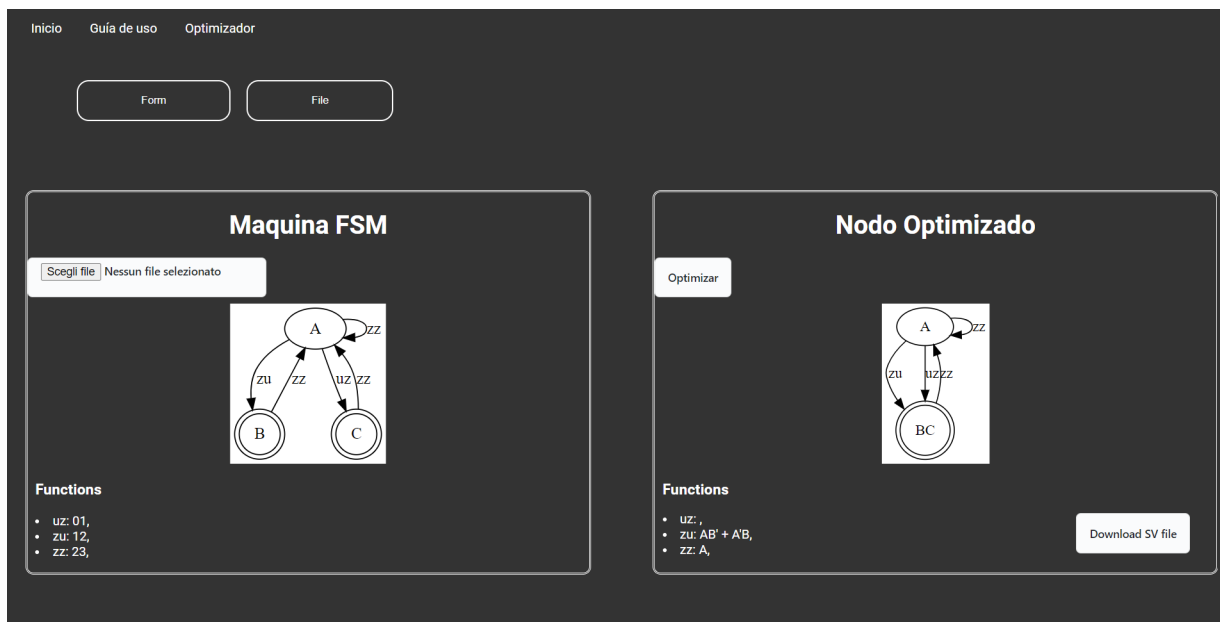


Figura 3.11: Ejemplo de dos Diagramas equivalentes. Diagrama de Flujo y MDS.

3.3.2. Diseño e Implementación de la API Flask

La implementación de la API que proporciona la funcionalidad esencial del algoritmo de optimización de grafos para máquinas FSM, se implementó utilizando Flask, un *Framework* de Python para el desarrollo de aplicaciones web [8].

La API de Flask se desarrolló siguiendo los principios de una arquitectura RESTful [5], asegurando que las funciones del algoritmo sean asequibles a través de solicitudes HTTP estandarizadas, como GET y POST. Esto también facilita la interoperabilidad y escalabilidad de la API.

Las funciones del algoritmo de optimización de las máquinas FSM se muestran a través de varias rutas de la API, cada una de las cuales corresponde a una funcionalidad específica del algoritmo. Aquí hay un desglose de las principales rutas y sus funciones asociadas: **insertData()**: Esta ruta recibe una descripción de un grafo de FSM como entrada en una solicitud POST. La entrada se procesa a través del algoritmo de optimización, que minimiza la cantidad de estados y optimiza las funciones de transición utilizando el algoritmo de Quine-McCluskey.

getData(): esta ruta acepta una máquina de estado finito optimizada y genera un archivo *System Verilog* correspondiente. El archivo se guarda en el servidor y se devuelve una ruta para descargar el archivo. Además, se entrega el diagrama MDS en formato JSON al *Front-End*.

Para garantizar la seguridad de los datos de entrada y salida, la API incluye medidas de seguridad como la validación de entradas y la gestión de sesiones seguras utilizando CORS [1]. Solo direcciones conocidas por la API pueden utilizar la API.

En resumen, la API Flask desempeña un papel crucial en la implementación del algoritmo de optimización de FSM. Al proporcionar una interfaz de usuario accesible y segura, facilita la interacción con el algoritmo y la obtención de resultados optimizados. La arquitectura RESTful de la API garantiza que sea escalable, mantenible y fácil de integrar con otras aplicaciones y servicios.

Capítulo 4

Pruebas y Resultados del Software de Optimización de Máquinas FSM

En este capítulo, se presentan las pruebas y resultados obtenidos del software de optimización de máquinas de estado finito desarrollado.

El enfoque se centra en el proceso de pruebas y los resultados que se obtuvieron al aplicar el software de optimización de máquinas de estado finito a diferentes casos de estudio. Se exploraron varias métricas de rendimiento, como la minimización del número de estados, la reducción del número de transiciones y la optimización de la latencia y el consumo de energía. También se analizaron los efectos de las diferentes estrategias de optimización, y se realizaron comparaciones con los métodos tradicionales de diseño de máquinas de estado finito.

La estructura del capítulo se organiza de la siguiente manera. En primer lugar, se presenta el software utilizado, la máquina FSM optimizada y la FPGA escogida, junto con el entorno de pruebas utilizado, que incluye las herramientas de simulación y síntesis.

A continuación, se presentan los casos de estudio seleccionados, que abarcan una variedad de aplicaciones y niveles de complejidad.

