

GBee



Máster Universitario en Desarrollo de
Software para Dispositivos Móviles

Trabajo Fin de Máster

Autor:

Ángel Jesús Terol Martínez

Tutor:

Luis Lucas Ibáñez



Mayo 2025

G Bee

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, Mayo 2025

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.

Resumen

GBee es una aplicación que funciona como **emulador de Game Boy**, desarrollado específicamente de forma nativa para dispositivos **Android** mediante el lenguaje de programación Kotlin, utilizando Android Studio como entorno de desarrollo, y se estructura en módulos que reflejan los componentes esenciales de la arquitectura de la consola.

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.

El núcleo del sistema es la CPU, que implementa las instrucciones del procesador Sharp LR35902, un derivado del Z80 de 8 bits. Para ello se ha realizado una interpretación directa de los opcodes, controlando ciclos de máquina y de reloj, y gestionando interrupciones. Además, se ha implementado un sistema de temporización precisa, basada en la frecuencia original, clave para asegurar una ejecución sincronizada con el resto de los subsistemas.

Uno de los componentes más complejos es la PPU, encargada del renderizado del fondo, la ventana y los sprites. Se ha replicado el comportamiento por estados, respetando los tiempos de cada uno y ajustando los cambios de modo en función de la línea de escaneo y otros registros. Se utiliza un SurfaceView para dibujar en pantalla, implementando una rutina de renderizado en bucle sincronizada con los 60 fotogramas por segundo que produce el hardware original.

El sistema de memoria incluye módulos para la RAM interna, RAM externa, I/O, video RAM, y memoria de cartucho, incluyendo soporte para bancos de memoria. Esto permite gestionar juegos más complejos que utilizan paginación de memoria tanto para la ROM como para la RAM.

En cuanto a la interacción con el usuario, se ha diseñado una interfaz de usuario intuitiva, con botones táctiles superpuestos al juego. El usuario puede seleccionar una ROM desde el almacenamiento del dispositivo mediante una funcionalidad de Android que permite navegar por el sistema de archivos.

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

3.1	Nintendo Logo - Mapa de Bits	23
3.2	Selección del banco de ROM en cartuchos grandes.	36
3.3	Selección del banco de ROM en modo 0.	37
7.1	Declaración de Registros	74
7.2	Ejemplo de Opcode	75
7.3	Actualización de Flags	75
7.4	Identificación de Opcode	76
7.5	Operaciones comunes	77
7.6	Operaciones LD	78
7.7	Operaciones ADD y ADC	78
7.8	Operaciones INC y DEC	79
7.9	Operación XOR	80
7.10	Operación CALL	81
7.11	Operaciones JR y JP	81
7.12	Operaciones RET y RETI	82
7.13	Operación CP	82
7.14	Operación CPL	83
7.15	Operación CCF	83
7.16	Operación DAA	83
7.17	Operación SCF	84
7.18	Operaciones RL y RR	85
7.19	Operaciones RLC y RRC	85
7.20	Operaciones SLA, SRA y SRL	86
7.21	Operación SWAP	87
7.22	Operación BIT	87
7.23	Operación RES	88
7.24	Operación SET	89
7.25	Operación NOP	90
7.26	Operaciones DI, EI	90
7.27	Operaciones PUSH, POP	91
7.28	Operaciones I/O	91
7.29	Declaraciones iniciales de Memoria	92
7.30	Métodos de lectura y escritura en memoria	93
7.31	Secuencia de arranque y logo de Nintendo	94
7.32	Copiado del Boot en memoria	97
7.33	Abrir archivos binarios en un Activity	98
7.34	Obtener EXTRA de un Intent en Android	99
7.35	Carga de ROM y manejo de errores durante el proceso.	100

7.36 Carga de ROM y manejo de errores durante el proceso.	100
7.37 Valores obtenidos tras la carga de ROM.	103
7.38 Visualización de la ROM cargada en la memoria virtual.	103
7.39 Inicialización del MBC.	104
7.40 Lectura en regiones ROM.	105
7.41 Escritura en regiones ROM.	106
7.42 Interrupciones en el método principal de la CPU.	107
7.43 Lógica principal del módulo Interrupt.	108
7.44 Lectura y escritura en registros de Timers.	109
7.45 Lectura y escritura en registros de Timers.	110
7.46 Lectura y escritura en Work RAM.	111
7.47 Lectura y escritura en Video RAM.	112
7.48 Inicio del proceso DMA.	113
7.49 Proceso DMA.	113
7.50 Inicialización de la PPU.	114
7.51 Lógica principal de la PPU.	115
7.52 Lógica del proceso de H-Blank.	116
7.53 Lógica del proceso de V-Blank.	117
7.54 Lógica del proceso de OAM Scan.	118
7.55 Obtención de sprites en modo OAM Scan.	118
7.56 Dibujado de píxeles en Modo 3.	119
7.57 Código principal del FIFO Fetcher.	121
7.58 FIFO Fetcher - Obtención de Tile.	122
7.59 FIFO Fetcher - Obtención de Tile de Background.	123
7.60 FIFO Fetcher - Obtención de Tile de Sprites.	123
7.61 FIFO Fetcher - Obtención de los Bits Bajos o Altos del Tile.	124
7.62 FIFO Fetcher - Obtención de Información de un Sprite en la Línea Actual.	125
7.63 FIFO Fetcher - Sleep.	126
7.64 FIFO Fetcher - Push.	126
7.65 FIFO Fetcher - Push de Píxeles al FIFO.	127
7.66 PPU - Paletas de Color.	127
7.67 Creación del Game Surface View.	128
7.68 Medición y Escalado del Canvas.	128
7.69 Medición y Escalado del Canvas.	129
7.70 Renderizado en el GameThread.	130
7.71 GameSurfaceView en Layout.	130
7.72 Estados de los Botones en el Módulo IO.	131
7.73 Escritura de los Estados de los Botones del Joypad	133
7.74 Enlace y Lógica de los Botones en la Actividad	134
7.75 MainActivity - Configuración del RecyclerView.	136
7.76 ROM Management - Añadir una ROM.	138
7.77 ROM Management - Eliminar una ROM.	139
7.78 ROM Management - Recarga de ROMs en Memoria.	140
7.79 ROM Adapter.	141
7.80 SettingsActivity - Inicialización.	143

7.81 SettingsActivity - Selección de Otro Fragmento.	143
7.82 SettingsActivity - Archivo de Preferencias.	144

1 Introducción

Desde su lanzamiento, la Game Boy se consolidó como una de las consolas portátiles más icónicas de la historia, marcando un hito en la industria de los videojuegos. Su diseño compacto y su amplia biblioteca de juegos la convirtieron en un fenómeno cultural, estableciendo un estándar para futuras consolas portátiles. Hoy en día, a pesar de la evolución tecnológica y la aparición de consolas más potentes, existe un gran interés por parte de desarrolladores y aficionados en revivir la experiencia de jugar a estos clásicos.

En paralelo, la emulación de consolas ha ganado popularidad, permitiendo a los usuarios disfrutar de juegos antiguos en plataformas modernas. La emulación no solo preserva la historia de los videojuegos, sino que también ofrece una oportunidad para explorar y comprender la arquitectura y el funcionamiento interno de estas consolas.

