Instituto Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Diseño de un algoritmo para la emulación por software del protocolo I^2C para la recepción de datos

Informe de Proyecto de Graduación para optar por el título de Ingeniero en Electrónica con el grado académico de Licenciatura

Iliak Flores Barrantes

Cartago, 28 de septiembre de 2025

Índice general

Índice de figuras

Índice de tablas

Lista de símbolos y abreviaciones

- SV: System Verilog
- $kbits$: Kilo bits por segundo
- $Mbits$: Mega bits por segundo
- MSB : Most Significant Bit
- LSB: Least Significant Bit
- I^2C.: Inter-Integrated Circuit
- ALU: Unidad aritmético lógica
- $RISC$: Reduced Instruction Set Computing

Capítulo 1

Introducción

En este capítulo inicial se presenta de forma clara el problema central que da origen al desarrollo del proyecto. Asimismo, se explica la relevancia de abordar dicha problemática y se introduce la solución propuesta para cumplir con los objetivos planteados.

1.1. Antecedentes

En el área de ingeniería electrónica existen múltiples formas de transmitir información entre dispositivos, ya sea de forma analógica o digital. La comunicación entre diferentes tipos de dispositivos es vital para la operación y trabajo de estos.

Esto resulta especialmente importante cuando se habla de microcontroladores, los cuales están diseñados para operar en conjunto con varios sistemas simultáneamente. La correcta transmisión y recepción de datos se vuelve crucial para el cumplimiento de sus tareas, y para ello se han desarrollado múltiples protocolos y formas de comunicación que permiten obtener resultados óptimos.

El protocolo I^2C, desarrollado por Philips Semiconductor en 1982, permite la comunicación entre varios dispositivos utilizando una cantidad mínima de líneas para la transmisión de datos y sincronización entre dispositivos. Hoy en día, su principal aplicación se encuentra en sensores, pero también se utiliza en memorias EEPROM, conversores ADC/DAC y pantallas LCD.

El Laboratorio de Diseño de Circuitos Integrados (DCILab), adscrito a la Escuela de Ingeniería Electrónica del Instituto Tecnológico de Costa Rica, realiza el flujo completo de diseño VLSI (*Very Large Scale Integration*) enfocado en el desarrollo, simulación y verificación de circuitos integrados. Entre los logros alcanzados en el DCILab destacan sistemas como detectores de señales biomédicas (por ejemplo, ritmo cardíaco), circuitos de radiofrecuencia (RFID) y el diseño del primer microcontrolador de 32 bits basado en la arquitectura RISC-V desarrollado en Costa Rica, conocido como **Siwa**.

El microcontrolador **Siwa** tiene como objetivo su aplicación en sistemas médicos implantables [1]. Tomando esto en cuenta, su desarrollo se enfocó en lograr un bajo costo, bajo consumo de potencia y un

tamaño reducido. Las interfaces de comunicación implementadas por hardware en el sistema corresponden a ocho pines GPIO (*General Purpose Input/Output*), una interfaz UART (*Universal Asynchronous Receiver-Transmitter*) y una interfaz SPI (*Serial Peripheral Interface*).

El diseño del **Siwa** se enfocó en un tamaño reducido y en un sistema de baja potencia; debido a esto, la implementación por hardware especializado para utilizar el protocolo I^2C no fue posible.

1.2. Problema existente e importancia de la solución

El protocolo I^2C sigue siendo ampliamente utilizado en la actualidad en múltiples sistemas embebidos. Esta característica permite mantener compatibilidad con una gran variedad de dispositivos que lo implementan, facilitando la comunicación para la transferencia o recepción de datos.

Actualmente, el microcontrolador Siwa posee la capacidad de realizar envíos de datos mediante un algoritmo que emula el protocolo I^2C por software. [2] Sin embargo, no cuenta con la capacidad de recepción de datos a través del mismo, lo que representa una limitación importante para el sistema, ya que le impide actualizar o recibir información desde otros dispositivos utilizando este protocolo.

El proyecto propuesto busca actualizar el algoritmo para poder realizar recepción de datos utilizando el estándar para el protocolo I^2C.

1.3. Solución seleccionada y requisitos de diseño

En el sector de la tecnología, especialmente en áreas como la ingeniería electrónica y la informática, se ha vuelto común enfrentar la escasez o inaccesibilidad de hardware especializado. Esta limitación ha impulsado el desarrollo de soluciones creativas basadas en software, donde se aprovechan al máximo las capacidades de procesamiento de los dispositivos existentes para suplir, replicar o incluso mejorar las funciones originalmente delegadas al hardware.

Una de las técnicas más poderosas dentro de este enfoque es la emulación. La emulación consiste en replicar mediante software el comportamiento de una arquitectura de hardware específica. El software emulador actúa como una especie de interprete entre el código o funciones diseñadas para una plataforma específica y la arquitectura física del dispositivo que realmente se está utilizando. Esto permite, por ejemplo, ejecutar un sistema operativo o una aplicación diseñada para una consola de videojuegos antigua, un microcontrolador obsoleto, o incluso una computadora de arquitectura completamente distinta, en un sistema moderno sin necesidad de recurrir al hardware original.

El microcontrolador Siwa se basa en una arquitectura RISC-V, una ISA (Instruction Set Architecture) abierta y modular que ofrece múltiples ventajas a la hora de programar a bajo nivel. Entre sus beneficios destacan la simplicidad de su conjunto de instrucciones, la facilidad para generar código optimizado y su compatibilidad con herramientas modernas de compilación y depuración. Estas características hacen que implementar protocolos como I^2C en software sea más manejable y eficiente, incluso en sistemas con

recursos limitados.

Tomando estos factores en cuenta, la emulación de I^2C mediante software de bajo nivel se vuelve posible y presenta la solución al problema presentado.

1.3.1. Requisitos de Diseño

- El algoritmo implementado no puede tener un peso mayor a 8 KB, de acuerdo con la memoria principal disponible en el microcontrolador Siwa.
- Debe ser desarrollado en lenguaje ensamblador RISC-V, específicamente bajo el conjunto de instrucciones RV32I, para garantizar compatibilidad con la arquitectura del procesador.
- Debe trabajar al menos en una velocidad definida por el estándar I^2C, asegurando la interoperabilidad con dispositivos externos.
- La implementación debe ser completamente simulable en entornos de verificación como SystemVerilog y VCS.
- El algoritmo debe ser funcional en la versión física del microcontrolador Siwa.
- Se debe garantizar la validación del protocolo según el estándar I^2C [3], incluyendo condiciones límite y casos de error en el proceso de recepción de datos.

Capítulo 2

Marco Teórico

En este capítulo se presentan las bases teóricas necesarias para el desarrollo del proyecto. Los fundamentos abordados incluyen la importancia e implementación de protocolos de comunicación en sistemas electrónicos, el protocolo I^2C, la emulación por software, la arquitectura RISC-V y el microcontrolador SIWA.

2.1. Protocolos de comunicación en sistemas electrónicos

Todos los sistemas electrónicos requieren de un mecanismo eficiente para transmitir datos. En el dominio analógico, esta transmisión puede representarse mediante niveles de voltaje que corresponden a los estados lógicos: un nivel alto se interpreta como un “1” lógico y un nivel bajo como un “0” lógico. Con estas definiciones básicas se establece el puente entre la electrónica y la teoría de la información, permitiendo codificar datos binarios en señales físicas.