Posteriormente, se analizan y discuten los resultados obtenidos en cada caso de estudio, destacando las mejoras logradas con el software de optimización. Este análisis es fundamental para demostrar la eficacia y el valor del software en el diseño de sistemas digitales avanzados.

Finalmente, se resumen las conclusiones y se discuten las perspectivas futuras de desarrollo, sentando las bases para futuras mejoras y avances en esta área de investigación.

Con este capítulo, se valida y evalúa el software de optimización de máquinas de estado finito, demostrando su eficacia y su potencial para el diseño de sistemas digitales más eficientes y sofisticados.

4.1. Software usado para validar la máquina FSM optimizada y FPGA utilizada

Quartus Prime y ModelSim son dos herramientas esenciales en el diseño y simulación de circuitos digitales utilizando dispositivos programables, como FPGA. Ambas son parte de la suite de diseño proporcionada por Intel (anteriormente Altera) para el desarrollo de sistemas digitales complejos.

Quartus Prime es un software de diseño de sistemas digitales desarrollado por Intel FPGA. Es utilizado para implementar circuitos lógicos en FPGA.

Quartus prime tiene un editor de diseño gráfico y HDL (*Hardware Description Language*) que facilita la creación y visualización de circuitos digitales. Soporte para varios lenguajes de descripción de hardware, como VHDL y Verilog y SystemVerilog.

Quartus Prime permite la visualización del código HDL escrito en *SystemVerilog* en RTL (*Register Transfer Level*). Esta es una representación gráfica del comportamiento del código representado directamente con *Flip-Flops* y compuertas lógicas. Además, Quartus Prime entrega un resumen de los recursos utilizados en la FPGA. Esto permite comparar las dos máquinas equivalentes (optimizada y sin optimizar) no solamente estructuralmente, sino que también en cuanto a los recursos utilizados por la FPGA.

Quartus Prime requiere la configuración de una FPGA específica con el objetivo de compilar su código. Para esta memoria se utilizó la tarjeta de desarrollo d1-soc que utiliza el FPGA Cyclon 5 de Intel FPGA mostrada en la figura 4.1. Este FPGA contiene 32070 ALM (Adaptive Logic Module) y 457 pins [11]. Las ALM se refieren a las unidades básicas de lógica de las FPGA. Estas contienen LUT programables y *Flip-Flops* o *Latch*.

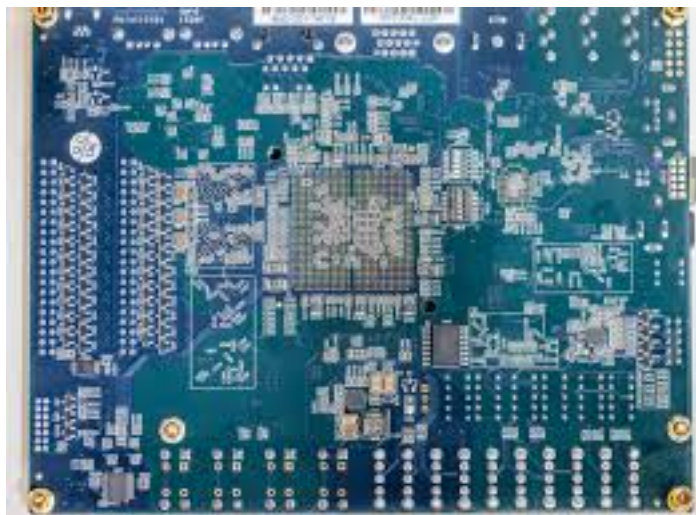


Figura 4.1: FPGA Cyclon V, Tarjeta de desarrollo DE1-SoC

Por otro lado, *ModelSim* permite comprobar que ambas máquinas entregan el mismo resultado a los mismos estímulos de señales[7].

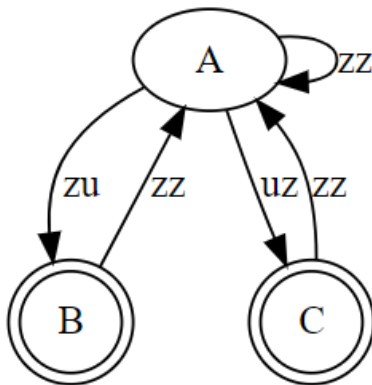
4.2. Optimización de máquina FSM de tres nodos

Para mostrar un ejemplo concreto de máquina de estado finito es y comparar su versión original con la versión optimizada obtenida a través del software de optimización, se muestra la optimización de una máquina de tres nodos como la que se muestra en la figura 4.3 donde se ingresó un archivo JSON mostrado en la figura 4.2.

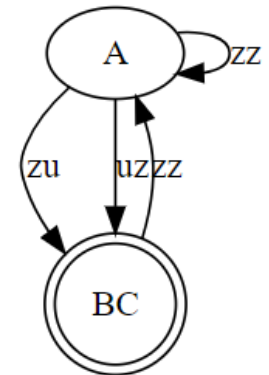
```
{
"1" : "[A,0],[B,zu],[C,uz],[A,zz]",
"2" : "[B,1],[A,zz]",
"3" : "[C,1],[A,zz]"
} [caption={caption text here}, label={mylabel}]
```

Figura 4.2: Máquina FSM de una máquina expendedora

Las funciones "zu", "zz" y "uz" no se especificaron en este ejemplo. Se utilizaron simplemente entradas *Booleanas*.



(a) Máquina FSM de 3 nodos



(b) Máquina FSM optimizada

Figura 4.3: Diagrama MDS de la máquina FSM de 3 nodos y su optimización.

