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 noviembre 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
- $Setup$: Configuración o Preparación.
- $Address$: Dirección en bits utilizada en protocolo I^2C
- $VLSI$: Very Large Scale Integration.
- CPI:Ciclos por Instruccion.

Dedicatoria

Para mi tío Alex

Agradecimientos

A mi padre y madre por siempre confiar en mi, e impulsarme hasta el final, se los agradezco de todo corazón. A mis tías y abuelas que siempre me apoyaron y estuvieron presentes dentro de todo el proceso. A todos mis amigos de la universidad, por los cuales tengo miles de experiencias y muchísimo apoyo. A Carlos, Iran, Denisse y Brian por siempre estar presentes y enseñarme cosas nuevas durante esta experiencia. También deseo agradecer a mis profesores de la universidad, sin los cuales no habría logrado este desarrollo, el profesor Roberto Molina, por todo su apoyo durante este trabajo, y al profesor Jose Miguel Barboza por motivarme y ayudarme en mi paso por la carrera. Agradezco igualmente a todos aquellos que contribuyeron con mi aprendizaje y experiencias.

Resumen

El presente proyecto desarrolla un algoritmo en lenguaje ensamblador RISC-V destinado a emular, por software, el funcionamiento de un receptor basado en el protocolo I^2C dentro del microcontrolador Siwa. Dado que este dispositivo no cuenta con un módulo I^2C por hardware y únicamente dispone de interfaces UART, SPI y pines GPIO, se implementó una solución que permite interpretar señales I^2C mediante muestreo directo de los GPIO, siguiendo estrictamente las condiciones del estándar.

Para validar la propuesta se diseñó un ambiente de simulación en SystemVerilog, incorporando un *testbench* modificado capaz de emular un transmisor I^2C. Se evaluaron distintas frecuencias de operación, múltiples direcciones de dispositivo y diversos conjuntos de datos, confirmándose la correcta detección de las fases *START*, dirección, respuesta *ACK* y la lectura secuencial de bytes a 100 kHz. Asimismo, se identificaron limitaciones en frecuencias superiores a 400 kHz debido al retardo intrínseco asociado a la ejecución de instrucciones en el procesador.

Los resultados demuestran que la emulación por software constituye una solución viable y eficiente para extender las capacidades del microcontrolador Siwa sin requerir módulos de hardware adicionales. El proyecto sienta las bases para futuras mejoras, entre ellas la optimización temporal, la verificación automatizada y la validación con dispositivos físicos.

Palabras clave: RISC-V, Siwa, I^2C, verificación, SystemVerilog, ensamblador.

Abstract

This project presents the development of a RISC-V assembly algorithm designed to emulate, in software, an I^2C reception module within the Siwa microcontroller. Since Siwa lacks a dedicated I^2C hardware peripheral and relies solely on UART, SPI, and GPIO interfaces, the proposed solution enables accurate interpretation of I^2C signals by directly sampling the GPIO lines in compliance with the protocol standard.

A SystemVerilog simulation environment was implemented, including a modified testbench capable of emulating an I^2C transmitter. Several operating frequencies, device addresses, and data sets were tested to validate the algorithm's behavior. The system successfully detected START conditions, address decoding, ACK handling, and sequential byte reception at 100 kHz.

The results confirm that software-based emulation is a feasible and cost–effective approach to expanding the communication capabilities of the Siwa microcontroller without additional hardware.This work provides a foundation for further improvements such as timing optimization, automated verification, and validation with physical I^2C devices.

Keywords: RISC-V, Siwa microcontroller, I^2C emulation, verification, SystemVerilog.

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 microprocesadores, 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 microprocesador de 32 bits basado en la arquitectura RISC-V desarrollado en Costa Rica, conocido como **Siwa**. El microprocesador **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*).

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 microprocesador 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 microprocesador obsoleto, o incluso una computadora de arquitectura completamente distinta, en un sistema moderno sin necesidad de recurrir al hardware original.

El microprocesador 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. Objetivo General

- Implementar el protocolo I^2C en ensamblador RV32I mediante el uso de los pines GPIO del microcontrolador Siwa para la recepción de datos.

1.3.2. Objetivos Específicos

- El algoritmo implementado no puede tener un peso mayor a 8 KB, de acuerdo con la memoria principal disponible en el microprocesador 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.
- 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 microprocesador 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^{2}C 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^{2}C 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^{2}C 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^{2}C

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 transmisor–receptor, donde el transmisor 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 transmisor hacia los receptores.

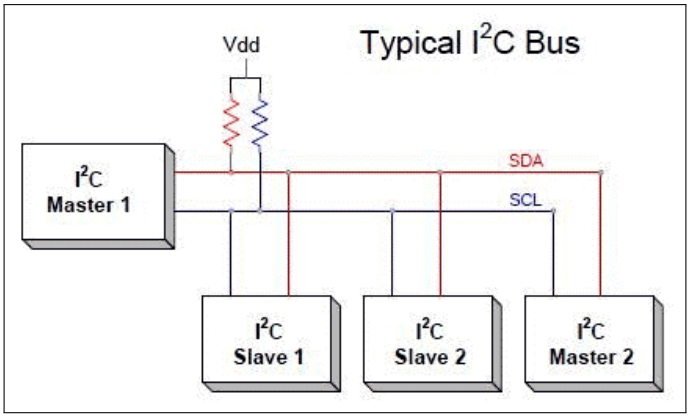


Figura 2.1. Bus de datos I^{2}C. *Tomado de [4].*

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 (Unicamente para transmisión)

Para iniciar la comunicación entre el transmisor y el receptor 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 transmisor 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.

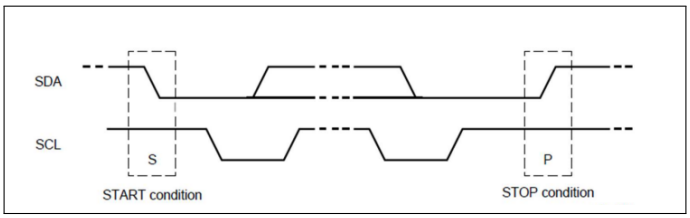


Figura 2.2. Condiciones de START y STOP en el protocolo I^2C. *Tomado de [4].*

Para la identificación de cada dispositivo receptor 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 transmisor realizará una transmisión de datos hacia el receptor (*Write*) o, por el contrario, si el transmisor recibirá datos desde el receptor (*Read*). La Figura 2.3 ilustra este formato, mostrando la estructura de la dirección de 7 bits y el bit de control asociado.

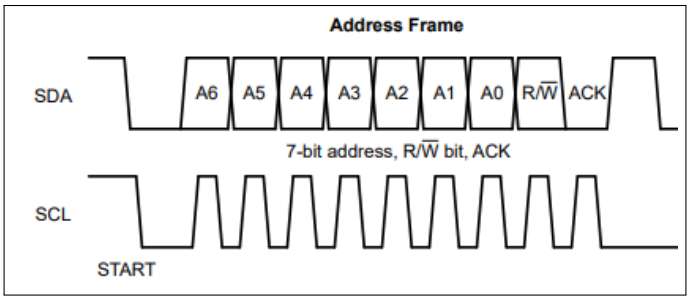


Figura 2.3. Estructura de la dirección de los receptores junto con el bit de control y el bit de *ACK*. *Tomado de [3].*

Después de la fase de transmisión de la dirección del receptor y del bit de control, el dispositivo receptor 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 transmisor 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 transmisor 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 receptor, si la operación es de lectura (*READ*), el bit de *ACK* lo genera el transmisor. 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.

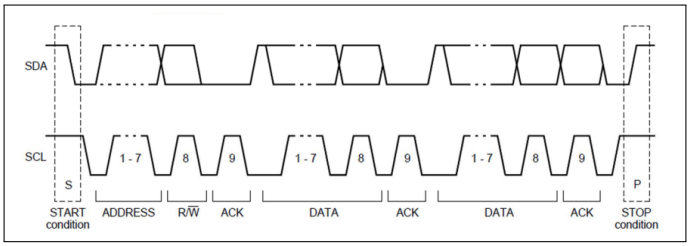


Figura 2.4. Flujo completo del protocolo. *Tomado de [4].*

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 microprocesador *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 microprocesador 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 [5]. 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.

Esta arquitectura emplea cuatro registros de propósito especial denominados *Control and Status Registers* (**CSR**), los cuales se subdividen en las siguientes categorías: `mtvec`, `mcause`, `mepc` y `mscratch`. El registro `mtvec` almacena la dirección del manejador de excepciones al que el procesador salta cuando ocurre una excepción. Durante este proceso, el procesador registra la causa de la excepción en el registro `mcause` y guarda el valor del contador de programa (*Program Counter, PC*) correspondiente a la instrucción que generó la excepción en el registro `mepc`.

Para la manipulación de los registros **CSR**, la arquitectura RISC-V ofrece tres instrucciones principales:

- `csrr`: lee el valor de un CSR y lo transfiere a un registro de propósito general.
- `csrw`: escribe el contenido de un registro de propósito general en un CSR.

- `csrrw`: realiza una operación combinada de lectura y escritura sobre un CSR.

Estas instrucciones permiten que el manejador de excepciones interactúe de forma eficiente con los registros **CSR**, transfiriendo valores entre registros especiales y registros de propósito general según sea necesario. Finalmente, una vez ejecutado el manejador de excepciones, la instrucción `mret` utiliza el valor almacenado en `mepc` para retornar al punto donde ocurrió la interrupción y continuar con la ejecución normal del programa.

2.4. Microarquitectura uniciclo y multiciclo

La microarquitectura describe la organización interna de un procesador, es decir, cómo se estructuran y conectan los registros, las unidades aritmético–lógicas (ALU), las máquinas de estados finitos (FSM), las memorias y los demás bloques lógicos necesarios para implementar una arquitectura. En una microarquitectura *uniciclo*, cada instrucción debe ejecutarse completamente dentro de un único ciclo de reloj. Todas las etapas —búsqueda de la instrucción, decodificación, ejecución y escritura del resultado— se realizan durante ese ciclo [5]. Aunque este enfoque presenta una unidad de control sencilla y un diseño fácil de comprender, la duración del ciclo debe ajustarse a la instrucción más lenta del conjunto, lo que limita el rendimiento general. Además, este tipo de procesador requiere memorias separadas para instrucciones y datos, una característica poco práctica en la mayoría de aplicaciones.

En contraste, la microarquitectura *multiciclo* divide la ejecución de cada instrucción en varios ciclos de reloj, procesándola de manera secuencial. Gracias a esto, las instrucciones simples requieren menos ciclos que las complejas, aprovechando mejor el tiempo de ejecución. Este tipo de organización permite reutilizar componentes costosos —como sumadores y memorias— en diferentes etapas del procesamiento; por ejemplo, una misma memoria puede emplearse inicialmente para la búsqueda de la instrucción y posteriormente para leer o escribir datos [5]. Esta reutilización reduce el área del chip y simplifica el diseño, por lo que los procesadores multiciclo son una opción habitual en sistemas de bajo costo.

2.5. Very Large Scale Integration (VLSI)

El término *Very Large Scale Integration* (VLSI) hace referencia al proceso mediante el cual es posible integrar decenas de miles hasta millones de transistores en un solo circuito integrado. En el contexto del diseño digital, VLSI engloba el conjunto de metodologías, herramientas y técnicas utilizadas para desarrollar hardware digital complejo, tales como microprocesadores, procesadores, sistemas embebidos y ASICs.

El flujo de diseño VLSI inicia con la especificación funcional del sistema y continúa con la descripción del hardware en un lenguaje HDL a nivel RTL (por ejemplo, SystemVerilog o VHDL). Posteriormente, se realiza la verificación funcional mediante simulaciones y testbenches para asegurar

que el comportamiento del diseño coincida con lo esperado. Una vez validado, el diseño se somete a síntesis lógica, donde el RTL se transforma en una red de puertas y celdas estándar optimizadas en área, desempeño y consumo. El siguiente paso consiste en el diseño físico, que incluye la colocación y ruteo de celdas dentro del chip, la construcción del árbol de reloj y el análisis de restricciones temporales, consumo y ruido. Finalmente, el diseño físico se valida mediante reglas de manufactura (DRC) y verificación eléctrica (LVS), previo a ser enviado a fabricación. La importancia del VLSI en el diseño digital radica en su capacidad para producir sistemas altamente eficientes, de bajo consumo y con un alto nivel de integración. Gracias a esta metodología, es posible desarrollar sistemas en chip (SoC) que combinan múltiples módulos digitales y analógicos en un único dispositivo, aumentando el desempeño y reduciendo los costos y el tamaño de las soluciones electrónicas modernas.

2.6. Lenguajes HDL

Un Lenguaje de Descripción de Hardware (HDL, por sus siglas en inglés) es un lenguaje formal utilizado para modelar, especificar y describir el comportamiento y la estructura de sistemas digitales. A diferencia de los lenguajes de programación tradicionales, los HDL permiten representar circuitos a distintos niveles de abstracción, desde puertas lógicas y operaciones combinacionales hasta arquitecturas completas de procesadores o sistemas digitales complejos [6, 7]. Los HDL permiten expresar tanto el comportamiento temporal como las relaciones lógicas entre señales, lo que facilita el diseño de elementos secuenciales, máquinas de estados, unidades aritmético–lógicas y componentes orientados al procesamiento digital. Entre los niveles de descripción más comunes se encuentran el nivel de transferencia entre registros (RTL), el nivel estructural y el nivel comportamental, cada uno proporcionando distintas capacidades para modelar hardware real [5]. Además, los HDL posibilitan la generación automática de hardware mediante herramientas de síntesis, que traducen las descripciones escritas en estos lenguajes a implementaciones físicas en FPGA o ASIC. Esto hace que los HDL sean fundamentales en el proceso de diseño de sistemas digitales modernos, permitiendo un flujo de trabajo ordenado, reproducible y escalable para la construcción de dispositivos electrónicos de alto desempeño [8].

SystemVerilog forma parte de la familia de los Lenguajes de Descripción de Hardware (HDL), utilizados para modelar y describir sistemas digitales a distintos niveles de abstracción. En este contexto, SystemVerilog surge como una extensión del lenguaje Verilog, ampliando sus capacidades para facilitar el diseño lógico y la descripción estructural de hardware moderno. Entre sus aportes más relevantes se encuentran la incorporación de tipos de datos más estrictos, el uso de estructuras empaquetadas y no empaquetadas, el empleo de bloques específicos como `always_comb`, `always_ff` y `always_latch`, así como el soporte nativo para modelar arquitecturas digitales completas en niveles comportamentales, estructurales y de transferencia entre registros (RTL) [5, 8]. Gracias a estas características, SystemVerilog permite describir de manera clara y eficiente módulos digitales tales como máquinas de estados, unidades aritmético–lógicas, decodificadores, controladores y subsistemas completos. Su sintaxis más expresiva y estricta reduce la probabilidad de errores en la inferencia de hardware y facilita la traducción posterior hacia implementaciones físicas en FPGA y

ASIC mediante herramientas de síntesis.

