Escuela
Politécnica
Superior

GBee

Máster Universitario en Desarrollo de Software para Dispositivos Móviles

Trabajo Fin de Máster

Autor:
Ángel Jesús Terol Martínez
Tutor:
Luis Lucas Ibáñez

Universitat d'Alacant
Universidad de Alicante

Octubre 2024

GBee

Emulador de GameBoy para dispositivos Android

Autor
Ángel Jesús Terol Martínez

Tutor
Luis Lucas Ibáñez
Tecnología Informática y Computación

Máster Universitario en Desarrollo de Software para Dispositivos Móviles

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Octubre 2024

Preámbulo

Este Trabajo Final de Máster (TFM) surge de la motivación por aprender cómo desarrollar un programa que emule el funcionamiento completo de una consola, integrando los conocimientos adquiridos durante el curso. Además de ser un reto personal y técnico, este proyecto permite explorar áreas clave como la arquitectura de sistemas, la optimización de recursos y la interacción con hardware virtualizado.

El objetivo es crear una aplicación capaz de funcionar en dispositivos móviles, permitiendo a amigos, familiares y otros usuarios revivir una experiencia nostálgica de la infancia. Desde un punto de vista técnico, la emulación representa un desafío complejo, ya que involucra aspectos como la sincronización de hardware, la gestión de ciclos de reloj, el manejo de gráficos y la interpretación precisa de código máquina. El resultado no solo será un aporte académico significativo, sino también una oportunidad para compartir una parte importante de la historia de los videojuegos con las nuevas generaciones.

Agradecimientos

Este máster en Desarrollo de Software para dispositivos móviles ha sido una experiencia enriquecedora, llena de aprendizaje y crecimiento a lo largo de los últimos dos años.

Quisiera expresar mi más sincero agradecimiento a mi familia, por su constante apoyo y aliento a lo largo de este viaje. A mi pareja, Carla, cuyo respaldo y motivación fueron esenciales para continuar en los momentos más difíciles. A mis amigos, Jose Malagón y Raquel González, compañeros de grado y amigos entrañables, con quienes he mantenido una valiosa amistad. A mis compañeros de trabajo, por su paciencia y apoyo en la tarea de compaginar estudios y empleo. Finalmente, un especial agradecimiento a mi tutor, Luis Lucas Ibáñez, por su interés en mi trabajo y su constante guía a lo largo del desarrollo de este proyecto.

El éxito consiste en hacer lo que amas
y amar lo que haces.

Satoru Iwata.

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

1 Resumen

GBee es una aplicación que funciona como **emulador de Game Boy**, desarrollado específicamente de forma nativa para dispositivos **Android**. Los usuarios podrán cargar sus ROMs, guardar el estado de sus partidas, y configurar a su gusto la interfaz gráfica.

El proyecto nace de la curiosidad por entender los aspectos técnicos y arquitectónicos de una consola, y cómo estos pueden ser emulados en un entorno moderno. A lo largo del trabajo, se abordan temas clave como la gestión de la Unidad Central de Procesamiento (CPU), la Unidad de Procesamiento de Gráficos (PPU), y la sincronización de ciclos para garantizar una emulación precisa.

Abstract

GBee is an application that functions as a **Game Boy emulator**, developed natively for **Android** devices. Users can load their ROMs, save their game states, and customize the graphical interface to their liking.

The project stems from a curiosity to understand the technical and architectural aspects of a console, and how these can be emulated in a modern environment. Throughout the development, key topics such as the management of the Central Processing Unit (CPU), the Graphics Processing Unit (PPU), and cycle synchronization are addressed to ensure accurate emulation.

2 Objetivos

Un **ingeniero** debe ser capaz en todo momento de **resolver los problemas** que se le planteen por si mismo y no basarse en buscar la solución de alguien anónimo.

Las arquitecturas de las máquinas actuales son **complejas y muy potentes** para que una persona les pueda sacar todo el potencial en un corto período de tiempo. Por ello, lo ideal es empezar por una consola más antigua como punto de partida, como lo puede ser la propia **Game Boy**.

La razón de escogerla como la consola sobre la que desarrollar este proyecto ha sido **subjetiva** debido al afecto que le tengo. Perfectamente podría haber escogido cualquier otra como la *NES* o la *Master System*. Por otro lado, con la documentación de esta memoria pretendo **ser de ayuda para más personas** que se propongan en un futuro realizar un juego para dicha consola.

A grandes rasgos, los objetivos serían los siguientes:

- **Entender y replicar el funcionamiento interno de la consola Game Boy.**
- **Desarrollar una aplicación nativa en Android.**
- **Analizar librerías y frameworks existentes.**
- **Comprender y aplicar la integración de un emulador con las interfaces gráficas de Android.**

Objetivos secundarios:

- **Publicar la aplicación en la Play Store.**

3 Terminología

A lo largo del documento se van a utilizar varias nomenclaturas para hacer la lectura más sencilla:

- **GB:** Game Boy.
- **GBC:** Game Boy Color.
- **CGB:** Forma alternativa de referirse a la Game Boy Color.
- **SNES:** Super Nintendo Entertainment System.
- **SGB:** Super Game Boy. Accesorio para la SNES.
- **SGB2:** Versión mejorada de la Super Game Boy.
- **MGB:** Mini Game Boy. Forma alternativa de referirse a la Game Boy Pocket.
- **GBL:** Game Boy Light.
- **DMG:** Dot Matrix Game. Abreviatura oficial del modelo original de la Game Boy. Hace referencia a la pantalla de matriz de puntos que utilizaba la consola.
- **AGB:** Game Boy Advance.
- **AGS:** Game Boy Advance SP.
- **N64:** Nintendo 64.
- **Bit:** Unidad mínima de información empleada en informática.
- **MSB:** Most Significant Bit. El bit de mayor valor en un número binario. Es el bit 7, que representa el valor más alto (128 en decimal).
- **LSB:** Least Significant Bit. El bit de menor valor en un número binario. Es el bit 0, que representa el valor más bajo (1 en decimal).
- **Nibble:** Unidad de información equivalente a la mitad de un byte (4 bits).
- **Byte:** Unidad de información equivalente a 8 bits.
- **KiB:** Unidad de información conocida como Kibibyte, equivalente a 2^{10} bytes.
- **CPU:** Central Processing Unit. Hardware que interpreta las instrucciones del programa.
- **GPU:** Graphics Processing Unit. Hardware dedicado al procesamiento de gráficos.

- **PPU:** Picture Processing Unit. Otra manera de nombrar la GPU.
- **RAM:** Random-Access Memory. Memoria de trabajo donde almacenamos nuestras variables.
- **SRAM:** Static Random-Access Memory.
- **ROM:** Read Only Memory. Zona de memoria donde se almacena el código del programa.
- **APU:** Unidad de Procesamiento de Audio.
- **VRAM:** Video RAM. Zona de memoria utilizada por el controlador gráfico para representar información de manera visual por pantalla.
- **HRAM:** High RAM. Zona de memoria accesible en el proceso DMA.
- **DMA:** Direct Memory Access. Característica de ciertos sistemas informáticos que permite acceder a RAM a un subsistema, indepentientemente de la CPU.
- **OAM:** Object Attribute Memory. Espacio de memoria en el que se almacenan los atributos de los sprites.
- **PC:** Program Counter. Almacena la dirección de la próxima instrucción a ejecutar.
- **SP:** Stack Pointer. Apunta a la última dirección usada en la pila.
- **Sprite:** Elemento visual activo en pantalla.
- **Tile:** Conjunto de pixeles de tamaño 8x8.
- **MBC:** Memory Bank Controller. Circuito que permite gestionar la memoria de los cartuchos de Game Boy.
- **Opcode:** Instrucción de máquina que indica la operación que debe realizar el procesador.
- **Activity:** Componente de Android que representa una pantalla con la que los usuarios pueden interactuar.
- **BCD:** Binary-Coded Decimal. Sistema de representación numérica que utiliza cuatro bits para codificar cada dígito decimal, permitiendo así que los números decimales se almacenen y manipulen de manera más sencilla en sistemas digitales.
- **FIFO:** First In, First Out. Estructura de datos que organiza elementos de manera que el primero en entrar es el primero en salir, garantizando un orden de procesamiento basado en la secuencia de llegada.
- **URI:** Uniform Resource Identifier. Es una cadena de carácteres que identifican un recurso online o local.
- **Intent:** objeto en Android que se utiliza para comunicar componentes.

4 Metodología y Planificación

4.1 Metodología Aplicada

La metodología adoptada para la planificación del proyecto es ágil, un enfoque que, si bien es comúnmente utilizado en proyectos colaborativos, resulta igualmente eficaz en proyectos individuales. La clave de esta metodología radica en la gestión eficiente del tiempo, permitiendo que cada tarea sea ejecutada dentro de un plazo bien definido, con el objetivo de maximizar los resultados y cumplir con los plazos establecidos.

4.2 Estructura del Proyecto

El proyecto está dividido en distintas etapas o iteraciones, cada una de las cuales ofrece la oportunidad de aprender y aplicar nuevos conocimientos relacionados con la emulación y el desarrollo. Al final de cada iteración, se lleva a cabo una revisión exhaustiva del progreso, lo que permite identificar áreas de mejora o posibles errores, minimizando así el tiempo perdido en futuras fases del desarrollo.

Durante el proceso de desarrollo, las tareas y objetivos están en constante evolución, dado que se parte de una base de conocimientos limitada que se incrementa a medida que avanza el proyecto. Este enfoque implica que, en muchas ocasiones, lo que en un inicio parecía correctamente implementado debe ser revisado o incluso reestructurado, conforme se adquiere una comprensión más profunda de los desafíos técnicos involucrados.

El proyecto se ha dividido en las siguientes etapas/iteraciones:

- **Diseño**

  Antes de iniciar el desarrollo del emulador, es fundamental tener claridad sobre los objetivos y el alcance del proyecto. Se realizará un análisis preliminar de diversos emuladores existentes, lo cual permitirá obtener una visión más sólida y concreta de las funcionalidades y desafíos que se enfrentarán durante la implementación.

- **Aprendizaje**

  En esta fase inicial, el enfoque será comprender en profundidad las especificaciones técnicas de la consola Game Boy, desde su CPU y PPU hasta la gestión de ciclos y sus interacciones con los componentes de hardware. Se realizarán pruebas exploratorias, cuyo propósito será construir una base sólida para el desarrollo posterior del emulador.

- **Desarrollo**

Esta es la etapa central y más intensiva del proyecto. Durante el desarrollo del emulador, se implementarán las funcionalidades principales como la emulación de la CPU, la gestión de gráficos, la sincronización de ciclos, y la integración con las interfaces gráficas en Android. Al mismo tiempo, se generará la documentación técnica detallada para acompañar el progreso y justificar las decisiones tomadas durante la implementación.

- **Revisión y Maquetación**

  La fase final se centrará en la revisión exhaustiva tanto del emulador como de la documentación. Se corregirán posibles errores detectados, se optimizará el rendimiento del emulador, y se pulirá la maquetación de la memoria, asegurando que todo el trabajo cumpla con los estándares de calidad requeridos.

Las **herramientas principales** que se van a utilizar son **Trello y Toggl**. Además, como herramienta de **control de versiones**, se va a hacer uso de **Git**.

En Trello se ha creado una tabla con la que poder visualizar de manera sencilla qué tareas están pendientes de realizar, tanto del producto como de la memoria. Todas estas tareas pasarán a la fase de revisión y, una vez dado el visto bueno, terminarán en el estado de finalizado.

Figura 4.1: Tabla Trello - Iteración 0

4.3 Mínimo Producto Viable

Lo normal en todo proyecto es que ocurran imprevistos que hagan al programador perder más tiempo en una tarea o incluso paralizar por completo el proyecto. Además, esto se junta con el hecho de que aquí no hay nadie que pueda ocupar nuestro puesto mientras ese problema se soluciona. Por esta razón, es importante tener en mente un **producto mínimo viable**, con el cual obtener un producto usable de en el tiempo disponible.

5 Diseño

5.1 Mockup

5.2 Diagrama de casos de uso

6 Marco Teórico

6.1 Game Boy

La **Game Boy** es una consola portátil de **8 bits** desarrollada y fabricada por Nintendo, lanzada al mercado el **21 de abril de 1989 en Japón**, tres meses después en América y el 28 de septiembre de 1990 en Europa. Fue la segunda consola portátil de la compañía, sucesora de la familia Game & Watch.

Figura 6.1: Game Boy

Entre sus características técnicas, la Game Boy incluía un procesador Z80, una pantalla LCD monocromática con ajuste de contraste, un pad direccional de 8 direcciones, dos botones de acción (A y B) y dos botones de control (Start y Select).

A pesar de recibir críticas por el tamaño de su pantalla y su limitada paleta de colores, la Game Boy logró un rotundo éxito comercial. Su éxito se atribuye principalmente a su bajo consumo energético, ya que funcionaba con cuatro pilas AA que ofrecían una gran autonomía en comparación con consolas rivales, como la Game Gear de SEGA. Además, la consola fue lanzada junto al exitoso juego Tetris, lo que contribuyó a su popularidad tanto entre niños como entre adultos.

La longevidad de la Game Boy en el mercado se debe en gran parte al modelo de negocio que Nintendo continúa aplicando hoy en día, basado en la introducción de revisiones compatibles con versiones anteriores en lugar de lanzar consolas completamente nuevas. Entre estas revisiones destacan la Game Boy Pocket en 1996, así como la Game Boy Light y la Game Boy Color en 1998.

Figura 6.2: Versiones DMG, MGB, GBL y CGB de Game Boy

6.1.1 Especificaciones Técnicas

Cuando pensamos en programar un emulador que simule una máquina en concreto lo primero que debemos hacer es conocer exáctamente **cómo es por dentro y cómo funciona**. En la siguiente tabla quedan reflejadas todas las **características técnicas de la Game Boy**:

Características	**Game Boy (DMG)**	**Game Boy Pocket (MGB)**	**Super Game Boy (SGB)**	**Game Boy Color (CGB)**
CPU	8-bit 8080-like Sharp CPU (speculated to be a SM83 core)			
Frecuencia CPU	4.194304 MHz		Depende de la revisión	Hasta 8.388608 MHz
Work RAM	8 KiB			32 KiB (4 + 7 × 4 KiB)
Video RAM	8 KiB			16 KiB (2 × 8 KiB)
Pantalla	LCD 4.7 × 4.3 cm	LCD 4.8 × 4.4 cm	CRT TV	TFT 4.4 × 4 cm
Resolución	160 × 144		160 × 144 dentro de 256 × 224 border	160 × 144
Sprites (OBJ)	8×8 o 8×16; máximo 40 por pantalla, 10 por línea			
Paletas	BG: 1 × 4, OBJ: 2 × 3		BG/OBJ: 1 + 4 × 3, border: 4 × 15	BG: 8 × 4, OBJ: 8 × 33
Colores	4 tonos de verde	4 tonos de gris	32768 colores (15-bit RGB)	
Sincronización horizontal	9.198 KHz		Complicado[1]	9.198 KHz
Sincronización vertical	59.73 Hz		Complicado[1]	59.73 Hz

Continúa en la siguiente página

Tabla 6.1 – *Continúa de la página anterior*

Características	**Game Boy (DMG)**	**Game Boy Pocket (MGB)**	**Super Game Boy (SGB)**	**Game Boy Color (CGB)**
Sonido	4 canales con salida estéreo		4 canales GB + audio SNES	4 canales con salida estéreo
Energía	DC 6V, 0.7 W	DC 3V, 0.7 W	Alimentado por SNES	DC 3V, 0.6 W

Tabla 6.1: Especificaciones técnicas de la Game Boy.

6.1.2 Mapa de Memoria

La Game Boy dispone de 64 Kb de memoria y utiliza direcciones de memoria de 16 bits, lo que le da la posibilidad de utilizar el rango de 0x0000 hasta 0xFFFF (65.535 bytes en total). Además, esta memoria se organiza en diferentes áreas para que el procesador pueda acceder a recursos clave como el código del juego, la memoria de video, la RAM y los registros de control. Cada una de estas áreas tiene una función específica, y algunas de ellas están sujetas a restricciones o condiciones de acceso durante la ejecución del sistema.

Inicio	**Fin**	**Descripción**
0000	3FFF	16 KiB Banco ROM 00
4000	7FFF	16 KiB Banco ROM 01–NN
8000	9FFF	8 KiB RAM de Vídeo (VRAM)
A000	BFFF	8 KiB RAM Externa
C000	CFFF	4 KiB Work RAM (WRAM)
D000	DFFF	4 KiB Work RAM (WRAM)
E000	FDFF	Echo RAM (espejo de C000–DDFF)
FE00	FE9F	Memoria de atributos de objetos (OAM)
FEA0	FEFF	No usable
FF00	FF7F	Registros de entrada/salida (I/O)
FF80	FFFE	High RAM (HRAM)
FFFF	FFFF	Registro de habilitación de interrupciones (IE)

Tabla 6.2: Mapa de memoria de Game Boy

6.1.2.1 ROM: [0x0000 - 0x7FFF]

En las primeras direcciones del mapa de memoria (0x0000–0x7FFF) se encuentran los bancos de memoria ROM, que almacenan el código del juego cargado desde el cartucho. La sección entre 0x0000 y 0x3FFF es el banco fijo de 16 KiB que se carga automáticamente al encender la consola. Por último, la sección entre 0x4000 y 0x7FFF corresponde a bancos

[1]La SGB ejecuta dos consolas de forma simultánea: la Game Boy dentro del cartucho y la SNES. La SNES captura y muestra los gráficos de la Game Boy, pero las velocidades de fotogramas de ambas no se sincronizan completamente, lo que provoca duplicados y/o eliminados

de ROM adicionales que pueden ser intercambiados (si el tipo de cartucho lo permite) para ofrecer acceso a más de 32 KiB de memoria.

6.1.2.2 VRAM: [0x8000 - 0x9FFF]

La VRAM es una memoria de 8 KiB que se utiliza para almacenar datos gráficos, como los sprites y tiles que se dibujan en pantalla. En los modelos CGB, esta sección de la memoria tiene dos bancos, con la posibilidad de intercambiarlos entre ellos y almacenar gráficos adicionales.

6.1.2.3 RAM Externa: [0xA000 - 0xBFFF]

Algunos cartuchos incluyen RAM adicional que se usa principalmente para guardar datos o estados del juego, como partidas guardadas. Esta RAM externa, que varía en tamaño según el cartucho, se puede acceder a través de esta región de memoria.