4.2.1. Compilación de Códigos para máquina FSM de tres nodos y su optimización

La figura 4.4 muestra el circuito RTL de la máquina FSM de tres nodos y la figura 4.5 la máquina FSM de tres nodos optimizada. Se puede notar que tienen las mismas entradas y salidas, pero que, sin embargo, no tienen la misma cantidad de lógica.

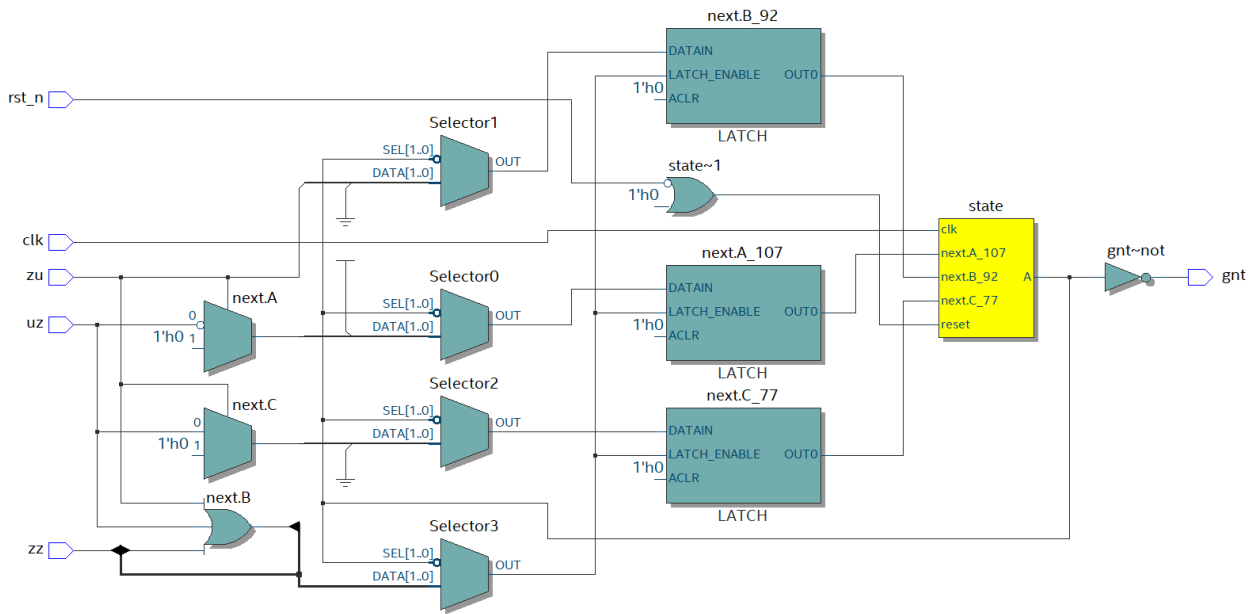


Figura 4.4: Circuito RTL de máquina FSM de 3 nodos

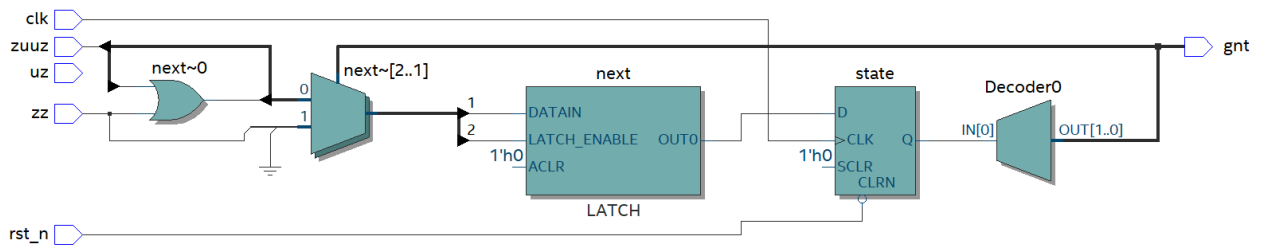


Figura 4.5: Circuito RTL de máquina FSM de 3 nodos optimizada

4.2.2. Resultados

Después de compilar un diseño en Quartus Prime, se obtiene un resumen de la síntesis y el proceso de implementación que incluye información importante sobre el diseño y los recursos utilizados. La figura 4.6 presenta el resumen obtenido luego de compilar la máquina FSM de 3 nodos presentada en la figura 4.3a. La figura 4.7 presenta el resumen obtenido luego de compilar la máquina FSM de 3 nodos optimizada presentada en la figura 4.3b.


| Flow Summary | |
|--|--|
|  <<Filter>> | |
| Flow Status | Successful - Thu Jul 20 16:10:21 2023 |
| Quartus Prime Version | 22.1std.1 Build 917 02/14/2023 SC Lite Edition |
| Revision Name | opt3 |
| Top-level Entity Name | opt3 |
| Family | Cyclone V |
| Device | 5CSEMA5F31C6 |
| Timing Models | Final |
| Logic utilization (in ALMs) | 3 / 32,070 (< 1 %) |
| Total registers | 1 |
| Total pins | 6 / 457 (1 %) |
| Total virtual pins | 0 |
| Total block memory bits | 0 / 4,065,280 (0 %) |
| Total DSP Blocks | 0 / 87 (0 %) |
| Total HSSI RX PCSs | 0 |
| Total HSSI PMA RX Deserializers | 0 |
| Total HSSI TX PCSs | 0 |
| Total HSSI PMA TX Serializers | 0 |
| Total PLLs | 0 / 6 (0 %) |
| Total DLLs | 0 / 4 (0 %) |