2.7. Verificación y Desarrollo de Testbench

La verificación constituye una de las etapas más importantes en cualquier proceso de diseño de sistemas digitales, ya que permite garantizar el correcto funcionamiento del circuito y detectar posibles fallos o comportamientos inesperados antes de su fabricación. En este contexto, la verificación formal puede llevarse a cabo mediante diversos métodos y enfoques, cada uno con características particulares y niveles de complejidad distintos. Un *testbench* corresponde a una estructura de código escrita en un lenguaje HDL, cuyo propósito es realizar pruebas, recopilar información y analizar el comportamiento del diseño dentro de un entorno controlado. Esta herramienta permite observar el flujo de trabajo del circuito digital, evaluar su funcionalidad bajo diferentes condiciones y detectar errores lógicos, comportamientos imprevistos o violaciones en los protocolos implementados.

A nivel formal, existen varios métodos de verificación ampliamente utilizados en la industria y la academia, entre los cuales se pueden mencionar:

- **Dirigidos (Directed Testing):** Se basa en la creación manual de casos de prueba específicos para ejercitar funciones concretas del diseño.
- **Basados en UVM (Universal Verification Methodology):** Utilizan una arquitectura modular, escalable y reutilizable que permite construir entornos de verificación complejos.
- **Aleatorios y Aleatorios Restringidos (Random / Constrained Random):** Generan patrones de estímulos no determinísticos, lo que incrementa la cobertura funcional y permite encontrar fallos no evidentes.
- **Self-checking:** Incorporan mecanismos automáticos para comparar resultados esperados y reales, reduciendo la intervención manual en la verificación.

La implementación de estos métodos puede realizarse en cualquier lenguaje HDL; sin embargo, para el procesador Siwa se emplea mayoritariamente SystemVerilog. Esto resulta especialmente beneficioso, ya que dicho lenguaje ofrece estructuras avanzadas para la verificación, una sintaxis más expresiva y alta compatibilidad con herramientas de simulación modernas, lo que facilita el desarrollo de entornos de prueba robustos y eficientes.

2.8. Microprocesador Siwa

El microprocesador **Siwa** es un *System-on-Chip* (SoC) personalizado de bajo consumo energético, orientado a aplicaciones portátiles e implantables [1]. Su núcleo de procesamiento se basa en la arquitectura libre **RISC-V**, implementando el conjunto de instrucciones **RV32I** bajo una

microarquitectura multiciclo no segmentada. En la Figura 2.6 se muestra el diagrama de alto nivel del sistema, el cual integra un controlador de interrupciones y diversas interfaces de comunicación estándar. Entre ellas se incluyen un *Universal Asynchronous Receiver-Transmitter* (**UART**), una *Serial Peripheral Interface* (**SPI**) y ocho puertos de entrada/salida de propósito general (**GPIO**). Estas interfaces permiten la interacción del microprocesador con periféricos externos y otros módulos digitales, garantizando flexibilidad y compatibilidad en aplicaciones de bajo consumo.

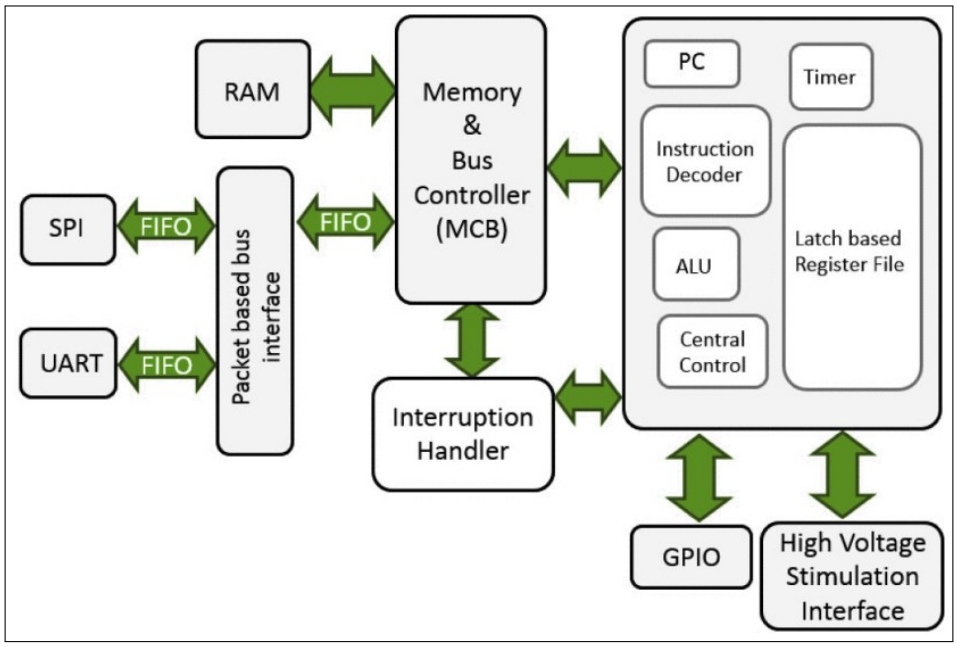


Figura 2.6. Diagrama de bloques a alto nivel de Siwa. *Tomado de [1].*

La comunicación con los pines **GPIO** se implementa a través de un esquema de *mapeo en memoria* (*memory-mapped I/O*), lo que permite acceder a cada puerto y realizar operaciones de lectura o escritura directamente mediante el mapa de memoria del procesador. El módulo encargado de gestionar estas transacciones es el **Memory and Bus Controller** (**MBC**), el cual coordina las solicitudes de acceso entre la *Unidad Central de Procesamiento* y los periféricos [9]. Cada puerto GPIO está asociado a una dirección específica, correspondiente a la región de memoria destinada a dispositivos de entrada/salida, como el **UART**, **SPI** y los propios **GPIO**. Mediante los *Control and Status Registers* (**CSR**) es posible habilitar o deshabilitar las funciones de los pines GPIO, así como configurar su modo de operación (entrada o salida). Además, estos registros permiten leer el valor actual presente en los pines.

2.9. Registros de Control y Estado

El microprocesador **Siwa** implementa un conjunto de *Control and Status Registers* (**CSR**) de acuerdo con el estándar de la arquitectura RISC-V. No obstante, presenta una particularidad respecto a las implementaciones convencionales: el registro `x0`, que en la arquitectura RISC-V clásica está

cableado permanentemente al valor cero, adquiere en Siwa una función distinta. En esta plataforma, el registro `x0` actúa como un registro de configuración especial, permitiendo la ejecución de ciertas pseudoinstrucciones y habilitando mecanismos adicionales de control dentro del sistema.

En la Figura 2.7 se detalla la distribución de los bits dentro del registro de configuración y su correspondencia con las distintas funcionalidades controladas, tales como la habilitación de periféricos, la gestión de interrupciones y la configuración de modos operativos internos.

Figura 2.7. Representación Grafica del registro X0, de configuración *Tomado de [10].*

La Tabla 2.1 muestra las direcciones de los registros CSR, mediante estas y su programación en el registro *x*0 es posible realizar la activación o manipulación de alguna de las funciones de Siwa.

Tabla 2.1. Direcciones de los Registros CSR

Register	**Implementation Dir.**
Registro configuración	0x00
mepc	0x01
Interrup1	0x02
Interrup2	0x03
GPIO	0x04
mvtec	0x05
comparación timer	0x06
Valor Timer	0x07
Full Range Level Shifter	0x08
IS_Valr	0x09
IS_Config	0x10
IS_Trigger	0x11

2.10. 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 microprocesador *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 personalizado, 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.

2.10.1. Acceso al servidor virtual y entorno de desarrollo

Para la utilización de las herramientas de diseño y verificación requeridas en el presente proyecto, se emplea el acceso remoto al servidor virtual de la Escuela de Ingeniería Electrónica. Dicho servidor proporciona un entorno controlado y homogéneo de trabajo, en el cual se encuentran instaladas las aplicaciones especializadas necesarias para el desarrollo y trabajo con el microprocesador *Siwa*, garantizando la compatibilidad de versiones, bibliotecas y licencias de software académico. Entre las herramientas disponibles se destaca **PuTTY**, una aplicación que permite establecer sesiones seguras de comunicación mediante el protocolo **SSH (Secure Shell)**. Este protocolo posibilita la conexión cifrada entre el equipo local y el servidor remoto, facilitando la ejecución de comandos y el acceso al entorno de desarrollo sin comprometer la seguridad de la información. Las herramientas disponibles para trabajar con *Siwa* requieren un entorno basado en sistemas operativos **Linux**, específicamente en una distribución **Red Hat Enterprise Linux (RHEL)**, debido a su estabilidad, compatibilidad con herramientas de diseño electrónico asistido por computadora (EDA, por sus siglas en inglés) y su amplio uso en entornos de desarrollo de sistemas embebidos y verificación de hardware. Una vez establecida la conexión mediante PuTTY, el usuario accede al entorno gráfico del servidor a través de la aplicación **VNC Viewer**, la cual implementa el protocolo *Virtual Network Computing*. Este protocolo permite la interacción visual con el escritorio remoto del sistema Red Hat, posibilitando el uso de interfaces gráficas de herramientas como *VCS* o entornos de desarrollo de verificación basados en *SystemVerilog* y *UVM*. Para acceder al servidor y realizar la conexión es posible referirse a este video. [11]

Capítulo 3

Procedimiento metodológico

Este capitulo detalla el procedimiento seguido para la implementación del proyecto, el cual contiene una parte de desarrollo en lenguaje System Verilog, y una parte en lenguaje ensamblador RISCV 32I, y su implementación con las herramientas provistas para la simulación del microprocesador Siwa.

3.1. Flujo de Operación de la Simulación y Desarrollo del Ambiente

El primer paso para trabajar con el microprocesador Siwa es acceder al servidor especializado que contiene las herramientas de simulación y verificación del microprocesador. Para ello, es necesario obtener los archivos correspondientes para la simulación completa del microprocesador Siwa. Estos archivos pueden ser obtenidos contactando a los profesores del DCILAB de la Escuela de Ingeniería Electrónica. Este flujo de operación se basa en [12] el cual permite crear archivos y correrlos. Una vez que se obtiene acceso a los archivos, es necesario entender el flujo de operación utilizado en la simulación del microprocesador Siwa. El entorno de simulación permite ejecutar un archivo de código en formato `.S`, escrito en lenguaje RISCV 32I, que funcionará como el programa que ejecutará la simulación de Siwa. El código presentado en el anexo 4 es un ejemplo de código `asm` escrito en lenguaje RISCV 32I.

El siguiente paso es la compilación del archivo `asm` hacia un formato de memoria compatible con Siwa. Debido a que el microprocesador tiene un desarrollo customizado, los archivos generados por el compilador básico de RISCV 32I no son necesariamente compatibles. Por esta razón, se implementó un script en Python, desarrollado en el DCILAB, que convierte los archivos de instrucciones de formato `.o` a un formato compatible con Siwa. El codigo presentado en el anexo 3 muestra el script de Python utilizado, el cual genera un archivo con formato `txt`. Este archivo `.txt` es necesario para el entorno de simulación en el que opera Siwa, y debe ser colocado en la carpeta donde se encuentran todos los archivos pertinentes a la simulación. Con los archivos adecuados, es posible realizar la simulación del microprocesador Siwa utilizando los componentes del procesador escritos en SystemVerilog. Estos

archivos pueden ser sintetizados utilizando simuladores como Vivado o VCS. Para realizar simulaciones en las que el procesador ejecute las instrucciones deseadas, es necesario incluir el archivo `.txt` generado por el script Python en la misma carpeta donde se encuentra el archivo `top.sv`. Con estos archivos configurados, se puede crear un archivo de *testbench* que ejecute diferentes tareas (`tasks`) y permita enviar datos al procesador y observar sus respuestas. La Figura 3.1 muestra un diagrama de bloques sobre el flujo de operación para realizar la simulación.

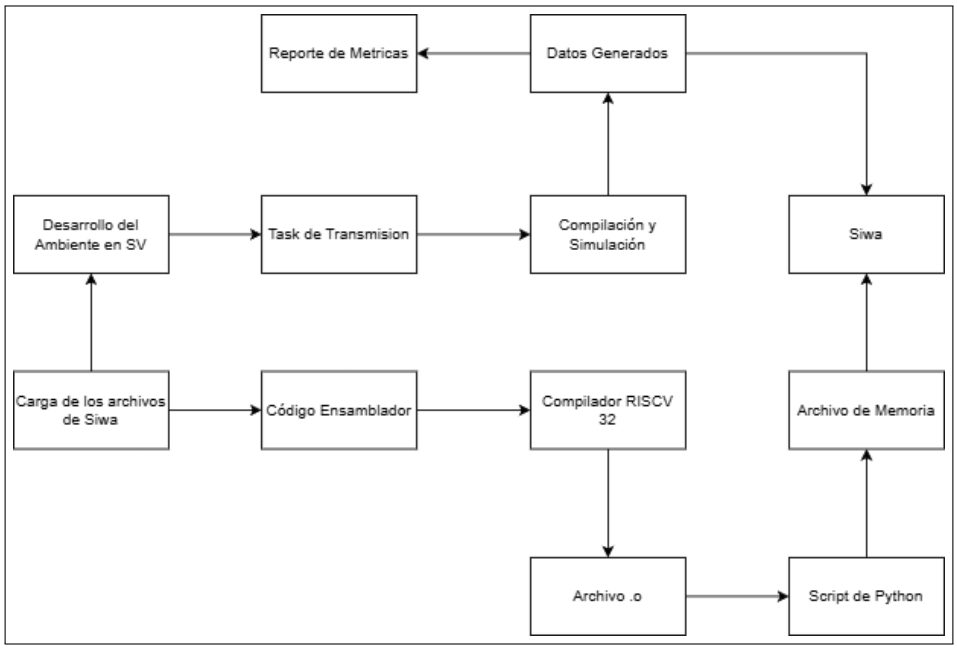


Figura 3.1. Flujo de operación para la simulación y ejecución del programa

3.2. Simulación del transmisor I^2C

El objetivo principal del proyecto se basa en evaluar la capacidad del microprocesador Siwa para funcionar como un receptor de datos utilizando el protocolo I^2C mediante el uso de sus puertos GPIO. Para validar la funcionalidad del código ensamblador que ejecutará Siwa, es necesario contar con un transmisor que envíe los datos correspondientes. Dado que el sistema físico no se encuentra disponible en esta etapa, se implementó una simulación de un transmisor utilizando un *testbench* desarrollado en SystemVerilog. El *testbench* empleado corresponde a uno previamente generado en el DCILAB, el cual se utiliza para realizar pruebas y métricas de diversos subsistemas de Siwa [9]. Además, este *testbench* incluye un conjunto de *tasks* diseñados para generar reportes de métricas internas del sistema. Entre la información obtenida se incluyen:

- estados internos de los registros,
- valores almacenados en dichos registros,

- instrucciones ejecutadas por el procesador,
- tiempos de ejecución de cada instrucción y evento según la simulación.