Aunque existen múltiples emuladores de Game Boy disponibles, muchos de ellos son de código cerrado y no permiten a los desarrolladores aprender sobre su funcionamiento interno.

Otro punto importante es el sistema de monetización de las aplicaciones. La mayoría de los emuladores disponibles en la Play Store o bien son de pago, lo que limita su accesibilidad, o bien están llenos de anuncios y funciones limitadas que dificultan la experiencia del usuario. Esto puede llevar a los usuarios a buscar alternativas no oficiales, que pueden no ser seguras o legales.

Este proyecto plantea el desarrollo de un emulador de Nintendo Game Boy para dispositivos Android, con el objetivo principal de replicar el comportamiento del hardware original de manera precisa y comprensible. La aplicación, llamada GBee, está diseñada desde cero en Kotlin y busca ofrecer una experiencia funcional y fiel a la consola original, permitiendo cargar y ejecutar ROMs reales en un entorno móvil.

Desde el punto de vista de desarrollador, la creación de un emulador completo es un reto técnico que engloba conocimientos de arquitectura de sistemas, sincronización de procesos, representación gráfica y diseño de interfaces táctiles. Se concibe como una herramienta de estudio, documentación y comprensión del funcionamiento interno de la consola, lo que lo hace especialmente útil como trabajo de fin de máster, y como base para futuras mejoras o estudios más profundos.

1.1 Motivación

El principal motivo de iniciar este proyecto ha sido el deseo de entender a fondo cómo puede llegar a simularse el comportamiento de una consola y entender el funcionamiento de la

Game Boy a un nivel más profundo. Asimismo, me resulta especialmente gratificante poder contribuir a que otras personas interesadas en este u otros proyectos similares encuentren en esta memoria una guía útil que les sirva de orientación.

Existe un valor personal y ha representado un reto técnico, tocando áreas como la programación de bajo nivel, el diseño gráfico y la interacción en dispositivos móviles.

1.2 Objetivos

El principal objetivo es desarrollar un emulador funcional de la consola portátil Game Boy para dispositivos Android. Este emulador, denominado GBee, tiene como finalidad reproducir de manera fiel el comportamiento de la consola original, permitiendo ejecutar ROMs comerciales y homebrew, manteniendo una experiencia lo más cercana posible a la que ofrecía el hardware original de Nintendo.

Para alcanzar este objetivo principal, se establecen los siguientes objetivos específicos:

- **Comprender funcionamiento interno de la consola Game Boy.**
- **Analizar librerías y frameworks existentes.**
- **Diseñar una arquitectura modular**, dividiendo el proyecto en componentes independientes como la CPU o la PPU.
- **Diseñar una interfaz gráfica funcional en Android**, que permita al usuario seleccionar una ROM, iniciar la emulación, visualizar la pantalla de la consola y controlar el juego mediante botones táctiles.
- **Ofrecer y garantizar el funcionamiento correcto en un conjunto representativo de juegos.**
- **Documentar el desarrollo del proyecto.**
- **Publicar la aplicación en la Play Store.**

1.3 Metodología

Para llevar a cabo el desarrollo del proyecto, se va a seguir una metodología de trabajo basada en fases iterativas y progresivas. Esta estrategia permite abordar de forma estructurada tanto los aspectos más técnicos como los de diseño. A continuación, se describen las principales etapas a seguir:

- **Estudio de mercado:** Investigar el estado actual de los emuladores ya existentes, tanto de plataformas de escritorio como móviles. Esto permite identificar patrones, puntos de interés y carencias de aplicaciones similares a la que se pretende desarrollar.
-

- **Diseño de la arquitectura:** Se define la estructura interna del emulador. Verificar si dividiendo el sistema en componentes independientes como CPU, PPU, Memoria, etc. puede favorecer la escalabilidad y facilitar el aislamiento de errores durante el desarrollo.
- **Selección de tecnologías:** Definir si el proyecto se va a implementar completamente en lenguaje Kotlin, o usar tecnologías adyacentes como Ionic. Estudiar también cómo se puede implementar el manejo de gráficos.
- **Implementación por fases:** Realizar la implementación de cada uno de los módulos de forma independiente, comenzando por los más sencillos y avanzando hacia los más complejos. Esto permite realizar pruebas unitarias y asegurar que cada componente funciona correctamente antes de integrarlo en el sistema completo.
- **Pruebas y validación:** Realizar pruebas exhaustivas de cada módulo y del sistema completo, asegurando que la emulación es precisa y que se pueden cargar y ejecutar juegos sin problemas.
- **Diseño de la interfaz de usuario:** Crear una interfaz gráfica intuitiva y fácil de usar, que permita a los usuarios interactuar con el emulador de manera sencilla. Esto incluye la implementación de controles táctiles y la visualización de la pantalla de la consola.
- **Mejoras:** Una vez que el emulador esté funcionando correctamente, se pueden implementar mejoras adicionales, como la optimización del rendimiento, la adición de funciones avanzadas (como guardado de estados o soporte para diferentes tipos de ROMs) y la corrección de errores.

1.4 Estructura del documento

La presente memoria se encuentra organizada en varios capítulos que abordan de manera progresiva los distintos aspectos relacionados con el desarrollo del emulador de Game Boy en Android. A continuación, se detalla brevemente el contenido de cada uno de ellos:

- **Capítulo 1: Introducción.** En este capítulo se presenta el contexto del proyecto, la motivación detrás de su desarrollo y los objetivos que se pretenden alcanzar.
 - **Capítulo 2: Terminología.** Se definen los términos y acrónimos utilizados a lo largo del documento, facilitando la comprensión de conceptos técnicos y específicos relacionados con la emulación y el desarrollo de software.
 - **Capítulo 3: Marco Teórico.** En este capítulo se presenta un análisis de la historia de la Game Boy y su evolución a lo largo del tiempo. Se abordan aspectos técnicos relevantes que implican sobre todo aspectos de hardware. Se definen y explica el funcionamiento de los registros más importantes que se deben tener en cuenta a la hora de implementar el emulador.
 - **Capítulo 4: Metodología y Planificación.** Se describe la metodología ágil de trabajo seguida durante el desarrollo, así como los requerimientos, herramientas a utilizar y etapas del proyecto.
-

- **Capítulo 5: Diseño.** En este capítulo se hace un estudio de los emuladores ya existentes que se van a utilizar como referentes durante el desarrollo. También se presenta un diseño inicial de la interfaz gráfica del emulador y cómo va a ser el flujo de trabajo del usuario.
- **Capítulo 6: Desarrollo.** Se detalla el proceso de desarrollo del emulador, incluyendo la implementación de los diferentes módulos y componentes. Se procura ofrecer ejemplos de código y explicaciones sobre las decisiones tomadas durante el desarrollo. Es el capítulo más extenso y técnico del documento.
- **Capítulo 7: Resultados.** En este capítulo se presentan los resultados obtenidos tras la implementación del emulador, incluyendo pruebas realizadas y su rendimiento en diferentes dispositivos Android.
- **Capítulo 8: Conclusiones.** Se detallan unas conclusiones finales sobre todo el proceso de desarrollo y se detallan unas posibles mejoras a futuro.