Sin embargo, no basta con definir únicamente los niveles de voltaje. Es necesario establecer parámetros adicionales, como la frecuencia de transmisión, que determina la velocidad a la cual se envían los bits, y el protocolo de comunicación, que especifica el significado de cada “1” o “0” transmitido en un contexto determinado. Del mismo modo, el orden en que se envían los bits (endianness, sincronización y estructura de tramas) influye directamente en la correcta interpretación de la información en el dispositivo receptor.

En este sentido, los sistemas digitales modernos se apoyan en protocolos de comunicación bien definidos, que regulan tanto la codificación de los datos como el flujo y la sincronización entre emisor y receptor, garantizando así una comunicación confiable y eficiente.

En el ámbito de los sistemas embebidos, existen múltiples protocolos de comunicación que se utilizan con mayor frecuencia para el intercambio de datos entre dispositivos. Entre los más comunes se encuentran UART (Universal Asynchronous Receiver-Transmitter), I^2C (Inter-Integrated Circuit), SPI (Serial Peripheral Interface), CAN (Controller Area Network) y USB (Universal Serial Bus).

Cada uno de estos protocolos presenta características particulares que los hacen más adecuados según el contexto de aplicación. Algunos se clasifican como síncronos, ya que requieren de una señal de reloj compartida para coordinar la transmisión y recepción de datos (como I^2C y SPI), mientras que otros son asíncronos, prescindiendo de esta señal (como UART). Otra diferencia fundamental radica en la cantidad de líneas necesarias para la comunicación: UART requiere dos, SPI puede requerir hasta cuatro o más dependiendo de la configuración, I^2C funciona con solo dos, mientras que USB y CAN tienen arquitecturas más complejas.

Finalmente, también se distinguen por su nivel de complejidad en la implementación. Protocolos como UART e I^2C suelen ser más sencillos de implementar en sistemas embebidos de recursos limitados, mientras que USB o CAN ofrecen mayores capacidades, pero con un costo en términos de hardware, consumo de energía y complejidad de software.

2.1.1. Protocolo I^2C

El protocolo I2C (Inter-Integrated Circuit) fue desarrollado por Philips en la década de 1980 y actualmente es utilizado en una gran variedad de dispositivos electrónicos como memorias EEPROM, pantallas LCD, sensores de temperatura y presión, entre otros.

Se basa en un esquema maestro–esclavo, donde el maestro controla la comunicación enviando la señal de reloj por la línea SCL y gestionando las transmisiones de datos a través de la línea SDA. Este mecanismo hace que el protocolo sea síncrono y altamente compatible con entornos de bajo consumo.

La Figura 2.1 muestra un diagrama típico donde se conectan las lineas SDA y SCL del maestro hacia los esclavos.

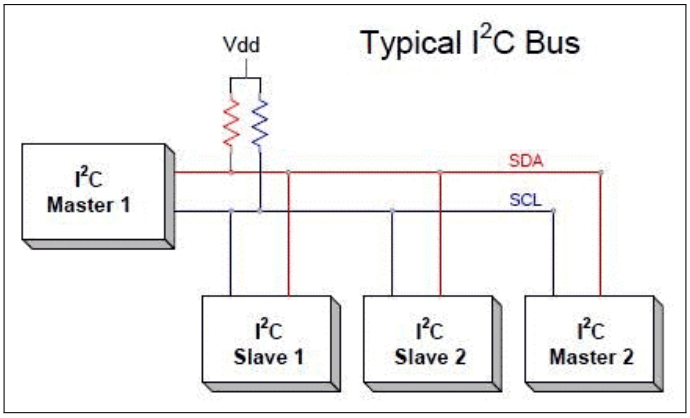


Figura 2.1. Bus de datos I^2C. *Tomado de [4].*

El estándar [3] define velocidades de transmisión de datos en los rangos de:

- Standard Mode: 100 kbits
- Fast Mode: 400 kbits

- Fast Mode Plus: 1 Mbits
- High Speed Mode: 3,4 Mbits
- Ultra Fast Mode: 5 Mbits

Para iniciar la comunicación entre el maestro y el esclavo en el protocolo I^2C, se realiza una transición de nivel alto a nivel bajo en la línea SDA mientras la línea SCL permanece en estado alto. Esta condición se conoce como *START* y tiene la función de indicar a todos los dispositivos esclavos conectados al bus que se dará inicio a una transmisión o recepción de datos.

De forma análoga, la finalización de la transmisión se establece mediante una transición de nivel bajo a nivel alto en la línea SDA, mientras la línea SCL se mantiene en estado alto. Esta condición se denomina *STOP* y señala que el maestro ha concluido la comunicación actual en el bus.

En la Figura 2.2 se ilustran gráficamente las condiciones de *START* y *STOP*, las cuales constituyen las señales fundamentales para la sincronización de la comunicación en I^2C.

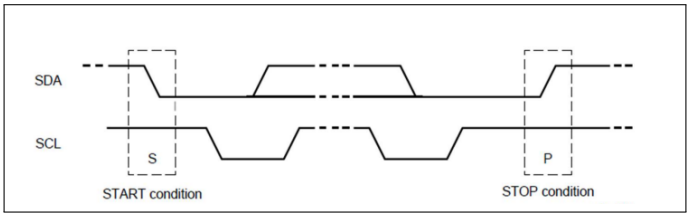


Figura 2.2. Condiciones de START y STOP en el protocolo I^2C. *Tomado de [4].*

Para la identificación de cada dispositivo esclavo conectado al bus, como se observa en la Figura 2.1, cada uno posee una dirección única de 7 bits. Esta dirección es transmitida inmediatamente después de la condición de *START* e indica al bus cuál de los dispositivos debe participar en la comunicación.

Adicionalmente, la dirección es seguida por un bit de control que determina la dirección de la transferencia: si el maestro realizará una transmisión de datos hacia el esclavo (*Write*) o, por el contrario, si el maestro recibirá datos desde el esclavo (*Read*). La Figura 2.3 ilustra este formato, mostrando la estructura de la dirección de 7 bits y el bit de control asociado.

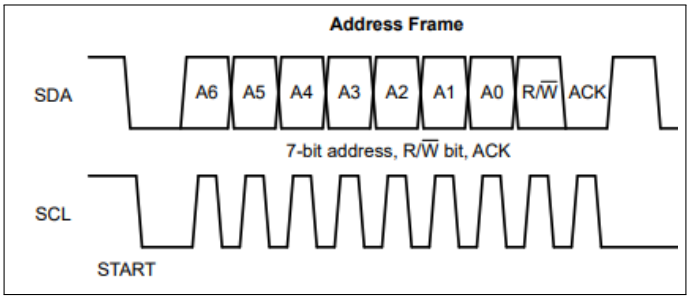


Figura 2.3. Estructura de la dirección de los esclavos junto con el bit de control y el bit de *ACK*. *Tomado de [3].*

Después de la fase de transmisión de la dirección del esclavo y del bit de control,el dispositivo esclavo responde mediante un bit de *ACK* (Acknowledgment) para confirmar que está listo para realizar la transmisión o recepción de datos. Este bit de confirmación es esencial, ya que una respuesta negativa interrumpe el proceso de comunicación y obliga al maestro a reiniciar la transmisión o a finalizarla por completo.

