GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

HARDWARE SECURITY PRIMITIVE DESIGN, SIMULATION, AND EVALUATION

Autor: Luis de la Mata Sánchez-Izquierdo

Director: Domenic Forte

Madrid

I declare, under my responsibility, that the Project presented with the title

Hardware security primitive design, simulation and evaluation

in the ETS of Engineering - ICAI of the Comillas Pontifical University in the academic year 2021/2022 is of my authorship, original and unpublished and has not been previously submitted for other purposes.

The Project is not plagiarism of another, neither totally nor partially and the information that has been taken from other documents is duly referenced.

Fdo.: Luis de la Mata Date: 05/ 07/ 2022

Authorized the delivery of the project

THE PROJECT MANAGER

Fdo.: Domenic Forte Date: 07/ 06/ 2022

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

HARDWARE SECURITY PRIMITIVE DESIGN, SIMULATION, AND EVALUATION

Autor: Luis de la Mata Sánchez-Izquierdo

Director: Domenic Forte

Madrid

TABLA DE CONTENIDO

HARDWARE SECURITY PRIMITIVE DESIGN, SIMULATION, AND EVALUATION

Autor: de la Mata Sánchez-Izquierdo, Luis

Director: Forte, Domenic.

Entidad Colaboradora: University of Florida, ICAI

RESUMEN DEL PROYECTO

En este proyecto, se desarrollará un circuito polimórfico mediante el algoritmo genético con el objetivo de aportar seguridad a un sistema. Para ello, el circuito ha de borrar automáticamente la información en caso de que ocurra algún ataque físico al sistema.

Palabras clave: Circuito polimórfico, Algoritmo genético, Seguridad

1. Introducción

Este proyecto se centra en el desarrollo de los circuitos polimórficos mediante el algoritmo genético y sus aplicaciones específicamente al ámbito de la ciberseguridad.

Los circuitos polimórficos a grandes rasgos son aquellos circuitos que de manera compacta incluyen más de una funcionalidad en un único modulo.

El algoritmo genético, es un algoritmo del ámbito de la inteligencia artificial que se basa en la teoría de la evolución de Darwin y durante varias iteraciones, mezcla los genes de los mejores individuos de cada generación para generar una nueva población.

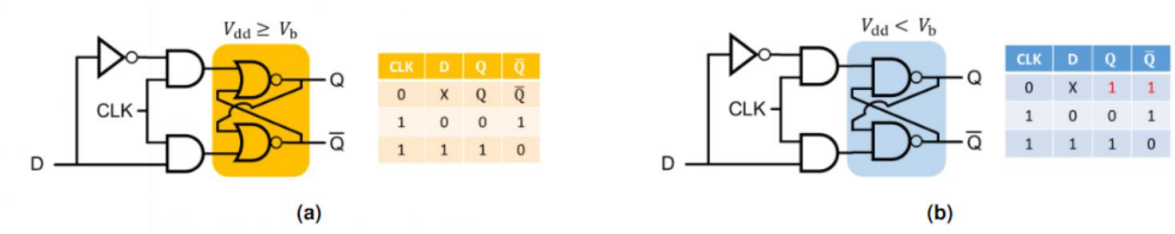
La motivación de este proyecto surge de un interés personal hacia los campos de la inteligencia artificial y ciberseguridad. Con este proyecto me surgió la oportunidad de conocer más acerca de ambos campos y al mismo tiempo solucionaba una vulnerabilidad de un sistema, los ataques físicos a circuitos.

2. Definición del Proyecto

Este proyecto se puede dividir en 3 objetivos principales. En primer lugar, desarrollar un algoritmo genético capaz de evolucionar circuitos hasta obtener el deseado. En segundo lugar, se modificará dicho algoritmo para obtener un circuito polimórfico deseado. Por último, se utilizará el circuito desarrollado anteriormente para prevenir ataques en un circuito físico.

3. Descripción del circuito a desarrollar

Como ya se ha mencionado, se tratará de desarrollar un flip-flop con comportamiento polimórfico. El comportamiento que se buscara es el mostrado en la Figura 1.1.



CLK	D	Q	$\overline{Q}$
0	X	Q	$\overline{Q}$
1	0	0	1
1	1	1	0

CLK	D	Q	$\overline{Q}$
0	X	1	1
1	0	0	1
1	1	1	0

Figura 1.1. Descripción del Flip-flop ha desarrollar

4. Resultados

El resultado obtenido a través del algoritmo genético fue un flip-flop con el resultado mostrado en el apartado anterior. Dependiendo del voltaje suministrado otorgaba unas salidas u otras. En la en los tiempos previos a 0.08 segundos el voltaje suministrado era 1.1 voltios y

tras este tiempo se suministraban 0.9 voltios. De forma que las salidas Q y $\bar{Q}$ son las representadas en la Figura 1.2.

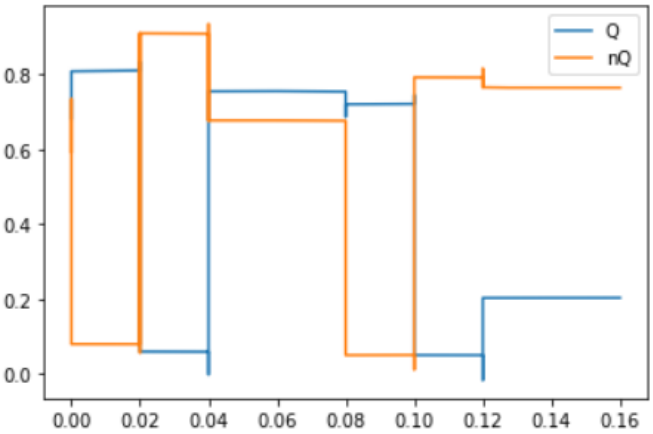


Figura 1.2. Simulación del flip-flop desarrollado

Este flip-flop, se puede utilizar para prevenir ataques físicos que manipulen el suministro de voltaje para obtener información confidencial. Esto es posible ya que cuando el circuito detecta que el voltaje sube por encima de 1.1 voltios la información que contiene se elimina.

5. Conclusiones

No solo se ha logrado desarrollar un algoritmo capaz de generar circuitos polimórficos de forma exitosa, sino que se ha logrado un circuito que presenta una nueva posibilidad para proteger información de manera eficaz de posibles ataques físicos.

HARDWARE SECURITY PRIMITIVE DESIGN, SIMULATION, AND EVALUATION

Autor: de la Mata Sánchez-Izquierdo, Luis

Proyect manager: Forte, Domenic.

Collaborating Entity: University of Florida, ICAI

ABSTRACT

In this project, a polymorphic circuit will be developed using the genetic algorithm in order to provide security to a system. For this purpose, the circuit has to automatically erase the information in case of any physical attack to the system.

Keywords: Polymorphic circuit, Genetic algorithm, Security.

1. Introduction

This project focuses on the development of polymorphic circuits using the genetic algorithm and their applications specifically to the field of cybersecurity.

Polymorphic circuits are those circuits that compactly include more than one functionality in a single module.

The genetic algorithm is an algorithm in the field of artificial intelligence that is based on Darwin's theory of evolution and during several iterations, mixes the genes of the best individuals of each generation to generate a new population.

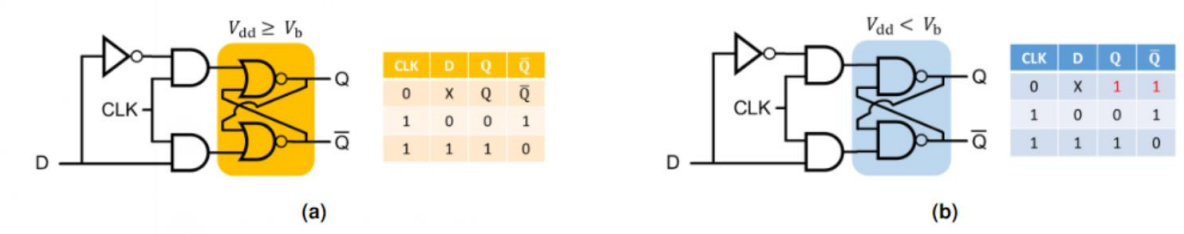
The motivation for this project stems from a personal interest in the fields of artificial intelligence and cybersecurity. With this project I had the opportunity to learn more about both fields and at the same time I was solving a vulnerability of a system, the physical attacks to circuits.

2.Project Definition

This project can be divided into 3 main objectives. First, to develop a genetic algorithm capable of evolving circuits until the desired one is obtained. Secondly, this algorithm will be modified to obtain a desired polymorphic circuit. Finally, the previously developed circuit will be used to prevent attacks on a physical circuit.

3.Description of the circuit to be developed

As already mentioned, we will try to develop a flip-flop with polymorphic behavior. The behavior that will be sought is the one shown in the Figure 1.



CLK	D	Q	$\overline{Q}$
0	X	Q	$\overline{Q}$
1	0	0	1
1	1	1	0

CLK	D	Q	$\overline{Q}$
0	X	1	1
1	0	0	1
1	1	1	0

Figure 1. Description of the flip-flop to be developed

4.Results

The result obtained through the genetic algorithm was a flip-flop with the result shown in the previous section. Depending on the supplied voltage it gave some outputs or others. At the

time before 0.08 seconds the voltage supplied was 1.1 volts and after this time 0.9 volts were supplied. So that the outputs Q and $\bar{Q}$ are those represented in the Figure 2.

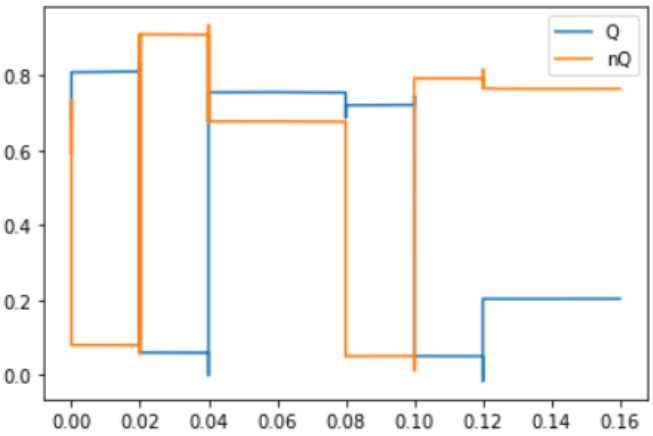


Figure 2. Developed flip-flop simulation

This flip-flop can be used to prevent physical attacks that manipulate the supply voltage to obtain confidential information. This is possible because when the circuit detects that the voltage rises above 1.1 volts the information it contains is removed.

5.Conclusions

Not only has it been possible to develop an algorithm capable of successfully generating polymorphic circuits, but it has also been possible to develop a circuit that presents a new possibility to effectively protect information from possible physical attacks.

1. Introducción

En los últimos años, hay dos campos que se están desarrollando mucho debido a su utilidad en el mundo moderno y a su vez están estrechamente relacionados con la ingeniería de telecomunicaciones. Estos campos son la ciberseguridad y la inteligencia artificial. Estos campos al juntarse en un proyecto pueden dar lugar a resultados realmente interesantes como se verá a lo largo del trabajo.

La idea de este proyecto surge debido a que, a pesar de la constante evolución de la ciberseguridad para proteger cualquier tipo de información, siguen ocurriendo ataques utilizando diferentes vulnerabilidades y una de estas es un circuito físico, por lo tanto, este proyecto busca una solución novedosa y eficaz para dicha vulnerabilidad de manera que la información este más segura.

En el ámbito de la inteligencia artificial se utilizará el algoritmo genético o de la evolución, este algoritmo tiene un número de usos incontable, pero en el ámbito de la electrónica uno de los usos que se le puede dar es generar un circuito con funcionalidad oculta, para ello, utiliza algunas propiedades de los transistores que normalmente no se tienen en cuenta, a este tipo de circuitos se les llama circuitos polimórficos y se explicarán exhaustivamente más adelante.

En cuanto a la ciberseguridad anteriormente mencionada, en este proyecto se tratará de borrar información almacenada en caso de que un agente externo trate de acceder a dicha información mediante un circuito físico. Esto se realizará con la ayuda de un flip-flop con esa función.

Debido a la complejidad que puede presentar el algoritmo genético y lo importancia que tiene su comprensión para entender el proyecto, el próximo apartado está dedicado a una explicación más en detalle de este.

1.1. Algoritmo genético

El nombre de este algoritmo viene de su origen, ya que, lo que realiza, es imitar la evolución genética de la naturaleza como la presento Darwin, con el objetivo de alcanzar una población final que tenga las características deseadas.

Algunas palabras clave que son necesarias comprender para este algoritmo son:

-Genes: Es el conjunto de datos que identifican a un individuo (cromosoma), por ejemplo, en nuestro caso (un circuito electrónico compuesto por transistores), los genes serían las conexiones entre los transistores, el tipo de transistores, la longitud de los transistores, la anchura de los transistores y el voltaje utilizado.

-Cromosoma: Es el individuo definido por un conjunto de genes, en este caso sería un circuito con las conexiones, los transistores y el voltaje definidos.

-Población: Es un grupo de cromosomas con diferentes genes de manera que podamos elegir los más adecuados para nuestro objetivo.

-Función objetivo: Es una función mediante la cual se evaluará en qué medida un individuo realiza las funciones deseadas.

-Padres: Los padres son 2 individuos de una población que han sido seleccionados y a partir de los cuales se generará la siguiente población.

La manera más sencilla de comprender este algoritmo es como un ciclo iterativo. Ya que el algoritmo se repetirá hasta que logre su objetivo, por lo tanto se ha realizado el siguiente diagrama como ayuda.

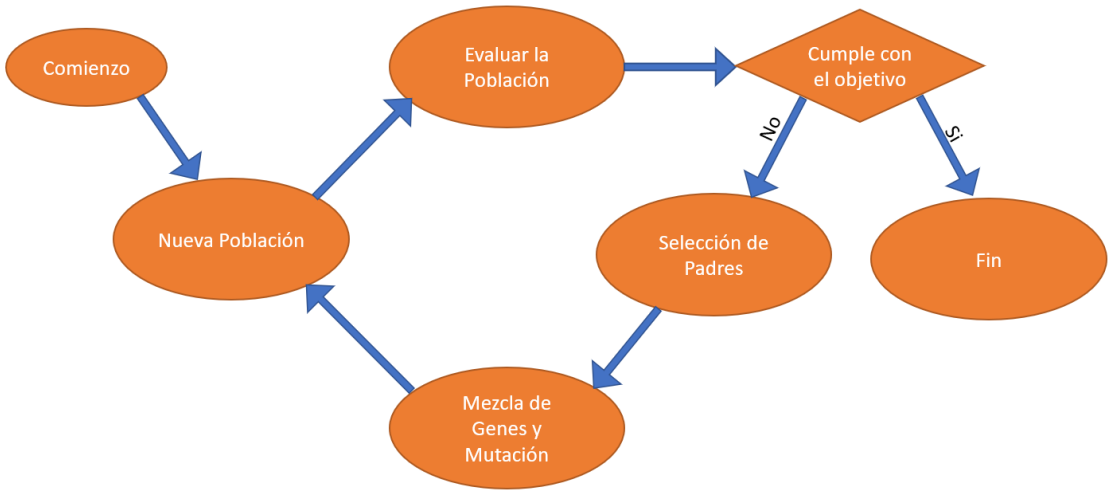


Figura 1.1. Diagrama del algoritmo genético

A continuación, se explicarán las diferentes funciones presentadas en el diagrama y que se implementarán en el algoritmo desarrollado.

1.1.1. Nueva población

En esta etapa del algoritmo, se genera ya sea una población inicial si es la primera generada o una nueva población a partir de la anterior. En caso de la inicial se generarán cromosomas aleatorios para tener diversidad genética y que resulte más sencillo llegar al objetivo.

1.1.2. Evaluar la población

Una vez tenemos una población, debemos evaluar cada cromosoma mediante la función objetivo. Esto será tendrá dos utilidades, en primer lugar, para determinar si la población cumple con la funcionalidad deseada y por lo tanto el algoritmo ha finalizado, además, servirá para asignar un valor de dicha función a cada individuo lo que será necesario para más adelante seleccionar los padres de la futura generación.

1.1.3. Selección de padres

Tras asignar un valor de la función objetivo a cada individuo, ya podemos seleccionar a los padres que serán el pilar para generar la siguiente población. Como criterio de selección para los padres existen distintas posibilidades.

- Selección aleatoria: Mediante este criterio se selecciona de forma completamente aleatoria los padres de la próxima generación entre la población actuar. Esto se realiza sin tener en cuenta ni siquiera el valor de la función objetivo de cada individuo. Debido al gran elemento de aleatoriedad que implica esta selección muchas veces se suele evitar.

- Selección por torneo: En este caso se obtienen un número de cromosomas de forma aleatoria de la población total y de este grupo se selecciona el que tenga una mejor función objetivo. Por ejemplo:

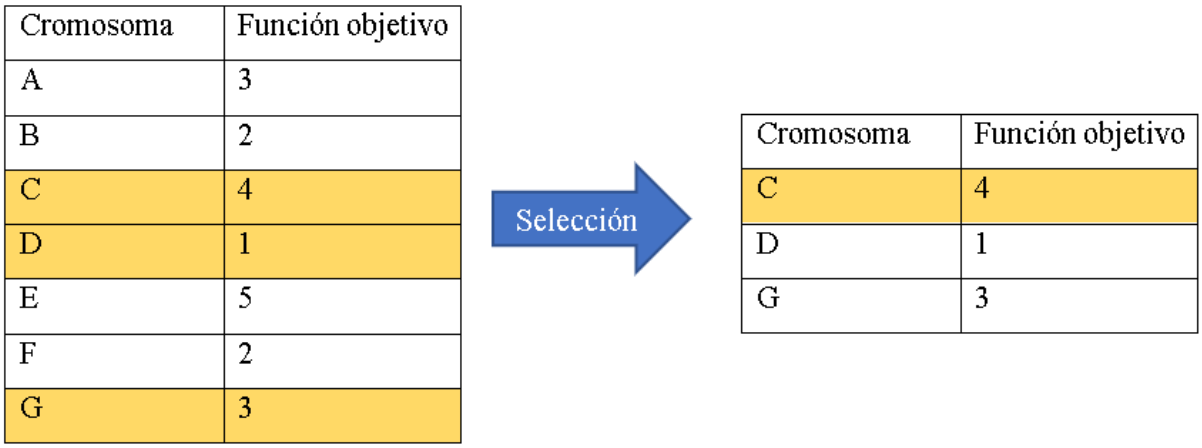


Figura 1.2. Ejemplo de selección de un padre

En este ejemplo se seleccionaría al individuo C como padre, este proceso se tendría que repetir de nuevo para seleccionar al segundo padre.

-Selección por ruleta: Este criterio lo que hará será asociar un porcentaje a cada individuo en base a el valor de la función objetivo del mismo (cuanto mayor sea el valor de la función mayor será el porcentaje asociado). Una vez con los porcentajes asociados, se genera una ruleta en la cual cada rango es un cromosoma y cada uno tiene la probabilidad de ser elegido igual al porcentaje asignado anteriormente. Ejemplo:

Tabla 1.1. Ejemplo de porcentajes de una población

Cromosoma	Función objetivo	Porcentaje
A	3	15%
B	2	10%
C	4	20%
D	1	5%
E	5	25%
F	2	10%
G	3	15%

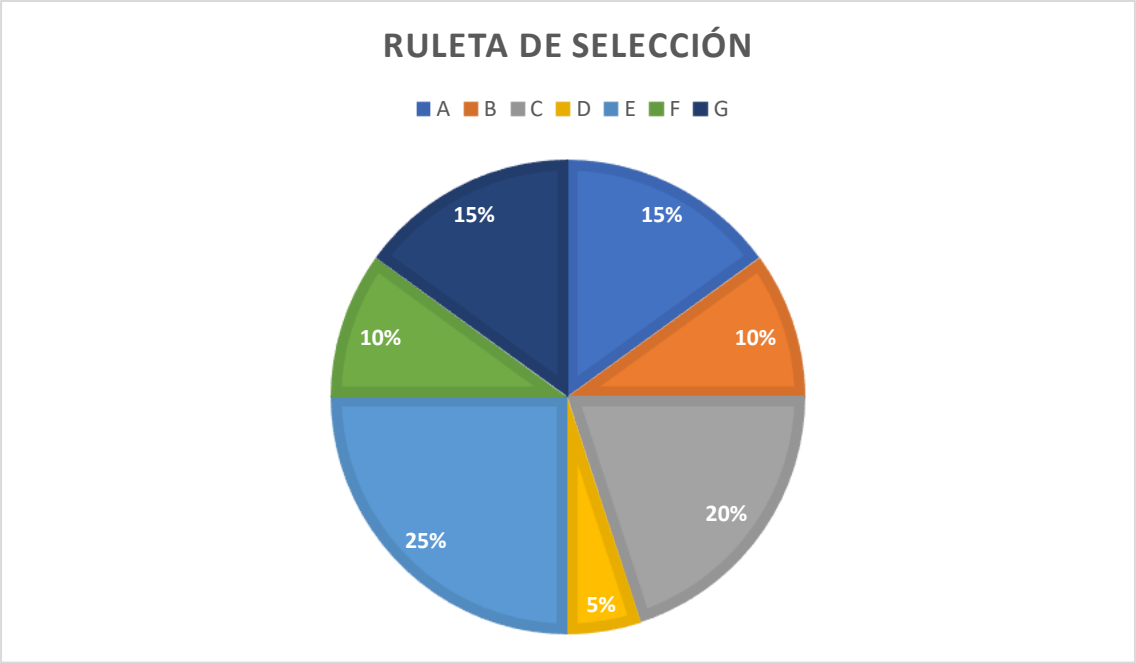


Figura 1.3. Ejemplo de ruleta de selección

En este ejemplo no sabemos cuál sería el padre finalmente, sino que habría que generar aleatoriamente una posición en la ruleta y observar en que rango está para seleccionar el padre finalmente. Al igual que en la selección por torneo habría que repetir el proceso una vez más para el segundo padre.

1.1.3 Mezcla de genes y mutación

Este paso se podría dividir en dos, pero se suele realizar al mismo tiempo, ya que mientras se mezclan los genes es cuando ocurre una mutación en algunos de estos.

La mezcla de genes a grandes rasgos consiste en obtener los genes de ambos padres y seleccionar varios genes de cada uno para generar a cada individuo de una nueva población. Para seleccionar los genes de cada padre que pasarán al hijo hay distintas maneras:

La manera más básica para la mezcla de genes es simplemente generar uno o varios puntos a partir de los cuales se cogen los genes de un padre o del otro.

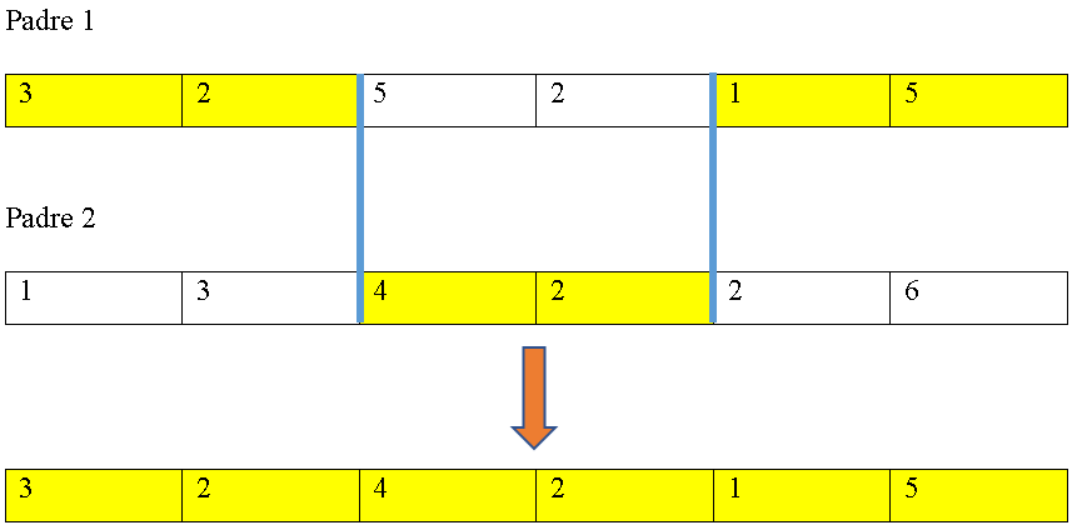


Figura 1.4. Ejemplo de mezcla 1

En el ejemplo mostrado en la Figura 1.4 cambiaría el padre del que se cogen los genes tras el elemento 2 y 4. De forma que el resultado es la mezcla mostrada finalmente.

Otra opción para ejecutar la mezcla de genes es para cada gen seleccionar aleatoriamente que padre proveerá dicho gen. Se muestra un ejemplo de esto en la.

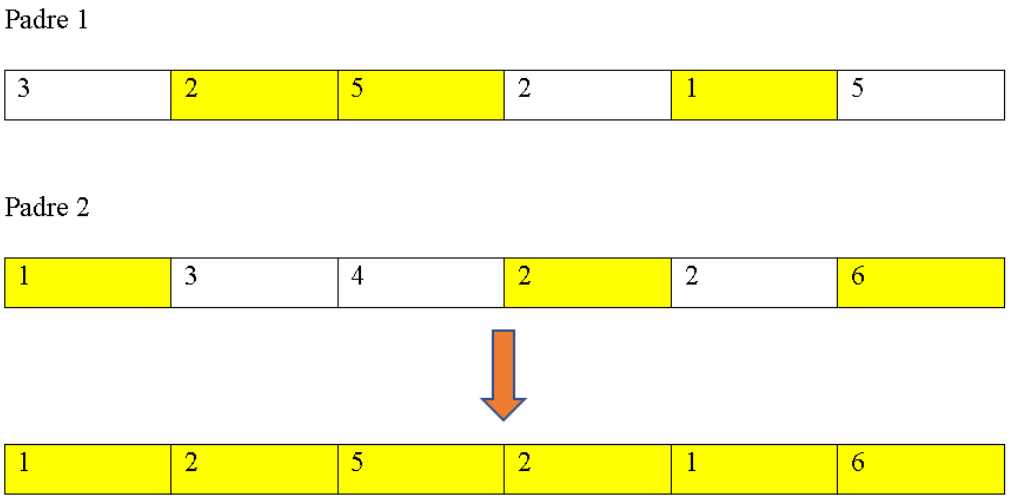


Figura 1.5. Ejemplo de mezcla 2

En cuanto a la mutación de los genes se refiere a un ligero cambio a estos de cualquier tipo, ya sea alterar el gen directamente o cambiar el orden de los genes o cualquier otro tipo de alteración. El objetivo de esto es aumentar la diversidad genética en la población y aumentar la eficacia del algoritmo.

1.2. Circuitos polimórficos

A lo largo de los últimos años se han producido grandes avances en la tecnología de transistores CMOS, esto ha permitido el desarrollo de nuevos circuitos no planteados anteriormente. Algunos de estos circuitos son los circuitos polimórficos, estos se caracterizan por ser capaces de implementar funciones lógicas y sensores en un solo circuito de manera compacta.

Este tipo de circuitos presentan varias aplicaciones de gran utilidad. Entre estas podemos encontrar:

- Circuitos capaces de detectar automáticamente errores.
- Sensores enfocados a una gran variedad de campos desde biométricos hasta industriales.
- Implementar una función oculta que se activa en determinadas condiciones externas

La aplicación de estos circuitos que se utilizará para este proyecto es la última mencionada y por ello a continuación se explicara más detenidamente.

La manera en la que se implementa la funcionalidad oculta en un circuito polimórfico es la siguiente. Consiste en un circuito que generalmente tiene un comportamiento particular, sin embargo, si se presentan unas condiciones específicas la funcionalidad del circuito cambia respecto a la original. Algunas de estas condiciones que pueden alterar el funcionamiento son: La temperatura externa, el voltaje que se le aporta o la luz. Además, este cambio de funcionalidad no es perceptible por el usuario ya que en un circuito normal la manera de implementar esto sería dividirlo en dos módulos y que, mediante un sensor, en unas condiciones determinadas cambiase entre estos. Sin embargo, en el tipo de circuito descrito, todo se implementa en un único modulo.

Para explicar de manera más comprensible este campo, a continuación se presenta un ejemplo básico de una puerta lógica polimórfica. La condición con la que varía este circuito hipotético es la temperatura de forma que en una temperatura ambiente (25º C) el circuito se comportaría como una puerta lógica AND y al pasar un umbral de temperatura (40ª) el circuito se comportará como una puerta lógica OR. Este circuito estaría formado por un único modulo y por lo tanto sería catalogado como circuito polimórfico.

1.3. Objetivos

Debido a la complejidad que se presentaba inicialmente al adentrarme en campos nuevos en este proyecto y a su vez el interés que me generaba, los objetivos se pueden dividir en tres.

1- Desarrollar un algoritmo de la evolución capaz de obtener un circuito objetivo:

En primer lugar, como herramienta que será la base del proyecto es necesario desarrollar un programa que, mediante el algoritmo genético anteriormente explicado, sea capaz de desarrollar un circuito si se le marcan los objetivos adecuados.

Este programa podrá ser utilizado no únicamente para el desarrollo del circuito buscado en este proyecto, sino que con ligeras adaptaciones al objetivo del circuito es capaz de generar cualquier otro circuito basado en transistores.

2- Desarrollar un circuito capaz de detectar cambios externos y cambiar la salida de este:

Dado que ya tenemos una herramienta capaz de desarrollar circuitos, se utilizará esta herramienta para generar un circuito capaz de contener funcionalidad oculta (circuito polimórfico). Además, se realizará un pequeño estudio analizando el funcionamiento del mismo.

3- Aplicar el circuito a un problema real de seguridad en circuitos físicos.

Una vez tenemos un circuito capaz de cambiar su respuesta a un input externo ya sea la temperatura o un cambio en el voltaje, el objetivo final de este proyecto es implementar el circuito en un problema real de seguridad para proponer una solución al mismo.

2. Herramientas Utilizadas

Para este proyecto además de la investigación realizada sobre los campos que se tratarán en el trabajo también ha sido de vital importancia un estudio sobre los programas que se utilizarán para desarrollarlo. Entre estos programas está LTspice para la simulación de circuitos, lo que será esencial para dar un valor de la función objetivo a cada circuito generado ya que se requerirá simular los circuitos para analizar su comportamiento. También se utilizará Python junto a las librerías que se han usado, especialmente PyLTSpice que mediante la cual se han modificado los circuitos utilizando el algoritmo genético.

2.1. LTspice

Esta es una herramienta dedicada principalmente a la simulación de circuitos, los conocimientos básicos del programa fueron adquiridos en la clase de Electrónica. Sin embargo, ha sido necesaria más indagación en el programa ya que se han necesitado algunas funciones que no se habían explorado anteriormente.

La herramienta que más importante para este proyecto del programa es la posibilidad de trabajar con netlists en vez de un entorno gráfico del circuito. Estas netlists son archivos de texto simulables mediante LTspice que presentan mucha facilidad a la hora de editarlos mediante un programa externo como puede ser Python. Esto ha sido de gran ayuda a la hora de realizar una gran cantidad de circuitos con ligeros cambios entre ellos.

SPICE Netlist: C:\Users\Luis\Documents\LTspiceXVII\Draft7.net

```
* C:\Users\Luis\Documents\LTspiceXVII\Draft7.asc
M1 N001 N002 N003 N001 PMOS l=1.5u w=3.452e-05
M2 N005 N004 0 N001 PMOS l=100n w=4.9430000000000006e-05
M3 N001 N001 N003 N001 PMOS l=75n w=4.1960000000000005e-05
M6 N001 N001 N001 N001 PMOS l=60n w=7.100000000000003e-07
M10 0 N002 N005 N001 PMOS l=100n w=3.063e-05
M5 N001 N001 N005 0 NMOS l=250n w=3.834e-05
M8 N003 N002 0 0 NMOS l=5.5u w=1.9350000000000003e-05
M9 N003 N004 0 0 NMOS l=1u w=4.58e-06
V1 N002 0 PULSE(0 1.1 0 5n 5n 20m 40m)
V2 N001 0 PULSE(0.9 1.3 0 5n 5n 80m 160m)
V3 N004 0 PULSE(0 1.1 0 5n 5n 40m 80m)
.model NMOS NMOS
.model PMOS PMOS
.lib C:\Users\Luis\Documents\LTspiceXVII\lib\cmp\standard.mos
.libC:\Users\Luis\Desktop\TFGCode/45nm_bulk.lib
.temp 27
.tran 160m
.backanno
.end
```

Figura 2.1. Ejemplo de una netlist

Además, al ejecutar este tipo de archivos mediante LTspice, se genera otro archivo con la extensión .raw que se puede leer para acceder a los valores de las señales. Estos serán los archivos que se utilizarán para medir hasta qué punto los circuitos cumplen con la funcionalidad esperada.