2 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, independientemente 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 caracteres que identifican un recurso online o local.
- **Intent:** objeto en Android que se utiliza para comunicar componentes.
- **SPI:** Serial Peripheral Interface. Protocolo de comunicación serial sincrónico que permite el intercambio de datos entre un dispositivo maestro y uno o más esclavos.

3 Marco Teórico

Antes de adentrarse en el desarrollo del emulador, es fundamental comprender en profundidad el funcionamiento del sistema original que se pretende replicar. Este capítulo tiene como objetivo presentar los fundamentos teóricos necesarios para entender la arquitectura y comportamiento del hardware de la consola, sentando así una base sólida sobre la que construir el proyecto.

En primer lugar, se ofrece una descripción general de la Game Boy, abordando su contexto histórico y las características que la convirtieron en un referente dentro del mundo de los videojuegos portátiles. A continuación, se detallan sus especificaciones técnicas principales, como la CPU o la resolución de pantalla.

Asimismo, se analizan los registros más relevantes que intervienen en el funcionamiento interno del sistema, incluyendo aquellos relacionados con el control de pantalla, scroll, temporización o manejo de sprites, entre otros. Estos registros juegan un papel clave en la emulación, ya que reflejan directamente el estado del hardware en cada instante.

Este marco teórico no solo proporciona los conocimientos necesarios para entender cómo funciona la Game Boy, sino que también sirve de guía para tomar decisiones de implementación durante las fases posteriores del desarrollo del emulador.

3.1 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 3.1: 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 3.2: Versiones DMG, MGB, GBL y CGB de Game Boy

3.1.1 Especificaciones Técnicas

Cuando pensamos en programar un emulador que simule una máquina en concreto lo primero que debemos hacer es conocer exactamente **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 Poc- ket (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
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 3.1: Especificaciones técnicas de la Game Boy.

¹La 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

3.1.2 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 3.2: Mapa de memoria de Game Boy

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

3.1.2.2 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.

Se divide en tres secciones:

- **0x8000 - 0x97FF:** Bloque de memoria en el que se almacena la información de los tiles. Cada tile ocupa 16 bytes, por lo que forma general accederemos a la información de cada uno multiplicando su índice por 16 (el tile 0 hará referencia a la posición 0x8000, y el tile 1 a la posición 0x8010).
 - **0x9800 - 0x9BFF:** Bloque de memoria de 1024 bytes en el que se almacena la información de los mapas de fondo (Background y Window). La información guardada deberán ser los índices de los tiles a dibujar.
 - **0x9C00 - 0x9FFF:** Espejo de la sección anterior, que se utiliza para almacenar la información de los mapas de fondo en caso de que se necesite más memoria.

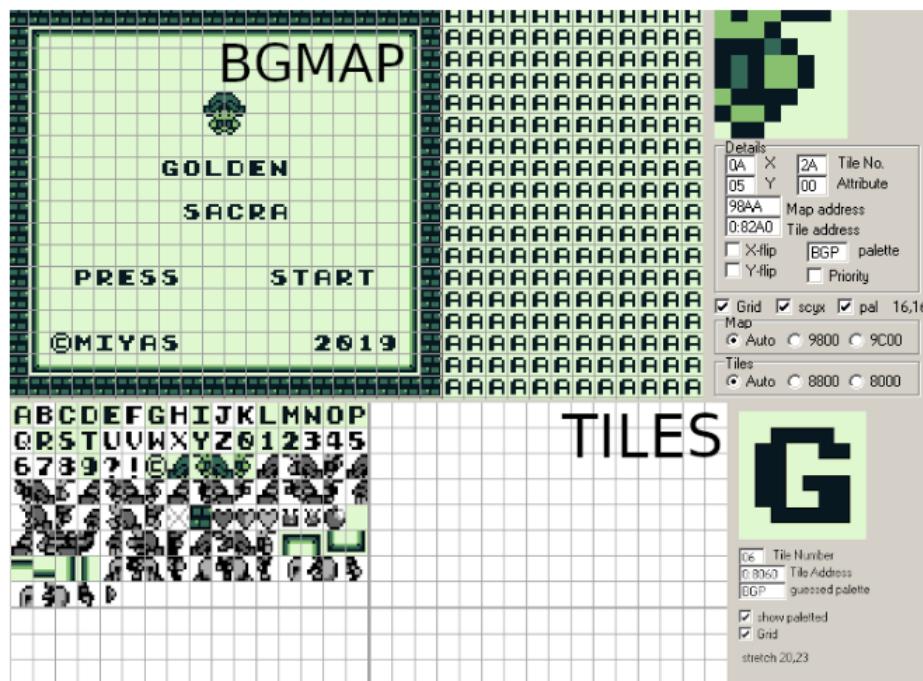


Figura 3.3: Visualización de la memoria VRAM (?)

3.1.2.3 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.

3.1.2.4 Work RAM: [0xC000 - 0xFFFF]

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 (0xD000–0xFFFF) también puede ser conmutada entre diferentes bancos para aumentar la capacidad de almacenamiento.

3.1.2.5 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.

3.1.2.6 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. Además, existe una limitación adicional por la que solamente se pueden evaluar hasta 3 sprites superpuestos en un mismo punto.

En la siguiente figura, se muestra cómo se guardarían los tiles 4 y 6 en memoria para que en pantalla se muestren de manera continua:

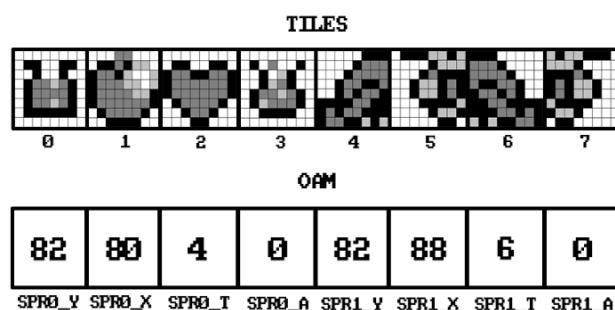


Figura 3.4: Datos de tiles en OAM (?)

A continuación se describen los datos de forma más específica:

- **Byte 0 - Posición Y**, el cual tiene un offset de 16 píxeles. Y = 0 dejaría el sprite escondido, mientras que Y = 16 lo mostraría en la parte superior de pantalla.

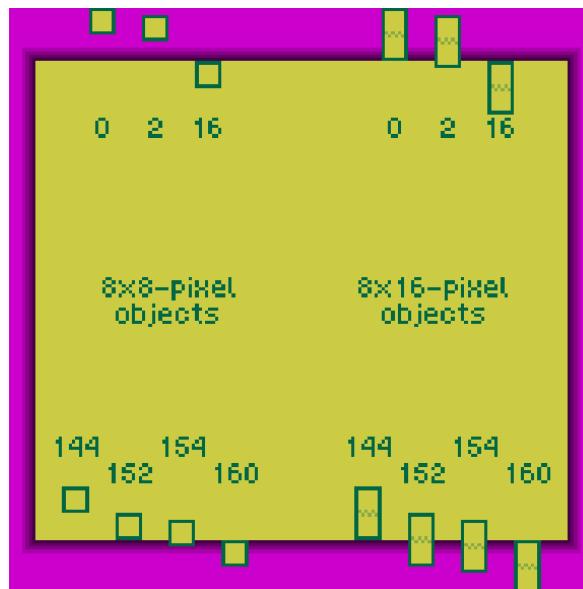


Figura 3.5: Posiciones Y de los sprites (?)

