



Escuela Técnica Superior
de Ingeniería Informática

Grado en Ingeniería Informática

Curso 2025–2026

Trabajo Fin de Grado

JAZZ80
JUST ANOTHER ZILOG Z80
SIMULADOR DE Z80

Autor: Miguel González Rodríguez

Tutor: Luis Rincón Córcoles

Agradecimientos

Breves agradecimientos o dedicatoria.

Resumen

Este Trabajo de Fin de Grado presenta el diseño, implementación y validación de un simulador completo del microprocesador Zilog Z80, una arquitectura de 8 bits fundamental en la historia de la computación. El proyecto, desarrollado íntegramente en el lenguaje de programación Rust, persigue el objetivo principal de proporcionar una herramienta educativa y de desarrollo moderna, eficiente y de código abierto, capaz de facilitar la comprensión profunda de la arquitectura de computadores y la programación en lenguaje ensamblador sin las barreras de acceso del hardware físico obsoleto.

La arquitectura del software se ha diseñado de forma modular, separando estrictamente el núcleo de emulación de la capa de presentación. El núcleo implementa con precisión a nivel de instrucción del Z80, gestionando fielmente el conjunto de registros, las banderas de estado, los diversos modos de direccionamiento y el sistema de interrupciones. Gracias al uso de la biblioteca gráfica **eframe** y la potencia de Rust, la aplicación ofrece portabilidad total, ejecutándose de manera nativa en sistemas operativos de escritorio (Windows, Linux, macOS) y en navegadores web modernos mediante compilación a WebAssembly (WASM), eliminando así la necesidad de instalaciones complejas para el usuario final.

El objetivo de este trabajo es proporcionar una plataforma accesible y educativa para estudiantes, incluidos de esta universidad. El resultado final es una interfaz completa (IDE), con ensamblador y capacidad de visualizar los cambios en la CPU y la memoria en tiempo real. Así como la posibilidad de correrlo en Web y no tener que descargar ningún archivo, ni siquiera el archivo de Assembly mismo.

Palabras clave:

- Zilog Z80
- Simulador
- Rust
- WebAssembly
- Ensamblador
- Emulación
- Arquitectura de Computadores

- Egui

Índice de contenidos

Índice de tablas	XII
Índice de figuras	XV
1. Introducción	1
1.1. Motivación y Origen del Proyecto	2
1.2. Justificación Académica	2
1.3. Objetivos del Trabajo	3
2. Ecosistema de Desarrollo y Tecnologías	4
2.1. Lenguaje de Programación: Rust	4
2.1.1. Propiedad (<i>Ownership</i>) y Préstamos (<i>Borrowing</i>)	5
2.1.2. Tipado: Estructuras y Enumeraciones	5
2.1.3. <i>Traits</i> e Interfaces	6
2.1.4. Polimorfismo: Despacho Dinámico (<i>Dispatching</i>)	7
2.1.5. Mutabilidad Interior y Punteros Inteligentes	8
2.1.6. Metaprogramación (<i>Macros</i>)	8
2.2. Compilación WebAssembly (WASM)	9
2.3. Integración GUI (Entorno de Trabajo eframe)	10
3. Objetivos	11
3.1. Descripción del problema	12
3.2. Estudio de alternativas	12
3.3. Metodología empleada	13
3.3.1. Tecnologías y herramientas	13
3.3.2. Fases de desarrollo	14
3.4. Evolución del Proyecto y Retos Tecnológicos	14
3.4.1. Implementación del Núcleo y Aritmética BCD	15
3.4.2. Adaptación a Arquitectura Guiada por Eventos	15
3.4.3. Diseño del Ensamblador de Dos Pasadas	16
3.4.4. Sincronización del Sistema de Interrupciones	16
3.4.5. Abstracción de Plataforma y Almacenamiento	17
3.4.6. Cobertura del Set de Instrucciones Extendido	17

4. Descripción informática	19
4.1. Arquitectura Interna	19
4.1.1. Modelo de Registros	20
4.1.2. La Unidad Aritmético-Lógica (ALU) y el Registro de Banderas	21
4.2. Ciclo de Instrucción y Temporización	22
4.2.1. Secuencia Fetch-Decode-Execute	22
4.3. Sistema de Interrupciones	23
4.3.1. Interrupción No Enmascarable (NMI)	23
4.3.2. Interrupciones Enmascarables (INT)	23
4.4. Espacio de Entrada/Salida (E/S)	23
4.5. Set de Instrucciones Extendido	24
5. Desarrollo e Implementación	25
5.1. Diseño del Sistema	25
5.1.1. Estructura de Módulos	25
5.1.2. Arquitectura y Diagrama de Clases	26
5.1.3. Diseño de Datos y Modelo de Registros	28
5.1.4. Polimorfismo mediante Despacho Dinámico (<i>Dynamic Dispatch</i>)	30
5.2. Implementación y Detalles Técnicos	31
5.2.1. El Bucle de Instrucción Principal	31
5.2.2. Mecanismo Estructurado de Decodificación (Base Octal)	34
5.2.3. Tipado Fuerte en la Decodificación: El <i>Trait TryFrom</i>	36
5.2.4. La ALU y el Registro de Banderas	37
5.2.5. Fidelidad y Comportamiento No Documentado	38
5.2.6. El Ensamblador de Dos Pasadas	39
5.2.7. Sistema de Periféricos y Dispositivos de E/S	41
5.2.8. Interfaz Gráfica y Renderizado	42
5.2.9. Ejecución en WebAssembly (WASM)	43
6. Experimentos / Validación	47
6.1. Metodología de Pruebas Unitarias	47
6.2. Infraestructura de Pruebas: <code>MockMemory</code>	48
6.3. Análisis de los Ficheros de Prueba	49
6.3.1. Validación Aritmética: <code>alu_tests.rs</code>	49
6.3.2. Operaciones Tabuladas: <code>alu_table_ops.rs</code>	50
6.3.3. Control de Flujo: <code>jumps.rs</code>	50
6.3.4. Interrupciones y NMI: <code>nmi_tests.rs</code>	51
6.4. Validación del Ensamblador	51
6.4.1. Tokenización: <code>mod.rs</code>	51
6.4.2. Resolución de Etiquetas	51

6.5. Validación de Aritmética BCD (Decimal Adjust)	52
6.5.1. Validación de la Unidad Aritmético-Lógica (ALU)	52
6.5.2. Validación de Aritmética BCD (Decimal Adjust)	53
6.6. Validación del Sistema de Interrupciones	54
6.7. Validación Funcional: Algoritmos Completos	54
6.7.1. Caso de Estudio: Sucesión de Fibonacci	55
6.8. Conclusiones de la Validación	56
7. Conclusiones	58
7.1. Logros principales alcanzados	58
7.2. Posibles trabajos futuros	58
Bibliografía	60
Apéndices	62
A. Este es el primer apéndice	64
A.1. Ejemplo de sección	64

Índice de tablas

5.1. Disección en campos de un opcode de 8 bits del Z80.	34
5.2. Mapeo de operandos base del procesador Z80 (r , cc , rp , rp2 , alu y rot).	35
5.3. Decodificación de modos de interrupción del Z80 (im).	35
5.4. Matriz de decodificación de instrucciones de bloque del Z80 (bli).	36
5.5. Estructura de los registros índice mostrando los componentes de los bytes alto y bajo.	38

Índice de figuras

2.1. Ejemplo de estructura en Rust para modelar la memoria de 64 KB del Z80.	6
2.2. Ejemplo de enumeración en Rust para modelar los registros de propósito general del Z80.	6
2.3. Ejemplo de <i>trait</i> en Rust para definir la interfaz de un dispositivo mapeado en memoria.	7
2.4. Ejemplo de macro declarativa en Rust para definir un sistema de <i>logs</i> de depuración en el emulador.	9
5.1. Diagrama de Clases UML estructural del Simulador Z80, mostrando la arquitectura de la capa de interfaz y la de emulación compartiendo el bus de contexto de memoria.	27
5.2. Diagrama de Clases y Enumeraciones que componen el estado interno de la CPU Z80A y el sistema de decodificación tipado. . . .	28
5.3. Implementación de Mem64k con arreglo de 64KB.	29
5.4. Estructura base de la CPU instanciando la memoria polimórfica. .	29
5.5. Implementación alternativa descartada usando Despacho Estático.	30
5.6. Funciones principales del bucle de ejecución <i>Fetch-Decode-Execute</i> .	32
5.7. Diagrama de Secuencia del bucle de aplicación y la interacción con la rutina de ejecución <code>tick()</code> de la CPU, revelando el aislamiento entre la abstracción del hardware <code>MemoryMapper</code> y el renderizado UI.	33
5.8. Implementación del <i>trait</i> <code>TryFrom</code> en las operaciones ALU.	36
5.9. Implementación de la copia de resultados en instrucciones con prefijos compuestos.	39
5.10. Diagrama de Secuencia del Ensamblador de Dos Pasadas interactuando con <code>Z80App</code> y <code>MemoryMapper</code> . Muestra cómo el ensamblador es estrictamente funcional (sin estado ni dependencias colaterales) delegando el volcado binario a la capa unificada de la aplicación. .	40
5.11. Declaración del <i>trait</i> <code>IIODevice</code> para dispositivos mapeados por puertos.	41
5.12. Extensión de dispositivo con capacidades gráficas mediante <code>DeviceWithUi</code> .	42
5.13. Actualización de la interfaz en <code>Egui</code> simulando pasos de la CPU. .	43

5.14. Diagrama de Secuencia ilustrando el bucle gráfico para el flujo de depuración (Manual Stepping) e inspección de la línea activa controlada por PC.	45
6.1. Estructura MockMemory para realizar las inyecciones de código en tests.	48
6.2. Uso básico de la macro #[case] para tests tabulares de rstest . . .	50
6.3. Prueba de salto relativo simulando un desbordamiento negativo. .	50
6.4. Prueba completa de rotación bit a bit (RLC) validando banderas. .	52
6.5. Lógica de <i>parsing</i> para <i>Golden File Testing</i> de la instrucción DAA . .	53
6.6. Prueba de inhibición de interrupciones enmascarables bajo un evento NMI	54
6.7. Programa de cálculo de la sucesión de Fibonacci en ensamblador del Z80	55

1

Introducción

La enseñanza de la Arquitectura de Computadores constituye uno de los pilares fundamentales en la formación académica de cualquier ingeniero informático. Comprender cómo el software interactúa con el hardware a bajo nivel, cómo se gestiona la memoria, y cómo una CPU ejecuta instrucciones secuenciales es vital para desarrollar una intuición profunda sobre el rendimiento y la optimización de sistemas. Sin embargo, en la era actual de la computación en la nube y los lenguajes de alto nivel, esta conexión con el “hierro” a menudo se diluye.

Históricamente, el microprocesador Zilog Z80 ha sido una de las arquitecturas más influyentes y pedagógicamente valiosas. Su diseño de 8 bits, basado en registros y acumulador, ofrece un equilibrio perfecto entre simplicidad —permitiendo a un estudiante abarcar mentalmente todo el estado de la máquina— y complejidad real, introduciendo conceptos como interrupciones, modos de direccionamiento y gestión de pila que siguen vigentes hoy en día.

Este Trabajo de Fin de Grado (TFG) nace con la vocación de revitalizar el estudio de estas arquitecturas clásicas mediante herramientas modernas. No se trata solo de preservar la historia de la informática, sino de utilizarla como vehículo docente. El proyecto consiste en el diseño e implementación de un simulador completo del Z80, desarrollado con tecnologías de vanguardia como el lenguaje Rust y WebAssembly, para eliminar las barreras de entrada que tradicionalmente han dificultado el acceso a estos entornos de aprendizaje.

1.1. Motivación y Origen del Proyecto

El origen de este proyecto es fruto de una afortunada coincidencia entre la curiosidad técnica personal y una necesidad académica real. Inicialmente, la idea surgió del deseo de explorar el desarrollo de emuladores y profundizar en el lenguaje de programación Rust, una tecnología que está redefiniendo la programación de sistemas por sus garantías de seguridad y rendimiento.

Paralelamente, en el contexto de la asignatura de Estructura de Computadores de la Universidad Rey Juan Carlos (URJC), se detectó la necesidad de contar con una herramienta didáctica renovada. Las herramientas existentes a menudo requerían instalaciones complejas, dependían de sistemas operativos obsoletos o no ofrecían la visualización interna necesaria para que un alumno comprendiera *qué ocurre* dentro de la CPU ciclo a ciclo.

De esta convergencia surgió la propuesta: desarrollar una herramienta que no solo sirviera como ejercicio de ingeniería de software, sino que tuviera una utilidad práctica inmediata para la docencia. El objetivo era crear un simulador que un alumno pudiera abrir en su navegador web, sin instalar nada, pegar su código ensamblador y ver cómo se iluminan los registros y la memoria. Una herramienta hecha por un estudiante, para estudiantes.

1.2. Justificación Académica

El uso de simuladores en el ámbito educativo presenta ventajas indudables frente al hardware físico, especialmente en cursos introductorios:

- **Observabilidad Total:** En un chip real, el estado interno es una “caja negra”. En un simulador, podemos exponer cada registro, cada bandera y cada dirección de memoria en tiempo real, permitiendo una depuración visual que acelera el aprendizaje.
- **Seguridad y Robustez:** Un error en código ensamblador en hardware real puede colgar la máquina o requerir un reinicio físico. En un entorno simulado en Rust, el sistema es seguro, recuperable y protege al estudiante de la frustración de los fallos catastróficos.
- **Accesibilidad Universal:** Al compilar el proyecto a WebAssembly (WASM), democratizamos el acceso. Cualquier estudiante con un navegador moderno, ya sea en un PC de gama alta, un portátil modesto o incluso una tablet, puede realizar sus prácticas de laboratorio sin barreras técnicas.

1.3. Objetivos del Trabajo

Aunque el objetivo técnico es la emulación fiel del microprocesador Z80, el objetivo último es la creación de un entorno de desarrollo integrado (IDE) educativo. Esto implica:

1. **Fidelidad Funcional:** Implementar el set de instrucciones del Z80 con suficiente precisión para ejecutar algoritmos estándar y programas educativos.
2. **Experiencia de Usuario (UX) Moderna:** Proveer un editor de código con resaltado de sintaxis, ensamblado en tiempo real y controles de ejecución paso a paso.
3. **Tecnología Sostenible:** Utilizar Rust para garantizar que el código base sea mantenible, seguro y eficiente, sirviendo como ejemplo de buenas prácticas de ingeniería de software.