A pesar de que el *testbench* original realiza varias verificaciones de subsistemas, no hay una aplicación especifica para la manipulación de los puertos GPIO. Por lo tanto, fue necesario aplicar modificaciones puntuales para habilitar la simulación adecuada del transmisor I^2C. En primera instancia, se efectuaron pruebas sobre la señal de reloj con el fin de identificar los ajustes requeridos para garantizar que la línea SCL cumpliera con las especificaciones temporales establecidas por el estándar I^2C. El *testbench* opera con una señal de reloj base de 13 MHz, frecuencia que resulta suficiente para derivar las distintas velocidades admitidas por el protocolo. Considerando lo anterior, se diseñó un *task* específico cuyo propósito es controlar los puertos GPIO y automatizar la inserción de los bits necesarios durante la transmisión. Dicho *task* manipula la línea SCL conforme a los requerimientos temporales del estándar, permitiendo generar de manera precisa las frecuencias de operación correspondientes. Para la generación de estas frecuencias se adoptó una metodología basada en el uso de una función de retardo parametrizable, cuyo valor se calculó para cada una de las velocidades definidas por el estándar I^2C. Esta función se emplea conjuntamente con la instrucción `@posedge` del reloj base, lo que posibilita un control determinístico del tiempo de subida, muestreo y caída de la línea SCL. La combinación de ambos mecanismos permite reproducir las condiciones temporales necesarias para la correcta simulación del transmisor. La Figura 3.2 muestra el *setup* necesario dentro del entorno de simulación para la ejecución del transmisor y la correcta operación del *task* de transmisión I^2C.

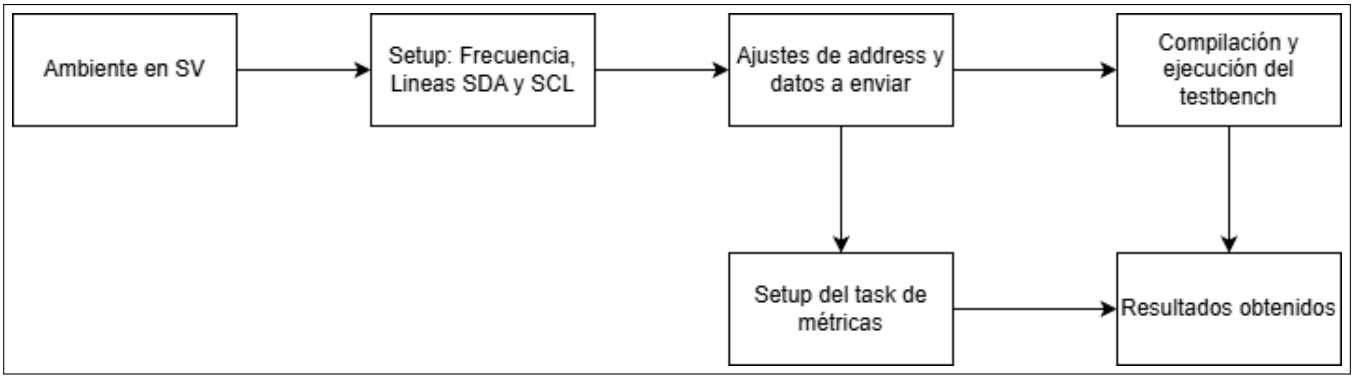
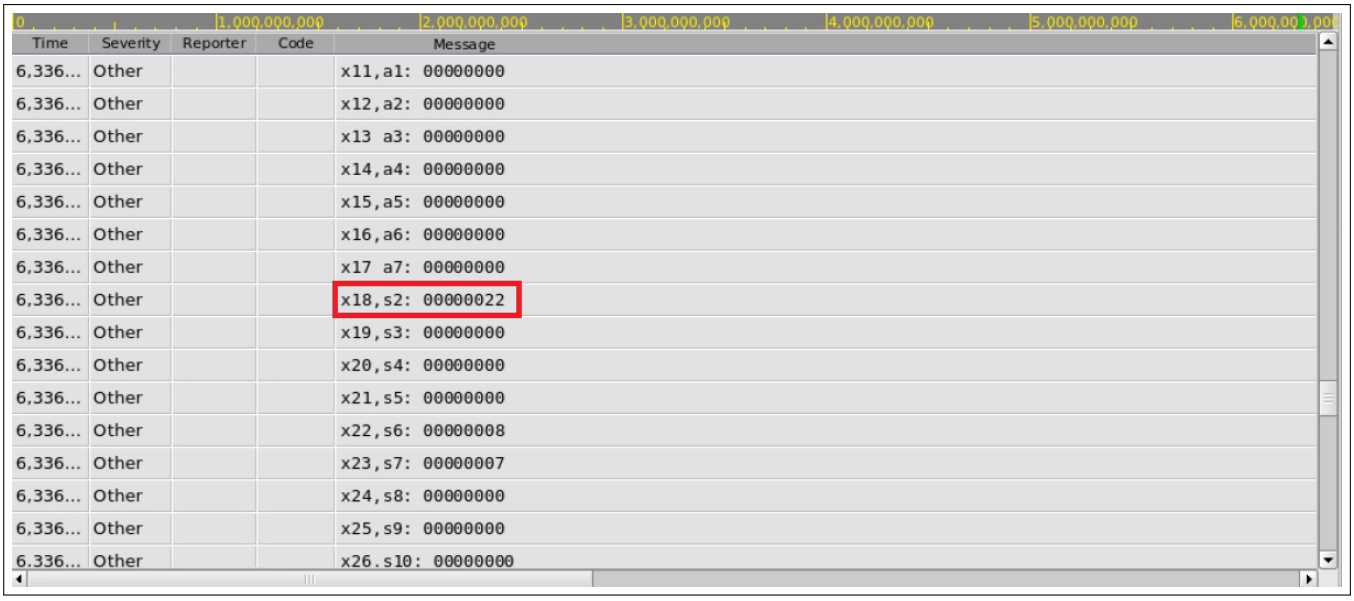


Figura 3.2. Diagrama de bloques para el ambiente en System Verilog

Estas herramientas permiten observar con precisión el comportamiento interno del procesador durante la recepción de datos mediante el protocolo I^2C. Con base en esta información es posible analizar los datos capturados, verificar su almacenamiento en los registros correspondientes y comparar los tiempos de operación. Todo esto facilita el proceso de depuración, permitiendo identificar errores o fallos en la recepción, sincronización o procesamiento de los datos transmitidos hacia Siwa.

La Figura 3.3 muestra como se observan el archivo de métricas desde Verdi, la interfaz gráfica utilizada para realizar la simulación del proyecto. Este archivo también se puede visualizar de forma

externa como un archivo de txt, pero resulta mas conveniente observarlo dentro de Verdi debido a que es posible sincronizar los tiempos con los tiempos de simulación y de esta forma analizar los datos.



Time	Severity	Reporter	Code	Message
6,336...	Other			x11,a1: 00000000
6,336...	Other			x12,a2: 00000000
6,336...	Other			x13 a3: 00000000
6,336...	Other			x14,a4: 00000000
6,336...	Other			x15,a5: 00000000
6,336...	Other			x16,a6: 00000000
6,336...	Other			x17 a7: 00000000
6,336...	Other			x18,s2: 00000022
6,336...	Other			x19,s3: 00000000
6,336...	Other			x20,s4: 00000000
6,336...	Other			x21,s5: 00000000
6,336...	Other			x22,s6: 00000008
6,336...	Other			x23,s7: 00000007
6,336...	Other			x24,s8: 00000000
6,336...	Other			x25,s9: 00000000
6,336...	Other			x26,s10: 00000000

Figura 3.3. Reporte observado desde la interfaz de Verdi

La verificación del sistema se llevó a cabo mediante el entorno *Verdi*, herramienta que permite la observación detallada de las señales en tiempo real y el análisis de su comportamiento durante la simulación. Esta metodología facilita la obtención de resultados precisos, así como la identificación de los instantes exactos en los que se producen los eventos relevantes. La correcta operación del *task* de transmisión depende de diversos factores, entre ellos la configuración del hardware asociada a los GPIO. En la Figura 3.4 se presenta la arquitectura correspondiente.

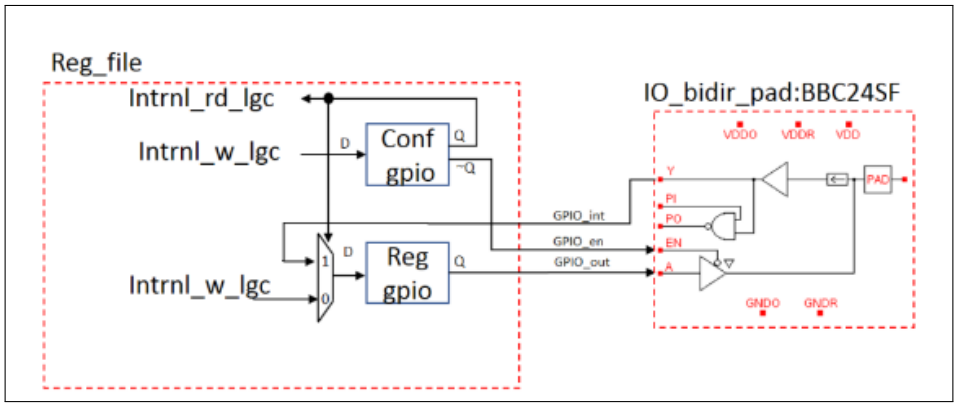


Figura 3.4. Diagrama de la arquitectura de los GPIOS de Siwa *Tomado de [10].*

Esta arquitectura es fundamental, ya que determina las señales disponibles durante la simulación. Las líneas denominadas como *GPIO_int* representan los datos provenientes del entorno externo hacia

el procesador; a través de estas líneas se realiza la transmisión simulada en el *testbench*. Por su parte, las señales *GPIO_en* definen el modo de operación de los puertos GPIO y se configuran mediante el registro CSR ubicado en la dirección `x0`. Esta configuración se realiza desde el programa en lenguaje ensamblador, y es ajena al task ejecutado en System Verilog. Dichas señales pueden tomar los valores "0" o "1": un valor de "0" coloca la línea en modo lectura, mientras que un valor de "1" habilita su operación en modo escritura por parte del puerto externo. Finalmente, las señales *GPIO_out* corresponden a las salidas generadas por los puertos GPIO hacia el *pad* externo, completando así el flujo de información entre el entorno simulado y la arquitectura del procesador.

El código presentado en el anexo 2 muestra un código de prueba utilizado para observar el funcionamiento de los GPIO, particularmente en el envío de datos desde la simulación.

En este código se configuran en alto los GPIO 2 y 3 durante un tiempo determinado y posteriormente se llevan a nivel bajo, lo cual simula el envío de un bit durante un intervalo de tiempo definido. Los resultados de esta prueba se muestran en la Figura 3.5.

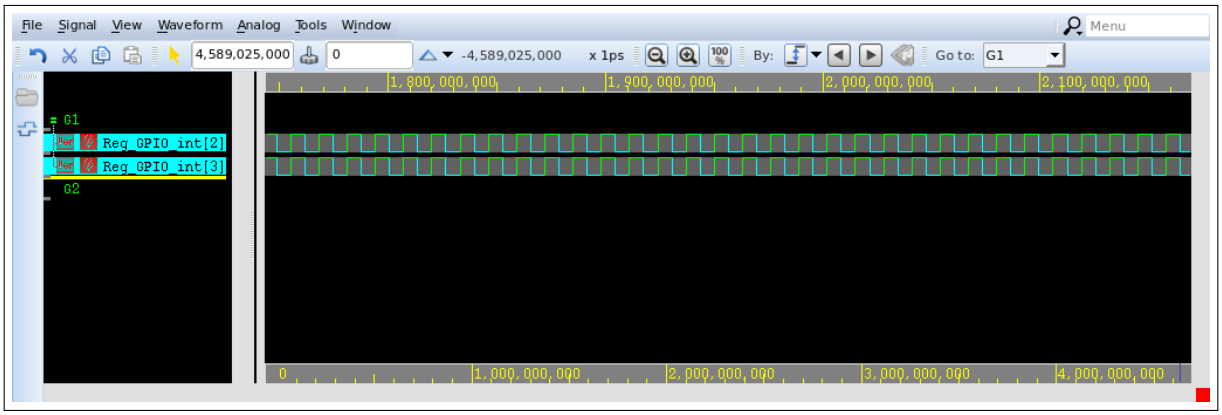


Figura 3.5. Prueba de manejo de GPIOs para envio de datos

Usando esta misma lógica es posible implementar un transmisor que utilice el protocolo I^2C para enviar datos al procesador a través de sus líneas GPIO. Para el desarrollo del proyecto se emplean las líneas GPIO 2 y GPIO 3 para simular las líneas SDA y SCL, respectivamente. El resultado del *task* implementado que simula el transmisor se presenta en la Figura 3.6. Dicho *task* hace uso de variables estáticas que permiten modificar la frecuencia de trabajo de la línea SCL, así como seleccionar qué GPIO se utilizarán como SCL y SDA.

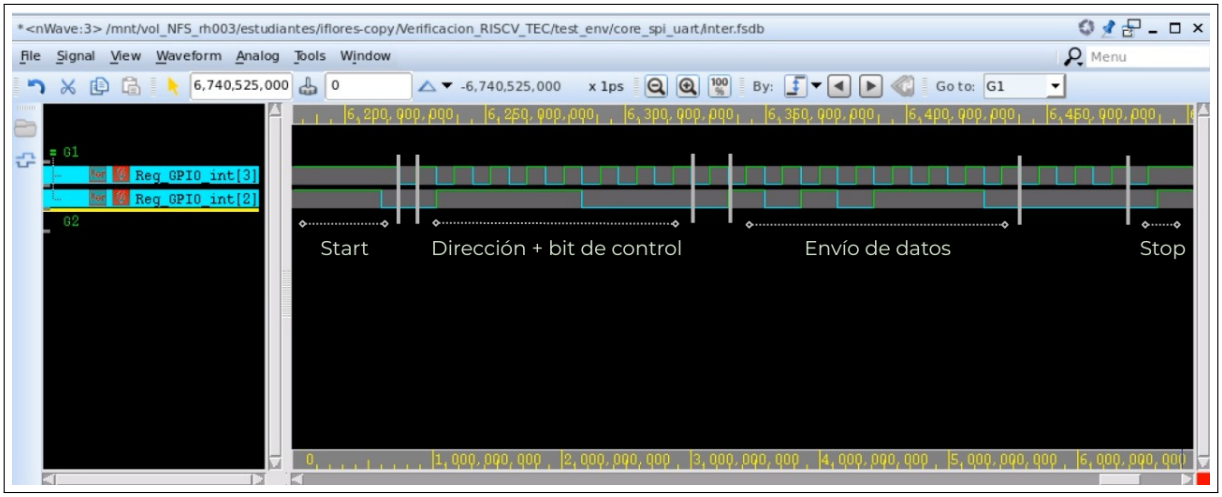


Figura 3.6. *Task* de transmisión I^2C.

En la Figura 3.6 se pueden identificar todos los procesos necesarios para la simulación de un transmisor I^2C, desde la condición de *START* hasta la condición de *STOP*. Para las pruebas realizadas se ejecutó la transmisión de un solo byte de datos; no obstante, este valor es ajustable en el *task* para permitir el envío de más datos según los requerimientos de la aplicación.

3.3. Lógica del programa a implementar en ensamblador

El objetivo del programa en ensamblador es realizar la lectura correcta de los datos enviados por medio del *testbench*, utilizando el estándar I^2C para la transmisión e interpretación de la información.