- **Byte 1 - Posición X**, el cual tiene un offset de 8 píxeles.
- **Byte 2 - Índice de tile**. En el modo 8x8, este byte especifica un índice único de tile, dentro del rango [0x00-0xFF]. La consola selecciona internamente el tile en la VRAM, dentro de la región de memoria 0x8000 - 0x97FF. En el modo 8x16, este índice corresponde al tile superior del objeto, y la consola utiliza el tile contiguo en memoria para el tile inferior, ajustando automáticamente la referencia en memoria. En modo CGB el índice se puede referir a los bancos 0 o 1 de VRAM, dependiendo del bit 3 del byte 3.
- **Byte 3 - Atributos**. Aplican ciertos efectos o ajustes al sprite:
 - **Bit 7**: Control de la prioridad del sprite. Si su valor es 1, los tiles de Background y Window cuyos índices de color vayan del 1 al 3 se renderizarán por encima del objeto.
 - **Bit 6**: Flip Y del sprite.
 - **Bit 5**: Flip X del sprite.
 - **Bit 4**: Selecciona la paleta de color para modo DMG. 0 = OBP0, 1 = OBP1.
 - **Bit 3**: Para modos CGB, selecciona el banco de VRAM a utilizar.
 - **Bit 2-0**: Selecciona la paleta de color para modo CGB. Va de OBP0 hasta OBP7.

3.1.2.6.1 DMA - 0xFF46: Mecanismo que permite la transferencia eficiente de datos desde la memoria ROM o RAM hacia la memoria OAM sin la intervención directa de la CPU. En lugar de copiar los datos byte a byte mediante instrucciones de la CPU, un controlador dedicado se encarga de mover grandes bloques de datos de manera optimizada.

Para iniciar una transferencia DMA, se debe escribir cualquier valor en el registro 0xFF46. El valor insertado se interpretará como la parte alta de la dirección de origen y se multiplicará por 0x100. Por ejemplo, si se escribe 0x80, la transferencia comenzará desde 0x8000, mientras que si se introduce 0xC2, se iniciará desde 0xC200.

Este proceso copia un total de **160 bytes** y requiere **160 ciclos de máquina**. En el modo *CGB Double Speed*, este tiempo se reduce a la mitad. Durante la transferencia, la **CPU queda inactiva y no puede ejecutar código alojado en ROM**, ya que solo mantiene acceso a la **HRAM** (0xFF80 - 0xFFFF).

3.1.2.7 No utilizable: [0xFEAO - 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.

3.1.2.8 I/O: [0xFEAO - 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.

3.1.2.9 HRAM: [0xFF80 - 0xFFFF]

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.

3.1.2.10 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.

3.2 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.

3.2.1 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).

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 3.3: 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.

3.2.2 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	Subtraction: 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 3.4: 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.

3.2.3 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.

3.2.3.1 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 llenando 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.

3.2.3.2 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	LD BC,d16	LD (BC),A	INC BC	INC B	DEC B	LD B,d8	RCL A	LD (a16),SP	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C,d8	RRCA	
1x	1 4	3 12	- - -	1 8	1 4	2 8	- - -	0 0 0 C	3 20	1 8	1 8	1 8	1 4	2 8	2 8	0 0 0 C	
	STOP	LD DE,d16	LD (DE),A	INC DE	INC D	DEC D	LD D,d8	RRA	JR +8	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,d8	RRA	
2x	2 4	3 12	- - -	1 8	1 4	2 8	- - -	0 0 0 C	1 4	2 8	1 8	1 8	1 4	2 8	2 8	1 4	
	JR NZ,r8	JR HL,d16	LD (HL),A	INC H	INC H	DEC H	LD H,d8	DAA	JR Z,r8	ADD HL,HL	LD A,(HL)	DEC HL	INC L	DEC L	LD L,d8	CPL	
3x	2 12/8	3 12	- - -	1 8	1 8	2 8	- - -	0 0 0 C	2 12/8	1 8	1 8	1 8	1 4	2 8	2 8	1 4	
	JR NC,r8	LD SP,d16	LD (HL),A	INC SP	INC (HL)	DEC (HL)	LD (HL),d8	SCF	JR C,r8	ADD HL,SP	LD A,(HL)	DEC SP	INC A	DEC A	LD A,d8	CCF	
4x	2 12/8	3 12	- - -	1 8	1 12	2 12	1 4	2 12/8	1 8	1 8	1 8	1 8	1 4	2 8	2 8	1 4	
	LD B,B	LD B,C	LD B,D	LD B,E	LD B,F	LD B,L	LD B,(HL)	LD B,A	LD C,B	LD C,(HL)	LD C,D	LD C,E	LD C,H	LD C,I	LD C,d8	LD C,(HL)	
5x	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	
	LD D,B	LD D,C	LD D,D	LD D,E	LD D,F	LD D,L	LD D,(HL)	LD D,A	LD E,B	LD E,(HL)	LD E,D	LD E,E	LD E,H	LD E,I	LD E,d8	LD E,(HL)	
6x	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	
	LD H,B	LD H,C	LD H,D	LD H,E	LD H,F	LD H,L	LD H,(HL)	LD H,A	LD I,B	LD I,(HL)	LD I,D	LD I,E	LD I,H	LD I,L	LD I,d8	LD I,(HL)	
7x	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A	LD A,B	LD A,(HL)	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A	
	1 4	1 8	1 8	1 8	1 8	1 8	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	
8x	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A	ADD A,B	ADD A,(HL)	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A	
	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	
	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	
9x	SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB (HL)	SUB A	SBC A,B	SBC A,(HL)	SBC A,D	SBC A,E	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A	
	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	
	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	
	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A	XOR C	XOR (HL)	XOR D	XOR E	XOR H	XOR L	XOR (HL)	XOR A	
Ax	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	
	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	
Bx	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A	CP B	CP C	CP D	CP E	CP H	CP L	CP (HL)	CP A	
	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	
	Z 0 0 0	Z 0 0 0	Z 0 0 0	Z 0 0 0	Z 0 0 0	Z 0 0 0	Z 0 0 0	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	
Cx	RET NZ	POP BC	JP NZ,a16	JP a16	CALL NZ,a16	PUSH BC	ADD A,8	RST 00H	RET Z	RET	JP Z,a16	PREFIX CB	CALL a16	CALL a16	ADC A,d8	RST 00H	
	1 20/8	1 12	3 16/12	3 16	3 14/12	1 16	1 16	1 16	1 16	1 16	3 16/12	1 4	3 24/12	3 24/12	1 16	1 16	
Dx	RET NC	POP DE	JP NC,a16	CALL NC,a16	PUSH DE	SUB d8	RST 10H	RET C	RET C	JP (HL)	JP C,a16	LD (a16),A	CALL C,a16	CALL C,a16	SBC A,d8	RST 18H	
	1 20/8	1 12	3 16/12	3 16/12	3 24/12	1 16	1 16	1 16	1 16	1 16	3 16/12	1 4	3 24/12	3 24/12	1 16	1 16	
Ey	LDH (a8),A	POP HL	LD (C),A	DI	PUSH AF	AM 8B	RST 20H	ADD SP,r8	JP (HL)	LD A,(a16)	LD A,(a16)	LD A,(a16)	XOR B	XOR B	XOR B	RST 28H	
	2 12	1 12	2 8	1 4	DI	1 16	1 16	2 16	1 4	1 16	3 16	1 4	3 24/12	3 24/12	1 16	1 16	
Fx	LDH A,(a8)	POP AF	LD A,(C)	DI	PUSH AF	OR d8	RST 30H	LD HL,SP+r8	LD SP,HL	LD A,(a16)	LD A,(a16)	LD A,(a16)	EE	EE	EE	CP d8	RST 38H
	2 12	1 12	2 8	1 4	DI	1 16	1 16	2 12	1 8	1 16	3 16	1 4	3 24/12	3 24/12	1 16	1 16	
	Z N H C	Z N H C	- - -	- - -	- - -	Z 0 0 0	Z 0 0 0	Z 0 0 0 C	Z 0 0 0 C	Z 0 0 0	Z 0 0 0	Z 0 0 0	Z 0 0 0	Z 0 0 0	Z 0 0 0	Z 0 0 0	