6.1.2.4 Work RAM: [0xC000 - 0xDFFF]

La WRAM es un área de memoria utilizada por el sistema para almacenar variables temporales, datos en proceso, y otra información necesaria para la ejecución de los programas. Está dividida en dos secciones de 4 KiB cada una. En los modelos CGB, la segunda parte (0xD000–0xDFFF) también puede ser conmutada entre diferentes bancos para aumentar la capacidad de almacenamiento.

6.1.2.5 Echo RAM: [0xE000 - 0xFDFF]

Esta es una copia exacta de la WRAM en las direcciones 0xC000–0xDDFF. Aunque se puede acceder a ella de la misma forma que la RAM original, Nintendo prohíbe su uso, ya que podría causar errores de sincronización o conflictos de acceso.

6.1.2.6 OAM: [0xFE00 - 0xFE9F]

La OAM es un pequeño bloque de memoria dedicado a almacenar información sobre los sprites que aparecen en pantalla, como su posición, prioridad y patrones de color. Puede contener hasta un total de 40 sprites, cada uno de 8x8 u 8x16 píxeles. Sin embargo, por limitaciones de hardware, solamente se pueden mostrar 10 sprites por scanline.

6.1.2.7 No utilizable: [0xFEA0 - 0xFEFF]

Esta pequeña área de memoria no se utiliza y está reservada. Cualquier intento de leer o escribir en esta región puede provocar comportamientos inesperados en el sistema.

Si se intenta acceder a la región, devolverá un valor 0xFF cuando el OAM está bloqueado, y de lo contrario, dependerá de la revisión del hardware:

- En DMG, MGB, SGB y SGB2, las lecturas durante el bloqueo de OAM provocan corrupción de OAM. De otra forma devolverán 0x00.

- En la revisión E de CGB, y los modelos AGB, AGS y GBP, devuelve el nibble alto del byte de la dirección inferior dos veces; por ejemplo, 0xFFA0 devuelve 0xAA, 0xFFB1 devuelve 0xBB, y así sucesivamente.

- En el resto de revisiones de CGB (0-D), la región es una sección de RAM única, pero enmascarada con un valor específico.

6.1.2.8 I/O: [0xFEA0 - 0xFEFF]

Estos registros se utilizan para controlar diferentes aspectos del hardware de la Game Boy, como la pantalla, los botones de entrada, el sonido y otros componentes del sistema. Acceder a estos registros permite al software interactuar directamente con el hardware.

6.1.2.9 HRAM: [0xFF80 - 0xFFFE]

La HRAM es un pequeño bloque de memoria de alta velocidad que contiene datos críticos que el procesador necesita acceder de manera rápida y frecuente.

6.1.2.10 IE: 0xFFFF

Registro de habilitación de interrupciones (Interrupt Enable), que permite activar o desactivar interrupciones específicas del sistema, cruciales para el manejo de eventos como temporizadores o actualizaciones gráficas.

6.2 CPU

También conocida como la Unidad Central de Procesamiento, que constituye el núcleo principal del sistema. Es la encargada de ejecutar los opcodes, que definen el comportamiento del software en la consola original.

6.2.1 Diferencias

LA Game Boy cuenta con un procesador único conocido como Sharp LR35902 y, como viene indicado en su nombre, fue desarrollado por la empresa Sharp Corporation. Esta CPU era una mezcla entre la conocida Zilog Z80 y el Intel 8080, a la cual se añadieron y eliminaron distintas instrucciones. Una diferencia importante, es que la Sharp no tiene instrucciones propias de I/O, a diferencia de las otras dos. En este caso, se puede acceder a los puertos de entrada/salida mediante comandos simples de carga (LD).

A continuación se muestra una tabla con las diferencias existentes en el abanico de instrucciones:

Opcode	Z80	Sharp LR35902
08	EX AF,AF	LD (nn),SP
10	DJNZ PC+dd	STOP
22	LD (nn),HL	LDI (HL),A
2A	LD HL,(nn)	LDI A,(HL)
32	LD (nn),A	LDD (HL),A
3A	LD A,(nn)	LDD A,(HL)
D3	OUT (n),A	-
D9	EXX	RETI
DB	IN A,(n)	-
DD	<IX>prefix	-
E0	RET PO	LD (FF00+n),A
E2	JP PO,nn	LD (FF00+C),A
E3	EX (SP),HL	-
E4	CALL P0,nn	-
E8	RET PE	ADD SP,dd
EA	JP PE,nn	LD (nn),A
EB	EX DE,HL	-
EC	CALL PE,nn	-
ED	<prefix>	-
F0	RET P	LD A,(FF00+n)
F2	JP P,nn	LD A,(FF00+C)
F4	CALL P,nn	-
F8	RET M	LD HL,SP+dd
FA	JP M,nn	LD A,(nn)
FC	CALL M,nn	-
FD	<IY>prefix	-
CB 3X	SLL r/(HL)	SWAP r/(HL)

Tabla 6.3: Comparación de opcodes entre Z80 y Sharp LR35902

Los opcodes que se indican con un "-" significan que han sido eliminados. Si intentáramos utilizar los originales, la Game Boy simplemente se congelaría y habría que reiniciar el sistema.

6.2.2 Registros

Un registro es una pequeña unidad de almacenamiento en un procesador utilizada para guardar temporalmente datos o instrucciones durante la ejecución de operaciones. Los registros que se emplean son: A, F, B, C, D, E, H y L, cada uno capaz de almacenar 1 byte de información. Estos registros también pueden agruparse en pares para manejar 2 bytes: AF, BC, DE y HL. Adicionalmente, los registros SP (Stack Pointer) y PC (Program Counter) desempeñan funciones específicas dentro de la CPU.

El primer registro a destacar es el registro A, conocido como el Acumulador, donde se almacena la mayoría de los datos procesados por la CPU. Es un registro al que se le pueden

asignar valores de forma directa, al igual que ocurre con los registros B, C, D, E, H y L.

Los registros B y C son comúnmente utilizados como contadores, mientras que los registros D y E se suelen emplear en pares para almacenar direcciones de memoria, facilitando operaciones de copia de datos. Sin embargo, el uso de estos registros no está limitado a estos propósitos; su aplicación depende de las necesidades y la conveniencia del programador.

El registro F o Flags es responsable de almacenar el estado actual del procesador, y es de solo lectura. Aunque no se puede modificar directamente, su combinación con el registro A es clave para realizar diversas operaciones. Su principal utilidad radica en la evaluación de los resultados de la instrucción anterior, siendo esencial para la toma de decisiones en la ejecución del programa.

En la siguiente tabla podemos ver el desglose del valor atribuido a cada bit del registro F:

Bit	Nombre	Definición
7	Z	Zero: indica si el resultado de la operación previa ha dado como resultado 0.
6	N	Substraction: indica si la operación previa ha sido una resta.
5	H	Semi-Carry: indica si se ha producido un acarreo desde el nibble bajo al alto en la operación de suma o resta anterior.
4	C o CY	Carry: indica si se ha producido un acarreo en el bit más significativo, es decir, el resultado ha sido mayor de 0xFF.
3-0	-	No se utilizan.

Tabla 6.4: Función de los bits del registro F o Flags

Por su parte, los registros H y L se utilizan para el acceso indirecto a una dirección de memoria. Este acceso indirecto se refiere al valor de 16 bits contenido en la pareja de registros HL, y resulta especialmente útil para recorrer arrays o acceder secuencialmente a posiciones de memoria.

El registro SP (Stack Pointer) señala la posición de memoria que se utiliza durante las llamadas a subrutinas. Al ejecutar una instrucción call, la pila aumenta, y al realizar un ret, la pila disminuye, permitiendo mantener el control del flujo de ejecución.

Finalmente, el registro PC (Program Counter) indica la dirección de memoria donde se encuentra la próxima instrucción que será ejecutada por la CPU, siendo esencial para el flujo de control del programa.

6.2.3 Opcodes

Los opcodes (códigos de operación) son las instrucciones que un procesador entiende y ejecuta directamente. Representan la parte de una instrucción de máquina que especifica la operación a realizar, como cargar datos en un registro, realizar una operación matemática o mover datos entre la memoria y el procesador. Cada opcode tiene un formato específico y puede estar compuesto por varios bytes que definen tanto la operación como los operandos involucrados.

6.2.3.1 Categorías

En total, existen 510 instrucciones diferentes, las cuales pueden ser agrupadas en las siguientes categorías:

- **Operaciones de carga (LD)**
  - LD A, (HL): Carga el valor de la dirección de memoria en el registro A.
  - LD (HL), B: Carga el valor del registro B en la memoria apuntada por HL.
- **Operaciones aritméticas y lógicas**
  - ADD A, B: Suma el contenido de B al registro A.
  - AND A: Realiza una operación lógica AND en el registro A.
  - SUB A, B: Resta el valor de B del registro A.
  - XOR A: Realiza una operación XOR entre el registro A consigo mismo, lo que siempre da como resultado 0. Este comando se utiliza para limpiar o reiniciar el registro A.
  - OR A: Realiza una operación OR del registro A consigo mismo, dejando el valor de A sin cambios. No afecta el valor, pero puede modificar los flags.
  - CP A: Compara el valor del registro A con el valor de otro registro o inmediato, estableciendo los flags según el resultado de la comparación. La operación es similar a una resta (A - valor) y no modifica el contenido de A.
  - CPL: Complementa el valor del registro A, invirtiendo todos sus bits. Esta operación afecta el flag de medio acarreo (H) y establece el flag de negativo (N).
  - CCF: Cambia el estado del flag de acarreo (C). Si el flag de acarreo está establecido, se limpia; si está limpio, se establece. Esta operación no afecta a los demás flags.
  - DAA: Ajusta el contenido del registro A para que represente un valor decimal válido, teniendo en cuenta el estado de los flags de acarreo (C) y medio acarreo (H). Esta operación es útil después de realizar operaciones aritméticas en el modo BCD.
  - SCF: Establece el flag de acarreo (C) y limpia el flag de acarreo (H). Esta operación no afecta a los demás flags y es útil para preparar operaciones que requieren un estado de acarreo conocido.
- **Operaciones de control de flujo**

  - JP nn: Salta a la dirección de memoria nn.
  - CALL nn: Llama a una subrutina en la dirección nn.
  - RET: Retorna de una subrutina.

- **Operaciones de rotación y desplazamiento**
  - RL A: Rota los bits del registro A hacia la izquierda a través del carry.
  - SLA B: Desplaza los bits de B hacia la izquierda.
  - RR A: Rota los bits del registro A hacia la derecha a través del carry, trasladando el bit menos significativo al carry y el carry al bit más significativo.
  - RLC A: Rota los bits del registro A hacia la izquierda, desplazando el bit más significativo al carry y reiniciando el bit más significativo con el valor original del carry.
  - RRC A: Rota los bits del registro A hacia la derecha, moviendo el bit menos significativo al carry y llenando el bit más significativo con el valor del carry.
  - SRA B: Desplaza los bits de B hacia la derecha, manteniendo el bit más significativo (signo) constante y colocando el bit menos significativo en el carry.
  - SWAP B: Intercambia los cuatro bits más significativos con los cuatro menos significativos del registro B.
  - SRL B: Desplaza los bits de B hacia la derecha, moviendo el bit menos significativo al carry y rellenando el bit más significativo con 0.

- **Operaciones de manipulación de bits**
  - BIT 0, A: Prueba si el bit 0 de A está establecido.
  - SET 1, (HL): Establece el bit 1 en la dirección de memoria HL.
  - RES 0, A: Reinicia (pone a 0) el LSB del registro A, dejando los demás bits sin cambios.

- **Operaciones especiales de sistema**
  - NOP: No realiza ninguna operación.
  - DI: Deshabilita interrupciones.
  - EI: Habilita interrupciones.

- **Operaciones con pila**
  - PUSH BC: Empuja el contenido del registro BC en la pila.
  - POP AF: Restaura el contenido de AF desde la pila.

- **Operaciones I/O**
  - LD (FF00+n), A: Carga el valor de A en la dirección de I/O FF00+n.
  - LD A, (FF00+C): Carga el valor de la dirección FF00+C en A.

6.2.3.2 Listado

Para poder empezar a implementar todos los opcodes, debemos hacer uso de la documentación (oficial o no) que nos indique qué Byte hace referencia a qué opcode. Las siguientes tablas suelen ser de gran ayuda para verlo de forma clara y concisa:

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NOP 1 4 - - - -	LD BC,d16 3 12 - - - -	LD (BC),A 1 8 - - - -	INC BC 1 8 - - - -	INC B 1 4 Z 0 H -	DEC B 1 4 Z 1 H -	LD B,d8 2 8 - - - -	RLCA 1 4 0 0 0 C	LD (a16),SP 3 20 - - - -	ADD HL,BC 1 8 - 0 H C	LD A,(BC) 1 8 - - - -	DEC BC 1 8 - - - -	INC C 1 4 Z 0 H -	DEC C 1 4 Z 1 H -	LD C,d8 2 8 - - - -	RRCA 1 4 0 0 0 C
1x	STOP 0 2 4 - - - -	LD DE,d16 3 12 - - - -	LD (DE),A 1 8 - - - -	INC DE 1 8 - - - -	INC D 1 4 Z 0 H -	DEC D 1 4 Z 1 H -	LD D,d8 2 8 - - - -	RLA 1 4 0 0 0 C	JR r8 2 12 - - - -	ADD HL,DE 1 8 - 0 H C	LD A,(DE) 1 8 - - - -	DEC DE 1 8 - - - -	INC E 1 4 Z 0 H -	DEC E 1 4 Z 1 H -	LD E,d8 2 8 - - - -	RRA 1 4 0 0 0 C
2x	JR NZ,r8 2 12/8 - - - -	LD HL,d16 3 12 - - - -	LD (HL+),A 1 8 - - - -	INC HL 1 8 - - - -	INC H 1 4 Z 0 H -	DEC H 1 4 Z 1 H -	LD H,d8 2 8 - - - -	DAA 1 4 Z - 0 C	JR Z,r8 2 12/8 - - - -	ADD HL,HL 1 8 - 0 H C	LD A,(HL+) 1 8 - - - -	DEC HL 1 8 - - - -	INC L 1 4 Z 0 H -	DEC L 1 4 Z 1 H -	LD L,d8 2 8 - - - -	CPL 1 4 - 1 1 -
3x	JR NC,r8 2 12/8 - - - -	LD SP,d16 3 12 - - - -	LD (HL-),A 1 8 - - - -	INC SP 1 8 - - - -	INC (HL) 1 12 Z 0 H -	DEC (HL) 1 12 Z 1 H -	LD (HL),d8 2 12 - - - -	SCF 1 4 - 0 0 1	JR C,r8 2 12/8 - - - -	ADD HL,SP 1 8 - 0 H C	LD A,(HL-) 1 8 - - - -	DEC SP 1 8 - - - -	INC A 1 4 Z 0 H -	DEC A 1 4 Z 1 H -	LD A,d8 2 8 - - - -	CCF 1 4 - 0 0 C
4x	LD B,B 1 4 - - - -	LD B,C 1 4 - - - -	LD B,D 1 4 - - - -	LD B,E 1 4 - - - -	LD B,H 1 4 - - - -	LD B,L 1 4 - - - -	LD B,(HL) 1 8 - - - -	LD B,A 1 4 - - - -	LD C,B 1 4 - - - -	LD C,C 1 4 - - - -	LD C,D 1 4 - - - -	LD C,E 1 4 - - - -	LD C,H 1 4 - - - -	LD C,L 1 4 - - - -	LD C,(HL) 1 8 - - - -	LD C,A 1 4 - - - -
5x	LD D,B 1 4 - - - -	LD D,C 1 4 - - - -	LD D,D 1 4 - - - -	LD D,E 1 4 - - - -	LD D,H 1 4 - - - -	LD D,L 1 4 - - - -	LD D,(HL) 1 8 - - - -	LD D,A 1 4 - - - -	LD E,B 1 4 - - - -	LD E,C 1 4 - - - -	LD E,D 1 4 - - - -	LD E,E 1 4 - - - -	LD E,H 1 4 - - - -	LD E,L 1 4 - - - -	LD E,(HL) 1 8 - - - -	LD E,A 1 4 - - - -
6x	LD H,B 1 4 - - - -	LD H,C 1 4 - - - -	LD H,D 1 4 - - - -	LD H,E 1 4 - - - -	LD H,H 1 4 - - - -	LD H,L 1 4 - - - -	LD H,(HL) 1 8 - - - -	LD H,A 1 4 - - - -	LD L,B 1 4 - - - -	LD L,C 1 4 - - - -	LD L,D 1 4 - - - -	LD L,E 1 4 - - - -	LD L,H 1 4 - - - -	LD L,L 1 4 - - - -	LD L,(HL) 1 8 - - - -	LD L,A 1 4 - - - -
7x	LD (HL),B 1 8 - - - -	LD (HL),C 1 8 - - - -	LD (HL),D 1 8 - - - -	LD (HL),E 1 8 - - - -	LD (HL),H 1 8 - - - -	LD (HL),L 1 8 - - - -	HALT 1 4 - - - -	LD (HL),A 1 8 - - - -	LD A,B 1 4 - - - -	LD A,C 1 4 - - - -	LD A,D 1 4 - - - -	LD A,E 1 4 - - - -	LD A,H 1 4 - - - -	LD A,L 1 4 - - - -	LD A,(HL) 1 8 - - - -	LD A,A 1 4 - - - -
8x	ADD A,B 1 4 Z 0 H C	ADD A,C 1 4 Z 0 H C	ADD A,D 1 4 Z 0 H C	ADD A,E 1 4 Z 0 H C	ADD A,H 1 4 Z 0 H C	ADD A,L 1 4 Z 0 H C	ADD A,(HL) 1 8 Z 0 H C	ADD A,A 1 4 Z 0 H C	ADC A,B 1 4 Z 0 H C	ADC A,C 1 4 Z 0 H C	ADC A,D 1 4 Z 0 H C	ADC A,E 1 4 Z 0 H C	ADC A,H 1 4 Z 0 H C	ADC A,L 1 4 Z 0 H C	ADC A,(HL) 1 8 Z 0 H C	ADC A,A 1 4 Z 0 H C
9x	SUB B 1 4 Z 1 H C	SUB C 1 4 Z 1 H C	SUB D 1 4 Z 1 H C	SUB E 1 4 Z 1 H C	SUB H 1 4 Z 1 H C	SUB L 1 4 Z 1 H C	SUB (HL) 1 8 Z 1 H C	SUB A 1 4 Z 1 H C	SBC A,B 1 4 Z 1 H C	SBC A,C 1 4 Z 1 H C	SBC A,D 1 4 Z 1 H C	SBC A,E 1 4 Z 1 H C	SBC A,H 1 4 Z 1 H C	SBC A,L 1 4 Z 1 H C	SBC A,(HL) 1 8 Z 1 H C	SBC A,A 1 4 Z 1 H C
Ax	AND B 1 4 Z 0 1 0	AND C 1 4 Z 0 1 0	AND D 1 4 Z 0 1 0	AND E 1 4 Z 0 1 0	AND H 1 4 Z 0 1 0	AND L 1 4 Z 0 1 0	AND (HL) 1 8 Z 0 1 0	AND A 1 4 Z 0 1 0	XOR B 1 4 Z 0 0 0	XOR C 1 4 Z 0 0 0	XOR D 1 4 Z 0 0 0	XOR E 1 4 Z 0 0 0	XOR H 1 4 Z 0 0 0	XOR L 1 4 Z 0 0 0	XOR (HL) 1 8 Z 0 0 0	XOR A 1 4 Z 0 0 0
Bx	OR B 1 4 Z 0 0 0	OR C 1 4 Z 0 0 0	OR D 1 4 Z 0 0 0	OR E 1 4 Z 0 0 0	OR H 1 4 Z 0 0 0	OR L 1 4 Z 0 0 0	OR (HL) 1 8 Z 0 0 0	OR A 1 4 Z 0 0 0	CP B 1 4 Z 1 H C	CP C 1 4 Z 1 H C	CP D 1 4 Z 1 H C	CP E 1 4 Z 1 H C	CP H 1 4 Z 1 H C	CP L 1 4 Z 1 H C	CP (HL) 1 8 Z 1 H C	CP A 1 4 Z 1 H C
Cx	RET NZ 1 20/8 - - - -	POP BC 1 12 - - - -	JP NZ,a16 3 16/12 - - - -	JP a16 3 16 - - - -	CALL NZ,a16 3 24/12 - - - -	PUSH BC 1 16 - - - -	ADD A,d8 2 8 Z 0 H C	RST 00H 1 16 - - - -	RET Z 1 20/8 - - - -	RET 1 16 - - - -	JP Z,a16 3 16/12 - - - -	PREFIX CB 1 4 - - - -	CALL Z,a16 3 24/12 - - - -	CALL a16 3 24 - - - -	ADC A,d8 2 8 Z 0 H C	RST 08H 1 16 - - - -
Dx	RET NC 1 20/8 - - - -	POP DE 1 12 - - - -	JP NC,a16 3 16/12 - - - -		CALL NC,a16 3 24/12 - - - -	PUSH DE 1 16 - - - -	SUB d8 2 8 Z 1 H C	RST 10H 1 16 - - - -	RET C 1 20/8 - - - -	RETI 1 16 - - - -	JP C,a16 3 16/12 - - - -		CALL C,a16 3 24/12 - - - -		SBC A,d8 2 8 Z 1 H C	RST 18H 1 16 - - - -
Ex	LDH (a8),A 2 12 - - - -	POP HL 1 12 - - - -	LD (C),A 2 8 - - - -			PUSH HL 1 16 - - - -	AND d8 2 8 Z 0 1 0	RST 20H 1 16 - - - -	ADD SP,r8 2 16 0 0 H C	JP (HL) 1 4 - - - -	LD (a16),A 3 16 - - - -				XOR d8 2 8 Z 0 0 0	RST 28H 1 16 - - - -
Fx	LDH A,(a8) 2 12 - - - -	POP AF 1 12 Z N H C	LD A,(C) 2 8 - - - -	DI 1 4 - - - -		PUSH AF 1 16 - - - -	OR d8 2 8 Z 0 0 0	RST 30H 1 16 - - - -	LD HL,SP+r8 2 12 0 0 H C	LD SP,HL 1 8 - - - -	LD A,(a16) 3 16 - - - -	EI 1 4 - - - -			CP d8 2 8 Z 1 H C	RST 38H 1 16 - - - -