Además, el bit de *ACK* cumple una función importante durante la fase de transmisión de datos. Después de cada byte de información transmitido o recibido, el receptor envía un bit de confirmación que asegura la correcta recepción de los datos y permite continuar con la comunicación.

La fase de transmisión o recepción de datos es iniciada por el maestro una vez que ha recibido el bit de *ACK* por parte del esclavo. Durante esta etapa, la información se envía o recibe en bloques de 8 bits, equivalentes a un byte. La transmisión se realiza de manera que el primer bit corresponde al *MSB* (Most Significant Bit) y el último al *LSB* (Least Significant Bit), siguiendo el orden estándar del protocolo I^2C.

Después de la transmisión de cada byte, se envía un bit de *ACK* para confirmar la correcta recepción de la información. La responsabilidad de generar este bit depende de la dirección de la comunicación definida por el bit de control: si la operación es de escritura (*WRITE*), el bit de *ACK* lo genera el esclavo; si la operación es de lectura (*READ*), el bit de *ACK* lo genera el maestro.

La transmisión de datos no esta restringida en cuanto a bytes, es posible enviar todos los necesarios durante el proceso, manteniendo el proceso de respuesta con el bit de *ACK*.

La Figura 2.4 ilustra un flujo completo del protocolo desde la condición de START hasta su condición de STOP, durante el cual se realiza la transmisión de la dirección asignada, el bit de control, los bits de *ACK* correspondientes, y los bits de información.

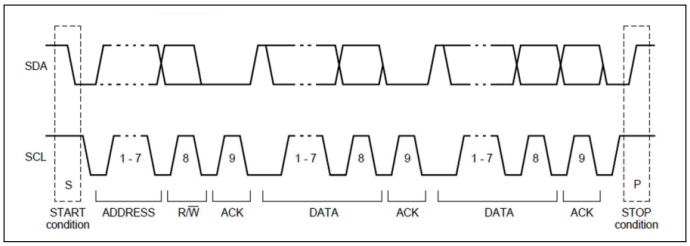


Figura 2.4. Flujo completo del protocolo. *Tomado de [4].*

2.2. Emulación de protocolos por software

La emulación por software es un proceso en el cual se emplea un programa para replicar las funciones de un componente de hardware. En el caso de los protocolos de comunicación, esta técnica permite recrear el comportamiento de un módulo especializado a partir del control directo de pines de propósito general (GPIO). Por ejemplo, es posible emular un bus SPI utilizando cuatro pines configurables como entradas o salidas, y siguiendo una rutina establecida por el software, logrando así reproducir las señales y la secuencia de comunicación propias de un periférico de hardware dedicado.

Si bien la emulación suele ser más lenta que un módulo físico, debido a la ejecución constante de instrucciones por parte del procesador, su principal ventaja radica en la flexibilidad que ofrece. Las rutinas implementadas pueden modificarse o adaptarse según los requerimientos de la aplicación, permitiendo ajustes directos en el programa sin necesidad de realizar cambios en la arquitectura del sistema.

Otro aspecto relevante es el costo. La implementación física de un periférico en hardware implica mayor consumo de área, energía y recursos económicos. En el diseño del microcontrolador *Siwa*, estos factores se han priorizado cuidadosamente para optimizar tamaño y eficiencia. Por lo tanto, la emulación por software representa una alternativa práctica y atractiva, ya que amplía la disponibilidad de protocolos de comunicación sin comprometer los objetivos de diseño, lo cual resulta particularmente ventajoso considerando la orientación de *Siwa* como microcontrolador de propósito general para aplicaciones médicas.

En el caso específico de *Siwa*, este no dispone de un bus I^2C por hardware que le permita comunicarse directamente con dispositivos externos que utilizan dicho protocolo, contando únicamente con un puerto SPI y un puerto UART.

Si bien el protocolo SPI comparte ciertas características con I^2C, su implementación en *Siwa* se encuentra limitada a un solo puerto, lo cual restringe la capacidad del sistema en aplicaciones donde se requiera el monitoreo simultáneo de múltiples dispositivos o la adquisición de datos en tiempo real.

2.3. Arquitectura RISCV

Una arquitectura de procesador define cómo éste interactúa con todos los elementos que lo componen, como memorias, registros, periféricos y demás. De esta forma, permite conocer qué instrucciones puede ejecutar, los tipos de registros disponibles y la forma en que se comunican los distintos componentes.

Para efectos prácticos, cuando se habla de microarquitectura se hace referencia al hardware que constituye el procesador o dispositivo electrónico en su diseño, es decir, a la implementación de elementos como registros, pipelines, unidades aritmético-lógicas (ALU), memorias, decodificadores, entre otros.

Por otro lado, la arquitectura del conjunto de instrucciones, comúnmente denominada *Instruction Set Architecture* (ISA), define el conjunto de instrucciones que el procesador puede ejecutar. Basándose en la microarquitectura implementada, las instrucciones son comandos que permiten realizar distintas operaciones utilizando el hardware disponible.

El conjunto de instrucciones correspondiente al proyecto es RISC-V, basado en los principios de RISC. Su diseño se centra en la simplicidad y la eficiencia.

RISC-V fue desarrollado en la Universidad de California, Berkeley, con el objetivo de crear una arquitectura abierta, libre de royalties y altamente escalable. A diferencia de otras ISAs comerciales, RISC-V permite a investigadores y fabricantes implementar procesadores sin restricciones de licencias, fomentando la innovación en sistemas embebidos, educación y procesadores de alto rendimiento.

Dentro de RISC-V, **RISCV32I** es el conjunto base obligatorio de instrucciones enteras de 32 bits. Su objetivo es proporcionar una plataforma mínima y funcional sobre la cual se pueden añadir extensiones opcionales.

Algunas de las características más importantes de este conjunto de instrucciones son:

- 32 registros de 32 bits cada uno.
- Instrucciones de 32 bits con formato fijo.
- Instrucciones aritméticas, lógicas, de carga/almacenamiento, saltos y control de flujo.
- Memoria direccionable byte a byte.

El conjunto base RISCV32I utiliza instrucciones de 32 bits con diferentes formatos, dependiendo de la operación a realizar:

- **Formato R:** operaciones entre registros.
- **Formato I:** operaciones con inmediatos y cargas.
- **Formato S:** almacenamiento en memoria.
- **Formato B:** saltos condicionales.
- **Formato U:** carga de valores grandes.
- **Formato J:** saltos incondicionales largos.

La Figura 2.5 muestra su manejo y codificación en lenguaje ensamblador.

32-bit RISC-V instruction formats

Format	Bit 31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- ***opcode* (7 bits):** Partially specifies one of the 6 types of *instruction formats*.
- ***funct7* (7 bits) and *funct3* (3 bits):** These two fields extend the *opcode* field to specify the operation to be performed.
- ***rs1* (5 bits) and *rs2* (5 bits):** Specify, by index, the first and second operand registers respectively (i.e., source registers).
- ***rd* (5 bits):** Specifies, by index, the destination register to which the computation result will be directed.

Figura 2.5. Tipos de Instrucciones y su manejo en RISC V.