Figura 3.6: 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 16 Z 0 0 C	RRC_(HL) 2 8 Z 0 0 C	RRC_A 2 8 Z 0 0 C
1x	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 16 Z 0 0 C	RRC_(HL) 2 8 Z 0 0 C	RRC_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 16 Z 0 0 0	SRA_(HL) 2 8 Z 0 0 0	SRA_A 2 8 Z 0 0 0
3x	SHLD_B 2 8 Z 0 0 0	SHLD_C 2 8 Z 0 0 0	SHLD_D 2 8 Z 0 0 0	SHLD_E 2 8 Z 0 0 0	SHLD_H 2 8 Z 0 0 0	SHLD_L 2 8 Z 0 0 0	SHLD_(HL) 2 16 Z 0 0 0	SHLD_A 2 8 Z 0 0 0	SHLD_B 2 8 Z 0 0 0	SHLD_C 2 8 Z 0 0 0	SHLD_D 2 8 Z 0 0 0	SHLD_E 2 8 Z 0 0 0	SHLD_H 2 8 Z 0 0 0	SHLD_L 2 16 Z 0 0 0	SHLD_(HL) 2 8 Z 0 0 0	SHLD_A 2 8 Z 0 0 0
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 16 Z 0 1 -	BIT_1,(HL) 2 8 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 16 Z 0 1 -	BIT_3,(HL) 2 8 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 16 Z 0 1 -	BIT_5,(HL) 2 8 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 16 Z 0 1 -	BIT_7,(HL) 2 8 Z 0 1 -	BIT_7,A 2 8 Z 0 1 -	
8x	RES_0,B 2 8 Z 0 1 -	RES_0,C 2 8 Z 0 1 -	RES_0,D 2 8 Z 0 1 -	RES_0,E 2 8 Z 0 1 -	RES_0,H 2 8 Z 0 1 -	RES_0,L 2 8 Z 0 1 -	RES_0,(HL) 2 16 Z 0 1 -	RES_0,A 2 8 Z 0 1 -	RES_1,B 2 8 Z 0 1 -	RES_1,C 2 8 Z 0 1 -	RES_1,D 2 8 Z 0 1 -	RES_1,E 2 8 Z 0 1 -	RES_1,H 2 16 Z 0 1 -	RES_1,(HL) 2 8 Z 0 1 -	RES_1,A 2 8 Z 0 1 -	
9x	RES_2,B 2 8 Z 0 1 -	RES_2,C 2 8 Z 0 1 -	RES_2,D 2 8 Z 0 1 -	RES_2,E 2 8 Z 0 1 -	RES_2,H 2 8 Z 0 1 -	RES_2,L 2 8 Z 0 1 -	RES_2,(HL) 2 16 Z 0 1 -	RES_2,A 2 8 Z 0 1 -	RES_3,B 2 8 Z 0 1 -	RES_3,C 2 8 Z 0 1 -	RES_3,D 2 8 Z 0 1 -	RES_3,E 2 8 Z 0 1 -	RES_3,H 2 16 Z 0 1 -	RES_3,(HL) 2 8 Z 0 1 -	RES_3,A 2 8 Z 0 1 -	
Ax	RES_4,B 2 8 Z 0 1 -	RES_4,C 2 8 Z 0 1 -	RES_4,D 2 8 Z 0 1 -	RES_4,E 2 8 Z 0 1 -	RES_4,H 2 8 Z 0 1 -	RES_4,L 2 8 Z 0 1 -	RES_4,(HL) 2 16 Z 0 1 -	RES_4,A 2 8 Z 0 1 -	RES_5,B 2 8 Z 0 1 -	RES_5,C 2 8 Z 0 1 -	RES_5,D 2 8 Z 0 1 -	RES_5,E 2 8 Z 0 1 -	RES_5,H 2 16 Z 0 1 -	RES_5,(HL) 2 8 Z 0 1 -	RES_5,A 2 8 Z 0 1 -	
Bx	RES_6,B 2 8 Z 0 1 -	RES_6,C 2 8 Z 0 1 -	RES_6,D 2 8 Z 0 1 -	RES_6,E 2 8 Z 0 1 -	RES_6,H 2 8 Z 0 1 -	RES_6,L 2 8 Z 0 1 -	RES_6,(HL) 2 16 Z 0 1 -	RES_6,A 2 8 Z 0 1 -	RES_7,B 2 8 Z 0 1 -	RES_7,C 2 8 Z 0 1 -	RES_7,D 2 8 Z 0 1 -	RES_7,E 2 8 Z 0 1 -	RES_7,H 2 16 Z 0 1 -	RES_7,(HL) 2 8 Z 0 1 -	RES_7,A 2 8 Z 0 1 -	
Cx	SET_0,B 2 8 Z 0 1 -	SET_0,C 2 8 Z 0 1 -	SET_0,D 2 8 Z 0 1 -	SET_0,E 2 8 Z 0 1 -	SET_0,H 2 8 Z 0 1 -	SET_0,L 2 8 Z 0 1 -	SET_0,(HL) 2 16 Z 0 1 -	SET_0,A 2 8 Z 0 1 -	SET_1,B 2 8 Z 0 1 -	SET_1,C 2 8 Z 0 1 -	SET_1,D 2 8 Z 0 1 -	SET_1,E 2 8 Z 0 1 -	SET_1,H 2 16 Z 0 1 -	SET_1,(HL) 2 8 Z 0 1 -	SET_1,A 2 8 Z 0 1 -	
Dx	SET_2,B 2 8 Z 0 1 -	SET_2,C 2 8 Z 0 1 -	SET_2,D 2 8 Z 0 1 -	SET_2,E 2 8 Z 0 1 -	SET_2,H 2 8 Z 0 1 -	SET_2,L 2 8 Z 0 1 -	SET_2,(HL) 2 16 Z 0 1 -	SET_2,A 2 8 Z 0 1 -	SET_3,B 2 8 Z 0 1 -	SET_3,C 2 8 Z 0 1 -	SET_3,D 2 8 Z 0 1 -	SET_3,E 2 8 Z 0 1 -	SET_3,H 2 16 Z 0 1 -	SET_3,(HL) 2 8 Z 0 1 -	SET_3,A 2 8 Z 0 1 -	
Ex	SET_4,B 2 8 Z 0 1 -	SET_4,C 2 8 Z 0 1 -	SET_4,D 2 8 Z 0 1 -	SET_4,E 2 8 Z 0 1 -	SET_4,H 2 8 Z 0 1 -	SET_4,L 2 8 Z 0 1 -	SET_4,(HL) 2 16 Z 0 1 -	SET_4,A 2 8 Z 0 1 -	SET_5,B 2 8 Z 0 1 -	SET_5,C 2 8 Z 0 1 -	SET_5,D 2 8 Z 0 1 -	SET_5,E 2 8 Z 0 1 -	SET_5,H 2 16 Z 0 1 -	SET_5,(HL) 2 8 Z 0 1 -	SET_5,A 2 8 Z 0 1 -	
Fx	SET_6,B 2 8 Z 0 1 -	SET_6,C 2 8 Z 0 1 -	SET_6,D 2 8 Z 0 1 -	SET_6,E 2 8 Z 0 1 -	SET_6,H 2 8 Z 0 1 -	SET_6,L 2 8 Z 0 1 -	SET_6,(HL) 2 16 Z 0 1 -	SET_6,A 2 8 Z 0 1 -	SET_7,B 2 8 Z 0 1 -	SET_7,C 2 8 Z 0 1 -	SET_7,D 2 8 Z 0 1 -	SET_7,E 2 8 Z 0 1 -	SET_7,H 2 16 Z 0 1 -	SET_7,(HL) 2 8 Z 0 1 -	SET_7,A 2 8 Z 0 1 -	