2

Ecosistema de Desarrollo y Tecnologías

El presente capítulo expone el conjunto de herramientas, lenguajes y bibliotecas que conforman el entorno de desarrollo del proyecto. Se detallan, en primer lugar, los fundamentos del lenguaje de programación Rust, haciendo especial hincapié en aquellas características que resultan determinantes para la construcción de un emulador de sistemas de 8 bits: el control exhaustivo de la memoria, las garantías de seguridad en tiempo de compilación y las abstracciones de coste cero. Asimismo, se describe el proceso de compilación a WebAssembly, que permite la ejecución del emulador en entornos web sin renunciar al rendimiento nativo, y se introduce el entorno de trabajo `eframe`, empleado para el desarrollo de la interfaz gráfica de usuario.

2.1. Lenguaje de Programación: Rust

Rust es un lenguaje de programación de sistemas, multiparadigma y de propósito general, desarrollado originalmente por Mozilla Research. Su diseño prioriza la seguridad de la memoria, la concurrencia sin condiciones de carrera y el rendimiento comparable al de C y C++, pero sin depender de un recolector de basura. Estas propiedades lo convierten en una elección idónea para el desarrollo de emuladores, donde la fidelidad temporal, el acceso directo a la memoria y la ausencia de latencias impredecibles son requisitos críticos.

A continuación, se examinan los pilares semánticos del lenguaje que resultan más relevantes para la comprensión de la arquitectura del emulador implementa-

do.

2.1.1. Propiedad (*Ownership*) y Préstamos (*Borrowing*)

El sistema de propiedad es, sin género de dudas, la contribución más distintiva de Rust frente a otros lenguajes de sistemas. En Rust, cada valor en memoria tiene un único propietario (*owner*) en cada instante del programa. Cuando el propietario sale del ámbito (*scope*), el valor se libera automáticamente. No se permite que existan múltiples propietarios simultáneos de un mismo recurso, lo que elimina, en tiempo de compilación, los errores de doble liberación (*double free*) y los accesos a memoria ya liberada (*use-after-free*).

Para permitir el acceso a un recurso sin transferir su propiedad, Rust define el mecanismo de préstamo (*borrowing*), materializado mediante referencias (*references*). El compilador impone, mediante el *borrow checker*, las siguientes reglas inmutables: en cualquier ámbito dado, puede existir una única referencia mutable o cualquier número de referencias inmutables, pero nunca ambas simultáneamente. Esta restricción, aunque exigente, garantiza la ausencia de aliasado mutable (*mutable aliasing*), erradicando una fuente común de comportamiento indefinido en lenguajes como C o C++.

En el contexto de un emulador, este modelo resulta particularmente ventajoso para la gestión del bus de datos y del espacio de direccionamiento. La memoria principal, los dispositivos de E/S mapeados en memoria y los registros de estado pueden modelarse como estructuras con propiedad bien definida, cediendo préstamos temporales a las distintas unidades funcionales —CPU, controlador de interrupciones, generador de vídeo— sin riesgo de corrupción de datos en tiempo de ejecución.

2.1.2. Tipado: Estructuras y Enumeraciones

Rust posee un sistema de tipos estático, fuerte y con inferencia parcial, que combina la eficiencia de los tipos de valor con la expresividad de las abstracciones algebraicas.

Las estructuras (**struct**) permiten agrupar datos relacionados bajo un único tipo. Rust distingue entre estructuras con campos nombrados, tuplas nombradas y tuplas unitarias, ofreciendo una granularidad adecuada para la modelización de componentes hardware. Por ejemplo, el estado interno de un registro del Z80 —valor, flags de condición, señales de control— puede encapsularse en una estructura con semántica de valor, garantizando que su representación en memoria sea compacta y predecible.

Figura 2.1: Ejemplo de estructura en Rust para modelar la memoria de 64 KB del Z80.

```

1 impl Mem64k {
2     pub fn new() -> Self {
3         Self { data: [0; 0x10000] }
4     }
5 }

```

Las enumeraciones (`enum`) en Rust son tipos algebraicos de datos (*algebraic data types*), capaces de albergar variantes con datos asociados de distintos tipos. A diferencia de las enumeraciones tradicionales de C, que se limitan a constantes enteras, las enumeraciones de Rust permiten modelar estados mutuamente excluyentes con carga útil (*payload*) diferenciada. Esta característica resulta idónea para representar las distintas fases del ciclo de instrucción del Z80 —obtención de código de operación, decodificación, ejecución, acceso a memoria— o para codificar los modos de direccionamiento de la CPU, donde cada variante puede contener los operandos específicos que le son propios.

Figura 2.2: Ejemplo de enumeración en Rust para modelar los registros de propósito general del Z80.

```

1 #[derive(PartialEq, Clone, Copy, Debug)]
2 pub enum GPR {
3     A,
4     F,
5     B,
6     C,
7     D,
8     E,
9     H,
10    L,
11 }

```

2.1.3. *Traits* e Interfaces

Los *traits* constituyen el mecanismo principal de abstracción polimórfica en Rust. Conceptualmente equivalentes a las interfaces de otros lenguajes, pero con semántica más rigurosa, un *trait* define un conjunto de métodos y comportamientos que un tipo concreto debe implementar para satisfacer dicho contrato.

La implementación de un *trait* es explícita (`impl TraitName for TypeName`) y puede extenderse incluso a tipos definidos en otros módulos, siempre que se respeten las reglas de coherencia (*orphan rules*). Esta propiedad, conocida como extensibilidad por *traits*, permite dotar a tipos primitivos —como los enteros sin signo de 8 bits (`u8`)— de funcionalidades específicas del dominio, como la

conversión a cadenas de representación hexadecimal o la manipulación de bits de estado.

En la arquitectura del emulador, los *traits* se emplean para definir contratos comunes entre dispositivos hardware: un *trait* `MemoryMapper` puede especificar métodos para lectura y escritura de memoria.

Figura 2.3: Ejemplo de *trait* en Rust para definir la interfaz de un dispositivo mapeado en memoria.

```

1 pub trait MemoryMapper {
2     fn read(&self, address: u16) -> u8;
3     fn write(&mut self, address: u16, data: u8);
4
5     fn read_word(&self, address: u16) -> u16 {
6         let low = self.read(address);
7         let high = self.read(address.overflowing_add(1).0);
8         ((high as u16) << 8) | (low as u16)
9     }
10
11     fn write_word(&mut self, address: u16, data: u16) {
12         self.write(address, (data & 0xFF) as u8);
13         self.write(address.overflowing_add(1).0, (data >>
14             8) as u8);
15     }
16 }

```

2.1.4. Polimorfismo: Despacho Dinámico (*Dispatching*)

Rust ofrece dos modalidades de polimorfismo: el despacho estático (*static dispatch*), basado en la monomorfización de *traits* genéricos, y el despacho dinámico (*dynamic dispatch*), materializado mediante *trait objects* (`dyn Trait`).

El despacho estático, resuelto en tiempo de compilación, genera código especializado para cada tipo concreto que satisface un *trait*. Carece de penalización en tiempo de ejecución y permite optimizaciones agresivas por parte del compilador, pero puede incrementar el tamaño del binario (*code bloat*). Por el contrario, el despacho dinámico emplea una tabla de métodos virtuales (*vtable*), permitiendo la heterogeneidad de tipos en tiempo de ejecución a costa de una indirección adicional.

En el desarrollo del emulador, el despacho dinámico resulta útil para la gestión de la memoria de la CPU o para la lista de periféricos conectados al bus.

2.1.5. Mutabilidad Interior y Punteros Inteligentes

Una de las tensiones inherentes al diseño de emuladores es la necesidad de compartir estado mutable entre múltiples subsistemas —por ejemplo, entre la CPU y el controlador de vídeo, que acceden concurrentemente a la memoria de vídeo— sin violar las reglas de exclusividad del *borrow checker*.

Rust resuelve esta tensión mediante el patrón de mutabilidad interior (*interior mutability*), que permite la modificación de datos a través de referencias inmutables, delegando la comprobación de seguridad a mecanismos de tiempo de ejecución. Los tipos principales son:

- `Cell<T>`: para tipos que implementan `Copy`, permite la mutación mediante sustitución completa del valor.
- `RefCell<T>`: realiza las comprobaciones de préstamo en tiempo de ejecución, generando un pánico en caso de violación de las reglas de aliasado.
- `Mutex<T>` y `RwLock<T>`: para entornos multihilo, proporcionan mutabilidad interior segura mediante sincronización.

Estos mecanismos se combinan frecuentemente con punteros inteligentes (*smart pointers*) que gestionan la propiedad compartida:

- `Box<T>`: propiedad exclusiva en el montículo (*heap*).
- `Rc<T>`: conteo de referencias (*reference counting*) para propiedad compartida en contextos de un único hilo.
- `Arc<T>`: variante atómica de `Rc`, segura para su uso entre múltiples hilos.

En la implementación del emulador, se utilizan `Rc<RefCell<T>` para modular punteros a tipos con cierto *trait*, como el caso de la memoria compartida (`Rc<RefCell<dyn MemoryMapper>>`), o para la lista de periféricos conectados al bus (`Vec<Rc<RefCell<dyn IODevice>>>`).

2.1.6. Metaprogramación (*Macros*)

Rust dispone de un sistema de metaprogramación basado en *macros*, que opera sobre la representación sintáctica del código fuente. Se distinguen dos categorías:

Las *macros declarativas* (`macro_rules!`) realizan sustitución de patrones mediante coincidencia de árboles de tokens, resultando útiles para la generación de

código repetitivo —por ejemplo, la definición masiva de instrucciones del Z80 con patrones de operando similares.

Las *macros procedimentales* (*procedural macros*) son funciones que transforman el *token stream* de entrada, habilitando la derivación automática de *traits* (`#[derive(...)]`) o la anotación de código con metadatos semánticos. Estas últimas se emplean extensivamente en el ecosistema de serialización y en el propio *framework* de interfaz gráfica, como se verá en la Sección 2.3.

La metaprogramación en Rust es intencionalmente más conservadora que en lenguajes como Lisp o C++, limitando la expansión de *macros* a contextos bien definidos para preservar la legibilidad y la capacidad de razonamiento local del código.

Figura 2.4: Ejemplo de macro declarativa en Rust para definir un sistema de *logs* de depuración en el emulador.

```

1 macro_rules! test_log {
2     ($self:expr, $fmt:expr $(, $($arg:tt)+)? ) => {
3         $self.test_callback.1(&format!($fmt $(, $($arg)+)?),
4                               &mut $self.test_callback.0)
5     };
6 }

```

2.2. Compilación WebAssembly (WASM)

WebAssembly (abreviado WASM) es un formato de código binario de bajo nivel, diseñado como objetivo de compilación para lenguajes de alto nivel con requisitos de rendimiento cercanos al nativo. Su modelo de ejecución se basa en una máquina virtual de pila, con un formato de instrucciones portátil y un modelo de seguridad basado en sandboxing.

La cadena de compilación de Rust, mediante el objetivo `wasm32-unknown-unknown`, permite la generación de módulos WASM independientes del sistema operativo, sin necesidad de vincularse a un entorno de ejecución específico. El compilador `rustc`, apoyado en LLVM, traduce el código fuente de Rust a bytecode WASM, preservando las optimizaciones de nivel intermedio y la eliminación de abstracciones de coste cero.

La integración de Rust con WASM resulta especialmente relevante para el presente proyecto, ya que permite desplegar el emulador como una aplicación web sin reescribir la lógica de emulación en JavaScript. El módulo compilado puede cargarse en un navegador moderno y ejecutarse a velocidades próximas a las del código nativo, aprovechando la capacidad de WASM para representar estructuras de datos compactas y operaciones aritméticas de enteros de 8 y 16

bits, fundamentales en la arquitectura del Z80.

Para la interoperabilidad entre el entorno WASM y el motor JavaScript del navegador, Rust ofrece la herramienta `wasm-bindgen`, que genera automáticamente los *bindings* necesarios para el intercambio de tipos complejos, la invocación de funciones y la manipulación del DOM. No obstante, en la arquitectura del proyecto, la interacción con el usuario se abstrae mediante el *framework* de interfaz gráfica, que gestiona de forma transparente la capa de comunicación con el entorno de ejecución web.

2.3. Integración GUI (Entorno de Trabajo `eframe`)

La interfaz gráfica de usuario del emulador se ha desarrollado sobre el entorno de trabajo `eframe`, que constituye la capa de integración nativa de la biblioteca `egui`. `egui` es una biblioteca de interfaz gráfica inmediata (*immediate mode GUI*), escrita en Rust, que prioriza la facilidad de uso, la portabilidad y el rendimiento.

El paradigma de *immediate mode* difiere del modelo de retención (*retained mode*) tradicional en que la interfaz se reconstruye completamente en cada fotograma, en lugar de mantener un árbol de widgets persistente entre ciclos de renderizado. Esta aproximación simplifica la gestión del estado de la interfaz, eliminando la necesidad de sincronizar explícitamente el modelo de datos con la representación visual, lo cual resulta particularmente conveniente para la visualización en tiempo real de registros de CPU, mapas de memoria y trazas de ejecución.

`eframe` proporciona los *backends* necesarios para la ejecución de aplicaciones `egui` en múltiples plataformas: nativamente mediante `winit` y `wgpu` (o `glow`) en escritorio, y en navegadores mediante la compilación a WASM y la integración con el canvas HTML5. Esta dualidad de despliegue permite mantener una única base de código para las versiones de escritorio y web del emulador, reduciendo la divergencia entre plataformas y facilitando el mantenimiento evolutivo del proyecto.

La elección de `eframe` y `egui` se justifica, asimismo, por su integración natural con el ecosistema de Rust, su ausencia de dependencias externas pesadas y su modelo de concurrencia compatible con el sistema de propiedad del lenguaje, evitando los problemas de seguridad de memoria habituales en el desarrollo de interfaces gráficas con acceso a bajo nivel.

3

Objetivos

El objetivo principal de este Trabajo de Fin de Grado es el diseño, desarrollo e implementación de un simulador completo de la arquitectura Zilog Z80, orientado específicamente a fines educativos y de desarrollo de software de bajo nivel.

El simulador, desarrollado en el lenguaje de programación Rust, busca ofrecer una plataforma moderna, robusta y accesible que permita a estudiantes y entusiastas comprender el funcionamiento interno de una CPU clásica de 8 bits, facilitando el aprendizaje de conceptos fundamentales de arquitectura de computadores y programación en lenguaje ensamblador sin la necesidad de hardware físico.

Los objetivos específicos que se persiguen son los siguientes:

- **Emulación fiel del núcleo Z80:** Implementar el conjunto de instrucciones completo del Z80, incluyendo los modos de direccionamiento, la gestión de banderas (flags), las interrupciones y el comportamiento de los registros, garantizando la precisión técnica necesaria para ejecutar software real.
- **Portabilidad y accesibilidad:** Desarrollar la aplicación de manera que pueda ejecutarse nativamente en los principales sistemas operativos (Windows, Linux, macOS) y, crucialmente, en navegadores web modernos mediante la compilación a WebAssembly (WASM).
- **Entorno de desarrollo integrado:** Proporcionar un editor de código ensamblador integrado que permita escribir, compilar y ejecutar programas en el mismo entorno, eliminando la fricción de utilizar herramientas externas.