2.2. PyLTSpice

Esta librería de Python ha sido de gran ayuda, ya que dispone de una gran cantidad de funciones que facilitan enormemente tanto la edición de netlists, como la simulación de estas mediante LTspice e incluso la obtención de los datos finales de forma completamente automática.

Las funciones utilizadas para el proyecto se pueden dividir en dos grupos diferentes, aquellas cuya función está enfocada en la edición y simulación de una netlist y aquellas que se dedican a la recogida de datos de un archivo .raw.

2.2.1. Edición y simulación de netlist:

En este grupo de funciones nos encontraremos con aquellas que trabajan con archivos .net (netlists).

- set_element_model(): Mediante esta función se puede cambiar un elemento del circuito ya sea el modelo, las conexiones o el valor de este.
- set_component_value(): Es similar a la anterior pero únicamente se cambia el valor, por ejemplo en una fuente de voltaje se podría cambiar tan solo el voltaje que aporta al circuito.
- add_instructions(): Es un función especializada en añadir las instrucciones a la hora de la simulación del circuito. Algunas instrucciones podrían ser la temperatura externa, el tiempo de simulación o la señal a medir.
- write_netlist(): Una vez se han modificado los componentes y añadido las instrucciones de simulación, se utiliza esta función para escribir todos esos datos actualizados en una nueva netlist.
- run(): Una vez la nueva netlist se ha generado podemos simularla mediante LTspice utilizando esta función.
- wait_completion(): Esta función espera a que se complete la simulación.
- reset_netlist(): Resetea la netlist para que los cambios realizados solamente se observen en la generada recientemente para ello.

2.2.2. Recogida de datos de archivo .raw:

En este grupo de funciones nos encontraremos con aquellas que trabajan con archivos .raw.

- LTSpiceRawRead: Esta función lee el archivo .raw para que sea posible guardar dicha información en una variable y facilitar el trabajo con ella.

  Las siguientes tres funciones tienen una funcionalidad conjunta. Esta consiste en obtener los valores numéricos de la señal de salida para poder medir la función objetivo de cada cromosoma.

- get_trace
- get_steps()
- get_wave(step)

3. Desarrollo del Algoritmo Genético

Previo al desarrollo del algoritmo, es necesario saber qué tipo de circuito vamos a evolucionar para obtener el deseado. Ya que no es lo mismo evolucionar un circuito con fuentes de voltaje, resistencias, condensadores y transistores que otro con únicamente resistencias. Además, también es vital establecer que partes del circuito se evolucionarán, ya que, de nuevo, evolucionar la estructura completa requiere un código distinto que evolucionar tan solo la longitud de un par de transistores. Por esto, lo primero que se presentará será el tipo de circuito a evolucionar mediante el algoritmo.

El circuito que se planteó como objetivo para evolucionar inicialmente está formado únicamente por transistores y fuentes de voltaje, se partirá de una estructura ya formada y se evolucionará el voltaje suministrado y la anchura de todos los transistores. Además, el tipo de circuito polimórfico que se desarrollara tendrá como funcionalidad deseada que cambie de función dependiendo del voltaje, por lo tanto, se tendrán que evolucionar dos voltajes, el inferior y el superior.

El código desarrollado se puede dividir en dos grandes partes, el algoritmo genético per se y la interacción con LTspice de este. A su vez, cada una de estas partes realizan múltiples funciones que más adelante se explicarán exhaustivamente.

Además, es importante conocer que la totalidad del código ha sido desarrollado en Python principalmente debido a la facilidad de interaccionar con LTspice mediante librerías.

3.1. Algoritmo Genético en Python

Para explicar el código desarrollado, en primer lugar se explicarán sus funciones individualmente para más adelante explicar el funcionamiento general. Las funciones del código son esencialmente las mismas que las etapas del algoritmo con alguna función extra y alguna modificada.

Como recordatorio de este algoritmo el orden del algoritmo es el siguiente:

1. Nueva Población
2. Evaluar Población
3. Selección de Padres
4. Mezcla de Genes y Mutación

3.1.1. Generación de una población inicial

En primer lugar, se presenta el código mediante el cual se genera una población aleatoria inicial de circuitos, para ello se han utilizado dos funciones, una de ellas se encarga de generar cada gen (anchura de transistor y voltaje suministrado) y la otra genera todos los cromosomas necesarios para la población inicial y los guarda en una variable población para que más adelante se puedan evaluar.

La función encargada de generar los genes se puede observar en la Figura 3.1.

```python
def generate_gene(lower_limit, upper_limit, step_size=1):
    """
    :param lower_limit: Lower limit allowed by a given technology node.
    :param upper_limit: Upper limit allowed by a given technology node.
    :param step_size: Determined based on the resolution provided by the foundry. Default set to 1.
    :return: Width value for a transistor in nanometer (nm) scale
    """

    genes = random.randrange(lower_limit, upper_limit, step_size)
    return genes * 1e-9
```

Figura 3.1. Función generadora de genes

Esta función tiene como argumentos el límite superior, inferior y la precisión en la medida que se requiere. La función genera un numero aleatorio con esos valores y lo devuelve multiplicado por 10^{-9}, esta multiplicación se realiza debido a que la mayoría de los genes son anchuras de transistores y el tamaño de estos es del orden del nanómetro/micrómetro.

Ejemplo de funcionamiento para generar la anchura de un transistor:

```
anchura=generate_gene(100,5000,10)
```

Esta línea guardará un valor aleatorio en la variable anchura entre 100 nanómetros y 5 micrómetros con una precisión de 10 nanómetros.

Para generar el voltaje se utilizaría esta línea:

```
voltaje = generate_gene(2,100,1) * 1e8
```

El valor que se guarda en la variable voltaje esta entre 0.2 y 10 voltios con una precisión de 0.1 voltio.

Una vez podemos generar los diferentes genes necesarios para la población inicial, debemos ordenarlos en cromosomas (circuitos individuales) y esto lo haremos mediante la función indicada en la Figura 3.2.

```python
def population_generator():
    lower_value, upper_value, step_size = 100, 5000, 10
    n = 8 ## Number of transistor widths needed in the netlist
    population=[[0 for c in range(10)] for b in range(100)]
    for i in range(100):
        BestParents = [generate_parent(lower_value,upper_value,step_size) for i in range(n)]
        vh = generate_parent(20,100,1)*1e8
        vl=vh-1.5
        chromosome=[BestParents,vh,vl]
        population[i]=chromosome

    return population
```

Figura 3.2. Función para generar la población inicial

Esta función no recibe nada como argumento y devuelve una población inicial de circuitos con valores aleatorios. Su funcionalidad completa consiste en generar todos los genes para cada circuito (mediante la última función explicada) y guardar todos los circuitos generados en un vector de circuitos que será la población inicial a partir de la cual se ejecutará el algoritmo.

3.1.2. Evaluar la población

Una vez el programa ya es capaz de generar una población inicial el siguiente objetivo del algoritmo es ser capaz de evaluarla. Esto será esencial para poder determinar si el algoritmo debe continuar o si ya ha alcanzado su objetivo, además también es necesario para poder seleccionar de forma óptima a los padres de la siguiente generación. En este caso solo se ha desarrollado una función para ello.

```
def get_fitness(Guesses,vh,vl,netlist1,netlist2,netlist3,netlist4,paint,n):
    """
    The 'netlist_modifier' and 'output_parser" functions are written to parse the given netlist and its corresponding
    measurement file after the simulation in LTSPICE. These functions are included in the parser.py file.

    The fitness function here is the propogation delay at the output. This can be modified accordingly.

    :param Guesses: represents the potential values of widths of the transistors
    :param spice_file_without_extension: the netlist which will be simulated in LTSPICE
    :return: the fitness value from the measurement file
    """

    works=netlist_modifier(NETLIST,
                Guesses,vh,vl,False,str(n))
    if works:
        outputs = output_parser(netlist1,netlist2,netlist3,netlist4,paint)
        for i in range(len(outputs)):
            for j in range(len(outputs[i])):
                if outputs[i][j]>1:
                    outputs[i][j]=1
    else:
        fitness=-40
        os.system("taskkill /f /im XVIIx64.exe")
    return fitness
```

Figura 3.3. Función para el cálculo de la función objetivo

La función recibe como argumento los genes de un circuito (la anchura de los transistores, el voltaje superior y el inferior) y las netlists en las que se encuentra el circuito. Y devuelve el valor de la función objetivo del cromosoma.

Para hallar el valor de la función objetivo, ha de modificar algunas netlists, ejecutarlas y obtener los resultados de la simulación, esto se logra mediante las funciones *netlist_modifier* y *output_parser* que se explicarán más adelante. Una vez se obtiene los resultados de la simulación, estos se comparan con los resultados deseados de manera que se puede medir lo bien o mal que funciona cada circuito y se le da un valor en función a ello. Debido a que dependiendo del circuito a desarrollar el cálculo de la función objetivo varía se aporta un ejemplo a continuación.

Suponiendo que el objetivo del algoritmo es desarrollar un circuito que a bajo voltaje se comporte como una puerta OR y a un voltaje superior sea una puerta AND sería importante considerar los siguientes factores. En primer lugar, debido a que hay que someter el circuito a

dos voltajes diferentes para comprobar su funcionalidad en ambos entornos, debemos tener en cuenta que la variable outputs proveniente de la función *output_parser* será una matriz de dos filas que contendrá la información de ambas situaciones. Además, debemos considerar cual es el resultado óptimo de cada entorno y compararlo con el real. Por ejemplo, consideremos que el circuito recibe las siguientes señales:

Input 1:

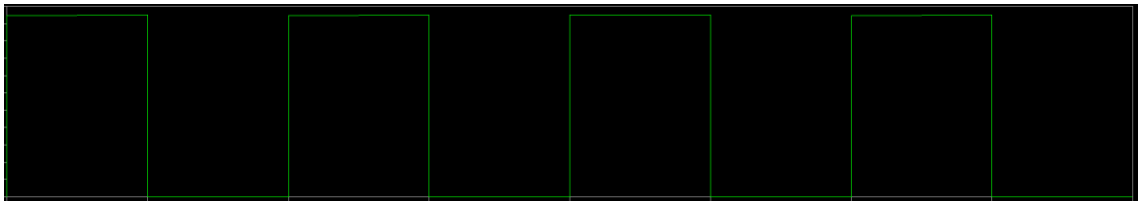


Figura 3.4. Ejemplo de una posible señal de entrada

Input 2:

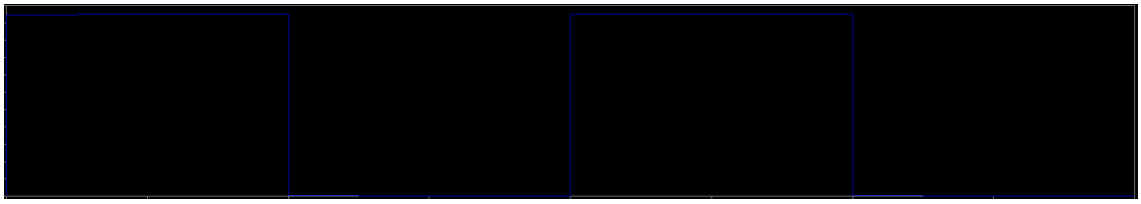


Figura 3.5. Ejemplo de otra posible señal de entrada

La señal de salida óptima para el voltaje inferior sería:

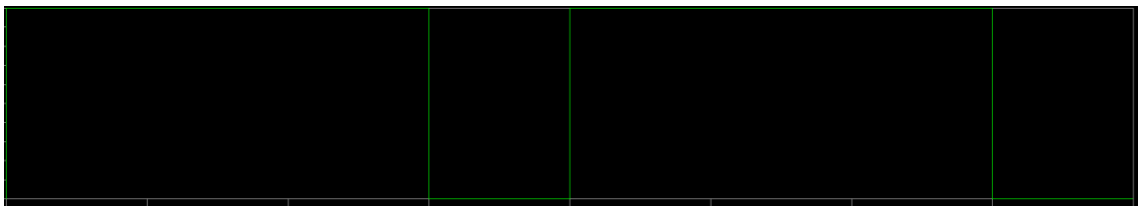


Figura 3.6. Salida óptima de puerta lógica OR

La señal de salida óptima para el voltaje superior:

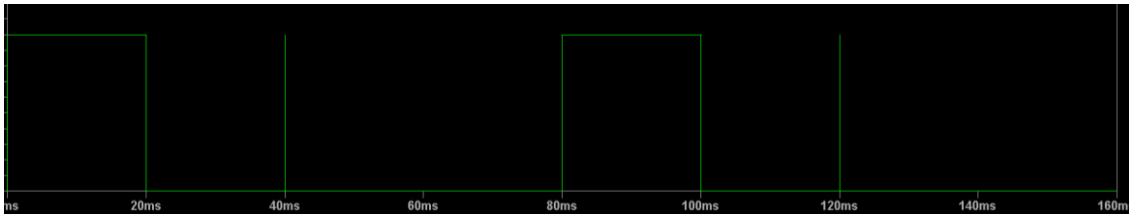


Figura 3.7. Salida óptima de puerta lógica AND

Para comprender el cálculo que se presentará a continuación es importante entender el contenido de la variable outputs. En esta variable encontraremos los valores de la señal que estamos interesados en medir, por ejemplo, en la última figura presentada en esta variable se encontrarían lo valores correspondientes a los tiempos: 10ms, 30ms, 50ms, 70ms, 90ms, 110ms, 130ms y 150ms para tener una medida cada T/2 (siendo T el periodo de la señal de entrada con mayor frecuencia).

Una vez conocidas las salidas óptimas para nuestro circuito y teniendo en cuenta que la primera fila de la matriz corresponderá al voltaje inferior y la segunda al superior, podemos plantear el cálculo de la función objetivo de la manera expuesta a continuación, además la variable outputs se modifica para que el máximo valor en esta sea 1:

$$
\begin{aligned}
f_{objetivo} = {} & (\text{outputs}[0][0] - 1) + (\text{outputs}[0][1] - 1) + (\text{outputs}[0][2] - 1) \\
& + (0 - \text{outputs}[0][3]) + (\text{outputs}[0][4] - 1) + (\text{outputs}[0][5] \\
& - 1) + (\text{outputs}[0][6] - 1) + (0 - \text{outputs}[0][7]) \\
& + (\text{outputs}[1][0] - 1) + (0 - \text{outputs}[1][1]) + (0 \\
& - \text{outputs}[1][2]) + (0 - \text{outputs}[1][3]) + (\text{outputs}[1][4] - 1) \\
& + (0 - \text{outputs}[1][5]) + (0 - \text{outputs}[1][6]) + (0 \\
& - \text{outputs}[1][7])
\end{aligned}
\tag{1}
$$

Con este cálculo, podemos medir la diferencia entre la señal deseada y la señal obtenida del circuito sometido a la función. De forma que un circuito que logre la función deseada

perfectamente, el resultado del cálculo sea 0 y el valor irá decreciendo conforme más lejos esté del funcionamiento óptimo.

3.1.3. Selección de Padres

Una vez se han evaluado todos los elementos de una población debemos seleccionar a los cromosomas que serán los padres de la próxima generación de circuitos, para ello disponemos de diversas posibilidades, entre ellas la selección aleatoria, la selección por torneo y la selección por ruleta. La primera opción que fue descartada fue la aleatoria ya que normalmente conlleva disminuir considerablemente la eficiencia del algoritmo, en cuanto a las otras dos opciones se ha escogido la selección por ruleta debido a que es la más usada generalmente y suele otorgar los mejores resultados.

La función desarrollada para la selección es la mostrada en la Figura 3.8.

```
def roulette(fitness_pop):
    max_fit=max(fitness_pop)
    probability=np.empty(100,float)
    accumulative=np.empty(100,float)
    ranges=np.empty(100,float)
    parent=[[0 for c in range(2)] for b in range(2)]
    mini=min(fitness_pop)
    #Get al the fitness values positive
    for i in range(len(fitness_pop)):
        fitness_pop[i]=fitness_pop[i]+mini

    #Calculating the ranges for the roulette
    totalfit=np.sum(fitness_pop)
    for i in range(len(fitness_pop)):
        probability[i]=fitness_pop[i]/totalfit
        if i>0:
            accumulative[i]=probability[i]+accumulative[i-1]
            ranges[i]=accumulative[i-1]
        else:
            accumulative[i]=probability[i]
            ranges[i]=0
    res=np.sum(probability)

    #Getting the parents from the roulette
    for j in range(2):
        ran=random.uniform(0, 1)
        for i in range(len(fitness_pop)):
            if ran>=ranges[i] and ran<accumulative[i]:
                parent[j]=[i,fitness_pop[i]-mini]
    return parent
```

Figura 3.8.Función de selección por ruleta

Como argumento recibe un vector con el valor de la función objetivo de todos los cromosomas de la población y la función devuelve los índices que en el vector de la población indican a los padres de la siguiente generación, además también devuelve el valor de la función objetivo de cada uno.

Para generar la ruleta, en primer lugar, es necesario asignar una probabilidad a cada individuo mediante su valor de la función objetivo. Además, los valores iniciales son negativos, ya que como se ha comentado antes el valor máximo es 0 y por lo tanto es necesario realizar un cambio para que estos sean positivos. Esto se realiza sumando a los valores de la función objetivo de cada circuito, el mínimo valor de la población, mediante este proceso lograremos que el valor mínimo sea 0.

Una vez contamos con la probabilidad de cada individuo debemos generar la ruleta que utilizaremos para seleccionar a los padres, para ellos generaremos un límite superior e inferior para cada cromosoma basado en la probabilidad que tienen asignada. Para ello se han creado dos variables una de ellas (*accumulative*) guarda la probabilidad acumulada de los individuos, de forma que este será el límite superior de cada cromosoma y otra variable (*ranges*) que contendrá el valor de la variable *accumulative* del anterior individuo, esto corresponde con el límite inferior de cada circuito. Una vez se ha generado la ruleta el siguiente paso es generar dos números aleatorios entre 0 y 1 de forma que los números resultantes pertenezcan a un rango de la ruleta y dichos rangos serán los elegidos como padres para la próxima generación. Ejemplo del proceso:

Tabla 3.1.Cálculo de la probabilidad de cada cromosoma

Cromosomas	Valor de la función objetivo	Probabilidad
A	-2	0.2857
B	-1	0.357
C	-3	0.214

D	-4	0.143
E	-6	0

De forma que al calcular la probabilidad acumulada obtenemos:

Tabla 3.2. Cálculo de límite superior e inferior de los cromosomas

Cromosomas	Probabilidad Acumulada (Límite superior)	Límite inferior
A	0.2857	0
B	0.6427	0.2857
C	0.8567	0.6427
D	1	0.8567
E	1	1

Lo que gráficamente daría lugar a la siguiente figura:

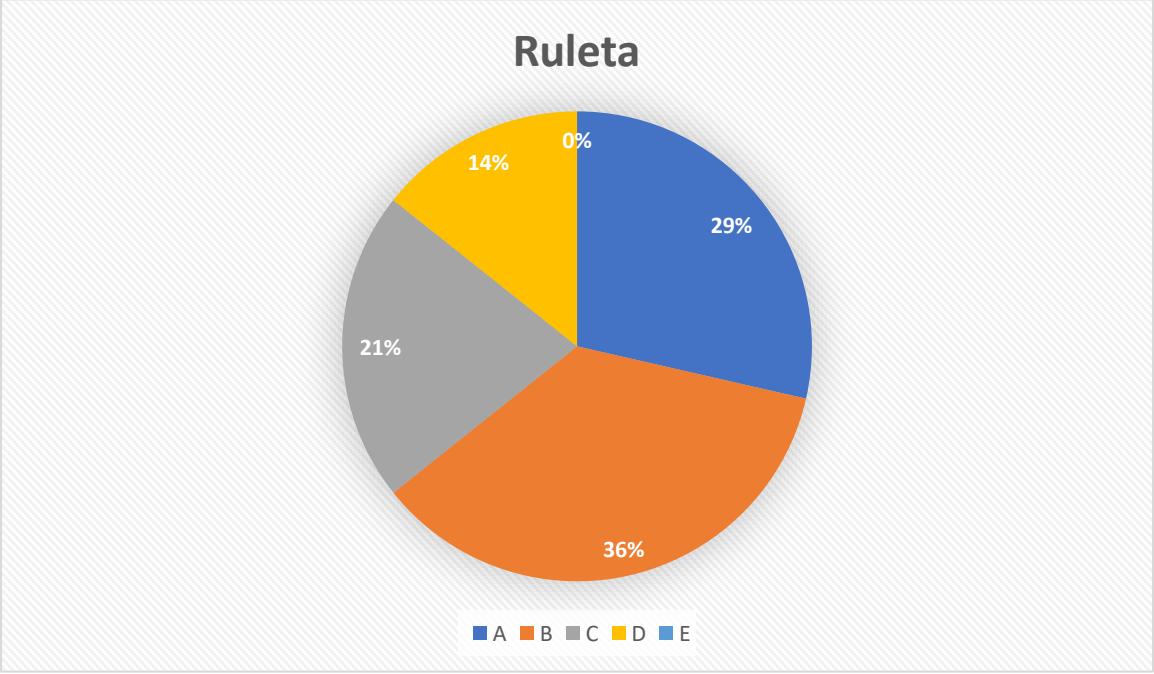


Figura 3.9. Ruleta resultante

A continuación, se generarían 2 números aleatorios entre 0 y 1, por ejemplo, 0.42 y 0.73. Estos números en la última tabla mostrada pertenecen a los rangos B y C respectivamente, por ellos estos serían los seleccionados como padres para generar la próxima población de circuitos polimórficos.

3.1.4. Mezcla de Genes y Mutación

Una vez hemos seleccionado los padres, debemos mezclar los genes de estos y aplicar mutación a algunos para generar la nueva población. En primer lugar, se explicará la mutación ya que esta forma parte de la función de mezcla de genes. Esta función se puede observar en la Figura 3.10.

```
def mutate(parent, lower_limit, upper_limit, step_size=1):
    """
    :param lower_limit: lower limit allowed by a given technology node.
    :param upper_limit: upper limit allowed by a given technology node.
    :param step_size: Determined based on the resolution provided by the foundry. Default set to 1.
    :return: a mutated random width value for a transistor in nanometer (nm) scale
    """
    childGene = parent
    geneSet = [i for i in np.arange(lower_limit, upper_limit, step_size)]
    newGene, alternate = np.random.choice(geneSet, 2)
    childGene = alternate if newGene == childGene else newGene
    return childGene*1e-9
```

Figura 3.10. Función de mutación

Los argumentos que recibe son los mismos que la función generadora de genes añadiendo uno, el gen a mutar (*parent*). La función devuelve el gen mutado, el valor del gen estará entre el rango superior e inferior dado en los argumentos. Para generar este valor la función realiza el siguiente proceso. En primer lugar, se genera un vector que contiene todos los valores entre el límite inferior y superior indicados con la precisión indicada mediante la variable *step_size*, una vez tenemos el vector seleccionamos aleatoriamente 2 valores de este, seleccionamos dos para asegurarnos de no elegir el mismo valor que el previo a la mutación. Una vez tenemos el valor lo multiplicamos por $10^{\wedge -9}$ para que sea del orden del nanómetro.

A continuación, se explicará la función de mezcla de genes mediante la cual se generará la nueva población de circuitos a partir de los padres seleccionados anteriormente. Se explicará la función en dos partes, la primera en la que se aplicar puramente la mezcla de genes, y la segunda donde existe la posibilidad de que exista mutación. La función de manera global recibe como argumentos, los 2 padres y el mejor valor de la función objetivo entre ambos y devuelve la nueva población de circuitos.

```
def crossover(parent1, parent2,bestFitness):
    t1=np.zeros(8)
    t2=np.zeros(8)
    new_population=[[0 for c in range(10)] for b in range(100)]

    for i in range(100):
        child=[[0 for c in range(8)] for b in range(3)]
        used1=np.empty(8,int)
        used2=np.empty(8,int)
        sizes=np.zeros(8)
        ctrl=0
        while(ctrl<7):
            p1=random.randrange(0, 8, 1)
            if p1 in used1:
                while p1 in used1:
                    p1=random.randrange(0, 8, 1)

            p2=random.randrange(10, 18, 1) - 10
            if p2 in used2:
                while p2 in used2:
                    p2=random.randrange(10, 18, 1)-10

            t1=parent1[0][p1]
            sizes[ctrl]=t1
            ctrl+=1
            t2=parent1[0][p2]
            sizes[ctrl]=t2
            ctrl+=1

        child[0]=sizes
        if random.randrange(0, 2, 1) > 0:
            vh=parent1[1]
            vl=parent1[2]
        else:
            vh=parent2[1]
            vl=parent2[2]

        child[1]=vh
        child[2]=vl
```

Figura 3.11.Función mezcla de genes parte 1

El objetivo de esta parte de la función es elegir para cada cromosoma de la nueva población, de que padre se escogerá los valores de cada transistor y de voltaje. Esto se realiza de forma

aleatoria y se guarda en la variable *child* que más adelante se guardará en un vector llamado *new_population*, el cual contendrá todos los cromosomas de la nueva población. La aplicación de la mutación a los nuevos circuitos se ha realizado como se indica en la Figura 3.12.

```
        if random.uniform(0, 10)< (MUTATION_RATE*10):
            for p in range(8):
                if random.uniform(0, 1)< (MUTATION_RATE+0.1):
                    rani=rangoi(child[0][p],bestFitness)
                    low=(child[0][p]-rani)*1e9

                    rans=rangos(child[0][p],bestFitness)
                    up=(child[0][p]+rans)*1e9

                    child[0][p]=mutate(child[0][p], low, up, 10)

        if random.uniform(0, 1)< MUTATION_RATE:
            lowvh=vh-rangoiv(child[1],bestFitness)
            upvh=vh+rangosv(child[1],bestFitness)
            vi=mutate(child[1]*10,lowvh*10,upvh*10,1)
            vh=vi*1e8
            vl=vh-1.5
            child[1]=vh
            child[2]=vl
        new_population[i]=child
    return new_population
```

Figura 3.12. Función mezcla de genes parte 2

La funcionalidad de esta parte consiste en ejercer la mutación en algunos genes. Para ello necesitamos varias cosas, en primer lugar, el ratio en el que ocurrirá la mutación y los limites inferior y superior entre los que pueden variar la anchura de los transistores y el voltaje. El ratio de mutación será una constante que se establecerá en 0.2 para que genere diversidad genética sin convertirse en un algoritmo aleatorio. En el caso de los limites inferior y superior se calculará teniendo en cuenta el mejor valor de la función objetivo, de manera que cuanto más cerca este el circuito de cumplir con su objetivo, la mutación realizará un menor cambio en los genes.

3.1.5. Mostrar el comportamiento del circuito

Una vez se logra que el programa sea capaz de completar el algoritmo con éxito, es importante mostrar en pantalla el comportamiento de los padres de cada población para comprobar el

funcionamiento de la evolución a través de las generaciones. Eso se ha realizado mediante la siguiente función:

```
def display(parent,pos):

    timeDiff = datetime.datetime.now() - startTime
    Guesses=parent[0]
    vh=parent[1]
    vl=parent[2]
    print(Guesses)
    print(vh)
    print(vl)
    fitness = get_fitness(Guesses,vh,vl,"netlist"+str(pos)+"0"+".raw","netlist"+str(pos)+"1"+".raw",
                          "netlist"+str(pos)+"2"+".raw","netlist"+str(pos)+"3"+".raw",True,0)
    print(fitness)
    return fitness
```

Figura 3.13.Función para mostrar el circuito

Esta función recibe como argumentos el circuito a mostrar y su posición en el vector *population* o *new_population*, imprime los genes que se han utilizado y el valor de la función objetivo del circuito. Además, mediante la función *output_parser* que se llama dentro de *get_fitness* se realiza un gráfico de la salida del circuito.

3.2. Interacción con LTspice

Una vez se dispone de un algoritmo genético funcional, es de vital importancia desarrollar funciones que puedan interaccionar con LTspice de manera que podamos obtener los datos necesarios para el algoritmo y simular los circuitos obtenidos, de forma que sea posible comprobar su funcionamiento.

Para el desarrollo de estas funciones, se utilizará la librería de Python PyLTSpice como herramienta para facilitar la interacción con LTspice. A continuación, se presentarán las funciones desarrolladas.

3.2.1. Modificar y simular netlists

En primer lugar, se explicará la función encargada de modificar las netlists y simularlas mediante LTspice en la Figura 3.14.

```
def netlist_modifier(netlist_file,modified_widths,vh,vl,save,n):
    # select spice model
    LTC = SimCommander(netlist_file)
    for i in range(2):
        LTC.set_element_model('M1', 'Vdd PMOS l=1.5u w='+str(modified_widths[0]))
        LTC.set_element_model('M2', 'Vdd PMOS l=100n w='+str(modified_widths[1]))
        LTC.set_element_model('M3', 'Vdd PMOS l=75n w='+str(modified_widths[2]))
        LTC.set_element_model('M4', 'Vdd PMOS l=60n w='+str(modified_widths[3]))
        LTC.set_element_model('M5', 'Vdd PMOS l=100n w='+str(modified_widths[4]))
        LTC.set_element_model('M6', '0 NMOS l=250n w='+str(modified_widths[5]))
        LTC.set_element_model('M7', '0 NMOS l=5.5u w='+str(modified_widths[6]))
        LTC.set_element_model('M8', '0 NMOS l=1u w='+str(modified_widths[7]))
        LTC.set_element_model('M9', 'N025 PMOS l=1.5u w='+str(modified_widths[0]))
        LTC.set_element_model('M10', 'N025 PMOS l=100n w='+str(modified_widths[1]))
        LTC.set_element_model('M11', 'N025 PMOS l=75n w='+str(modified_widths[2]))
        LTC.set_element_model('M12', 'N025 PMOS l=60n w='+str(modified_widths[3]))
        LTC.set_element_model('M13', 'N025 PMOS l=100n w='+str(modified_widths[4]))
        LTC.set_element_model('M14', '0 NMOS l=250n w='+str(modified_widths[5]))
        LTC.set_element_model('M15', '0 NMOS l=5.5u w='+str(modified_widths[6]))
        LTC.set_element_model('M16', '0 NMOS l=1u w='+str(modified_widths[7]))
        LTC.set_component_value('V1', 'PULSE(0 '+str((vh+vl)/2)+' 0 5n 5n 20m 40m)')
        LTC.set_component_value('V2', 'PULSE(0 '+str((vh+vl)/2)+' 0 5n 5n 40m 80m)')
        LTC.set_component_value('V3', str((vh+vl)/2))
        LTC.set_component_value('V12', str((vh+vl)/2))
        LTC.set_component_value('V6', str((vh+vl)/2))

        if i<1:
            LTC.set_component_value('V5', str(vh))
            LTC.set_component_value('V4', str(vh))
        else:
            LTC.set_component_value('V5', str(vl))
            LTC.set_component_value('V4', str(vl))

        LTC.add_instructions(
            "; Simulation settings",
            ".tran 0 80m",
            ".TEMP 27",
            ".print V(O1)"
        )
        if save:
            LTC.write_netlist("netlistf"+n+str(i)+".net")
            LTC.run(run_filename="netlistf"+n+str(i)+".net")
            comp=LTC.wait_completion()
            LTC.reset_netlist()
        else:

            print("Netlist number:"+n+str(i))
            LTC.write_netlist("netlist"+n+str(i)+".net")
            LTC.run(run_filename="netlist"+n+str(i)+".net")
            comp=LTC.wait_completion()
            LTC.reset_netlist()
    return comp
```

Figura 3.14. Función dedicada a trabajar con netlists

La función recibe como argumentos el nombre de la netlist (con la dirección en la que se encuentra), los distintos genes (transistores y voltajes), una variable que indica si queremos guardar dicha netlist para el futuro y por último el número de la netlist para que luego se pueda encontrar fácilmente. Devuelve un comprobante (*comp*) para saber si la simulación de la netlist ha funcionado correctamente.

El funcionamiento completo de la función es el siguiente. Se obtiene la netlist inicial mediante el comando *SimCommnader()*, una vez con ella será necesario crear una netlist nueva por cada voltaje que vayamos a probar (generalmente 2 netlists), además en caso de que necesitemos obtener más de una señal de salida, el número de netlists a generar sería el número de voltajes distintos multiplicado por el número de salidas, ya que cada señal de salida hay que medirla en los dos voltajes. En el caso planteado en la imagen únicamente sería necesario generar dos netlists. Para cada netlist es necesario indicar el valor de los genes recibidos, las instrucciones de simulación y la netlist en la que se escribirá. Una vez hemos incluido todo lo necesario y escrito la netlist el siguiente paso es simularla y esperar a que se complete la simulación y que se genere el archivo. raw que será el que se leerá para obtener la información de la simulación, en caso de que esta tarde demasiado, se interrumpirá y se indicará mediante la variable *comp* que la simulación no ha sido exitosa. Por último, es esencial una vez se genera una netlist nueva reiniciar la inicial para que los cambios no se vean reflejados en esta.