Figura 3.7: 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.

3.2.4 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.

3.2.4.1 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.

3.2.4.2 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.

3.3 ROM

3.3.1 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 3.5: Resumen de las variaciones de las secuencias de arranque

3.3.1.1 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 ®.

3.3.1.2 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.

3.3.2 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:

3.3.2.1 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.

3.3.2.2 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 3.1: Nintendo Logo - Mapa de Bits

```

1 CE ED 66 66 CC 0D 00 0B 03 73 00 83 00 0C 00 0D
2 00 08 11 1F 88 89 00 0E DC CC 6E E6 DD DD D9 99
3 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.

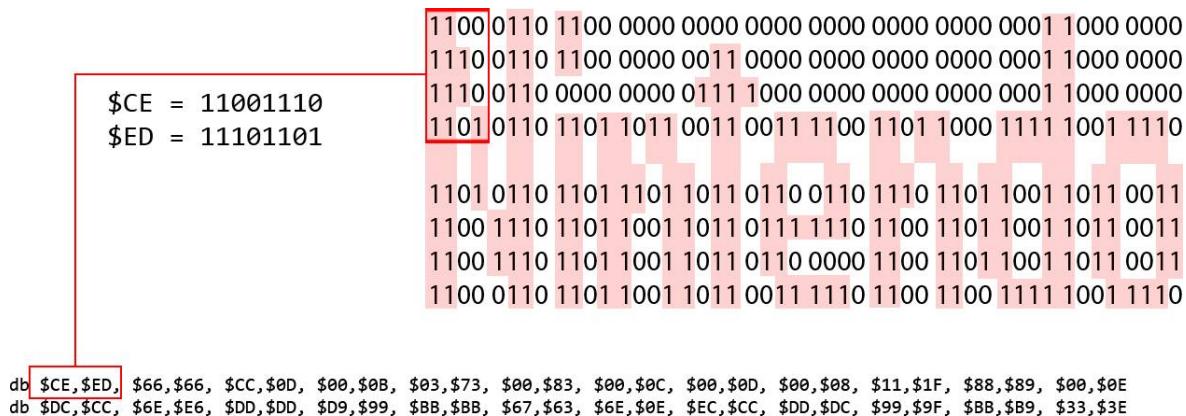


Figura 3.8: Decodificación del logo de Nintendo

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).

3.3.2.3 0x0134 - 0x0143: Título

Esta región de bytes contienen el título del juego, con caracteres ASCII y completamente en mayúsculas. Si el título tiene menos de 16 caracteres, 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 caracteres (siempre empezando en la dirección 0x134).

3.3.2.4 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.

3.3.2.5 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 monocromática ("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 retro-compatible 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.

3.3.2.6 0x0144 – 0x0145: Nuevo código de licencia

Esta región contiene un código representado por dos caracteres 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 3.6: Código de licencia y su editor.**3.3.2.7 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.

3.3.2.8 0x0147: Tipo de cartucho

Especifica qué tipo de hardware está presente en el cartucho (para ser más específico, 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 3.7: Tipos de mapeadores (MBC) y sus códigos**3.3.2.9 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 3.8: Tamaño y número de bancos de ROM según el valor especificado.**3.3.2.10 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 3.9: 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.

3.3.2.11 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.

3.3.2.12 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	Trifix 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 3.10: Antiguos códigos de licencia y su editor.

3.3.2.13 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).

3.3.2.14 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.

3.3.2.15 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).

3.4 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. Se va a describir a continuación el funcionamiento de dos tipos de MBC, lo que para este proyecto es suficiente.

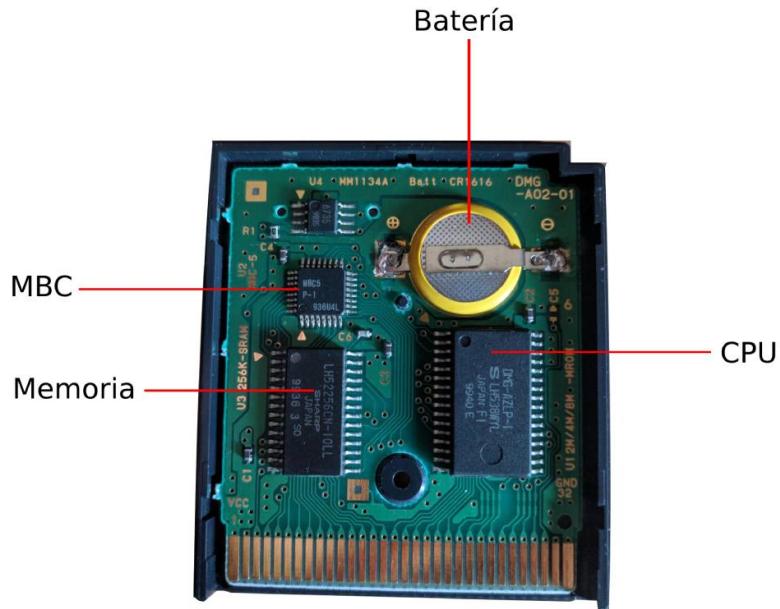


Figura 3.9: Diagrama - Cartucho de Game Boy MBC5

3.4.1 No MBC: 0x00

Los juegos pequeños, que tienen un tamaño de ROM de hasta 32 KiB, no necesitan 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.



Figura 3.10: PCB sin MBC (?)

3.4.2 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.

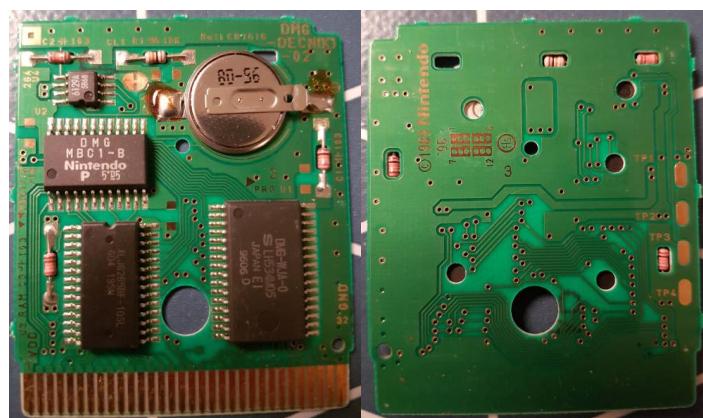


Figura 3.11: PCB MBC1 (?)

3.4.2.1 Memoria

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

3.4.2.1.2 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.

3.4.2.1.3 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.

3.4.2.2 Registros

Las siglas ROM hacen referencia a la Read Only Memory, es decir, memoria de solo lectura. Este tipo de memoria tiene la característica de ser no modificable durante la ejecución del programa. En el caso de la Game Boy, a pesar de que algunas regiones de la memoria ROM se comportan como registros, no se puede escribir en ellas directamente. La diferencia radica en que, aunque las instrucciones pueden apuntar a estas regiones como si se tratase de registros, los valores no se almacenan físicamente en la ROM. En su lugar, la operación tiene efecto según la región designada, sin que el valor sea guardado en esa área específica de la ROM. Este comportamiento es gestionado por el controlador, que permite la alternancia entre distintas partes de la ROM en ciertos momentos durante la ejecución del juego, sin modificar el contenido original de la memoria.

Todos estos registros tienen por defecto el valor 0x00. Para el banco de ROM será tratado como 0x01.

3.4.2.2.1 Habilitar RAM Externa: 0x0000-0x1FFF Para poder acceder a la RAM externa, antes debe ser activada mediante este registro. La forma de hacerlo es escribir en la región cualquier valor cuyos 4 nibbles inferiores sean el valor 0xA.

Se recomienda a los desarrolladores deshabilitar la RAM externa una vez se haya accedido a ella, ya que si se extrae el cartucho o se apaga la consola se pueden corromper los datos.

3.4.2.2.2 Número de Banco ROM: 0x2000–0x3FFF Este registro considera únicamente los primeros 5 bits del valor escrito, lo que implica que los valores efectivos estarán en el rango de 0x01 a 0x1F.

En caso de que se intente escribir el valor 0x00, el sistema lo interpretará como 0x01, debido a que el banco fijo de ROM no puede ser duplicado en ambas regiones de memoria.

Si se selecciona un número de banco mayor que el número de bancos disponibles en el cartucho, el valor del banco se ajustará (enmascarará) al número máximo permitido. Por ejemplo, en un cartucho de 256 KiB, que solo requiere 4 bits para direccionar uno de sus 16 bancos, el bit 5 será ignorado.

En el caso de ROMs más pequeñas, que no requieren el uso completo de los 5 bits, existe una excepción que permite duplicar el banco 0 en la región 0x4000–0x7FFF. Si se escribe el valor 0x10, los primeros 4 bits indicarán la selección del banco 0, que normalmente se traduciría al banco 1. Sin embargo, dado que el bit 5 está activado, el valor completo no será interpretado como 0x00, lo que evita la traducción 00→01. Esto permite tener el banco 0 mapeado tanto en las regiones de memoria 0x0000–0x3FFF como en 0x4000–0x7FFF.

En los casos en que la ROM es muy grande y utilice parte de la RAM para llegar a un máximo de 2MiB, se podrá llegar a utilizar un total de 128 bancos. Con 5 bits, no es posible representar todos ellos (el máximo es 0x1F). Para poder hacerlo, se deberán utilizar los dos bits del registro de la región 0x4000–0x5FFF de forma complementaria. El banco de ROM resultante se calcularía de la siguiente manera: $BancoSeleccionado = (RegistroSecundario \ll 5) + BancoROM$.

Por ejemplo, si se quisiese seleccionar el banco número 101, nuestro registro secundario debería tener un valor de 0x03 y el registro de ROM un 0x05:

Código 3.2: Selección del banco de ROM en cartuchos grandes.