- **Visualización y depuración:** Ofrecer herramientas visuales que permitan inspeccionar en tiempo real el estado de la memoria, los registros y el flujo de ejecución, facilitando la depuración y la comprensión del ciclo de instrucción.

3.1. Descripción del problema

La enseñanza de la arquitectura de computadores a menudo se beneficia del uso de modelos simplificados o arquitecturas clásicas que permiten al alumno visualizar el flujo de datos y control sin la complejidad abrumadora de los procesadores modernos (pipelines profundos, ejecución especulativa, cachés multinivel, etc.). En este contexto, las arquitecturas basadas en acumulador, como la del Zilog Z80, representan un punto de equilibrio ideal: son lo suficientemente simples para ser comprendidas en su totalidad, pero lo suficientemente complejas para ilustrar problemas reales de computación.

Sin embargo, el acceso a hardware basado en Z80 (como los ordenadores de 8 bits de los años 80 o kits de desarrollo específicos) es cada vez más difícil y costoso de mantener en un entorno académico. Además, el ciclo de desarrollo en hardware real suele ser lento y propenso a errores difíciles de diagnosticar debido a la falta de herramientas de visualización interna.

Este proyecto surge específicamente para cubrir una necesidad docente detectada: la falta de una herramienta moderna, gratuita y de fácil acceso que simule una CPU de acumulador con un entorno visual claro. La iniciativa nace de la convergencia entre el interés académico por disponer de un simulador pedagógico actualizado y el interés técnico por aplicar lenguajes de programación de sistemas modernos como Rust a la emulación de hardware.

3.2. Estudio de alternativas

Típicamente, un proyecto de ingeniería comienza con un análisis exhaustivo de las soluciones existentes en el mercado para determinar si es necesario un desarrollo a medida. En el caso de este TFG, la motivación principal no fue la inexistencia absoluta de simuladores de Z80 — existen numerosos emuladores, desde el clásico *Z80pack* hasta diversos proyectos en JavaScript o C++ —, sino la oportunidad de construir una solución específica que se alineara tanto con los requisitos docentes como con los objetivos de aprendizaje del alumno.

La decisión de desarrollar un simulador propio en lugar de adoptar uno existente se fundamenta en los siguientes puntos:

1. **Adaptación pedagógica:** Muchos emuladores existentes están orientados a la preservación de videojuegos (GameBoy, ZX Spectrum, MSX) y optimizan la velocidad o la compatibilidad de audio/video por encima de la visibilidad del estado interno de la CPU. Este proyecto prioriza la claridad: ver cómo cambian los registros, las banderas y la memoria paso a paso.
2. **Modernización tecnológica:** Gran parte del software educativo existente para esta arquitectura es “legacy”, escrito hace décadas, con interfaces obsoletas o dependencias de sistemas operativos antiguos. La creación de una nueva herramienta permite utilizar tecnologías actuales (Rust, WebAssembly) que garantizan seguridad de memoria, rendimiento y ejecución en la web sin instalaciones.
3. **Requisito académico:** El proyecto responde a una demanda concreta del entorno docente, donde se requería específicamente un emulador de una CPU de acumulador que pudiera ser modificado, extendido y utilizado libremente por los alumnos sin barreras de entrada.

Por tanto, más que una selección entre alternativas competidoras, este proyecto representa la creación de una herramienta a medida diseñada para llenar un hueco específico en la docencia de arquitectura de computadores mediante tecnología moderna.

3.3. Metodología empleada

Para el desarrollo del simulador se ha seguido una metodología de desarrollo iterativa e incremental, dividiendo el proyecto en módulos funcionales que pueden ser probados y validados de forma independiente.

3.3.1. Tecnologías y herramientas

- **Lenguaje de programación:** Se ha seleccionado **Rust** por sus garantías de seguridad de memoria sin recolector de basura, su sistema de tipos robusto y su excelente soporte para WebAssembly. Esto permite prevenir errores comunes en la emulación (como accesos inválidos a memoria) en tiempo de compilación.
- **Interfaz gráfica:** Se utiliza **egui**, una biblioteca de modo inmediato (*immediate mode*) para Rust, que es ligera, portátil y se integra fácilmente con WebGL para el renderizado web.

- **Control de versiones:** Se ha empleado **Git** para la gestión del código fuente, permitiendo un seguimiento detallado de los cambios y facilitando la experimentación mediante ramas.

3.3.2. Fases de desarrollo

El trabajo se ha estructurado en las siguientes etapas:

1. **Implementación del núcleo (Core):** Modelado de los registros, la memoria y la unidad aritmético-lógica (ALU). Implementación del ciclo *fetch-decode-execute* básico.
2. **Decodificación de instrucciones:** Implementación progresiva del conjunto de instrucciones del Z80 (carga, aritmética, saltos, manipulación de bits), validando cada grupo con pruebas unitarias.
3. **Implementación incremental de instrucciones:** Se ha seguido un enfoque incremental, implementando primero las instrucciones más básicas (movimientos de datos, operaciones aritméticas simples) y luego avanzando hacia instrucciones más complejas (manipulación de bits, operaciones de E/S, saltos, manipulación del stack).
4. **Desarrollo del Ensamblador:** Creación de un módulo capaz de traducir texto mnemónico a código máquina binario, gestionando etiquetas y directivas básicas.
5. **Integración de la Interfaz (GUI):** Construcción de las ventanas de visualización (memoria, desensamblado, registros) y los controles de interacción.
6. **Gestión de interrupciones:** Implementación del sistema de interrupciones del Z80, incluyendo modos de interrupción y manejo de eventos externos. La existencia de la UI facilita crear dispositivos E/S (Input/Output) simulados que generen interrupciones, como un temporizador o un teclado virtual.
7. **Adaptación a WebAssembly:** Ajuste de dependencias y optimización para asegurar un rendimiento fluido en navegadores, incluyendo la gestión de eventos y la persistencia de datos local.

3.4. Evolución del Proyecto y Retos Tecnológicos

El desarrollo de este simulador ha seguido un proceso evolutivo caracterizado por la resolución progresiva de desafíos de ingeniería de creciente complejidad. A

lo largo de la implementación, se han abordado problemas críticos relacionados con la fidelidad de la emulación, la arquitectura de software concurrente y la optimización para entornos restringidos como la web. A continuación, se detallan los principales hitos técnicos y las soluciones adoptadas durante el ciclo de vida del proyecto.

3.4.1. Implementación del Núcleo y Aritmética BCD

La fase inicial se centró en la construcción de la arquitectura base del sistema: el banco de registros y el ciclo de instrucción básico. Uno de los primeros y más significativos desafíos fue la correcta implementación de la Unidad Aritmético-Lógica (ALU), específicamente en lo referente al Registro de Banderas (*Flags*).

El microprocesador Z80 posee un comportamiento aritmético sutil, particularmente en operaciones de *Decimal Adjust* (instrucción DAA) utilizadas para aritmética BCD (*Binary Coded Decimal*). La correcta ejecución de DAA depende no solo del valor en el acumulador, sino del estado previo de las banderas de Acarreo (C), Acarreo Medio (H) y Resta (N). Esto obligó a una revisión exhaustiva de todas las instrucciones aritméticas y lógicas previas (ADD, SUB, INC, DEC) para garantizar que actualizaran correctamente estas banderas ocultas, que a menudo son ignoradas en emuladores de menor precisión. La validación de este comportamiento requirió la creación de baterías de pruebas unitarias específicas que comparaban el estado del simulador contra trazas de ejecución de hardware real.

Asimismo, la implementación de saltos relativos (JR) introdujo la complejidad de la aritmética de punteros con signo en complemento a dos, vital para que el cálculo de direcciones de memoria en bucles y condicionales fuese exacto al byte.

3.4.2. Adaptación a Arquitectura Guiada por Eventos

Una vez estabilizado el núcleo, el proyecto pivotó hacia la integración de la interfaz gráfica. Este paso implicó un cambio de paradigma arquitectónico fundamental. Los emuladores de consola tradicionales operan en un bucle infinito bloqueante. Sin embargo, la integración con `egui` y el entorno web requería una arquitectura no bloqueante y cooperativa.

El desafío técnico residió en desacoplar el ciclo de reloj de la CPU de la frecuencia de refresco de la pantalla. Si la CPU ejecutara una instrucción por cuadro, el sistema sería inusablemente lento. La solución adoptada fue la implementación de un “presupuesto de ciclos” (*cycle budget*): el emulador calcula cuántos ciclos de CPU deben transcurrir en el tiempo real entre dos cuadros de video (aproximadamente 16.6 ms) y ejecuta ese lote de instrucciones de manera síncrona

antes de ceder el control al motor de renderizado. Esto garantiza una velocidad de emulación realista manteniendo la interfaz de usuario fluida y receptiva.

3.4.3. Diseño del Ensamblador de Dos Pasadas

Con la introducción de capacidades de edición de código, surgió la necesidad de un ensamblador robusto. Inicialmente concebido como un traductor línea a línea, pronto se evidenciaron las limitaciones de este enfoque, especialmente al manejar etiquetas (*labels*) y saltos hacia adelante (referencias a código que aún no ha sido procesado).

Para resolver esto, se rediseñó el módulo ensamblador siguiendo una arquitectura clásica de dos pasadas:

- **Primera pasada (Resolución de Símbolos):** El ensamblador recorre el código fuente calculando el tamaño en bytes de cada instrucción y construyendo una tabla de símbolos que mapea los nombres de las etiquetas a direcciones de memoria concretas.
- **Segunda pasada (Generación de Código):** Se vuelve a recorrer el código, esta vez sustituyendo las referencias a etiquetas por las direcciones calculadas y generando el código máquina final.

Este rediseño permitió además la implementación de directivas complejas como `ORG` (para reubicar código en memoria) y definiciones de datos (`DB`, `DW`), dotando al simulador de características propias de un IDE profesional.

3.4.4. Sincronización del Sistema de Interrupciones

Quizás el componente más complejo de modelar fue el sistema de interrupciones del Z80, debido a la estricta precisión temporal requerida. El Z80 muestrea la línea de interrupción únicamente en el último ciclo de máquina de la instrucción en curso. Además, existen interacciones críticas de temporización, como la latencia de las instrucciones `EI` (Enable Interrupts) y `DI` (Disable Interrupts), que no tienen efecto inmediato, sino que se aplican tras la ejecución de la *siguiente* instrucción.

La implementación del soporte para el **Modo de Interrupción 2 (Vectorizado)** presentó dificultades particulares. Este modo requiere una coordinación exacta entre la CPU y el dispositivo periférico simulado para colocar un vector de interrupción en el bus de datos en el momento preciso. Se detectaron y corrigieron condiciones de carrera donde las interrupciones anidadas o la coincidencia de una interrupción enmascarable (`INT`) con una no enmascarable (`NMI`) podían

dejar la CPU en un estado inconsistente. La solución implicó refactorizar el bucle principal `step()` para gestionar explícitamente las prioridades y los estados de inhibición de interrupciones.

3.4.5. Abstracción de Plataforma y Almacenamiento

Un requisito clave era la portabilidad a WebAssembly. Esto presentó un desafío estructural: el acceso al sistema de archivos. Rust estándar (`std::fs`) no está disponible en un entorno de navegador web por razones de seguridad.

Para superar esto, se implementó un patrón de diseño *Adapter* para la capa de persistencia. Se definió un *trait* abstracto de almacenamiento que posee dos implementaciones concretas:

1. Una implementación nativa que utiliza el sistema de archivos del sistema operativo para guardar y cargar proyectos.
2. Una implementación web que interactúa con el *Local Storage* del navegador, permitiendo persistir el código del usuario entre sesiones sin necesidad de un servidor backend.

3.4.6. Cobertura del Set de Instrucciones Extendido

Finalmente, para garantizar una compatibilidad total con software legado, fue necesario implementar la totalidad del set de instrucciones, incluyendo aquellas con prefijos de manipulación de bits (CB) y registros de índice (IX, IY).

La combinatoria de prefijos (por ejemplo, instrucciones que usan DDCB para rotar un bit en una dirección indexada por IX) amenazaba con explotar la complejidad del código fuente, requiriendo cientos de variantes de funciones casi idénticas. Para mitigar esto, se refactorizó el sistema de decodificación utilizando intensivamente el sistema de *Traits* y genéricos de Rust. Esto permitió escribir la lógica de las instrucciones una sola vez y parametrizarla según el registro destino o el modo de direccionamiento, manteniendo el código base mantenible y libre de redundancias (principio DRY - *Don't Repeat Yourself*).

4

Descripción informática

El microprocesador Zilog Z80 es una unidad central de procesamiento (CPU) de 8 bits diseñada y fabricada por Zilog, lanzada al mercado en julio de 1976. Concebido como una mejora compatible con el Intel 8080, el Z80 se convirtió rápidamente en una de las CPUs más populares y versátiles de la historia de la computación, impulsando desde ordenadores personales como el ZX Spectrum o el Amstrad CPC hasta sistemas embebidos, consolas de videojuegos y equipos industriales que continúan en uso hoy en día.

Su éxito radica en un diseño pragmático que abordaba las limitaciones de sus predecesores: requería una única fuente de alimentación de +5V (frente a las tres del 8080), incorporaba un mecanismo de refresco de memoria DRAM integrado y ofrecía un conjunto de instrucciones significativamente ampliado con capacidades de manipulación de bits y bloques de memoria.

4.1. Arquitectura Interna

El Z80 es una máquina de arquitectura CISC (Complex Instruction Set Computer) basada en acumulador. Internamente, opera con un bus de datos de 8 bits y un bus de direcciones de 16 bits, lo que le permite direccionar directamente hasta 64 KB (2^{16} bytes) de memoria.

4.1.1. Modelo de Registros

Una de las características distintivas del Z80 es su masivo banco de registros, mucho mayor que el de sus contemporáneos. Los registros se dividen en dos conjuntos: el banco principal y el banco alternativo (shadow registers), además de varios registros de propósito especial e índices.

Registros de Propósito General

El Z80 dispone de un acumulador (A) y seis registros auxiliares de 8 bits (B, C, D, E, H, L). Estos pueden utilizarse individualmente para operaciones de 8 bits o agruparse en pares de 16 bits (BC, DE, HL) para direccionamiento de memoria y aritmética de 16 bits.