Figura 4.6: Resumen de los resultados del código compilado de la maquina FSM de 3 nodos.


| Flow Summary | |
|--|--|
|  <<Filter>> | |
| Flow Status | Successful - Thu Jul 20 16:30:50 2023 |
| Quartus Prime Version | 22.1std.1 Build 917 02/14/2023 SC Lite Edition |
| Revision Name | opt3Done |
| Top-level Entity Name | opt3Done |
| Family | Cyclone V |
| Device | 5CSEMA5F31C6 |
| Timing Models | Final |
| Logic utilization (in ALMs) | 2 / 32,070 (< 1 %) |
| Total registers | 1 |
| Total pins | 6 / 457 (1 %) |
| Total virtual pins | 0 |
| Total block memory bits | 0 / 4,065,280 (0 %) |
| Total DSP Blocks | 0 / 87 (0 %) |
| Total HSSI RX PCSs | 0 |
| Total HSSI PMA RX Deserializers | 0 |
| Total HSSI TX PCSs | 0 |
| Total HSSI PMA TX Serializers | 0 |
| Total PLLs | 0 / 6 (0 %) |
| Total DLLs | 0 / 4 (0 %) |

Figura 4.7: Resumen de los resultados del código compilado de la maquina FSM de 3 nodos optimizada.

4.3. Ejemplo de una Máquina Expendedora

El controlador desarrollado para la máquina expendedora de chicles se diseñó específicamente como una prueba de concepto para evaluar la eficacia y funcionalidad del código del optimizador de máquinas de estado finito (FSM). Al implementar el controlador con ocho estados y tres funciones de transición ("zu", "zz" y "uz"), se buscó simular un escenario realista de interacción entre el usuario y la máquina, donde se depositan monedas y se realiza una compra. La complejidad del controlador permitió poner a prueba la capacidad del optimizador para reducir el número de estados, minimizar las transiciones y optimizar los recursos necesarios para la implementación en una FPGA. La máquina expendedora de chicles proporcionó un entorno práctico para evaluar cómo el optimizador podía mejorar la eficiencia y el rendimiento del diseño de máquinas de estados finitos en aplicaciones reales.

El controlador desarrollado para la máquina expendedora de chicles es un sistema de ocho estados diseñado para gestionar la interacción con los usuarios y el manejo de monedas para la compra de chicles. La máquina acepta monedas de 100 y 50 pesos, y el precio de un chicle es de 150 pesos. Una característica importante es que la máquina no proporciona cambio, pero tiene la capacidad de recordar el dinero excedente si el valor de las monedas depositadas supera el costo del chicle.

El controlador consta de ocho estados distintos que representan diferentes etapas del proceso de compra y control de las monedas:

Estado Inicial A: la máquina se encuentra en el estado inicial, aún no ha recibido ninguna moneda y está lista para recibir una.

Estado B: en este estado, la máquina recibió una moneda de 50 pesos, el controlador pasará al Estado D si recibe una segunda moneda de 50 pesos, y si se inserta una moneda de 100 pesos, pasará al Estado E.

Estado C: cuando se ha ingresado una moneda de 100 pesos, el controlador pasará al Estado G si recibe una segunda moneda de 100 pesos, y si se inserta una moneda de 50 pesos, pasará al Estado F.

Estado D: solo se puede pasar desde el estado B. El controlador pasará al Estado I si recibe una moneda de 100 pesos, y si se inserta una moneda de 50 pesos, pasará al Estado H.

Estado E: solo se puede pasar desde el estado B. El controlador pasará al Estado A inmediatamente si no se inserta ninguna moneda.

Estado F: solo se puede pasar desde el estado B. El controlador pasará al Estado A inmediatamente si no se inserta ninguna moneda.

Estado G: solo se puede llegar desde el estado C. El controlador pasa al estado B si no se ingresa nada.

Estado H: solo se puede pasar desde el estado D. El controlador pasará al Estado A inmediatamente si no se inserta ninguna moneda.

Estado I: solo se puede llegar desde el estado D. El controlador pasa al estado B si no se ingresa nada.

```
{
  "1" : "[A,0],[B,zu],[C,uz],[A,zz]",
  "2" : "[B,0],[D,zu],[E,uz],[B,zz]",
  "3" : "[C,0],[F,zu],[G,uz],[C,zz]",
  "4" : "[D,0],[H,zu],[I,uz],[D,zz]",
  "5" : "[E,1],[A,zz]",
  "6" : "[F,1],[A,zz]",
  "7" : "[G,1],[B,zz]",
  "8" : "[H,1],[A,zz]",
  "9" : "[I,1],[B,zz]"
}
```

Figura 4.8: Máquina FSM de una máquina expendedora

Las funciones de transición del controlador se denominan "zu", "zz" y "uz" para denotar diferentes situaciones de entrada y salida entre los estados. Estas son equivalentes a las señales que vienen de los sensores necesarios para detectar las monedas ingresadas a la máquina expendedora. "uz" representa la transición desde el estado actual al siguiente estado cuando se ingresa una moneda de 100 pesos. "zu" es la transición que ocurre cuando se ingresa una moneda de 50 pesos, y "zz" es la transición que se activa cuando no se ha ingresado ninguna moneda adicional, es decir, la máquina se mantiene en el mismo estado.