2.4. Microcontrolador SIWA

2.5. Herramientas de diseño

La principal herramienta de diseño a utilizar será **Synopsys**, la cual proporciona un conjunto de soluciones orientadas a la simulación, verificación y síntesis de hardware descrito en lenguajes HDL. Dentro de este portafolio, la herramienta **VCS** (*Verilog Compiler Simulator*) será la de mayor relevancia para este proyecto, ya que constituye un simulador de alto rendimiento ampliamente utilizado en la industria para validar diseños digitales.

VCS permite compilar modelos escritos en *Verilog* y *SystemVerilog*, generando un ejecutable optimizado que simula el comportamiento del hardware descrito a nivel RTL. Adicionalmente, integra características como generación de cobertura funcional, manejo de aserciones (*SystemVerilog Assertions*), depuración avanzada mediante la herramienta *Verdi*, y soporte para bancos de pruebas (*testbenches*) modulares. Estas capacidades permiten no solo verificar la funcionalidad del procesador *Siwa*, sino también detectar condiciones límite, validar secuencias de comunicación complejas y garantizar la correcta implementación del protocolo I^2C en software.

Por otra parte, el **compilador RISC-V 32I** desempeña un papel fundamental en la integración entre hardware y software. Este compilador traduce programas escritos en lenguaje ensamblador o en C hacia instrucciones binarias específicas de la arquitectura RISC-V de 32 bits (conjunto base RV32I). El resultado de este proceso es un archivo ejecutable que contiene las instrucciones máquina necesarias para ser cargadas en la memoria del procesador.

Finalmente, se hará uso de **Python** como herramienta complementaria dentro del flujo de trabajo. Dado que el procesador *Siwa* es un desarrollo *in-house*, presenta un direccionamiento de memoria particular que difiere de los esquemas convencionales. Por esta razón, es necesario procesar y actualizar los archivos generados por el compilador RISC-V 32I, asegurando que las instrucciones cargadas a memoria correspondan correctamente al mapeo definido en el hardware. Esta flexibilidad que ofrece Python permite automatizar ajustes, generar scripts de verificación y garantizar la correcta integración entre el compilador y el modelo RTL.

Capítulo 3

Procedimiento metodológico

Capítulo 4

Ánalisis de resultados

Capítulo 5

Conclusiones y Recomendaciones

Bibliografía

[1] R. Garcia-Ramirez, A. Chacon-Rodriguez, R. Castro-Gonzalez, A. Arnaud, M. Miguez, J. Gak, R. Molina-Robles, G. Madrigal-Boza, M. Oviedo-Hernandez, E. Solera-Bolanos, D. Salazar-Sibaja, D. Sanchez-Jimenez, M. Fonseca-Rodriguez, J. Arrieta-Solorzano, and R. Rimolo-Donadio, "Siwa: a risc-v rv32i based micro-controller for implantable medical applications," in *2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*, 2020, pp. 1–4.

[2] R. Molina-Robles, R. García-Ramírez, A. Chacón-Rodríguez, R. Rimolo-Donadio, and A. Arnaud, "Low-level algorithm for a software-emulated i2c i/o module in general purpose risc-v based microcontrollers," in *2021 IEEE URUCON*, 2021, pp. 90–94.

[3] T. Instruments, "A Basic Guide to I2C," Texas Instruments, Tech. Rep. SBAA565, 2023, accessed: June 24, 2025. [Online]. Available: https://www.ti.com/lit/an/sbaa565/sbaa565.pdf

[4] A. Kwiatkowski, "Embedded systems development technology: Communication interfaces," Gdańsk University of Technology (GUT), Technical Report, 2015, ṪRSW CI: Communication interfaces.. Disponible en: https://metrologia.eti.pg.gda.pl/~TRSW/TRSW_CI.pdf.

Capítulo 6

Anexos

Anexo A: Extracción de *netlist* en *Custom Compiler*

Partiendo de un esquemático realizado en *Custom Compiler* como el que se observa en la Figura 6.1. Este corresponde a un comparador el cuál se le pretende extraer el *netlist* al realizar un análisis dc.

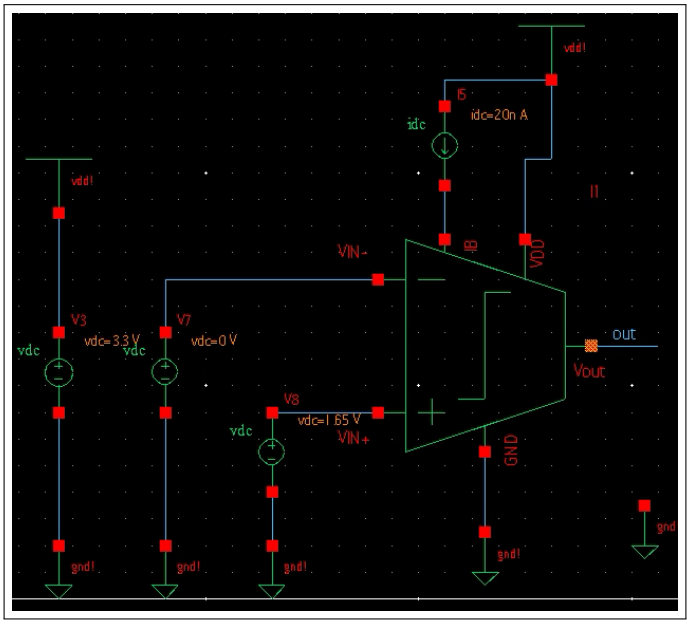
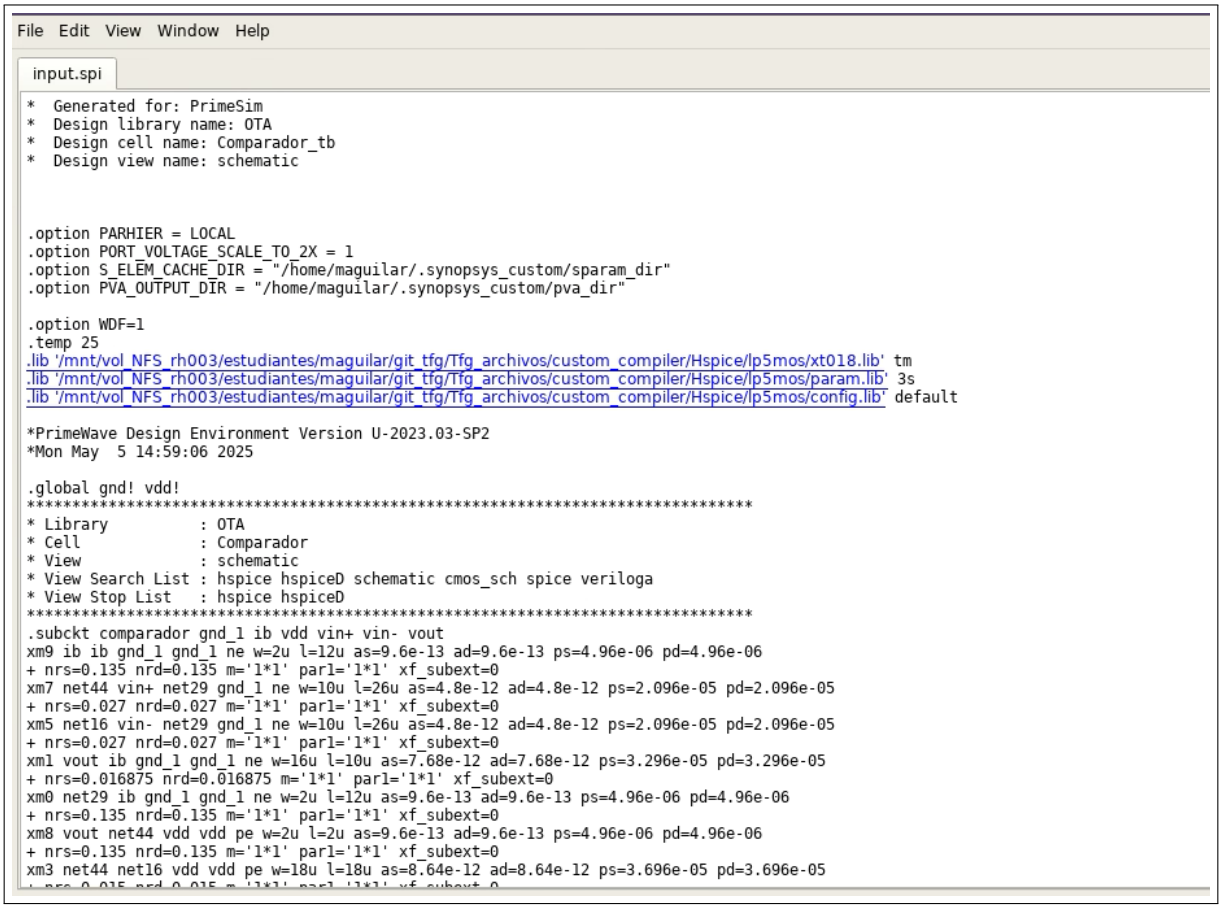