3.2.2. Obtener los valores de la simulación

Con el fin de obtener los valores de cada simulación, se han realizado dos funciones. Una de ellas tiene como objetivo puramente obtener los datos de la simulación, mientras que la otra la misión que tiene es realizar una gráfica que ayude a interpretarlos y enviárselos al algoritmo genético. En primer lugar, se explica la pura obtención de los valores deseados en la Figura 3.15.

```python
def output_points(x,IR):
    time=x.get_time_axis(step=0).tolist()
    rounded_time = [round(num, 3) for num in time]
    out=IR.get_wave(step=0)

    found=False
    ttime=0.020
    while found==False:
        if ttime in rounded_time:
            found=True
        else:
            ttime=round(ttime-0.001,3)
    t1=rounded_time.index(ttime)

    found=False
    ttime=0.040
    while found==False:
        if ttime in rounded_time:
            found=True
        else:
            ttime=round(ttime-0.001,3)
    t2=rounded_time.index(ttime)

    found=False
    ttime=0.041
    while found==False:
        if ttime in rounded_time:
            found=True
        else:
            ttime=round(ttime+0.001,3)
        if (ttime>0.08):
            ttime=-1
            break
    if(ttime!=-1):
        t3=rounded_time.index(ttime)
    else:
        t3=-1


    results=[out[t1],out[t2],out[t3],out[-1]]
    return results
```

Figura 3.15. Función para obtener valores de una simulación

Esta función recibe como argumentos el eje de tiempo y todos los valores obtenidos en la simulación, y devuelve los valores importantes a la hora de medir el comportamiento del circuito. Es decir, devuelve los valores cada T/2 (siendo T el periodo de la señal con mayor frecuencia). Para hallar estos valores se busca en el eje de tiempo los valores que corresponden cada T/2, al no ser un eje de tiempo continuo se buscan los valores cada 0.001 segundos hasta que se encuentra una coincidencia. Una vez se encuentra un tiempo se obtiene su índice en el vector de tiempos de forma que ese mismo índice en el vector de los valores de la simulación resultará en el dato deseado.

Si se dispone de una función capaz de obtener los valores de tiempo deseados, ya solo es necesario representar la gráfica y mandarle los datos al algoritmo genético para que ejerza su función de la mejor forma posible. Por lo tanto, se realizará la siguiente función que cumplirá con esos requisitos tal y como se aprecia en la Figura 3.16.

```
def output_parser(output_file1,output_file2,paint):
    LTR0 = LTSpiceRawRead(output_file1)
    LTR1 = LTSpiceRawRead(output_file2)
    IR0 = LTR0.get_trace("V(out)")
    IR1 = LTR1.get_trace("V(out)")
    x0 = LTR0.get_trace('time')  # Gets the time axis
    x1 = LTR1.get_trace('time')  # Gets the time axis
    if paint:
        steps0 = LTR0.get_steps()
        steps1 = LTR1.get_steps()
        for step in range(len(steps0)):
            plt.plot(x0.get_time_axis(step), IR0.get_wave(step))

        for step in range(len(steps1)):
            plt.plot(x1.get_time_axis(step), IR1.get_wave(step))

        plt.legend()  # order a legend
        plt.show()

    v0=output_points(x0, IR0)
    v1=output_points(x1, IR1)
    measurements=[v0,v1]
    return measurements
```

Figura 3.16. Función para mostrar gráfica de simulación y comunicarse con el algoritmo

La función recibe los archivos .raw de los que se quieren obtener los valores y una variable que indica si se requiere una representación gráfica de la simulación o no. Además, devuelve los valores deseados de la simulación. Para ello se ha de utilizar la última función mencionada (*output_points()*) y por lo tanto se debe obtener el vector de tiempos y el de valores de la simulación. Esto se logra utilizando los comandos de la librería PyLTSpice que leen el archivo .raw y obtienen los vectores adecuados, estos son *LTSpiceRawRead()* y *LTSpiceRawRead().getTrace()* respectivamente. Una vez con estos vectores, se puede utilizar la función *output_points()* para obtener los valores serán utilizados por el algoritmo principal. Además, en caso de que la variable *paint* lo indique se pintará una gráfica con los valores obtenidos de la simulación.

3.3. Ejecución de las funciones en el algoritmo

Una vez se han desarrollado todas las funciones necesarias, estas han de ordenarse para que el programa se pueda ejecutar y obtenga unos resultados óptimos. Esto se ha realizado de la siguiente forma.

```
if __name__=='__main__':
    newParent=0
    cont=0
    random.seed()
    startTime = datetime.datetime.now()
    bestFitness=-20
    globalBestFitness=-20
    population=population_generator()
    fitness_pop=np.empty(len(population),float)
    childs=population
    while bestFitness<-0.6:
        cont+=1
        print("New Population")
        print("Number: "+str(cont))
        for i in range(len(childs)):
            chromosome=childs[i]
            fitness_pop[i]=get_fitness(chromosome[0], chromosome[1], chromosome[2], "netlist"+str(i)+"0"+".raw",
                                       "netlist"+str(i)+"1"+".raw","netlist"+str(i)+"2"+".raw","netlist"+str(i)+"3"+".raw", False,i)
        ind=roulette(fitness_pop)
        parent1=population[ind[0][0]]
        fit1=display(parent1,ind[0][0])
        parent2=population[ind[1][0]]
        fit2=display(parent2,ind[1][0])
        bestFitness=max(fit1,fit2)
        if bestFitness > globalBestFitness:
            globalBestFitness=bestFitness
            if fit1 > fit2:
                bestParent=parent1
            else:
                bestParent=parent2
            netlist_modifier("C:\\Users\\Luis\\Documents\\LTspiceXVII\Draft9.asc",
                        bestParent[0],bestParent[1],bestParent[2],True,str(newParent))
            newParent+=1
        childs=crossover(parent1,parent2,bestFitness)
    endt=datetime.datetime.now()
    print("Algorithm finished")
    print("Number: "+str(cont))
    print("Total time = "+str(endt-startTime))
```

Figura 3.17. Función main

En primer lugar, se obtiene la población inicial mediante la función *population_generator()*, una vez con la primera población generada, comienza un bucle que continuará hasta que el valor objetivo de un padre sea superior a -0.6 u otro valor indicado. Cada iteración del bucle realizará en primer lugar una evaluación de los individuos en la población, para ello utilizará la función *get_fitness()*, una vez con el vector de evaluaciones (*fitness_pop*) actualizado se seleccionarán

a los padres de la siguiente población mediante la función *roulette()*. Con los padres ya seleccionados, se muestra el gráfico de la/s señales de salidas de estos y se comprueba si han mejorado el mejor valor de la función objetivo hasta el momento, en caso de que así sea se guarda el padre que lo haya logrado en una netlist para que sea fácilmente accesible. Tras esto, se mezclan y mutan los genes de los padres seleccionados mediante la función *crossover* para generar una nueva población de circuitos y se vuelve a empezar el bucle. En caso de que el bucle termine debido a que el valor de evaluación sea superior al valor establecido, se indica el número de poblaciones y el tiempo que ha sido necesario para completar el algoritmo.

4. Desarrollo de un circuito polimórfico mediante el algoritmo

Una vez se ha desarrollado un algoritmo genético capaz de evolucionar circuitos debemos ponerlo a prueba desarrollando un circuito polimórfico. Para ello se debe plantear que circuito polimórfico se desarrollará. En este caso se tratará de desarrollar un circuito que tenga dos comportamientos dependiendo del voltaje, en caso de que el voltaje sea inferior a cierto punto el circuito debe comportarse como una puerta lógica NOR y cuando sea superior al mismo se comporte como una puerta NAND.

Una vez conocemos el circuito que queremos desarrollar debemos aplicar los cambios necesarios en el algoritmo desarrollado para que este sea capaz de alcanzar el resultado deseado. Además, debido a que el algoritmo no evoluciona la estructura del circuito también se debe escoger la estructura con la que se trabajará.

4.1. Selección de estructura del circuito

Para la elección de la estructura se realizó una investigación sobres algunos circuitos polimórficos desarrollados anteriormente para ponerlas a prueba y comprobar cual otorgaba un mejor resultado tras evolucionar los diferentes genes. Algunas de las estructuras planteadas son:

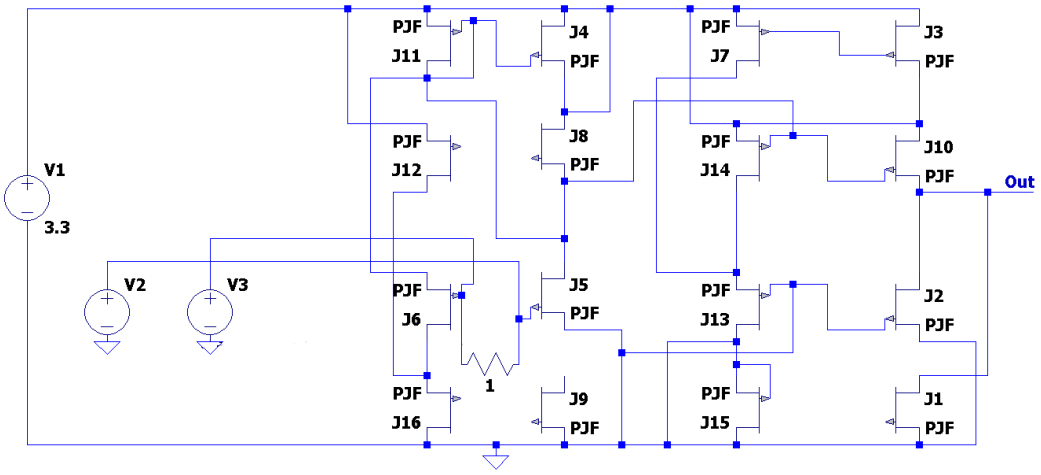


Figura 4.1. Estructura de circuito A

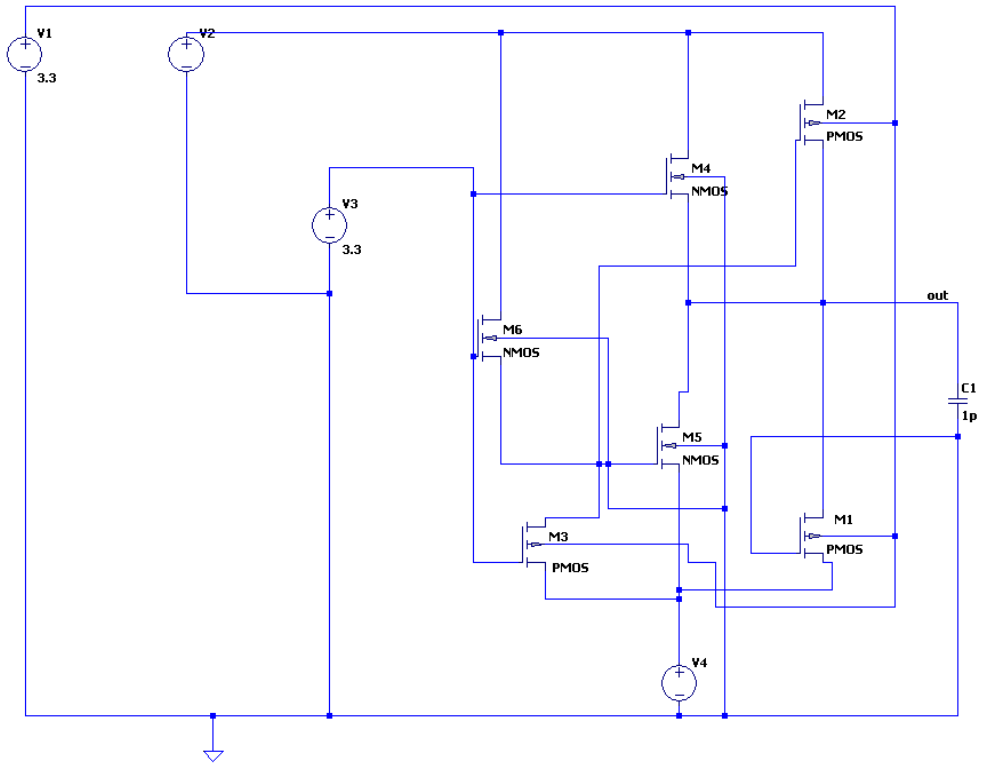


Figura 4.2. Estructura de circuito B

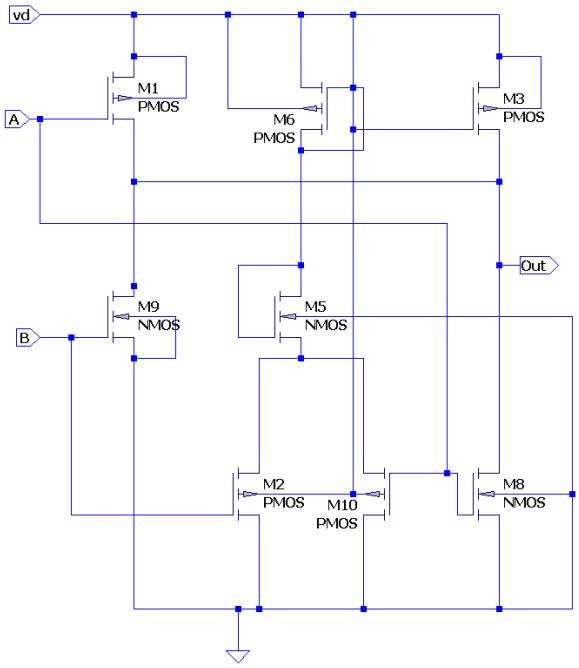


Figura 4.3. Estructura de circuito C

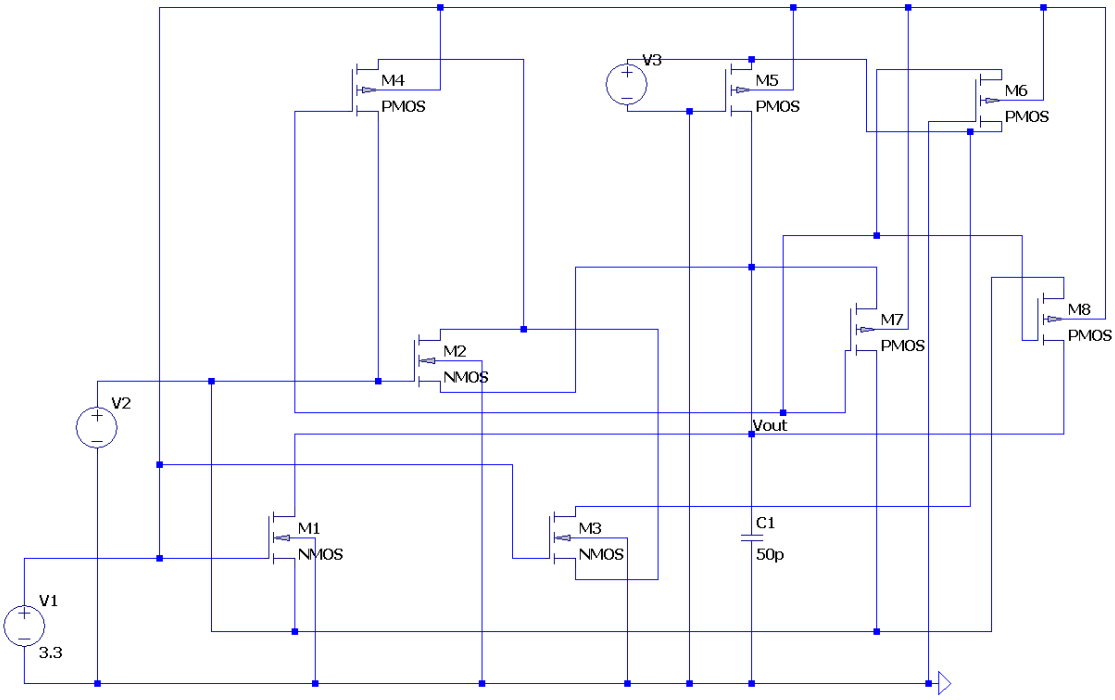


Figura 4.4. Estructura de circuito D

Finalmente, tras someter todas estas estructuras al algoritmo genético la única con la que se consiguió un resultado óptimo fue la opción C, por lo tanto, está es la estructura escogida para el desarrollo del circuito polimórfico. E las siguientes secciones se presentarán las modificaciones necesarias para el funcionamiento del algoritmo y un análisis de los resultados obtenidos.

Además, también se debe escoger el tipo de transistor a utilizar, en este caso se utilizará el transistor PTM 45nm cuyas especificaciones vienen en el anexo 2. Debido a que permite un tamaño de transistor muy pequeño utilizando la tecnología moderna.

4.2. Modificaciones al algoritmo

Para que el algoritmo se adapte a la estructura planteado y el circuito objetivo se deben plantear algunos cambios, entre estos están, el cálculo de la función objetivo para los circuitos o el número de transistores utilizados.

4.2.1. Cálculo de la función objetivo

Para plantear este cálculo, debemos evaluar las señales de entrada que tendrá el circuito y la señal de salida esperada dependiendo de la fuente de voltaje ya que, si se visualiza esto se facilita el desarrollo de la fórmula adecuada. Se utilizarán las mismas señales de entrada que se plantearon en el ejemplo inicial como se ve en la Figura 4.5 y la Figura 4.6.

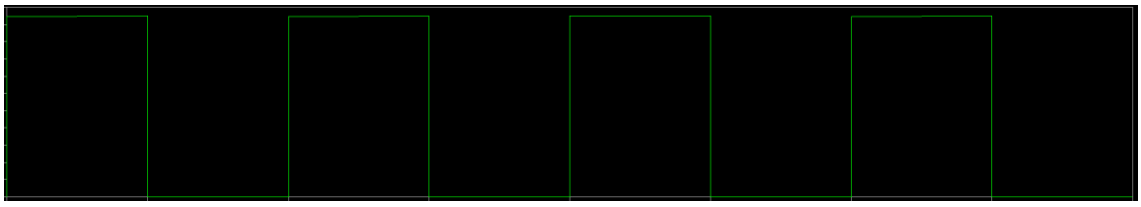


Figura 4.5. Señal de entrada 1

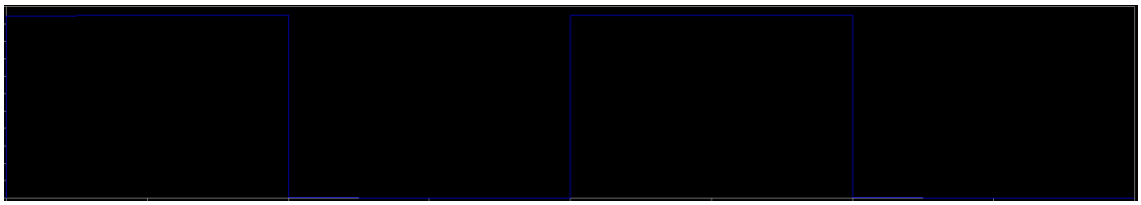


Figura 4.6. Señal de entrada 2

Con estas señales la señal a la salida que esperamos ver cuando el voltaje suministrado al circuito es inferior al punto x es la indicada en la Figura 4.7.

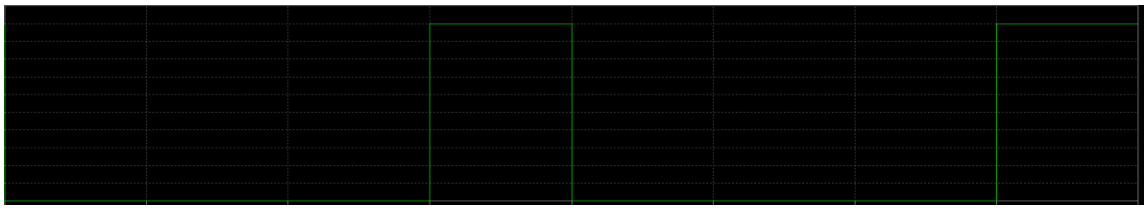


Figura 4.7. Señal de salida óptima para el voltaje inferior

Y cuando el voltaje es superior a dicho punto la salida deberá ser la indicada en la Figura 4.8.

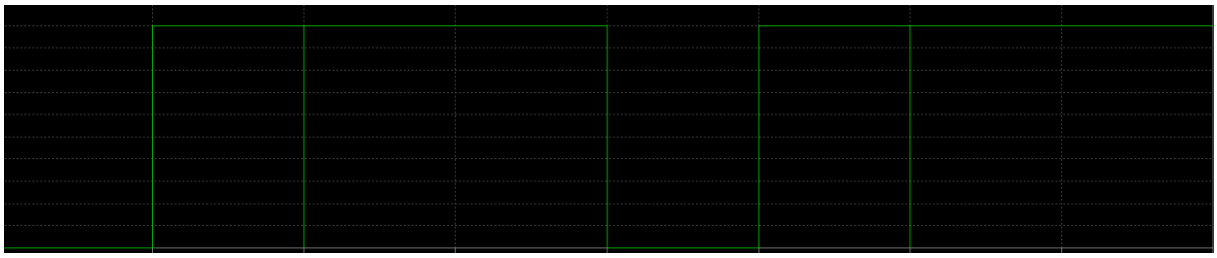


Figura 4.8. Señal de salida óptima para el voltaje superior

De forma que los valores óptimos para el vector en caso del voltaje inferior serían: [0,0,0,1,0,0,0,1], y en el caso del voltaje superior: [0,1,1,1,0,1,1,1]. Por lo tanto, el cálculo a realizar para la función objetivo, teniendo en cuenta que en la primera fila de la matriz outputs se encuentra el vector que contiene los valores de la simulación para el voltaje inferior, sería el mostrado en la fórmula (2).

$$f_{objetivo} = (0 - outputs[0][0]) + (0 - outputs[0][1]) + (0 - outputs[0][2]) + (outputs[0][3] - 1) + (0 - outputs[0][4]) + (0 - outputs[0][5]) + (0 - outputs[0][6]) + (outputs[0][7] - 1) + (0 - outputs[1][0]) + (outputs[1][1] - 1) + (outputs[1][2] - 1) + (outputs[1][3] - 1) + (0 - outputs[1][4]) + (outputs[1][5] - 1) + (outputs[1][6] - 1) + (outputs[1][7] - 1) \quad (2)$$

4.2.2. Edición de la netlist

Debido a que tenemos definida la estructura del circuito a utilizar, podemos especificar como queremos editar la netlist con la función netlist_modifier(), de forma que se generen netlists partiendo de la estructura inicial utilizando los genes indicados por el algoritmo.

Partiendo de la estructura seleccionada, la función desarrollada es la mostrada en la Figura 4.9.

```
def netlist_modifier(netlist_file,modified_widths,vh,vl,save,n):
    # select spice model
    LTC = SimCommander(netlist_file)
    for i in range(2):
        if i==0:
            v=vl
        else:
            v=vh
        LTC.set_element_model('M1', 'N001 PMOS L=1.5u w='+str(modified_widths[0]))
        LTC.set_element_model('M2', 'N001 PMOS L=100n w='+str(modified_widths[1]))
        LTC.set_element_model('M3', 'N001 PMOS L=75n w='+str(modified_widths[2]))
        LTC.set_element_model('M6', 'N001 PMOS L=60n w='+str(modified_widths[3]))
        LTC.set_element_model('M10', 'N001 PMOS L=100n w='+str(modified_widths[4]))
        LTC.set_element_model('M5', '0 NMOS L=250n w='+str(modified_widths[5]))
        LTC.set_element_model('M8', '0 NMOS L=5.5u w='+str(modified_widths[6]))
        LTC.set_element_model('M9', '0 NMOS L=1u w='+str(modified_widths[7]))
        LTC.set_component_value('V1', 'PULSE(0 '+str((vh+vl)/2)+' 0 5n 5n 20m 40m)')
        LTC.set_component_value('V3', 'PULSE(0 '+str((vh+vl)/2)+' 0 5n 5n 40m 80m)')
        LTC.set_component_value('V2', str(v))

        LTC.add_instructions(
            "; Simulation settings",
            ".tran 0 80m",
            ".TEMP 27",
            ".print V(out)"
        )
        if save:
            LTC.write_netlist("netlistf"+n+str(i)+".net")
            LTC.run(run_filename="netlistf"+n+str(i)+".net")
            comp=LTC.wait_completion()
            LTC.reset_netlist()
        else:

            print("Netlist number:"+n+str(i))
            LTC.write_netlist("netlist"+n+str(i)+".net")
            LTC.run(run_filename="netlist"+n+str(i)+".net")
            comp=LTC.wait_completion()
            LTC.reset_netlist()
    return comp
```

Figura 4.9. Función netlist_modifier modificada

Únicamente ha sido necesario cambiar los transistores, el resto puede permanecer igual ya que las señales de entrada son las mismas.

4.3. Ejecución del programa

Una vez se han realizado los cambios pertinentes para la estructura propuesta, el siguiente paso es ejecutar el programa para obtener el circuito deseado. Antes de mostrar los resultados obtenidos se mencionarán los parámetros importantes relacionados con el algoritmo genético que se han utilizado. Entre estos están:

- Ratio de mutación: Para este valor se ha escogido 0.2, ya que como se mencionó anteriormente otorga un buen equilibrio entre diversidad genética y evitar demasiada aleatoriedad en el proceso.
- Tamaño de la población: Para este tipo de algoritmo se necesita una muestra bastante grande que permita una muestra de población aceptable, se ha escogido un tamaño de 100 individuos ya que el tiempo de ejecución del algoritmo ya es bastante elevado y 100 individuos ya aporta un resultado bastante bueno tras varias generaciones.
- Valor de la función objetivo a partir del cual se considerará suficientemente bueno el circuito: Para decidir cuándo terminar el algoritmo, se ha marcado este valor como -0.5, ya que los circuitos con una evaluación superior a esta presentan un comportamiento muy cercano al deseado.
- Anchura de los transistores: Para el valor de estos se ha determinado un rango entre 100 nanómetros y 5 micrómetros.
- Modelo de transistor: Se ha seleccionado el transistor PTM 45nm BSIM4.
- Fuente de voltaje: El rango seleccionado es para el voltaje superior entre 2 y 10 voltios y para el voltaje inferior entre 0.5 y 8.5 voltios.

Al ejecutar el programa, han de pasar varias generaciones de circuitos para obtener el circuito deseado. El número de generaciones varía mucho, ya que depende en gran medida de la población inicial que se genera de manera aleatoria y de las mutaciones que ocurran que también son aleatorias. A continuación, se presenta un ejemplo de la evolución del circuito a través de las generaciones. Marcando en estas el valor de la función objetivo de forma que se pueda observar de forma más sencilla la evolución en cada población. Además, para el tiempo de simulación, se utilizará un periodo de la señal de entrada con menor frecuencia, en este caso la mostrada en la Figura 4.6, con esto se logra reducir el tiempo de ejecución del algoritmo. Esto implica que las señales de salida objetivo mostradas en la Figura 4.7 y la Figura 4.8 también incluirán únicamente un periodo.

En primer lugar, se presentan los padres que se han seleccionado de la población inicial y que darán lugar a la segunda generación. En las figuras que se mostrarán, la gráfica naranja

representa el comportamiento del circuito al voltaje superior mientras que la azul lo representa al voltaje inferior.

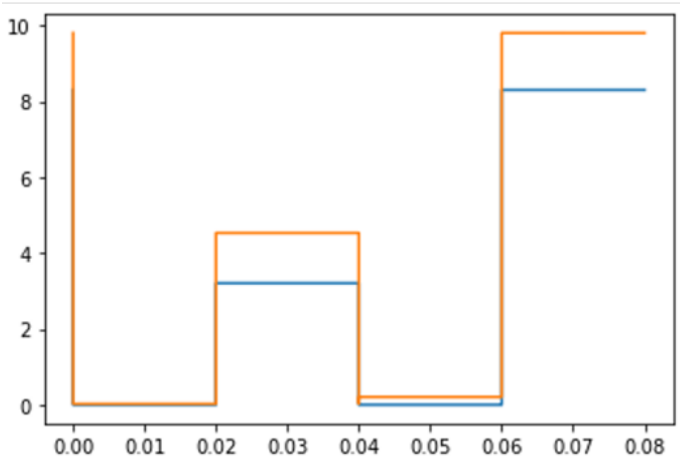


Figura 4.10. Primera generación, padre 1

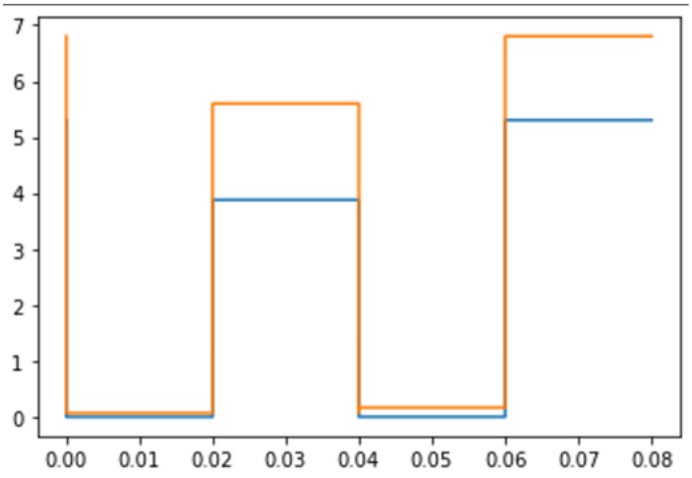


Figura 4.11. Primera generación, padre 2

Debido a que el resultado obtenido en esta generación no cumple con el objetivo, se repite el proceso. A continuación, se muestran en la Figura 4.12 y Figura 4.13 los padres de la siguiente población.

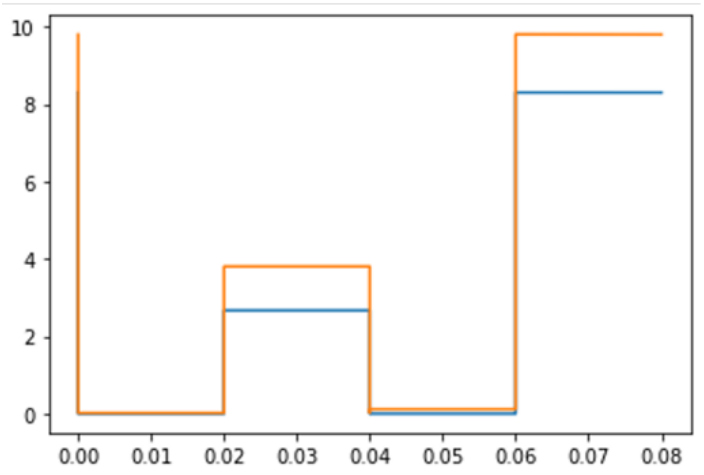


Figura 4.12. Segunda generación, padre 1

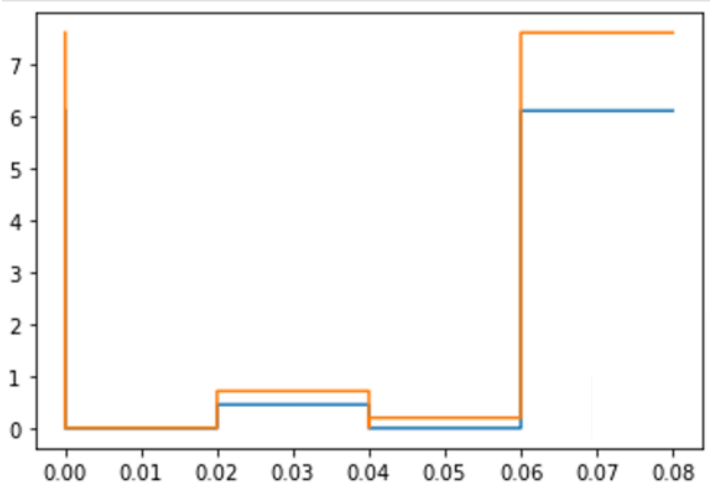


Figura 4.13. Segunda generación, padre 2

De nuevo, el resultado tras evaluarlo continúa sin ser el deseado y se repite el algoritmo de forma que obtenemos los padres de la siguiente generación y sus respectivas simulaciones en la Figura 4.14 y la Figura 4.15.