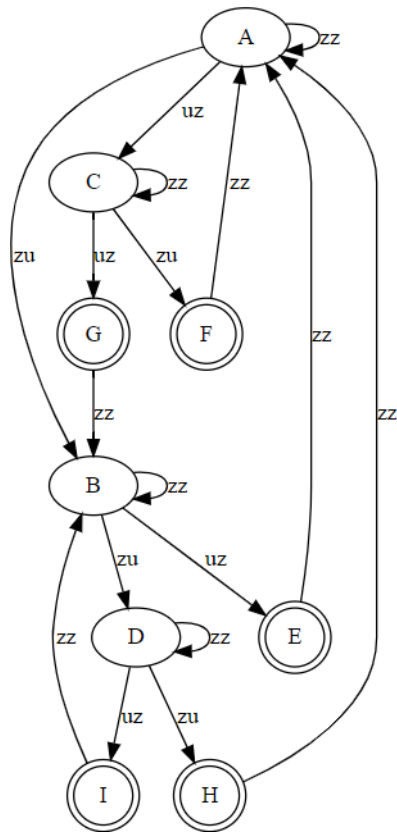


Figura 4.9: Máquina FSM de una máquina expendedora

Se puede ver en la figura 4.10 que a diferencia de la figura 4.9 que los estados "E", "F" y "H" son equivalentes, ya que los tres tienen solo una transición y pasan directamente al estado A. Los estados "G" e "I" son equivalentes igualmente, ya que solo tienen una transición al estado "B".

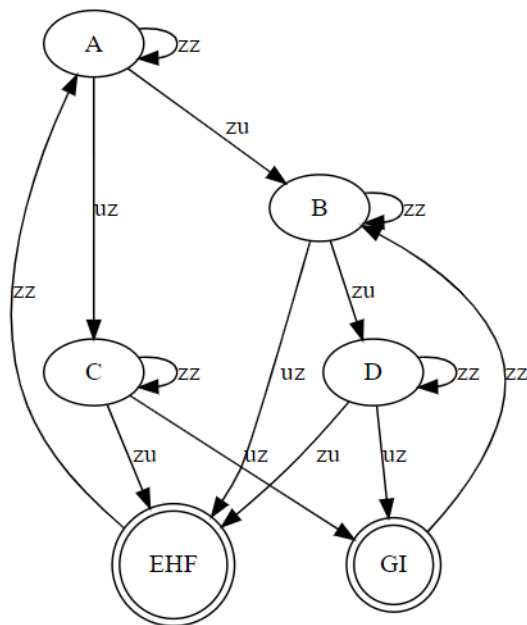


Figura 4.10: Máquina FSM de una máquina expendedora

4.3.1. Compilación de Códigos

La figura 4.11 muestra el circuito RTL equivalente de la máquina FSM de la máquina expendedora presentada en la figura 4.9. La figura 4.12 muestra el circuito RTL equivalente de la máquina expendedora optimizada presentada en la figura 4.10. Se puede notar que tienen las mismas entradas y salidas, pero que, sin embargo, no tienen la misma cantidad de lógica.