Figura 6.1. Esquemático del comparador. *Elaboración propia.*

En *PrimeWave*(este se encuentra en la barra superior en Tools), una vez se hayan configurado correctamente los tipos de simulación a realizar, es posible extraer el *netlist*. Para ello, en la barra superior seleccione la opción *Simulation*, luego *Netlist* y, finalmente, presione *Create* (Simulation → Netlist → Create). Esta acción mostrará en pantalla el *netlist* del circuito, como se observa en la Figura 6.2. En esta misma ventana, desde la barra superior, en la opción *File*, es posible guardar este archivo, el

cual corresponde al archivo.sp que se utiliza posteriormente en el algoritmo genético.



File Edit View Window Help

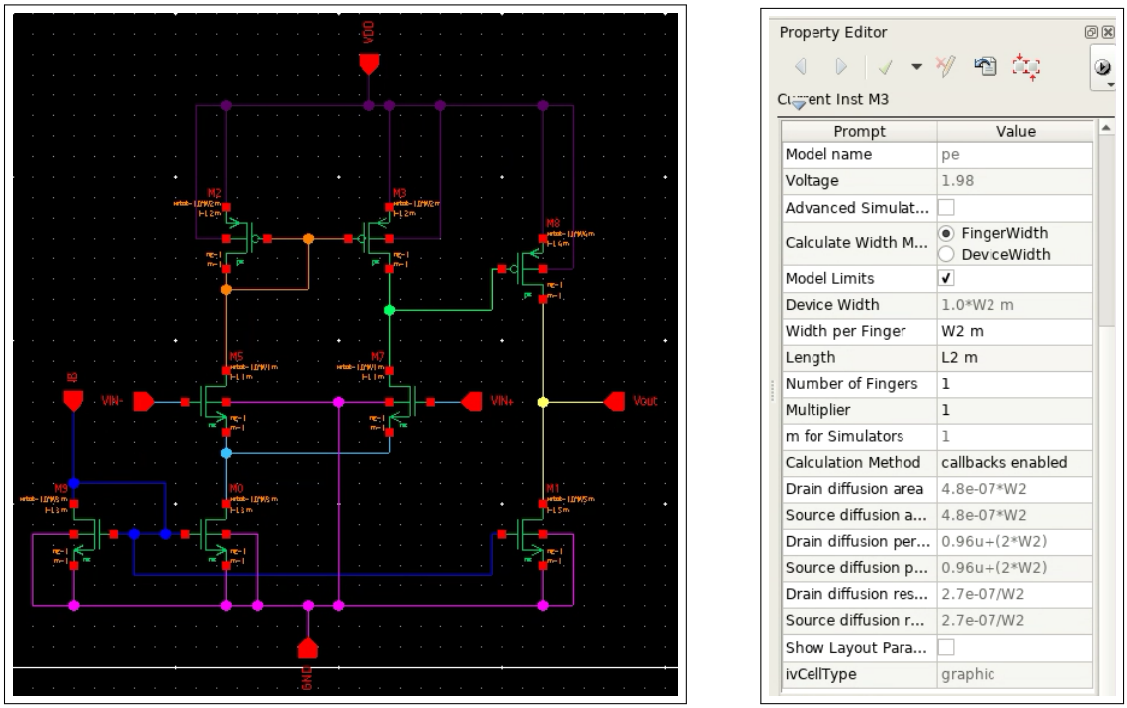
input.spi

```
*  Generated for: PrimeSim
*  Design library name: OTA
*  Design cell name: Comparador_tb
*  Design view name: schematic



.option PARHIER = LOCAL
.option PORT_VOLTAGE_SCALE_TO_2X = 1
.option S_ELEM_CACHE_DIR = "/home/maguilar/.synopsys_custom/sparam_dir"
.option PVA_OUTPUT_DIR = "/home/maguilar/.synopsys_custom/pva_dir"

.option WDF=1
.temp 25
.lib '/mnt/vol_NFS_rh003/estudiantes/maguilar/git_tfg/Tfg_archivos/custom_compiler/Hspice/lp5mos/xt018.lib' tm
.lib '/mnt/vol_NFS_rh003/estudiantes/maguilar/git_tfg/Tfg_archivos/custom_compiler/Hspice/lp5mos/param.lib' 3s
.lib '/mnt/vol_NFS_rh003/estudiantes/maguilar/git_tfg/Tfg_archivos/custom_compiler/Hspice/lp5mos/config.lib' default

*PrimeWave Design Environment Version U-2023.03-SP2
*Mon May  5 14:59:06 2025

.global gnd! vdd!
********************************************************************************
* Library          : OTA
* Cell             : Comparador
* View             : schematic
* View Search List : hspice hspiceD schematic cmos_sch spice veriloga
* View Stop List   : hspice hspiceD
********************************************************************************
.subckt comparador gnd_1 ib vdd vin+ vin- vout
xm9 ib ib gnd_1 gnd_1 ne w=2u l=12u as=9.6e-13 ad=9.6e-13 ps=4.96e-06 pd=4.96e-06
+ nrs=0.135 nrd=0.135 m='1*1' parl='1*1' xf_subext=0
xm7 net44 vin+ net29 gnd_1 ne w=10u l=26u as=4.8e-12 ad=4.8e-12 ps=2.096e-05 pd=2.096e-05
+ nrs=0.027 nrd=0.027 m='1*1' parl='1*1' xf_subext=0
xm5 net16 vin- net29 gnd_1 ne w=10u l=26u as=4.8e-12 ad=4.8e-12 ps=2.096e-05 pd=2.096e-05
+ nrs=0.027 nrd=0.027 m='1*1' parl='1*1' xf_subext=0
xm1 vout ib gnd_1 gnd_1 ne w=16u l=10u as=7.68e-12 ad=7.68e-12 ps=3.296e-05 pd=3.296e-05
+ nrs=0.016875 nrd=0.016875 m='1*1' parl='1*1' xf_subext=0
xm0 net29 ib gnd_1 gnd_1 ne w=2u l=12u as=9.6e-13 ad=9.6e-13 ps=4.96e-06 pd=4.96e-06
+ nrs=0.135 nrd=0.135 m='1*1' parl='1*1' xf_subext=0
xm8 vout net44 vdd vdd pe w=2u l=2u as=9.6e-13 ad=9.6e-13 ps=4.96e-06 pd=4.96e-06
+ nrs=0.135 nrd=0.135 m='1*1' parl='1*1' xf_subext=0
xm3 net44 net16 vdd vdd pe w=18u l=18u as=8.64e-12 ad=8.64e-12 ps=3.696e-05 pd=3.696e-05
```