```

1 00000011 << 5 = 01100000 --> el valor 0x03 (0011) pasa a valer 0x60 (96 en decimal)
2
3   0110 0000
4   + 0000 0101
5   -----
6   0110 0101 = 101 en decimal

```

3.4.2.2.3 Número de Banco RAM / Complementario de Banco ROM: 0x4000–0x5FFF

Registro de 2 bits que se utiliza para seleccionar el número de banco de RAM (0x00–0x03) o especificar de forma complementaria el banco de ROM. Si la ROM y la RAM no son lo suficientemente grandes, este registro no tiene ningún uso.

3.4.2.2.4 Modo de Banco: 0x6000–0x7FFF Este registro de 1 bit controla el comportamiento del registro de 2 bit mencionado anteriormente. Los dos comportamientos pueden

ser:

- **0x00:** Modo por defecto. Los dos bits del registro 0x4000–0x5FFF se utilizan como bits superiores en la selección del banco de ROM. En este modo solamente se puede acceder al banco 0 de RAM. Este modo permite acceder a un tamaño máximo de 8KiB de RAM y 2MiB de ROM.
- **0x01:** Modo avanzado en el que las regiones 0x0000-0x3FFF y 0xA000-0xBFFF no son fijas. Los dos bits del registro 0x4000–0x5FFF se utilizan para seleccionar el banco de RAM. Esto permite un máximo de 32KiB de RAM y 512KiB de ROM. Solamente se podrán acceder a los bancos del 0x00 al 0x1F de ROM.

La forma en la que estos modos funcionan es mediante unas puertas AND que el controlador tiene entre los dos registros de selección de bancos y el segundo bit más alto de la dirección de memoria. Estas puertas en el modo 1 se deshabilitan. Un ejemplo del modo 0 a continuación:

Supongamos que el registro de banco ROM contiene un 5, siendo la región de memoria 0x0000-0x3FFF:

Código 3.3: Selección del banco de ROM en modo 0.