- **Acumulador (A):** Es el destino principal de todas las operaciones aritméticas y lógicas de 8 bits.
- **Registro de Banderas (F):** Contiene los indicadores de estado que resultan de la última operación de la ALU (Zero, Carry, Sign, etc.).
- **Par HL:** El par de registros H y L actúa como el “acumulador de direcciones” primario de la máquina. Muchas instrucciones utilizan HL como un puntero indirecto a memoria (M), permitiendo un acceso eficiente a estructuras de datos.
- **Pares BC y DE:** Utilizados frecuentemente como contadores (B/C) o punteros auxiliares de datos.

Banco de Registros Alternativo

El Z80 duplica el conjunto de registros generales (A', F', B', C', D', E', H', L'). Estos registros “sombra” no son accesibles directamente por la mayoría de las instrucciones. En su lugar, dos instrucciones especiales de intercambio (**EX AF**, **AF'** y **EXX**) permiten conmutar instantáneamente entre el banco principal y el alternativo.

Esta característica fue revolucionaria para el manejo de interrupciones: en lugar de guardar laboriosamente el estado de la CPU en la pila (stack), una rutina de interrupción podía simplemente ejecutar **EXX** y **EX AF**, **AF'**, disponer de un juego “limpio” de registros y restaurarlos al finalizar con un coste de ciclos mínimo.

Registros de Índice (IX e IY)

Dos registros de 16 bits, IX e IY, permiten un modo de direccionamiento indexado. Las instrucciones pueden acceder a memoria en la dirección $(IX + d)$ o $(IY + d)$, donde d es un desplazamiento de 8 bits con signo. Esto facilita enormemente la implementación de estructuras de datos complejas como arrays, registros de activación de funciones o tablas, sin necesidad de recalcular direcciones base manualmente.

Registros de Control y Especiales

- **Program Counter (PC):** Registro de 16 bits que contiene la dirección de memoria de la próxima instrucción a ejecutar. Se incrementa automáticamente tras cada fetch.
- **Stack Pointer (SP):** Registro de 16 bits que apunta a la cima de la pila en memoria. La pila crece hacia direcciones decrecientes (descendente).
- **Registro de Interrupciones (I):** Utilizado en el Modo de Interrupción 2 para formar la parte alta de la dirección del vector de interrupción.
- **Registro de Refresco (R):** Un contador de 7 bits que se incrementa automáticamente tras cada ciclo de búsqueda de instrucción (opcode fetch). Su propósito original era generar direcciones para refrescar las memorias DRAM dinámicas de la época sin hardware externo adicional. Curiosamente, en software moderno se utiliza a menudo como fuente de entropía para generadores de números pseudoaleatorios.

4.1.2. La Unidad Aritmético-Lógica (ALU) y el Registro de Banderas

La ALU es el corazón de cálculo del Z80. Realiza operaciones de suma, resta, AND, OR, XOR, rotaciones, desplazamientos y comparaciones. Aunque es una ALU de 4 bits internamente en algunas implementaciones físicas, funcionalmente se comporta como una unidad de 8 bits.

El registro de Banderas (F) es fundamental para el control de flujo condicional. Sus bits tienen los siguientes significados:

- **S (Signo, bit 7):** Copia del bit más significativo del resultado (MSB). Indica si un número es negativo en complemento a dos.
- **Z (Cero, bit 6):** Se activa (1) si el resultado de la operación es cero.

- **Y (bit 5) y X (bit 3)**: Bits no documentados oficialmente, generalmente copias de los bits 5 y 3 del resultado.
- **H (Half Carry, bit 4)**: Indica un acarreo del bit 3 al bit 4. Es esencial para la instrucción **DAA** (Decimal Adjust Accumulator) para corregir operaciones en formato BCD (Binary Coded Decimal).
- **P/V (Paridad/Desbordamiento, bit 2)**: Tiene doble función. En operaciones lógicas indica la paridad del resultado (número par de bits a 1). En operaciones aritméticas indica desbordamiento (overflow) de complemento a dos.
- **N (Add/Subtract, bit 1)**: Indica si la última operación fue una resta. Usado exclusivamente por **DAA** para distinguir si debe corregir una suma o una resta BCD.
- **C (Carry, bit 0)**: Indica desbordamiento en aritmética sin signo (acarreo del bit 7). También actúa como acumulador de bits en operaciones de rotación.

4.2. Ciclo de Instrucción y Temporización

El Z80 ejecuta instrucciones en una secuencia estricta de ciclos de máquina (M-cycles). Cada ciclo de máquina consta de varios ciclos de reloj (T-states).

4.2.1. Secuencia Fetch-Decode-Execute

1. **M1 (Opcode Fetch)**: Es el ciclo más importante. La CPU coloca el contenido del PC en el bus de direcciones y activa la señal de lectura. El byte leído se interpreta como el código de operación. Este ciclo incluye dos estados T extra para el refresco de memoria transparente.
2. **Decodificación**: El byte leído determina la longitud de la instrucción y si se requieren operandos adicionales (lectura de memoria M2, M3...).
3. **Ejecución**: La ALU procesa los datos, modifica registros o escribe resultados en memoria/IO.

Las instrucciones varían en duración desde 4 ciclos de reloj (como **NOP** o **LD A, B**) hasta 23 ciclos o más para instrucciones complejas como **EX (SP)**, **HL** o manipulaciones de bits indexadas.

4.3. Sistema de Interrupciones

El Z80 posee un sistema de interrupciones versátil y potente, diseñado para sistemas de tiempo real y control.

4.3.1. Interrupción No Enmascarable (NMI)

La NMI tiene prioridad absoluta y no puede ser deshabilitada por software. Al activarse, la CPU finaliza la instrucción actual, empuja el PC a la pila y salta incondicionalmente a la dirección `0x0066`. Es típica para eventos críticos como fallos de alimentación o botones de reset.

4.3.2. Interrupciones Enmascarables (INT)

Controladas por los flip-flops de habilitación de interrupción (IFF1 e IFF2) y las instrucciones `EI` (Enable Interrupts) y `DI` (Disable Interrupts). El Z80 soporta tres modos de respuesta programables (instrucción `IM 0/1/2`):

- **Modo 0 (8080 Compatible):** El dispositivo que interrumpe coloca una instrucción en el bus de datos (generalmente `RST`), que la CPU ejecuta.
- **Modo 1 (Call `0x0038`):** La CPU salta automáticamente a la dirección `0x0038`. Es el modo más simple y el usado por defecto en muchos sistemas como el ZX Spectrum y la Master System.
- **Modo 2 (Vectorizado):** El modo más potente. El dispositivo coloca un byte (vector) en el bus. La CPU combina este byte con el contenido del registro `I` para formar una dirección de 16 bits que apunta a una tabla de vectores en memoria, desde donde lee la dirección real de la rutina de servicio. Esto permite vectorizar interrupciones para múltiples periféricos con rutinas independientes.

4.4. Espacio de Entrada/Salida (E/S)

A diferencia de arquitecturas que utilizan E/S mapeada en memoria (como el 6502), el Z80 mantiene un espacio de direcciones de E/S separado de 64 KB, accedido mediante las instrucciones `IN` y `OUT`. Aunque el bus de direcciones permite 16 bits de dirección de puerto (usando `BC`), muchas implementaciones simplificadas usan solo los 8 bits bajos, proporcionando 256 puertos. Este aislamiento libera todo el espacio de memoria para RAM y ROM.

4.5. Set de Instrucciones Extendido

Aunque compatible con el 8080, el Z80 expande el set de instrucciones de su predecesor utilizando prefijos de byte (0xCB, 0xED, 0xDD, 0xFD).

- **Manipulación de Bits:** Instrucciones como SET, RES y BIT permiten establecer, resetear o probar cualquier bit individual de cualquier registro o posición de memoria.
- **Operaciones de Bloque:** Instrucciones como LDIR (Load Increment Repeat) o CPIR (Compare Increment Repeat) permiten mover o buscar bloques de memoria enteros con una sola instrucción, ejecutándose como bucles hardware ultra-rápidos.

Esta riqueza de instrucciones permitió a los programadores de la época escribir código extremadamente compacto y eficiente, exprimiendo al máximo los limitados recursos de memoria de los sistemas de 8 bits.

5

Desarrollo e Implementación

Nota sobre los fragmentos de código: A lo largo de este y posteriores capítulos se incluyen diversas piezas de código fuente extraídas directamente del núcleo del proyecto. Para facilitar su comprensión, propósitos de demostración y mejorar su encaje en el formato del documento, algunos de estos fragmentos han sido ligeramente simplificados o aislados de su contexto general, y sus comentarios originales en inglés han sido traducidos al español.

5.1. Diseño del Sistema

El diseño de la aplicación sigue una arquitectura modular y orientada a eventos, implementada en el lenguaje de programación Rust. La principal preocupación del diseño ha sido la separación estricta entre el núcleo de emulación (la lógica de la CPU y la memoria) y la capa de presentación (la interfaz gráfica), para garantizar que la simulación pueda ejecutarse de manera determinista y eficiente independientemente de la plataforma de visualización.

5.1.1. Estructura de Módulos

El proyecto se organiza en una jerarquía de módulos que refleja esta separación de responsabilidades:

- `src/`

- **assembler/**: Contiene la lógica del ensamblador de dos pasadas.
 - **mod.rs**: Punto de entrada del ensamblador y analizador sintáctico (parser).
 - **instructions.rs**: Definiciones de mnemónicos y modos de operación.
- **components/**: Abstracciones de hardware.
 - **memories/**: Implementaciones de memoria (RAM plana de 64K).
- **cpu/**: El núcleo de la emulación.
 - **mod.rs**: Estructura principal Z80A y bucle de ejecución.
 - **alu/**: Módulo de la Unidad Aritmético-Lógica.
 - ◊ **mod.rs**: Operaciones aritméticas generales.
 - ◊ **rot.rs**: Operaciones de rotación y desplazamiento de bits.
 - **tests/**: Batería de pruebas unitarias.
- **gui/**: Interfaz gráfica basada en **egui**.
 - **mod.rs**: Lógica de la aplicación **eframe**, gestión de ventanas y bucle de eventos.
 - **highlighting.rs**: Motor de resaltado de sintaxis para el editor de código.
- **main.rs**: Punto de entrada de la aplicación. Configura el entorno (nativo o web) e inicia el bucle de eventos.
- **traits.rs**: Definición de *traits* (interfaces) fundamentales como **MemoryMapper** o **IODevice**, que permiten conectar componentes abstractos.
- **ui_traits.rs**: Traits específicos para componentes que requieren representación visual.

5.1.2. Arquitectura y Diagrama de Clases

Para ilustrar de forma concisa el modelado de componentes del ecosistema y su fuerte abstracción cruzada, a continuación se proporciona el diagrama UML de las estructuras principales que interactúan tanto en el núcleo emulado como en la vista (GUI).

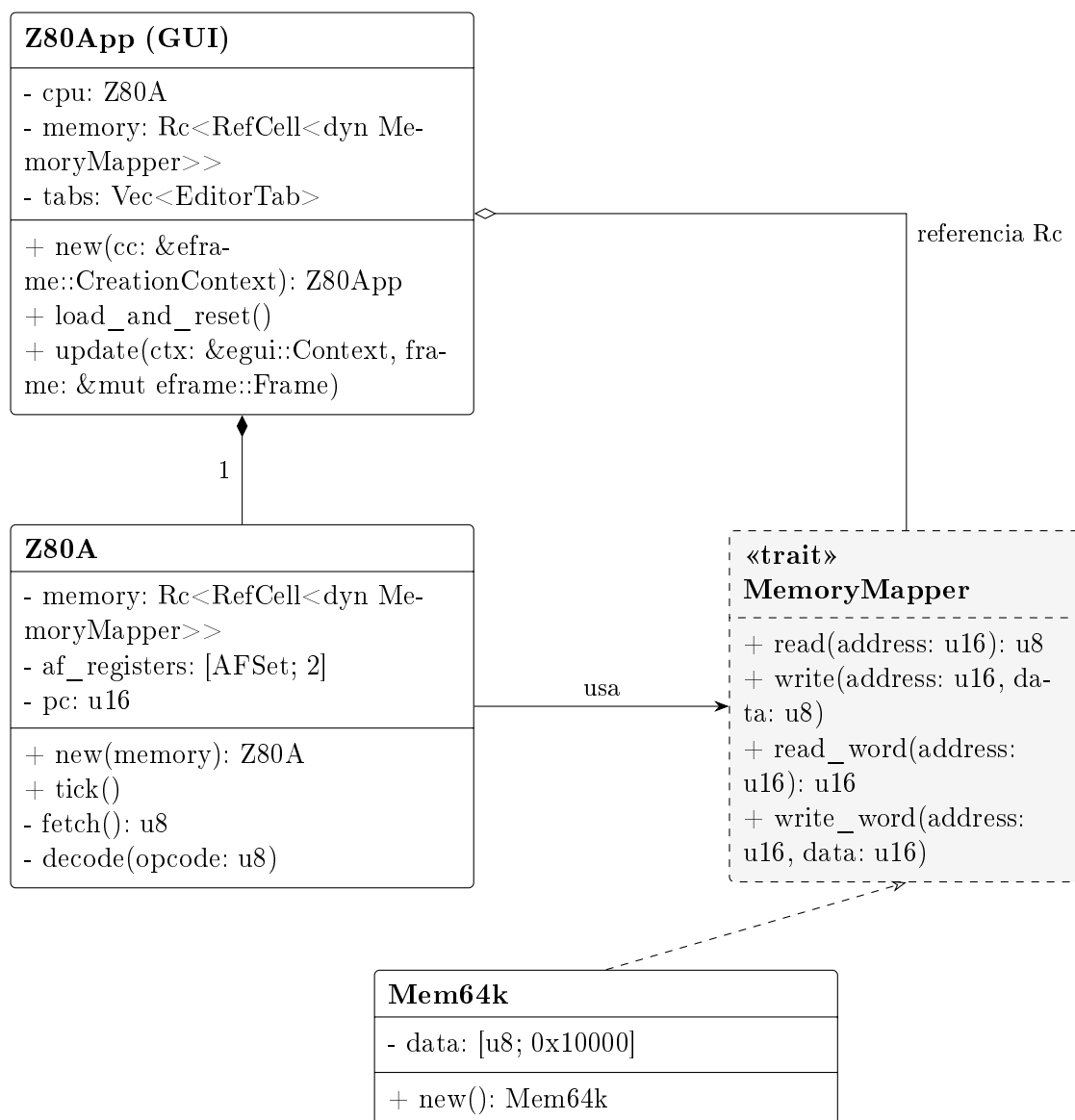


Figura 5.1: Diagrama de Clases UML estructural del Simulador Z80, mostrando la arquitectura de la capa de interfaz y la de emulación compartiendo el bus de contexto de memoria.