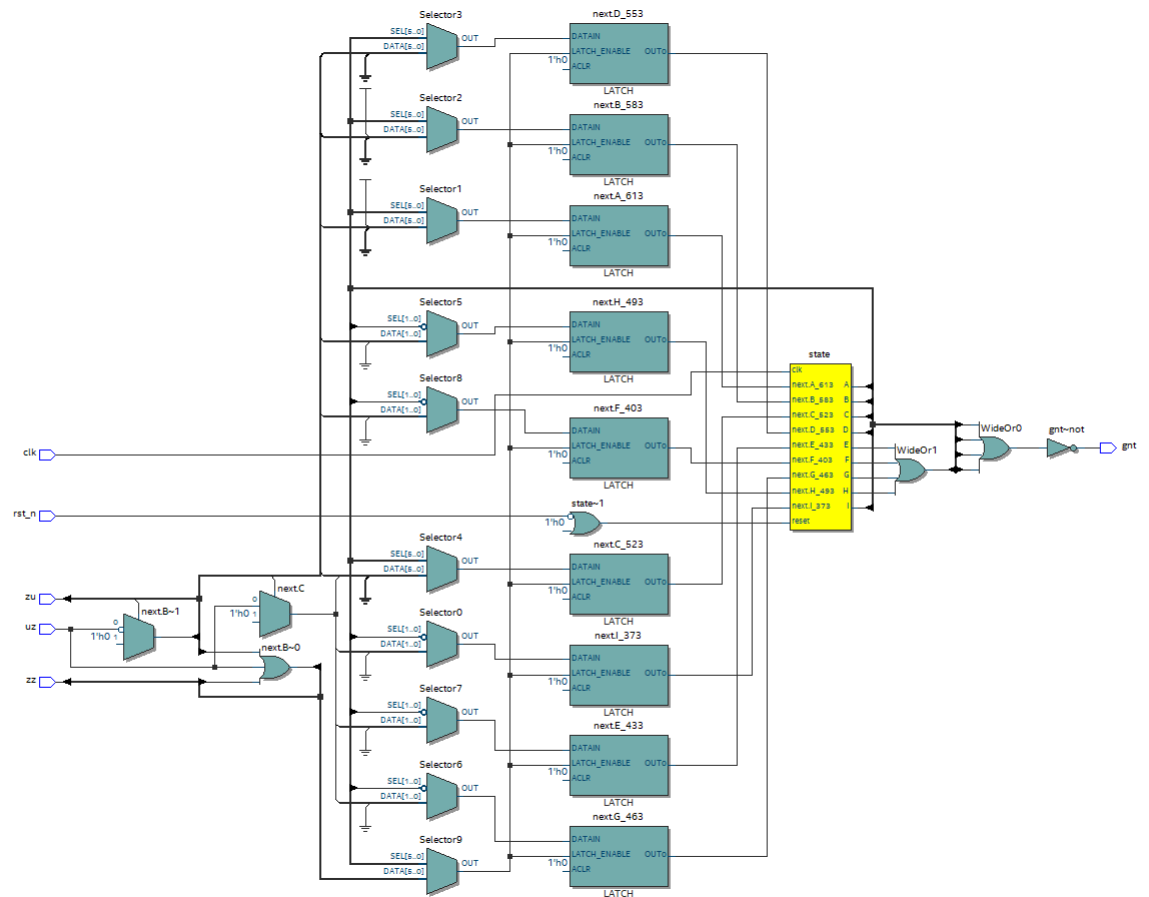


Figura 4.11: Circuito equivalente RTL de una máquina expendedora

Se puede apreciar que el circuito RTL de la máquina expendedora optimizada posee 3 registros menos que la máquina expendedora antes de ser optimizada.

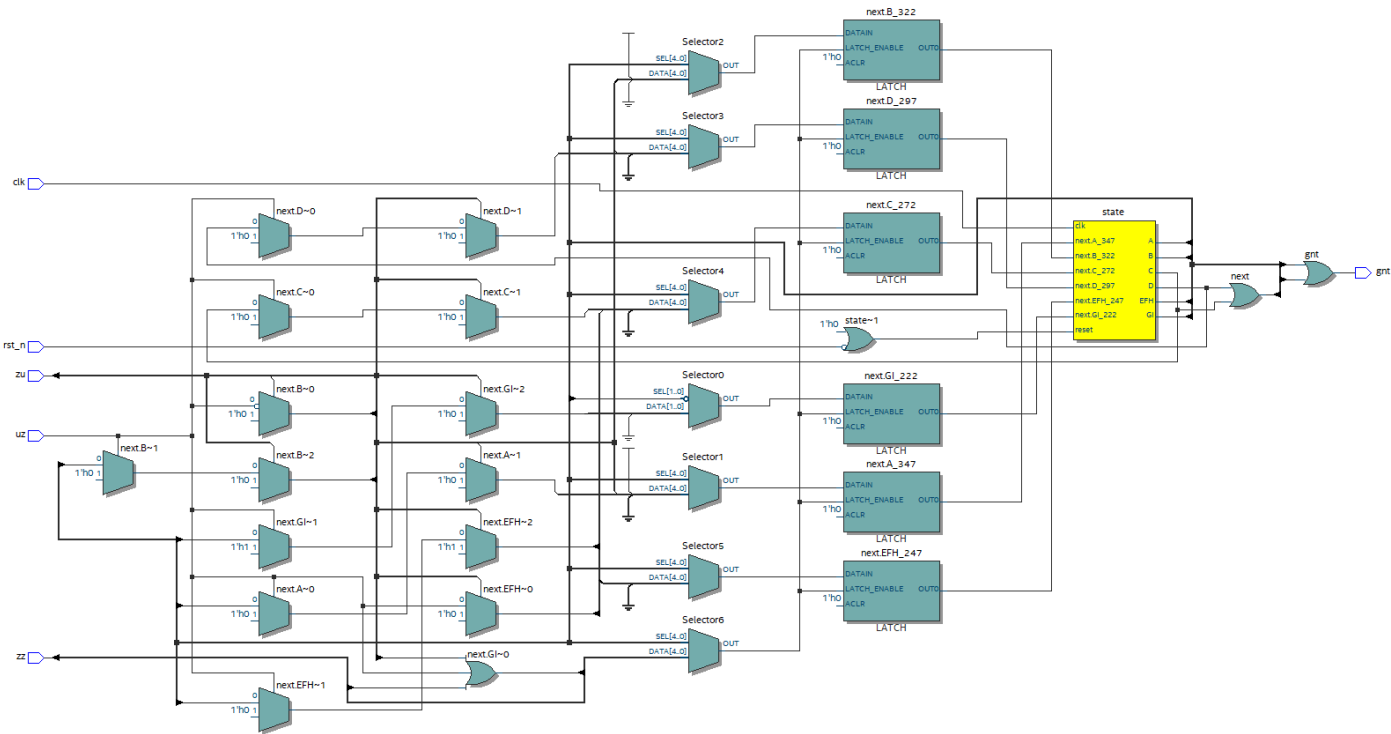


Figura 4.12: Circuito equivalente RTL de una máquina expendedora optimizada

4.3.2. Resultados

La figura 4.13 presenta el resumen obtenido luego de compilar la máquina FSM de 3 nodos presentada en la figura 4.9. La figura 4.14 presenta el resumen obtenido luego de compilar la máquina FSM de 3 nodos optimizada presentada en la figura 4.10.

Circuitos de lógica ALM pasan de doce a nueve y registros pasan de nueve a seis. La cantidad de entradas y salidas utilizadas se mantienen.


| Flow Summary | |
|--|--|
|  <<Filter>> | |
| Flow Status | Successful - Thu Jul 20 17:02:00 2023 |
| Quartus Prime Version | 22.1std.1 Build 917 02/14/2023 SC Lite Edition |
| Revision Name | vendingmachine |
| Top-level Entity Name | vendingmachine |
| Family | Cyclone V |
| Device | 5CSEMA5F31C6 |
| Timing Models | Final |
| Logic utilization (in ALMs) | 12 / 32,070 (< 1 %) |
| Total registers | 9 |
| Total pins | 6 / 457 (1 %) |
| Total virtual pins | 0 |
| Total block memory bits | 0 / 4,065,280 (0 %) |
| Total DSP Blocks | 0 / 87 (0 %) |
| Total HSSI RX PCSs | 0 |
| Total HSSI PMA RX Deserializers | 0 |
| Total HSSI TX PCSs | 0 |
| Total HSSI PMA TX Serializers | 0 |
| Total PLLs | 0 / 6 (0 %) |
| Total DLLs | 0 / 4 (0 %) |