Para el desarrollo del programa se emplean tres instrucciones fundamentales para obtener los valores presentes en los GPIO: `csrrs`, `andi` y `beq`/`bne`. En conjunto, estas instrucciones permiten:

1. Leer el estado actual de todos los GPIO desde el registro CSR correspondiente.
2. Aislar el bit asociado a un GPIO específico mediante una máscara.
3. Tomar decisiones de control (*branch*) según el valor lógico de dicho bit.

Lectura de los GPIO con `csrrs`

En RISC-V, la instrucción `csrrs` tiene el formato:

```
csrrs rd, csr, rs1
```

donde:

- `rd` es el registro destino donde se almacena el valor leído del CSR.

- `csr` es la dirección del registro de estado y control (*Control and Status Register*).
- `rs1` se utiliza, opcionalmente, para hacer una operación OR sobre el CSR. Cuando se utiliza `x0` en `rs1`, no se modifica el contenido del CSR y la instrucción funciona como una simple lectura.

En el contexto de este programa, el acceso a los GPIO se realiza mediante un CSR dedicado a dichas líneas. Un ejemplo típico de lectura podría representarse como:

```
csrrs t1, 4, x0
```

De esta forma, el registro `t1` contiene el estado instantáneo de todos los GPIO en forma de vector de bits.

Uso de `andi` como máscara de bits

Una vez que el valor de los GPIO se encuentra almacenado en `t1`, es necesario extraer el estado de un GPIO específico. Para ello se utiliza la instrucción `andi`, cuyo formato es:

```
andi rd, rs1, imm
```

donde `imm` es un valor inmediato que actúa como máscara de bits.

En este programa, cada GPIO se selecciona mediante una máscara distinta:

- Para leer el valor de GPIO3 se utiliza la máscara `0x08`
- Para leer el valor de GPIO2 se utiliza la máscara `0x04`.

Ejemplo lógico de código:

```
csrrs t1, 4, x0          # t1 <- todos los GPIO
andi  t2, t1, 0x08       # t2 <- solo el bit de GPIO3 (SCL)
andi  t3, t1, 0x04       # t3 <- solo el bit de GPIO2 (SDA)
```

Después de estas operaciones:

- `t2` será `0x08` si `GPIO3` está en “1” y `0x00` si está en “0”.
- `t3` será `0x04` si `GPIO2` está en “1” y `0x00` si está en “0”.

Lógicamente, `andi` se interpreta como: “conserva solo el bit de interés y fuerza el resto a cero”, lo que transforma un vector de múltiples GPIO en una variable booleana codificada como cero o distinto de cero.

Decisiones de control con `beq/bne`

Una vez aislado el bit correspondiente a la línea de interés, se pueden tomar decisiones de control mediante las instrucciones de salto condicional `beq` (branch if equal) y `bne` (branch if not equal). En particular, se utiliza con frecuencia la variante `beqz` (*branch if equal to zero*), que compara el registro con cero.

Por ejemplo, para tomar una decisión según el estado de la línea SCL (`GPIO3`):

```
csrrs t1, CSR_GPIO, x0    # Leer todos los GPIO
andi  t2, t1, 0x08        # Aislar GPIO3 (SCL)
beqz  t2, etiqueta
```

Desde el punto de vista lógico:

- `beqz t2, etiqueta` significa "si el bit muestreado está en cero, saltar a `etiqueta`".
- `bnez t2, etiqueta` significaría "si el bit vale uno (distinto de cero), saltar a `etiqueta`".

En el algoritmo desarrollado, esta lógica se aplica principalmente sobre la línea SCL (GPIO3), dado que el estándar I^2C establece que:

> El receptor muestrea los datos durante el pulso en alto de SCL, mientras que los cambios de datos en SDA deben ocurrir cuando SCL está en bajo. [3]

Esto se traduce en el código como una estructura de sondeo continuo de SCL: el programa permanece en un bucle leyendo el CSR y efectuando `andi`+`beq`/`bne` hasta detectar que SCL ha tomado el nivel lógico requerido (alto para muestrear, bajo para cambiar datos).

Guardado de los datos entrantes en la linea SDA

El muestreo de los datos presentes en la línea SDA sigue un proceso similar al descrito previamente para la lectura de la línea SCL. Sin embargo, en este caso no es necesario realizar una instrucción de salto condicional, ya que el flujo del algoritmo se limita a extraer el bit recibido y construir progresivamente el byte completo en un registro acumulador. El proceso puede dividirse en cuatro etapas lógicas fundamentales:

1. **Aislamiento del bit en SDA.** Primero se lee el CSR correspondiente a los GPIO mediante la instrucción `csrrs`. Luego, se aplica una máscara con `andi` utilizando el valor inmediato `0x4`, que corresponde al bit asociado a GPIO2 (SDA). De esta forma se elimina cualquier otro bit del registro, conservando únicamente el valor leído en la línea SDA:

```
csrrs t2, 4, x0
andi  t2, t2, 0x4
```

2. **Normalización del bit leído.** Debido a que el bit aislado podría encontrarse en una posición distinta a la menos significativa (LSB), se utiliza la instrucción `srli` para desplazarlo hacia la posición cero del registro. Esto convierte el valor leído en un '0' o '1' lógico estrictamente equivalente:

```
srli t2, t2, 2
```

3. **Preparación del registro acumulador.** Antes de insertar el nuevo bit, se desplaza el contenido del registro acumulador (`s2`) una posición hacia la izquierda. Este desplazamiento garantiza que el espacio para el nuevo bit quede libre y que los bits previamente almacenados se mantengan en su orden correspondiente:

```
slli s2, s2, 1
```

4. **Inserción del bit en el registro acumulador.** Finalmente, se introduce el bit recién leído mediante la instrucción `or`, que copia el valor de `t2` (0 o 1) en la posición menos significativa del registro acumulador sin modificar los bits previamente desplazados:

```
or   s2, s2, t2
```

La combinación de estas operaciones permite reconstruir un byte completo recibido por la línea SDA, respetando el orden de llegada de los bits según el protocolo I^2C, y sin riesgo de sobreescritura accidental.

El algoritmo presenta una limitación inherente al hecho de que el procesador sólo puede observar valores lógicos “1” o “0” en los registros, esta restricción dificulta la detección precisa de las condiciones de *START* y *STOP* del protocolo I^2C. En el caso de la condición de *START*, el protocolo define que ésta ocurre cuando la línea SDA realiza una transición de alto a bajo mientras la línea SCL se encuentra en nivel alto. Implementar esta detección de manera directa requeriría muestrear ambas líneas repetidamente y comparar cada lectura con la anterior para identificar un flanco descendente, lo cual implicaría un número considerable de ciclos de instrucción y una pérdida significativa de eficiencia. Para evitar este sobrecosto, se adopta una estrategia alternativa basada en el estado de reposo del bus. Durante la condición de espera, tanto SDA como SCL permanecen en un nivel lógico “1” de forma estable. Aprovechando esta característica, el algoritmo realiza el muestreo únicamente sobre la línea SDA: mientras SDA permanezca en “1”, el procesador continúa muestreando; cuando se detecta un cambio a “0”, se asume que se ha producido una condición de *START* y se procede a la etapa de muestreo de la dirección. Esta solución evita la necesidad de almacenar valores previos y reduce significativamente la cantidad de instrucciones requeridas.

La condición de *STOP* presenta un problema similar. El protocolo establece que SDA debe realizar una transición de bajo a alto mientras SCL está en nivel alto. Detectar este evento requeriría nuevamente múltiples muestreos y comparaciones, lo cual no resulta factible desde el punto de vista del desempeño. Para resolver esta limitación, se implementó un mecanismo de control basado en la cantidad de bits de reconocimiento (ACK) que deben recibirse durante una transmisión. Conforme al estándar, el maestro debe recibir al menos dos bits de ACK por parte del receptor: uno correspondiente a la dirección y otro al primer byte de datos (mínimo un byte transmitido). A partir de esta propiedad, es posible definir una condición de finalización para el algoritmo. Durante la configuración inicial se especifica cuántos bytes serán enviados por el maestro; con esta información, el procesador puede calcular cuántas secuencias de ACK debe esperar. De esta manera, el control del flujo queda determinado por la suma del bit de ACK de la dirección más el número de bytes a recibir, eliminando la necesidad de detectar directamente la condición de *STOP* y permitiendo completar la transmisión cuando el número esperado de ACK haya sido procesado correctamente.

Tomando las consideraciones y las instrucciones presentadas, es posible implementar un programa que sigue la secuencia descrita en el diagrama de flujo mostrado en la Figura 3.7. A partir de dicha implementación, se realiza el muestreo de datos conforme al protocolo I^2C, configurando al microprocesador para operar como un receptor y permitiendo la correcta interpretación de la información recibida a través de las líneas GPIO. El programa completo desarrollado en ensamblador puede ser encontrado en el anexo 1.

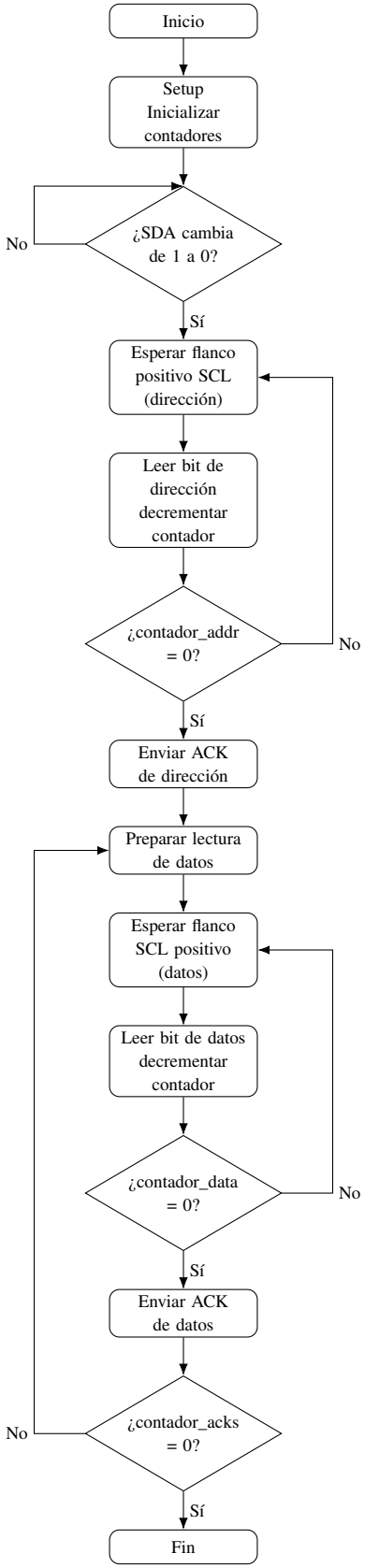


Figura 3.7. Diagrama de flujo del algoritmo de recepción I^2C.

Capítulo 4

Ánalisis de resultados

Esta sección presenta los resultados obtenidos durante la simulación del algoritmo, los cuales abarcan pruebas realizadas con distintas frecuencias en la línea SCL, pruebas utilizando diversas direcciones de dispositivo y ensayos con diferentes conjuntos de datos. El objetivo de estas evaluaciones es verificar la integridad del funcionamiento del algoritmo y analizar los resultados obtenidos durante su implementación. La Figura 4.1 muestra el proceso implementado con las pruebas, así como la verificación del algoritmo. Esta estrategia permitió corregir errores así como verificar adecuadamente la integridad con diferentes sets de datos.

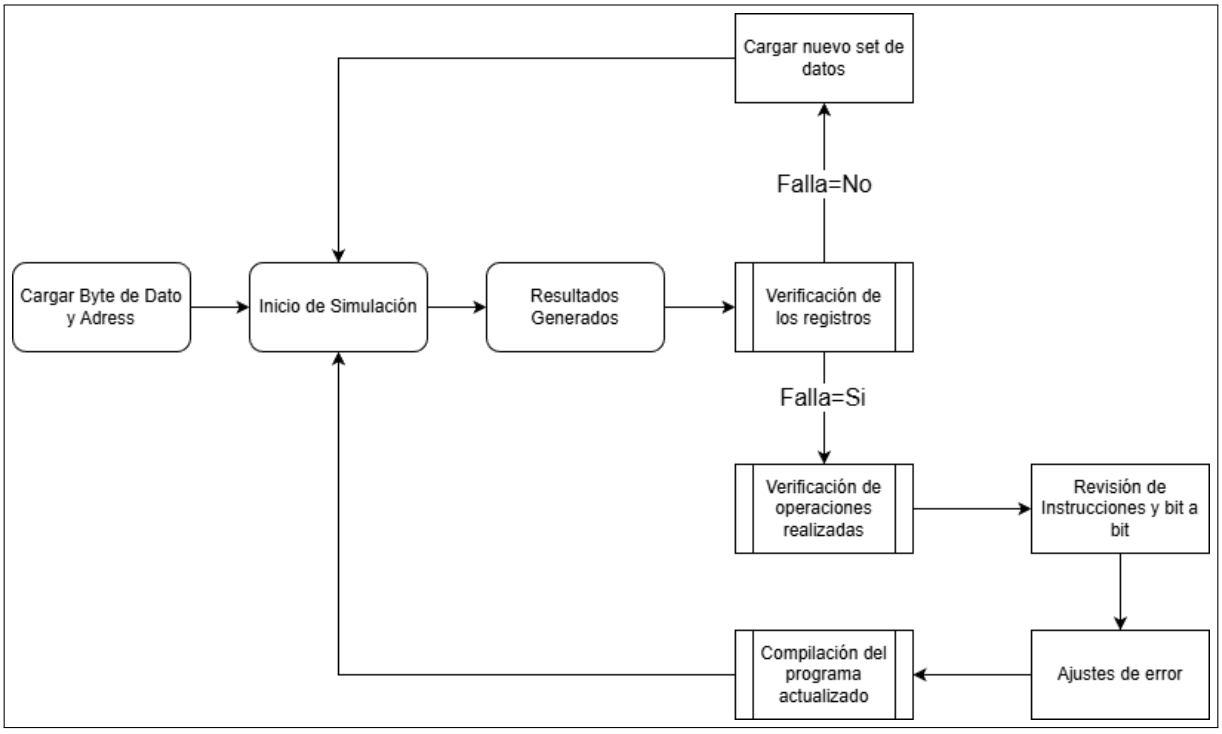


Figura 4.1. Diagrama de bloques del flujo de trabajo sobre las pruebas.

4.1. Resultados a diferentes frecuencias de operación

Como parte fundamental de las pruebas, resulta indispensable determinar hasta qué punto el algoritmo es capaz de manejar el muestreo de las líneas SCL y SDA de manera adecuada. Dado que cada instrucción ejecutada por el procesador introduce un retardo asociado, es necesario evaluar si es posible mantener una sincronización correcta entre los muestreos y, simultáneamente, obtener datos precisos que permitan validar el funcionamiento del receptor.