El diagrama de clases ilustra con claridad las características en torno al sistema de *Ownership* del ecosistema de Rust. **Z80App** contiene los hilos referenciales persistentes (**Rc**) que componen el estado emulado: construye en cascada la estructura de ejecución **Z80A** agregándola como CPU del ecosistema y pasándole el hilo anudado de mutabilidad interior (**Rc<RefCell<dyn MemoryMapper>>**). De este modo, la capa principal (App) puede interceptar e inspeccionar puramente la memoria durante pausas para dibujar a 60 FPS la pantalla gráfica, mientras que el núcleo (**Z80A**) invoca primitivas mutables enlazando con la base abstracta

sin dependencia directa del hardware RAM instanciado por debajo (`Mem64k`).

5.1.3. Diseño de Datos y Modelo de Registros

Para representar el estado interno de la CPU y la categorización de sentencias lógicas en el sistema de decodificación, se hace uso extensivo de las enumeraciones (`enum`) de Rust con tipado fuerte. Esto previene sobrecargas en la capa de ejecución mitigando primitivas ambiguas y organizando los campos de acción (`Condition`, `flags` de la ALU, y registros destinos `RegOps`).

A continuación, el diagrama de clases muestra esta anatomía con las agrupaciones principales:

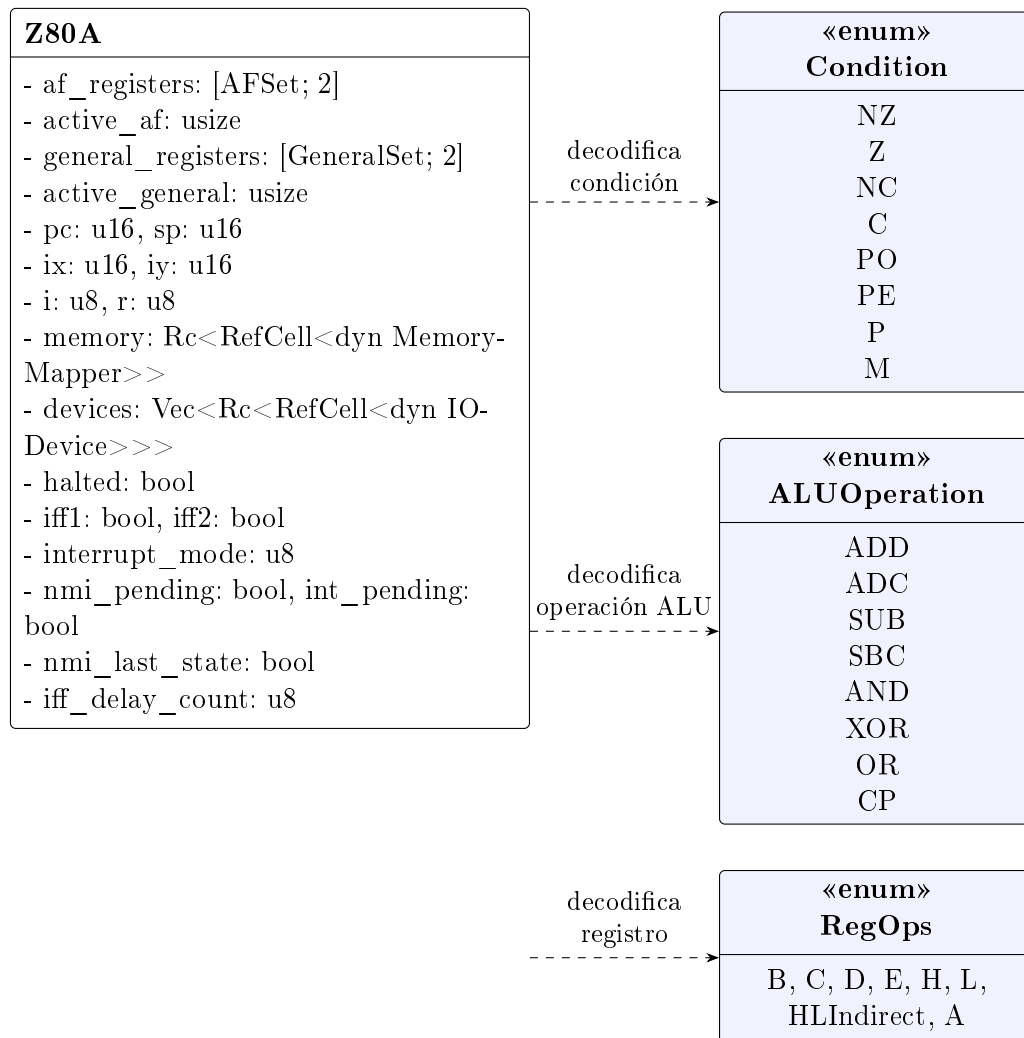


Figura 5.2: Diagrama de Clases y Enumeraciones que componen el estado interno de la CPU Z80A y el sistema de decodificación tipado.

La memoria principal del sistema no se modela dinámicamente, sino que se proyecta sobre un bloque estático de 64 KB encapsulado en la estructura `Mem64k`, que gestiona un array en bruto de un solo bloque (`[u8; 0x10000]`). Esta estructura implementa el *trait* `MemoryMapper`, abstrayendo el medio físico subyacente para el resto de componentes. Su implementación es sencilla pero vital:

Figura 5.3: Implementación de `Mem64k` con arreglo de 64KB.

```

1  use crate::traits::MemoryMapper;
2
3  pub struct Mem64k {
4      data: [u8; 0x10000],
5  }
6
7  impl Mem64k {
8      pub fn new() -> Self {
9          Self { data: [0; 0x10000] }
10     }
11 }
12
13 impl MemoryMapper for Mem64k {
14     fn read(&self, address: u16) -> u8 {
15         self.data[address as usize]
16     }
17
18     fn write(&mut self, address: u16, data: u8) {
19         self.data[address as usize] = data;
20     }
21 }

```

A la hora de integrar esta memoria en el procesador junto con el resto de componentes del bus, la estructura de la CPU mantiene referencias a través de *Smart Pointers* (punteros inteligentes):

Figura 5.4: Estructura base de la CPU instanciando la memoria polimórfica.

```

1  pub struct Z80A {
2      // ... registros, banderas y estado interno del
3      // procesador ...
4      pub memory: Rc<RefCell<dyn MemoryMapper>>,
5      pub devices: Vec<Rc<RefCell<dyn IODevice>>>,
6  }

```

Esta formulación (`Rc<RefCell<dyn MemoryMapper>`) permite la mutabilidad compartida entre la propia CPU y otros subsistemas de la simulación (como el IDE integrado o el analizador temporal) desde distintos puntos, algo normalmente bloqueado por el *Borrow Checker* estricto de Rust. Es esta organización la que permite que, por ejemplo, la interfaz gráfica lea y escriba en la memoria simula-

da durante las pausas del procesador (*breakpoints*), comportándose ambos como dueños independientes del mismo espacio de direcciones virtual.

5.1.4. Polimorfismo mediante Despacho Dinámico (*Dynamic Dispatch*)

A nivel arquitectónico, un aspecto clave de las definiciones anteriores es el uso sistemático de la palabra clave `dyn`, visible en los constructos `dyn MemoryMapper` y `dyn IODevice`. El lenguaje Rust ofrece dos aproximaciones para implementar polimorfismo basado en *traits* (interfaces): el despacho estático (*Static Dispatch*) mediante el uso de genéricos, y el despacho dinámico (*Dynamic Dispatch*) mediante objetos de rasgo o *Trait Objects*.

Si la estructura principal de la CPU se hubiera implementado recurriendo al despacho estático, su definición sería similar a:

Figura 5.5: Implementación alternativa descartada usando Despacho Estático.

```
1 pub struct Z80A<M: MemoryMapper> {
2     pub memory: Rc<RefCell<M>>,
3     // ...
4 }
```

Bajo este modelo, conocido por su alto rendimiento a través de la *monomorfización* durante la compilación, el procesador se ligaría irremediablemente a un solo tipo concreto de memoria perdiendo capacidad adaptativa. Pero las penalizaciones arquitectónicas de dicho enfoque se constatan principalmente al organizar la matriz de periféricos E/S.

Puesto que la colección estándar vectorial (`Vec<T>`) impone que todos sus elementos sean atómicamente del mismo tamaño y tipo `T` en una región contigua, si utilizásemos *Static Dispatch* no nos sería posible construir un bus de periféricos en el que convivan dispositivos de distinta naturaleza. Estaríamos relegados a programar un `Vec` exclusivo para controladores de pantalla, otro distinto para teclados y sucesivos vectores singulares, creando combinaciones rígidas incompatibles con la abstracción necesaria de un bus de expansión emulado de los clásicos ordenadores de 8-bits.

Para solventar esto, se ha optado por implementar el **Despacho Dinámico**. Al envolver las plantillas lógicas bajo tipos `dyn Trait`, el compilador maneja memorias y dispositivos como *Trait Objects* opacos. Un *Trait Object* a bajo nivel se implementa como un puntero “gordo” o envoltorio (*Fat Pointer*) que aglutina:

1. Un puntero nativo que direcciona los datos concretos del componente a instanciar guardado en el *heap*.

2. Un puntero apuntando hacia su correspondiente *Virtual Method Table* (**vtable**) o Tabla de Métodos Virtuales.

Dentro de esta *vtable* se recogen las direcciones en código máquina de los métodos declarados por cada clase subyacente. Durante la emulación, cuando el Z80 ejecuta la lectura del entorno mediante una invocación a `device.read_in(port)`, la CPU no conoce ni la estructura, ni el tamaño, ni la clase particular del dispositivo contactado; simplemente extrae la implementación correcta de lectura consultando su desvío alojado desde la propia *vtable*.

Esta estrategia de despacho dinámico penaliza marginalmente los ciclos de ejecución intrínsecos de CPU debido a la indirección doble del salto de puntero, a cambio de proporcionar una ductilidad y modularidad indispensables en una arquitectura de simulador. Gracias a su implementación:

- Habilita la estructuración de colecciones heterogéneas. El diseño unifica todos los implementadores periféricos (*pantallas auxiliares 7-segmentos, registros seriales, emuladores de cartuchos ROM...*) de forma polimórfica en iteraciones de lista sencillas `Vec<...<dyn IODevice>>`.
- Introduce un paradigma *Plug and Play*: El usuario del visualizador en entorno GUI o web, está facultado para inicializar, acoplar y desvincular periféricos del puerto local sobre la marcha en mitad del proceso sin precisar recompilar todo el aplicativo.
- Simplifica profundamente las pruebas automatizadas (tests de inyección de dependencias) pudiendo permutar entre una memoria real asignada de bloque entero de 64K, y objetos lógicos como `MockMemory` con instrumentales que rastrean los vectores de lectura.

5.2. Implementación y Detalles Técnicos

La implementación del simulador se sustenta en las capacidades de metaprogramación y seguridad de memoria de Rust. Se ha optado por una arquitectura donde la CPU es dueña de sus datos, pero accede a la memoria y periféricos a través de *smart pointers* compartidos (`Rc<RefCell<...>`), lo que modela fielmente la naturaleza compartida del bus de un ordenador.

5.2.1. El Bucle de Instrucción Principal

El núcleo del emulador reside en el ciclo *fetch-decode-execute*. A diferencia de implementaciones simplistas, este diseño considera los ciclos de reloj (T-states)

para una simulación temporalmente correcta, aunque no sea *cycle-accurate* a nivel de bus.

El bucle principal de ejecución, simplificado, sigue esta estructura:

Figura 5.6: Funciones principales del bucle de ejecución *Fetch-Decode-Execute*.

```

1  // 1. Tick:
2  fn tick(&mut self){
3      ...
4      let opcode = self.fetch();
5      self.decode(opcode);
6      ...
7  }
8
9  // 2. Decodificación de prefijo:
10 fn decode(&mut self, opcode: u8) -> () {
11     match opcode {
12         0xCB => {
13             let op = self.fetch();
14             self.decode_cb(op)
15         }
16         0xED => {
17             let op = self.fetch();
18             self.decode_ed(op)
19         }
20
21         0xDD => {
22             let op = self.fetch();
23             self.decode_dd(op)
24         }
25         0xFD => {
26             let op = self.fetch();
27             self.decode_fd(op)
28         }
29
30         _ => self.decode_unprefixed(opcode, PrefixAddressing
31             ::HL),
32     }
33
34 // 3. Decodificación de instrucción y ejecución:
35 fn decode_unprefixed(&mut self, opcode: u8, addressing:
36 PrefixAddressing) -> () {
37     ...
38     let (x, y, z, p, q) = decode_opcode(opcode);
39     match x {
40         0 => match z {
41             0 => match y {

```

```

42 // Saltos relativos y operaciones misceláneas
43 0 => test_log!(self, "NOP"), // NOP
44
45 1 => {
46     test_log!(self, "EX AF, AF'");
47     self.ex_af_af_prime();
48 }
49 }
50 }
51 }
52 }
    
```

Para visualizar la dinámica temporal expuesta en el código anterior, desde el entorno macro que desencadena su llamada en cada fotograma hasta la interacción concreta del emulador con la abstracción sobre el bus, se presenta a continuación el diagrama de secuencia.

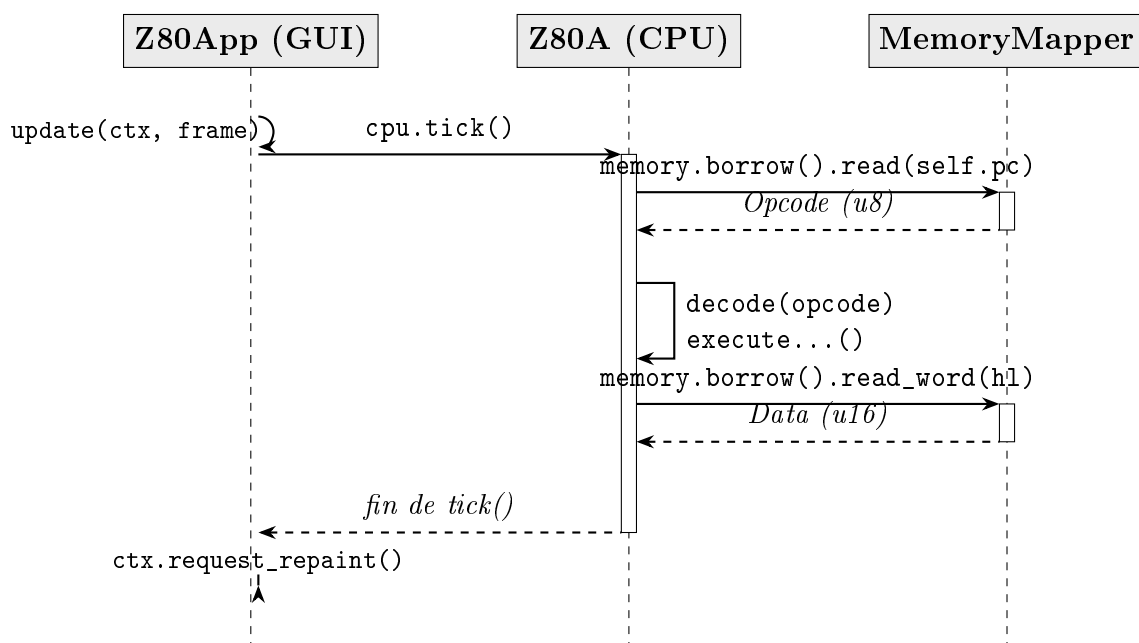


Figura 5.7: Diagrama de Secuencia del bucle de aplicación y la interacción con la rutina de ejecución `tick()` de la CPU, revelando el aislamiento entre la abstracción del hardware `MemoryMapper` y el renderizado UI.