Figura 6.3: Set de Instrucciones

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	RLC B 2 8 Z 0 0 C	RLC C 2 8 Z 0 0 C	RLC D 2 8 Z 0 0 C	RLC E 2 8 Z 0 0 C	RLC H 2 8 Z 0 0 C	RLC L 2 8 Z 0 0 C	RLC (HL) 2 16 Z 0 0 C	RLC A 2 8 Z 0 0 C	RRC B 2 8 Z 0 0 C	RRC C 2 8 Z 0 0 C	RRC D 2 8 Z 0 0 C	RRC E 2 8 Z 0 0 C	RRC H 2 8 Z 0 0 C	RRC L 2 8 Z 0 0 C	RRC (HL) 2 16 Z 0 0 C	RRC A 2 8 Z 0 0 C
1x	RL B 2 8 Z 0 0 C	RL C 2 8 Z 0 0 C	RL D 2 8 Z 0 0 C	RL E 2 8 Z 0 0 C	RL H 2 8 Z 0 0 C	RL L 2 8 Z 0 0 C	RL (HL) 2 16 Z 0 0 C	RL A 2 8 Z 0 0 C	RR B 2 8 Z 0 0 C	RR C 2 8 Z 0 0 C	RR D 2 8 Z 0 0 C	RR E 2 8 Z 0 0 C	RR H 2 8 Z 0 0 C	RR L 2 8 Z 0 0 C	RR (HL) 2 16 Z 0 0 C	RR A 2 8 Z 0 0 C
2x	SLA B 2 8 Z 0 0 C	SLA C 2 8 Z 0 0 C	SLA D 2 8 Z 0 0 C	SLA E 2 8 Z 0 0 C	SLA H 2 8 Z 0 0 C	SLA L 2 8 Z 0 0 C	SLA (HL) 2 16 Z 0 0 C	SLA A 2 8 Z 0 0 C	SRA B 2 8 Z 0 0 0	SRA C 2 8 Z 0 0 0	SRA D 2 8 Z 0 0 0	SRA E 2 8 Z 0 0 0	SRA H 2 8 Z 0 0 0	SRA L 2 8 Z 0 0 0	SRA (HL) 2 16 Z 0 0 0	SRA A 2 8 Z 0 0 0
3x	SWAP B 2 8 Z 0 0 0	SWAP C 2 8 Z 0 0 0	SWAP D 2 8 Z 0 0 0	SWAP E 2 8 Z 0 0 0	SWAP H 2 8 Z 0 0 0	SWAP L 2 8 Z 0 0 0	SWAP (HL) 2 16 Z 0 0 0	SWAP A 2 8 Z 0 0 0	SRL B 2 8 Z 0 0 C	SRL C 2 8 Z 0 0 C	SRL D 2 8 Z 0 0 C	SRL E 2 8 Z 0 0 C	SRL H 2 8 Z 0 0 C	SRL L 2 8 Z 0 0 C	SRL (HL) 2 16 Z 0 0 C	SRL A 2 8 Z 0 0 C
4x	BIT 0,B 2 8 Z 0 1 -	BIT 0,C 2 8 Z 0 1 -	BIT 0,D 2 8 Z 0 1 -	BIT 0,E 2 8 Z 0 1 -	BIT 0,H 2 8 Z 0 1 -	BIT 0,L 2 8 Z 0 1 -	BIT 0,(HL) 2 16 Z 0 1 -	BIT 0,A 2 8 Z 0 1 -	BIT 1,B 2 8 Z 0 1 -	BIT 1,C 2 8 Z 0 1 -	BIT 1,D 2 8 Z 0 1 -	BIT 1,E 2 8 Z 0 1 -	BIT 1,H 2 8 Z 0 1 -	BIT 1,L 2 8 Z 0 1 -	BIT 1,(HL) 2 16 Z 0 1 -	BIT 1,A 2 8 Z 0 1 -
5x	BIT 2,B 2 8 Z 0 1 -	BIT 2,C 2 8 Z 0 1 -	BIT 2,D 2 8 Z 0 1 -	BIT 2,E 2 8 Z 0 1 -	BIT 2,H 2 8 Z 0 1 -	BIT 2,L 2 8 Z 0 1 -	BIT 2,(HL) 2 16 Z 0 1 -	BIT 2,A 2 8 Z 0 1 -	BIT 3,B 2 8 Z 0 1 -	BIT 3,C 2 8 Z 0 1 -	BIT 3,D 2 8 Z 0 1 -	BIT 3,E 2 8 Z 0 1 -	BIT 3,H 2 8 Z 0 1 -	BIT 3,L 2 8 Z 0 1 -	BIT 3,(HL) 2 16 Z 0 1 -	BIT 3,A 2 8 Z 0 1 -
6x	BIT 4,B 2 8 Z 0 1 -	BIT 4,C 2 8 Z 0 1 -	BIT 4,D 2 8 Z 0 1 -	BIT 4,E 2 8 Z 0 1 -	BIT 4,H 2 8 Z 0 1 -	BIT 4,L 2 8 Z 0 1 -	BIT 4,(HL) 2 16 Z 0 1 -	BIT 4,A 2 8 Z 0 1 -	BIT 5,B 2 8 Z 0 1 -	BIT 5,C 2 8 Z 0 1 -	BIT 5,D 2 8 Z 0 1 -	BIT 5,E 2 8 Z 0 1 -	BIT 5,H 2 8 Z 0 1 -	BIT 5,L 2 8 Z 0 1 -	BIT 5,(HL) 2 16 Z 0 1 -	BIT 5,A 2 8 Z 0 1 -
7x	BIT 6,B 2 8 Z 0 1 -	BIT 6,C 2 8 Z 0 1 -	BIT 6,D 2 8 Z 0 1 -	BIT 6,E 2 8 Z 0 1 -	BIT 6,H 2 8 Z 0 1 -	BIT 6,L 2 8 Z 0 1 -	BIT 6,(HL) 2 16 Z 0 1 -	BIT 6,A 2 8 Z 0 1 -	BIT 7,B 2 8 Z 0 1 -	BIT 7,C 2 8 Z 0 1 -	BIT 7,D 2 8 Z 0 1 -	BIT 7,E 2 8 Z 0 1 -	BIT 7,H 2 8 Z 0 1 -	BIT 7,L 2 8 Z 0 1 -	BIT 7,(HL) 2 16 Z 0 1 -	BIT 7,A 2 8 Z 0 1 -
8x	RES 0,B 2 8 - - - -	RES 0,C 2 8 - - - -	RES 0,D 2 8 - - - -	RES 0,E 2 8 - - - -	RES 0,H 2 8 - - - -	RES 0,L 2 8 - - - -	RES 0,(HL) 2 16 - - - -	RES 0,A 2 8 - - - -	RES 1,B 2 8 - - - -	RES 1,C 2 8 - - - -	RES 1,D 2 8 - - - -	RES 1,E 2 8 - - - -	RES 1,H 2 8 - - - -	RES 1,L 2 8 - - - -	RES 1,(HL) 2 16 - - - -	RES 1,A 2 8 - - - -
9x	RES 2,B 2 8 - - - -	RES 2,C 2 8 - - - -	RES 2,D 2 8 - - - -	RES 2,E 2 8 - - - -	RES 2,H 2 8 - - - -	RES 2,L 2 8 - - - -	RES 2,(HL) 2 16 - - - -	RES 2,A 2 8 - - - -	RES 3,B 2 8 - - - -	RES 3,C 2 8 - - - -	RES 3,D 2 8 - - - -	RES 3,E 2 8 - - - -	RES 3,H 2 8 - - - -	RES 3,L 2 8 - - - -	RES 3,(HL) 2 16 - - - -	RES 3,A 2 8 - - - -
Ax	RES 4,B 2 8 - - - -	RES 4,C 2 8 - - - -	RES 4,D 2 8 - - - -	RES 4,E 2 8 - - - -	RES 4,H 2 8 - - - -	RES 4,L 2 8 - - - -	RES 4,(HL) 2 16 - - - -	RES 4,A 2 8 - - - -	RES 5,B 2 8 - - - -	RES 5,C 2 8 - - - -	RES 5,D 2 8 - - - -	RES 5,E 2 8 - - - -	RES 5,H 2 8 - - - -	RES 5,L 2 8 - - - -	RES 5,(HL) 2 16 - - - -	RES 5,A 2 8 - - - -
Bx	RES 6,B 2 8 - - - -	RES 6,C 2 8 - - - -	RES 6,D 2 8 - - - -	RES 6,E 2 8 - - - -	RES 6,H 2 8 - - - -	RES 6,L 2 8 - - - -	RES 6,(HL) 2 16 - - - -	RES 6,A 2 8 - - - -	RES 7,B 2 8 - - - -	RES 7,C 2 8 - - - -	RES 7,D 2 8 - - - -	RES 7,E 2 8 - - - -	RES 7,H 2 8 - - - -	RES 7,L 2 8 - - - -	RES 7,(HL) 2 16 - - - -	RES 7,A 2 8 - - - -
Cx	SET 0,B 2 8 - - - -	SET 0,C 2 8 - - - -	SET 0,D 2 8 - - - -	SET 0,E 2 8 - - - -	SET 0,H 2 8 - - - -	SET 0,L 2 8 - - - -	SET 0,(HL) 2 16 - - - -	SET 0,A 2 8 - - - -	SET 1,B 2 8 - - - -	SET 1,C 2 8 - - - -	SET 1,D 2 8 - - - -	SET 1,E 2 8 - - - -	SET 1,H 2 8 - - - -	SET 1,L 2 8 - - - -	SET 1,(HL) 2 16 - - - -	SET 1,A 2 8 - - - -
Dx	SET 2,B 2 8 - - - -	SET 2,C 2 8 - - - -	SET 2,D 2 8 - - - -	SET 2,E 2 8 - - - -	SET 2,H 2 8 - - - -	SET 2,L 2 8 - - - -	SET 2,(HL) 2 16 - - - -	SET 2,A 2 8 - - - -	SET 3,B 2 8 - - - -	SET 3,C 2 8 - - - -	SET 3,D 2 8 - - - -	SET 3,E 2 8 - - - -	SET 3,H 2 8 - - - -	SET 3,L 2 8 - - - -	SET 3,(HL) 2 16 - - - -	SET 3,A 2 8 - - - -
Ex	SET 4,B 2 8 - - - -	SET 4,C 2 8 - - - -	SET 4,D 2 8 - - - -	SET 4,E 2 8 - - - -	SET 4,H 2 8 - - - -	SET 4,L 2 8 - - - -	SET 4,(HL) 2 16 - - - -	SET 4,A 2 8 - - - -	SET 5,B 2 8 - - - -	SET 5,C 2 8 - - - -	SET 5,D 2 8 - - - -	SET 5,E 2 8 - - - -	SET 5,H 2 8 - - - -	SET 5,L 2 8 - - - -	SET 5,(HL) 2 16 - - - -	SET 5,A 2 8 - - - -
Fx	SET 6,B 2 8 - - - -	SET 6,C 2 8 - - - -	SET 6,D 2 8 - - - -	SET 6,E 2 8 - - - -	SET 6,H 2 8 - - - -	SET 6,L 2 8 - - - -	SET 6,(HL) 2 16 - - - -	SET 6,A 2 8 - - - -	SET 7,B 2 8 - - - -	SET 7,C 2 8 - - - -	SET 7,D 2 8 - - - -	SET 7,E 2 8 - - - -	SET 7,H 2 8 - - - -	SET 7,L 2 8 - - - -	SET 7,(HL) 2 16 - - - -	SET 7,A 2 8 - - - -

Figura 6.4: Set de Instrucciones Extendidas

Con estas tablas obtenemos las siguientes características de cada instrucción:

- **Byte** asignado.

- **Ciclos** de reloj.
- **Flags** a ignorar, actualizar o resetear.
- **Categoría**, agrupados por colores.
- **Longitud**, es decir, los bytes que la instrucción va a ocupar en memoria.

6.2.4 Ciclos

Los ciclos de CPU son unidades de tiempo en las que se realizan instrucciones. Son utilizadas para saber con exactitud cuánto tiempo dura la ejecución de una instrucción concreta en el hardware original. Hay dos conceptos distintos en este contexto: ciclos de máquina y ciclos de reloj.

Cada componente, como la CPU, la memoria, el PPU y el APU, opera en función de los ciclos de reloj. La coordinación precisa entre estos módulos asegura que las instrucciones se ejecuten en el momento adecuado y que los datos se transfieran de manera eficiente.

Por ejemplo, el procesador necesita esperar que el PPU complete la renderización de un cuadro antes de actualizar la pantalla, lo que implica un control cuidadoso del tiempo. Si un módulo se desincroniza, puede resultar en fallos gráficos, sonido entrecortado o un rendimiento general deficiente. Por lo tanto, la medición de ciclos de reloj permite establecer un ritmo de operación coherente, asegurando que todos los componentes funcionen armónicamente, lo que es fundamental para la experiencia de juego fluida y efectiva que caracteriza a la Game Boy.

Para la Game Boy, 1 ciclo de máquina equivale a 4 ciclos de reloj.

6.2.4.1 Ciclos de Máquina

Los ciclos de máquina se refieren a la cantidad de ciclos de CPU necesarios para ejecutar una instrucción específica. Cada instrucción puede requerir un número diferente de ciclos de máquina, dependiendo de su complejidad.

6.2.4.2 Ciclos de Reloj

Los ciclos de reloj, por otro lado, son las unidades de tiempo que marcan el ritmo del funcionamiento del procesador. Cada ciclo de reloj representa un pulso generado por un oscilador interno en el procesador, que sincroniza las operaciones del mismo. La frecuencia del reloj, medida en hertzios (Hz), determina cuántos ciclos de reloj se producen por segundo.

6.3 ROM

6.3.1 Secuencia de Arranque

Existe una secuencia de arranque, conocido como **Boot ROM**, guardado dentro de la propia CPU. Esta secuencia de arranque comienza su ejecución en la dirección de memoria

0x000, y no en la oficial de 0x100. Este programa es responsable de la animación de arranque que se reproduce antes de que el control sea transferido a la ROM del cartucho, además de inicializar distintos registros y direcciones de memoria.

Existen en total 9 programas de boot (conocidos hasta la fecha):

Nombre	**Tamaño (bytes)**
DMG0	256
DMG	256
MGB	256
SGB	256
SGB2	256
CGB0	256 + 1792
CGB	256 + 1792
AGB0	256 + 1792
AGB	256 + 1792

Tabla 6.5: Resumen de las variaciones de las secuencias de arranque

6.3.1.1 DMG / MGB

Lo primero que hacen es leer el logo desde el cartucho, descomprimirlo en VRAM y comenzar a desplazarlo lentamente hacia abajo. Dado que las lecturas de un cartucho ausente generalmente devuelven 0xFF, esto explica por qué, al encender la consola sin un cartucho, aparece un cuadro negro desplazándose. Además, conexiones defectuosas o sucias pueden hacer que los datos leídos se corrompan, lo que resulta en un logo desordenado.