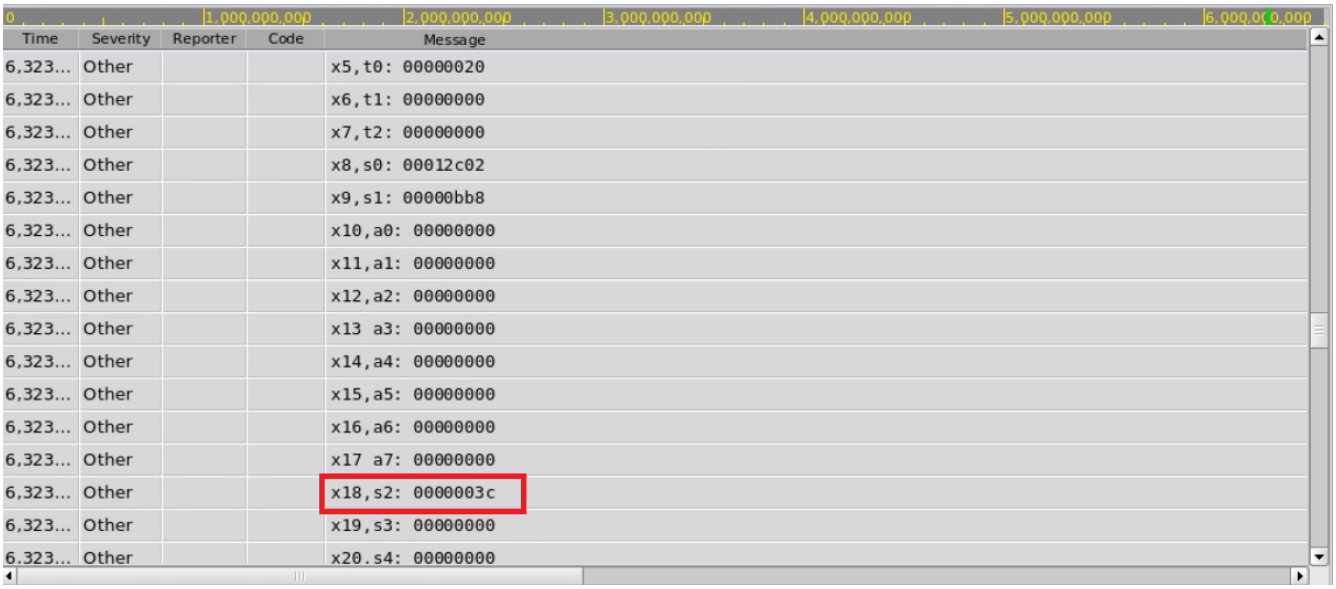
Para las pruebas realizadas a diferentes frecuencias, se centró la atención principalmente en los bits que componen el campo de dirección (*address*). Al tratarse del primer byte transmitido durante una comunicación I^2C, representa un punto crítico para identificar fallos tempranos en la capacidad del controlador de muestrear correctamente las señales. Un error en esta etapa indica típicamente que la frecuencia es demasiado alta para ser procesada con la secuencia actual de instrucciones, o que el algoritmo requiere optimizaciones adicionales. La dirección utilizada para todas las pruebas de frecuencia corresponde al valor hexadecimal `0x3C`, el cual en binario se representa como `0111100`. Este valor se guarda en el registro `S2` y se analizará en el momento en que finalice el envío de la dirección.

En la Figura 4.2a se pueden observar los valores almacenados en los registros después de la ejecución del programa, así como el tiempo de ejecución, que coincide con el tiempo mostrado en la Figura 4.2b. El cursor fue colocado en la trama correspondiente al inicio del bit de ACK, habiendo pasado previamente por las fases de *START* y *address*. En particular, el registro `S2` contiene la dirección recibida y se puede observar que concuerda con los parámetros establecidos. Por otro lado, el registro `S3` contiene un contador que decrece con cada bit agregado a la dirección, el cual se observa en cero una vez completada la recepción de los 8 bits de la dirección.

Para la prueba a 400 kHz es necesario tener en cuenta que el periodo de tiempo es menor, dado que se utiliza una frecuencia mayor en la línea SCL. Los resultados obtenidos muestran que el algoritmo no es capaz de realizar el muestreo de manera adecuada. Esto se puede observar en la Figura 4.3a en el registro `S2`, donde la dirección final registrada es `0x02`, y el contador en el registro `S3` muestra el valor de 6. Al comparar este resultado con la Figura 4.3b, se puede observar que, para un tiempo de 6.23 s, ya se han completado las fases de *START* y *address*.

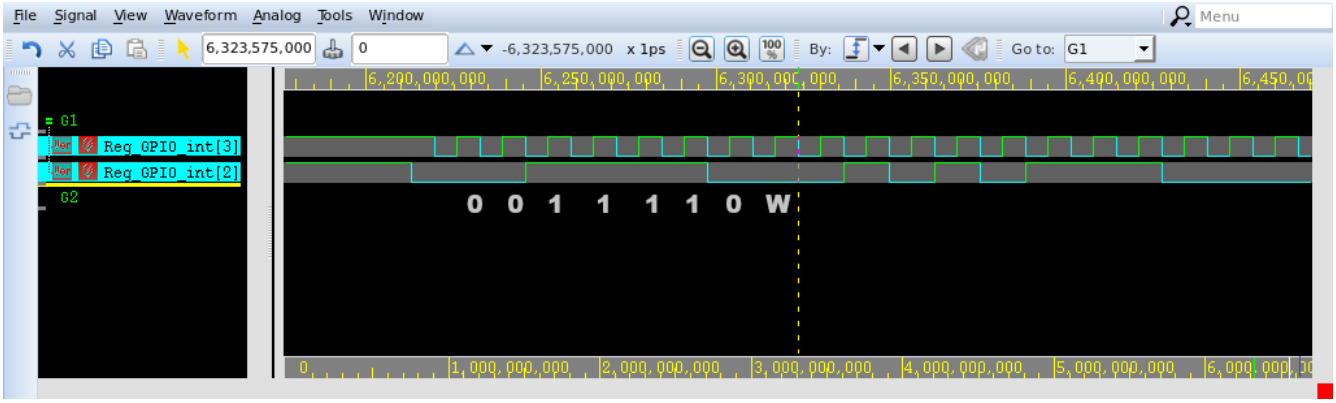
Este comportamiento es crucial para conocer los límites del algoritmo en cuanto al muestreo de frecuencias en una línea GPIO. Las limitaciones observadas indican que, para frecuencias superiores a 100 kHz, los resultados no serán correctos, lo que establece un límite práctico para la implementación del algoritmo en sistemas que operan a frecuencias mayores.

4.1.1. Pruebas a 100 kHz



Time	Severity	Reporter	Code	Message
6,323...	Other			x5,t0: 00000020
6,323...	Other			x6,t1: 00000000
6,323...	Other			x7,t2: 00000000
6,323...	Other			x8,s0: 00012c02
6,323...	Other			x9,s1: 00000bb8
6,323...	Other			x10,a0: 00000000
6,323...	Other			x11,a1: 00000000
6,323...	Other			x12,a2: 00000000
6,323...	Other			x13 a3: 00000000
6,323...	Other			x14,a4: 00000000
6,323...	Other			x15,a5: 00000000
6,323...	Other			x16,a6: 00000000
6,323...	Other			x17 a7: 00000000
6,323...	Other			x18,s2: 0000003c
6,323...	Other			x19,s3: 00000000
6,323...	Other			x20,s4: 00000000

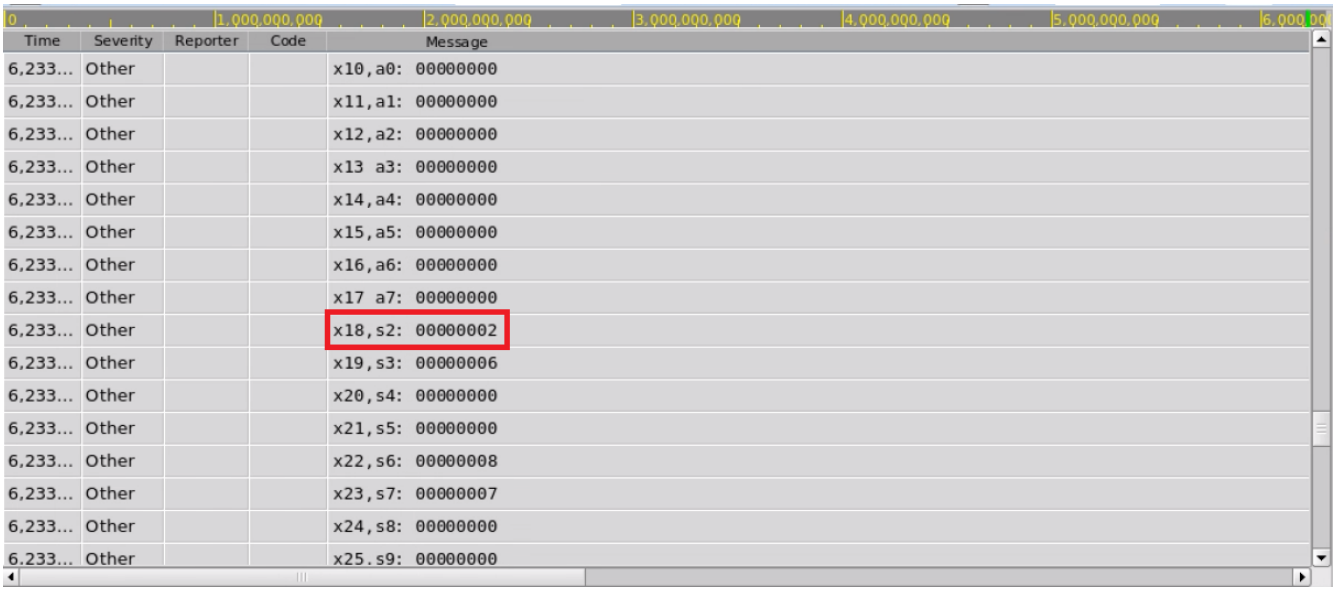
(a) Resultado en los registros



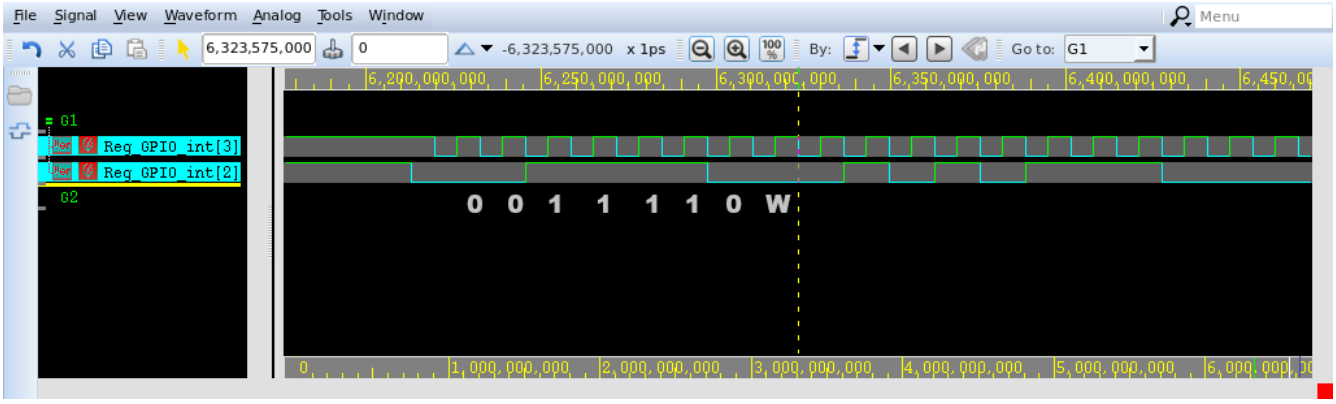
(b) Bits enviados de dirección

Figura 4.2. Resultados a 100 kHz

4.1.2. Pruebas a 400 kHz



(a) Resultado en los registros



(b) Bits enviados de dirección

Figura 4.3. Resultados a 400 kHz

4.2. Resultados con diferentes direcciones de dispositivo

Las pruebas de dirección se realizaron utilizando tres direcciones: `0x3C`, `0x3E` y `0x22`, todas ejecutadas a la misma frecuencia de 100 kHz. El objetivo de estas pruebas fue verificar el funcionamiento del algoritmo en condiciones controladas, observando si la recepción de datos se realizaba de manera adecuada y sin variaciones significativas entre las diferentes direcciones. Cualquier discrepancia en los resultados podría indicar un error en la recepción o en la transmisión de datos, lo que podría comprometer la integridad de la comunicación.

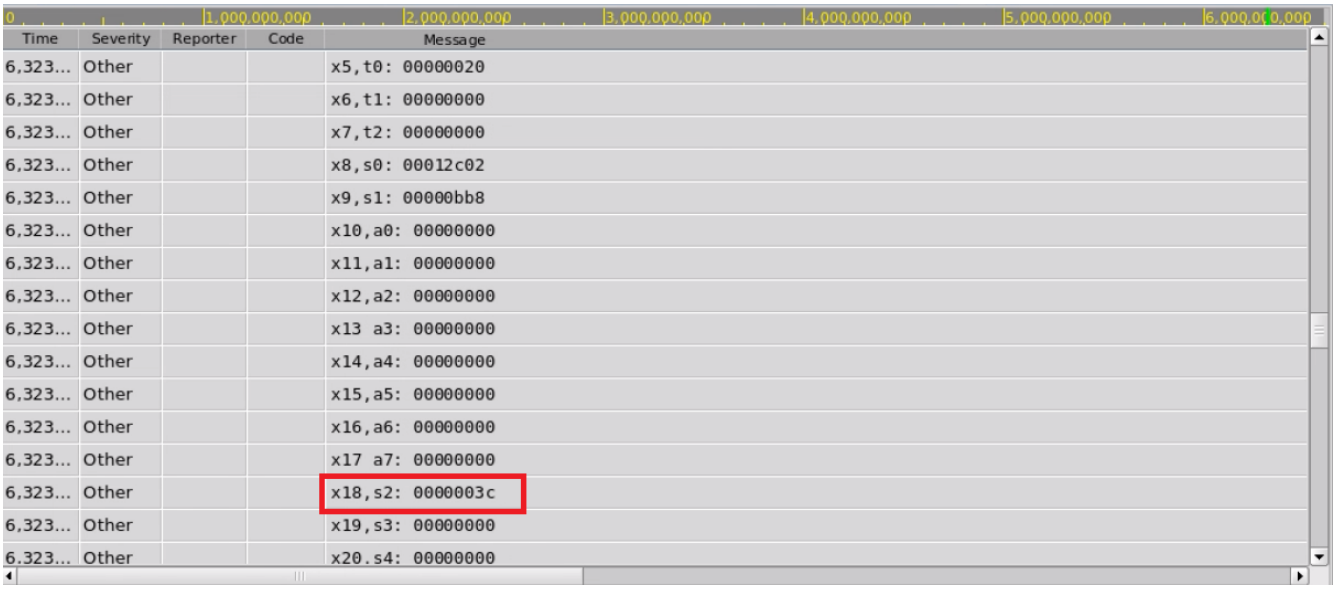
Los valores en binario de las direcciones utilizadas son los siguientes:

- `0x3C` = `0111100`
- `0x3E` = `0011111`
- `0x22` = `0100010`

Los resultados obtenidos en las pruebas realizadas concuerdan con los bits enviados desde el *testbench*, lo que indica que la comunicación fue exitosa en las etapas iniciales de *START* y *address*. Durante la fase de *START*, se verificó que la transición de la línea SDA de alto a bajo, mientras la línea SCL se mantenía en alto, se detectaba correctamente, lo que permitió iniciar la comunicación de manera adecuada. Posteriormente, en la etapa de *address*, se observó que los 7 bits de dirección y el bit de lectura/escritura (R/W) fueron correctamente transmitidos y muestreados. Las Figuras 4.4a y 4.4b muestran que el proceso de detección para la dirección `0x3C` enviada fue exitoso, se almacena dicha dirección en el registro `S2` y se procede a las siguientes etapas de comunicación. Esto aplica para las pruebas también con las direcciones dadas por `0x3E` y`0x22`, los cuales pueden ser observados en las Figuras 4.5 y 4.6. Este comportamiento indica que el algoritmo es capaz de manejar de manera precisa la transmisión de la dirección, sin pérdida de bits.