El Z80 utiliza una arquitectura CISC donde ciertas instrucciones requieren prefijos para ampliar el set de instrucciones base del Intel 8080. El emulador maneja esto mediante funciones dedicadas para cada prefijo:

- **Instrucciones sin prefijo:** Se decodifican normalmente, pero pueden verse

afectadas por el contexto de prefijos anteriores (por ejemplo, una instrucción sin prefijo después de un DD se interpreta como operando sobre IX).

- **Prefijo CB (Bit Instructions):** Desplaza, rota o prueba bits individuales. Al encontrar 0xCB, el emulador lee el siguiente byte y lo interpreta como una instrucción de manipulación de bits.
- **Prefijos ED (Extended Instructions):** Instrucciones avanzadas como LDI, CPI o E/S de bloque.
- **Prefijos DD/FD (Index Registers IX/IY):** Modifican la instrucción siguiente para que opere sobre los registros de índice IX o IY en lugar de HL, a menudo añadiendo un byte de desplazamiento (*displacement*) d .
- **Prefijos DDCB/FDCB:** Instrucciones con prefijo CB pero indexadas con IX o IY, respectivamente

Esta estructura recursiva en la decodificación permite mantener el código limpio y organizado por grupos funcionales.

5.2.2. Mecanismo Estructurado de Decodificación (Base Octal)

Por diseño, el repertorio de instrucciones del Z80 no es aleatorio en absoluto, sino que está pensado para minimizar la complejidad del circuito decodificador en silicio. A nivel interno, el *opcode* de 8 bits se divide convencionalmente no en mitades hexadecimales, sino en combinaciones de campos en formato octal identificados matemáticamente como x , y , z , p y q . Esto significa segmentar el byte en bloques de 2 y 3 bits según la siguiente plantilla estructural:

Bits (MSB → LSB)	7	6	5	4	3	2	1	0
Estructura Principal	x		y			z		
Sub-estructura de y			p		q			

Tabla 5.1: Disección en campos de un opcode de 8 bits del Z80.

A partir de esta disección rápida usando desplazamientos y máscaras de bits, las variables resultan en:

- x = bits 7-6 (El “primer dígito octal” o grupo maestro, valores 0-3).
- y = bits 5-3 (El “segundo dígito octal”, comúnmente usado para el registro destino o la operación ALU, valores 0-7).
- z = bits 2-0 (El “tercer dígito octal”, registro origen, valores 0-7).

- **p** = bits 5-4. Es equivalente a $y \gg 1$ (valores 0-3).
- **q** = bit 3. Es equivalente a y (mód 2) (valores 0-1).

El diseñador del procesador proyectó tablas de equivalencia (“tablas de ensamblador”) directamente ligadas a estos índices numéricos para extraer operandos unívocos. A continuación, se detallan todas las tablas de decodificación que forman el corazón del set de instrucciones del Z80, modeladas en el código base.

Tabla r (Registros de 8 bits)								
Índice	0	1	2	3	4	5	6	7
Valor	B	C	D	E	H	L	(HL)	A

Tabla cc (Condiciones)								
Índice	0	1	2	3	4	5	6	7
Valor	NZ	Z	NC	C	PO	PE	P	M

Tabla rp (Pares con SP)				
Índice	0	1	2	3
Valor	BC	DE	HL	SP

Tabla rp2 (Pares con AF)				
Índice	0	1	2	3
Valor	BC	DE	HL	AF

Tabla alu (Aritmética/Lógica)								
Índice	0	1	2	3	4	5	6	7
Valor	ADD	ADC	SUB	SBC	AND	XOR	OR	CP

Tabla rot (Rotación y Desplazamiento)								
Índice	0	1	2	3	4	5	6	7
Valor	RLC	RRC	RL	RR	SLA	SRA	SLL	SRL

Tabla 5.2: Mapeo de operandos base del procesador Z80 (**r**, **cc**, **rp**, **rp2**, **alu** y **rot**).

Tabla im (Modos de Interrupción)								
Índice	0	1	2	3	4	5	6	7
Valor	0	0/1	1	2	0	0/1	1	2

Tabla 5.3: Decodificación de modos de interrupción del Z80 (**im**).

Por último, las instrucciones de bloque (**bli**), utilizadas para la transferencia masiva, búsqueda en memoria y E/S de bloques (precedidas típicamente por el prefijo extendido **0xED**), requieren dos índices (y interactuando con z):

Gracias a este esquema predecible, los cientos de instrucciones del microprocesador quedan algorítmicamente reducidos a fórmulas sencillas. Por ejemplo,

Tabla bli (Instrucciones de Bloque)				
Índice	$z = 0$	$z = 1$	$z = 2$	$z = 3$
$y = 4$	LDI	CPI	INI	OUTI
$y = 5$	LDD	CPD	IND	OUTD
$y = 6$	LDIR	CPIR	INIR	OTIR
$y = 7$	LDDR	CPDR	INDR	OTDR

Tabla 5.4: Matriz de decodificación de instrucciones de bloque del Z80 (bli).

en el espacio donde $x = 2$, todos los *opcodes* calculan internamente operativas aritmético-lógicas contra el acumulador (A) empleando la directriz genérica `alu[y] r[z]`. Si la CPU extrae el byte `0x83` (10 000 011 en binario), la disección arroja $x = 2$, $y = 0$ y $z = 3$. Según las tablas definidas correspondientes de arriba, la orden unívoca a ejecutar es por tanto `ADD A, E`, evitando así implementar una instrucción totalmente desconectada para cada combinación de matriz.

5.2.3. Tipado Fuerte en la Decodificación: El *Trait* TryFrom

En implementaciones ingenuas en el lenguaje C u otros menos rigurosos, estos desplazamientos a nivel de bits se suelen arrastrar como simples números, dando frecuentemente lugar a estructuras enormes sustentadas sobre *números mágicos*, afectando a la mantenibilidad del proyecto *open source*. Aprovechando en su lugar la filosofía de programación de Rust, este proyecto centraliza exhaustivamente y aísla la carga cognitiva asociada a estas traducciones analizadas en la sección previa, mediante las conversiones seguras que provee el *trait* estándar: `TryFrom<u8>`.

A lo largo del código de la CPU (como puede probarse en el módulo `src/cpu/mod.rs`), esta abstracción tipa fuertemente todos los elementos semánticos que forman una instrucción (la condición de salto `Condition`, los registros destino `RegOps`, la operación ALU elegida `ALUOperation`, etc.).

Un fragmento que refleja dicha arquitectura aplicada la operación de ALU:

Figura 5.8: Implementación del *trait* TryFrom en las operaciones ALU.

```

1  impl TryFrom<u8> for ALUOperation {
2      type Error = String;
3
4      fn try_from(value: u8) -> Result<Self, Self::Error> {
5          match value {
6              0 => Ok(ALUOperation::ADD),
7              1 => Ok(ALUOperation::ADC),
8              2 => Ok(ALUOperation::SUB),
9              3 => Ok(ALUOperation::SBC),

```

```

10         4 => Ok(ALUOperation::AND),
11         5 => Ok(ALUOperation::XOR),
12         6 => Ok(ALUOperation::OR),
13         7 => Ok(ALUOperation::CP),
14         _ => Err(format!("Invalid ALU operation value: {}",
15                           value)),
15     }
16 }
17 }

```

Inyectando en cada fase esta verificación intermedia (que el compilador traduce a un coste de validación casi nulo), el resto del desarrollo ha garantizado al programador y al evaluador que funciones críticas como `fn execute_alu_op(op: ALUOperation)` nunca jamás puedan ser invocadas con números arbitrarios e inconexos, logrando una decodificación predecible y una cobertura blindada sobre el modelo Z80 subyacente.

5.2.4. La ALU y el Registro de Banderas

La Unidad Aritmético Lógica (ALU) es responsable de realizar operaciones matemáticas y lógicas (Suma, Resta, AND, OR, XOR) y actualizar el registro de banderas **F** (Flags) acorde al resultado.

El registro F del Z80 es particularmente complejo debido a banderas no documentadas o sutiles:

- **S (Flag de Signo)**: Indica si el resultado es negativo (bit 7).
- **Z (Flag de Cero)**: Indica si el resultado es cero.
- **F5/X (Flag no documentada)**: Copia el bit 5 del resultado, afectando ciertas instrucciones.
- **H (Flag de Medio Acarreo)**: Indica si hubo acarreo entre los bits 3 y 4, crucial para operaciones BCD.
- **F3/Y (Flag no documentada)**: Copia el bit 3 del resultado, afectando ciertas instrucciones.
- **P/V (Flag de Paridad/Desbordamiento)**: Indica si el resultado tiene un número par de bits a 1 (paridad) o si hubo desbordamiento en operaciones aritméticas.
- **N (Flag de Resta)**: Indica si la última operación fue una resta (1) o una suma (0).

- **C (Flag de Acarreo):** Indica si hubo acarreo en la operación, afectando operaciones de suma, resta y rotaciones.

5.2.5. Fidelidad y Comportamiento No Documentado

A la hora de desarrollar un simulador preciso y riguroso, uno de los mayores retos consiste en decidir si emular o no comportamientos derivados de fallos de diseño del silicio original y peculiaridades que, aunque no figuran en los manuales oficiales de Zilog, fueron descubiertos por la comunidad [1, 2, 3] y utilizados para exprimir y optimizar código en la era de los 8 bits. Este proyecto toma la firme decisión de implementar con máxima fidelidad los comportamientos no documentados para asegurar una compatibilidad absoluta con el software *legacy*.

Entre las peculiaridades implementadas, destacan:

- **Gestión fiel de banderas implícitas (X e Y):** Como se mencionó anteriormente, el registro de estado almacena copias de los bits 3 y 5 de la ALU tras muchas instrucciones, comportamientos que el simulador valida y testea de forma rigurosa.
- **Partición no oficial de registros índice (IXH, IXL, IYH, IYL):** El Z80 permite de forma no documentada acceder a los bytes alto y bajo de los registros IX e IY utilizando los mismos códigos operacionales que accederían a H e L, pero precedidos por los bytes identificadores 0xDD o 0xFD. El núcleo de este simulador recrea y expone este funcionamiento en su decodificador de prefijos.

IX (15–0)	
IXH (15–8)	IXL (7–0)

IY (15–0)	
IYH (15–8)	IYL (7–0)

Tabla 5.5: Estructura de los registros índice mostrando los componentes de los bytes alto y bajo.

- **La instrucción SLL (o SL1):** Una instrucción lógica no listada en el repertorio oficial que funciona idénticamente a SLA (Shift Left Arithmetic), pero que a diferencia de ella, introduce un valor constante de 1 en el bit menos significativo [4]. El código base en Rust incorpora explícitamente el opcode virtual en `alu::rot::sll`.
- **Copia de resultados en instrucciones dependientes de bits:** Al invocar instrucciones basadas en los prefijos compuestos 0xDDCB o 0xFDCB

combinadas con los campos para *Set*, *Reset* o rotaciones, el microprocesador original ejecuta la decodificación en memoria apoyado en el índice, además deposita una copia del resultado final en uno de los registros de la CPU, asumiendo que este es referenciado correctamente. En el núcleo central (`src/cpu/mod.rs`), este suceso se reproduce fielmente del siguiente modo:

Figura 5.9: Implementación de la copia de resultados en instrucciones con prefijos compuestos.

```

1      0 => {
2          // rot[y] (IX/IY+d)
3          test_log!(self, "rot[y] ({:?} + d)", indexing)
4          ;
5          // Obtener valor de (IX/IY+d)
6          let value = self.read_8(addressing_mode);
7          let operation = self.table_rot(y);
8
9          // Ejecutar rotación
10         let (result, flags) = ...
11
12         self.set_register(GPR::F, flags);
13         self.write_8(addressing_mode, result);
14
15         // No documentado, copia el resultado en el
16         // registro r[z] si z != 6
17         if z != 6 {
18             let dest_reg = self.table_r(z);
19             self.write_8(dest_reg.into(), result);
20         }
21     }

```

5.2.6. El Ensamblador de Dos Pasadas

Para permitir al usuario escribir programas directamente en mnemónicos Z80, se ha implementado un ensamblador integrado. Debido a la naturaleza de las etiquetas (labels) que pueden referenciarse antes de ser definidas (forward references), se utiliza una estrategia de dos pasadas:

1. Pass 1 (Symbol Resolution):

- Recorre el código fuente tokenizado.
- Mantiene un contador de programa (PC) virtual.
- Registra todas las etiquetas y sus direcciones en un `HashMap<String, u16>`.

- Calcula el tamaño de cada instrucción para incrementar el PC correctamente, pero no genera código máquina.

2. Pass 2 (Code Generation):

- Vuelve a recorrer el código.
- Traduce mnemónicos a opcodes.
- Resuelve los operandos que son etiquetas usando la tabla generada en la fase 1.
- Calcula saltos relativos (JR) verificando que el salto esté dentro del rango de un byte con signo (+127/-128).

Este diseño desacopla la lógica de análisis léxico de la generación de código, facilitando el soporte de pseudoinstrucciones como `ORG` (origen) o `DB` (definir byte).