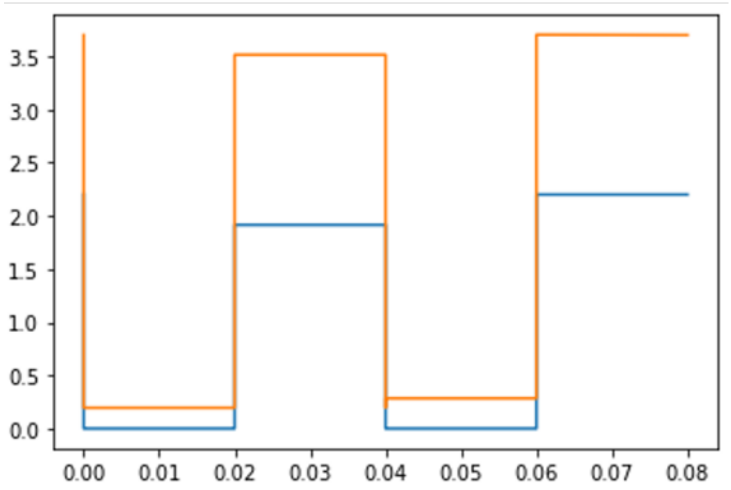


Figura 4.14. Tercera generación, padre 1

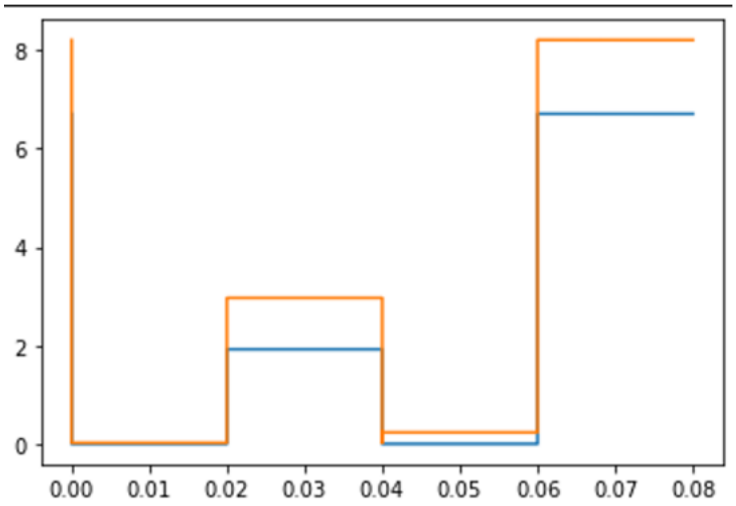


Figura 4.15. Tercera generación, padre 2

De nuevo, los resultados obtenidos continúan sin parecerse al objetivo del algoritmo y por lo tanto se continúa con la ejecución.

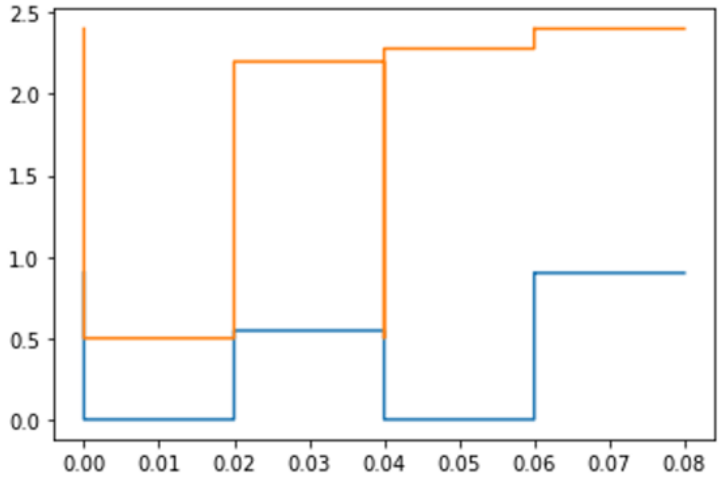


Figura 4.16. Cuarta generación, padre 1

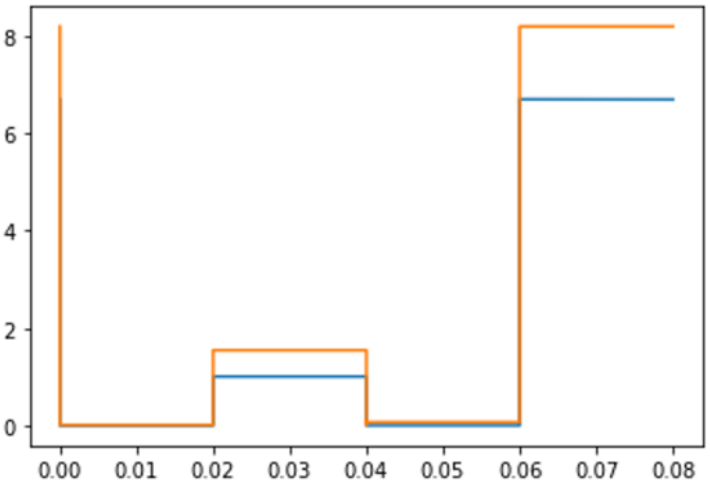


Figura 4.17. Cuarta generación, padre 2

En esta generación, ya podemos observar en la Figura 4.16 que el padre 1 de esta generación se comienza a acercar al objetivo del algoritmo, a pesar de esto el algoritmo se repetirá una vez por que aún el valor de la función objetivo de dicho circuito no es lo suficientemente bueno.

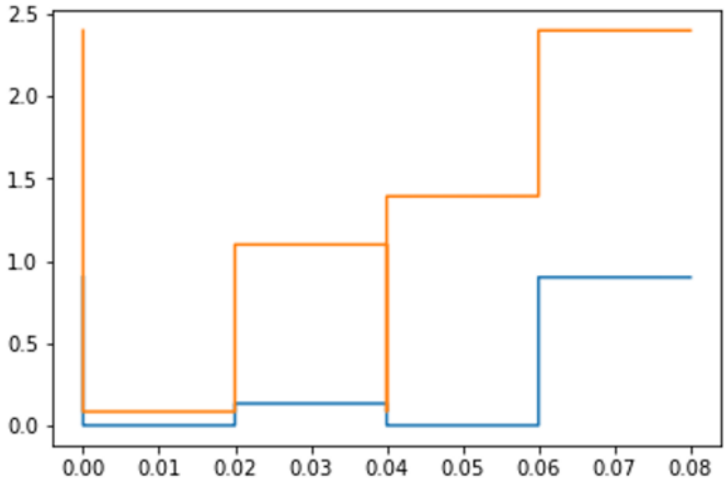


Figura 4.18. Padre de la quinta generación y circuito final

El algoritmo se detuvo aquí, ya que podemos observar que el comportamiento ya cumple con el objetivo y por lo tanto hemos logrado obtener el circuito deseado capaz de cambiar su comportamiento dependiendo del voltaje suministrado. En el caso de este circuito, para un voltaje inferior a 1.5 voltios se comportará como una puerta lógica NOR y para un voltaje superior a 1.6 voltios se comportará como una puerta NAND. Por lo tanto, el algoritmo desarrollado se muestra funcional y capaz de desarrollar el circuito polimórfico deseado, una vez cumpliendo con este objetivo, se tratará de implementarlo en un bloque más grande para que sometiéndolo de nuevo al algoritmo genético, obtener una solución al problema de seguridad planteado al inicio del proyecto.

4.3.1. Evolución del valor de la función objetivo durante las generaciones

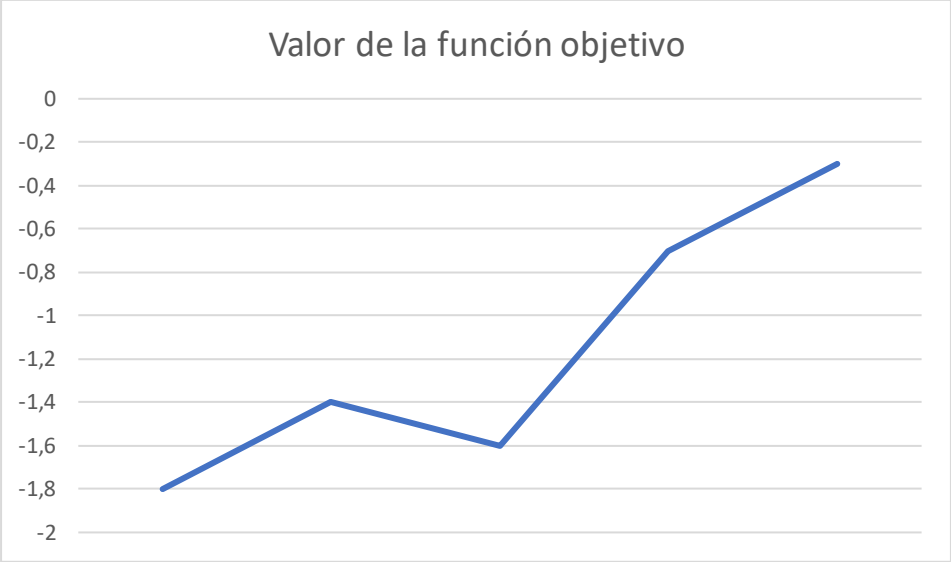


Figura 4.19. Evolución del valor de la función objetivo

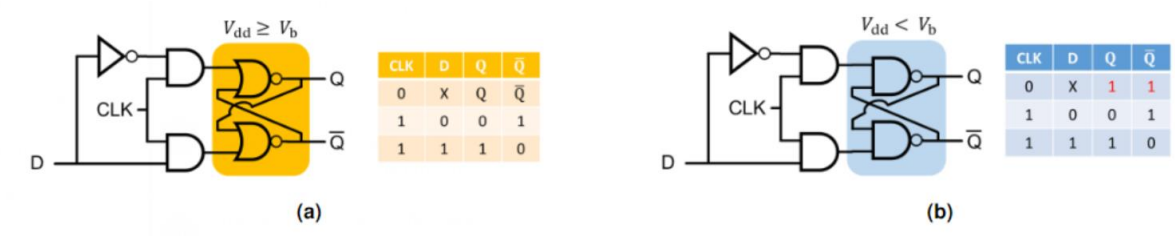
4.3.2. Análisis del desarrollo del circuito y el circuito final

Tras un número reducido de generaciones, se ha obtenido el circuito deseado, el bajo número de generaciones probablemente se deba a un factor suerte a la hora de la población inicial y en la mezcla de genes y mutación. En cuanto al circuito final, podemos observar que cumple ampliamente con el comportamiento deseado del circuito. Esto se puede apreciar ya que para el voltaje superior el circuito se comporta como una puerta NAND y para el inferior el comportamiento es casi idéntico a una puerta NOR.

5. Desarrollo Flip-Flop polimórfico

En esta etapa, el objetivo es desarrollar un flip-flop cuyo comportamiento varie dependiendo del voltaje que se utilice. Este flip-flop se utilizará más adelante para solucionar el problema de seguridad mencionado anteriormente. La faceta de la seguridad se explicará en secciones posteriores, este apartado se centrará en explicar exclusivamente el desarrollo del circuito y su funcionamiento. El circuito que queremos generar está basado en un flip-flop tipo D.

Además, se quiere que el circuito altere su comportamiento ante determinadas situaciones. En caso de que el voltage suministrado sea inferior o superior a cierto punto el circuito se comportará como es mostrado en la Figura 5.1.



CLK	D	Q	$\overline{Q}$
0	X	Q	$\overline{Q}$
1	0	0	1
1	1	1	0

CLK	D	Q	$\overline{Q}$
0	X	1	1
1	0	0	1
1	1	1	0

Figura 5.1. Comportamiento deseado del flip-flop

Como se puede observar en la Figura 5.1, para el voltaje superior dos puertas lógicas deberán comportarse como NAND y para el inferior deberán comportarse como NOR. Debido a que los dos subcircuitos a evolucionar son puertas lógicas NOR/NAND, se utilizará el circuito desarrollado en la sección anterior como base para que con ligeras modificaciones seamos capaces de desarrollar el circuito necesario. Esto lo haremos utilizando la población final de la anterior sección como población inicial de este algoritmo. A pesar de esto, debemos realizar algunas modificaciones en el código. Pero antes se presentará la estructura completa del circuito en LTspice para comprender las modificaciones.

5.1. Estructura del circuito

En este caso el circuito está formado por varias puertas lógicas además de las formadas mediante el algoritmo genético. Para estas puertas no es necesario utilizar el algoritmo, de forma que se han utilizado módulos comunes con estas funciones. En el circuito a parte de los módulos polimórficos se utilizan dos puertas AND y un NOT. Estos se han realizado como viene indicado en la Figura 5.2 y la Figura 5.3 respectivamente.

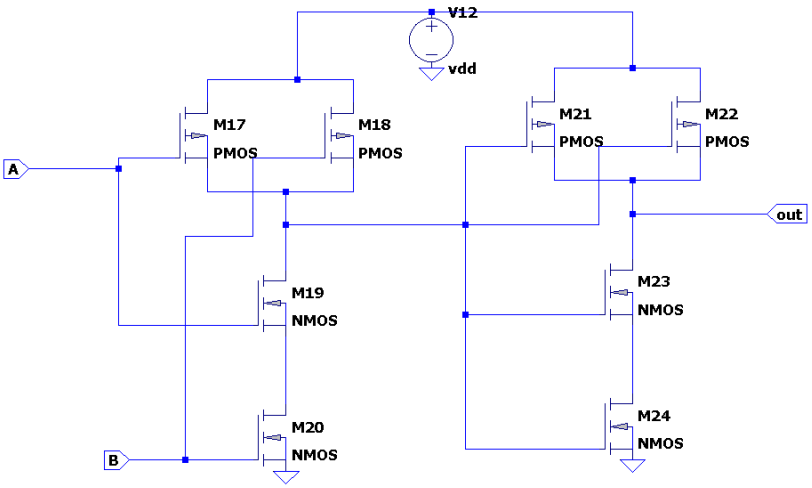


Figura 5.2. Estructura AND

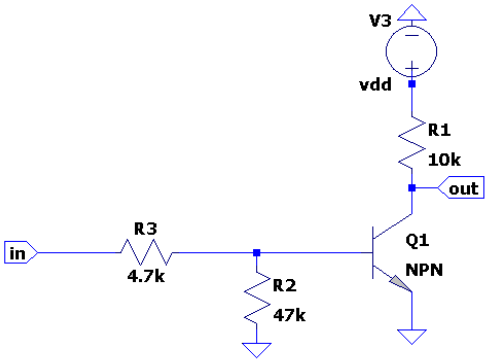


Figura 5.3. Estructura NOT

Una vez se conocen las puertas lógicas que se usarán en el circuito, se plantea la estructura al completo del mismo en la Figura 5.4.

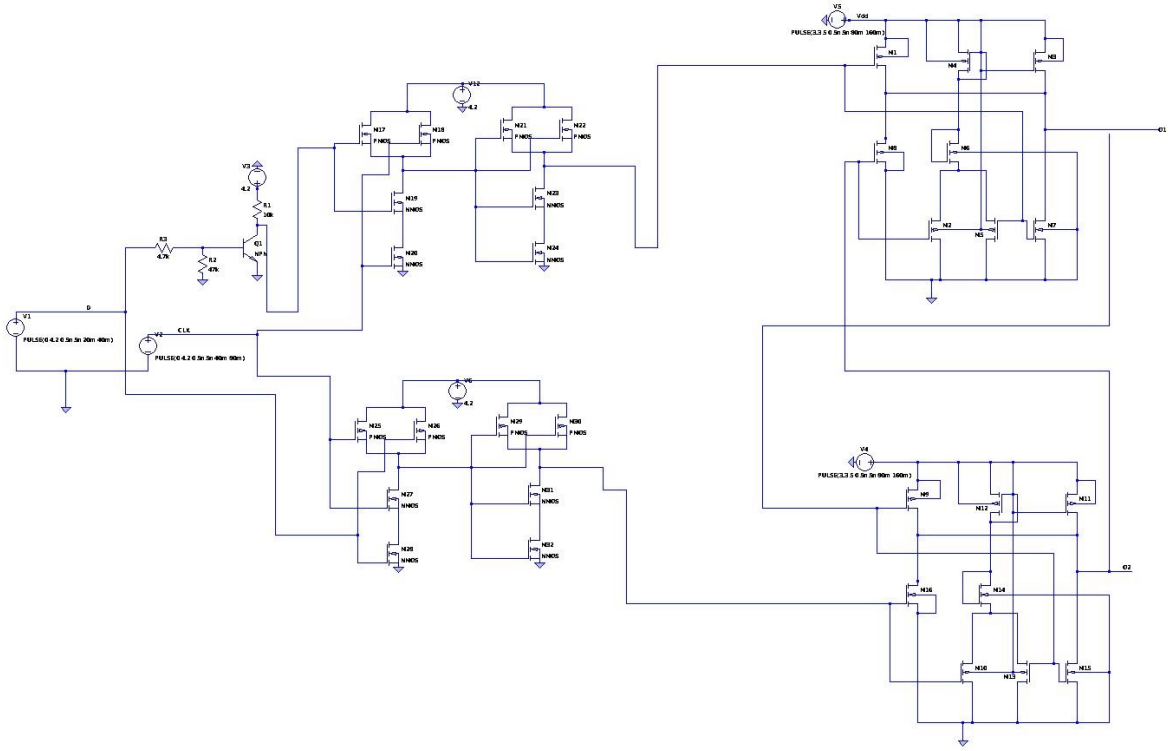


Figura 5.4. Estructura completa

El modelo de transistor que se utilizará será el mismo que el utilizado en el desarrollo de la puerta polimórfica NOR/NAND.

5.2. Modificaciones al algoritmo para el Flip-Flop

Para adaptar el algoritmo a este circuito, debemos adaptar esencialmente las mismas partes que en la sección 4.2 ya que el resto del algoritmo es constante e invariable independientemente del circuito a desarrollar.

5.2.1. Edición de la netlist

Al igual que al desarrollar la puerta lógica polimórfica, se deben realizar algunos cambios en la función encargada de modificar la netlist para que el programa funcione correctamente. Además, al tratar con dos salidas diferentes lo que se va a hacer es que la fuente que suministra voltaje a las puertas polimórficas será una señal que comience en el voltaje superior y tras un periodo de las señales de salida, cambie al voltaje inferior, esto logrará que solo sea necesario generar 2 netlists en vez de 4. Estos cambios se pueden apreciar en la Figura 5.5 en las fuentes de voltaje V4 y V5.

```
def netlist_modifier(netlist_file,modified_widths,vh,vl,save,n):
    # select spice model
    LTC = SimCommander(netlist_file)
    for i in range(2):
        LTC.set_element_model('M1', 'Vdd PMOS L=1.5u W='+str(modified_widths[0]))
        LTC.set_element_model('M2', 'Vdd PMOS L=100n W='+str(modified_widths[1]))
        LTC.set_element_model('M3', 'Vdd PMOS L=75n W='+str(modified_widths[2]))
        LTC.set_element_model('M4', 'Vdd PMOS L=60n W='+str(modified_widths[3]))
        LTC.set_element_model('M5', 'Vdd PMOS L=100n W='+str(modified_widths[4]))
        LTC.set_element_model('M6', '0 NMOS L=250n W='+str(modified_widths[5]))
        LTC.set_element_model('M7', '0 NMOS L=5.5u W='+str(modified_widths[6]))
        LTC.set_element_model('M8', '0 NMOS L=1u W='+str(modified_widths[7]))
        LTC.set_element_model('M9', 'N025 PMOS L=1.5u W='+str(modified_widths[0]))
        LTC.set_element_model('M10', 'N025 PMOS L=100n W='+str(modified_widths[1]))
        LTC.set_element_model('M11', 'N025 PMOS L=75n W='+str(modified_widths[2]))
        LTC.set_element_model('M12', 'N025 PMOS L=60n W='+str(modified_widths[3]))
        LTC.set_element_model('M13', 'N025 PMOS L=100n W='+str(modified_widths[4]))
        LTC.set_element_model('M14', '0 NMOS L=250n W='+str(modified_widths[5]))
        LTC.set_element_model('M15', '0 NMOS L=5.5u W='+str(modified_widths[6]))
        LTC.set_element_model('M16', '0 NMOS L=1u W='+str(modified_widths[7]))
        LTC.set_component_value('V1', 'PULSE(0 1 0 5n 5n 20m 40m)')
        LTC.set_component_value('V2', 'PULSE(0 1 0 5n 5n 40m 80m)')
        LTC.set_component_value('V3', '1')
        LTC.set_component_value('V12', '1')
        LTC.set_component_value('V6', '1')
        LTC.set_component_value('V4', 'PULSE('+str(vl)+' '+str(vh)+' 0 5n 5n 80m 160m)')
        LTC.set_component_value('V5', 'PULSE('+str(vl)+' '+str(vh)+' 0 5n 5n 80m 160m)')

        if i==0:
            LTC.add_instructions(
                "; Simulation settings",
                ".tran 0 160m",
                ".TEMP 27",
                ".print V(01)"
            )
        else:
           LTC.add_instructions(
                "; Simulation settings",
                ".tran 0 160m",
                ".TEMP 27",
                ".print V(02)"
            )
        if save:
            LTC.write_netlist("netlistf"+n+str(i)+".net")
            LTC.run(run_filename="netlistf"+n+str(i)+".net")
            comp=LTC.wait_completion()
            LTC.reset_netlist()
        else:

            print("Netlist number:"+n+str(i))
            LTC.write_netlist("netlist"+n+str(i)+".net")
            LTC.run(run_filename="netlist"+n+str(i)+".net")
            comp=LTC.wait_completion()
            LTC.reset_netlist()
    return comp
```

Figura 5.5. Función netlist_modifier para flip-flop

Entre los cambios que se han realizado a la función, se ha tenido que añadir el doble de transistores debido a que en este caso contamos con dos puertas polimórficas, de forma que del transistor M1 al M8 son pertenecientes a la primera puerta y del M9 al M16 forman parte

de la segunda, de todas formas, solo se calcularán las anchuras de 8 transistores, ya que las puertas ambas puertas deben de ser iguales. A parte de esto, lo mencionado anteriormente para incluir tanto el voltaje inferior como el superior en cada netlist, utilizando una señal que varie entre estos.

5.2.2. Obtención de valores de la simulación

Debido a que se deben obtener los valores simulados de dos señales diferentes, se deben realizar algunos cambios a la función encargada de obtener los vectores estos circuitos y de realizar una gráfica con ellos (*output_parser()*). Los cambios se pueden apreciar en la Figura 5.6.

```
def output_parser(output_file1,output_file2,paint):
    LTR0 = LTSpiceRawRead(output_file1)
    LTR1 = LTSpiceRawRead(output_file2)
    IR0 = LTR0.get_trace("V(o1)")
    IR1 = LTR1.get_trace("V(o2)")
    x0 = LTR0.get_trace('time')  # Gets the time axis
    x1 = LTR1.get_trace('time')  # Gets the time axis
    v0=output_points(x0, IR0)
    v1=output_points(x1, IR1)
    if paint:
        steps0 = LTR0.get_steps()
        steps1 = LTR1.get_steps()
        for step in range(len(steps0)):
            plt.plot(x0.get_time_axis(step), IR0.get_wave(step), label='Q')

        for step in range(len(steps1)):
            plt.plot(x1.get_time_axis(step), IR1.get_wave(step), label='nQ')

        plt.legend()  # order a legend
        plt.show()

    v0l=output_points(x0, IR0)
    v1l=output_points(x1, IR1)
    measurements=[v0,v1]
    return measurements
```

Figura 5.6. Función output_parser para flip-flop

Los cambios realizados a la función para trabajar con dos señales de salida en vez de una, son principalmente cambiar el nombre de las señales a capturar. Además, se han añadido etiquetas en las gráficas para que se aprecie con mayor facilidad que señal es cual.

5.2.3. Cálculo de la función objetivo

Para este circuito, de nuevo debemos actualizar este cálculo debido a que no solo tenemos dos salidas en vez de una, sino que ninguna de ellas es la misma que la esperada en el último circuito desarrollado. Para ayudar a la explicación de esta fórmula se mostrarán de nuevo las señales deseadas a las salidas en cada voltaje. Para ello, se tendrá en cuenta que las señales que se utilizarán como entrada son las mostradas anteriormente en la Figura 4.5 y la Figura 4.6 siendo la 4.6 la señal de entrada y la 4.7 la señal de reloj.

Las señales de salida esperadas cuando se suministra el voltaje inferior son las representadas en la Figura 5.7 y la Figura 5.8 para Q y $\bar{Q}$ respectivamente.

Figura 5.7. Salida Q en voltaje inferior

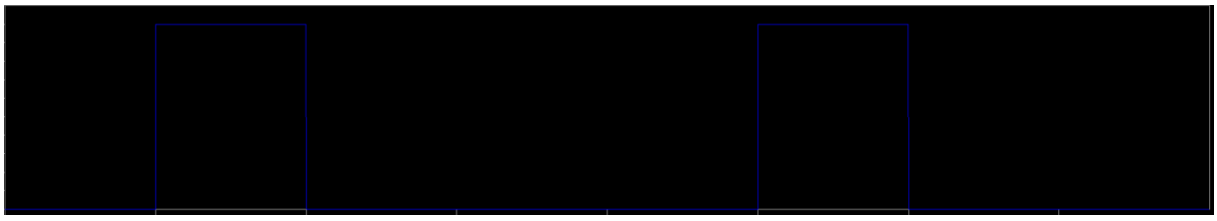


Figura 5.8. Salida $\bar{Q}$ en voltaje inferior

Las señales de salida esperadas cuando se suministra el voltaje inferior son las representadas en la Figura 5.9 y la Figura 5.10 para Q y $\bar{Q}$ respectivamente.

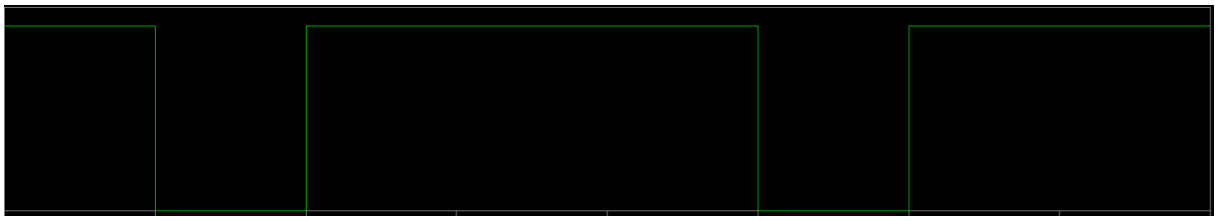


Figura 5.9. Salida Q en voltaje superior

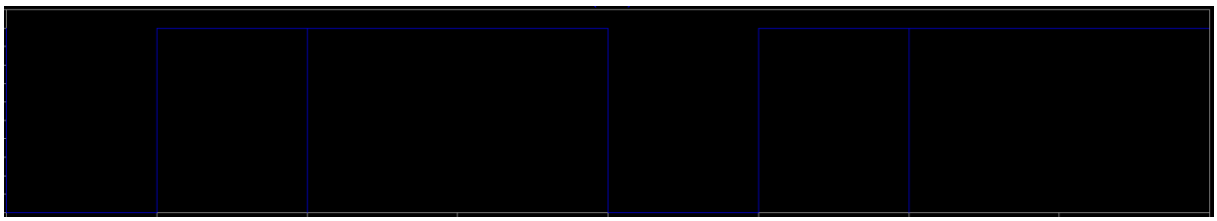


Figura 5.10. Salida $\bar{Q}$ en voltaje superior

Para el cálculo de la función se tendrá en cuenta únicamente la primera mitad de las señales ya que estas son periódicas y la segunda mitad no aporta información extra. La matriz que contiene los valores deseados de la simulación (*outputs*) se divide en los siguientes rangos:

- [0] [0]-[0][3] Contiene la señal Q a alto voltaje
- [1][0]-[1][3] Contiene la señal $\bar{Q}$ a alto voltaje
- [0][4]-[0][7] Contiene la señal Q a bajo voltaje
- [1][4]-[1][7] Contiene la señal $\bar{Q}$ a bajo voltaje

En este caso, debido a que tenemos dos señales, el cálculo de la función objetivo es algo más complejo y se ha dividido en 3 partes. En primer lugar, se evalúan las dos salidas de voltaje superior como viene indicado en la fórmula (3):

$$f_{objetivo1} = (outputs[0][0] - 1) + (0 - outputs[0][1]) + (outputs[0][2] - 1) + (outputs[0][3] - 1) + (0 - outputs[1][0]) + (outputs[1][1] - 1) + (outputs[1][2] - 1) + (outputs[1][3] - 1) \quad (3)$$

Para las señales procedentes de la simulación al voltaje inferior, el cálculo se dividirá en dos partes. Primero, se evaluará la sección inicial en la que el reloj esta activo y por lo tanto sabemos el valor que esperamos de la salida como se aprecia en la formula (4).

$$f_{objetivo2} = (outputs[0][4] - 1) + (0 - outputs[0][5]) + (0 - outputs[1][4]) + (outputs[1][5] - 1) \quad (4)$$

En el caso en el que el reloj esta inactivo, es decir su valor sea 0, en el voltaje inferior. Tanto Q como $\bar{Q}$ deben tener mantener su último valor, por lo tanto, Q debe ser 0 y $\bar{Q}$ debe ser 1. Esto se realiza como viene indicado en la fórmula (5).

$$f_{objetivo3} = (0 - outputs[0][6]) + (0 - outputs[0][7]) + (outputs[1][6] - 1) + (outputs[1][7] - 1) \quad (5)$$

5.3. Ejecución del algoritmo

Una vez adaptado el código para desarrollar el circuito deseado, se tratará de obtener dicho circuito. Al ser este un circuito más complejo que el desarrollado anteriormente, en primer lugar, se plantearán unos parámetros iniciales para tratar de lograr el circuito, sin embargo, es posible que sea necesario variar estos parámetros para lograrlo. Por lo tanto, se irá mostrando los resultados obtenidos con cada grupo de parámetros iniciales.

5.3.1. Primer grupo de parámetros iniciales

Como planteamiento inicial, se han utilizado los mismos parámetros iniciales que en el desarrollo de la puerta lógica polimórfica con la excepción del rango del voltaje utilizado, debido a que este se fijará a ±0.1V del voltaje nominal del modelo de transistor utilizado (1

voltio), además se aumentará el tamaño de la población para tratar de lograr un mejor resultado, debido a que la complejidad del circuito ha aumentado. Como recordatorio los parámetros serían los siguientes.

- Ratio de mutación: Para este valor se ha escogido 0.2, ya que como se mencionó anteriormente otorga un buen equilibrio entre diversidad genética y evitar demasiada aleatoriedad en el proceso.
- Tamaño de la población: Para este tipo de algoritmo se necesita una muestra bastante grande que permita una muestra de población aceptable, se ha escogido un tamaño de 300 individuos.
- Valor de la función objetivo a partir del cual se considerará suficientemente bueno el circuito: Para decidir cuándo terminar el algoritmo, se ha marcado este valor como -1, ya que los circuitos con una evaluación superior a esta presentan un comportamiento muy cercano al deseado. Se ha aumentado el valor debido a que el número de valores que se tienen en cuenta para la función objetivo se duplica y por lo tanto, un pequeño error en todos se reflejara en mayor medida en el valor obtenido de la función objetivo.
- Anchura de los transistores: Para el valor de estos se ha determinado un rango entre 100 nanómetros y 5 micrómetros.
- Modelo de transistor: Se ha seleccionado el transistor PTM 45nm BSIM4.

5.3.1.1. Proceso de ejecución

Una vez se han realizado los cambios pertinentes, se utilizará el algoritmo partiendo del circuito desarrollado para la sección anterior. Se mostrará la evolución de los circuitos a través de las generaciones representando las simulaciones de los padres de cada generación. Además, en estas figuras, la primera mitad (previo a 0.08 segundos), se muestra la simulación cuando a las puertas polimórficas se les suministra el voltaje superior (1.1V), y la segunda mitad (a partir de los 0.08 segundos), se muestra la simulación con el voltaje inferior (0.9V). A continuación, en la Figura 5.11 y la Figura 5.12 se pueden observar las simulaciones de los padres provenientes de la población inicial.

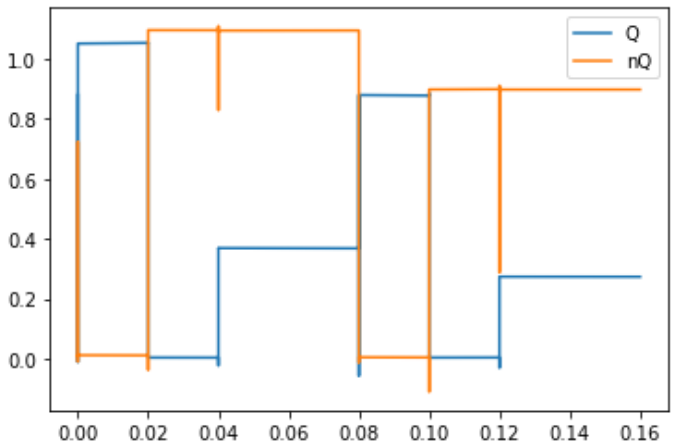


Figura 5.11. Primera generación, padre 1

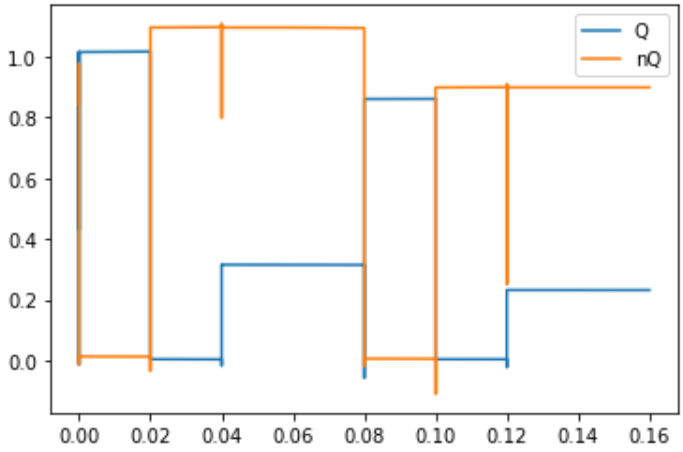


Figura 5.12. Primera generación, padre 2

Al no cumplir con el objetivo marcado, se repite el algoritmo.