Figura 6.2. *Netlist* del comparador. *Elaboración propia.*

Anexo B: Parametrización en *Synopsys*

En el esquemático de transistores del circuito para este ejemplo se empleará igualmente el comparador. Para parametrizar los circuitos, se debe hacer clic sobre cada transistor y presionar la tecla q, lo cual abrirá el *Property Editor*. En este editor, en los campos *Width per Finger* y *Length*, se debe ingresar el carácter correspondiente que será identificado como parámetro. En este ejemplo, se utilizará *Width per Finger*= $W2$ y *Length*= $L2$, como se muestra en la Figura 6.3. Este procedimiento debe repetirse para todos los elementos del circuito que se deseen parametrizar.



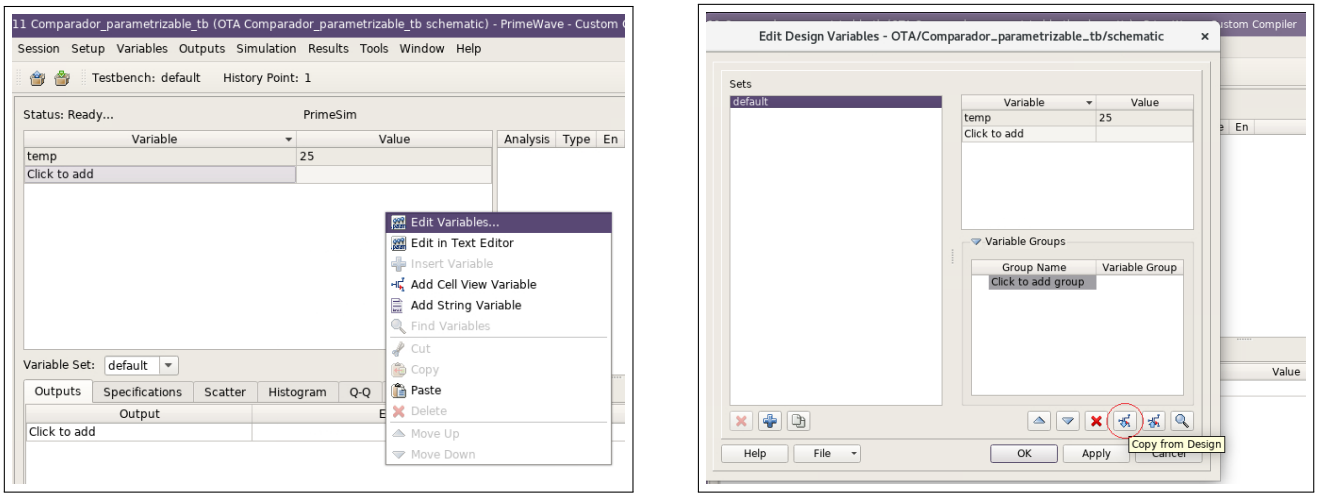
(a) (b)

Figura 6.3. Parametrización en *Synopsys* (a) Esquemático y (b) *Property Editor*. *Elaboración propia.*

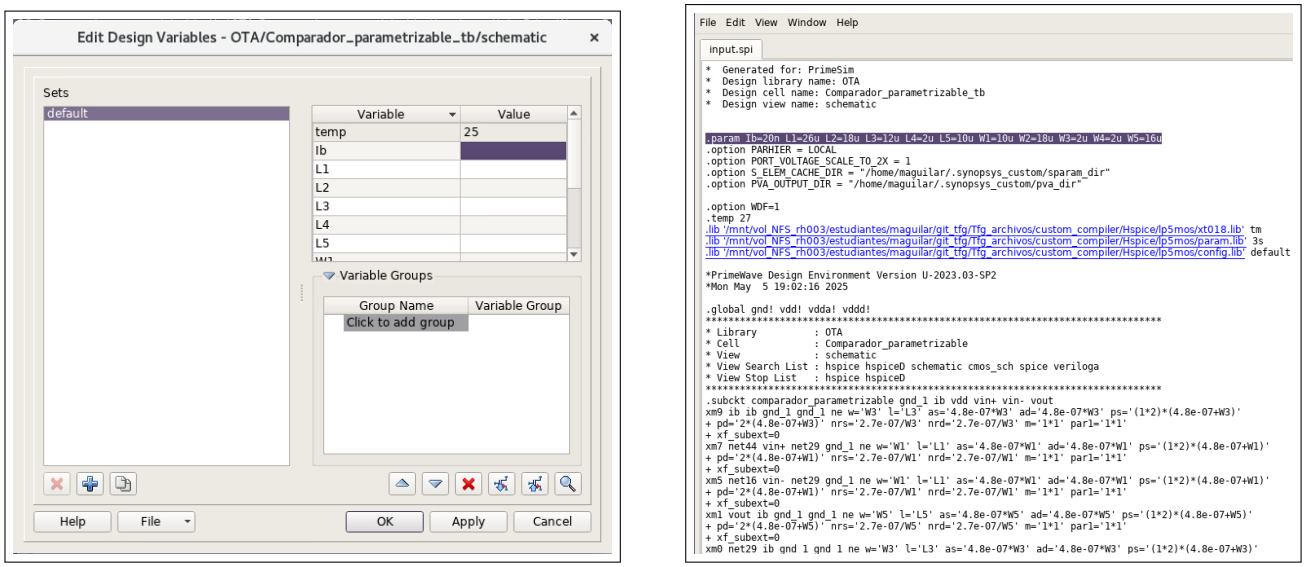
Una vez se han asignado todos los caracteres para la parametrización, el programa los detecta automáticamente. Al abrir *PrimeWave* para visualizar los parámetros definidos, ubíquese en el cuadro donde aparece la variable *temp*, haga clic derecho y seleccione la opción *Edit Variables* como se muestra en la Figura 6.4a. Se desplegará el menú mostrado en la Figura 6.4b. En este menú, al presionar el botón *Copy from Design*, se cargarán en el cuadro correspondiente (donde se encuentra la variable temp) todos los parámetros del circuito definidos previamente, como se observa en la Figura 6.4c. A cada uno de estos parámetros se le debe asignar un valor inicial para su posterior uso en el algoritmo genético. El procedimiento completo puede observarse en la Figura 6.4.

Posteriormente, se procede a extraer el *netlist*, tal como se describe en el Anexo 6.