El flujo continuo y analítico desde la interfaz al núcleo de emulación se representa en el siguiente diagrama de secuencia:

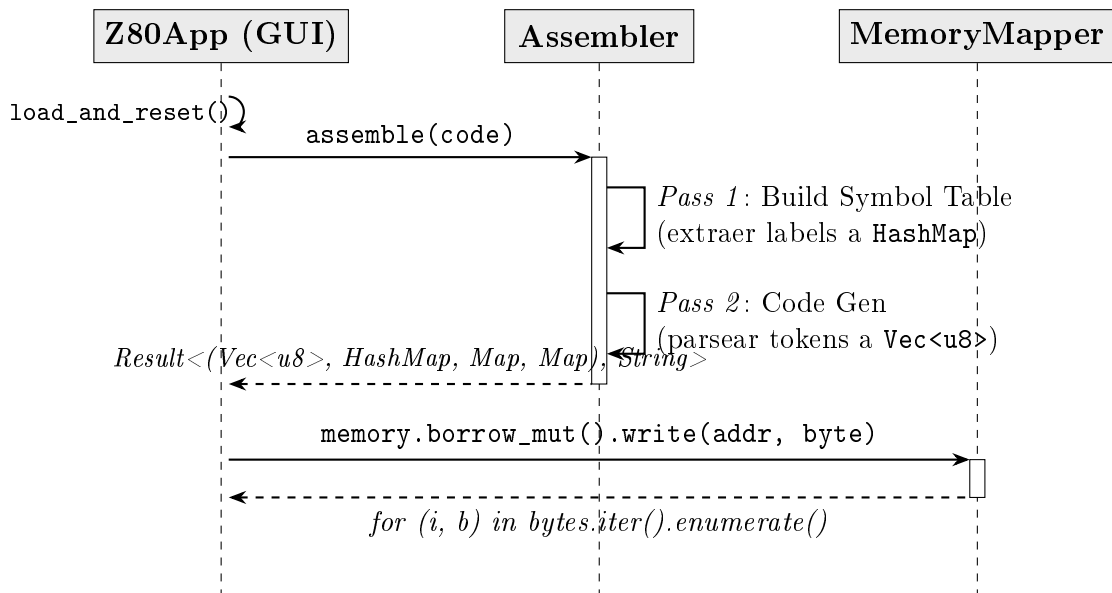


Figura 5.10: Diagrama de Secuencia del Ensamblador de Dos Pasadas interactuando con **Z80App** y **MemoryMapper**. Muestra cómo el ensamblador es estrictamente funcional (sin estado ni dependencias colaterales) delegando el volcado binario a la capa unificada de la aplicación.

5.2.7. Sistema de Periféricos y Dispositivos de E/S

El diseño del simulador contempla un bus de entrada/salida (E/S) extensible mediante el uso de *Traits* (interfaces) de Rust, permitiendo la creación de dispositivos virtuales complejos que pueden interactuar tanto con la CPU como con el usuario a través de la interfaz gráfica.

Abstracción del Hardware: El Trait IODevice

Para desacoplar la lógica interna del Z80 de los periféricos específicos, se ha definido el trait `IODevice` en el módulo `traits.rs`. Este contrato establece cómo cualquier componente de hardware debe comportarse frente a las señales de control de la CPU:

Figura 5.11: Declaración del *trait* `IODevice` para dispositivos mapeados por puertos.

```

1  pub trait IODevice {
2      // Lectura desde un puerto (IN A, (n))
3      fn read_in(&mut self, port: u16) -> Option<u8>;
4
5      // Escritura en un puerto (OUT (n), A)
6      fn write_out(&mut self, port: u16, data: u8) -> bool;
7
8      // Consulta de interrupciones enmascarables (INT)
9      fn poll_interrupt(&self) -> bool { false }
10
11     // Respuesta al reconocimiento de interrupción (INTA)
12     // Devuelve el vector de interrupción o un opcode (RST)
13     fn ack_interrupt(&mut self) -> u8 { 0xFF }
14
15     // Consulta de interrupción no enmascarable (NMI)
16     fn poll_nmi(&self) -> bool { false }
17 }

```

La estructura principal Z80A mantiene un vector de objetos que implementan este trait (`Vec<Rc<RefCell<dyn IODevice>>`). Durante la ejecución de una instrucción de E/S, la CPU itera sobre este vector. Si un dispositivo reconoce el puerto direccionado, realiza la operación y devuelve el control. Este diseño permite conectar múltiples dispositivos al mismo bus virtual sin modificar el código del núcleo del emulador, simulando la extensibilidad de un sistema real. Además, el método `poll_interrupt` permite simular la línea física de interrupción cableada en “O” (*Wired-OR*), donde cualquier dispositivo puede solicitar atención de la CPU en cualquier momento.

Interfaz Gráfica de Dispositivos: El Trait DeviceWithUi

Dado que el objetivo es un simulador visual, los dispositivos no solo deben responder a señales eléctricas, sino también mostrar su estado al usuario. Para ello, se ha creado el trait `DeviceWithUi` en `ui_traits.rs`, que extiende la funcionalidad de `IIODevice`:

Figura 5.12: Extensión de dispositivo con capacidades gráficas mediante `DeviceWithUi`.

```

1  pub trait DeviceWithUi: IIODevice {
2      // Método de dibujo en Egui
3      fn draw(&mut self, ctx: &egui::Context);
4
5      fn get_name(&self) -> String;
6      fn get_window_open_state(&self) -> bool;
7      fn set_window_open_state(&mut self, open: bool);
8
9      fn toggle_window(&mut self) {
10         let state = self.get_window_open_state();
11         self.set_window_open_state(!state);
12     }
13 }
```

Gracias al sistema de propiedad compartida de Rust (`Rc<RefCell<...>`), una misma instancia de un dispositivo (como un teclado matricial o un display de 7 segmentos) puede ser poseída simultáneamente por:

- **La CPU:** Que lo ve como un `IIODevice` opaco para leer teclas o escribir en el display mediante puertos.
- **La Interfaz Gráfica:** Que lo ve como un `DeviceWithUi` para dibujar la ventana del dispositivo y capturar clics del ratón.

Esta arquitectura permite que la interacción sea bidireccional y en tiempo real: si el usuario pulsa una tecla en la ventana del “Teclado Virtual”, el estado interno del dispositivo cambia inmediatamente, y cuando la CPU lea el puerto correspondiente en el siguiente ciclo, recibirá el nuevo valor. De igual manera, si la CPU escribe en un puerto de salida, el método `draw` reflejará el cambio en la pantalla (encendiendo un LED virtual, por ejemplo) en el siguiente fotograma.

5.2.8. Interfaz Gráfica y Renderizado

La interfaz de usuario utiliza `egui`, una librería de modo inmediato (*Immediate Mode GUI*). A diferencia de las GUIs retenidas (como QT o GTK) donde se crea

un árbol de widgets persistente, en **egui** la interfaz se reconstruye completamente en cada frame

Esto simplifica enormemente la gestión de estado mutable compartido con el emulador, ya que la UI es simplemente una vista del estado actual de la CPU y la memoria en ese instante exacto.

Visualizador de Memoria Virtualizado

Dado que la memoria del Z80 es de 64KB, renderizar 65536 celdas cada frame sería costoso. Se ha implementado una técnica de *virtual scrolling*: solo se dibujan las filas de memoria que son visibles actualmente en la ventana, calculando dinámicamente qué rango de direcciones mostrar basado en la posición de la barra de desplazamiento.

5.2.9. Ejecución en WebAssembly (WASM)

Uno de los requisitos clave era la portabilidad a la web. Rust tiene soporte de primera clase para WASM, pero el modelo de ejecución web (single-thread, event loop del navegador) impide el uso de hilos bloqueantes para la simulación.

Para resolver esto sin complicar la arquitectura con Web Workers, se ha optado por integrar el ciclo de la CPU directamente en el bucle de renderizado de **eframe**. Tanto en la versión nativa como en la web, la aplicación funciona dirigida por eventos.

En cada frame de animación, si la CPU está en estado “running”, se ejecuta un lote fijo de instrucciones (batching). Esto evita congelar la interfaz de usuario mientras se mantiene una velocidad de emulación adecuada.

Figura 5.13: Actualización de la interfaz en Egui simulando pasos de la CPU.

```

1  impl eframe::App for Z80App {
2      fn update(&mut self, ctx: &egui::Context, _frame: &mut
          eframe::Frame) {
3          // ... Lógica de gestión de archivos y paneles ...
4
5          if self.is_running {
6              let mut steps = 0;
7              // Se ejecutan 10,000 instrucciones por frame para
              // mantener
8              la UI responsiva
9              while steps < 10000 {
10                 if self.cpu.is_halted() {
11                     // Si está en HALT, seguimos haciendo tick para
                     procesar

```

```
12         interrupciones
13         self.cpu.tick();
14         if !self.cpu.is_halted() {
15             steps += 1;
16             continue;
17         }
18         break;
19     }
20
21     // Verificación de breakpoints
22     // ...
23
24     self.cpu.tick(); // Ejecuta una instrucción
25                       completa
26     steps += 1;
27 }
28
29 if self.is_running {
30     // Solicitar un nuevo frame inmediatamente para
31     // animación fluida
32     ctx.request_repaint();
33 }
34
35 // ... Renderizado de componentes UI ...
36 }
```

Este diseño asegura que la simulación progrese suavemente sin monopolizar el hilo principal, permitiendo que el navegador atienda eventos de entrada (teclado, ratón) entre frames.

Esto asegura que la interfaz siga respondiendo a eventos del usuario mientras la CPU simulada avanza a una velocidad percibida correcta. Por otro lado, la aplicación admite interceptar la ejecución para realizar *Stepping* visual paso a paso, integrando el flujo gráfico con las reglas lógicas emuladas como se muestra en el siguiente diagrama:

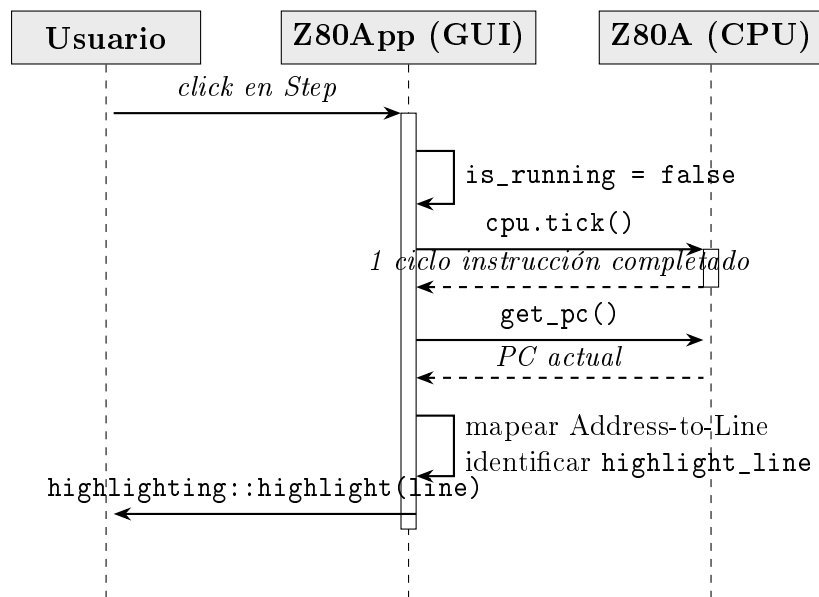


Figura 5.14: Diagrama de Secuencia ilustrando el bucle gráfico para el flujo de depuración (Manual Stepping) e inspección de la línea activa controlada por PC.

6

Experimentos / Validación

La validación de un emulador de CPU es un proceso crítico y exhaustivo, dado que cualquier error sutil en la lógica de una instrucción o en el manejo de banderas puede resultar en la ejecución incorrecta de programas complejos. Para garantizar la fidelidad con el hardware original Zilog Z80, se ha diseñado una estrategia de validación en múltiples niveles, combinando pruebas unitarias automatizadas con la ejecución de algoritmos de integración.

6.1. Metodología de Pruebas Unitarias

La corrección de un emulador depende de la precisión de sus componentes más pequeños. Para asegurar esta precisión, se ha adoptado una metodología de pruebas unitarias exhaustiva, apoyada en el ecosistema de herramientas de calidad de Rust.

El proyecto utiliza el framework de pruebas nativo (`cargo test`), pero se ha potenciado significativamente mediante la integración de la librería externa `rstest`. Esta librería introduce en Rust el concepto de *fixtures* y pruebas parametrizadas, similar a `pytest` en Python o `JUnit 5` en Java, lo cual ha sido fundamental para gestionar la combinatoria del set de instrucciones del Z80.

La elección de `rstest` responde a tres necesidades clave del proyecto:

1. **Modularidad y Reutilización:** Permite definir la lógica de verificación de una instrucción (por ejemplo, “ejecutar opcode, comprobar acumulador,

comprobar flags”) una única vez y reutilizarla para todas las variantes de esa instrucción.

2. **Pruebas Guiadas por Datos (Data-Driven Testing):** Mediante el atributo `#[case]`, los casos de prueba se expresan como una tabla de entradas y salidas esperadas. Esto convierte el código del test en una especificación legible del comportamiento del hardware.
3. **Trazabilidad de Errores:** A diferencia de un bucle `for` dentro de un test tradicional, `rstest` genera una instancia de prueba independiente para cada caso. Si una variante falla (por ejemplo, la suma con acarreo), el reporte de error identifica exactamente qué combinación de parámetros causó el fallo sin detener la ejecución del resto de casos.

Esta estrategia de *Testing paramétrico* es vital para una CPU donde una sola instrucción como `ADD` puede tener comportamientos sutilmente diferentes según el estado de las banderas previas. Gracias a `rstest`, se ha podido desacoplar la complejidad de la configuración del test de la complejidad de los datos de prueba, resultando en un código más limpio, mantenible y auto-documentado.

El código fuente del simulador contiene una extensa suite de tests ubicada en los directorios `src/cpu/tests/instructions`. Este capítulo detalla la arquitectura de las pruebas y disecciona los grupos de validación más críticos.

6.2. Infraestructura de Pruebas: MockMemory

Para realizar pruebas unitarias aisladas de la CPU sin necesidad de inicializar todo el sistema gráfico o el bus de periféricos real, se ha desarrollado una abstracción de memoria específica denominada `MockMemory`. Esta estructura se encuentra definida en el módulo `src/cpu/tests/mod.rs`.

`MockMemory` implementa el trait `MemoryMapper`, comportándose externamente como una memoria RAM de 64KB, pero simplificando su construcción para facilitar la inyección de código máquina directamente en las pruebas.

Figura 6.1: Estructura `MockMemory` para realizar las inyecciones de código en tests.

```
1 struct MockMemory {
2     data: [u8; 0x10000], // Array plano de 64KB
3 }
4
5 impl MemoryMapper for MockMemory {
6     fn read(&self, address: u16) -> u8 {
7         self.data[address as usize]
```

```
8     }  
9     fn write(&mut self, address: u16, data: u8) {  
10         self.data[address as usize] = data;  
11     }  
12 }
```

Cada prueba individual instancia una CPU “limpia” utilizando la función auxiliar `setup_cpu()`, que conecta el procesador a una nueva instancia de `MockMemory`. Esto garantiza que el estado residual de una prueba (registros modificados, escrituras en memoria) nunca contamine la ejecución de la siguiente, un principio fundamental de las pruebas unitarias confiables.