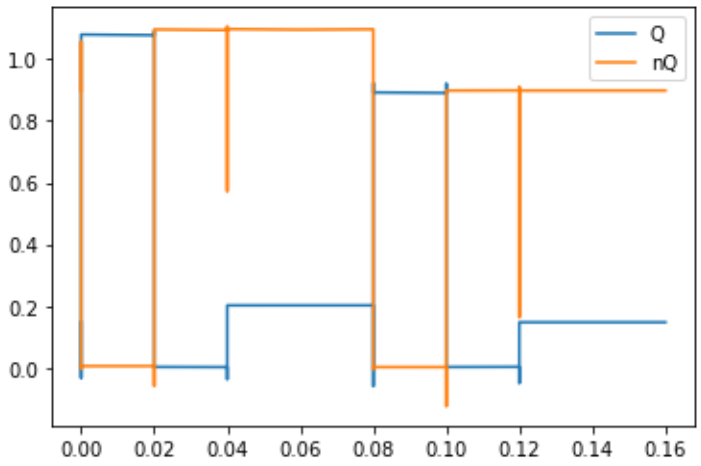


Figura 5.13. Tercera generación, padre 1

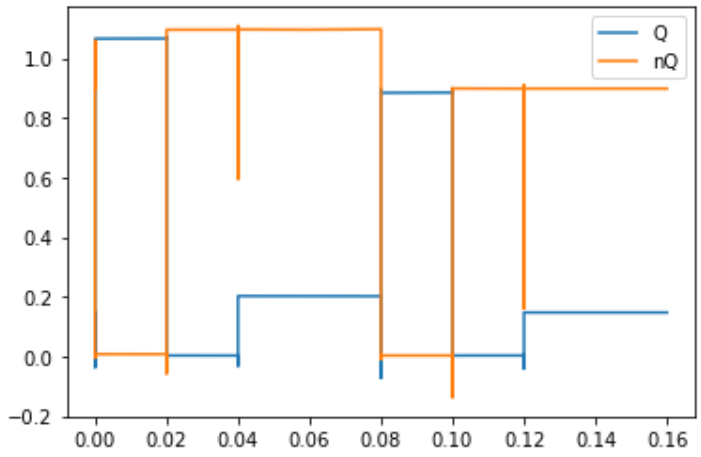


Figura 5.14. Tercera generación, padre 2

De nuevo se sigue sin cumplir el objetivo del circuito por lo que algoritmo continuará.

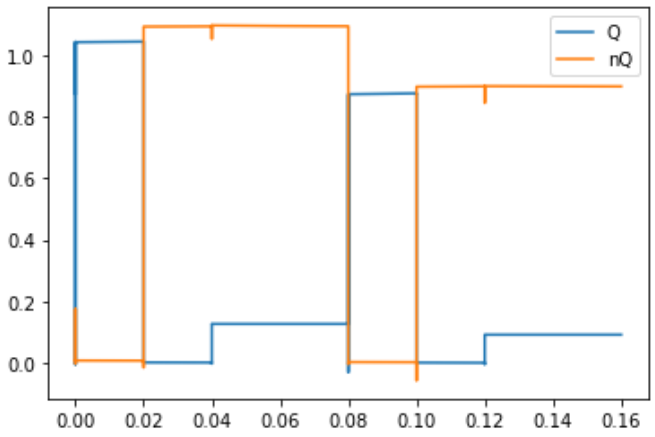


Figura 5.15. Sexta generación, padre 1

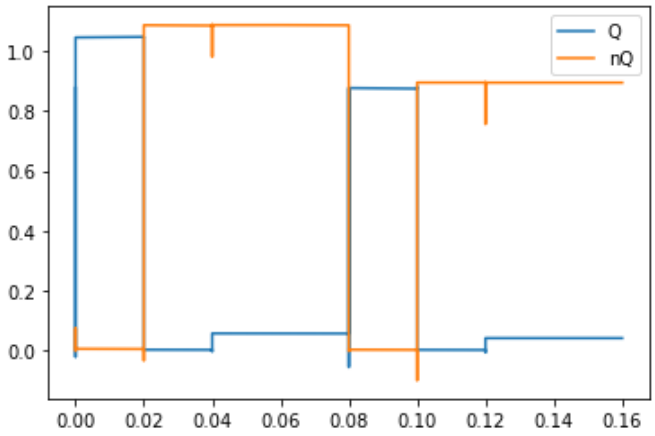


Figura 5.16. Sexta generación, padre 2

De nuevo se sigue sin cumplir con el objetivo por lo que se itera de nuevo.

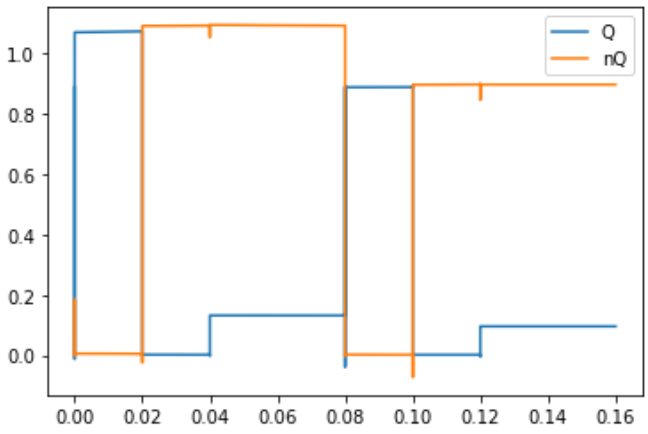


Figura 5.17. Novena generación, padre 1

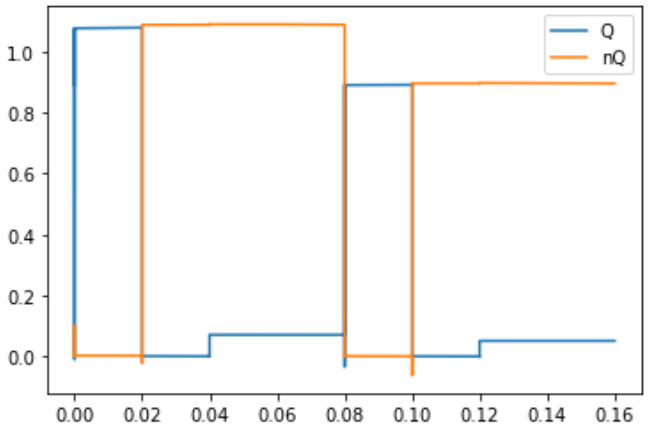


Figura 5.18. Novena generación, padre 2

Tras nueve generaciones, el algoritmo se ha parado manualmente, debido a la baja variedad de resultados obtenidos y a la escasa mejora en las simulaciones.

5.3.1.2. Evolución del valor objetivo durante las generaciones

A lo largo de las diversas generaciones presentadas, la evaluación de estas se ve representada en la Figura 5.19.

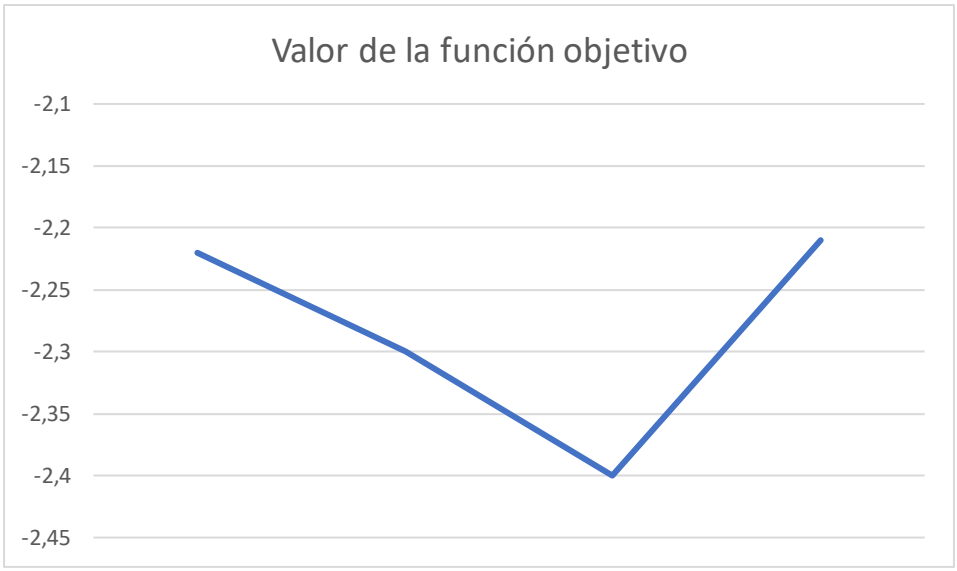


Figura 5.19. Evolución del valor de la función objetivo

5.3.1.3. Análisis de la evolución del circuito

Tras 9 generaciones de circuitos lo que representa un total de 2700 circuitos simulados, las diferencias apreciadas tantos en los gráficos de las simulaciones como en el valor de la función objetivo de las mismas, son muy escasas por lo tanto se planteará modificar los parámetros iniciales para aumentar la variación de los circuitos.

5.3.2. Segundo grupo de parámetros iniciales

En este caso nos basaremos en los últimos parámetros iniciales, sin embargo, para aumentar la variación en los circuitos se modificarán dos parámetros. En primer lugar, se ampliará el rango en el varía la anchura de los transistores. Además, se evolucionará la longitud de los transistores además de la anchura. Mediante estos dos cambios, se busca que aumenten las posibilidades y por lo tanto la variación entre los circuitos. Los parámetros iniciales quedan de la siguiente forma.

- Ratio de mutación: Para este valor se ha escogido 0.2.
- Tamaño de la población: 300 individuos.
- Valor de la función objetivo a partir del cual se considerará suficientemente bueno el circuito: -1.
- Anchura de los transistores: Para el valor de la anchura se ha determinado un rango entre 100 nanómetros y 50 micrómetros.
- Longitud de los transistores: Para el valor de este dato se ha elegido el mismo que para la anchura.
- Modelo de transistor: Se ha seleccionado el transistor PTM 45nm BSIM4.

5.3.2.1. Modificaciones en el algoritmo

Debido a la incorporación de la variable longitud de los transistores en el algoritmo se deben realizar ciertos cambios en el algoritmo desarrollado para que se puede ejecutar correctamente. Las funciones que se modificarán son aquellas que trabajan con los genes, ya que estamos añadiendo uno nuevo. Entre estas están, *population_generator(), crossover() y*

netlist_modifier(). Hay otras funciones que también requieren cambios, pero estos son mínimos.

5.3.2.1.1 Generación de la población

Como se puede ver en la Figura 5.20 las modificaciones de esta función son necesarias para incluir la longitud de los transistores en el vector de la población.

```
def population_generator():
    lower_value, upper_value, step_size = 100, 50000, 10
    n = 8 ## Number of transistor widths needed in the netlist
    population=[[0 for c in range(18)] for b in range(300)]
    for i in range(300):
        widths = [generate_parent(lower_value,upper_value,step_size) for i in range(n)]
        lengths = [generate_parent(lower_value,upper_value,step_size) for i in range(n)]
        vh = 1.1
        vl= 0.9
        chromosome=[lengths,widths,vh,vl]
        population[i]=chromosome

    return population
```

Figura 5.20. Modificaciones a la función generadora de una población

5.3.2.1.2 Mezcla de genes

En este caso, se debe añadir la longitud cuando se mezclen los genes de los padres, además, se mezclarán los transistores completos de forma que al elegir de que padre se elegirá cada gen, se escogerán la anchura y longitud del mismo padre tal y como se muestra en la Figura 5.21.

```
def crossover(parent1, parent2,bestFitness):
    t1=np.zeros(8)
    t2=np.zeros(8)
    new_population=[[0 for c in range(10)] for b in range(300)]

    for i in range(300):
        child=[[0 for c in range(8)] for b in range(4)]
        used1=np.empty(8,int)
        used2=np.empty(8,int)

        lengths=np.zeros(8)
        widths=np.zeros(8)
        ctrl=0
        while(ctrl<7):
            p1=random.randrange(0, 8, 1)
            if p1 in used1:
                while p1 in used1:
                    p1=random.randrange(0, 8, 1)

            p2=random.randrange(10, 18, 1) - 10
            if p2 in used2:
                while p2 in used2:
                    p2=random.randrange(10, 18, 1)-10


            used1[ctrl]=p1
            used2[ctrl]=p2
            l1=parent1[0][p1]
            lengths[ctrl]=l1
            w1=parent1[1][p1]
            widths[ctrl]=w1
            ctrl+=1
            l2=parent1[0][p2]
            lengths[ctrl]=l2
            w2=parent1[1][p2]
            widths[ctrl]=w2
            ctrl+=1

        child[0]=lengths
        child[1]=widths

        child[2]=parent1[2]
        child[3]=parent1[3]
```

Figura 5.21. Modificaciones a la mezcla de genes

Esta función también incluye la mutación de los genes y se tendrá que añadir la longitud para que exista la posibilidad de que esta mute, esto se realiza tal y como se ve en la Figura 5.22.

```
    if random.uniform(0, 10)< (MUTATION_RATE*10):
        for p in range(8):
            if random.uniform(0, 1)< (MUTATION_RATE+0.1):
                ranil=rangoi(child[0][p],bestFitness)
                lowl=(child[0][p]-ranil)*1e9

                ransl=rangos(child[0][p],bestFitness)
                upl=(child[0][p]+ransl)*1e9

                raniw=rangoi(child[1][p],bestFitness)
                loww=(child[1][p]-raniw)*1e9

                ransw=rangos(child[0][p],bestFitness)
                upw=(child[1][p]+ransw)*1e9

                child[0][p]=mutate(child[0][p], lowl, upl, 10)
                child[1][p]=mutate(child[1][p], loww, upw, 10)

    new_population[i]=child
return new_population
```

Figura 5.22. Modificaciones a la mutación

5.3.2.1.3 Modificación de netlists

En esta función se debe añadir las longitudes de los transistores para que se tengan en cuenta en la simulación. Esto se realiza como se puede observar en la Figura 5.23.

```
LTC.set_element_model('M1', 'Vdd PMOS L='+str(modified_lengths[0])+' W='+str(modified_widths[0]))
LTC.set_element_model('M2', 'Vdd PMOS L='+str(modified_lengths[1])+' W='+str(modified_widths[1]))
LTC.set_element_model('M3', 'Vdd PMOS L='+str(modified_lengths[2])+' W='+str(modified_widths[2]))
LTC.set_element_model('M4', 'Vdd PMOS L='+str(modified_lengths[3])+' W='+str(modified_widths[3]))
LTC.set_element_model('M5', 'Vdd PMOS L='+str(modified_lengths[4])+' W='+str(modified_widths[4]))
LTC.set_element_model('M6', '0 NMOS L='+str(modified_lengths[5])+' W='+str(modified_widths[5]))
LTC.set_element_model('M7', '0 NMOS L='+str(modified_lengths[6])+' W='+str(modified_widths[6]))
LTC.set_element_model('M8', '0 NMOS L='+str(modified_lengths[7])+' W='+str(modified_widths[7]))
LTC.set_element_model('M9', 'N025 PMOS L='+str(modified_lengths[0])+' W='+str(modified_widths[0]))
LTC.set_element_model('M10', 'N025 PMOS L='+str(modified_lengths[1])+' W='+str(modified_widths[1]))
LTC.set_element_model('M11', 'N025 PMOS L='+str(modified_lengths[1])+' W='+str(modified_widths[2]))
LTC.set_element_model('M12', 'N025 PMOS L='+str(modified_lengths[3])+' W='+str(modified_widths[3]))
LTC.set_element_model('M13', 'N025 PMOS L='+str(modified_lengths[4])+' W='+str(modified_widths[4]))
LTC.set_element_model('M14', '0 NMOS L='+str(modified_lengths[5])+' W='+str(modified_widths[5]))
LTC.set_element_model('M15', '0 NMOS L='+str(modified_lengths[6])+' W='+str(modified_widths[6]))
LTC.set_element_model('M16', '0 NMOS L='+str(modified_lengths[7])+' W='+str(modified_widths[7]))
LTC.set_component_value('V1', 'PULSE(0 '+str((vh+vl)/2)+' 0 5n 5n 20m 40m)')
LTC.set_component_value('V2', 'PULSE(0 '+str((vh+vl)/2)+' 0 5n 5n 40m 80m)')
LTC.set_component_value('V4', 'PULSE('+str(vl)+' '+str(vh)+' 0 5n 5n 80m 160m)')
LTC.set_component_value('V5', 'PULSE('+str(vl)+' '+str(vh)+' 0 5n 5n 80m 160m)')
LTC.set_component_value('V3', str((vh+vl)/2))
LTC.set_component_value('V12', str((vh+vl)/2))
LTC.set_component_value('V6', str((vh+vl)/2))
```

Figura 5.23. Modificaciones a la función modificadora de netlists

5.3.2.2. Proceso de ejecución

Una vez, se han realizado los cambios pertinentes para añadir la longitud de los transistores al algoritmo, se procederá a ejecutarlo para tratar de obtener el circuito objetivo. Al igual que la ejecución anterior, se mostrará la evolución a través de las generaciones. Los gráficos seguirán el mismo formato que ha sido utilizado en el apartado 5.3.1.1. En primer lugar, se muestran los primeros padres en la Figura 5.24 y la Figura 5.25.

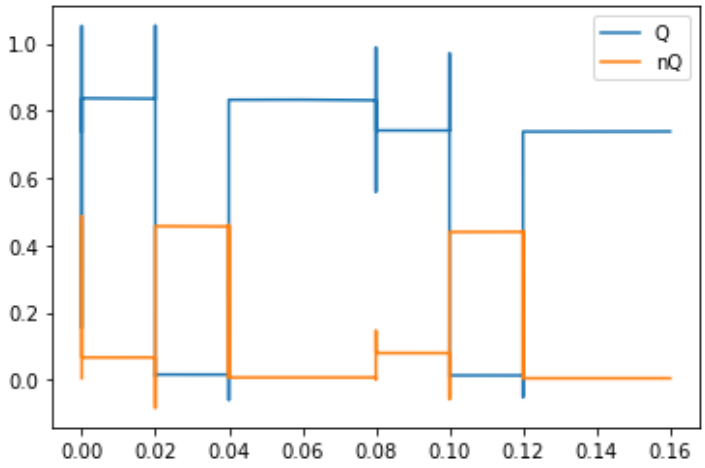


Figura 5.24. Primera generación, padre 1

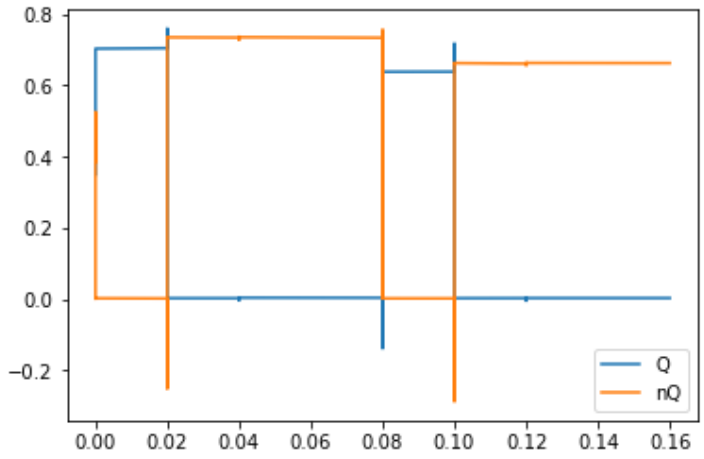


Figura 5.25. Primera generación, padre 2

Al no cumplir con el objetivo el algoritmo continúa.

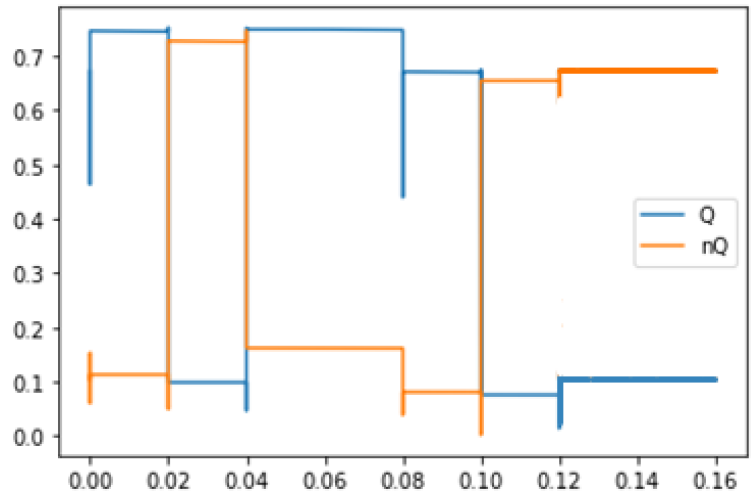


Figura 5.26. Segunda generación, padre 1

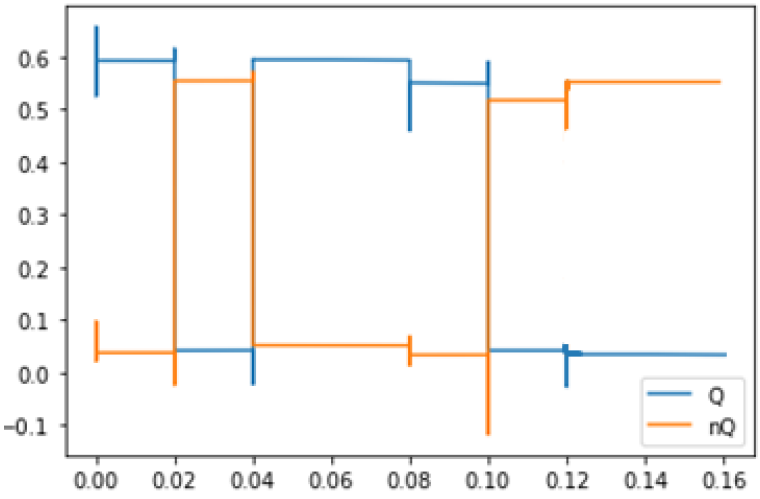


Figura 5.27. Segunda generación, padre 2

De nuevo no cumplen con el objetivo, por lo que se utilizarán para crear la siguiente población.

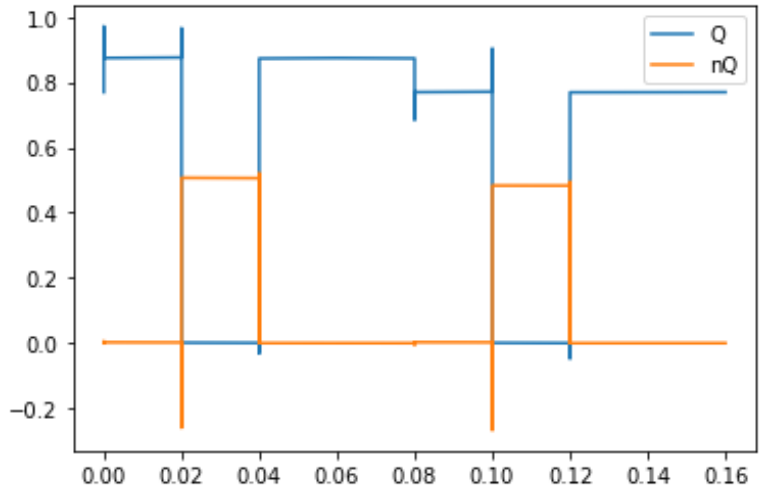


Figura 5.28. Tercera generación, padre 1

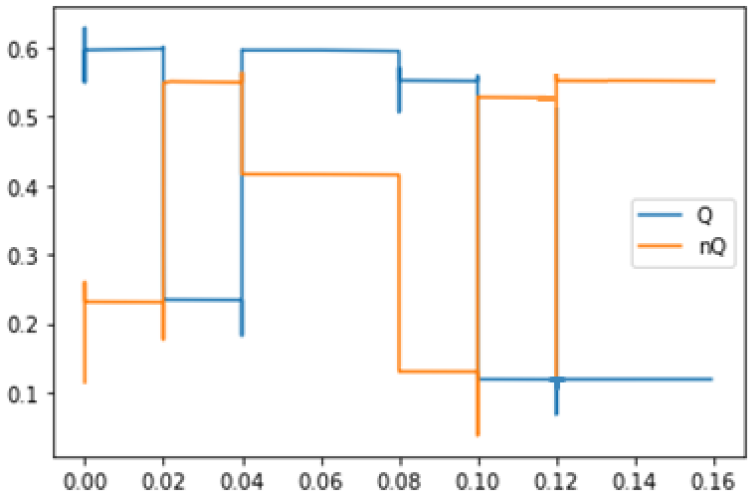


Figura 5.29. Tercera generación, padre 2

Podemos observar que se empieza a acercar al objetivo del algoritmo, sin embargo, aún no cumple con lo esperado.

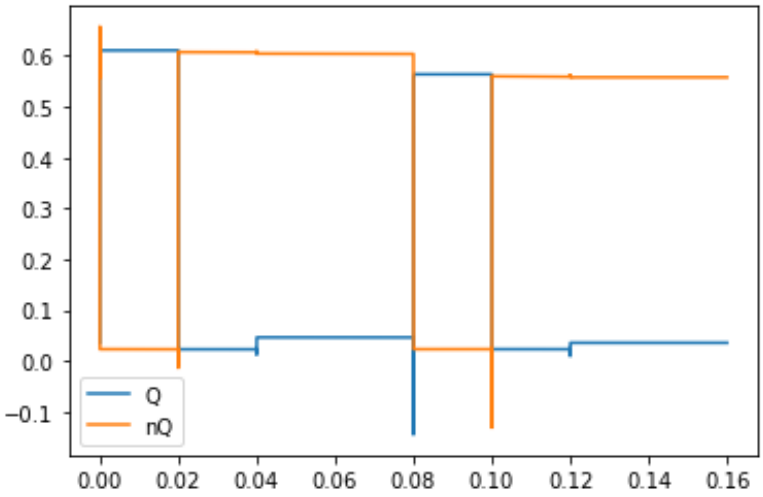


Figura 5.30. Cuarta generación, padre 1

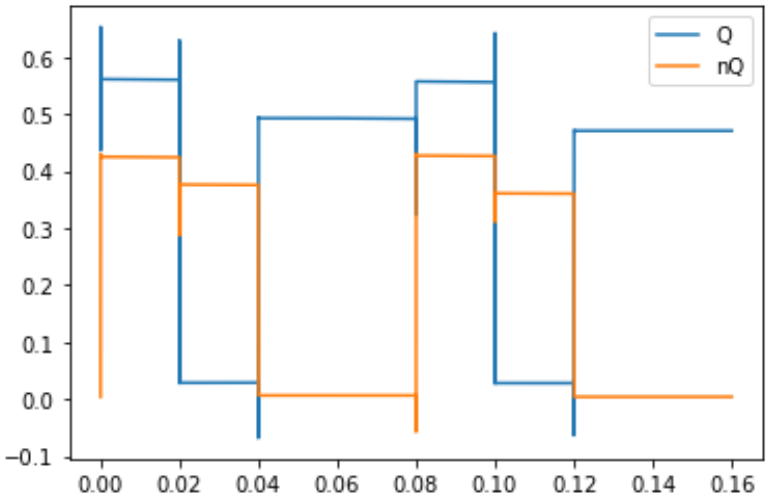


Figura 5.31. Cuarta generación, padre 2

El algoritmo realiza otra iteración al no obtener el resultado deseado.

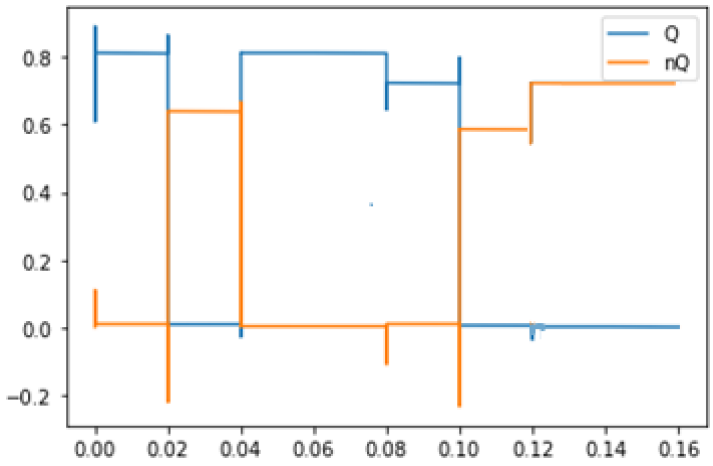


Figura 5.32. Quinta generación, padre 1

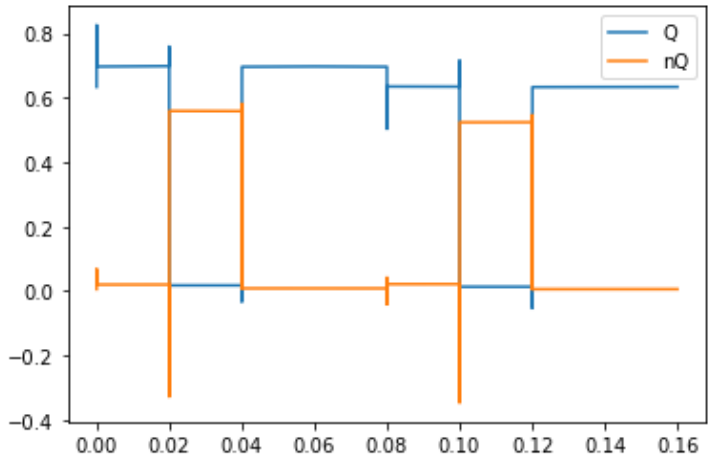


Figura 5.33. Quinta generación, padre 2

La evolución sigue sin ser suficiente por lo que será necesario continuar sometiéndolo al algoritmo genético.

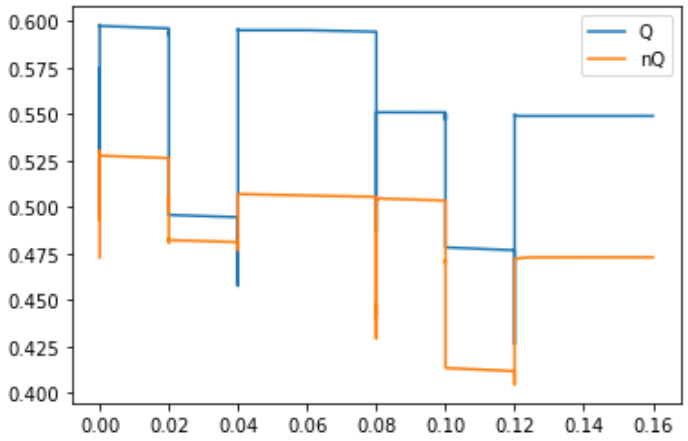


Figura 5.34. Sexta generación, padre 1

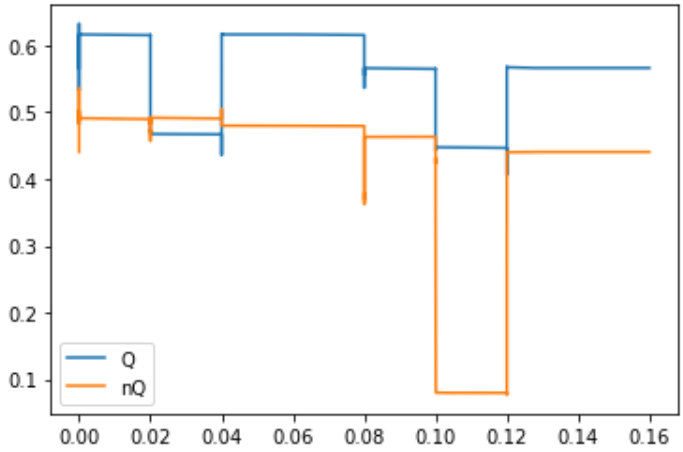


Figura 5.35. Séptima generación, padre 2

En este caso, los nuevos padres muestran un claro peor rendimiento que en la anterior generación por lo que el programa debe continuar, además más adelante se deberá revisar el método mediante el cual se escogen los padres.