Una vez que el logo ha terminado de desplazarse, la boot ROM reproduce el famoso sonido *"ba-ding!"* y vuelve a leer el logo, esta vez comparándolo con una copia almacenada en ROM. También calcula el checksum del encabezado y lo compara con el checksum almacenado en el encabezado. Si alguna de estas verificaciones falla, la boot ROM se bloquea y nunca se transfiere el control a la ROM del cartucho.

Finalmente, la boot ROM escribe en el registro BANK en 0xFF50, lo que desasigna la boot ROM.

Las diferencias con DMG0 (algunos primeros modelos de DMG), es que al fallar el checksum, la pantalla empieza a parpadear a la hora de bloquear la ROM, y que el logo de Nintendo no tiene el símbolo ®.

6.3.1.2 CGB / AGB

Estas secuencias de arranque son mucho más complejas, en especial por su comportamiento de retro-compatibilidad.

La ROM de arranque es más grande. Aún debe ser mapeada comenzando en 0x0000, ya que es donde comienza la CPU, pero también debe acceder al encabezado del cartucho en 0x0100-0x014F. Por lo tanto, la ROM de arranque se divide en dos partes: una de 0x0000-0x00FF y otra de 0x0200-0x08FF.

Primero, las ROMs de arranque descomponen el logo de Nintendo en VRAM, al igual que los modelos antiguos, y copian el logo a un búfer en HRAM al mismo tiempo.

Luego, el logo se lee y descomprime nuevamente, pero sin redimensionamiento, lo que produce un logo mucho más pequeño colocado debajo del gran *"GAME BOY"*. La ROM de arranque luego configura las paletas de compatibilidad, como se describe más adelante, y reproduce la animación del logo con el sonido de *"ba-ding!"*.

Durante la animación del logo, se permite al usuario elegir una paleta para anular la seleccionada para compatibilidad (con distintas secuencias de botones). Cada nueva elección evita que la animación termine durante 30 fotogramas, lo que retrasa el checksum y la transición final.

Finalmente, la ROM de arranque desvanece todas las paletas de Background a blanco y establece el hardware en modo de compatibilidad (recordemos que existen juegos como Pokémon Oro/Plata que son de GBC pero se pueden jugar en DMG). En función del valor del byte de compatibilidad CGB, los valores a insertar en distintos registros de CGB variarán.

6.3.2 Cabecera del Cartucho

Todos los cartuchos de Game Boy contienen una cabecera ubicada en el rango de direcciones 0x100-0x14F. Esta cabecera contiene información esencial para el funcionamiento del juego y del sistema, permitiendo a los desarrolladores configurar diversos parámetros que describen las características del cartucho. Entre estos parámetros se incluyen el título del juego, el código de licencia, el tipo de cartucho (que define si utiliza RAM, batería, o expansión como MBC), y otros datos relevantes. A continuación, se detallan los registros presentes en este rango de direcciones:

6.3.2.1 0x0100 - 0x0103: Punto de entrada

Después de mostrar el logotipo de Nintendo, el PC salta a la dirección 0x0100, para a posterior saltar al inicio del programa del juego. La mayoría de los desarrolladores llenan esta área de 4 bytes con una instrucción NOP seguida de un JP 0x0150.

6.3.2.2 0x0104 - 0x0133: Nintendo logo

Esta área contiene una imagen en mapa de bits que se muestra cuando se enciende la consola. Debe coincidir con el siguiente volcado (en hexadecimal); de lo contrario, la ROM de arranque no permitirá que el juego se ejecute:

Código 6.1: Nintendo Logo - Mapa de Bits

```
CE ED 66 66 CC 0D 00 0B 03 73 00 83 00 0C 00 0D
00 08 11 1F 88 89 00 0E DC CC 6E E6 DD DD D9 99
BB BB 67 63 6E 0E EC CC DD DC 99 9F BB B9 33 3E
```

La forma en la que estos bytes se decodifican es la siguiente:

- Los bytes en el rango 0x104-0x011B representan la mitad superior del logo, mientras que los del rango 0x11C-0x133 representan la mitad inferior.
- Por cada byte, cada nibble codifica 4 píxeles. Un pixel está encendido si su bit correspondiente tiene un valor de 1.
- Cada dos bytes componen un grupo, el cual representa una parte (inferior o superior) de una letra del logo.

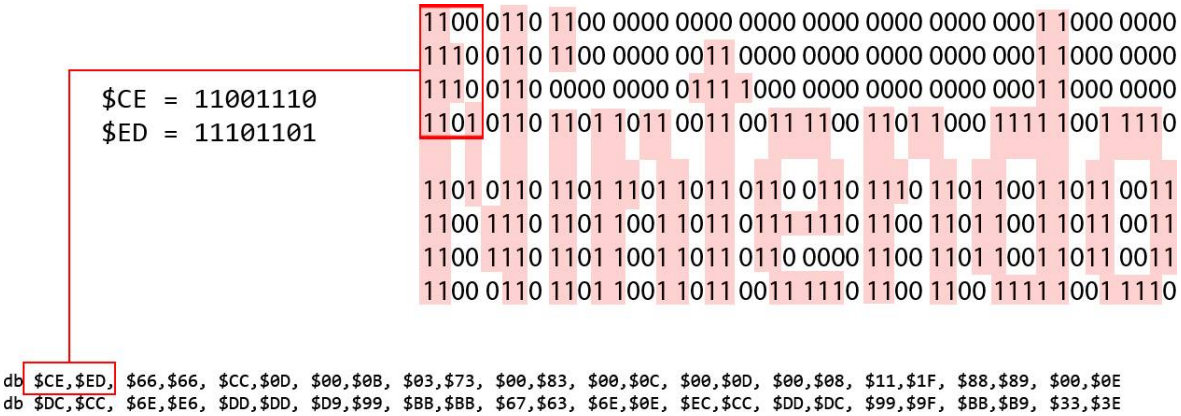

Figura 6.5: Decodificación del logo de Nintendo

El procedimiento de arranque de la Game Boy primero muestra el logo y luego verifica que coincida con el volcado anterior (conocido como *checksum*). Si no coincide, la ROM de arranque se bloquea.

Los modelos a partir del CGB solo verifican la mitad superior (los primeros 18 bytes).

6.3.2.3 0x0134 - 0x0143: Título

Esta región de bytes contienen el título del juego, con carácteres ASCII y completamente en mayúsculas. Si el título tiene menos de 16 carácteres, el resto de bytes deberán rellenarse con 0x00's.

En versiones posteriores de los cartuchos, partes de esta área tienen un significado diferente, lo que reduce el tamaño real a 15 u 11 carácteres (siempre empezando en la dirección 0x134).

6.3.2.4 0x013F - 0x0142: Código de fabricante

En los cartuchos más antiguos, estos bytes formaban parte del título. En los cartuchos más nuevos, contienen un código de fabricante de 4 caracteres (en mayúsculas ASCII). Se

desconoce su propósito.

6.3.2.5 0x0143: CGB Flag

Este byte, al igual que con el código de fabricante, formaba parte del título. Los modelos CGB y posteriores lo interpretan para decidir si habilitan el modo Color (”Modo CGB”) o si retroceden al modo de compatibilidad monocromático (”Modo no CGB”).

Los valores más típicos son:

- 0x00: No indicaría nada realmente, lo que implica que el juego solamente funciona en modelos DMG/MGB.
- 0x80: Indica que el juego admite mejoras propias del modelo CGB, pero es retrocompatible con modelos anteriores (DMG, MGB, etc...).
- 0xC0: El juego solamente funciona en CGB.

Valores que tengan activados los bits 7, 3 o 2, activarán un estado poco utilizado, conocido como ”Modo PGB” (Pseudo Game Boy Mode). Es una especie de modo intermedio que brinda compatibilidad con juegos diseñados para el Game Boy Color cuando se ejecutan en un Game Boy original o en un Super Game Boy, pero con mejoras visuales muy limitadas en comparación con el modo CGB completo.

Hay poca información al respecto de este modo y varias fuentes agradecen su estudio y documentación.

6.3.2.6 0x0144 – 0x0145: Nuevo código de licencia

Esta región contiene un código representado por dos carácteres ASCII, el cual indica el editor/desarrollador del juego. Solamente se utiliza en caso de que el antiguo código de licencia tenga un valor de 0x33 (suele ser el caso para los juegos publicados después de la salida de SGB).

Los códigos son los siguientes:

Código	**Editor**
00	Ninguno
01	Nintendo Research & Development 1
08	Capcom
13	EA (Electronic Arts)
18	Hudson Soft
19	B-AI
20	KSS
22	Oficina de Planificación WADA
24	PCM Complete
25	San-X

28	Kemco
29	SETA Corporation
30	Viacom
31	Nintendo
32	Bandai
33	Ocean Software/Acclaim Entertainment
34	Konami
35	HectorSoft
37	Taito
38	Hudson Soft
39	Banpresto
41	Ubi Soft
42	Atlus
44	Malibu Interactive
46	Angel
47	Bullet-Proof Software
49	Irem
50	Absolute
51	Acclaim Entertainment
52	Activision
53	Sammy USA Corporation
54	Konami
55	Hi Tech Expressions
56	LJN
57	Matchbox
58	Mattel
59	Milton Bradley Company
60	Titus Interactive
61	Virgin Games Ltd.
64	Lucasfilm Games
67	Ocean Software
69	EA (Electronic Arts)
70	Infogrames
71	Interplay Entertainment
72	Broderbund
73	Sculptured Software
75	The Sales Curve Limited
78	THQ
79	Accolade
80	Misawa Entertainment
83	lozc
86	Tokuma Shoten
87	Tsukuda Original

91	Chunsoft Co.
92	Video System
93	Ocean Software/Acclaim Entertainment
95	Varie
96	Yonezawa/s'pal
97	Kaneko
99	Pack-In-Video
9H	Bottom Up
A4	Konami (Yu-Gi-Oh!)
BL	MTO
DK	Kodansha

Tabla 6.6: Código de licencia y su editor.

6.3.2.7 0x0146: SGB Flag

Especifica si el juego es compatible con funciones del modelo SGB. El SGB ignorará cualquier transferencia de datos si el valor de este byte no es 0x03.

6.3.2.8 0x0147: Tipo de cartucho

Especifica qué tipo de hardware está presente en el cartucho (para ser más especifico, su *mapper*).

Los mappers son los siguientes:

Código	**Tipo**
0x00	ROM ONLY
0x01	MBC1
0x02	MBC1+RAM
0x03	MBC1+RAM+BATTERY
0x05	MBC2
0x06	MBC2+BATTERY
0x08	ROM+RAM
0x09	ROM+RAM+BATTERY
0x0B	MMM01
0x0C	MMM01+RAM
0x0D	MMM01+RAM+BATTERY
0x0F	MBC3+TIMER+BATTERY
0x10	MBC3+TIMER+RAM+BATTERY
0x11	MBC3
0x12	MBC3+RAM
0x13	MBC3+RAM+BATTERY
0x19	MBC5

0x1A	MBC5+RAM
0x1B	MBC5+RAM+BATTERY
0x1C	MBC5+RUMBLE
0x1D	MBC5+RUMBLE+RAM
0x1E	MBC5+RUMBLE+RAM+BATTERY
0x20	MBC6
0x22	MBC7+SENSOR+RUMBLE+RAM+BATTERY
0xFC	POCKET CAMERA
0xFD	BANDAI TAMA5
0xFE	HuC3
0xFF	HuC1+RAM+BATTERY

Tabla 6.7: Tipos de mapeadores (MBC) y sus códigos

6.3.2.9 0x0148: Tamaño de ROM

Indica el tamaño de ROM en el cartucho. En la mayoría de casos, se calcula como $32KiB * (1 << valor)$. Los valores conocidos son:

Valor	Tamaño	Número de bancos
0x00	32 KiB	2
0x01	64 KiB	4
0x02	128 KiB	8
0x03	256 KiB	16
0x04	512 KiB	32
0x05	1 MiB	64
0x06	2 MiB	128
0x07	4 MiB	256
0x08	8 MiB	512

Tabla 6.8: Tamaño y número de bancos de ROM según el valor especificado.

6.3.2.10 0x0149: Tamaño de RAM

Indica el tamaño de RAM (si lo hay). Si el tipo de cartucho no incluye "RAM" en su nombre, el valor de este registro debe ser 0x00.

Los posibles tamaños son los siguientes:

Código	Tamaño de SRAM	Comentario
0x00	0	Sin RAM
0x01	–	No utilizado
0x02	8 KiB	1 banco
0x03	32 KiB	4 bancos de 8 KiB cada uno
0x04	128 KiB	16 bancos de 8 KiB cada uno
0x05	64 KiB	8 bancos de 8 KiB cada uno

Tabla 6.9: Tamaño de SRAM según el código del cartucho.

El valor 0x01 aparece en algunos documentos no oficiales con un tamaño de 2KiB. Sin embargo, jamás se llegó a utilizar un chip de RAM con este tamaño. Algunas ROMs de dominio público utilizan este valor, aunque en su código no hacen uso de ningún tipo de RAM de cartucho.

6.3.2.11 0x014A: Destino

Especifica si el juego está destinado a ser vendido en Japón o en cualquier otro lugar del mundo. Existen dos posibles valores:

- 0x00: Japón. Se puede vender en el extranjero.
- 0x01: Solamente en el extranjero.

6.3.2.12 0x014B: Antiguo código de licencia

Utilizado en cartuchos anteriores al lanzamiento del SGB. Al igual que el nuevo (0x0144-0x0145), especifica el editor/publisher. Si el valor es 0x33, se deberán utilizar los nuevos códigos.

A continuación la lista de códigos y sus editores:

Código	Editor
00	Ninguno
01	Nintendo
08	Capcom
09	HOT-B
0A	Jaleco
0B	Coconuts Japan
0C	Elite Systems
13	EA (Electronic Arts)
18	Hudson Soft
19	ITC Entertainment
1A	Yanoman
1D	Japan Clary
1F	Virgin Games Ltd.3

24	PCM Complete
25	San-X
28	Kemco
29	SETA Corporation
30	Infogrames5
31	Nintendo
32	Bandai
33	Se debe usar el nuevo código de licencia.
34	Konami
35	HectorSoft
38	Capcom
39	Banpresto
3C	Entertainment Interactive (stub)
3E	Gremlin
41	Ubi Soft1
42	Atlus
44	Malibu Interactive
46	Angel
47	Spectrum HoloByte
49	Irem
4A	Virgin Games Ltd.3
4D	Malibu Interactive
4F	U.S. Gold
50	Absolute
51	Acclaim Entertainment
52	Activision
53	Sammy USA Corporation
54	GameTek
55	Park Place13
56	LJN
57	Matchbox
59	Milton Bradley Company
5A	Mindscape
5B	Romstar
5C	Naxat Soft14
5D	Tradewest
60	Titus Interactive
61	Virgin Games Ltd.3
67	Ocean Software
69	EA (Electronic Arts)
6E	Elite Systems
6F	Electro Brain
70	Infogrames5

71	Interplay Entertainment
72	Broderbund
73	Sculptured Software6
75	The Sales Curve Limited7
78	THQ
79	Accolade15
7A	Triffix Entertainment
7C	MicroProse
7F	Kemco
80	Misawa Entertainment
83	LOZC G.
86	Tokuma Shoten
8B	Bullet-Proof Software2
8C	Vic Tokai Corp.16
8E	Ape Inc.17
8F	I'Max18
91	Chunsoft Co.8
92	Video System
93	Tsubaraya Productions
95	Varie
96	Yonezawa19/S'Pal
97	Kemco
99	Arc
9A	Nihon Bussan
9B	Tecmo
9C	Imagineer
9D	Banpresto
9F	Nova
A1	Hori Electric
A2	Bandai
A4	Konami
A6	Kawada
A7	Takara
A9	Technos Japan
AA	Broderbund
AC	Toei Animation
AD	Toho
AF	Namco
B0	Acclaim Entertainment
B1	ASCII Corporation or Nexsoft
B2	Bandai
B4	Square Enix
B6	HAL Laboratory

B7	SNK
B9	Pony Canyon
BA	Culture Brain
BB	Sunsoft
BD	Sony Imagesoft
BF	Sammy Corporation
C0	Taito
C2	Kemco
C3	Square
C4	Tokuma Shoten
C5	Data East
C6	Tonkin House
C8	Koei
C9	UFL
CA	Ultra Games
CB	VAP, Inc.
CC	Use Corporation
CD	Meldac
CE	Pony Canyon
CF	Angel
D0	Taito
D1	SOFEL (Software Engineering Lab)
D2	Quest
D3	Sigma Enterprises
D4	ASK Kodansha Co.
D6	Naxat Soft14
D7	Copya System
D9	Banpresto
DA	Tomy
DB	LJN
DD	Nippon Computer Systems
DE	Human Ent.
DF	Altron
E0	Jaleco
E1	Towa Chiki
E2	Yutaka
E3	Varie
E5	Epoch
E7	Athena
E8	Asmik Ace Entertainment
E9	Natsume
EA	King Records
EB	Atlus

EC	Epic/Sony Records
EE	IGS
F0	A Wave
F3	Extreme Entertainment
FF	LJN

Tabla 6.10: Antiguos códigos de licencia y su editor.

6.3.2.13 0x014C: Número de versión de ROM

Este byte especifica el número de versión del juego. Generalmente es 0x00. Puede ser útil para los desarrolladores y emuladores, por ejemplo, para aplicar parches o actualizaciones específicas que se hayan diseñado para esa versión (importante en juegos que recibieron revisiones).

6.3.2.14 0x014D: Checksum

Este byte contiene una suma de verificación de 8 bits calculada a partir de los bytes de cabecera del cartucho 0x0134–0x014C. Si el byte en este registro no coincide con los 8 bits inferiores de la suma de verificación, la ROM de arranque se bloqueará y el programa en el cartucho no se ejecutará.

6.3.2.15 0x014E - 0x014F: Checksum global

Estos bytes contienen una suma de verificación de 16 bits que se calcula simplemente como la suma de todos los bytes de la ROM del cartucho (excepto estos dos bytes).

Solamente se ha llegado a utilizar en Pokémon Stadium (N64) para detectar errores del Transfer Pak con los Pokémon Verde/Rojo/Azul/Amarillo (DMG) y Plata/Oro/Cristal (CGB).

6.4 MBCs

Los MBCs se utilizan para expandir la memoria limitada de la Game Boy, tanto en términos de ROM como de RAM. Estos controladores son chips que se encuentran en el cartucho del juego, no en la consola misma. El MBC1 es el chip más antiguo, lanzado en 1989 junto a la propia consola. En contraste, el MBC7 es el más reciente, con un lanzamiento estimado en 1997 junto a la Game Boy Color.
Cada juego especifica qué tipo de controlador utiliza mediante el registro 0x0147, como se ha visto en el apartado anterior.