6.3. Análisis de los Ficheros de Prueba

La validación del set de instrucciones se distribuye en múltiples archivos, cada uno enfocado en un subconjunto funcional de la arquitectura Z80. A continuación, se describen los componentes más relevantes del sistema de validación.

6.3.1. Validación Aritmética: `alu_tests.rs`

Este archivo, ubicado en `src/cpu/tests/alu_tests.rs`, es el guardián de la lógica bit a bit. Su misión es validar las operaciones que manipulan bits individuales y afectan de manera compleja al registro de banderas.

Las pruebas en este módulo cubren:

- **Rotaciones** (RLC, RRC, RL, RR): Verifican que los bits roten correctamente a través del acarreo o circularmente. Por ejemplo, se prueba exhaustivamente que RLC (Rotate Left Circular) copie el bit 7 tanto al bit 0 como al flag de acarreo (C).
- **Desplazamientos** (SLA, SRA): Se valida la preservación del signo en los desplazamientos aritméticos a la derecha (SRA), asegurando que el bit más significativo (MSB) se replique para mantener el signo del número en complemento a dos.
- **Manipulación de Bits** (BIT, SET, RES): Se comprueba que BIT actualice correctamente la bandera Z (Zero) sin modificar el operando, mientras que SET y RES deben alterar el bit objetivo sin afectar ninguna bandera.

El uso de `rstest` permite definir vectores de prueba compactos:

Figura 6.2: Uso básico de la macro `#[case]` para tests tabulares de `rstest`.

```

1  #[rstest]
2  #[case(0x80, 0x01, true)] // RLC 0x80 -> 0x01 (Carry=1)
3  #[case(0x01, 0x02, false)] // RLC 0x01 -> 0x02 (Carry=0)
4  fn test_rlc(#[case] val: u8, #[case] res: u8, #[case] cy:
    bool) { ... }

```

6.3.2. Operaciones Tabuladas: `alu_table_ops.rs`

Ubicado en `src/cpu/tests/instructions/alu_table_ops.rs`, este archivo utiliza macros para generar automáticamente pruebas para las operaciones aritméticas estándar (ADD, ADC, SUB, SBC, AND, XOR, OR, CP).

Dada la regularidad del set de instrucciones del Z80 (donde el opcode para ADD A, B es 0x80 y ADD A, C es 0x81), este módulo verifica que la decodificación del registro fuente sea correcta. Si el emulador confundiera el registro B con el C, estas pruebas lo detectarían inmediatamente al pre-cargar valores distintos en cada registro y verificar cuál se ha sumado al acumulador.

Se presta especial atención a la bandera H (Half Carry), vital para la corrección BCD, verificando transiciones de 0x0F a 0x10 (desbordamiento de nibble bajo).

6.3.3. Control de Flujo: `jumps.rs`

El archivo `src/cpu/tests/instructions/jumps.rs` valida la capacidad de la CPU de alterar su contador de programa (PC).

Saltos Relativos (JR)

Se verifica la aritmética de punteros en complemento a dos. Un salto relativo JR -5 debe restar 5 al PC actual.

Figura 6.3: Prueba de salto relativo simulando un desbordamiento negativo.

```

1  #[case(0x1000, -5i8 as u8, 0xFFFD)] // 0x1000 - 5 + 2 (
    len) = 0xFFFD
2  fn test_jr_relative(...)

```

Es crucial notar que el salto se calcula desde la dirección de la *siguiente* instrucción (PC + 2).

Bucle (DJNZ)

La instrucción DJNZ (Decrement Jump Non-Zero) se prueba para asegurar que decrementa el registro B y salta solo si B no llega a cero. Las pruebas cubren el caso donde B se decrementa a cero, confirmando que el salto no ocurre, y el caso donde B se decrementa a un valor no cero, confirmando que el salto se ejecuta correctamente.

6.3.4. Interrupciones y NMI: `nmi_tests.rs`

Este módulo es fundamental para garantizar la estabilidad del sistema ante eventos asíncronos. Valida la máquina de estados de las interrupciones:

1. **Prioridad:** NMI siempre interrumpe, incluso con DI (Disable Interrupts).
2. **Retorno:** La instrucción `RETN` debe restaurar el estado de las interrupciones copiando IFF2 de vuelta a IFF1.
3. **Enmascaramiento:** Al aceptar una INT, las interrupciones deben deshabilitarse automáticamente (IFF1=0, IFF2=0).

6.4. Validación del Ensamblador

Además de la CPU, se valida el proceso de traducción de código fuente a máquina. Las pruebas se encuentran en `src/assembler/tests`.

6.4.1. Tokenización: `mod.rs`

El primer paso del ensamblador es convertir texto en tokens. El test `test_tokenize_all_tokens` verifica que cadenas complejas como `LD A, (IX+10)` se descompongan correctamente en: `[Identifier(LD), Identifier(A), Comma, OpenParen, Identifier(IX), Plus, Number(10), CloseParen]`.

6.4.2. Resolución de Etiquetas

Se valida que el ensamblador calcule correctamente los desplazamientos para las etiquetas. Si se define una etiqueta `LOOP` al inicio, y se insertan 10 bytes de instrucciones, un salto `JP LOOP` al final debe generar el código máquina `C3 00 00` (asumiendo `org 0`).

6.5. Validación de Aritmética BCD (Decimal Adjust)

6.5.1. Validación de la Unidad Aritmético-Lógica (ALU)

Las operaciones aritméticas y lógicas son el núcleo del procesador. Se han implementado pruebas específicas para cada grupo de operaciones, verificando no solo el resultado numérico en el acumulador, sino el estado exacto de las seis banderas del registro F (S, Z, Y, H, X, P/V, N, C).

Operaciones de Rotación y Desplazamiento

El archivo `alu_tests.rs` verifica las instrucciones de manipulación de bits como RLC, RRC, RL, RR, SLA, SRA y SLL. Estas instrucciones son propensas a errores en el manejo del acarreo (*Carry*) y el desbordamiento.

Por ejemplo, para la instrucción RLC (Rotate Left Circular), se comprueban casos como:

- **Rotación de bits extremos:** Rotar el valor `0b1000_0001` debe resultar en `0b0000_0011` (el bit 7 pasa al bit 0 y al Carry), con las banderas `C=1` y `P/V=1` (paridad par).
- **Rotación simple:** Rotar `0b0000_0001` debe dar `0b0000_0010` con `C=0`.

El uso de macros `#[case]` permite definir estos vectores de prueba de manera declarativa:

Figura 6.4: Prueba completa de rotación bit a bit (RLC) validando banderas.

```

1  #[rstest]
2  #[case(0b1000_0001, 0b0000_0011, 0b0000_0101)] // result
   =0x03,
3  flags: C=1, PV=1
4  #[case(0b0000_0001, 0b0000_0010, 0b0000_0000)] // result
   =0x02, flags: PV=1
5  fn test_rlc(#[case] value: u8, #[case] expected_result:
   u8, #[case]
6  expected_flags: u8) {
7      let (result, flags) = rlc(value);
8      assert_eq!(result, expected_result, "Result mismatch");
9      assert_eq!(flags, expected_flags, "Flags mismatch");
10 }

```

De manera similar, se validan instrucciones aritméticas de desplazamiento aritmético (SRA), donde es crucial verificar que el bit de signo (bit 7) se propague correctamente al desplazarse a la derecha (preservación del signo).

Aritmética Fundamental y Tablas de Operaciones

Para instrucciones como ADD, SUB, ADC y SBC, se utiliza el archivo `alu_table_ops.rs`. Este test valida que la decodificación de operandos (ya sean registros individuales como A, B, C, pares de registros HL, o valores inmediatos) se realice correctamente antes de invocar a la ALU.

Se prueban escenarios específicos para la bandera *Half Carry* (H), esencial para la aritmética BCD posterior. Por ejemplo, al sumar `0x0F + 0x01`, el resultado es `0x10`, pero debe activarse la bandera H debido al desbordamiento del nibble bajo (bits 0-3), aunque no haya acarreo global.

6.5.2. Validación de Aritmética BCD (Decimal Adjust)

Una de las instrucciones más complejas del Z80 es DAA (Decimal Adjust Accumulator). Su comportamiento depende de una combinación no trivial del valor del acumulador y las banderas N, H y C previas.

Para validar esta instrucción, no se usaron casos aislados, sino que se implementó un test basado en trazas (*Golden File Testing*) en el archivo `decimal.rs`. El test lee un archivo externo `daa.txt` que contiene miles de líneas con estados pre y post ejecución obtenidos de un Z80 real. El test parsea cada línea, configura la CPU con el estado inicial, ejecuta DAA y compara el resultado bit a bit.

Figura 6.5: Lógica de *parsing* para *Golden File Testing* de la instrucción DAA.

```
1 fn parse_line_to_test_case(line: &str) -> Option<(u8,  
    bool, bool,  
2 bool, u8, bool, bool, bool)> {  
3     // Parsea: A_inicial Flags_iniciales -> A_esperado  
    Flags_esperadas  
4     // Ejemplo de línea: N 0 C 0 H 0 99 -> N 0 C 1 H 0 99  
5 }
```

Este enfoque garantiza una compatibilidad del 100 % con el comportamiento “peculiar” del hardware original en aritmética decimal.

6.6. Validación del Sistema de Interrupciones

El sistema de interrupciones, especialmente la Interrupción No Enmascarable (NMI), tiene una lógica de prioridades estricta. Las pruebas en `nmi_tests.rs` verifican el comportamiento de la CPU en situaciones de concurrencia de eventos.

1. **Prioridad de NMI:** Se verifica que una NMI interrumpa el flujo normal, empujando el PC a la pila y saltando a la dirección `0x0066`.
2. **Inhibición de Interrupciones Enmascarables:** Se comprueba que al aceptar una NMI, la CPU copia el estado de `IFF1` a `IFF2` y luego deshabilita `IFF1` para prevenir interrupciones anidadas no deseadas de menor prioridad.
3. **Retorno de NMI (RETN):** Se valida que la instrucción especial `RETN` restaure correctamente el estado de las interrupciones copiando `IFF2` de vuelta a `IFF1`, permitiendo al sistema recuperar su estado anterior a la interrupción crítica.

Figura 6.6: Prueba de inhibición de interrupciones enmascarables bajo un evento NMI.

```

1  #[test]
2  fn test_nmi_disables_maskable_interrupts() {
3      let mut cpu = setup_cpu();
4      cpu.iff1 = true; // Interrupciones habilitadas
5
6      cpu.nmi();       // Se dispara NMI
7      cpu.tick();      // Se procesa
8
9      // El hardware debe deshabilitar IFF1 automaticamente
10     assert_eq!(cpu.iff1, false);
11     // Pero guardar el estado en IFF2
12     assert_eq!(cpu.iff2, true);
13     assert_eq!(cpu.pc, 0x0066);
14 }

```

6.7. Validación Funcional: Algoritmos Completos

Más allá de las instrucciones individuales, se ha validado el sistema ejecutando programas completos que integran múltiples subsistemas (memoria, saltos, pila, ALU).

6.7.1. Caso de Estudio: Sucesión de Fibonacci

Uno de los tests principales de integración fue el cálculo de la sucesión de Fibonacci. Este programa es especialmente útil para validar la interacción entre la aritmética de 16 bits, el control de flujo y el acceso a memoria.

Objetivos de la prueba:

- Verificar ADD HL, DE: Suma de 16 bits sin acarreo.
- Verificar JR NZ, ...: Saltos relativos condicionales precisos.
- Verificar pseudo-instrucciones del ensamblador (LD HL, 1) que se traducen en cargas inmediatas.

El código fuente utilizado para la validación es el siguiente:

Figura 6.7: Programa de cálculo de la sucesión de Fibonacci en ensamblador del Z80.

```

1  ; =====
2  ; Z80 Fibonacci Validation
3  ; =====
4  ORG 0000h
5  JP START
6
7  N:      DB 0Ah          ; Calcular F(10)
8  RESULT: DW 0           ; Resultado esperado: 55 (0x37)
9
10 START:
11 LD A, (N)
12 OR A
13 JR Z, FIB_ZERO ; Caso base n=0
14
15 CP 1
16 JR Z, FIB_ONE  ; Caso base n=1
17
18 LD DE, 0       ; fib0 = 0
19 LD HL, 1       ; fib1 = 1
20
21 DEC A          ; Ajustar contador de bucle
22 DEC A
23
24 JR Z, DONE
25
26 FIB_LOOP:
27 ; Mover HL -> BC (fib1 -> temp)
28 LD B, H

```

```

29 LD C, L
30
31 ADD HL, DE      ; HL = fib1 + fib0 (Nuevo fib1)
32
33 ; Mover BC -> DE (temp -> fib0)
34 LD D, B
35 LD E, C
36
37 DEC A
38 JR NZ, FIB_LOOP
39
40 DONE:
41 LD (RESULT), HL ; Almacenar resultado en memoria
42 HALT
43
44 FIB_ZERO:
45 LD HL, 0
46 LD (RESULT), HL
47 HALT
48
49 FIB_ONE:
50 LD HL, 1
51 LD (RESULT), HL
52 HALT

```

Resultado: Tras la ejecución, se inspeccionó la memoria en la dirección `RESULT`. El valor obtenido fue `0x0262` (610 en decimal), confirmando que el simulador ejecutó correctamente tanto la lógica del bucle como las operaciones aritméticas de 16 bits. Este test valida implícitamente que la bandera Z funciona correctamente tras operaciones de decremento y que los saltos relativos calculan correctamente el desplazamiento negativo en complemento a dos.

6.8. Conclusiones de la Validación

La combinación de pruebas unitarias exhaustivas basadas en tablas, validación con trazas *golden master* para instrucciones críticas como `DAA`, y pruebas de integración con algoritmos reales, permite asegurar con un alto grado de confianza que el simulador es una réplica funcionalmente fiel del Zilog Z80, apta para la ejecución de software educativo y de legado.

7

Conclusiones

Se presentan las conclusiones principales del trabajo y se proponen líneas de trabajo futuro.

7.1. Logros principales alcanzados

Resumen de los objetivos cumplidos y contribuciones del trabajo.

7.2. Posibles trabajos futuros

Propuestas para continuar o mejorar el proyecto.

Bibliografía

- [1] C. V., “Z80 instruction set decoding,” <http://www.z80.info/decoding.htm>, accedido: 2026-03-12.
- [2] jnz.dk, “Z80 undocumented instructions and behaviors,” <https://jnz.dk/z80/>, accedido: 2026-03-12.
- [3] ClrHome, “Z80 instruction set table,” <https://clrhome.org/table/>, accedido: 2026-03-12.
- [4] S. Young, *The Undocumented Z80 Documented*, <http://www.z80.info/zip/z80-documented.pdf>, 2005, accedido: 2026-03-12.

Apéndice



Este es el primer apéndice

A.1. Ejemplo de sección

Sección del apéndice