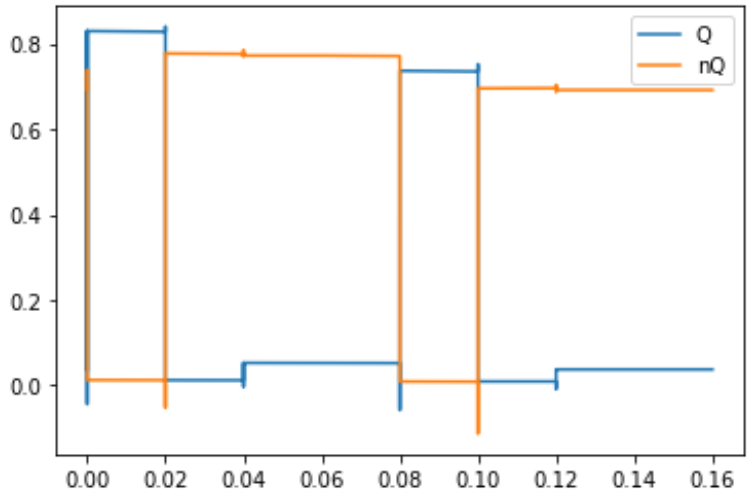


Figura 5.36. Séptima generación, padre 1

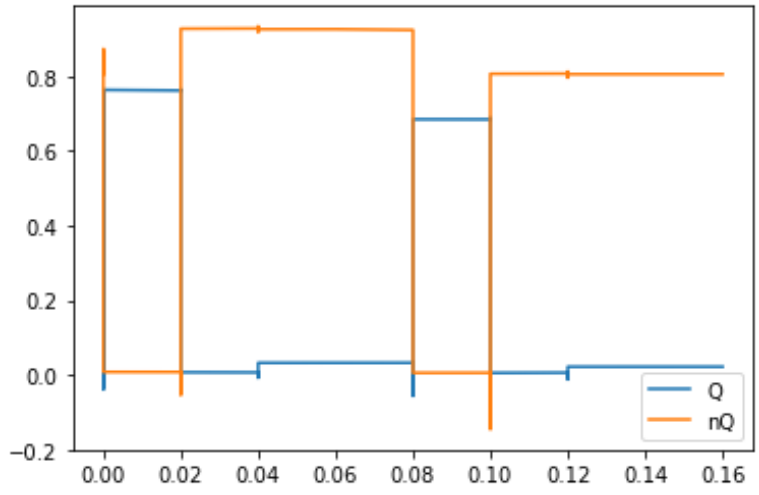


Figura 5.37. Séptima generación, padre 2

Los padres siguen sin cumplir con el objetivo establecido.

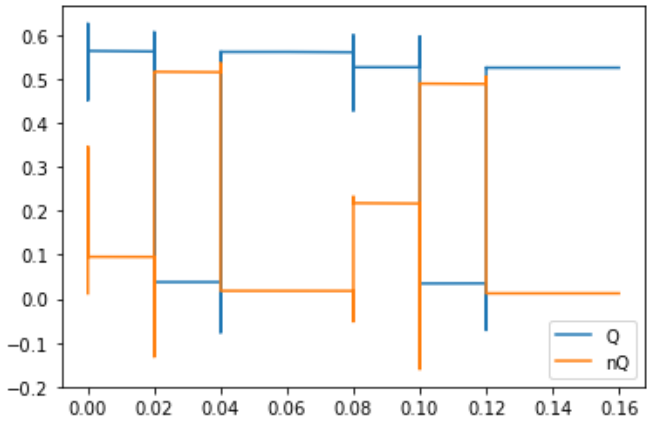


Figura 5.38. Octava generación, padre 1

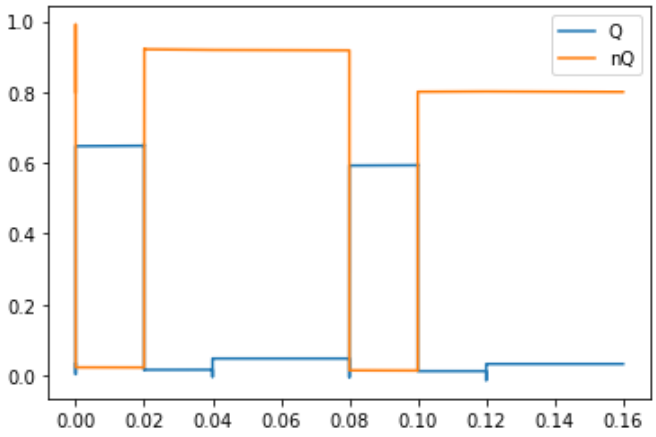


Figura 5.39. Octava generación, padre 2

De nuevo, será necesario continuar con el algoritmo.

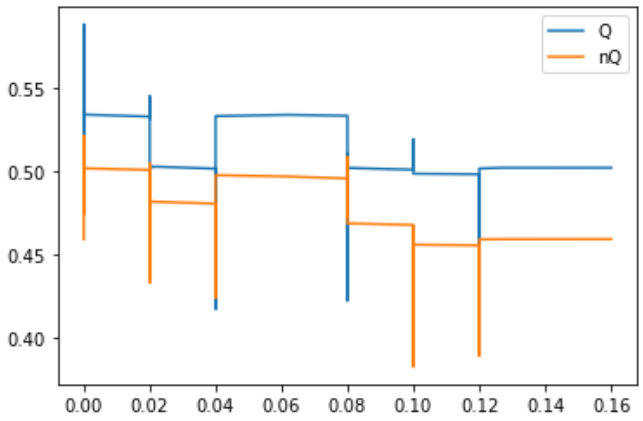


Figura 5.40. Novena generación, padre 1

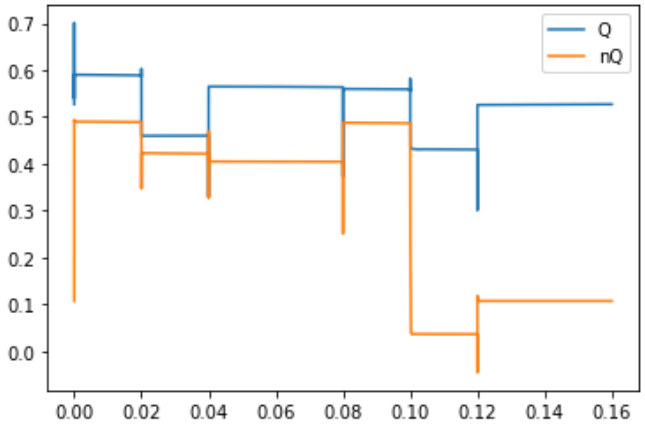


Figura 5.41. Novena generación, padre 2

El resultado ha empeorado claramente respecto a la generación anterior, como esto ha ocurrido anteriormente, se va a tratar de averiguar la fuente del problema y solucionarlo.

5.3.2.3. Evolución del valor objetivo durante las generaciones

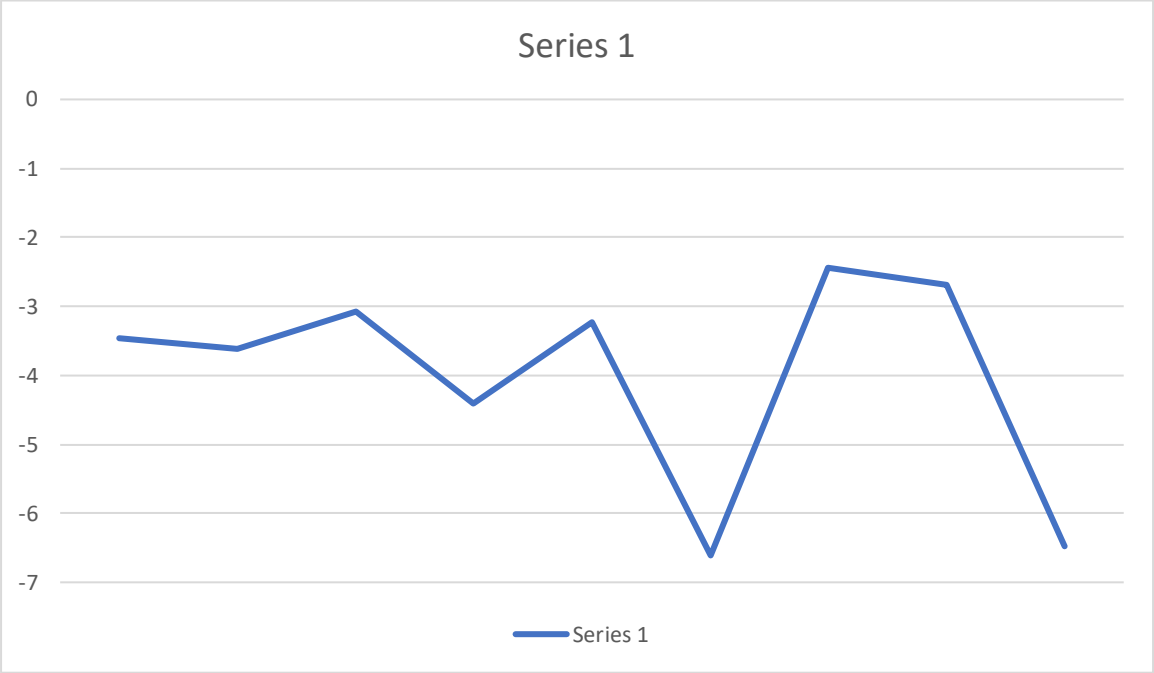


Figura 5.42. Evolución del valor objetivo en las generaciones

5.3.2.4. Análisis de la evolución del circuito

Como se ha mencionado anteriormente, durante la evolución del circuito hay ciertas generaciones en las que se empeora en gran medida el comportamiento del circuito. Por lo tanto, se ha investigado el motivo de esto y este se encuentra en la selección de padres de cada generación, el problema de esta es el gran número de individuos en cada población y la relativamente escasa diferencia en el valor de la función objetivo de un buen individuo y uno malo. Por ejemplo, entre un individuo con una evaluación de -2 y otro con una de -5, al tener 300 individuos, la diferencia entre las probabilidades de escoger uno u otro es muy bajo y por lo tanto en muchas ocasiones se escogerá un individuo claramente peor que otros posibles. Por esto se plantearán cambios a la función de selección de padres para optimizar esto y que no ocurra con tanta probabilidad.

5.3.3. Terceros parámetros iniciales

Tras observar que no se han logrado los resultados deseados en los experimentos anteriores, se realizará un cambio en la función que se encarga de seleccionar los padres. Esto se debe a que, debido al gran volumen de la población y el parecido del valor de la función objetivo de los individuos, era poco probable que se escogiesen los mejores padres en cada generación. Por esto se ha decidido probar a cambiar la elección de padres a la selección por torneo que se explicó en la introducción de forma que aumente la probabilidad con la que se escogen los mejores individuos de cada población. De forma que esta función ha quedado como se ve en la Figura 5.43.

```
def tournament(fitness_pop):
    parent=[[0 for c in range(2)] for b in range(2)]
    vec=[]
    tournament_pop=[]

    #Get a part of the population for the tournament
    tournament_pop=np.random.choice(fitness_pop,200)
    vec=tournament_pop

    #Calculate the maximum of the fitness values and its population index
    max1=max(vec)
    indexx=np.where(fitness_pop==max1)
    index=int(indexx[0][0])

    #Deleting the first father from the vector so we don't chose it again
    selectionIndex=np.where(tournament_pop==max1)
    vec=np.delete(vec, selectionIndex[0][0])

    #Second father selection
    max2=max(vec)
    indexs2=np.where(tournament_pop==max2)
    index2=int(indexs2[0][0])
    parent[0]=[index,max1]
    parent[1]=[index2,max2]

    return parent
```

Figura 5.43. Función selección por torneo

El resto de los parámetros iniciales quedarían de la siguiente forma:

- Ratio de mutación: 0.2.
- Tamaño de la población: 300 individuos.
- Valor de la función objetivo a partir del cual se considerará suficientemente bueno el circuito: -1

- Anchura de los transistores: Para el valor de estos se ha determinado un rango entre 100 nanómetros y 50 micrómetros.
- Longitud de los transistores: Mismo rango que la anchura.
- Modelo de transistor: Se ha seleccionado el transistor PTM 45nm BSIM4.

5.3.3.1. Proceso de ejecución

De nuevo, se mostrará el proceso de ejecución con los cambios planteados para ver la evolución que presenta y tratar de lograr el circuito deseado. Una vez más el formato será el mismo. Los padres resultantes de la población inicial son los que aportan el resultado en la y la Figura 5.44 y la Figura 5.45.

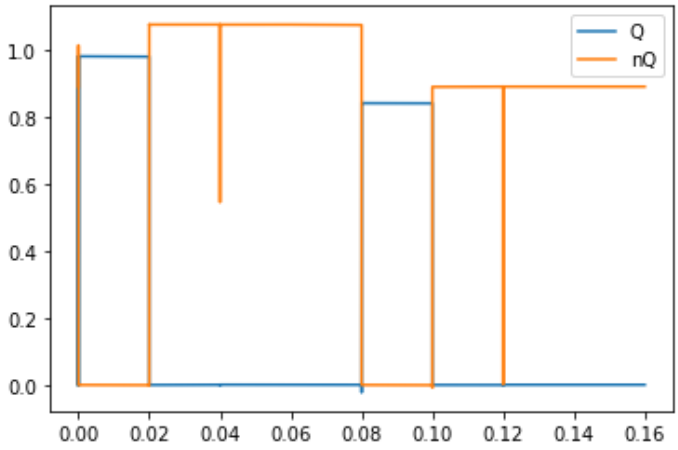


Figura 5.44. Primera generación, padre 1

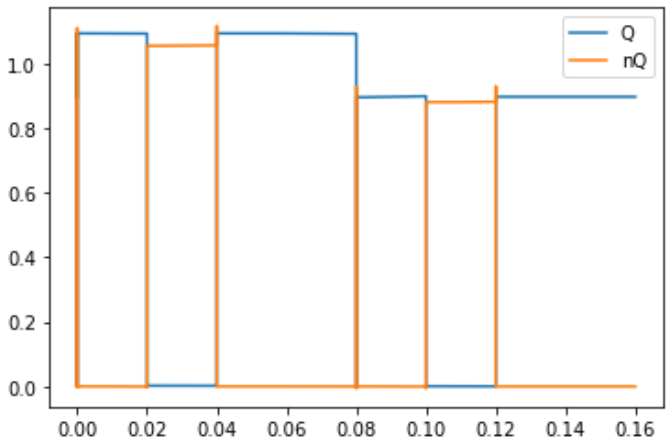


Figura 5.45. Primera generación, padre 2

Es una buena primera iteración, sin embargo, no presenta las características del circuito polimórfico que se buscan.

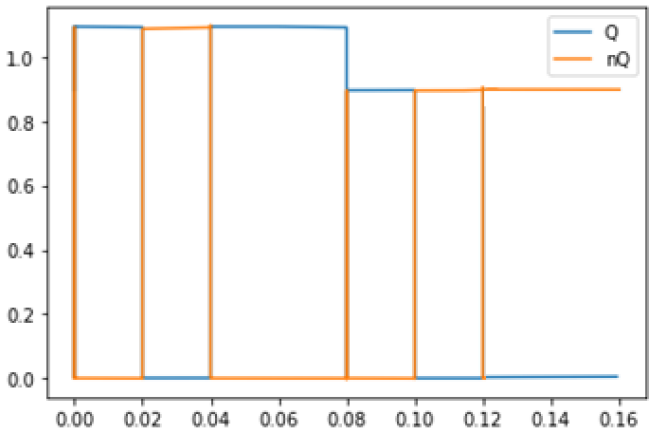


Figura 5.46. Segunda generación, padre 1

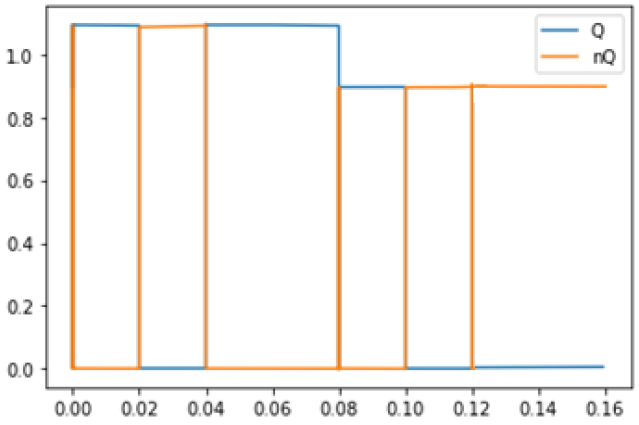


Figura 5.47. Segunda generación, padre 2

Estos padres, presentan un resultado parecido al anterior y por lo tanto, el algoritmo debe continuar.

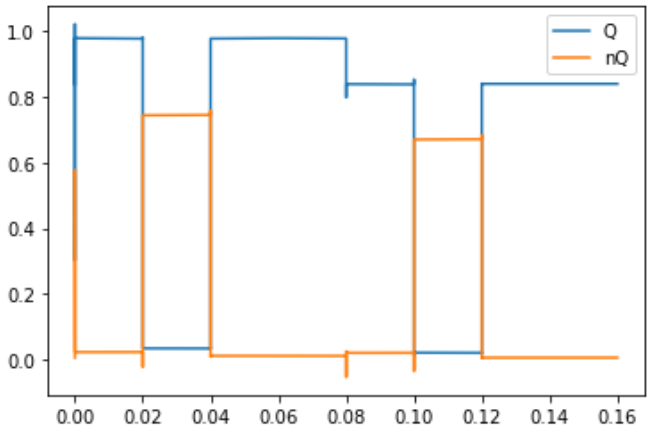


Figura 5.48. Tercera generación, padre 1

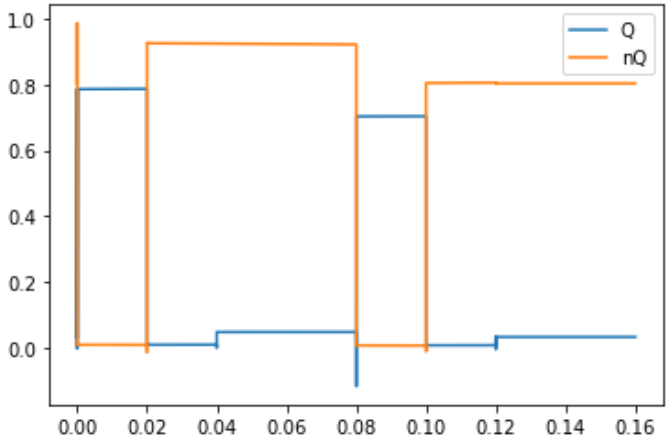


Figura 5.49. Tercera generación, padre 2

El segundo padre empieza a presentar ligeros atisbos del circuito polimórfico a desarrollar, pero la evolución debe continuar.

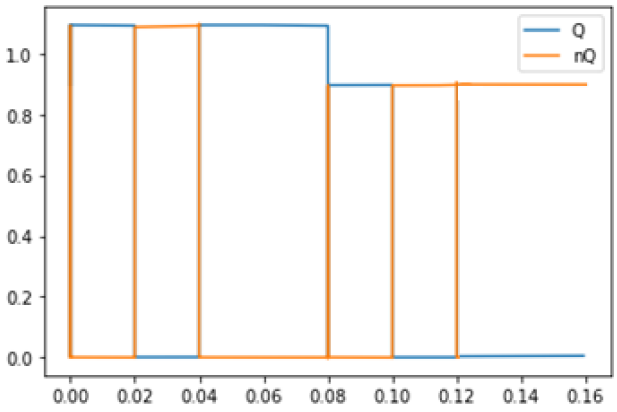


Figura 5.50. Cuarta generación, padre 1

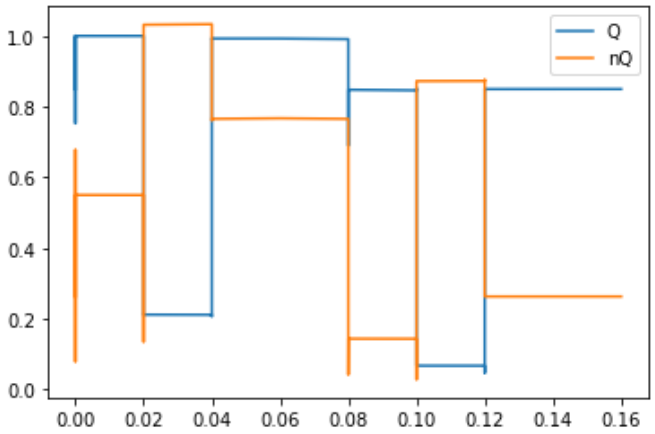


Figura 5.51. Cuarta generación, padre 2

El segundo padre, presenta un comportamiento bastante parecido al deseado, sin embargo, no es suficiente para que el circuito funcione como se busca.

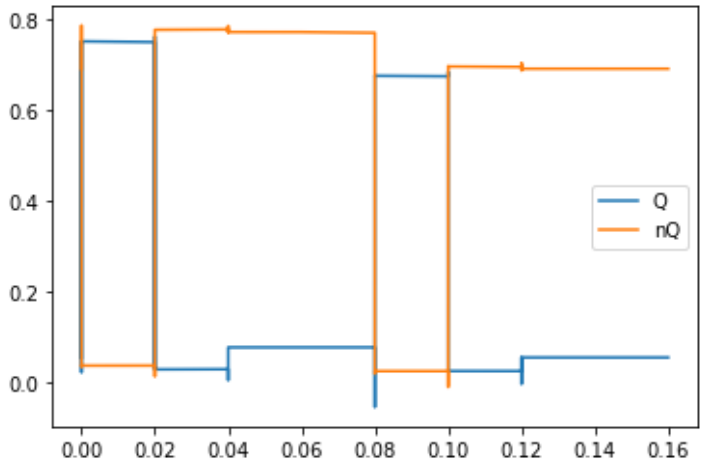


Figura 5.52. Quinta generación, padre 1

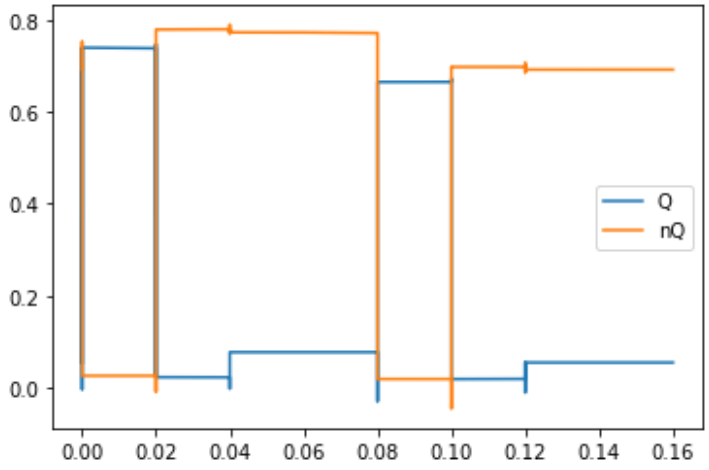


Figura 5.53. Quinta generación, padre 2

Volvemos a observar un circuito que presenta un escaso comportamiento polimórfico, por lo cual el algoritmo continuará.

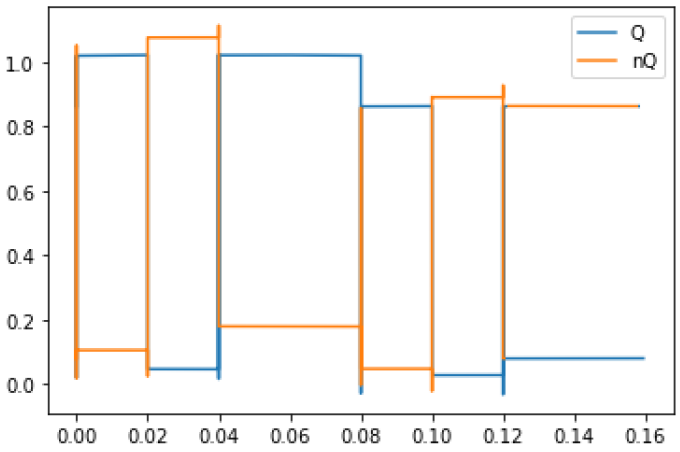


Figura 5.54. Sexta generación, padre 1

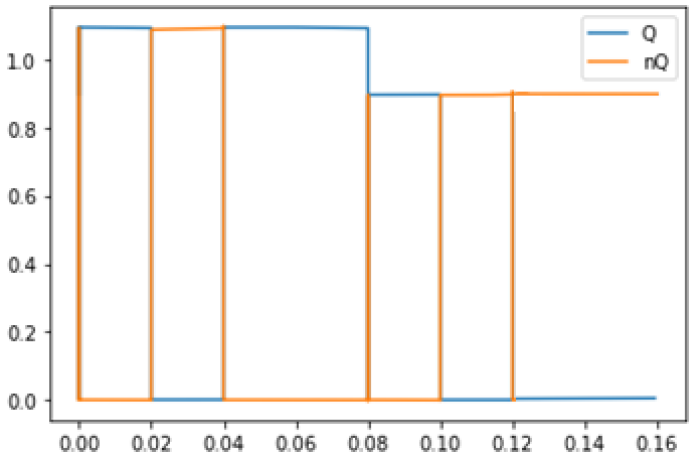


Figura 5.55. Sexta generación, padre 2

En este caso, el padre 1 presenta algo del comportamiento que se busca, pero no el suficiente.

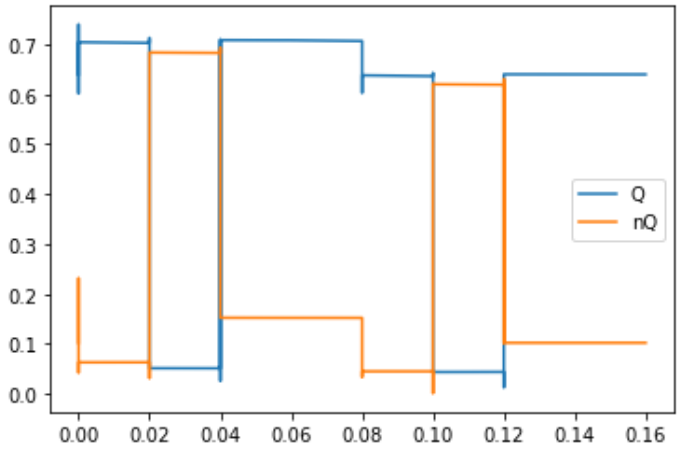


Figura 5.56. Séptima generación, padre 1

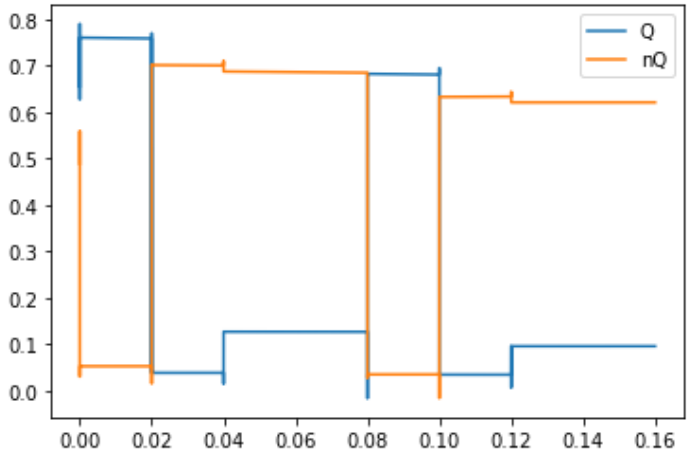


Figura 5.57. Séptima generación, padre 2

De nuevo es perceptible el comportamiento que se busca, pero de manera insuficiente.

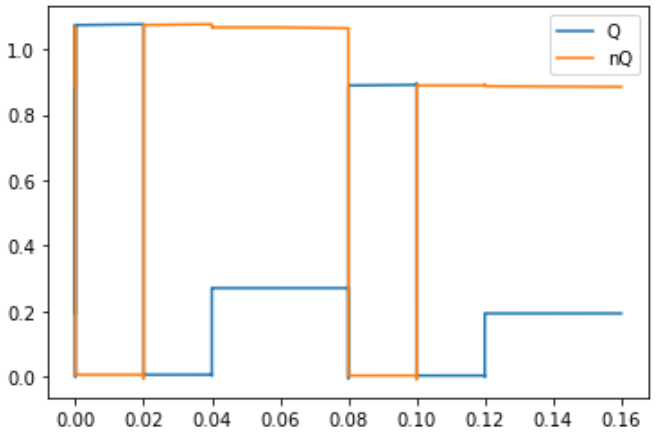


Figura 5.58. Octava generación, padre 1

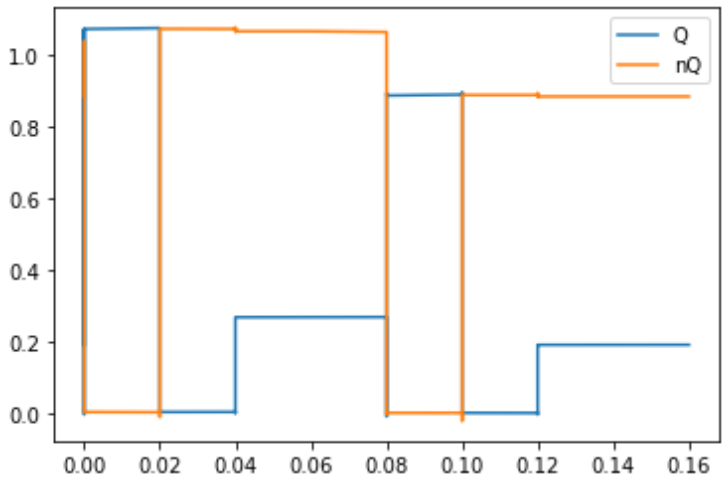


Figura 5.59. Octava generación, padre 2

Sigue sin cumplir con los objetivos del circuito.

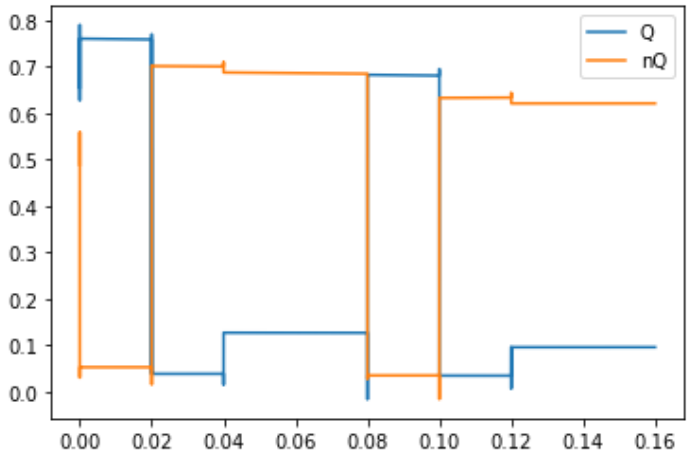


Figura 5.60. Novena generación, padre 1

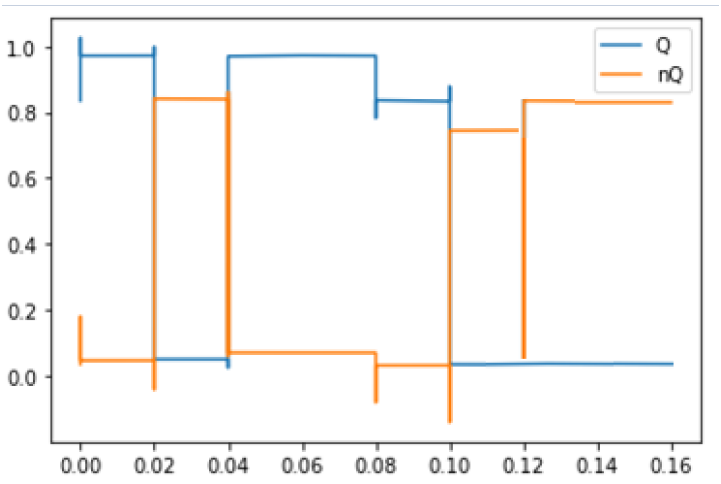
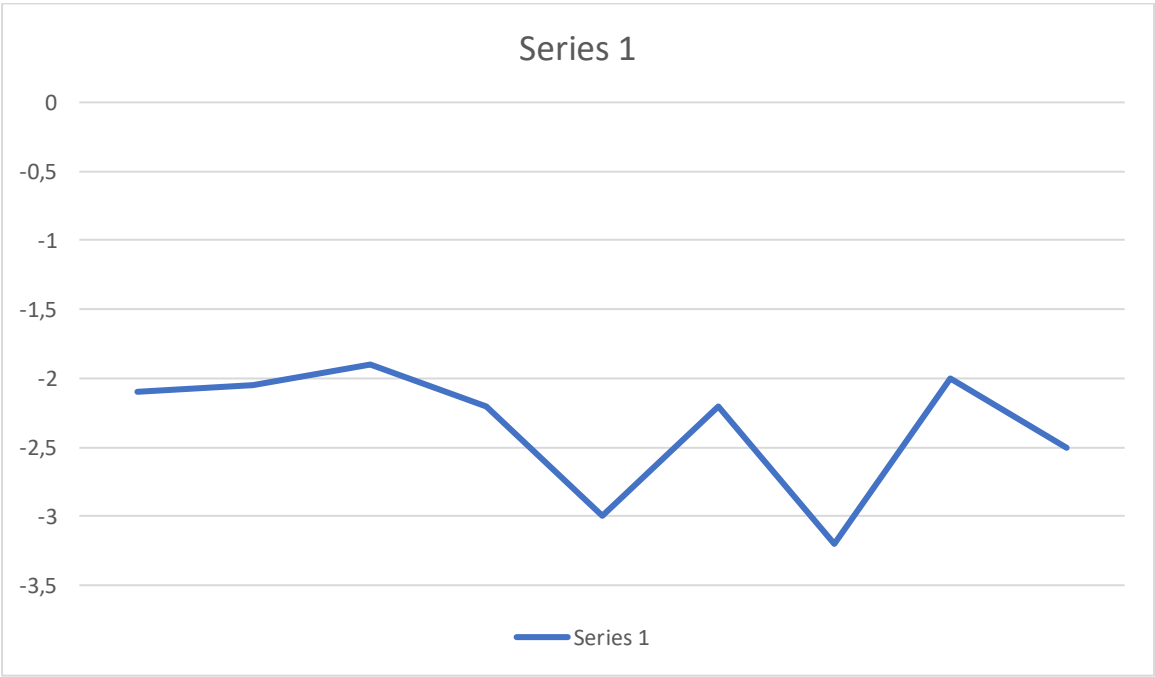


Figura 5.61. Novena generación, padre 2

Este último de nuevo no presenta el comportamiento buscado.