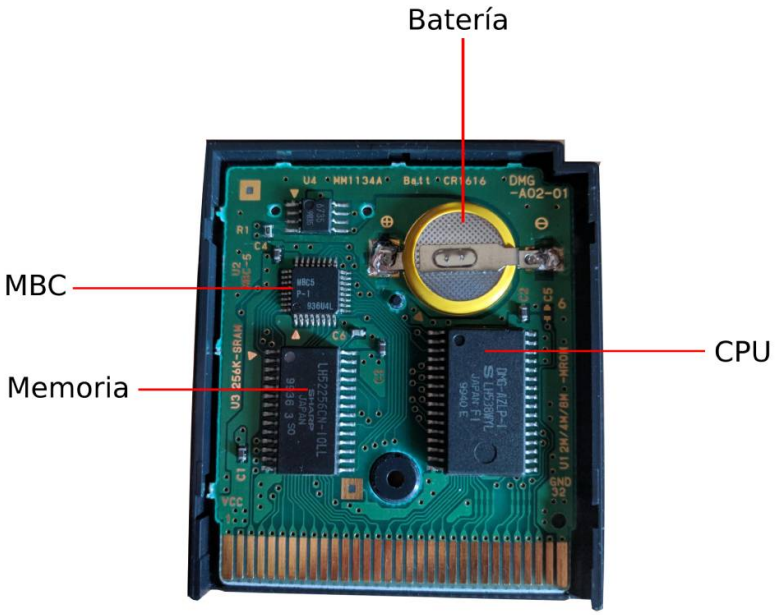

Figura 6.6: Cartucho de Game Boy MBC5

6.4.1 No MBC: 0x00

Los juegos pequeños, que tienen un tamaño de ROM de hasta 32 KiB, no necesitan un utilizar bancos de memoria. En su lugar, la ROM se coloca directamente en la memoria en el rango de direcciones 0x0000-0x7FFF. Además, si se requiere RAM adicional (hasta 8 KiB), se puede conectar en el rango 0xA000-0xBFFF utilizando un circuito lógico.

6.4.2 MBC1: 0x01-0x03

Es el modelo de chip más antiguo. Pese a ello, trabaja de forma muy similar a los más recientes, lo que facilitó la actualización de los juegos o la posibilidad de hacerlos retro-compatibles.

En la configuración predeterminada (valor 0x00) el MBC1 soporta hasta 512KiB de ROM y un total de 32KiB de RAM mediante el uso de bancos. Esta configuración puede cambiar, por ejemplo para utilizar la parte de RAM como ROM, aumentando esta última hasta 2MiBs. Es importante tener en cuenta que la memoria en el rango de direcciones 0x0000–0x7FFF se usa tanto para leer desde la ROM como para escribir en los registros de control del MBC.

6.4.2.1 Memoria

6.4.2.1.1 Banco de ROM 0x00: 0x0000-0x3FFF Esta región usualmente contiene los primeros 16KiB de ROM del cartucho (suele ser fija, es decir, no cambia).

6.4.2.1.2 Bancos de ROM 0x01-0x7F: 0x4000-0x7FFF Esta región puede contener cualquiera de los bancos de 16KiB posibles. La elección del banco se realiza a través del registro

0x2000-0x3FFF.

6.4.2.1.3 Banco de RAM 0x00-0x03: 0xA000-0xBFFF Esta región se utiliza para leer o escribir en la RAM externa presente en algunos cartuchos (generalmente de tipo 0x02 o 0x03). El acceso a esta RAM solo es posible si está activada mediante la escritura en los registros de control del MBC; de lo contrario, las lecturas devolverán un valor indefinido (comúnmente 0xFF) y las escrituras serán ignoradas.

Los tamaños disponibles de RAM son 8 KiB (que cubre toda la región mapeada) o 32 KiB, distribuidos en 4 bancos de 8 KiB cada uno. La segunda opción solo está disponible en cartuchos con ROM de hasta 512 KiB.

La RAM externa suele estar alimentada por una batería, lo que permite el almacenamiento de datos incluso cuando la consola está apagada. Esta batería, normalmente una celda de botón soldada en la placa del cartucho, garantiza la persistencia de los datos. Además, dado que la velocidad de la RAM externa es comparable a la RAM interna de la Game Boy, algunos desarrolladores la utilizan como memoria de trabajo adicional (WRAM), aunque esto puede limitar el espacio disponible para el almacenamiento de datos.

7 Desarrollo

El desarrollo del emulador se estructura en diversas fases que abordan los principales componentes de la consola, comenzando por la CPU, continuando con la gestión de gráficos (GPU/PPU) y memoria (RAM/ROM), y concluyendo con la implementación de las interfaces de entrada y salida (I/O). Cada uno de estos módulos es fundamental para asegurar una emulación fiel al hardware original, por lo que se prestará especial atención a la precisión y al rendimiento.

La primera fase se centrará en la implementación de la CPU, que es responsable de ejecutar las instrucciones del juego. Cada paso del desarrollo irá acompañado de pruebas y validaciones para garantizar que el emulador reproduzca el comportamiento de la consola original de manera eficiente. Esto último se conseguirá mediante la implementación de pruebas unitarias que verifiquen el correcto comportamiento de los opcodes.

7.1 Módulos / Estructura del Proyecto

El emulador que se va a implementar constará de distintos módulos, cada uno con una función específica y clara, lo que facilitará tanto su comprensión como el desarrollo de cada uno de los aspectos del emulador. A continuación, se describen brevemente los módulos principales del proyecto:

- **Emulator**: Este módulo centraliza la gestión de los otros componentes, coordinando su interacción y asegurando que el ciclo de emulación se ejecute correctamente. Controla el flujo general del programa.
- **CPU**: Responsable de la ejecución de las instrucciones. Se encarga de la implementación del conjunto de instrucciones de la Game Boy y la simulación de los registros, el Program Counter (PC), el Stack Pointer (SP) y las operaciones con la ALU.
- **Memory**: Gestiona el acceso a la memoria principal del sistema. Proporciona una interfaz para leer y escribir datos en las distintas áreas de la memoria del emulador, incluyendo ROM, RAM y áreas de E/S.
- **ROM**: Módulo encargado de cargar y gestionar la memoria ROM, que contiene el código del juego o programa a emular. Lee los datos directamente desde el archivo del juego y los pone a disposición del emulador.
- **RAM**: Controla la memoria de acceso aleatorio del sistema, donde se almacenan temporalmente datos durante la ejecución del programa. Es volátil y se borra cada vez que se reinicia el sistema.

- **PPU (Pixel Processing Unit)**: Se encarga de la representación gráfica. Simula la unidad de procesamiento de píxeles de la consola, manejando la creación y renderización de sprites y fondos en pantalla.
- **Timer**: Simula los temporizadores de la consola, necesarios para sincronizar eventos y gestionar interrupciones relacionadas con el tiempo, como el reloj del sistema y los temporizadores de la CPU.
- **Interrupt**: Maneja las interrupciones generadas por los eventos del sistema, como teclas presionadas, cambios en el PPU o eventos de temporización. Se encarga de priorizarlas y derivarlas a las funciones correspondientes.
- **IO (Input/Output)**: Administra las interacciones de entrada y salida, como las pulsaciones de los botones del usuario y la comunicación con dispositivos externos.
- **DMA**: Permite la transferencia de datos entre la memoria de vídeo y la memoria principal sin la intervención del CPU. Facilita la copia eficiente de gráficos y sprites, optimizando el rendimiento al liberar al CPU para otras tareas durante las transferencias de datos.
- **FifoFetcher**: Gestiona la recuperación de datos de píxeles de la memoria de video, utilizando una estructura FIFO para almacenar temporalmente la información de tiles y atributos. Su función principal es asegurar una carga eficiente de píxeles para la representación gráfica en pantalla.
- **Audio**: Genera y manipula sonidos en la Game Boy, gestionando canales de audio como ondas de pulso y ruido. Controla la frecuencia, duración y mezcla de los sonidos.

7.2 CPU

El desarrollo comienza con la implementación de la Unidad Central de Procesamiento. Debido a su papel fundamental en la correcta reproducción del funcionamiento del sistema, esta etapa se centra en implementar todas las instrucciones para asegurar que el emulador pueda procesar cada byte de información con precisión.

7.2.1 Registros

Para definirlos en nuestro programa, utilizaremos el tipo Byte o UByte de Kotlin. En mi caso por desconocimiento del segundo tipo, comencé a programarlo con Byte. La diferencia entre ambos es que Byte utiliza el rango de [-128:128] y UByte el de [0:255] (igual que la GB). Podemos hacer uso de cualquiera de los dos mientras tengamos en mente que, si el primero lo convertimos a Integer, será un valor distinto al original. En cuanto a PC y SP, optaremos por utilizar Integers, ya que son valores de 2 bytes y siempre contienen una dirección de memoria.

Código 7.1: Declaración de Registros

```
1   var A: Byte = 0
2   var F: Byte = 0 // Contains the 4 flags (11110000 -> ZNHC0000)
3   var B: Byte = 0
```

```
4    var C: Byte = 0
5    var D: Byte = 0
6    var E: Byte = 0
7    var H: Byte = 0
8    var L: Byte = 0
9
10   // 16 bits registers
11   var SP: Int = 0xFFFE // Stack Pointer
12   var PC: Int = 0 // Program Counter
```

Para operar con ellos a nivel de byte, deberemos pasarlos a Integer, aplicarles una operación AND con el valor 0xFF (para eliminar el signo en caso de se utilice el tipo Byte originalmente), se ejecutarían las operaciones necesarias y al resultado se le aplicaría el mismo AND, para justo después volver a convertirlo a Byte:

Código 7.2: Ejemplo de Opcode

```
1    fun add_a_c(): Int{
2        val intA = A.toInt() and 0xFF // Conversión de Byte a Integer sin signo
             -> A
3        val intC = C.toInt() and 0xFF // Conversión de Byte a Integer sin signo
             -> C
4        val result = intA + intC // Se suman ambos valores (ADD)
5        A = (result and 0xFF).toByte() // Al resultado se le quita el signo por
             precaución y se convierte a Byte
6
7        updateAddOperationFlags(intA, intC, result) // Se actualizan los Flags
             correspondientes
8
9        return CYCLES_4 // Devolvemos los ciclos que la CPU debe tardar en
             ejecutar la instrucción
10   }
```

Para la actualización de los flags, se implementarán funciones específicas que gestionarán su estado de manera adecuada. Adicionalmente, se declararán constantes que facilitarán la identificación del estado de cada flag en cualquier momento. Estas constantes corresponden a los bits asociados con un valor de 1 en formato hexadecimal (por ejemplo, 0x80 corresponde a 0b10000000).

Código 7.3: Actualización de Flags

```
1
2    // Flags --> Booleans
3    const val FLAG_Z = 0x80 // Zero Flag
4    const val FLAG_N = 0x40 // Subtract Flag
5    const val FLAG_H = 0x20 // Half Carry Flag
6    const val FLAG_C = 0x10 // Carry Flag
7
8    [...]
9
10   fun setFlag(flag: Int) {
11       F = (F.toInt() or flag).toByte()
```

```
12    }
13
14    fun clearFlag(flag: Int) {
15        F = (F.toInt() and flag.inv()).toByte()
16    }
17
18    fun updateFlag(flag: Int, condition: Boolean) {
19        if (condition) {
20            setFlag(flag)
21        } else {
22            clearFlag(flag)
23        }
24    }
25
26    fun flagIsSet(flag: Int): Boolean{
27        return (F.toInt() and flag) != 0
28    }
```

7.2.2 Opcodes

Lo primero es identificar qué operación ejecutar dependiendo del byte que nos llegue. Podemos hacerlo de muchas maneras, en este caso lo vamos a manejar mediante un switch:

Código 7.4: Identificación de Opcode

```
1     fun execute(opcode: Byte): Int {
2         return when (opcode.toInt() and 0xFF) {
3             0x00 -> nop()
4             0x01 -> ld_bc_nn() // LD BC, nn
5             0x02 -> ld_bc_a() // LD [BC], A
6             0x03 -> inc_bc() // INC BC
7             0x04 -> inc_b() // INC B
8             0x05 -> dec_b() // DEC B
9             0x06 -> ld_b_n() // LD B, n
10
11            [...]
12
13            0xFA -> ld_a_nn() // LD A, [NN]
14            0xFB -> ei() // EI
15            0xFE -> cp_n() // CP N
16            0xFF -> rst(0x0038) // RST 38H
17            else -> throw IllegalArgumentException("Instruction not supported: $↩
                  ↪ {opcode.toInt() and 0xFF}")
18        }
19    }
```

Para las **instrucciones extendidas**, se implementará otro switch que, de la misma forma, hará también distinción por Byte, pero al que solamente se llegará en caso de que en este primero encontremos el **Byte 0xCB**.

En caso de que el input que nos llegue por parámetro no represente ninguna instrucción

conocida, el emulador lanzará una excepción y finalizará la ejecución.

Por cada instrucción, como ya hemos visto, deberemos tener en cuenta las características mencionadas previamente. Vamos a mostrar ejemplos de instrucciones ya implementadas por cada categoría:

7.2.2.0.1 Funciones comunes: Algunas funciones comunes que la gran mayoría de instrucciones van a utilizar.

Código 7.5: Operaciones comunes

```
fun fetch(): Byte {
    val byte = Memory.getByteOnAddress(PC)

    if(!cpu_halt_bug){
        PC = (PC + 1) and 0xFFFF
    }else{
        cpu_halt_bug = false
    }

    return byte
}

fun fetch16(): Int {
    val low = fetch().toInt() and 0xFF
    val high = fetch().toInt() and 0xFF
    return return get_16bit_address(high, low)
}

fun get_16bit_address(high: Byte, low: Byte): Int{
    return ((high.toInt() and 0xFF) shl 8) or (low.toInt() and 0xFF)
}

fun set_16bit_address_value(high: Byte, low: Byte, value: Byte){
    val address = get_16bit_address(high, low)
    Memory.writeByteOnAddress(address, value)
}
```

La función *fetch()* tiene como objetivo principal obtener el byte almacenado en la dirección de memoria señalada por el PC, e incrementarlo posteriormente (si no se produce el fallo de la CPU, que será explicado más adelante).

Por su parte, *fetch16()* realiza dos llamadas consecutivas a *fetch()* y combina los dos bytes obtenidos mediante la función *get_16bit_address()*, que utiliza las operaciones SHL y OR.

Finalmente, la función *set_16bit_address_value()*, recibe una dirección como parámetro y delega al módulo de memoria la escritura del valor proporcionado, si es posible hacerlo.

7.2.2.0.2 Carga - LD: Añadimos de ejemplo cuatro funciones de las cuales podemos derivar el resto. En todos ellos se van a devolver los ciclos de reloj correspondientes.

Código 7.6: Operaciones LD

```
fun ld_c_l(): Int{
    C = L
    return CYCLES_4
}

fun ld_c_hl(): Int{
    val hl = get_16bit_address(H, L)
    C = Memory.getByteOnAddress(hl)
    return CYCLES_8
}

fun ld_hl_b(): Int{
    set_16bit_address_value(H, L, B)
    return CYCLES_8
}

fun ld_hl_nn(): Int{
    val low = fetch().toInt() and 0xFF
    val high = fetch().toInt() and 0xFF
    H = high.toByte()
    L = low.toByte()

    return CYCLES_12
}
```

En la primera función (LD C, L) simplemente se carga el contenido del registro L en C.

En la siguiente (LD C, [HL]) lo que se debe hacer es obtener el byte almacenado en la dirección de memoria señalada por HL y cargarlo en C (para ello hacemos uso del módulo de memoria).

La tercera función (LD [HL], B) es justo lo contrario a la segunda. En este caso lo que hacemos es guardar el byte del registro B en la dirección de memoria ya especificada en HL.

Por último, en la cuarta función (LD [HL], NN), no nos viene especificado un registro, si no que debemos obtener los dos siguientes bytes del PC. Hay que recordar que los bytes se guardan en memoria en **Little Endian**, por lo que el primero que se obtiene es el Low. Asignamos al registro H el High y al registro L el Low, y al devolver los ciclos correspondientes quedaría implementada la instrucción.

7.2.2.0.3 Aritméticas y Lógicas: En esta categoría entran todas las instrucciones de ADD, ADC, AND, SUB, SBC, DEC, INC, XOR, OR, CP, CPL, CCF, DAA y SCF. Vamos a ver algunos ejemplos de cada una de ellas:

Código 7.7: Operaciones ADD y ADC

```
fun updateAddOperationFlags(val1: Int, val2: Int, result: Int){
```

```
        updateFlag(FLAG_Z, result == 0)
        clearFlag(FLAG_N)
        updateFlag(FLAG_H, (val1 and 0xF) + (val2 and 0xF) > 0xF)
        updateFlag(FLAG_C, result > 0xFF)
    }

    fun add_a_b(): Int{
        val intA = A.toInt() and 0xFF
        val intB = B.toInt() and 0xFF
        val result = intA + intB
        A = (result and 0xFF).toByte()

        updateAddOperationFlags(intA, intB, result)

        return CYCLES_4
    }

    fun adc_a_b(): Int{
        val carry = if (flagIsSet(FLAG_C)) 1 else 0
        val intA = A.toInt() and 0xFF
        val intB = B.toInt() and 0xFF
        val result = intA + (intB + carry)
        A = (result and 0xFF).toByte()

        updateAddOperationFlags(intA, intB + carry, result)

        return CYCLES_4
    }
```

Para la instrucción **ADD** (en este caso **ADD A, B**), la operación consiste en convertir ambos registros a enteros sin signo y sumar sus valores.

La diferencia entre ADD y ADC reside en que en este último al resultado también se le suma el valor del carry y, por ende, se debe tener en cuenta en la actualización del flag Half-Carry.

En los casos que impliquen el uso de direcciones de memoria (operaciones de 2 bytes), se puede emplear el código utilizado en las instrucciones de carga (LD) 7.2.2.0.2 como referencia.