Figura 4.13: Resumen de los resultados del código compilado de la máquina expendedora


| Flow Summary | |
|--|--|
|  <<Filter>> | |
| Flow Status | Successful - Thu Jul 20 17:26:34 2023 |
| Quartus Prime Version | 22.1std.1 Build 917 02/14/2023 SC Lite Edition |
| Revision Name | vendingmachineDone |
| Top-level Entity Name | vendingmachineDone |
| Family | Cyclone V |
| Device | 5CSEMA5F31C6 |
| Timing Models | Final |
| Logic utilization (in ALMs) | 8 / 32,070 (< 1 %) |
| Total registers | 3 |
| Total pins | 6 / 457 (1 %) |
| Total virtual pins | 0 |
| Total block memory bits | 0 / 4,065,280 (0 %) |
| Total DSP Blocks | 0 / 87 (0 %) |
| Total HSSI RX PCSs | 0 |
| Total HSSI PMA RX Deserializers | 0 |
| Total HSSI TX PCSs | 0 |
| Total HSSI PMA TX Serializers | 0 |
| Total PLLs | 0 / 6 (0 %) |
| Total DLLs | 0 / 4 (0 %) |

Figura 4.14: Resumen de los resultados del código compilado de la máquina expendedora optimizada.

4.4. Análisis de los Resultados

En esta sección, se presentarán y analizarán los resultados obtenidos en los dos casos de estudio. Se evaluarán las mejoras realizadas en los nodos pequeños, como la reducción del número de estados y transiciones, así como la simplificación de la lógica subyacente. Se analizará el impacto de estas optimizaciones en términos de rendimiento, consumo de recursos y complejidad del diseño. Además, se compararán las versiones original y optimizada de las máquinas de estado finito en cada caso de estudio, resaltando las ventajas y las limitaciones de cada enfoque.

4.4.1. Máquina FSM de 3 nodos

Se ve en las figuras 4.3a y 4.3b que el software es capaz de condensar los estados "B" y "C" en el estado "BC". Para representar 3 estados se requiere utilizar una codificación de tres bits ("A = 00", "B = 01" y "C = 10"). En particular se escoge esta codificación tomando en cuenta la menor distancia de Hamming entre estados adyacentes. Por otro lado, para representar 2 estados solo se requiere 1 bit ("A = 0" y "BC = 1"). Esto se traduce en una optimización de recursos sustancial tomando en cuenta el tamaño del circuito RTL en la figura 4.4 y en la figura 4.5. Más concretamente, se puede ver en las figuras 4.6 y 4.7 que la máquina optimizada solamente requiere dos bloques de lógica ALM mientras que la máquina sin optimizar requiere tres.

4.4.2. Máquina expendedora

Para demostrar la capacidad del software de optimización se utiliza el controlador de una máquina expendedora. Como se ve en las figuras 4.9 y 4.10 el software logra condensar los estados "E", "H" y "F" en el estado "EHF", y los estados "G" e "I" en el estado "GI". Esto transforma una máquina de 9 estados en una máquina de 6. Para representar una máquina de 6 estados solamente se requieren 3 bits, mientras que para representar 9 estados se requieren 4 bits. Esto se traduce en una optimización de recursos sustancial tomando en cuenta el tamaño del circuito RTL en la figura 4.11 y en la figura 4.12. Más concretamente, se puede ver en las figuras 4.13 y 4.14 que la máquina optimizada solamente requiere ocho bloques de lógica ALM mientras que la máquina sin optimizar requiere 12 y que la máquina sin optimizar requiere 9 registros, mientras que la máquina optimizada requiere 3.

Capítulo 5

Conclusiones y Trabajo a Futuro

Con base en los objetivos establecidos y los resultados obtenidos en este trabajo, se puede concluir lo siguiente:

Se realizó un exhaustivo análisis de los diferentes métodos de optimización disponibles para reducir el consumo de recursos de un FPGA. En cuanto a la optimización de la energía, se puede concluir que al utilizar menos recursos dentro del FPGA, esta requerirá menos energía para mantener sus componentes activos, especialmente en la optimización de registros, *Flip-Flops* y *Latch*. En cuanto a la optimización de la velocidad de la máquina FSM, esta depende más del algoritmo en el que se programe que de la cantidad de estados o la codificación de los mismos. Sin embargo, se menciona el algoritmo de Quine-McCluskey, que se implementa en el capítulo 3. Este algoritmo, sin duda, logra reducir la velocidad de transición entre estados, optimizando los minitérminos de la función de transición.

Se logró el diseño y desarrollo de un algoritmo eficiente capaz de tomar una máquina de estados finitos como entrada y aplicar de manera efectiva los métodos de optimización estudiados. Como resultado, el algoritmo genera una nueva máquina de estados optimizada. Este se implementa en el archivo Python utilizando Flask, tal como se detalla en el capítulo 3.1.1.

Además, se creó e implementó una interfaz gráfica en JavaScript que permite utilizar la herramienta de optimización de máquinas de estado finito. Esta interfaz gráfica aumenta la accesibilidad y usabilidad de la herramienta, lo que la convierte en una herramienta práctica para diseñadores de circuitos FPGA con diferentes niveles de experiencia.