5.3.3.2. Evolución del valor objetivo durante las generaciones



5.3.3.3. Análisis de la evolución del circuito

En esta ejecución, se han podido observar algunas simulaciones que se acercaban bastante al circuito que se quiere desarrollar, sin embargo, el tiempo que requeriría para lograr una buena solución sería muy elevado ya que cada simulación requiere un breve periodo de tiempo, pero se realizan 2 simulaciones por circuito y hay 300 circuitos por población. Por lo tanto, se tratará de realizar algunos cambios para llegar a una buena solución de manera más rápida.

5.3.4. Cuartos parámetros iniciales

Para aumentar la velocidad con la que se encuentra el circuito objetivo, se va a realizar un cambio en el cálculo del valor de la función objetivo. Lo que se realizará, es darles más importancia a las partes de la simulación donde el reloj tenga 0 como valor, ya que era en estos puntos donde las simulaciones estaban más lejos del comportamiento deseado. Para hacer esto se multiplicará la evaluación en estas zonas por 1.5 para que una desviación en estas zonas tenga más impacto que anteriormente. Además, esto reflejará un cambio en el valor objetivo mínimo para que el circuito sea válido.

- Ratio de mutación: 0.2.
- Tamaño de la población: 300 circuitos.
- Valor de la función objetivo a partir del cual se considerará suficientemente bueno el circuito: Se establece en -1.5 debido al cambio en realizado en el cálculo de la función objetivo.
- Anchura de los transistores: Para el valor de estos se ha determinado un rango entre 100 nanómetros y 50 micrómetros.
- Longitud de los transistores: Mismos valores que la anchura de los transistores.
- Modelo de transistor: Se ha seleccionado el transistor PTM 45nm BSIM4.

5.3.4.1. Modificación en el cálculo de la función objetivo

Se mostrarán los cambios en la evaluación de los circuitos para facilitar su comprensión. Se mostrarán las fórmulas correspondientes al voltaje superior (1.1V) y al voltaje inferior en caso de que el reloj este en 0 voltios ya que en el caso del voltaje inferior con el reloj a 1V, la

fórmula no cambia. En la fórmula (6) podemos observar la evaluación para el voltaje superior.

$$f_{objetivo1} = (outputs[0][0] - 1) + (0 - outputs[0][1]) + (outputs[0][2] - 1) * 1.5 + (outputs[0][3] - 1) * 1.5 + (0 - outputs[1][0]) + (outputs[1][1] - 1) + (outputs[1][2] - 1) * 1.5 + (outputs[1][3] - 1) * 1.5 \quad (6)$$

Para el voltaje inferior cuando el reloj está a 0 voltios, la evaluación se realizará mediante la fórmula (7).

$$f_{objetivo3} = \big((0 - outputs[2][2]) + (0 - outputs[2][3]) + (outputs[3][2] - 1) + (outputs[3][3] - 1), (outputs[2][2] - 1) + (outputs[2][3] - 1) + (0 - outputs[3][2]) + (0 - outputs[3][3])\big) * 1.5 \quad (7)$$

5.3.4.2. Proceso de ejecución

Una vez con los cambios incluidos en el algoritmo, se procederá a ejecutar el programa de forma que la evolución de los circuitos se observa en las siguientes figuras.

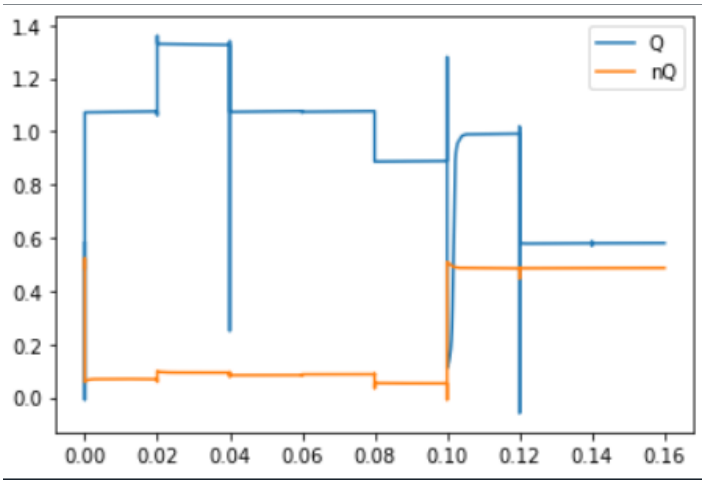


Figura 5.62. Primera generación, padre 1

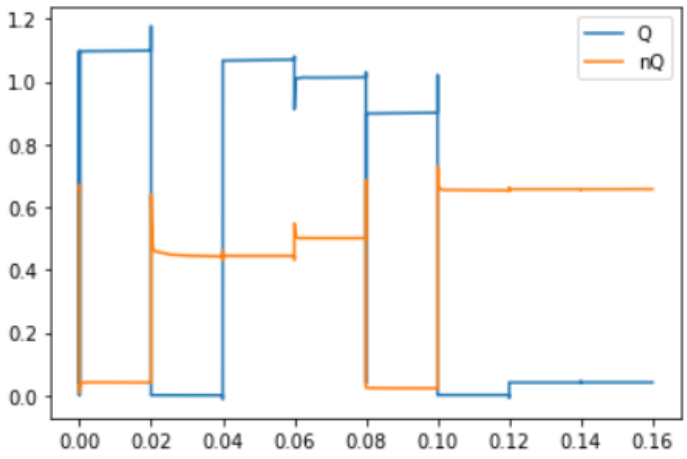


Figura 5.63. Primera generación, padre 2

Ya en la primera generación, podemos observar como el comportamiento se acerca al deseado bastante, pero aún no es suficiente.

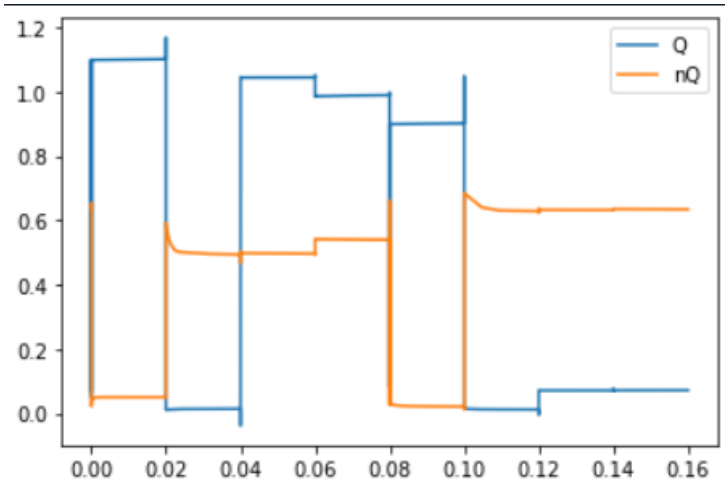


Figura 5.64. Segunda generación, padre 1

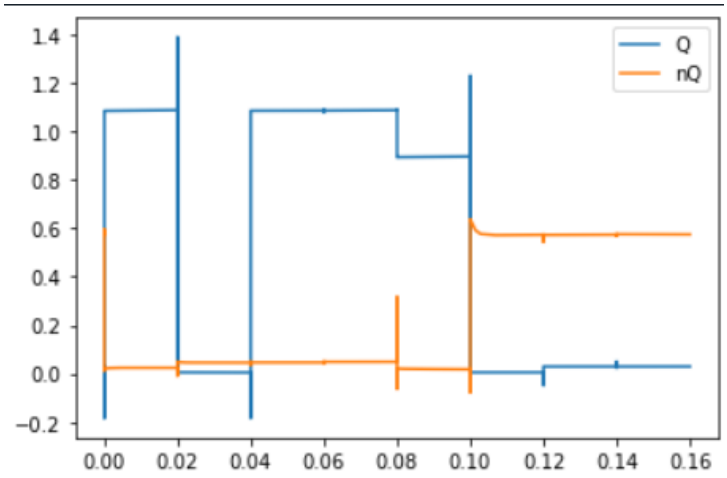


Figura 5.65. Segunda generación, padre 2

De nuevo se puede observar un buen comportamiento respecto al deseado, sin embargo, aún no cumple con el requisito final.

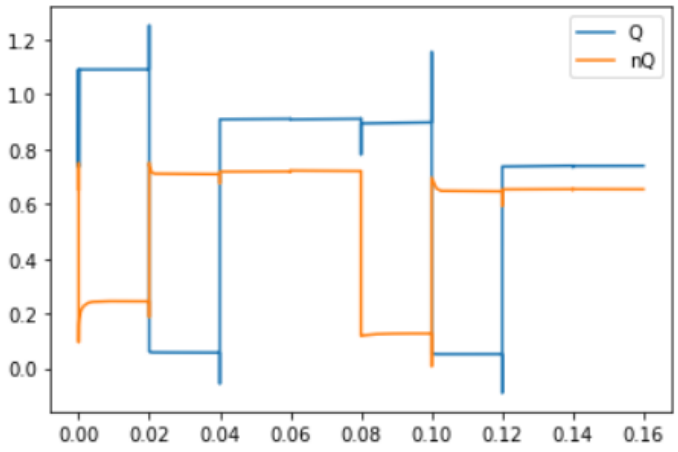


Figura 5.66. Tercera generación, padre 1

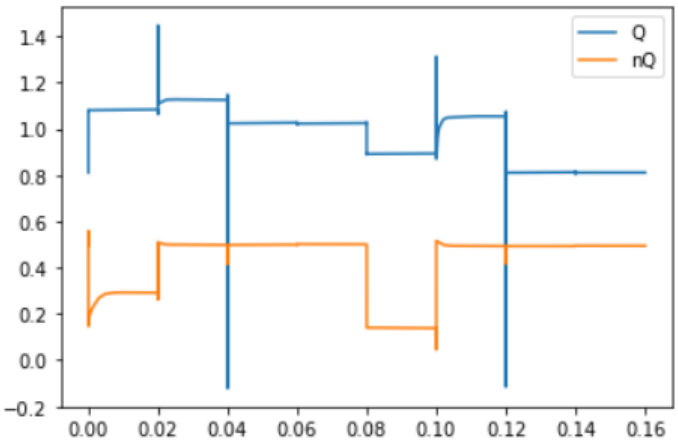


Figura 5.67. Tercera generación, padre 2

En este caso, el comportamiento ha empeorado respecto a anteriores padres, por lo tanto, el algoritmo continuará.

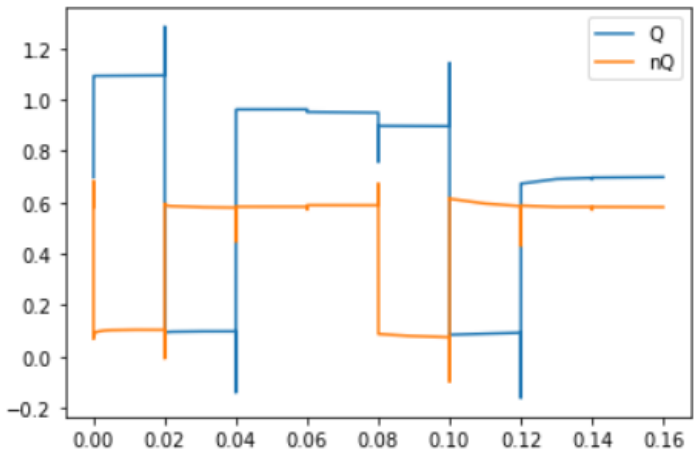


Figura 5.68. Cuarta generación, padre 1

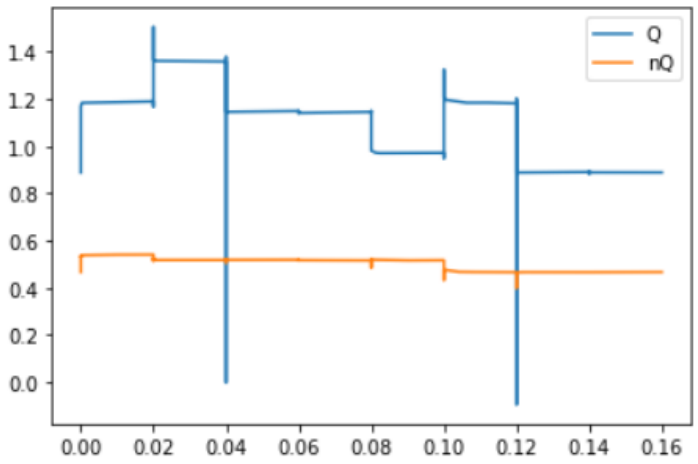


Figura 5.69. Cuarta generación, padre 2

Estos padres, también presentan una simulación parecida a la deseada, pero aún no se cumple el criterio de finalización del algoritmo.

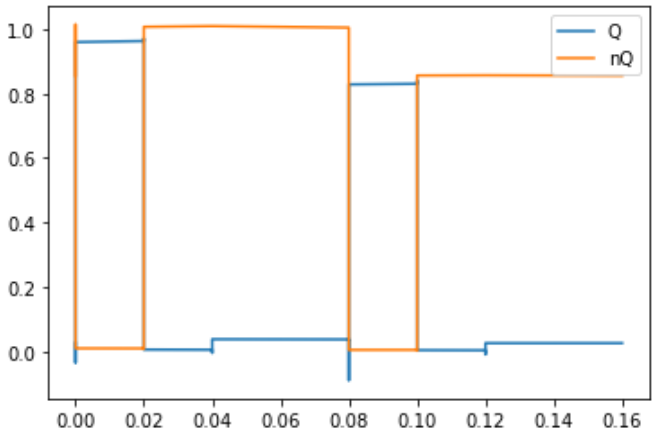


Figura 5.70. Quinta generación, padre 1

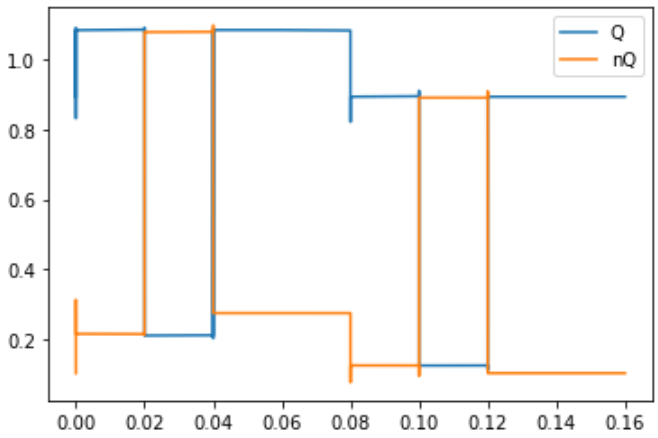


Figura 5.71. Quinta generación, padre 2

Las simulaciones continúan sin cumplir con las especificaciones del circuito final por lo que se proseguirá con la siguiente iteración.

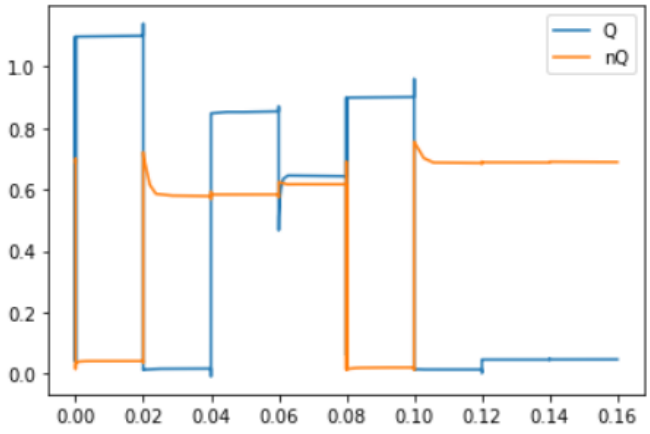


Figura 5.72. Sexta generación, padre 1

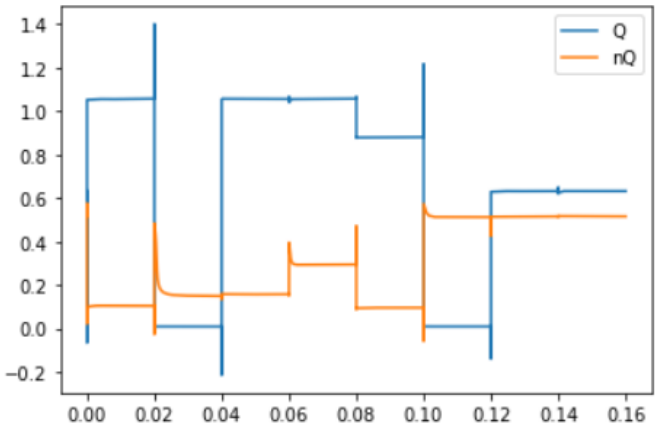


Figura 5.73. Sexta generación, parte 2

Al no cumplir con los requisitos el algoritmo realiza otra iteración.

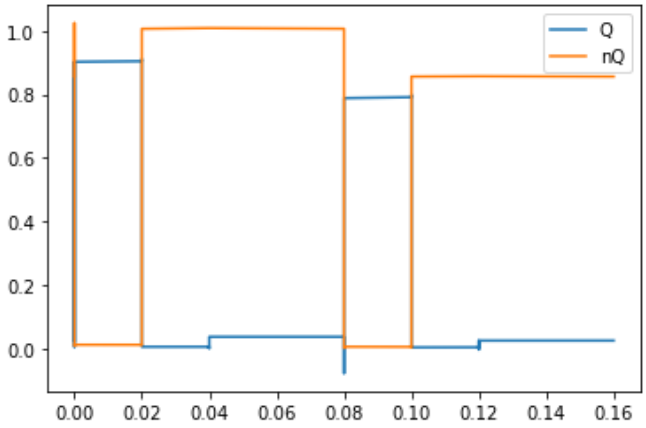


Figura 5.74. Séptima generación, padre 1

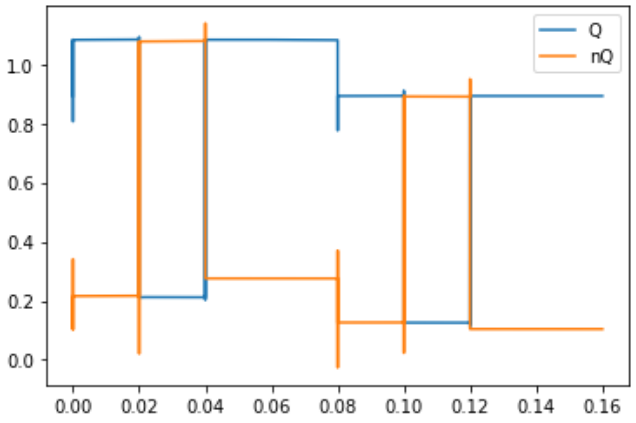


Figura 5.75. Séptima generación, padre 2

El valor de la función objetivo continúa sin ser suficiente.

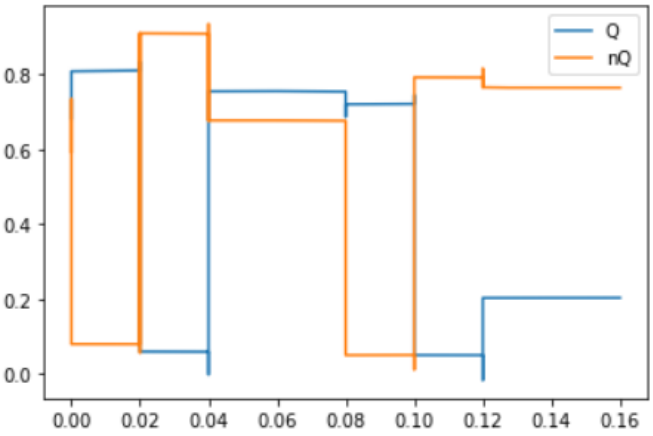
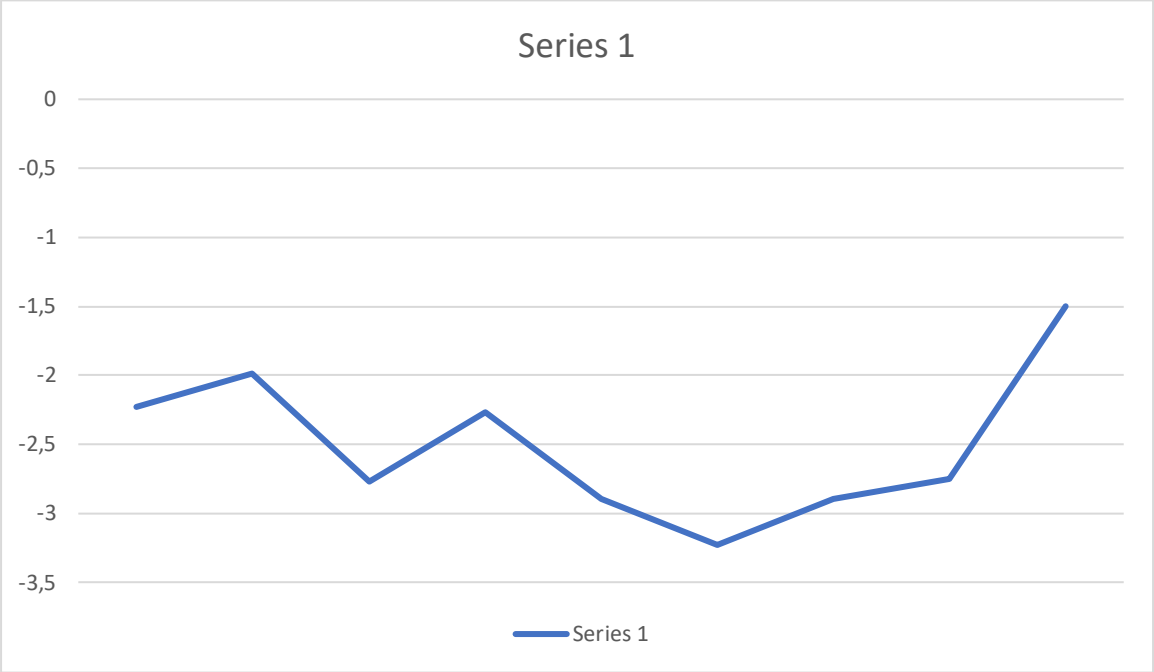


Figura 5.76. Octava generación, padre 1

Este padre de la octava generación, ha superado el requisito para finalizar el algoritmo, y por lo tanto será utilizado como el flip-flop polimórfico que ayude al problema de seguridad más adelante.

5.3.4.3. Evolución del valor objetivo durante las generaciones



5.3.4.4. Análisis de la evolución del circuito y el circuito final

Tras ocho generaciones de circuitos, se ha obtenido un circuito que cumpla la función planteada inicialmente. Sin embargo, al no ser perfecto hay que tener en cuenta que para los valores mayores de Vdd/2 se considerará un 1 digital y para valores menores se considerará un 0. Además, en la Figura 5.77 y la Figura 5.78 se presentan las diferentes secciones de las simulaciones para poder observar claramente como cumple con la tabla de verdad planteada inicialmente.

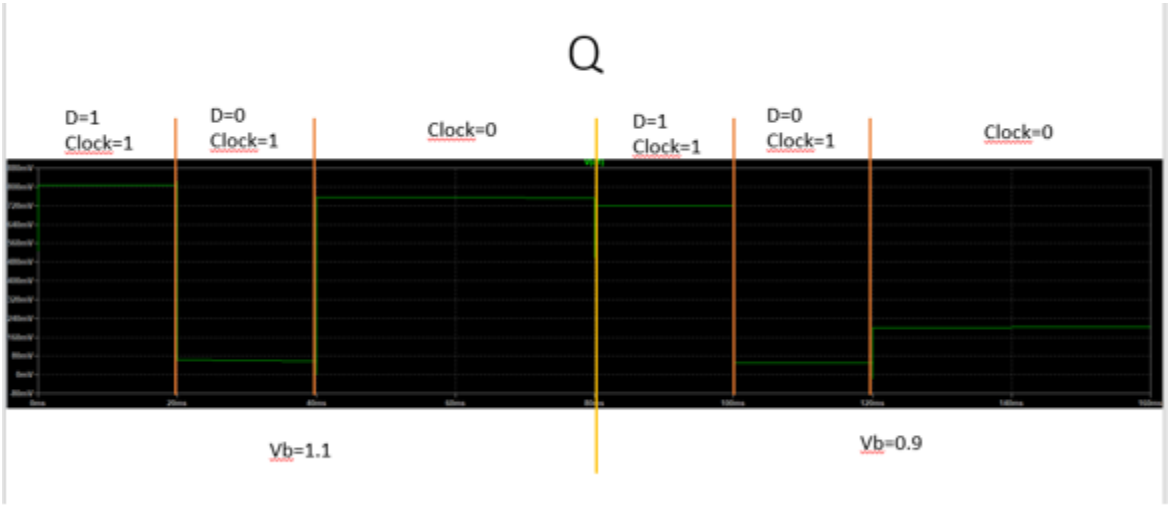


Figura 5.77. Simulación detallada de Q

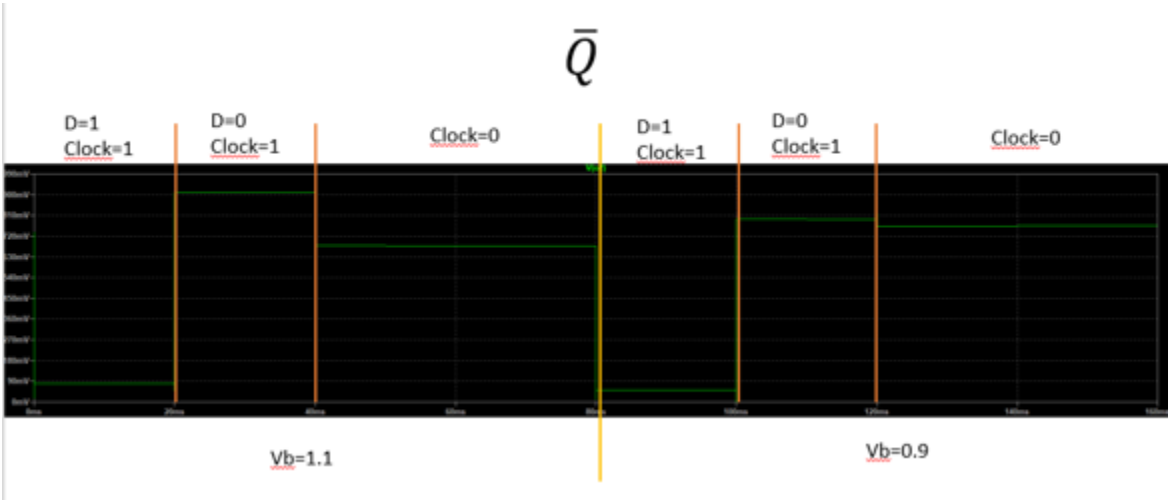


Figura 5.78. Simulación detallada de $\bar{Q}$

Estas simulaciones comparándolas con la tabla de verdad mostrada en la Figura 5.1 y utilizando lo mencionado anteriormente respecto a establecer 1 y 0 digitales, podemos observar, que su comportamiento cumple con la función deseada para del flip-flop y por lo tanto el desarrollo del circuito se puede dar por completado.

6. Flip-flop aplicado a la seguridad

Una vez se ha desarrollado el flip-flop, se explicará sus aplicaciones en el ámbito de la seguridad y como protegerá un sistema de vulnerabilidades físicas. Sin embargo, previo a esto se explicarán los ataques físicos que puede sufrir un circuito para explicar como se evitarían con el circuito polimórfico desarrollado.

6.1. Posibles ataques a un circuito físico

Un circuito puede recibir distintos tipos de ataques. Un ataque pasivo, ataque activo o un ataque semipasivo. El tipo de ataque en el que nos centraremos es el ataque pasivo. En este ataque se trata de utilizar los pines externos de un chip para causar un glitch en el sistema. Por ejemplo, se puede manipular el suministro de voltaje del sistema para que el circuito no funcione como lo esperado, esto podría hacer que no se reciba una instrucción critica sobre la seguridad del sistema y por lo tanto facilitar un ataque al mismo. Un ejemplo de este ataque es el explicado por Buhren et al. (2021). Estos autores, mencionan la posibilidad de manipular el suministro de voltaje para privar a la tecnología AMD’s Secure Encrypted Virtualization de la confidencialidad de su información.

6.2. Solución planteada por el circuito desarrollado

Con el flip-flop polimórfico se plantea una solución a los ataques físicos mediante la manipulación de voltaje. Como se ha mostrado en el ejemplo de la tecnología AMD's Secure Encrypted Virtualization, la información de un sistema puede ser obtenida por un atacante utilizando las modificaciones en el voltaje. Sin embargo, utilizando nuestro circuito, en el caso de que el voltaje subiese del umbral indicado se eliminaría automáticamente la información guardada en el sistema y por lo tanto se protegería la confidencialidad de esta, ya que el flip-flop borraría el antiguo estado y establecería ambos bits en 1.

6.2.1. Mejores ante otras soluciones al problema de seguridad

La mayoría de las opciones que presentan una solución a este problema de seguridad, proponen añadir un sensor que perciba cambios en el voltaje y envíe una señal para proteger la información. Sin embargo, esto tiene dos problemas. En primer lugar, si se desactiva el sensor, el sistema perdería su protección, esto no es posible con el flip-flop desarrollado por que en este no existe un sensor per se, sino que esta intrínseco en el circuito. Además, otro problema de la implementación de un sensor es que, a diferencia del circuito desarrollado, la eliminación de datos no es instantánea y por lo tanto los atacantes tienen un margen de tiempo para obtener la información.

7. Conclusión

Tras el desarrollo del proyecto en su totalidad, se observará si se han logrado los objetivos propuestos inicialmente. El primer objetivo, indicaba desarrollar un algoritmo genético con la

posibilidad de evolucionar cualquier circuito basado en transistores. Esto ha sido cumplido ya que mediante el mismo algoritmo realizándole ligeros cambios, se ha logrado desarrollar varios circuitos con objetivos diferentes.

El segundo objetivo, implicaba desarrollar un circuito polimórfico mediante el algoritmo genético. Esto también se ha logrado al desarrollar dos circuitos polimórficos, la puerta lógica NAND/NOR y el flip-flop.

Por último, se planteaba solucionar un problema real de seguridad ante ataques físicos. Para ello, se desarrollo un flip-flop con comportamiento polimórfico el cual, tras una alteración en el voltaje, borraba la información. Esto implica, que los ataques mediante la manipulación de voltaje no son posibles ante este circuito. Además, este flip-flop presenta una mejora ante las soluciones actuales de esta vulnerabilidad.

8. Bibliografía

Brum, N. (2021). *PyLTSpice's documentation*. Obtenido de pyltspice.readthedocs: https://pyltspice.readthedocs.io/en/latest/

Liu,Y., Tanaka, K., Iwata, M., Higuchi, T., y Yasunaga,M . (2008). *Evolvable Systems: From Biology to Hardware 4th International Conference ICES 2001 Tokyo, October 3-5, 2001 Proceedings.* Springer.

Nanoscale Integration and Modeling (NIMO) Group, ASU. (2012). *Predictive Technology Model (PTM)*. http://ptm.asu.edu/

Ruzicka, R., Sekanina, L., y Prokop, R. (2008, 6-9 de julio). Physical Demonstration of Polymorphic Self-checking Circuits. En 14th IEEE International On-Line Testing Symposium 2008 (pp. 31-36). Grecia

Software Testing Help. (2022, 13 de junio). *Introduction To Genetic Algorithms In Machine Learning*. SoftwareTestingHelp: https://www.softwaretestinghelp.com/genetic-algorithms-in-ml/

Buhren, R., Jacob, H. N., Krachenfels, T., & Seifert, J. P. (2021, November). One glitch to rule them all: Fault injection attacks against AMD's secure encrypted virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (pp. 2875-2889)*.