Código 7.8: Operaciones INC y DEC

```
    fun inc_8bit_register(register: Byte): Byte{
        val toReturn = (register.toInt() + 1).toByte()
        updateFlag(FLAG_Z, toReturn.toInt() == 0x00)
        clearFlag(FLAG_N)
        updateFlag(FLAG_H, ((register.toInt() and 0xF) + 1) and 0x10 != 0x00)
        return toReturn
    }

    fun dec_8bit_register(register: Byte): Byte{
```

```
        val toReturn = (register.toInt() - 1).toByte()
        updateFlag(FLAG_Z, toReturn.toInt() == 0x00)
        setFlag(FLAG_N)
        updateFlag(FLAG_H, (register.toInt() and 0xF == 0x00))
        return toReturn
    }

    fun inc_bc(): Int{
        val oldValue = get_16bit_address(B, C)
        val newValue = (oldValue + 1) and 0xFFFF
        B = (newValue shr 8).toByte()
        C = newValue.toByte()
        return CYCLES_8
    }

    fun dec_b(): Int{
        B = dec_8bit_register(B)
        return CYCLES_4
    }

    fun inc_b(): Int{
        B = inc_8bit_register(B)
        return CYCLES_4
    }
```

Las instrucciones de INC y DEC son muy similares. INC suma 1 al valor y afecta los flags del procesador: el Zero (Z) se activa si el resultado es 0, el Half-carry (H) se activa si hay un acarreo entre los bits 3 y 4, y el Subtract (N) siempre se borra. Por su parte, DEC resta 1 al valor y afecta los mismos flags, pero siempre activa el Subtract (N) ya que es una operación de sustracción. Ambos opcodes no afectan el Carry flag (C) y se utilizan tanto para registros de 8 bits como para posiciones de memoria.

Código 7.9: Operación XOR

```
    fun xor_b(): Int{
        A = (((A.toInt() and 0xFF) xor (register.toInt()) and 0xFF)).toByte()

        updateFlag(FLAG_Z, (A.toInt() and 0xFF) == 0)
        clearFlag(FLAG_N)
        clearFlag(FLAG_H)
        clearFlag(FLAG_C)

        return CYCLES_4
    }
```

La operación XOR se realizan siempre al registro A, utilizando su valor y el del registro indicado por el opcode (en este caso B). Tras la operación, verifica si el resultado es cero para activar el flag Z, y limpia los flags N, H y C, ya que no son relevantes.

Las operaciones OR son idénticas en Kotlin, simplemente deberemos cambiar el operando 'xor' por 'or'.

7.2.2.0.4 Control de flujo: Las instrucciones de control de flujo, como CALL, JP y RETI, permiten modificar la secuencia de ejecución del programa. Estas instrucciones desvían el flujo normal de instrucciones al saltar a direcciones específicas de memoria o retornar desde subrutinas o interrupciones. Tenemos instrucciones como CALL, JP o JR que son utilizadas para saltar a una nueva dirección, con CALL almacenando la dirección de retorno en la pila para permitir volver al punto de origen.

Vamos a analizarlas de una en una:

Código 7.10: Operación CALL

```
fun call_nz_nn(): Int{
    val address = fetch16()

    if (!flagIsSet(FLAG_Z)) {
        SP = (SP - 1) and 0xFFFF
        Memory.writeByteOnAddress(SP, (PC ushr 8).toByte()) // Alto
        SP = (SP - 1) and 0xFFFF
        Memory.writeByteOnAddress(SP, (PC and 0xFF).toByte()) // Bajo

        PC = address
        return CYCLES_24
    }

    return CYCLES_12
}
```

La instrucción **CALL** salta a una subrutina especificada, guardando la dirección de retorno en la pila. El PC se actualiza con la dirección de destino, y tras ejecutar la subrutina, el programa puede volver al punto original usando la instrucción RET, restaurando la dirección desde la pila. Esta último instrucción no se implementa en el propio CALL, si no que debe ser gestionada a posterior por parte del desarrollador, utilizando el valor previo del PC (guardado en el SP antes de su actualización).

Además, en el ejemplo expuesto, se nos indica que el CALL solamente se debe ejecutar si el Flag Z no está activo. En caso contrario, lo único que haría son 2 *fetch()* seguidos y se hace uso de menos ciclos de reloj.

Código 7.11: Operaciones JR y JP

```
fun jr_n(): Int{
    val offset = fetch()
    PC += offset.toInt()
    return CYCLES_12
}

fun jp_nn(): Int{
    val address = fetch16()
    PC = address
    return CYCLES_16
}
```

Las instrucciones **JR** y **JP** son similares a CALL, pero no almacenan el valor actual del PC en el stack.

La instrucción JP salta directamente a una dirección de memoria especificada, actualizando el PC, lo que permite saltos largos a cualquier posición en la memoria.

JR, en cambio, realiza un salto relativo, ajustando el PC en función de un desplazamiento positivo o negativo, permitiendo saltos más cortos dentro de un rango cercano. Dado que los **saltos relativos** pueden cubrir un **máximo de 0xFF bytes** hacia arriba o abajo, JR consume menos ciclos de reloj.

Código 7.12: Operaciones RET y RETI

```
fun executeRetOperation(){
    val low = Memory.getByteOnAddress(SP).toInt() and 0xFF
    SP = (SP + 1) and 0xFFFF
    val high = Memory.getByteOnAddress(SP).toInt() and 0xFF
    SP = (SP + 1) and 0xFFFF

    PC = (high shl 8) or low
}

fun ret(): Int{
    executeRetOperation()
    return CYCLES_16
}

fun reti(): Int{
    executeRetOperation()
    Interrupt.enableInterrupts(true)
    return CYCLES_16
}
```

Las instrucciones **RET** y **RETI** se utilizan para retornar de una subrutina, recuperando la dirección de retorno almacenada en el stack. RET restaura el valor del registro PC desde el stack, permitiendo así continuar la ejecución desde donde se dejó al llamar a la subrutina. En contraste, RETI realiza la misma operación, pero se utiliza específicamente para el retorno de una interrupción, asegurando que se manejen correctamente las interrupciones pendientes antes de restaurar el flujo de ejecución.

Código 7.13: Operación CP

```
fun cp_b(): Int{
    val intA = A.toInt() and 0xFF
    val intRegister = B.toInt() and 0xFF

    val result = (intA - intRegister) and 0xFF

    updateFlag(FLAG_Z, result == 0)
    setFlag(FLAG_N)
```

```
        updateFlag(FLAG_H, (intA and 0xF) < (intRegister and 0xF))
        updateFlag(FLAG_C, intA < intRegister)

        return CYCLES_4
    }
```

La función CP B (Compare B) se encarga de comparar el valor del registro A con el valor del registro B, estableciendo los flags correspondientes según el resultado de la comparación. Primero, convierte los registros A y B a enteros de 8 bits, luego calcula el resultado de la resta entre A y B, enmascarando el resultado para asegurarse de que se mantenga dentro del rango de 8 bits. A continuación, actualiza el flag Z si el resultado es igual a cero, establece el flag N para indicar que se realizó una comparación, y determina el estado del flag H al verificar si el nibble menos significativo de A es menor que el de B. Por último, establece el flag C si el valor de A es menor que el de B.

Código 7.14: Operación CPL

```
    fun cpl(): Int{

        A = (A.toInt() xor 0xFF).toByte()

        setFlag(FLAG_N)
        setFlag(FLAG_H)

        return CYCLES_4
    }
```

La función CPL (Complementary) se encarga de complementar el valor del registro A, invirtiendo todos sus bits mediante una operación XOR con 0xFF. Esto transforma todos los bits de A en sus opuestos, cambiando ceros por unos y viceversa. Después de realizar la operación, la función establece el flag N para indicar que se ha realizado una operación que afecta al signo del número, y también establece el flag H para indicar que puede haber un acarreo en la operación.

Código 7.15: Operación CCF

```
    fun ccf(): Int{

        val newCarry = ((F.toInt() and 0xFF) and FLAG_C) == 0
        updateFlag(FLAG_C, newCarry)
        setFlag(FLAG_N)
        setFlag(FLAG_H)

        return CYCLES_4
    }
```

La función CCF (Complement Carry Flag) se encarga de complementar el valor del flag C. Primero, verifica si el flag de acarreo está actualmente activado; si no lo está, lo activa, y si lo está, lo desactiva, utilizando el operador lógico AND para determinar su estado anterior. Además, establece los flags H y N como activos.

Código 7.16: Operación DAA

```
fun daa(): Int{

    var result = A.toInt() and 0xFF

    if (!flagIsSet(FLAG_N)) { // Addition

        if ((result and 0x0F) > 9 || flagIsSet(FLAG_H)) // Lower nibble
            result += 0x06

        if ((result and 0xF0) > 0x90 || flagIsSet(FLAG_C)) // Higher nibble
            result += 0x60

    }else{ // Substraction
        if (flagIsSet(FLAG_H)) // Lower nibble
            result -= 0x06

        if (flagIsSet(FLAG_C)) // Higher nibble
            result -= 0x60
    }

    updateFlag(FLAG_Z, (result and 0xFF) == 0x00)
    clearFlag(FLAG_H)
    updateFlag(FLAG_C, result > 0xFF)

    result = result and 0xFF
    A = result.toByte()

    return CYCLES_4
}
```

La función DAA (Decimal Adjust for Addition) es una implementación que ajusta el valor del registro A para operaciones aritméticas en formato decimal después de una suma o resta. El método primero convierte el valor de A a un entero de 8 bits, luego verifica si la operación anterior fue una suma o una resta basándose en el estado del flag N.

Código 7.17: Operación SCF

```
fun scf(): Int{

    clearFlag(FLAG_N)
    clearFlag(FLAG_H)
    setFlag(FLAG_C)

    return CYCLES_4
}
```

La función SCF (Set Carry Flag) establece el flag C a 1 y borra los flags N y H, lo que indica que el próximo cálculo tendrá en cuenta que se ha producido un acarreo.

7.2.2.0.5 Rotación y desplazamiento: En esta categoría tenemos las instrucciones de RL, RR, RLC, RRC, SLA, SRA, SWAP y SRL. Vamos a ver algunos ejemplos de cada una de ellas:

Código 7.18: Operaciones RL y RR

```
fun rl_b(): Int{
    val bByte = B.toInt() and 0xFF
    val oldCarry = if (flagIsSet(FLAG_C)) 1 else 0
    val newCarry = (bByte ushr 7) and 0x1

    B = ((bByte shl 1) or oldCarry).toByte()

    updateFlag(FLAG_Z, B == 0.toByte())
    clearFlag(FLAG_N)
    clearFlag(FLAG_H)
    updateFlag(FLAG_C, newCarry == 1)

    return CYCLES_8
}

fun rr_b(): Int{
    val bByte = B.toInt() and 0xFF
    val oldCarry = if (flagIsSet(FLAG_C)) 1 else 0
    val newCarry = (bByte ushr 7) and 0x1

    B = ((bByte shr 1) or (oldCarry shl 7)).toByte()

    updateFlag(FLAG_Z, B == 0.toByte())
    clearFlag(FLAG_N)
    clearFlag(FLAG_H)
    updateFlag(FLAG_C, newCarry == 1)

    return CYCLES_8
}
```

Las funciones RL B y RR B realizan rotaciones de bits en el registro B, pero difieren en la dirección y el manejo del carry. La función RL rota los bits hacia la izquierda; el MSB se desplaza a la izquierda y se introduce en el LSB, utilizando el valor del carry anterior para completar la rotación. En cambio, RR rota los bits hacia la derecha; el LSB se mueve al carry y el carry anterior se coloca en el MSB. Ambas funciones actualizan los indicadores de estado, como el flag Z si el resultado es cero, y el flag C según el bit que se desplaza.

Código 7.19: Operaciones RLC y RRC

```
fun rlc_b(): Int{
    val bByte = B.toInt() and 0xFF
    val carry = (bByte ushr 7) and 0x1
    B = ((bByte shl 1) or carry).toByte()

    updateFlag(FLAG_Z, B == 0.toByte())
    clearFlag(FLAG_N)
```

```
        clearFlag(FLAG_H)
        updateFlag(FLAG_C, carry == 1)

        return CYCLES_8
    }

    fun rrc_b(): Int{
        val bByte = B.toInt() and 0xFF
        val carry = bByte and 0x1
        B = ((bByte shr 1) or (carry shl 7)).toByte()

        updateFlag(FLAG_Z, B == 0.toByte())
        clearFlag(FLAG_N)
        clearFlag(FLAG_H)
        updateFlag(FLAG_C, carry == 1)

        return CYCLES_8
    }
```

La función RLC B realiza una rotación a la izquierda del registro B, desplazando el MSB hacia el LSB y estableciendo el nuevo valor del MSB en el carry. Actualiza todos los flags en función del resultado. En contraste, la función RRC B efectúa una rotación a la derecha, desplazando el LSB hacia el MSB y estableciendo su nuevo valor en el carry.

Código 7.20: Operaciones SLA, SRA y SRL

```
    fun sla_b(): Int{
        val bByte = B.toInt() and 0xFF
        val newCarry = (bByte ushr 7) and 0x1
        B = ((bByte shl 1) and 0xFE).toByte()

        updateFlag(FLAG_Z, B == 0.toByte())
        clearFlag(FLAG_N)
        clearFlag(FLAG_H)
        updateFlag(FLAG_C, newCarry == 1)

        return CYCLES_8
    }

    fun sra_b(): Int{
        val bByte = B.toInt() and 0xFF
        val oldBit7 = bByte and 0x80
        val newCarry = bByte and 0x1

        B = ((bByte shr 1) or oldBit7).toByte()

        updateFlag(FLAG_Z, B == 0.toByte())
        clearFlag(FLAG_N)
        clearFlag(FLAG_H)
        updateFlag(FLAG_C, newCarry == 1)

        return CYCLES_8
```

```
27      }
28
29      fun srl_b(): Int{
30          val bByte = B.toInt() and 0xFF
31          val newCarry = bByte and 0x1
32          B = ((bByte shr 1) and 0x7F).toByte()
33
34          updateFlag(FLAG_Z, B == 0.toByte())
35          clearFlag(FLAG_N)
36          clearFlag(FLAG_H)
37          updateFlag(FLAG_C, newCarry == 1)
38
39          return CYCLES_8
40      }
```

La función SLA B realiza un desplazamiento lógico a la izquierda del registro B, moviendo todos los bits una posición a la izquierda y estableciendo el LSB en 0, mientras que el nuevo carry se toma del antiguo MSB. En contraste, SRA B realiza un desplazamiento aritmético a la derecha, manteniendo el bit más significativo y moviendo el resto de los bits hacia la derecha, con el nuevo carry tomado del antiguo LSB. Por otro lado, SRL B también realiza un desplazamiento lógico a la derecha, pero establece el MSB en 0 y mueve los bits a la derecha, con el nuevo carry tomado del antiguo LSB.

Código 7.21: Operación SWAP

```
1       fun swap_b(): Int{
2           val bByte = B.toInt() and 0xFF
3           val low = (bByte and 0x0F) shl 4
4           val high = (bByte and 0xF0) shr 4
5           B = (low or high).toByte()
6
7           updateFlag(FLAG_Z, B == 0.toByte())
8           clearFlag(FLAG_N)
9           clearFlag(FLAG_H)
10          clearFlag(FLAG_C)
11
12          return CYCLES_8
13      }
```

La función SWAP B intercambia los nibbles (4 bits) del registro B, moviendo los 4 bits menos significativos a la posición de los 4 bits más significativos y viceversa. Actualiza el flag Z si el nuevo valor es cero, y limpia los flags N, H y C.

7.2.2.0.6 Manipulación de bits:

Encontramos las instrucciones BIT, RES y SET.

Código 7.22: Operación BIT

```
1       fun updateBitOperationFlags(result: Boolean){
2           updateFlag(FLAG_Z, result)
3           clearFlag(FLAG_N)
4           setFlag(FLAG_H)
5       }
```

```
6
7   fun bit_operation(register: Int, bitNumber: Int): Int{
8
9       require(bitNumber in 0..7) { "Bit must be between 0 and 7" }
10      require(register in 1..8) { "Register must be between 1 and 8" }
11
12      var bitZero = false
13      var cyclesToReturn = CYCLES_8
14      val bit = 0x1 shl bitNumber
15
16      when (register) {
17          1 -> bitZero = ((B.toInt() and 0xFF) and bit) == 0
18          2 -> bitZero = ((C.toInt() and 0xFF) and bit) == 0
19          3 -> bitZero = ((D.toInt() and 0xFF) and bit) == 0
20          4 -> bitZero = ((E.toInt() and 0xFF) and bit) == 0
21          5 -> bitZero = ((H.toInt() and 0xFF) and bit) == 0
22          6 -> bitZero = ((L.toInt() and 0xFF) and bit) == 0
23          7 -> {
24              val address = get_16bit_address(H, L)
25              bitZero = ((Memory.getByteOnAddress(address).toInt() and 0xFF) ↩
                    ↪ and bit) == 0
26              cyclesToReturn = CYCLES_16
27          }
28          8 -> bitZero = ((A.toInt() and 0xFF) and bit) == 0
29      }
30
31      updateBitOperationFlags(bitZero)
32      return cyclesToReturn
33  }
```

La operación BIT verifica el estado de un bit específico (de 0 a 7) en un registro determinado (de 1 a 8). Utiliza condiciones para identificar qué registro se está evaluando y calcula si el bit indicado está apagado (0) o encendido (1). Si el registro es 7, que representa una dirección de memoria, obtiene el byte correspondiente desde esa dirección, y el tiempo de ciclo se ajusta a 16. Posteriormente, actualiza el flag Z, limpia el flag N y establece el flag H. Al final, devuelve el tiempo de ciclo correspondiente, que es 8 para los registros de 1 a 6 y el 8, y 16 para el registro 7.

Código 7.23: Operación RES

```
1   fun res_operation(register: Int, bitNumber: Int): Int{
2
3       require(bitNumber in 0..7) { "Bit must be between 0 and 7" }
4       require(register in 1..8) { "Register must be between 1 and 8" }
5
6       var cyclesToReturn = CYCLES_8
7       val bit = (0x1 shl bitNumber).inv()
8
9       when (register) {
10          1 -> B = ((B.toInt() and 0xFF) and bit).toByte()
11          2 -> C = ((C.toInt() and 0xFF) and bit).toByte()
```

```
            3 -> D = ((D.toInt() and 0xFF) and bit).toByte()
            4 -> E = ((E.toInt() and 0xFF) and bit).toByte()
            5 -> H = ((H.toInt() and 0xFF) and bit).toByte()
            6 -> L = ((L.toInt() and 0xFF) and bit).toByte()
            7 -> {
                val address = get_16bit_address(H, L)
                val result = ((Memory.getByteOnAddress(address).toInt() and 0xFF)
                    ↪ and bit).toByte()
                Memory.writeByteOnAddress(address, result)
                cyclesToReturn = CYCLES_16
            }
            8 -> A = ((A.toInt() and 0xFF) and bit).toByte()
        }

        return cyclesToReturn
    }
```

La operación RES se encarga de reiniciar (poner a 0) un bit específico (de 0 a 7) en un registro determinado (de 1 a 8). Utiliza condiciones para determinar cuál registro se está manipulando y genera una máscara de bit que apaga el bit correspondiente. Si el registro es 7, la función obtiene la dirección de memoria desde los registros H y L, lee el byte almacenado en esa dirección, reinicia el bit correspondiente y escribe el nuevo valor de vuelta en memoria. El tiempo de ciclo se gestiona de la misma manera que en la operación BIT.