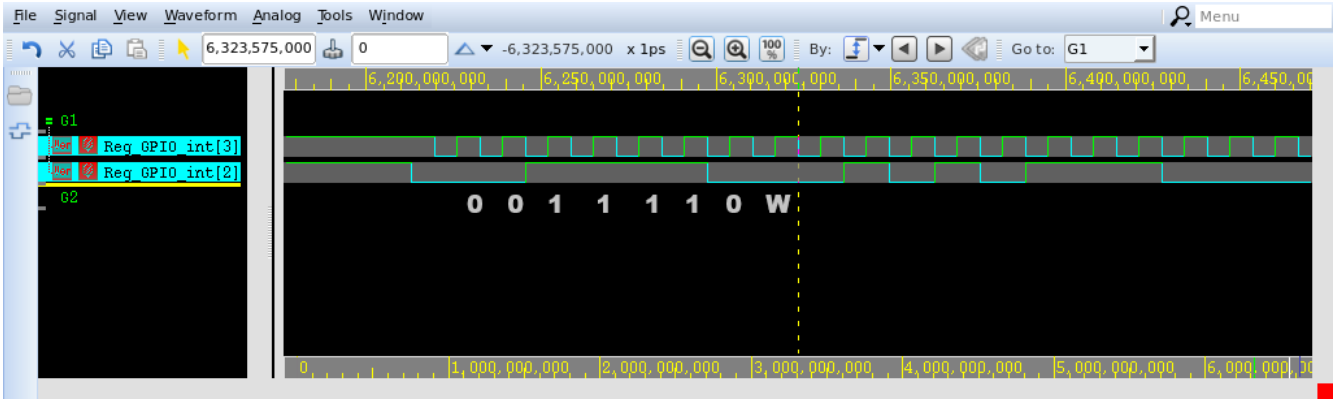
El éxito de estas dos fases de la comunicación valida que el algoritmo implementado es capaz de realizar el muestreo de datos correctamente en las condiciones de funcionamiento especificadas.

4.2.1. Pruebas con dirección 3C



Time	Severity	Reporter	Code	Message
6,323...	Other			x5,t0: 00000020
6,323...	Other			x6,t1: 00000000
6,323...	Other			x7,t2: 00000000
6,323...	Other			x8,s0: 00012c02
6,323...	Other			x9,s1: 00000bb8
6,323...	Other			x10,a0: 00000000
6,323...	Other			x11,a1: 00000000
6,323...	Other			x12,a2: 00000000
6,323...	Other			x13 a3: 00000000
6,323...	Other			x14,a4: 00000000
6,323...	Other			x15,a5: 00000000
6,323...	Other			x16,a6: 00000000
6,323...	Other			x17 a7: 00000000
6,323...	Other			x18,s2: 0000003c
6,323...	Other			x19,s3: 00000000
6,323...	Other			x20,s4: 00000000

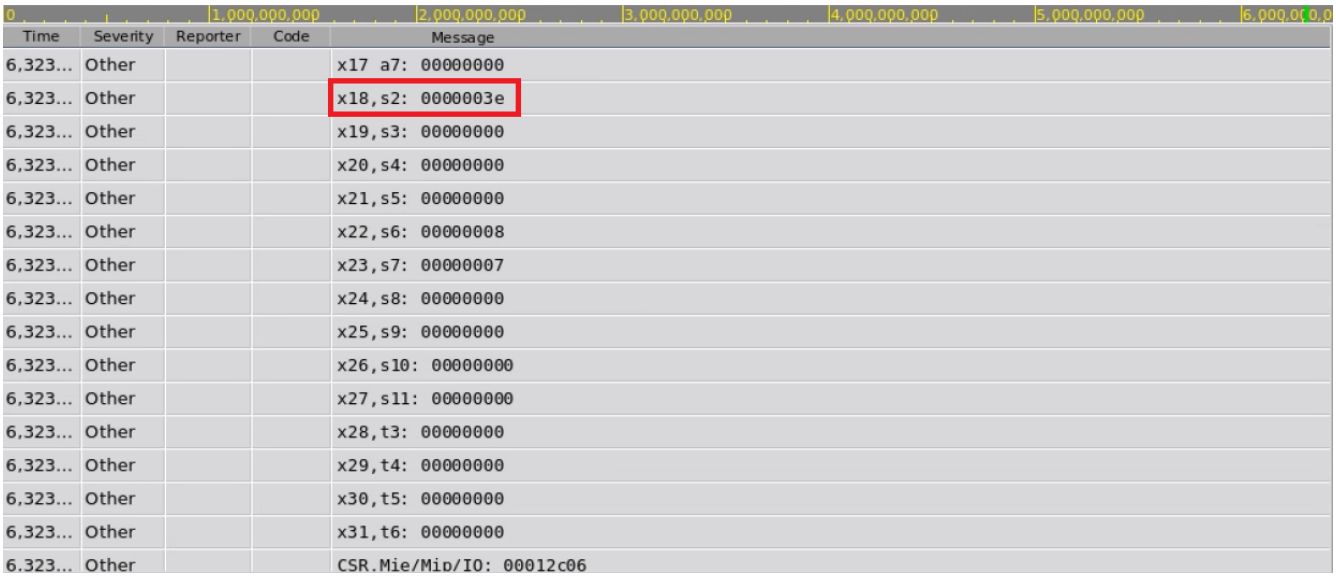
(a) Resultado en los registros



(b) Bits enviados de dirección 3C

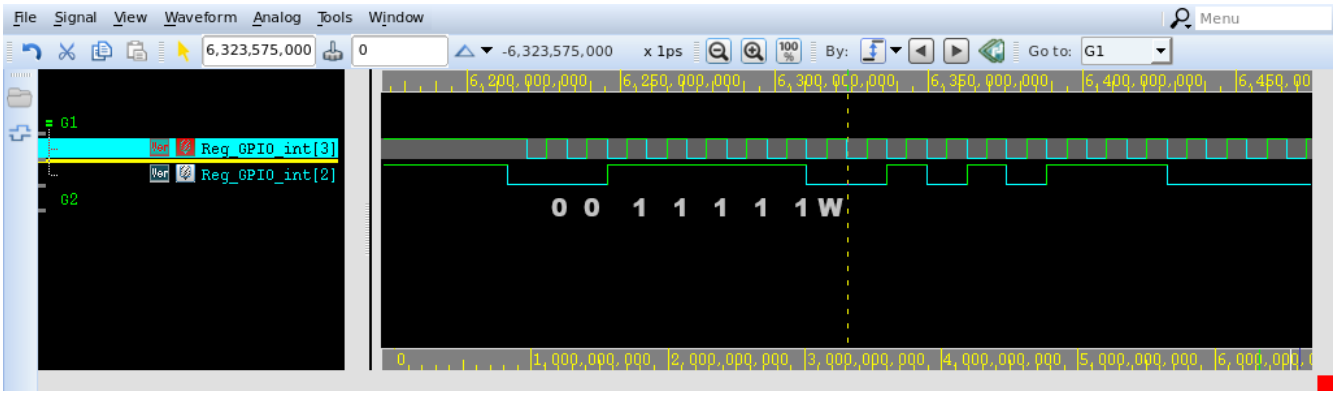
Figura 4.4. Resultados con la dirección 3C

4.2.2. Pruebas con dirección 3E



Time	Severity	Reporter	Code	Message
6,323...	Other			x17 a7: 00000000
6,323...	Other			x18,s2: 0000003e
6,323...	Other			x19,s3: 00000000
6,323...	Other			x20,s4: 00000000
6,323...	Other			x21,s5: 00000000
6,323...	Other			x22,s6: 00000008
6,323...	Other			x23,s7: 00000007
6,323...	Other			x24,s8: 00000000
6,323...	Other			x25,s9: 00000000
6,323...	Other			x26,s10: 00000000
6,323...	Other			x27,s11: 00000000
6,323...	Other			x28,t3: 00000000
6,323...	Other			x29,t4: 00000000
6,323...	Other			x30,t5: 00000000
6,323...	Other			x31,t6: 00000000
6,323...	Other			CSR.Mie/Mip/IO: 00012c06

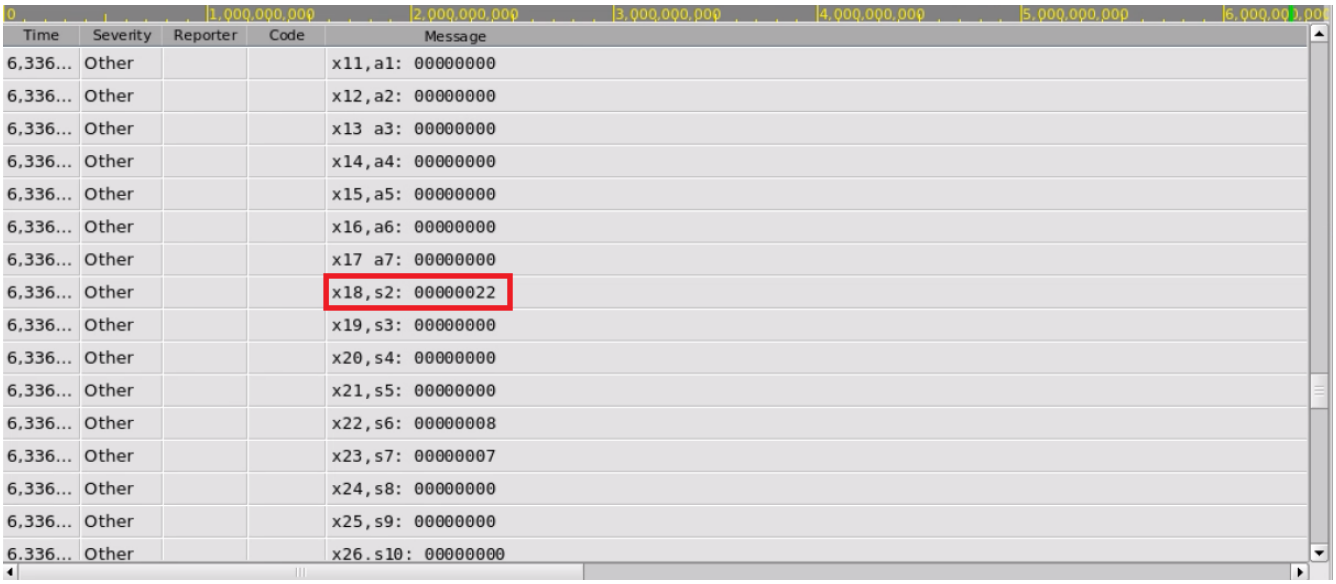
(a) Resultado en los registros



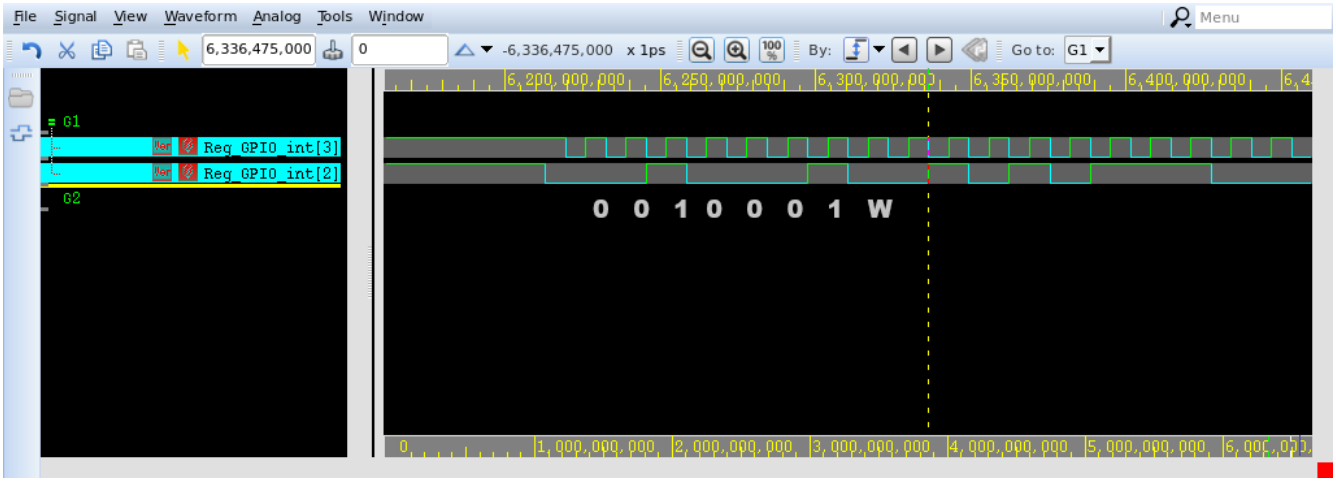
(b) Bits enviados de dirección 3E

Figura 4.5. Resultados con la dirección 3E

4.2.3. Pruebas con dirección 22



(a) Resultado en los registros



(b) Bits enviados de dirección 22

Figura 4.6. Resultados con la dirección 22

4.3. Resultados con diferentes sets de datos

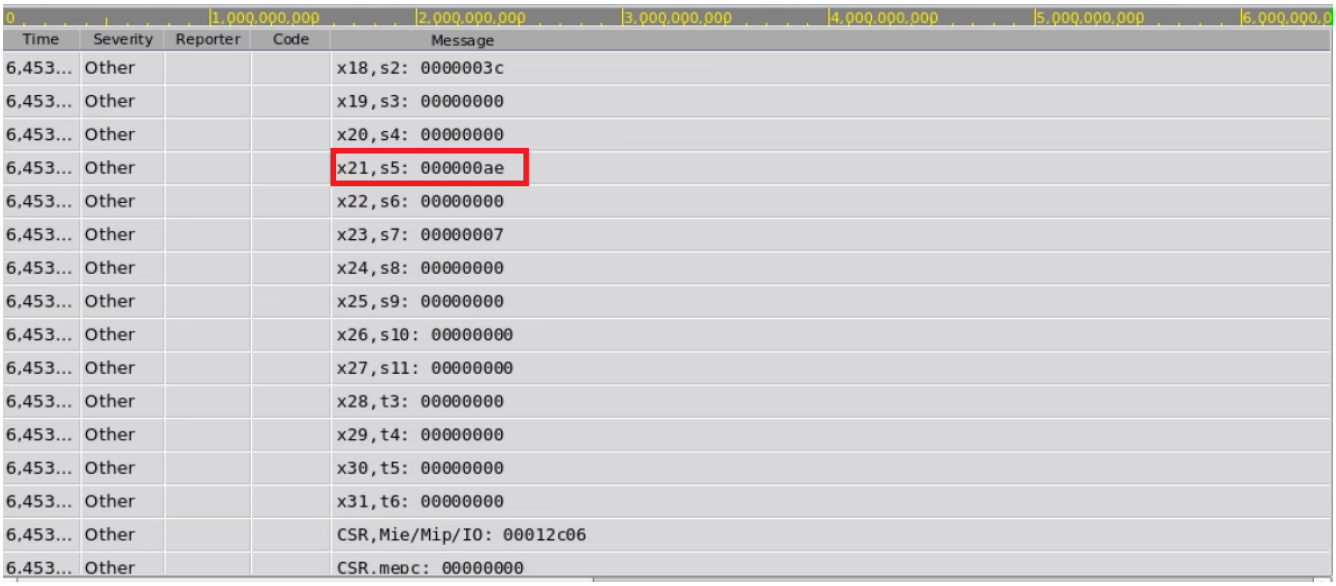
Para las pruebas con diferentes conjuntos de datos, el procedimiento se realizó de manera similar a las pruebas de direcciones, utilizando tres conjuntos de datos distintos y validando su correcta transmisión mediante la sincronización en la simulación. Los conjuntos de datos utilizados corresponden a 8 bits, ya que, aunque el transmisor simulado es configurable para manejar más de un byte, esto aumenta considerablemente el tiempo de simulación. Este aumento en el tiempo de

simulación resulta en un proceso de depuración más largo y complejo, lo que dificulta la identificación rápida de posibles errores en los datos transmitidos. Los conjuntos de datos seleccionados corresponden a los valores `AE`, `FF`, y `5C`, los cuales tienen las siguientes representaciones en binario:

- `AE` = `10101110`
- `FF` = `11111111`
- `5C` = `01011100`

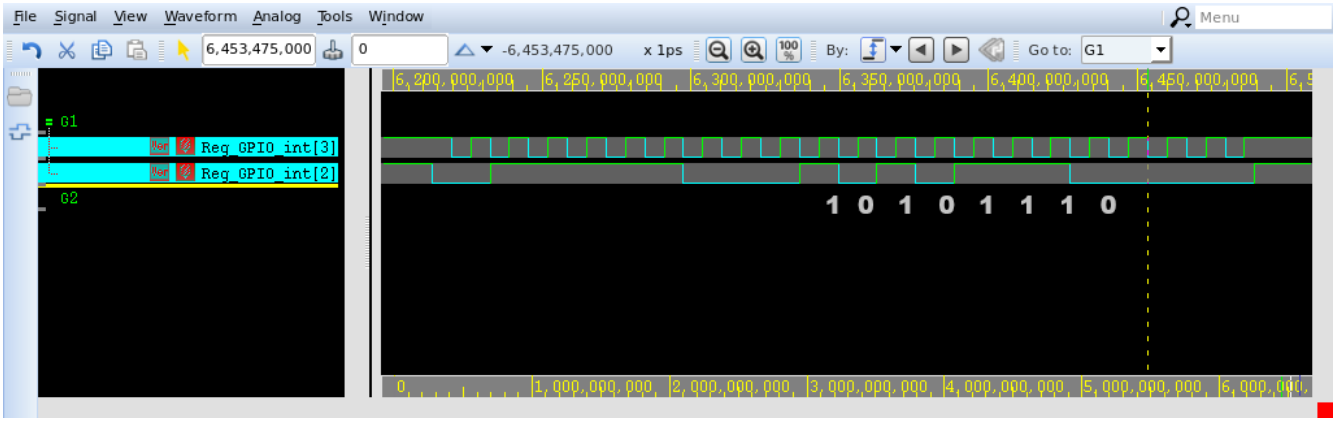
Estos valores fueron elegidos para verificar la correcta transmisión y recepción de datos, asegurando que el sistema pueda manejar diferentes patrones de bits y validando la integridad de la comunicación durante las pruebas de transmisión. El byte enviado se almacena en el registro `S5`, mientras que el registro `S6` funciona como un contador de datos. Este contador es reseteado cada vez que se requiere enviar un bit de *ACK*, lo que permite mantener un flujo de trabajo eficiente en el cual, por cada 8 bits transmitidos, se genera un *ACK* y, a continuación, se reanuda el muestreo de datos en caso de que se deban enviar más bytes. Este mecanismo asegura que el sistema pueda procesar múltiples bytes de manera continua, garantizando que cada byte enviado sea correctamente validado por el receptor antes de la transmisión del siguiente byte. El uso del contador en `S6` permite sincronizar el envío de *ACK* y el posterior muestreo de la línea SDA, manteniendo así la integridad y sincronización de la transmisión de datos.

Las pruebas de datos fueron exitosas para los bytes enviados, lo que permitió verificar tanto el funcionamiento como la lógica del programa en los tres conjuntos de datos utilizados. Para el primer set de datos `0xAE` es posible observar en la Figura 4.7 el envío de datos así como su estado en los registros, estos se encuentran sincronizados con el cursor de simulación, el cual se posiciona en el tiempo 6,453 s donde es posible analizar la trama de datos enviada en su totalidad. Así mismo es posible observar que el contador de datos en el registro `S6` se encuentra en 0, después de ejecutar 8 ciclos de recepción, consistentes con la cantidad de bits transmitidos. Este mismo procedimiento se puede observar en las Figuras 4.8 y 4.9, las cuales poseen el mismo comportamiento para los diferentes sets de datos utilizados. Con esto se puede validar el funcionamiento del algoritmo en la parte de recepción de datos y verificar la transmisión de la información deseada de forma exitosa. Estos resultados validan la lógica del programa, demostrando que el proceso de muestreo y la transmisión de datos se llevó a cabo de acuerdo con las especificaciones del protocolo I^2C.



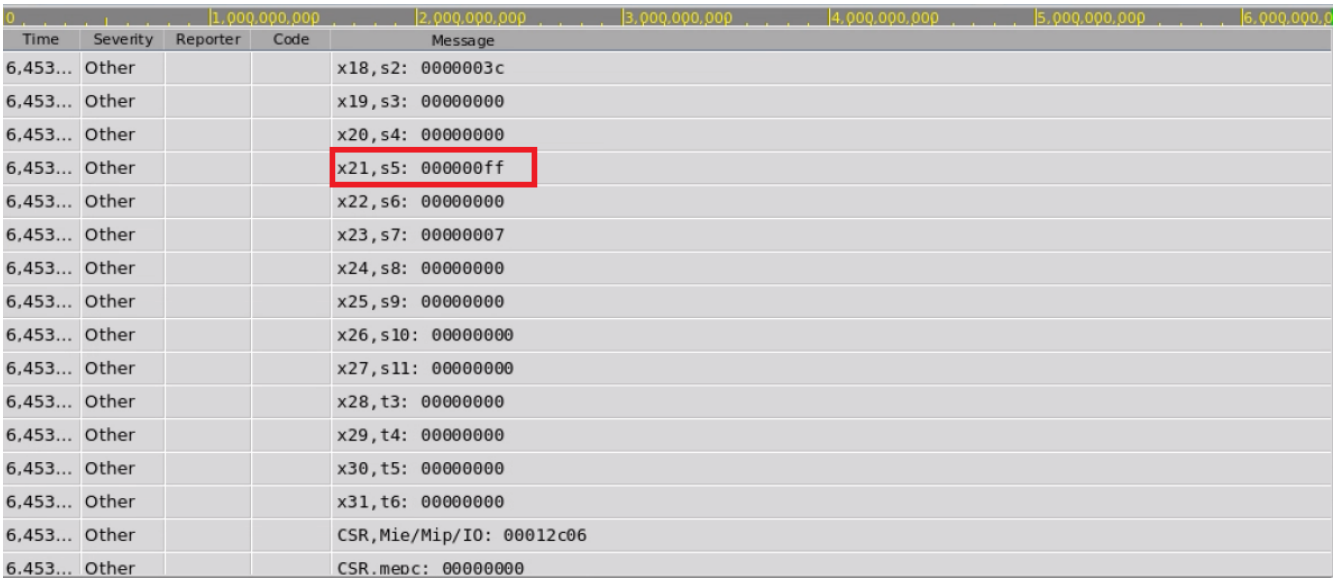
Time	Severity	Reporter	Code	Message
6,453...	Other			x18,s2: 0000003c
6,453...	Other			x19,s3: 00000000
6,453...	Other			x20,s4: 00000000
6,453...	Other			x21,s5: 000000ae
6,453...	Other			x22,s6: 00000000
6,453...	Other			x23,s7: 00000007
6,453...	Other			x24,s8: 00000000
6,453...	Other			x25,s9: 00000000
6,453...	Other			x26,s10: 00000000
6,453...	Other			x27,s11: 00000000
6,453...	Other			x28,t3: 00000000
6,453...	Other			x29,t4: 00000000
6,453...	Other			x30,t5: 00000000
6,453...	Other			x31,t6: 00000000
6,453...	Other			CSR,Mie/Mip/IO: 00012c06
6,453...	Other			CSR,mepc: 00000000

(a) Resultado en los registros



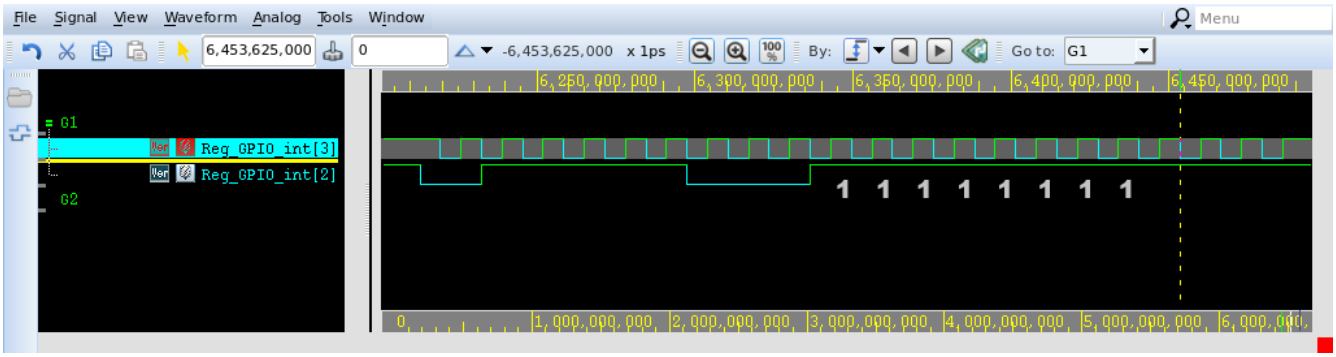
(b) Byte enviado

Figura 4.7. Resultados con el byte AE



Time	Severity	Reporter	Code	Message
6,453...	Other			x18,s2: 0000003c
6,453...	Other			x19,s3: 00000000
6,453...	Other			x20,s4: 00000000
6,453...	Other			x21,s5: 000000ff
6,453...	Other			x22,s6: 00000000
6,453...	Other			x23,s7: 00000007
6,453...	Other			x24,s8: 00000000
6,453...	Other			x25,s9: 00000000
6,453...	Other			x26,s10: 00000000
6,453...	Other			x27,s11: 00000000
6,453...	Other			x28,t3: 00000000
6,453...	Other			x29,t4: 00000000
6,453...	Other			x30,t5: 00000000
6,453...	Other			x31,t6: 00000000
6,453...	Other			CSR,Mie/Mip/IO: 00012c06
6,453...	Other			CSR,mepc: 00000000

(a) Resultado en los registros

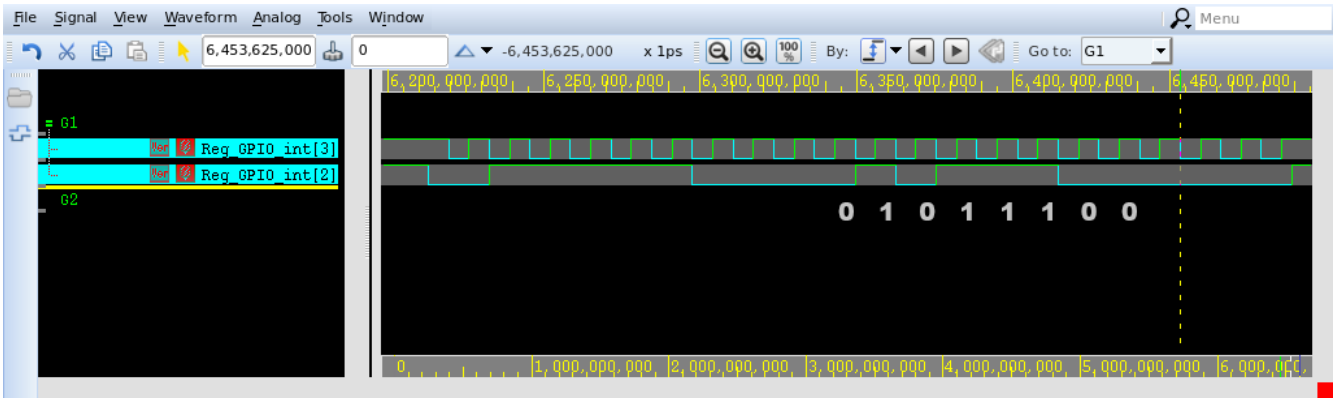


(b) Byte enviado

Figura 4.8. Resultados con el byte FF

Time	Severity	Reporter	Code	Message
6,453...	Other			x14,a4: 00000000
6,453...	Other			x15,a5: 00000000
6,453...	Other			x16,a6: 00000000
6,453...	Other			x17 a7: 00000000
6,453...	Other			x18,s2: 0000003c
6,453...	Other			x19,s3: 00000000
6,453...	Other			x20,s4: 00000000
6,453...	Other			x21,s5: 0000005c
6,453...	Other			x22,s6: 00000000
6,453...	Other			x23,s7: 00000007
6,453...	Other			x24,s8: 00000000
6,453...	Other			x25,s9: 00000000
6,453...	Other			x26,s10: 00000000
6,453...	Other			x27,s11: 00000000
6,453...	Other			x28,t3: 00000000
6,453...	Other			x29,t4: 00000000

(a) Resultado en los registros



(b) Byte enviado

Figura 4.9. Resultados con el byte 5C

En base a los resultados obtenidos, es posible realizar una comparación con implementaciones similares en microprocesadores, tomando en cuenta sus arquitecturas y las implementaciones asociadas. La tabla 4.1 presenta varias familias de microprocesadores y sus características. Estos microprocesadores comparten una arquitectura similar, basada en un conjunto de instrucciones RISC, con la excepción del microprocesador 8751BH, el cual emplea una arquitectura propietaria de Intel. No obstante, sus características son similares, y la implementación de un protocolo I^2C emulado por software permite realizar una comparación en términos de funcionamiento.