Al generar el *netlist*, se podrá observar en el archivo una línea similar a la que se muestra en la Figura 6.4d, donde se listan todos los parámetros previamente definidos.



(a) (b)



(c) (d)

Figura 6.4. Pasos para la parametrización del circuito. *Elaboración propia.*

Anexos C: Configuración para la extracción de datos

Modificaciones al archivo de simulación .sp

Una vez se haya generado el netlist, es necesario realizar algunas modificaciones al archivo .sp extraído. En particular, las líneas mostradas en el Código 6.1 pueden ser eliminadas, ya que generan archivos que no son requeridos para la ejecución del algoritmo genético.

```
.option PARHIER = LOCAL
.option PORT_VOLTAGE_SCALE_TO_2X = 1
.option S_ELEM_CACHE_DIR = "/home/maguilar/.synopsys_custom/sparam_dir"
.option PVA_OUTPUT_DIR = "/home/maguilar/.synopsys_custom/pva_dir"
.option WDF=1
```

Código 6.1: Fragmento del .sp a eliminar para el uso con el algoritmo genético

A continuación, se debe agregar el Código 6.2. En este fragmento, la instrucción `.OPTION NUMDGT=6` establece que la salida numérica tendrá seis dígitos significativos, mientras que `.OPTION INGOLD=2` configura la salida para que los valores numéricos se presenten en notación científica. Esta última opción es indispensable para el correcto funcionamiento del algoritmo genético.

```
.option numdgt=6
.option ingold=2
```

Código 6.2: Fragmento que se debe agregar al archivo .sp

Del archivo .sp, es necesario generar un archivo de salida que contenga los resultados de todas las simulaciones que se desean ejecutar. Sin embargo, si se utiliza la instrucción `.probe`, los resultados solo se mostrarán en la consola y no se almacenarán en el archivo de salida. Por lo tanto, cada línea que contenga `.probe` debe ser reemplazada por `.print`. Una vez realizado este cambio, es posible eliminar las líneas `.probe`, incluidas aquellas con formato como `.probe dc v(*) level = 1`. En el Código 6.3 se muestra el fragmento original extraído, y en el Código 6.4 se presenta el código con las modificaciones correspondientes, este ejemplo corresponde al comparador que se quiere realizar un análisis *.dc*, *.ac* y *.tran*.

Dado que no se utilizarán las instrucciones `.probe`, las líneas `.option opfile=1 split_dp=1` y `.option probe=1` pueden ser eliminadas del archivo.sp, ya que no aportan funcionalidad relevante para el proceso de simulación ni para el algoritmo genético.

```
.option opfile=1 split_dp=1
.option probe=1
.probe dc v(*) level=1
.probe dc v(vout_dc)
.probe ac v(*) level=1
.probe ac v(vout_ac)
.probe tran v(*) level=1
.probe tran v(vin_tran) v(vout_trans) i(v19)
```

Código 6.3: Fragmento original extraído del archivo.sp

```
.print dc v(vout_dc)
.print ac v(vout_ac)
.print tran v(vin_tran) v(vout_trans) i(v19)
```

Código 6.4: Modificaciones realizadas al fragmento extraído

La última modificación que se debe realizar en el archivo .sp consiste en agregar una instrucción `.MEASURE` justo debajo de cada uno de los tipos de simulación que se deseen ejecutar. Esta inclusión es fundamental para que el script `extraer_datos` pueda realizar correctamente la extracción de resultados. En el Código 6.5 se muestran los tipos de simulación definidos inicialmente, mientras que en el Código 6.6 se presentan dichos bloques con las modificaciones correspondientes. Es importante destacar que, si se desea incluir algún otro tipo de simulación, también se deberá agregar la instrucción `.MEASURE` respectiva.

```
.dc v33 1.60 1.70 0.00005
.ac DEC 100 1 10meg
.tran 0.1m 4m start=0
```

Código 6.5: Fragmento del .sp correspondiente a los tipos de simulación a realizar

```
.dc v33 1.60 1.70 0.00005
.MEASURE Limi_dc FIND V(c26) AT= 1MEG
.ac DEC 100 1 10meg
.MEASURE Limi_ac FIND V(c26) AT= 1MEG
.tran 0.1m 4m 0 10e-6
.MEASURE Limi_tran  FIND v(vin_tran) AT=2ms
```

Código 6.6: Modificaciones realizadas para la correcta extracción de los datos

El elemento especificado en la instrucción `.MEASURE` puede corresponder a cualquier nodo o componente del circuito. La finalidad de incluir esta instrucción es que, en el archivo de resultados generado por la simulación, se inserte una línea identificable debajo de cada bloque de resultados correspondientes a los distintos tipos de análisis. Esto permite que el script `extraer_datos` pueda localizar y extraer adecuadamente la información requerida para el procesamiento posterior.

Implementación del script `extraer_datos`

Una vez realizadas todas las modificaciones al archivo de simulación *.sp*, se debe ejecutar el comando mostrado en el Código 6.7 desde la consola para iniciar la simulación. En este proceso, todos los resultados se almacenarán en el archivo de salida *.lis*, el cual podrá ser utilizado posteriormente para el análisis de datos.

```
hspice archivo.sp > archivo.lis
```

Código 6.7: Ejecución de HSPICE

En el archivo *.lis*, cada bloque de resultados cuenta con un encabezado que señala el tipo de simulación correspondiente, además de la instrucción `.MEASURE` que se agregó previamente.

En el Código 6.8 se muestra un fragmento del script *extraer_datos*, el cual requiere algunas modificaciones específicas:

- En la variable input_file se debe indicar el nombre del archivo .lis generado por la simulación, el cual contiene los datos a procesar.
- En cada llamada a la función ‘`extract_section`‘ se debe especificar el título exacto que identifica la simulación dentro del archivo *.lis*. Como se observa en el ejemplo, se extraen tres tipos de simulaciones, por lo tanto, se realizan tres llamadas a esta función.
- El tercer argumento de la función corresponde a la etiqueta definida mediante ‘`.MEASURE`‘ (en este caso: ‘limi_dc‘, ‘limi_ac‘ y ‘limi_tran‘), que permite delimitar el bloque de datos a extraer.
- Finalmente, se debe indicar el nombre del archivo de salida en el cual se almacenarán los datos correspondientes a cada simulación. Si se planea realizar más tipos de simulación, será necesario agregar nuevas llamadas a ‘`extract_section`‘ siguiendo el mismo esquema y definir las etiquetas respectivas mediante ‘`.MEASURE`‘.

```
def main_extraer():
input_file = "comparador.lis"
extract_section(input_file, "dc transfer curves", "limi_dc", "dc_data.txt")
extract_section(input_file, "ac analysis tnom", "limi_ac", "ac_data.txt")
extract_section(input_file, "transient analysis", "limi_tran",
   "tran_data.txt")
print("Extraccion completada.")
```

Código 6.8: Modificaciones al script para extracción de datos

Los archivos extraídos, debido al formato propio de *HSPICE*, contienen encabezados formados por dos renglones. La función `main_ordenar` se encarga de unificar adecuadamente estos encabezados en un único renglón, lo que facilita la posterior generación del archivo final. Para su uso, deben indicarse los

archivos generados previamente por la función `main_extraer`. En el Código 6.9 se puede observar como en `in_files` se agregan los archivos obtenidos del Código 6.8. En caso de que se incorporen otros tipos de simulación, es necesario agregar su correspondiente procesamiento dentro de esta función para garantizar su correcto funcionamiento.

```
def main_ordenar():
    # Archivos de entrada
    in_files = ["dc_data.txt", "ac_data.txt", "tran_data.txt"]
    # Archivos de salida
    out_files = ["dc_out.txt", "ac_out.txt", "tran_out.txt"]
```

Código 6.9: Función main_ordenar.