Código 7.24: Operación SET

```
    fun set_operation(register: Int, bitNumber: Int): Int{

        require(bitNumber in 0..7) { "Bit must be between 0 and 7" }
        require(register in 1..8) { "Register must be between 1 and 8" }

        var cyclesToReturn = CYCLES_8
        val bit = 0x1 shl bitNumber

        when (register) {
            1 -> B = ((B.toInt() and 0xFF) or bit).toByte()
            2 -> C = ((C.toInt() and 0xFF) or bit).toByte()
            3 -> D = ((D.toInt() and 0xFF) or bit).toByte()
            4 -> E = ((E.toInt() and 0xFF) or bit).toByte()
            5 -> H = ((H.toInt() and 0xFF) or bit).toByte()
            6 -> L = ((L.toInt() and 0xFF) or bit).toByte()
            7 -> {
                val address = get_16bit_address(H, L)
                val result = ((Memory.getByteOnAddress(address).toInt() and 0xFF)
                    ↪ or bit).toByte()
                Memory.writeByteOnAddress(address, result)
                cyclesToReturn = CYCLES_16
            }
            8 -> A = ((A.toInt() and 0xFF) or bit).toByte()
        }

```

```
25          return cyclesToReturn
26      }
```

La operación SET se encarga de establecer (poner a 1) un bit específico (de 0 a 7) en un registro determinado (de 1 a 8). Utiliza condiciones para identificar qué registro se está manipulando y genera una máscara de bit que activa el bit correspondiente. Si el registro es 7, la función obtiene la dirección de memoria a partir de los registros H y L, lee el byte almacenado en esa dirección, activa el bit correspondiente y escribe el nuevo valor de vuelta en la memoria. El tiempo de ciclo se gestiona de la misma manera que en la operación BIT.

7.2.2.0.7 Especiales de sistema:

Encontramos las instrucciones NOP, DI y EI:

Código 7.25: Operación NOP

```
1   fun nop(): Int{
2       return CYCLES_4
3   }
```

La operación NOP (No Operation) realiza una operación que no tiene efecto en el estado del procesador; es decir, no cambia los registros, la memoria o los flags. Su principal función es ocupar espacio en el ciclo de ejecución, permitiendo la sincronización en la ejecución de instrucciones o como un marcador para pausas en el código. A la hora de implementarlo, simplemente devolvemos los ciclos correspondientes.

Código 7.26: Operaciones DI, EI

```
1   private var pendingEI = false
2
3   [...]
4
5   fun ei(): Int{
6       pendingEI = true
7       return CYCLES_4
8   }
9
10  fun di(): Int{
11      Interrupt.enableInterrupts(false)
12      return CYCLES_4
13  }
```

El opcode EI habilita las interrupciones en el procesador (se detallará más adelante). Es importante destacar que las interrupciones no se activan de manera inmediata; en su lugar, deben esperar hasta el siguiente ciclo de la CPU para entrar en efecto. Por esta razón, se utiliza una variable para almacenar el estado de las interrupciones.

Por otro lado, DI deshabilita las interrupciones en el procesador, bloqueando la capacidad del sistema para responder a señales externas hasta que se vuelvan a habilitar mediante un EI. En este caso, los cambios si que tienen efecto inmediato.

7.2.2.0.8 Con pila:

Encontramos las instrucciones PUSH y POP:

Código 7.27: Operaciones PUSH, POP

```
fun push_bc(): Int{
    SP = (SP - 1) and 0xFFFF
    Memory.writeByteOnAddress(SP, B) // high
    SP = (SP - 1) and 0xFFFF
    Memory.writeByteOnAddress(SP, C) // low

    return CYCLES_16
}

fun pop_bc(): Int{

    C = Memory.getByteOnAddress(SP)
    SP = (SP + 1) and 0xFFFF
    B = Memory.getByteOnAddress(SP)
    SP = (SP + 1) and 0xFFFF

    return CYCLES_12
}
```

La función PUSH BC almacena los valores de los registros B y C en la pila. Primero, decrece el puntero de la pila (SP) en 1 y escribe el contenido de B en la dirección de memoria apuntada, luego vuelve a decrecer la pila y escribe el contenido de C. Esta operación asegura que el registro C se almacene en la parte baja de la pila y B en la parte alta.

Por otro lado, la función POP BC recupera los valores de los registros B y C desde la pila. Comienza leyendo el byte almacenado en la dirección de memoria apuntada por SP y lo asigna al registro C, luego incrementa la pila para apuntar al siguiente byte. A continuación, lee el byte en la nueva dirección y lo asigna al registro B, y nuevamente incrementa SP.

Código 7.28: Operaciones I/O

```
fun ldh_n_a(): Int{

    val byte = fetch().toInt() and 0xFF
    val address = (0xFF00 + byte) and 0xFFFF
    Memory.writeByteOnAddress(address, A)

    return CYCLES_12
}

fun ld_cn_a(): Int{
    val address = (0xFF00 + (C.toInt() and 0xFF)) and 0xFFFF
    Memory.writeByteOnAddress(address, A)
    return CYCLES_8
}
```

La función LDH N, A carga el valor del registro A en una dirección de memoria específica determinada por un byte que se obtiene mediante la función fetch(). Este byte se suma a la dirección base 0xFF00 (dirección en la que empiezan los registros de I/O) para formar la dirección final donde se almacenará el valor de A.

Por otro lado, la función LD CN, A almacena el valor del registro A en una dirección de memoria también basada en el registro C. Aquí, se suma el valor de C a la dirección base 0xFF00 para calcular la dirección final, donde se escribirá el valor de A.

Ambas funciones son importantes para la manipulación de datos en la memoria del sistema.

7.3 Memory

El módulo de memoria actúa como un bus de datos que redirecciona las operaciones de lectura y escritura hacia otros módulos del emulador, como la CPU, la ROM y otros componentes. Además de su función de interconexión, este módulo contiene toda la memoria virtual principal de nuestro emulador, que abarca los 64 KB de espacio de memoria direccionable. Esto incluye tanto la memoria de trabajo, como la RAM, como la memoria de mapeo de la ROM, que se utiliza para cargar los juegos.

Comenzaremos la implementación declarando algunas constantes y la variable correspondiente que reservará esos 64KB de memoria:

Código 7.29: Declaraciones iniciales de Memoria

```
1   const val ROM_START = 0x0000
2   const val ROM_SW_START = 0x4000
3   const val ROM_END = 0x7FFF
4   const val BOOT_END = 0x00FF
5   const val VRAM_START = 0x8000
6   const val VRAM_END = 0x9FFF
7   const val EXTERNAL_RAM_START = 0xA000
8   const val WRAM_START = 0xC000
9   const val FIXED_WRAM_END = 0xCFFF
10  const val SWITCHABLE_WRAM_START = 0xD000
11  const val ECHO_RAM_START = 0xE000
12  const val OAM_START = 0xFE00
13  const val RESERVED_MEM_START = 0xFEA0
14  const val IO_START = 0xFF00
15  const val HRAM_START = 0xFF80
16  const val HRAM_END = 0xFFFE
17
18  const val MEMORY_SIZE = 0x10000 // 64 KB
19  const val BOOT_SIZE = 0xFF
20
21  [...]
22
23  private val memory = ByteArray(MEMORY_SIZE)
```

7.3.1 Lectura / Escritura

Al realizar operaciones de lectura y escritura, no podemos simplemente asignar un valor directamente a la dirección proporcionada como parámetro. Existen áreas de memoria donde los desarrolladores tienen restricciones para escribir y otras a las que no se puede acceder en determinados momentos de la ejecución. Por esta razón, delegaremos las funciones de acceso a los módulos apropiados según la región de memoria correspondiente a la dirección que se pase como parámetro.

Por ejemplo, si la dirección solicitada es menor de la dirección en la que empieza la VRAM (0x8000), querrá decir que se está intentando escribir en ROM (0x0000 - 0x7FFF).

Código 7.30: Métodos de lectura y escritura en memoria

```
fun writeByteOnAddress(address: Int, value: Byte){
    if(address < VRAM_START) { // ROM DATA
        ROM.writeToROM(address, value)
    }else if(address < EXTERNAL_RAM_START) { // VRAM DATA
        PPU.writeToVRAM(address, value)
    }else if(address < WRAM_START) { // EXTERNAL / CARTRIDGE RAM DATA
        ROM.writeToROM(address, value)
    }else if(address < ECHO_RAM_START){ // WRAM
        RAM.writeToWRAM(address, value)
    }else if(address < OAM_START) { // ECHO RAM -- CANT BE USED !
        return
    }else if(address < RESERVED_MEM_START) { // OAM DATA
        if(!DMA.transferring())
            PPU.writeToOAM(address, -1, value)
    }else if(address < IO_START) { // RESERVED MEMORY - CANT BE USED !
        return
    }else if(address < HRAM_START) { // IO DATA
        IO.writeToIO(address, value)
    }else if(address < IE){ // HRAM DATA
        RAM.writeToHRAM(address, value)
    }else if(address == IE){ // IE FLAG DATA
        write(address, value)
    }
}

fun getByteOnAddress(address: Int): Byte{
    if(address < VRAM_START) { // ROM DATA
        return ROM.readFromROM(address)
    }else if(address < EXTERNAL_RAM_START) { // VRAM DATA
        return PPU.readFromVRAM(address)
    }else if(address < WRAM_START) { // EXTERNAL / CARTRIDGE RAM DATA
        return ROM.readFromROM(address)
    }else if(address < ECHO_RAM_START){ // WRAM
        return RAM.readFromWRAM(address)
    }else if(address < OAM_START) { // ECHO RAM -- CANT BE USED !
        return 0
    }else if(address < RESERVED_MEM_START) { // OAM DATA
```

```
38            if(DMA.transferring()) return 0xFF.toByte()
39            return PPU.readFromOAM(address)
40        }else if(address < IO_START) { // RESERVED MEMORY - CANT BE USED !
41            return 0
42        }else if(address < HRAM_START) { // IO DATA
43            return IO.readFromIO(address)
44        }else if(address < IE){ // HRAM DATA
45            return RAM.readFromHRAM(address)
46        }else if(address == IE){ // IE FLAG DATA
47            return read(IE)
48        }
49
50        throw IllegalArgumentException("Not valid $address")
51    }
52
53    fun read(address: Int): Byte{
54        return memory[address]
55    }
56
57    fun write(address: Int, value: Byte){
58        memory[address] = value
59    }
```

Las últimas dos funciones son las que leen o escriben directamente de nuestra memoria virtual, y serán utilizadas en última instancia por los módulos delegados una vez hayan verificado que se pueden ejecutar.

7.3.2 Secuencia de Arranque

La secuencia de arranque o boot lo vamos a guardar en este módulo, ya que van a ser datos fijos que deberemos copiar al inicio del programa en nuestra ROM. Existen varias versiones del boot ya desensambladas por internet, siguiendo paso a paso las instrucciones originales. Lo que nosotros vamos a hacer es guardarnos todos los bytes para ejecutarlos más en adelante:

Código 7.31: Secuencia de arranque y logo de Nintendo

```
1   private val nintendoLogo: ByteArray = byteArrayOf(
2       0xCE.toByte(), 0xED.toByte(), 0x66.toByte(), 0x66.toByte(), 0xCC.toByte↩
        ↪ (), 0x0D.toByte(), 0x00.toByte(), 0x0B.toByte(),
3       0x03.toByte(), 0x73.toByte(), 0x00.toByte(), 0x83.toByte(), 0x00.toByte↩
        ↪ (), 0x0C.toByte(), 0x00.toByte(), 0x0D.toByte(),
4       0x00.toByte(), 0x08.toByte(), 0x11.toByte(), 0x1F.toByte(), 0x88.toByte↩
        ↪ (), 0x89.toByte(), 0x00.toByte(), 0x0E.toByte(),
5       0xDC.toByte(), 0xCC.toByte(), 0x6E.toByte(), 0xE6.toByte(), 0xDD.toByte↩
        ↪ (), 0xDD.toByte(), 0xD9.toByte(), 0x99.toByte(),
6       0xBB.toByte(), 0xBB.toByte(), 0x67.toByte(), 0x63.toByte(), 0x6E.toByte↩
        ↪ (), 0x0E.toByte(), 0xEC.toByte(), 0xCC.toByte(),
7       0xDD.toByte(), 0xDC.toByte(), 0x99.toByte(), 0x9F.toByte(), 0xBB.toByte↩
        ↪ (), 0xB9.toByte(), 0x33.toByte(), 0x3E.toByte()
8   )
9
```

```
10    private val bootstrapRom = byteArrayOf(
11        0x31.toByte(), 0xFE.toByte(), 0xFF.toByte(), // LD SP, 0xFFFE
12        0xAF.toByte(), // XOR A
13        0x21.toByte(), 0xFF.toByte(), 0x9F.toByte(), // LD HL, 0x9FFF
14        0x32.toByte(), // LD (HL-), A
15        0xCB.toByte(), 0x7C.toByte(), // BIT 7, H
16        0x20.toByte(), 0xFB.toByte(), // JR NZ, PC + 0xFB
17        0x21.toByte(), 0x26.toByte(), 0xFF.toByte(), // LD HL, 0xFF26
18        0x0E.toByte(), 0x11.toByte(), // LD C, 0x11
19        0x3E.toByte(), 0x80.toByte(), // LD A, 0x80
20        0x32.toByte(), // LD [HL-], A
21        0xE2.toByte(), // LD (0xFF00+C), A
22        0x0C.toByte(), // INC C
23        0x3E.toByte(), 0xF3.toByte(), // LD A, 0xF3
24        0xE2.toByte(), // LD (0xFF00+C), A
25        0x32.toByte(), // LD (HL-), A
26        0x3E.toByte(), 0x77.toByte(), // LD A, 0x77
27        0x77.toByte(), // LD [HL], A
28        0x3E.toByte(), 0xFC.toByte(), // LD A, 0xFC
29        0xE0.toByte(), 0x47.toByte(), // LDH [0xFF00 + 0x47], A
30        0x11.toByte(), 0x04.toByte(), 0x01.toByte(), // LD DE, 0x0104
31        0x21.toByte(), 0x10.toByte(), 0x80.toByte(), // LD HL, 0x8010
32        0x1A.toByte(), // LD A, [DE]
33        0xCD.toByte(), 0x95.toByte(), 0x00.toByte(), // CALL 0x0095
34        0xCD.toByte(), 0x96.toByte(), 0x00.toByte(), // CALL 0x0096
35        0x13.toByte(), // INC DE
36        0x7B.toByte(), // LD A, E
37        0xFE.toByte(), 0x34.toByte(), // CP 0x34
38        0x20.toByte(), 0xF3.toByte(), // JR NZ, PC + 0xF3
39        0x11.toByte(), 0xD8.toByte(), 0x00.toByte(), // LD DE, 0x00D8
40        0x06.toByte(), 0x08.toByte(), // LD B, 0x08
41        0x1A.toByte(), // LD A, [DE]
42        0x13.toByte(), // INC DE
43        0x22.toByte(), // LD [HL+], A
44        0x23.toByte(), // INC HL
45        0x05.toByte(), // DEC B
46        0x20.toByte(), 0xF9.toByte(), // JR NZ, PC + 0xF9
47        0x3E.toByte(), 0x19.toByte(), // LD A, 0x19
48        0xEA.toByte(), 0x10.toByte(), 0x99.toByte(), // LD [0x9910], A
49        0x21.toByte(), 0x2F.toByte(), 0x99.toByte(), // LD HL, 0x992F
50        0x0E.toByte(), 0x0C.toByte(), // LD C, 0x0C
51        0x3D.toByte(), // DEC A
52        0x28.toByte(), 0x08.toByte(), // JR Z, PC + 0x08
53        0x32.toByte(), // LD (HL-), A
54        0x0D.toByte(), // DEC C
55        0x20.toByte(), 0xF9.toByte(), // JR NZ, PC + 0xF9
56        0x2E.toByte(), 0x0F.toByte(), // LD L, 0x0F
57        0x18.toByte(), 0xF3.toByte(), // JR PC + 0xF3
58        0x67.toByte(), // LD H, A
59        0x3E.toByte(), 0x64.toByte(), // LD A, 0x64
60        0x57.toByte(), // LD D, A
```

```
61        0xE0.toByte(), 0x42.toByte(), // LDH [0xFF00 + 0x42], A
62        0x3E.toByte(), 0x91.toByte(), // LD A, 0x91
63        0xE0.toByte(), 0x40.toByte(), // LDH [0xFF00 + 0x40], A
64        0x04.toByte(), // INC B
65        0x1E.toByte(), 0x02.toByte(), // LD E, 0x02
66        0x0E.toByte(), 0x0C.toByte(), // LD C, 0x0C
67        0xF0.toByte(), 0x44.toByte(), // LDH A, [0xFF00 + 0x44]
68        0xFE.toByte(), 0x90.toByte(), // CP 0x90
69        0x20.toByte(), 0xFA.toByte(), // JR NZ, PC + 0xFA
70        0x0D.toByte(), // DEC C
71        0x20.toByte(), 0xF7.toByte(), // JR NZ, PC + 0xF7
72        0x1D.toByte(), // DEC E
73        0x20.toByte(), 0xF2.toByte(), // JR NZ, PC + 0xF2
74        0x0E.toByte(), 0x13.toByte(), // LD C, 0x13
75        0x24.toByte(), // INC H
76        0x7C.toByte(), // LD A, H
77        0x1E.toByte(), 0x83.toByte(), // LD E, 0x83
78        0xFE.toByte(), 0x62.toByte(), // CP 0x62
79        0x28.toByte(), 0x06.toByte(), // JR Z, PC + 0x06
80        0x1E.toByte(), 0xC1.toByte(), // LD E, 0xC1
81        0xFE.toByte(), 0x64.toByte(), // CP 0x64
82        0x20.toByte(), 0x06.toByte(), // JR NZ, PC + 0x06
83        0x7B.toByte(), // LD A, E
84        0xE2.toByte(), // LD [0xFF00+C], A
85        0x0C.toByte(), // INC C
86        0x3E.toByte(), 0x87.toByte(), // LD A, 0x87
87        0xE2.toByte(), // LD [0xFF00+C], A
88        0xF0.toByte(), 0x42.toByte(), // LDH A, [0xFF00 + 0x42]
89        0x90.toByte(), // SUB B
90        0xE0.toByte(), 0x42.toByte(), // LDH [0xFF00 + 0x42], A
91        0x15.toByte(), // DEC D
92        0x20.toByte(), 0xD2.toByte(), // JR NZ, PC + 0xD2
93        0x05.toByte(), // DEC B
94        0x20.toByte(), 0x4F.toByte(), // JR NZ, PC + 0x4F
95        0x16.toByte(), 0x20.toByte(), // LD D, 0x20
96        0x18.toByte(), 0xCB.toByte(), // JR PC + 0xCB
97        0x4F.toByte(), // LD C, A
98        0x06.toByte(), 0x04.toByte(), // LD B, 0x04
99        0xC5.toByte(), // PUSH BC
100       0xCB.toByte(), 0x11.toByte(), // RL C
101       0x17.toByte(), // RLA
102       0xC1.toByte(), // POP BC
103       0xCB.toByte(), 0x11.toByte(), // RL C
104       0x17.toByte(), // RLA
105       0x05.toByte(), // DEC B
106       0x20.toByte(), 0xF5.toByte(), // JR NZ, PC + 0xF5
107       0x22.toByte(), // LD [HL+], A
108       0x23.toByte(), // INC HL
109       0x22.toByte(), // LD [HL+], A
110       0x23.toByte(), // INC HL
111       0xC9.toByte(), // RET
```

```
112         *nintendoLogo,
113         0x3C.toByte(), 0x42.toByte(), 0xB9.toByte(), 0xA5.toByte(), 0xB9.toByte↩
                ↪ (), 0xA5.toByte(), 0x42.toByte(), 0x3C.toByte(),
114         0x21.toByte(), 0x04.toByte(), 0x01.toByte(),
115         0x11.toByte(), 0xA8.toByte(), 0x00.toByte(),
116         0x1A.toByte(), // LD A, [DE]
117         0x13.toByte(), // INC DE
118         0xBE.toByte(), // CP [HL]
119         0x20.toByte(), 0xFE.toByte(), // JR NZ, PC + 0xFE
120         0x23.toByte(), // INC HL
121         0x7D.toByte(), // LD A, L
122         0xFE.toByte(), 0x34.toByte(), // CP 0x34
123         0x20.toByte(), 0xF5.toByte(), // JR NZ, PC + 0xF5
124         0x06.toByte(), 0x19.toByte(), // LD B, 0x19
125         0x78.toByte(), // LD A, B
126         0x86.toByte(), // ADD A, [HL]
127         0x23.toByte(), // INC HL
128         0x05.toByte(), // DEC B
129         0x20.toByte(), 0xFB.toByte(), // JR NZ, PC + 0xFB
130         0x86.toByte(), // ADD A, [HL]
131         0x20.toByte(), 0xFE.toByte(), // JR NZ, 0xFE
132         0x3E.toByte(), 0x01.toByte(), // LD A, 0x01
133         0xE0.toByte(), 0x50.toByte()) // LDH [0xFF00 + 0x50], A
```