También, se desarrolló un algoritmo adicional que acepta una máquina de estados como entrada y genera un archivo en *SystemVerilog* de la máquina de estados optimizada. Este archivo, que también se implementa en la API Flask, puede ser fácilmente integrado en el flujo de diseño de FPGA, lo que simplifica y agiliza el proceso de implementación de los diseños optimizados.

Este trabajo ha cumplido con todos sus objetivos específicos y ha logrado proporcionar una solución integral para la optimización de máquinas de estado finito y el rendimiento de circuitos digitales en FPGA. Además, el enfoque en la accesibilidad mediante una interfaz

gráfica hace que esta herramienta sea valiosa para diseñadores y desarrolladores de diversos niveles de experiencia.

5.1. Trabajo a Futuro

Se logró desarrollar una página web de optimización de máquinas de estado finito que brinda a los diseñadores de FPGA una herramienta poderosa y accesible para simplificar sus procesos de diseño. Aún queda trabajo para hacer esta herramienta más versátil y más intuitiva para los usuarios. En esta sección se exploran los trabajos a futuro que son esenciales para llevar esta plataforma al siguiente nivel.

- Diseñar una Aplicación móvil para usuarios que busquen utilizar la aplicación web desde equipos que no sean un computador, dar la posibilidad de una versión móvil de la página web o una aplicación para celulares y tablets.
- Dar la posibilidad de descargar el código en un lenguaje diferente a *SystemVerilog*. Esto puede servir para usuarios que están más familiarizados con VHDL o Verilog.
- Generar una sección de comentarios por posibles mejoras que los usuarios puedan proponer al sistema.

Bibliografía

- [1] Cross-origin resource sharing (cors). <https://www.w3.org/TR/cors/>, Acceso en línea: 2023. Especificación técnica utilizada en el desarrollo web para controlar el acceso a recursos entre diferentes dominios.
- [2] Claude Berge. *The theory of graphs*. Courier Corporation, 2001.
- [3] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. An asm semantics for uml activity diagrams. In *International Conference on Algebraic Methodology and Software Technology*, pages 293–308. Springer, 2000.
- [4] Adrián Domínguez Hernández. Síntesis de alto nivel basada en xilinx vivado para aceleradores hardware. Master’s thesis, 2014.
- [5] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, June 2000.
- [6] Emden Gansner. Graphviz-react. Software, 2020.
- [7] Mentor Graphics. Modelsim. Software, 2023.
- [8] Miguel Grinberg. *Flask web development: developing web applications with python*. “O’Reilly Media, Inc.”, 2018.
- [9] Sneha H.L. <https://www.allaboutcircuits.com/technical-articles/purpose-and-internal-functionality-of-fpga-look-up-tables/>, 2023.
- [10] John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pages 189–196. Elsevier, 1971.
- [11] Terasic Technologies Inc. *DE1-SoC User Manual*. Terasic Technologies Inc.
- [12] Intel Corporation. *Quartus Prime User Guide*. Ciudad.
- [13] Ramón Invarato. Hamming, Feb 2022.
- [14] Tarun Kumar Jain, Dharmender Singh Kushwaha, and Arun Kumar Misra. Optimization of the quine-mccluskey method for the minimization of the boolean expressions. In *Fourth International Conference on Autonomic and Autonomous Systems (ICAS’08)*, pages 165–168. IEEE, 2008.

- [15] Johannes Leindecker, Maximilian Zechmeister-Machhart, Felix Gauss, and Philipp Wiegard. A tutorial-oriented approach to argesim benchmark c11 'scara robot' in matlab, simulink and stateflow. *Simulation Notes Europe*, 2021.
- [16] MathWorks. Matlab simulink. Software, 2023.
- [17] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34:1045–1079, 1955.
- [18] Edward F. Moore. Gedanken-experiments on sequential machines. *Automata Studies*, pages 129–153, 1956.
- [19] Mohammad Norouzi, David J Fleet, and Russ R Salakhutdinov. Hamming distance metric learning. *Advances in neural information processing systems*, 25, 2012.
- [20] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273. International World Wide Web Conferences Steering Committee, 2016.
- [21] Francisco Javier Rivera Serrano. *Diseño e Implementación de la Correlación y de la Co-rrrentropía Cruzada, Utilizando FPGA*. Tesis de magister, Universidad de Chile, Santiago de Chile, enero 2018.
- [22] Lattice Semiconductor. *Lattice Diamond User Manual*.
- [23] Bear Shelton. <https://www.lucidchart.com/pages/what-is-a-flowchart-tutorial>, 2023.
- [24] Umair Saeed Solangi, Tayyab Din Memon, Abdul Sattar Noonari, and Omair Ahmed Ansari. An intelligent vehicular traffic signal control system with state flow chart design and fpga prototyping. *Mehran University Research Journal of Engineering & Technology*, 36(2):343–352, 2017.
- [25] K Venkatraman, B Dastagiri Reddy, MP Selvan, S Moorthi, N Kumaresan, and N Ammasai Gounden. Online condition monitoring and power management system for standalone micro-grid using fpgas. *IET Generation, Transmission & Distribution*, 10(15):3875–3884, 2016.
- [26] Xilinx Inc. *Vivado Design Suite User Guide*. San Jose, CA.
- [27] Zhongjiang Yan, Hangchao Jiang, Bo Li, and Mao Yang. A flowchart based finite state machine design and implementation method for fpga. In *IoT as a Service: 6th EAI International Conference, IoTaaS 2020, Xi'an, China, November 19–20, 2020, Proceedings 6*, pages 295–310. Springer, 2021.