De manera similar, varios algoritmos han sido diseñados para emular la comunicación a través del protocolo I^2C. Las referencias a las arquitecturas de cada uno de los microprocesadores y sus datos completos pueden ser consultadas en [13], [14] y [15].

Para el microprocesador MSP430, se implementa una emulación utilizando tanto el lenguaje C como el lenguaje ensamblador [16]. Esta implementación se basa en una máquina de estados y tiene dos

Tabla 4.1. Comparación de la arquitectura Siwa con otros microprocesadores

Core	Siwa	PIC16C5X	8751BH	MSP430
Instruction word size (bits)	32	12	8	16
Program Memory	8 kB	512 B-2 kB	4-8 kB	2-16 kB
Clock freq	20 MHz	20-40 MHz	8-12 MHz	4-16 MHz
Average CPI	4	1.8	–	1

configuraciones: comunicación desde el MSP como transmisor y como receptor. Para los fines de esta comparación, se analiza la implementación como receptor. En una configuración con un reloj de 8 MHz, se logra una frecuencia de transmisión en la línea SCL de 100 kHz utilizando lenguaje ensamblador, obteniendo un resultado comparable con la implementación realizada en el microprocesador Siwa, en el cual, con un reloj de 13 MHz, se alcanzó la misma frecuencia de transmisión de 100 kHz. En términos de memoria de programa, el código generado para ensamblador en el MSP430 ocupa menos de 1 kB. Por otro lado, para el microprocesador Siwa, el código generado en ensamblador tiene un tamaño de 827 Bytes, lo cual se encuentra dentro de los rangos establecidos y produce resultados similares a los obtenidos con el MSP430. Otro aspecto relevante a analizar es la implementación directa del protocolo en el MSP430. Este microprocesador incluye una función que permite no solo recibir, sino también realizar escritura de datos hacia el transmisor. Sin embargo, esta función no está implementada en el algoritmo actual del microprocesador Siwa, lo que representa una limitación, dependiendo de la aplicación a desarrollar.

El microprocesador PIC16C5 ofrece una implementación de I^2C por software, con resultados similares a los obtenidos en el microprocesador Siwa [17]. En esta implementación, se desarrolla únicamente el controlador como receptor. De manera similar a la implementación en el microprocesador MSP430, la frecuencia máxima de operación para la línea SDA está limitada a 100 kHz. Si bien los resultados obtenidos son comparables a los alcanzados con la implementación en Siwa, es importante señalar que la implementación del PIC16C5 no especifica detalles acerca de la velocidad de reloj utilizada. Según los datos obtenidos del datasheet, el PIC16C5 presenta un CPI promedio superior al de Siwa, lo que puede influir en el rendimiento de la transmisión y recepción de datos. Este aspecto debe ser considerado, ya que una mayor capacidad de ejecución de instrucciones por ciclo de reloj facilita un mejor muestreo de las líneas SCL y SDA, lo que a su vez mejora la precisión en la emulación del protocolo I^2C.

Realizando la comparación con la implementación en el microprocesador 8751BH, se observa que este realiza primero una recepción de datos, los cuales son luego transferidos a un display para observar los resultados y verificar la integridad de la información transmitida. Para efectos de comparación, se enfoca la atención en la parte de recepción y sus resultados [18]. Los resultados obtenidos indican que, para un reloj interno con una frecuencia de 12 MHz, la velocidad máxima alcanzada en la línea SCL corresponde a 66.7 kHz. En comparación con los resultados obtenidos con el microprocesador Siwa, este último presenta una capacidad de manejo de la línea SCL mayor, lo que implica que la implementación en Siwa tiene un rendimiento ligeramente superior en cuanto a la velocidad de comunicación a través de I^2C.

La Tabla 4.2 ofrece un resumen de los resultados de las comparaciones.

Tabla 4.2. Comparación de las implementaciones de I^2C en diferentes microprocesadores

microprocesador	**Frecuencia de Reloj**	**Frecuencia SCL**	**Tamaño del Código (Bytes)**	**CPI Promedio**
MSP430	8 MHz	100 kHz	<1 kB	Mayor al de Siwa
Siwa	13 MHz	100 kHz	827 Bytes	4
PIC16C5	No especificado	100 kHz	No especificado	Mayor al de Siwa
8751BH	12 MHz	66.7 kHz	No especificado	No especificado

Capítulo 5

Conclusiones y Recomendaciones

El presente capítulo resume los principales hallazgos obtenidos a lo largo del desarrollo del proyecto, destacando los resultados más relevantes y el grado de cumplimiento de los objetivos planteados. Asimismo, se presentan recomendaciones orientadas a mejorar, complementar o ampliar el trabajo realizado, tanto desde el punto de vista técnico como metodológico.

5.1. Conclusiones

Las pruebas realizadas demostraron que el algoritmo implementado es capaz de emular correctamente el funcionamiento de Siwa como un receptor en el protocolo I^2C. Se validaron con éxito las fases críticas de la comunicación, tales como las condiciones de *START* y *address*, hasta la fase de *STOP* con resultados coherentes en los registros del sistema. Aunque el algoritmo funcionó correctamente a 100 kHz, se identificaron limitaciones a frecuencias superiores, especialmente a 400 kHz y más allá. Esto señala la necesidad de optimizaciones adicionales o de emplear hardware dedicado para comunicaciones a frecuencias más altas. A pesar de las limitaciones a frecuencias altas, la implementación en software del protocolo I^2C en el microcontrolador Siwa ha mostrado ser una solución viable y económica para emular dicho protocolo sin la necesidad de hardware especializado. Esto abre la puerta a la flexibilidad en sistemas embebidos que no cuentan con interfaces I^2C físicas. El tamaño total del programa corresponde a 827 Bytes, el cual se encuentra dentro de los rangos establecidos inicialmente, donde se especificó que el programa no debía superar los 8 KB. Esto fue posible gracias a un manejo adecuado de los datos y a la optimización del código, evitando la sobrecarga innecesaria de instrucciones.

5.2. Recomendaciones

- **Optimización para altas frecuencias:** Se recomienda optimizar el algoritmo para mejorar el muestreo de datos a frecuencias superiores a 100 kHz. Esto podría lograrse mediante la reducción de ciclos de instrucciones innecesarios o la implementación de técnicas de muestreo más eficientes.
- **Pruebas adicionales con dispositivos reales:** Si es posible, sería beneficioso realizar pruebas con dispositivos reales utilizando el protocolo I^2C, para confirmar el comportamiento del sistema bajo condiciones de operación más dinámicas.
- **Verificación Automatizada:** Implementar algún tipo de verificación automatizada en SystemVerilog resulta necesario, dado que el proceso actual de validación es lento y depende de un análisis manual bit a bit para la identificación de errores y fallas. Considerando la gran cantidad de permutaciones posibles tanto en los bits de *address* como en los datos, se vuelve imperativo diseñar e integrar un sistema de verificación automatizado que permita detectar discrepancias de manera eficiente, reducir la intervención humana y aumentar la confiabilidad del proceso de verificación.

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, T̈RSW CI: Communication interfaces.̈ Disponible en: https://metrologia.eti.pg.gda.pl/~TRSW/TRSW_CI.pdf.

[5] D. M. Harris and S. L. Harris, *Digital Design and Computer Architecture*, 2nd ed. Waltham, MA: Morgan Kaufmann, 2012.

[6] J. F. Wakerly, *Digital Design: Principles and Practices*, 4th ed. Upper Saddle River, NJ: Pearson, 2005.

[7] C. H. Roth and L. L. Kinney, *Fundamentals of Logic Design*, 7th ed. Cengage Learning, 2013.

[8] V. A. Pedroni, *Digital Design and Verilog HDL Fundamentals*. Prentice Hall, 2007.

[9] R. Molina-Robles, E. Solera-Bolanos, R. García-Ramírez, A. Chacón-Rodríguez, A. Arnaud, and R. Rimolo-Donadio, "A compact functional verification flow for a risc-v 32i based core," in *2020 IEEE Conference Proceedings*. IEEE, 2020, pp. 1–6, accessed via internal document.

[10] E. S. Bolaños, "Register file: Especificación del sistema siwa," DCILAB, Instituto Tecnológico de Costa Rica, Reporte Técnico Interno, August 2018, versión: Registro de CSR, configuración y arquitectura del SIWA.

[11] R. M. Robles, “Tutorial 01: Acceso al servidor,” 2023, accedido: 2025-11-17. [Online]. Available: https://youtu.be/_9Ou5SbmWZc

[12] ——, *Tutorial: Flujo para creación de programas SIWA*, Escuela de Ingeniería Electrónica, Instituto Tecnológico de Costa Rica, 2023, accedido: 2023-11-17. [Online]. Available: https://virtual.ie.tec.ac.cr/

[13] M. T. Inc., “Pic16c5x data sheet,” https://www.microchip.com, 2002, preliminary, DS30453D. [Online]. Available: https://www.microchip.com

[14] I. Corporation, “Mcs® 51 8-bit control-oriented microcontrollers,” https://www.intel.com, 1994, order Number 272318-002, October 1994. [Online]. Available: https://www.intel.com

[15] T. I. Inc., “Msp430g2x53 data sheet,” https://www.ti.com, 2011, sLAS735J, Revised May 2013. [Online]. Available: https://www.ti.com

[16] T. Instruments, “Software i2c on msp430 mcus,” Texas Instruments, Tech. Rep., 2018, application Report SLAA703A, Revised July 2018. [Online]. Available: https://www.ti.com/lit/zip/slaa703

[17] D. Lekei, “Using a pic16c5x as a smart i2c peripheral,” 1993, microchip Technology Application Note, AN541. [Online]. Available: https://www.microchip.com/wwwproducts/en/PIC16C5X

[18] S. D. Quarles, “How to implement i2c serial communication using intel mcs-51 microcontrollers,” 1993, intel Application Note, AP-476. [Online]. Available: http://ww1.microchip.com/downloads/en/AppNotes/00541e.pdf

Capítulo 6

Anexos

El capítulo de anexos reúne la documentación complementaria que respalda y amplía la información presentada en el cuerpo principal de esta tesis.

Anexo A: Código Ensamblador del Programa Presentado

Enlace al Repositorio con todos los códigos del trabajo

Flujo del Algoritmo

El flujo general del algoritmo sigue los siguientes pasos:

- Inicialización y configuración de los GPIO.
- Muestreo de la línea SCL para detectar el inicio de la transmisión.
- Muestreo de SDA para leer la dirección y los datos, insertándolos en el registro correspondiente.
- Generación del ACK después de cada byte recibido.
- Ejecución de la condición de *STOP* al finalizar la transmisión de datos.

Anexo B: Código de Prueba para GPIOS

```
task Pulse();
    begin
        // Espera inicial (opcional)
        $display("Iniciando pulso simple en GPIOs");

        // Poner SCL y SDA en 1
        Reg_GPIO_int[SCL] = 1'b1;
        Reg_GPIO_int[SDA] = 1'b1;
        $display("Reg GPIO int (HIGH) = %08b",
    Reg_GPIO_int);

        // Mantenerlos en el cierto tiempo (por ejemplo
    FI2C ciclos de clk)
        repeat (FI2C) @(posedge clk);

        // Poner SCL y SDA en 0
        Reg_GPIO_int[SCL] = 1'b0;
        Reg_GPIO_int[SDA] = 1'b0;
        $display("Reg GPIO int (LOW) = %08b",
    Reg_GPIO_int);

        // Espera final (opcional)
        repeat (FI2C) @(posedge clk);

        $display("Termina SimpleGpioPulse");
    end
endtask : Pulse
```

Código 6.1: Código de ejemplo para generar un pulso en los GPIO

Anexo C: Código para generacion del archivo TXT

```python
import sys

file_rep=sys.argv[1]

file = open(file_rep, "r")
content = file.readlines()
file.close()
mem_len = len(content)
mem_new=[0 for i in range (0,(mem_len+2)*4)]
for x in range(0,mem_len):
  mem_new[x*4]= content[x][6:8]
  mem_new[x*4+1] = content[x][4:6]
  mem_new[x*4+2] = content[x][2:4]
  mem_new[x*4+3] = content[x][0:2]

end_bst = mem_len+1

file2 = open("mem_model.txt","w")
for x in range(0,end_bst*4):
  if(x<end_bst*4-4):
    file2.write(mem_new[x]+"\n")
  else:
    if(x<end_bst*4-1):
      file2.write("ff\n")
    else:
      file2.write("ff")
file2.close()
```

Código 6.2: Codigo de Python para Generacion de Archivo

Anexo D: Código Ensamblador de Prueba

```
   .section .text
    .globl _start

_start:
    # Cargar constantes
    li      t0, 5          # t0 = 5
    li      t1, 3          # t1 = 3

    # Suma
    add     t2, t0, t1     # t2 = t0 + t1 = 8

    # Resta el segundo operando
    sub     t3, t2, t1     # t3 = t2 - t1 = 5

    # Bucle infinito para que no siga ejecutando basura
end:
    j       end
```

Código 6.3: Codigo ensamblador de ejemplo

INSTITUTO TECNOLÓGICO DE COSTA RICA

ESCUELA DE INGENIERÍA ELECTRÓNICA

TRABAJO FINAL DE GRADUACIÓN

ACTA DE APROBACIÓN

Defensa del Trabajo Final de Graduación
Requisito para optar por el título de Ingeniero en Electrónica
Grado Académico de Licenciatura
Instituto Tecnológico de Costa Rica

El Tribunal Evaluador aprueba la defensa del Trabajo Final de Graduación denominado *Diseño de un algoritmo para la emulación por software del protocolo I2C para la recepción de datos*, realizado por el señor Iliak Flores Barrantes y, hace constar que cumple con las normas establecidas por la Escuela de Ingeniería Electrónica del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal Evaluador

Dr. Ing. Alfonso Chacón Rodríguez

Profesor lector

Dr. Ing. Rónny García Ramírez

Profesor lector

Dr. Ing. Roberto Molina Robles

Profesor asesor

Cartago, 28 de Noviembre de 2025

INSTITUTO TECNOLÓGICO DE COSTA RICA
ESCUELA DE INGENIERÍA ELECTRÓNICA
TRABAJO FINAL DE GRADUACIÓN
TRIBUNAL EVALUADOR
ACTA DE EVALUACIÓN

Defensa del Trabajo Final de Graduación
Requisito para optar por el título de Ingeniero en Electrónica
Grado Académico de Licenciatura
Instituto Tecnológico de Costa Rica

Estudiante: **Iliak Flores Barrantes** Carné: 2019046899

Nombre del Trabajo Final de Graduación: *Diseño de un algoritmo para la emulación por software del protocolo I2C para la recepción de datos*

Los miembros de este Tribunal hacen constar que este Trabajo Final de Graduación ha sido aprobado y cumple con las normas establecidas por la Escuela de Ingeniería Electrónica del Instituto Tecnológico de Costa Rica

Nota de Trabajo Final de Graduación: 80%

Miembros del Tribunal

Dr. Ing. Alfonso Chacón Rodríguez
Profesor lector

Dr. Ing. Ronny García Ramírez
Profesor lector

Dr. Ing. Roberto Molina Robles
Profesor asesor

Cartago, 28 de Noviembre de 2025