Los archivos generados por separado para cada tipo de simulación pueden contener múltiples secciones, cada una con distintas columnas de datos. Para facilitar su uso dentro del algoritmo genético, se utiliza la función `main_colum`, la cual une estas secciones en una única sección con todas sus columnas correspondientes. Para su correcto funcionamiento, únicamente se deben especificar los nombres de los archivos generados previamente por la función `main_ordenar`. En el Código 6.10 se puede observar como los datos del lado izquierdo corresponden a los archivos dados por `out_files` del Código 6.9. En caso de que se incorporen otros tipos de simulación, es necesario agregar el respectivo archivo para garantizar su correcto funcionamiento.

```
def main_colum():
    archivos = [
        ('dc_out.txt', 'dc_out_col.txt'),
        ('ac_out.txt', 'ac_out_col.txt'),
        ('tran_out.txt', 'tran_out_col.txt')
```

Código 6.10: Función main_colum.

La función `archivo_final` se encarga de unir todos los archivos previamente procesados en un único archivo. Además, agrega una columna adicional al extremo izquierdo denominada `iter`, la cual actúa como contador. Esta columna es útil para la implementación del algoritmo genético, ya que permite identificar cuántos datos corresponden a cada tipo de simulación. En el Código 6.11, se debe especificar en `archivo_salida` el nombre del archivo final que será utilizado por el algoritmo genético para la lectura de datos, y en `archivo_entrada` se deben listar los archivos generados por la función `main_colum`.

```
def archivo_final():
    archivo_salida = "datos_comparador.txt"
    archivos_entrada = ["dc_out_col.txt", "ac_out_col.txt",
   "tran_out_col.txt"]
```

Código 6.11: Función archivo_final.

Anexos D: Actualización del Makefile y regeneración de bibliotecas ltilib-pareto

Al realizar la copia de la carpeta denominada *Optimizacion*, se recomienda eliminar las bibliotecas previamente compiladas, ya que pueden generar errores de directorio al intentar acceder a archivos que no existen en la nueva ubicación. Para evitar estos errores, se deben seguir los siguientes pasos:

- Dirigirse a la dirección mostrada en el Código 6.12.

```
../Optimizacion/ltilib-pareto/linux
```

Código 6.12: Dirección del directorio

- Localice el archivo llamado `Makefile` y modifique la línea correspondiente a `LTIBASE`, colocando la ruta actual del directorio donde se encuentra la carpeta.
- Ejecute el comando `make clean-all`. Al revisar el contenido de la carpeta `lib`, se observará que las bibliotecas anteriores han sido eliminadas.
- Ejecute el comando `make` para generar nuevamente las bibliotecas necesarias.

Posteriormente, se recomienda continuar con los tutoriales presentados en [**?**] y [**?**].

Anexo E: Configuración *simInterface*

En el directorio donde se realizó la copia de la carpeta denominada *Optimizacion*, específicamente dentro de la subcarpeta `simInterface`, es posible visualizar todas las optimizaciones realizadas previamente. Para la implementación del flujo utilizando herramientas de *Synopsys*, se recomienda copiar los archivos contenidos en los directorios `compa_marco` o `OTA_2025` dentro de dicha carpeta, ya que estos incluyen los elementos necesarios para la ejecución, como el archivo `extraer_datos`.

Se va a tomar como ejemplo la dirección del Código 6.13.

```
/mnt/vol_NFS_rh003/estudiantes/maguilar/Optimizacion/simInterface/compa_marco
```

Código 6.13: Dirección del directorio ejemplo

Para el `simInterface` se necesitan hacer los siguientes cambios en los archivos:

Archivo `start`

Modificar el archivo `start` de la siguiente manera:

- Asegurarse que el comando `cd` apunte al directorio correspondiente dentro de la carpeta copiada. Para el ejemplo presentado, la modificación puede observarse en el Código 6.14.
- El comando `source` debe referenciar el script que inicializa las herramientas de HSPICE. La ruta específica dependerá de la ubicación del archivo de configuración en el sistema del usuario.
- Ejecutar el comando mostrado en el Código 6.7. En el caso particular del ejemplo, su ejecución se ve reflejada en el Código 6.14.
- El comando `tail` permite visualizar las últimas líneas del archivo generado por HSPICE (archivo `.lis`). Este paso es útil para verificar si la ejecución del archivo `.sp` se realizó correctamente. En caso de errores, será necesario corregirlos. Una vez se compruebe el correcto funcionamiento, esta línea puede comentarse.
- Finalmente, ejecutar el archivo `extraer_datos.py`. Si los anexos anteriores han sido realizados correctamente, se generará el archivo de salida requerido por el algoritmo genético.

```
cd /mnt/vol_NFS_rh003/estudiantes/maguilar/Optimizacion/simInterface/compa_marco/
source /mnt/vol_NFS_rh003/profesores/rmolina/Synopsys_Scripts/synopsys_tools.sh
hspice comparador.sp > comparador.lis
tail comparador.lis
python3 extraer_datos.py
```

Código 6.14: Ejecución de simulación y procesamiento de datos en HSPICE

Archivo `IOGenetic.cpp`

A continuación se enumeran las modificaciones que se debe realizar en este archivo:

- Definir los arreglos necesarios así como sus tamaños específicos para los que que requieren leerse del archivo de entrada este corresponde al archivo de salida de la función *archivo_final* del Código 6.11. El ejecutar ese código se implementa la columna `iter` esta como se explicó anteriormente corresponde a un contador, esto para que estos límites sean más sencillos de colocar, esto debe ser colocado en el Código 6.15.

```
const int quantity_samples_cd = 2001;  # de muestras en total para CD (1-2001)
const int quantity_samples_ac = 701;   # de muestras en total para AC
    (2002-2702)
const int quantity_samples_trans = 41; # de muestras en total para TRANS
    (2703-2743)
```

Código 6.15: Definición del número de muestras por tipo de simulación

- En las funciones `writeparameters` (primera línea del Código 6.16) y `Read_Parameter_Value` (segunda línea del mismo código), se debe especificar la ruta correspondiente al directorio donde se encuentra el archivo `.sp`. La tercera línea del Código 6.16 corresponde a la función `main()`. En esta función, se debe actualizar la variable `int comma` con la ruta del archivo `start`, y la variable `string filename` con el nombre del archivo de salida generado por la función `archivo_final` del script `extraer_datos.py`.

```
file.open("../Optimizacion/simInterface/compa_marco/comparador.sp", ios::out
    | ios::in );
fstream Parameter_File
    ("../Optimizacion/simInterface/compa_marco/comparador.sp", ios::in );
int comma = system("../Optimizacion/simInterface/compa_marco/start");
string filename = "datos_comparador.txt";
```

Código 6.16: Definición del número de muestras por tipo de simulación