Todas esas instrucciones se deberán insertar al inicio del programa a partir de la dirección 0x0000, terminando en 0x00FF (255 Bytes en total). Ello lo podemos hacer de forma muy sencilla con un bucle:

Código 7.32: Copiado del Boot en memoria

```
1   fun insertBootstrapToMemory(){
2       for (i in bootstrapRom.indices) {
3           memory[ROM_START + i] = bootstrapRom[i]
4       }
5   }
```

Se deben implementar otros módulos antes de que la secuencia de Boot pueda funcionar, como las interrupciones y los modos de PPU (espera ciclos de VBlank).

7.4 ROM

Las principales funciones de nuestro módulo ROM serán:

- Abrir y leer el contenido de un archivo GB o GBC.
- Obtener datos básicos como el título de juego, el tipo de cartucho, el tipo de consola, etc...
- Dependiendo del tipo de cartucho, gestionar de forma correcta la escritura a ROM o External RAM.

La escritura en la external RAM se implementa en el módulo de ROM porque muchas de las ROMs de los juegos de Game Boy utilizan cartuchos que incluyen su propia memoria RAM externa. Esta memoria se utiliza, entre otras cosas, para guardar el progreso del juego (mediante la funcionalidad de ”batería” en los cartuchos).

En este contexto, el módulo de ROM se encarga no solo de la lectura de los datos de la ROM, sino también de la gestión de la RAM externa. Esto se debe a que el cartucho puede incluir tanto la ROM del juego como una porción de RAM adicional para almacenar información temporal. Esta información temporal la deberemos guardar en un fichero temporal con la nomenclatura *Titulo_Del_Juego.battery*.

7.4.1 Lectura de ROM

Para poder abrir un fichero en Android, lo primero que se debe preparar es un Activity para que el usuario sea capaz de seleccionar un fichero guardado en la memoria interna de su dispositivo.

De momento crearemos en el MainActivity un botón que al pulsarlo y seleccionar un fichero, inmediatamente lo pasará como parámetro en el intent a otro Activity llamado EmuActivity:

Código 7.33: Abrir archivos binarios en un Activity

```
1  private lateinit var selectRomButton: Button
2  private lateinit var binding: ActivityMainBinding
3
4  private val openFileLauncher = registerForActivityResult(↩
       ↪ ActivityResultContracts.OpenDocument()) { uri ->
5      uri?.let {
6
7          val intent = Intent(this, EmuActivity::class.java).apply {
8              putExtra(ROM_URI_EXTRA, it.toString())
9          }
10         startActivity(intent)
11     }
12 }
13
14 override fun onCreate(savedInstanceState: Bundle?) {
15     super.onCreate(savedInstanceState)
16
17     binding = ActivityMainBinding.inflate(layoutInflater)
18     setContentView(binding.root)
19
20     selectRomButton = binding.selectRomButton
21
22     selectRomButton.setOnClickListener {
23         openFilePicker()
24     }
25 }
26
27 private fun openFilePicker() {
```

```
28          openFileLauncher.launch(arrayOf("application/octet-stream"))
29      }
```

Con *registerForActivityResult(ActivityResultContracts.OpenDocument())* se puede registrar el lanzador para abrir documentos. Si el URI que se devuelve no es nulo, se procederá a crear el Intent, añadiendo el URI como un EXTRA.

El método *onFilePicker()* se ejecuta al pulsar el botón añadido al layout e inicia el proceso de selección de documentos mediante *openFileLauncher.launch(arrayOf("application/octet-stream"))*. El tipo *application/octet-stream* indica que se deben aceptar archivos binarios genéricos.

Para obtener el parámetro y sus bytes correspondientes en la actividad de destino, se ejecutará lo siguiente:

Código 7.34: Obtener EXTRA de un Intent en Android

```
1   val romUri: Uri? = intent.getStringExtra(ROM_URI_EXTRA)?.let { Uri.parse(it↩
        ↪ ) }
2
3   romUri?.let {
4       val inputStream = contentResolver.openInputStream(it)
5       val romBytes = inputStream?.readBytes()
6       inputStream?.close()
7
8       emulator.run(romBytes)
9   }
```

Con *intent.getStringExtra()* se puede obtener la URI del archivo que se ha proporcionado anteriormente en el Intent. Es necesario especificar la constante que identifica la clave bajo la cual se guardó el parámetro; en este caso, se utiliza *ROM_URI_EXTRA*.

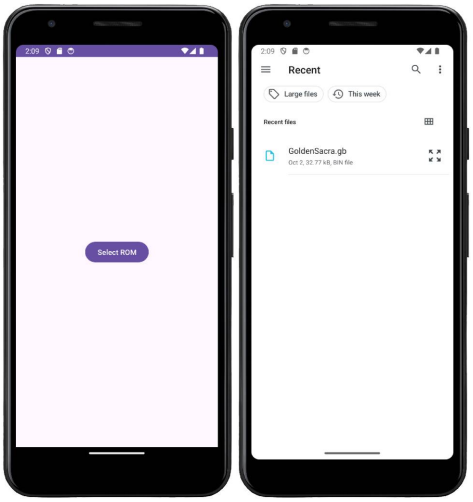

Figura 7.1: Selección de archivo desde la memoria SD del dispositivo.

A continuación, se procede a obtener los bytes del archivo y se envían al módulo de Emulator. En este momento, nos centraremos únicamente en la parte correspondiente a la ROM:

Código 7.35: Carga de ROM y manejo de errores durante el proceso.

```
fun load_rom(romBytes: ByteArray): Boolean{

    try {
        if(romBytes.isNotEmpty()){
            val cartSize = min(romBytes.size, ROM_END - ROM_START + 1)

            for (i in 0 until cartSize) {
                Memory.write(ROM_START + i, romBytes[i])
            }

            return rom_init(romBytes)
        }
    }catch (ex: Exception){
        println("Error loading ROM: $ex")
    }

    return false
}
```

En la función, se comprueba que los bytes pasados como parámetro no sean nulos. A continuación, se calcula el tamaño total de la ROM para asegurar que no exceda el tamaño máximo permitido (rango de 0x0000 a 0x7FFF). Se escriben todos los bytes en la memoria del emulador, y se invoca la función *rom_init()* para obtener datos como el título. Si alguna de las operaciones anteriores falla, se devuelve false y la ejecución se detiene.

Código 7.36: Carga de ROM y manejo de errores durante el proceso.

```
enum class CONSOLE_TYPE(val value : Int){
    DMG(0),
    DMG_CGB(0x80),
    CGB(0xC0),
    UNKNOWN(-1);

    companion object {
        fun fromValue(value: Int): CONSOLE_TYPE {
            return entries.find { it.value == value } ?: UNKNOWN
        }
    }
}

private var bootSection = ByteArray(0xFF)

private var cartTitle : String = "Unknown"
private var licenseCode : String = "None"
private var cartType : Int = -1
private var romSize : Int = -1
private var ramSize : Int = -1
```

```
21  private var romVersion : Int = -1
22  private var console: CONSOLE_TYPE = CONSOLE_TYPE.UNKNOWN
23
24  [...]
25
26  val newLicenseCodes: Map<String, String> = mapOf(
27      "00" to "None",
28      "01" to "Nintendo R&D1",
29      "08" to "Capcom",
30      "13" to "Electronic Arts",
31      [...]
32      "A4" to "Konami (Yu-Gi-Oh!)",
33  )
34
35  val oldLicenseCodes : Map<Int, String> = mapOf(
36      0x00 to "None",
37      0x01 to "Nintendo",
38      0x08 to "Capcom",
39      0x09 to "HOT-B",
40      [...]
41      0xFF to "LJN"
42  )
43
44  val cartTypes : Map<Int, String> = mapOf(
45      0x00 to "ROM ONLY",
46      0x01 to "MBC1",
47      0x02 to "MBC1+RAM",
48      0x03 to "MBC1+RAM+BATTERY",
49      0x05 to "MBC2",
50      [...]
51      0xFF to "HuC1+RAM+BATTERY"
52  )
53
54  val ramSizes : Map<Int, Int> = mapOf(
55      0x00 to 0, // KiB
56      0x01 to 2,
57      0x02 to 8,
58      0x03 to 32,
59      0x04 to 128,
60      0x05 to 64
61  )
62
63  [...]
64
65  fun convertBytesToString(bytes: ByteArray): String {
66      val title = bytes.takeWhile { it != 0.toByte() && it.toInt() in 32..126 ↩
        ↪ } // Filter only ASCII characters
67      return String(title.toByteArray(), Charsets.US_ASCII)
68  }
69
70  fun rom_init(romBytes: ByteArray): Boolean{
```

```
71
72      // Compare Cartridge Header with the Boot fixed one
73
74      // Get header section from the romBytes
75      val bootByteArray = Memory.getNintendoLogo()
76      val cartByteArray = extractByteArray(romBytes, N_LOGO_START, N_LOGO_END,↩
            ↪  true) // Nintendo Logo on Cartridge goes from 0x104 to 0x133
77
78      if(memcmp(Memory.getNintendoLogo(), cartByteArray, bootByteArray.size) ↩
            ↪ != 0){
79          return false
80      }
81
82      cartTitle = convertBytesToString(extractByteArray(romBytes, TITLE_START,↩
            ↪  TITLE_END, true))
83
84      licenseCode = if((extractByte(romBytes, OLD_LCNS_CODE).toInt() and 0xFF)↩
            ↪  == NEW_LICENSE_CODE){
85          getNewLicenseNameFromIndex(convertBytesToString(extractByteArray(↩
                ↪ romBytes, LCNS_CODE_START, LCNS_CODE_END, true)))
86      }else{
87          getOldLicenseNameFromIndex(extractByte(romBytes, OLD_LCNS_CODE).↩
                ↪ toInt() and 0xFF)
88      }
89
90      cartType = extractByte(romBytes, CART_TYPE).toInt() and 0xFF
91      romSize = 32 * (1 shl extractByte(romBytes, ROM_SIZE).toInt() and 0xFF) ↩
            ↪ // Value in KiB
92      ramSize = extractByte(romBytes, RAM_SIZE).toInt() and 0xFF
93      romVersion = extractByte(romBytes, ROM_V_NUM).toInt() and 0xFF
94      console = CONSOLE_TYPE.fromValue(extractByte(romBytes, TITLE_END).toInt↩
            ↪ () and 0xFF)
95
96      println("ROM Loaded Successfully!")
97
98      bootSection = extractByteArray(romBytes, 0x00, 0xFF, true) // Save ↩
            ↪ portion of code where the boot is going to load
99
100     return true
101 }
```

Desglosemos el código:

1. Se compara el logotipo de Nintendo almacenado en el módulo de memoria con el presente en el cartucho. Si no coinciden, se procede a finalizar la ejecución del programa.

2. Se obtienen los bytes correspondientes al título del juego y se convierten a una cadena de texto utilizando el conjunto de caracteres ASCII. Dado que la longitud del título depende de la versión del cartucho, se detendrá la lectura al encontrar el primer valor 0x00.

3. Se extrae el código de licencia. Primero, se verifica si el valor antiguo contiene 0x33

para utilizar los nuevos códigos de licencia. De lo contrario, se empleará el listado antiguo. Ambos listados pueden definirse mediante un par de mapas (*Maps*) para acceder rápidamente al valor correspondiente según el código obtenido.

4. A continuación, se extraen de manera directa los valores correspondientes al tipo de cartucho, tamaño de la ROM/RAM, versión de la ROM y el tipo de mapper.

5. Se realiza una copia de la región 0x0000-0x00FF del cartucho. La Game Boy "oculta" esta porción de código hasta que la secuencia de arranque ha finalizado.

Aquí los resultados obtenidos con la carga de las ROMS *Super Marioland*, *Pokémon Azul* y *Harry Potter y la Piedra Filosofal*:

Código 7.37: Valores obtenidos tras la carga de ROM.

```
--- Super Marioland
Cart Title: SUPER MARIOLAND
License: Nintendo
Cart Type: MBC1
ROM Size: 64
ROM Version Number: 0
RAM Size: 0
Console Type: DMG

--- Pokemon Azul
Cart Title: POKEMON BLUE
License: Nintendo R&D1
Cart Type: MBC5+RAM+BATTERY
ROM Size: 1024
ROM Version Number: 0
RAM Size: 3
Console Type: DMG

--- Harry Potter y la Piedra Filosofal
Cart Title: HARRYPOTTERBHVE
License: EA (Electronic Arts)
Cart Type: MBC5+RAM+BATTERY
ROM Size: 4096
ROM Version Number: 0
RAM Size: 2
Console Type: CGB
```

Además, se puede observar que la ROM se ha cargado correctamente en nuestra memoria virtual. En el caso de *Harry Potter* y la piedra filosofal, los siguientes bytes son visibles en la región 0x0000-0x015F:

Código 7.38: Visualización de la ROM cargada en la memoria virtual.

```
0000: E1 CD 9D 31 E9 87 87 87 85 6F 3E 00 8C 67 7E C9 | ...1.....o>..g~.
0010: 7A BC C0 7B BD C9 FF FF 3E 01 E0 A6 C9 FF FF FF | z..{....>.......
0020: AF E0 A6 3E 02 E0 A8 C9 7C E0 51 7D E0 52 7A E0 | ...>....|.Q}.Rz.
0030: 53 7B E0 54 79 E0 55 C9 40 F5 D1 7B EA BA CD C9 | S{.Ty.U.@..{....
0040: C3 80 36 FF FF FF FF FF C3 CE C0 C3 AC 23 FF FF | ..6..........#..
0050: FB C3 AC 39 FF FF FF FF C3 A3 38 FF FF FF FF FF | ...9......8.....
0060: D9 D7 38 04 D5 54 5D E1 CB 2A CB 1B CB 2A CB 1B | ..8..T]..*...*..
0070: CB 2A CB 1B 19 19 19 C9 CB 7C C8 F5 7D 2F C6 01 | .*.......|..}/..
0080: 6F 7C 2F CE 00 67 F1 C9 F5 79 2F C6 01 4F 78 2F | o|/..g...y/..Ox/
0090: CE 00 47 F1 C9 CB 7A C8 F5 7B 2F C6 01 5F 7A 2F | ..G...z..{/.._z/
00A0: CE 00 57 F1 C9 C5 06 08 AF 87 CB 15 30 04 84 30 | ..W.........0..0
00B0: 01 2C 05 20 F4 65 6F C1 C9 AF B5 20 03 65 37 C9 | .,. .eo.... .e7.
00C0: C5 D5 06 08 4D 2E 00 CB 14 CB 15 5D 7D 91 6F 3F | ....M......]}.o?
```

```
00D0: 38 01 6B 05 20 F1 CB 14 B7 D1 C1 C9 C5 4D 44 3E | 8.k. ........MD>
00E0: 0F 21 00 00 CB 23 CB 12 30 01 09 29 3D 20 F5 CB | .!...#..0..)= ..
00F0: 7A 28 01 09 C1 C9 19 3E 02 EA 00 20 5E 23 56 C9 | z(.....>... ^#V.
0100: 00 C3 50 01 CE ED 66 66 CC 0D 00 0B 03 73 00 83 | ..P...ff.....s..
0110: 00 0C 00 0D 00 08 11 1F 88 89 00 0E DC CC 6E E6 | ..............n.
0120: DD DD D9 99 BB BB 67 63 6E 0E EC CC DD DC 99 9F | ......gcn.......
0130: BB B9 33 3E 48 41 52 52 59 50 4F 54 54 45 52 42 | ..3>HARRYPOTTERB
0140: 48 56 45 C0 36 39 00 1B 07 02 01 33 00 D7 B4 78 | HVE.69.....3...x
0150: A7 FE 11 3E 00 20 01 3C E0 EF 31 FF CF F0 EF B7 | ...>. .<..1.....
```

7.4.2 Lectura/Escritura en ROM