Djellid-Ouar, A., Cathébras, G., Bancel, F. (2006, septiembre). Supply Voltage Glitches Effects on CMOS Circuits. DTIS'06: Design and Test of Integrated Systems in Nanoscale Technology, Tunis (Tunisia), pp.257-261. fflirmm-00093365f

Goikoetxea, A., Arslan, T., Pickles, S. (2008, septiembre). Detecting Voltage Glitch Attacks on Secure Devices. In *Bio-inspired, Learning and Intelligent Systems for Security, 2008, pp. 75-80.* doi: 10.1109/BLISS.2008.26

9. Anexo I Alineación del proyecto con los ODS

Los campos que incluyen este proyecto tienen una amplia relación con los objetivos de desarrollo sostenible. Por ejemplo, el algoritmo genético que se ha desarrollado puede utilizarse para aumentar la eficiencia de muchos otros sistemas, entre estos el riego y fertilización de la agricultura o la energía renovable. Todo esto a parte de presentar una gran innovación, también ayudaría a lograr una fuente de energía asequible y no contaminante.

La relación de la ciberseguridad es realmente cercana con la mayoría de los ODS, esto se debe la seguridad tecnológica en el mundo moderno no podría ser mas importante. Desde contratos, transacciones o simplemente el uso de lo información todos estos requieren de una seguridad que los proteja. Más específicamente, el circuito desarrollado tiene como objetivo proteger información, lo que es esencial para un mercado sostenible que vendría acompañado de un crecimiento económico. En general, aumentar la seguridad de la información, a su vez aumentará el bienestar de las personas al percibir una verdadera privacidad.

Tras todo lo mencionado se puede concluir que los objetivos de desarrollo sostenible más alineados con este proyecto serían:

- Salud y bienestar
- Energía asequible y no contaminante
- Trabajo decente y crecimiento económico
- Industria, innovación e infraestructura
- Ciudades y comunidades sostenibles
- Producción y consumo responsables

10. Anexo II Código desarrollado algoritmo genético

```
# -*- coding: utf-8 -*-
"""
Created on Sun May 22 06:24:29 2022

@author: Luis de la Mata
"""

'Comment 1: The parser_init.py file contains specific functions to
parse the PUF netlist. ' \
'This parser can be configured to the netlist of your choice'

from parser_init import * ### See Comment 1 above
import numpy as np
import random
import datetime
import math
import os

MUTATION_RATE = 0.2
NETLIST=r"C:\Users\luisd\OneDrive\Documentos\LTspiceXVII\Draft9.asc"
 #File with the basic structure
def generate_parent(lower_limit, upper_limit, step_size=1):
    """
    :param lower_limit: lower limit allowed by a given technology
```

```
node.
    :param upper_limit: upper limit allowed by a given technology
node.
    :param step_size: Determined based on the resolution provided by
the foundry. Default set to 1.
    :return: width value for a transistor in nanometer (nm) scale
    """

    genes = random.randrange(lower_limit, upper_limit, step_size)
    return genes * 1e-9

def get_fitness(lengths,widths,vh,vl,netlist1,netlist2,paint,n):
    """
    The 'netlist_modifier' and 'output_parser" functions are written
to parse the given netlist and its corresponding
    measurement file after the simulation in LTSPICE. These
functions are included in the parser.py file.

    The fitness function here is the propagation delay at the
output. This can be modified accordingly.

    :param Guesses: represents the potential values of widths of the
transistors
    :param spice_file_without_extension: the netlist which will be
simulated in HSPICE
    :return: the fitness value from the measurement file
    """

    works=netlist_modifier(NETLIST,
                lengths,widths,vh,vl,False,str(n))
    if works:
        outputs = output_parser(netlist1,netlist2,paint)
        for i in range(len(outputs)):
            for j in range(len(outputs[i])):
```

```
                if outputs[i][j]>0.9:
                    outputs[i][j]=0.9

        fitnessv1=(outputs[0][0]-0.9)+(0-
outputs[0][1])+(outputs[0][2]-0.9)*1.5+(outputs[0][3]-0.9)*1.5+(0-
outputs[1][0])+(outputs[1][1]-0.9)+(outputs[1][2]-
0.9)*1.5+(outputs[1][3]-0.9)*1.5
        fitnessv2=(outputs[0][4]-0.9)+(0-outputs[0][5])+(0-
outputs[1][4])+(outputs[1][5]-0.9)
        fitt=((0-outputs[0][6])+(0-outputs[0][7])+(outputs[1][6]-
0.9)+(outputs[1][7]-0.9))*1.5
        fit=fitnessv1+fitnessv2+fitt
        fitness = np.prod(fit)
        if paint:
            print(outputs)
    else:
        fitness=-40
        os.system("taskkill /f /im XVIIx64.exe")
    return fitness

def population_generator():
    lower_value, upper_value, step_size = 100, 50000, 10
    n = 8 ## Number of transistor widths needed in the netlist
    population=[[0 for c in range(18)] for b in range(100)]
    for i in range(100):
        widths = [generate_parent(lower_value,upper_value,step_size)
 for i in range(n)]
        lengths = [generate_parent(lower_value,upper_value,step_size
) for i in range(n)]
        vh = 1.1
        vl= 0.9
        chromosome=[lengths,widths,vh,vl]
        population[i]=chromosome
```

```
    return population

def crossover(parent1, parent2,bestFitness):
    t1=np.zeros(8)
    t2=np.zeros(8)
    new_population=[[0 for c in range(10)] for b in range(100)]

    for i in range(100):
        child=[[0 for c in range(8)] for b in range(4)]
        used1=np.empty(8,int)
        used2=np.empty(8,int)

        lengths=np.zeros(8)
        widths=np.zeros(8)
        ctrl=0
        while(ctrl<7):
            p1=random.randrange(0, 8, 1)
            if p1 in used1:
                while p1 in used1:
                    p1=random.randrange(0, 8, 1)

            p2=random.randrange(10, 18, 1) - 10
            if p2 in used2:
                while p2 in used2:
                    p2=random.randrange(10, 18, 1)-10


            used1[ctrl]=p1
            used2[ctrl]=p2
            l1=parent1[0][p1]
            lengths[ctrl]=l1
            w1=parent1[1][p1]
            widths[ctrl]=w1
            ctrl+=1
```

```
            l2=parent1[0][p2]
            lengths[ctrl]=l2
            w2=parent1[1][p2]
            widths[ctrl]=w2
            ctrl+=1

        child[0]=lengths
        child[1]=widths

        if random.randrange(0, 2, 1) > 0:
            vii=parent1[2]
            vi=parent1[3]
        else:
            vii=parent2[2]
            vi=parent2[3]
        if vi>=vii:
            vh=vi
            vl=vii
        else:
            vh=vii
            vl=vi

        child[2]=vh
        child[3]=vl

        if random.uniform(0, 10)< (MUTATION_RATE*10):
            for p in range(8):
                if random.uniform(0, 1)< (MUTATION_RATE+0.1):
                    ranil=rangoi(child[0][p],bestFitness)
                    lowl=(child[0][p]-ranil)*1e9

                    ransl=rangos(child[0][p],bestFitness)
                    upl=(child[0][p]+ransl)*1e9
```

```
                    raniw=rangoi(child[1][p],bestFitness)
                    loww=(child[1][p]-raniw)*1e9

                    ransw=rangos(child[0][p],bestFitness)
                    upw=(child[1][p]+ransw)*1e9

                    child[0][p]=mutate(child[0][p], lowl, upl, 10)
                    child[1][p]=mutate(child[1][p], loww, upw, 10)
        new_population[i]=child
    return new_population


def tournament(fitness_pop):
    parent=[[0 for c in range(2)] for b in range(2)]
    vec=[]
    tournament_pop=[]

    #Get a part of the population for the tournament
    tournament_pop=np.random.choice(fitness_pop,90)
    vec=tournament_pop

    #Calculate the maximum of the fitness values and its population
index
    max1=max(vec)
    indexx=np.where(fitness_pop==max1)
    index=int(indexx[0][0])

    #Deleting the first father so we don't chose it again
    selectionIndex=np.where(tournament_pop==max1)
    vec=np.delete(vec,selectionIndex[0][0])

    #Second father selection
    max2=max(vec)
    indexs2=np.where(tournament_pop==max2)
```

```
    index2=int(indexs2[0][0])

    parent[0]=[index,max1]
    parent[1]=[index2,max2]
    return parent

def mutate(parent, lower_limit, upper_limit, step_size=1):
    """
    :param lower_limit: lower limit allowed by a given technology
node.
    :param upper_limit: upper limit allowed by a given technology
node.
    :param step_size: Determined based on the resolution provided by
the foundry. Default set to 1.
    :return: a mutated random width value for a transistor in
nanometer (nm) scale
    """
    childGene = parent
    geneSet = [i for i in np.arange(lower_limit, upper_limit, step_s
ize)]
    newGene, alternate = np.random.choice(geneSet, 2)
    childGene = alternate if newGene == childGene else newGene
    return childGene*1e-9

def display(parent,pos):

    timeDiff = datetime.datetime.now() - startTime
    lengths=parent[0]
    widths=parent[1]
    vh=parent[2]
    vl=parent[3]
    fitness = get_fitness(lengths,widths,vh,vl,"netlist"+str(pos)+"0
"+".raw","netlist"+str(pos)+"1"+".raw",True,0)
    print(fitness)
```

```
    return fitness
def rangoi(n,fit):
    rani=(n-500e-9)/4
    if fit>-1.5:
        rani=(n-500e-9)/4
    if fit>-1:
        rani=(n-500e-9)/5
    return rani

def rangos(n,fit):
    rans=(500000e-9-n)/4
    if fit>-1.5:
        rans=(500000e-9-n)/4
    if fit>-1:
        rans=(500000e-9-n)/5
    return rans

if __name__=='__main__':
    newParent=0
    cont=0
    random.seed()
    fitness_vec=[]
    startTime = datetime.datetime.now()
    bestFitness=-20
    globalBestFitness=-20
    population=population_generator()
    fitness_pop=np.empty(len(population),float)
    childs=population
    while bestFitness<-1.2:
        cont+=1
        print("New Population")
        print("Number: "+str(cont))
        for i in range(len(childs)):
            chromosome=childs[i]
```

```
            fitness_pop[i]=get_fitness(chromosome[0], chromosome[1],
 chromosome[2],chromosome[3], "netlist"+str(i)+"0"+".raw","netlist"+
str(i)+"1"+".raw", False,i)
        ind=tournament(fitness_pop)
        parent1=population[ind[0][0]]
        fit1=display(parent1,ind[0][0])
        parent2=population[ind[1][0]]
        fit2=display(parent2,ind[1][0])
        bestFitness=max(fit1,fit2)
        fitness_vec.append(bestFitness)
        if bestFitness > globalBestFitness:
            globalBestFitness=bestFitness
            if fit1 > fit2:
                bestParent=parent1
            else:
                bestParent=parent2
            netlist_modifier(NETLIST,
                        bestParent[0],bestParent[1],bestParent[2],be
stParent[3],True,str(newParent))
            newParent+=1
        childs=crossover(parent1,parent2,bestFitness)
    endt=datetime.datetime.now()
    print("Algorithm finished")
    print("Number: "+str(cont))
    print("Total time = "+str(endt-startTime))
```

11. Anexo III Código desarrollado para la interacción con LTspice

```
import time
from PyLTSpice.LTSpice_RawRead import LTSpiceRawRead
from PyLTSpice.LTSpiceBatch import SimCommander
from matplotlib import pyplot as plt

def output_parser(output_file1,output_file2,paint):
    LTR0 = LTSpiceRawRead(output_file1)
    LTR1 = LTSpiceRawRead(output_file2)
    IR0 = LTR0.get_trace("V(o1)")
    IR1 = LTR1.get_trace("V(o2)")
    x0 = LTR0.get_trace('time')  # Gets the time axis
    x1 = LTR1.get_trace('time')  # Gets the time axis
    v0=output_points(x0, IR0)
    v1=output_points(x1, IR1)
    if paint:
        steps0 = LTR0.get_steps()
        steps1 = LTR1.get_steps()
```

```
        for step in range(len(steps0)):
            plt.plot(x0.get_time_axis(step), IR0.get_wave(
step), label='Q')

        for step in range(len(steps1)):
            plt.plot(x1.get_time_axis(step), IR1.get_wave(
step), label='nQ')

        plt.legend()  # order a legend
        plt.show()

    v0l=output_points(x0, IR0)
    v1l=output_points(x1, IR1)
    measurements=[v0,v1]
    return measurements


def output_points(x,IR):
    time=x.get_time_axis(step=0).tolist()
    rounded_time = [round(num, 3) for num in time]
    out=IR.get_wave(step=0)

    found=False
    ttime=0.020
    while found==False:
        if ttime in rounded_time:
            found=True
        else:
            ttime=round(ttime-0.001,3)
```

```
    t1=rounded_time.index(ttime)

    found=False
    ttime=0.040
    while found==False:
        if ttime in rounded_time:
            found=True
        else:
            ttime=round(ttime-0.001,3)
    t2=rounded_time.index(ttime)

    found=False
    ttime=0.060
    while found==False:
        if ttime in rounded_time:
            found=True
        else:
            ttime=round(ttime-0.001,3)
    t3=rounded_time.index(ttime)

    found=False
    ttime=0.080
    while found==False:
        if ttime in rounded_time:
            found=True
        else:
            ttime=round(ttime-0.001,3)
    t4=rounded_time.index(ttime)
```

```
    found=False
    ttime=0.081
    while found==False:
        if ttime in rounded_time:
            found=True
        else:
            ttime=round(ttime+0.001,3)
    t5=rounded_time.index(ttime)

    found=False
    ttime=0.120
    while found==False:
        if ttime in rounded_time:
            found=True
        else:
            ttime=round(ttime-0.001,3)
    t6=rounded_time.index(ttime)

    found=False
    ttime=0.121
    while found==False:
        if ttime in rounded_time:
            found=True
        else:
            ttime=round(ttime+0.001,3)
    t7=rounded_time.index(ttime)

    results=[out[t1],out[t2],out[t3],out[t4],out[t5],out[t
6],out[t7],out[-1]]
```

```
    return results


def netlist_modifier(netlist_file,modified_lengths,modifie
d_widths,vh,vl,save,n):
    # select spice model
    LTC = SimCommander(netlist_file)
    for i in range(2):
        LTC.set_element_model('M1', 'Vdd PMOS
l='+str(modified_lengths[0])+'
w='+str(modified_widths[0]))
        LTC.set_element_model('M2', 'Vdd PMOS
l='+str(modified_lengths[1])+'
w='+str(modified_widths[1]))
        LTC.set_element_model('M3', 'Vdd PMOS
l='+str(modified_lengths[2])+'
w='+str(modified_widths[2]))
        LTC.set_element_model('M4', 'Vdd PMOS
l='+str(modified_lengths[3])+'
w='+str(modified_widths[3]))
        LTC.set_element_model('M5', 'Vdd PMOS
l='+str(modified_lengths[4])+'
w='+str(modified_widths[4]))
        LTC.set_element_model('M6', '0 NMOS
l='+str(modified_lengths[5])+'
w='+str(modified_widths[5]))
        LTC.set_element_model('M7', '0 NMOS
l='+str(modified_lengths[6])+'
w='+str(modified_widths[6]))
        LTC.set_element_model('M8', '0 NMOS
```

```
l='+str(modified_lengths[7])+'
w='+str(modified_widths[7]))
        LTC.set_element_model('M9', 'N025 PMOS
l='+str(modified_lengths[0])+'
w='+str(modified_widths[0]))
        LTC.set_element_model('M10', 'N025 PMOS
l='+str(modified_lengths[1])+'
w='+str(modified_widths[1]))
        LTC.set_element_model('M11', 'N025 PMOS
l='+str(modified_lengths[1])+'
w='+str(modified_widths[2]))
        LTC.set_element_model('M12', 'N025 PMOS
l='+str(modified_lengths[3])+'
w='+str(modified_widths[3]))
        LTC.set_element_model('M13', 'N025 PMOS
l='+str(modified_lengths[4])+'
w='+str(modified_widths[4]))
        LTC.set_element_model('M14', '0 NMOS
l='+str(modified_lengths[5])+'
w='+str(modified_widths[5]))
        LTC.set_element_model('M15', '0 NMOS
l='+str(modified_lengths[6])+'
w='+str(modified_widths[6]))
        LTC.set_element_model('M16', '0 NMOS
l='+str(modified_lengths[7])+'
w='+str(modified_widths[7]))
        LTC.set_component_value('V1', 'PULSE(0
'+str((vh+vl)/2)+' 0 5n 5n 20m 40m)')
        LTC.set_component_value('V2', 'PULSE(0
```

```
'+str((vh+vl)/2)+' 0 5n 5n 40m 80m)')
        LTC.set_component_value('V4', 'PULSE('+str(vl)+'
'+str(vh)+' 0 5n 5n 80m 160m)')
        LTC.set_component_value('V5', 'PULSE('+str(vl)+'
'+str(vh)+' 0 5n 5n 80m 160m)')
        LTC.set_component_value('V3', str((vh+vl)/2))
        LTC.set_component_value('V12', str((vh+vl)/2))
        LTC.set_component_value('V6', str((vh+vl)/2))

        if i==0:
            LTC.add_instructions(
                "; Simulation settings",
                ".tran 0 160m",
                ".TEMP 27",
                ".print V(O1)"
            )
        else:
           LTC.add_instructions(
                "; Simulation settings",
                ".tran 0 160m",
                ".TEMP 27",
                ".print V(O2)"
            )
        if save:
            LTC.write_netlist("netlistf"+n+str(i)+".net")
            LTC.run(run_filename="netlistf"+n+str(i)+".net
")
            comp=LTC.wait_completion()
            LTC.reset_netlist()
```

```
        else:

            print("Netlist number:"+n+str(i))
            LTC.write_netlist("netlist"+n+str(i)+".net")
            LTC.run(run_filename="netlist"+n+str(i)+".net"
)
            comp=LTC.wait_completion()
            LTC.reset_netlist()
    return comp
```

12. Anexo IV Transistor utilizado (PTM 45nm BSIM4)

```
* Beta Version released on 2/22/06

* PTM 45nm NMOS

.model  nmos  nmos  level = 54

+version = 4.0          binunit = 1           paramchk= 1
mobmod  = 0
+capmod  = 2            igcmod  = 1           igbmod  = 1
geomod  = 1
+diomod  = 1            rdsmod  = 0           rbodymod= 1
rgatemod= 1
+permod  = 1            acnqsmod= 0           trnqsmod= 0

+tnom    = 27           toxe    = 1.75e-9     toxp    = 1.1e-9       toxm
= 1.75e-9
+dtox    = 0.65e-9      epsrox  = 3.9         wint    = 5e-009       lint
= 3.75e-009
+ll      = 0            wl      = 0           lln     = 1            wln
= 1
+lw      = 0            ww      = 0           lwn     = 1            wwn
= 1
+lwl     = 0            wwl     = 0           xpart   = 0
toxref  = 1.75e-9
+xl      = -20e-9
+vth0    = 0.466        k1      = 0.4         k2      = 0.0          k3
= 0
+k3b     = 0            w0      = 2.5e-006    dvt0    = 1            dvt1
= 2
```

```
+dvt2     = 0             dvt0w    = 0             dvt1w    = 0             dvt2w
= 0
+dsub     = 0.1           minv     = 0.05          voffl    = 0             dvtp0
= 1.0e-010
+dvtp1    = 0.1           lpe0     = 0             lpeb     = 0             xj
= 1.4e-008
+ngate    = 2e+020        ndep     = 3.24e+018     nsd      = 2e+020        phin
= 0
+cdsc     = 0.000         cdscb    = 0             cdscd    = 0             cit
= 0
+voff     = -0.13         nfactor  = 2.1           eta0     = 0.0049        etab
= 0
+vfb      = -0.55         u0       = 0.04398       ua       = 6e-010        ub
= 1.2e-018
+uc       = 0             vsat     = 147390        a0       = 1.0           ags
= 0
+a1       = 0             a2       = 1.0           b0       = 0             b1
= 0
+keta     = 0.04          dwg      = 0             dwb      = 0             pclm
= 0.02
+pdiblc1  = 0.001         pdiblc2  = 0.001         pdiblcb  = -0.005        drout
= 0.5
+pvag     = 1e-020        delta    = 0.01          pscbe1   = 8.14e+008
pscbe2   = 1e-007
+fprout   = 0.2           pdits    = 0.08          pditsd   = 0.23
pditsl   = 2.3e+006
+rsh      = 5             rdsw     = 155           rsw      = 80            rdw
= 80
+rdswmin  = 0             rdwmin   = 0             rswmin   = 0             prwg
= 0
+prwb     = 0             wr       = 1             alpha0   = 0.074
alpha1   = 0.005
+beta0    = 30            agidl    = 0.0002        bgidl    = 2.1e+009      cgidl
= 0.0002
+egidl    = 0.8

+aigbacc  = 0.012         bigbacc  = 0.0028        cigbacc  = 0.002
```

```
+nigbacc = 1            aigbinv = 0.014        bigbinv = 0.004
cigbinv = 0.004
+eigbinv = 1.1          nigbinv = 3            aigc    = 0.012       bigc
= 0.0028
+cigc    = 0.002        aigsd   = 0.012        bigsd   = 0.0028      cigsd
= 0.002
+nigc    = 1            poxedge = 1            pigcd   = 1           ntox
= 1

+xrcrg1  = 12           xrcrg2  = 5
+cgso    = 1.1e-010     cgdo    = 1.1e-010     cgbo    = 2.56e-011   cgdl
= 2.653e-10
+cgsl    = 2.653e-10    ckappas = 0.03         ckappad = 0.03        acde
= 1
+moin    = 15           noff    = 0.9          voffcv  = 0.02

+kt1     = -0.11        kt11    = 0            kt2     = 0.022       ute
= -1.5
+ua1     = 4.31e-009    ub1     = 7.61e-018    uc1     = -5.6e-011   prt
= 0
+at      = 33000

+fnoimod = 1            tnoimod = 0

+jss     = 0.0001       jsws    = 1e-011       jswgs   = 1e-010      njs
= 1
+ijthsfwd= 0.01         ijthsrev= 0.001        bvs     = 10          xjbvs
= 1
+jsd     = 0.0001       jswd    = 1e-011       jswgd   = 1e-010      njd
= 1
+ijthdfwd= 0.01         ijthdrev= 0.001        bvd     = 10          xjbvd
= 1
+pbs     = 1            cjs     = 0.0005       mjs     = 0.5         pbsws
= 1
+cjsws   = 5e-010       mjsws   = 0.33         pbswgs  = 1
cjswgs  = 3e-010
```

```
+mjswgs   = 0.33          pbd      = 1             cjd      = 0.0005        mjd
= 0.5
+pbswd    = 1             cjswd    = 5e-010        mjswd    = 0.33
pbswgd   = 1
+cjswgd   = 5e-010        mjswgd   = 0.33          tpb      = 0.005         tcj
= 0.001
+tpbsw    = 0.005         tcjsw    = 0.001         tpbswg   = 0.005
tcjswg   = 0.001
+xtis     = 3             xtid     = 3

+dmcg     = 0e-006        dmci     = 0e-006        dmdg     = 0e-006        dmcgt
= 0e-007
+dwj      = 0.0e-008      xgw      = 0e-007        xgl      = 0e-008

+rshg     = 0.4           gbmin    = 1e-010        rbpb     = 5             rbpd
= 15
+rbps     = 15            rbdb     = 15            rbsb     = 15            ngcon
= 1

* PTM 45nm PMOS

.model   pmos   pmos   level = 54

+version = 4.0           binunit = 1              paramchk= 1
mobmod   = 0
+capmod   = 2            igcmod   = 1             igbmod   = 1
geomod   = 1
+diomod   = 1            rdsmod   = 0             rbodymod= 1
rgatemod= 1
+permod   = 1            acnqsmod= 0              trnqsmod= 0

+tnom     = 27           toxe     = 1.85e-009     toxp     = 1.1e-009      toxm
= 1.85e-009
+dtox     = 0.75e-9      epsrox   = 3.9           wint     = 5e-009        lint
= 3.75e-009
+ll       = 0            wl       = 0             lln      = 1             wln
= 1
```

```
+lw       = 0             ww       = 0             lwn      = 1             wwn
= 1
+lwl      = 0             wwl      = 0             xpart    = 0
toxref   = 1.85e-009
+xl       = -20e-9
+vth0     = -0.4118       k1       = 0.4           k2       = -0.01         k3
= 0
+k3b      = 0             w0       = 2.5e-006      dvt0     = 1             dvt1
= 2
+dvt2     = -0.032        dvt0w    = 0             dvt1w    = 0             dvt2w
= 0
+dsub     = 0.1           minv     = 0.05          voffl    = 0             dvtp0
= 1e-011
+dvtp1    = 0.05          lpe0     = 0             lpeb     = 0             xj
= 1.4e-008
+ngate    = 2e+020        ndep     = 2.44e+018     nsd      = 2e+020        phin
= 0
+cdsc     = 0.000         cdscb    = 0             cdscd    = 0             cit
= 0
+voff     = -0.126        nfactor  = 2.1           eta0     = 0.0049        etab
= 0
+vfb      = 0.55          u0       = 0.00440       ua       = 2.0e-009      ub
= 0.5e-018
+uc       = 0             vsat     = 70000         a0       = 1.0           ags
= 1e-020
+a1       = 0             a2       = 1             b0       = 0             b1
= 0
+keta     = -0.047        dwg      = 0             dwb      = 0             pclm
= 0.12
+pdiblc1  = 0.001         pdiblc2  = 0.001         pdiblcb  = 3.4e-008      drout
= 0.56
+pvag     = 1e-020        delta    = 0.01          pscbe1   = 8.14e+008
pscbe2   = 9.58e-007
+fprout   = 0.2           pdits    = 0.08          pditsd   = 0.23
pditsl   = 2.3e+006
+rsh      = 5             rdsw     = 155           rsw      = 75            rdw
= 75
```

```
+rdswmin = 0            rdwmin   = 0            rswmin   = 0            prwg
= 0
+prwb    = 0            wr       = 1            alpha0   = 0.074
alpha1   = 0.005
+beta0   = 30           agidl    = 0.0002       bgidl    = 2.1e+009     cgidl
= 0.0002
+egidl   = 0.8

+aigbacc = 0.012        bigbacc = 0.0028        cigbacc = 0.002
+nigbacc = 1            aigbinv = 0.014         bigbinv = 0.004
cigbinv = 0.004
+eigbinv = 1.1          nigbinv = 3             aigc     = 0.69         bigc
= 0.0012
+cigc    = 0.0008       aigsd   = 0.0087        bigsd    = 0.0012       cigsd
= 0.0008
+nigc    = 1            poxedge = 1             pigcd    = 1            ntox
= 1

+xrcrg1  = 12           xrcrg2  = 5
+cgso    = 1.1e-010     cgdo    = 1.1e-010      cgbo     = 2.56e-011    cgdl
= 2.653e-10
+cgsl    = 2.653e-10    ckappas = 0.03          ckappad = 0.03         acde
= 1
+moin    = 15           noff    = 0.9           voffcv   = 0.02

+kt1     = -0.11        kt1l    = 0             kt2      = 0.022        ute
= -1.5
+ua1     = 4.31e-009    ub1     = 7.61e-018     uc1      = -5.6e-011    prt
= 0
+at      = 33000

+fnoimod = 1            tnoimod = 0

+jss     = 0.0001       jsws    = 1e-011        jswgs    = 1e-010       njs
= 1
+ijthsfwd= 0.01         ijthsrev= 0.001         bvs      = 10           xjbvs
= 1
```

```
+jsd      = 0.0001        jswd     = 1e-011        jswgd    = 1e-010        njd
= 1
+ijthdfwd= 0.01           ijthdrev= 0.001          bvd      = 10            xjbvd
= 1
+pbs      = 1             cjs      = 0.0005        mjs      = 0.5           pbsws
= 1
+cjsws    = 5e-010        mjsws    = 0.33          pbswgs   = 1
cjswgs   = 3e-010
+mjswgs   = 0.33          pbd      = 1             cjd      = 0.0005        mjd
= 0.5
+pbswd    = 1             cjswd    = 5e-010        mjswd    = 0.33
pbswgd   = 1
+cjswgd   = 5e-010        mjswgd   = 0.33          tpb      = 0.005         tcj
= 0.001
+tpbsw    = 0.005         tcjsw    = 0.001         tpbswg   = 0.005
tcjswg   = 0.001
+xtis     = 3             xtid     = 3

+dmcg     = 0e-006        dmci     = 0e-006        dmdg     = 0e-006        dmcgt
= 0e-007
+dwj      = 0.0e-008      xgw      = 0e-007        xgl      = 0e-008

+rshg     = 0.4           gbmin    = 1e-010        rbpb     = 5             rbpd
= 15
+rbps     = 15            rbdb     = 15            rbsb     = 15            ngcon
= 1
```

13. Anexo V Netlist del Flip-Flop polimórfico

V1 D 0 PULSE(0 1.0 0 5n 5n 20m 40m)

V2 CLK 0 PULSE(0 1.0 0 5n 5n 40m 80m)

V5 Vdd 0 PULSE(0.9 1.1 0 5n 5n 80m 160m)

V4 N025 0 PULSE(0.9 1.1 0 5n 5n 80m 160m)

M1 Vdd N001 O1 Vdd PMOS l=8.07e-06 w=2.881e-05

M2 N009 O2 0 Vdd PMOS l=3.2370000000000003e-05 w=1.081e-05

M3 Vdd Vdd O1 Vdd PMOS l=4.869e-05 w=1.063e-05

M4 Vdd Vdd Vdd Vdd PMOS l=2.994e-05 w=2.0300000000000002e-05

M5 0 N001 N009 Vdd PMOS l=2.7200000000000002e-06 w=4.7340000000000004e-05

M6 Vdd Vdd N009 0 NMOS l=2.12e-05 w=3.516e-05

M7 O1 N001 0 0 NMOS l=3.719e-05 w=6.54e-06

M8 O1 O2 0 0 NMOS l=9.34e-06 w=8.44e-06

M9 N025 O1 O2 N025 PMOS l=8.07e-06 w=2.881e-05

M10 N032 N024 0 N025 PMOS l=3.2370000000000003e-05 w=1.081e-05

M11 N025 N025 O2 N025 PMOS l=3.2370000000000003e-05 w=1.063e-05

M12 N025 N025 N025 N025 PMOS l=2.994e-05 w=2.0300000000000002e-05

M13 0 O1 N032 N025 PMOS l=2.7200000000000002e-06 w=4.7340000000000004e-05

M14 N025 N025 N032 0 NMOS l=2.12e-05 w=3.516e-05

```
M15 O2 O1 0 0 NMOS l=3.719e-05 w=6.54e-06

M16 O2 N024 0 0 NMOS l=9.34e-06 w=8.44e-06

Q1 N006 N015 0 0 2SC4097

R1 N010 N006 10k

R2 N015 0 47k

R3 N015 D 4.7k

V3 N010 0 1.0

M17 N002 N006 N003 N007 PMOS l=1u w=1u

M19 N003 N006 N014 N012 NMOS l=1u w=1u

V12 N002 0 1.0

V6 N018 0 1.0

M18 N002 N003 N001 N004 PMOS l=1u w=1u

M21 N002 N003 N001 N005 PMOS l=1u w=1u

M22 N018 CLK N019 N022 PMOS l=1u w=1u

M25 N018 D N019 N023 PMOS l=1u w=1u

M26 N018 N019 N024 N020 PMOS l=1u w=1u

M29 N018 N019 N024 N021 PMOS l=1u w=1u

M20 N014 CLK 0 N017 NMOS l=1u w=1u

M23 N001 N003 N013 N011 NMOS l=1u w=1u

M24 N013 N003 0 N016 NMOS l=1u w=1u

M27 N019 CLK N029 N027 NMOS l=1u w=1u

M28 N029 D 0 N031 NMOS l=1u w=1u
```

```
M30 N024 N019 N028 N026 NMOS l=1u w=1u

M31 N028 N019 0 N030 NMOS l=1u w=1u

M32 N002 CLK N003 N008 PMOS l=1u w=1u

.model NPN NPN

.model PNP PNP

.lib C:\Users\Luis\Documents\LTspiceXVII\lib\cmp\standard.bjt

.model NMOS NMOS

.model PMOS PMOS

.lib C:\Users\Luis\Documents\LTspiceXVII\lib\cmp\standard.mos

.tran 0 160m

.lib C:\Users\Luis\Desktop\TFGCode/45nm_bulk.lib

; Simulation settings

.TEMP 27

.backanno

.end
```