```

1      00101 -> 5 en binario
2      AND 00000 -> (segundo bit más alto de la dirección en la región 0x0000-0x3FFF)
3      -----
4      00000 (resultado, siempre 0)
```

Resultado: Siempre se accede al banco 0 en esta región de memoria, sin importar el valor del registro de banco. Es por ello que esta región es "fija". Con la región de RAM 0xA000–0xBFFF pasa exactamente lo mismo, debido a que el segundo bit más alto de la dirección 0xA000 es 0.

3.5 Interrupciones

Las interrupciones son mecanismos que permiten a la CPU pausar temporalmente la ejecución del programa principal para saltar a una posición específica de memoria, conocida como vector de interrupción. Este proceso resulta útil porque puede ocurrir en cualquier parte del código, lo que facilita la respuesta a eventos externos en tiempo real.

En la mayoría de las CPU, existe un *master flag* para controlar las interrupciones, y la CPU Z80 de la Game Boy no es una excepción. Sin embargo, además de este flag, la Game Boy cuenta con registros específicos para gestionar las interrupciones. El registro maestro es el Interrupt Master Enable (IME), que se desactiva con el opcode DI, evitando que ocurra cualquier interrupción. Por el contrario, el opcode EI activa el IME, permitiendo que se procesen las interrupciones especificadas en el registro Interrupt Enable (IE, ubicado en 0xFFFF). Complementariamente, existe el registro Interrupt Flag (IF, ubicado en 0xFF0F), que marca qué interrupción en particular ha sido activada.

Cuando ocurre una interrupción, se sigue el siguiente procedimiento:

- Al activarse una interrupción, su bit correspondiente en el registro IF se establece en 1.
- Si tanto el IME como el bit correspondiente en el registro IE están activos, se realizan tres pasos adicionales.
- El IME se desactiva, evitando que otra interrupción interrumpa la que ya está en proceso.
- El contador de programa (Program Counter, PC) se guarda en la pila de memoria (stack).
- La CPU salta al vector de la interrupción correspondiente, ejecutando el código específico para manejar ese evento.

Este procedimiento garantiza que las interrupciones se gestionen de forma controlada, evitando conflictos y permitiendo que la CPU responda adecuadamente a eventos críticos en tiempo real.

Cuando se producen varias solicitudes de interrupción de forma simultánea, se otorga prioridad a aquellas con un valor de bit menor. En este esquema de prioridades, el bit 0 (interrupción de VBlank) tiene la mayor prioridad, mientras que el bit 5 (interrupción del Joypad) posee la prioridad más baja.

3.5.1 Tipos

- **VBlank (0x40)**: Como indica el nombre, esta interrupción se lanza cada vez que la PPU entra en el modo 1, también conocido como VBlank.
- **LCD STAT (0x48)**: Hay distintas maneras de que esta interrupción se active, todas ellas relacionadas con los estados del barrido de pantalla.
- **Timer (0x50)**: Se activa a intervalos regulares según la configuración del temporizador (TIMA).
- **Serial (0x58)**: Se utiliza para la comunicación serie, permitiendo que dos consolas intercambien datos a través del cable Link.
- **Joypad (0x60)**: Esta interrupción se activa cuando el jugador presiona un botón. Tiene ciertos inconvenientes, ya que el hardware de la consola tiene el problema del rebote de interruptor, provocando se detecten más de una vez las transiciones de los bits al pulsar un mismo botón.

3.5.2 Registros

3.5.2.1 IME (Interrupt Master Enable)

Se trata de un flag interno de solo escritura (no está enlazado a ninguna dirección de memoria). Controla que cualquier tipo de interrupción se pueda lanzar. Al inicio de ejecución de cualquier juego, está deshabilitada. Se maneja a través de las operaciones:

- **EI:** Habilita las interrupciones (IME = 1). No toma efecto hasta la ejecución de la siguiente instrucción.
- **DI:** Deshabilita las interrupciones (IME = 0).
- **RETI:** Habilita las interrupciones y ejecuta un RET.
- **Cuando una interrupción está siendo procesada:** En estos momentos, para que no se solapen varias interrupciones de manera simultánea, se deshabilita (IME = 0).

3.5.2.2 IE (Interrupt Enable - 0xFFFF)

Registro de 5 bits que controla qué interrupciones **pueden** ser lanzadas:

- **Bit 0:** VBlank.
- **Bit 1:** LCD STAT.
- **Bit 2:** Timer.
- **Bit 3:** Serial.
- **Bit 4:** Joypad.

3.5.2.3 IF (Interrupt Flag - 0xFF0F)

Registro de 5 bits que indica qué interrupciones **deben** ser lanzadas. Los bits se corresponden con los del IE. Una vez la CPU maneja la interrupción, el valor del bit vuelve de manera automática a 0.

3.6 Temporizadores / Timers

Los temporizadores son mecanismos esenciales para la sincronización y el control de eventos en el sistema. Ayudan a la CPU a gestionar y coordinar actividades en el juego, como controlar la velocidad de animaciones, gestionar el conteo de frames y mantener la precisión en la ejecución de instrucciones de tiempo crítico.

3.6.1 DIV (Divisor de reloj - 0xFF04)

Contador general de 16 bits que incrementa a una frecuencia fija de 16384Hz (incrementa cada 256 ciclos de la CPU) y aproximadamente 16779Hz en SGB. Al intentar escribir cualquier valor en el registro, se restablecen los 8 primeros bits a 0x00. En caso de que se ejecute una instrucción STOP, dejará de actualizarse hasta que se reinicie la ejecución. Solamente los 8 bits más altos (8-15) son visibles para el desarrollador, los cuales son únicamente de lectura.

Bit	Incremento por ciclos de máquina	Incremento por ciclos de reloj
0	1	4
1	2	8
2	4	16
3	8	32
4	16	64
5	32	128
6	64	256
7	128	512
8	256	1024
9	512	2048
10	1024	4096
11	2048	8192
12	4096	16384
13	8192	32768
14	16384	65536
15	32768	131072

Tabla 3.11: Incrementos del registro DIV según ciclos de máquina y de reloj

3.6.2 Temporizador de contador (TIMA - 0xFF05)

Temporizador principal que permite ejecutar acciones basadas en su cuenta. Esto es útil para activar eventos en el juego, como cambios de pantalla o actualizaciones de elementos en pantalla. Se incrementa en base a la frecuencia definida por el registro TAC (0xFF07). Cuando su valor supera 0xFF, se restablece al valor especificado en el registro TMA (0xFF06) y se genera su interrupción.

3.6.3 Temporizador de módulo (TMA - 0xFF06)

Este registro almacena el valor al cual se restablece el temporizador TIMA cuando se produce un desbordamiento. Cuando TIMA llega a 0xFF y genera una interrupción, este toma el valor de TMA para continuar con el conteo.

3.6.4 Controlador de temporizador (TAC - 0xFF07)

Su bit 2 controla que el contador TIMA pueda ser incrementado (0 = Pausado, 1 = Habilitado). Los bits del 3 al 7 no tienen utilidad. Por último, con los bits 0 y 1 se puede controlar la frecuencia a la que el contador avanza:

Selección de Reloj	Incremento	DMG, SGB2, CGB (velocidad simple)	SGB1	CGB (velocidad doble)
00	256 M-cycles	4096Hz	~4194Hz	8192Hz
01	4 M-cycles	262144Hz	~268400Hz	524288Hz
10	16 M-cycles	65536Hz	~67110Hz	131072Hz
11	64 M-cycles	16384Hz	~16780Hz	32768Hz

Tabla 3.12: Selección de modos de reloj y sus frecuencias.

3.6.5 Funcionamiento del Temporizador

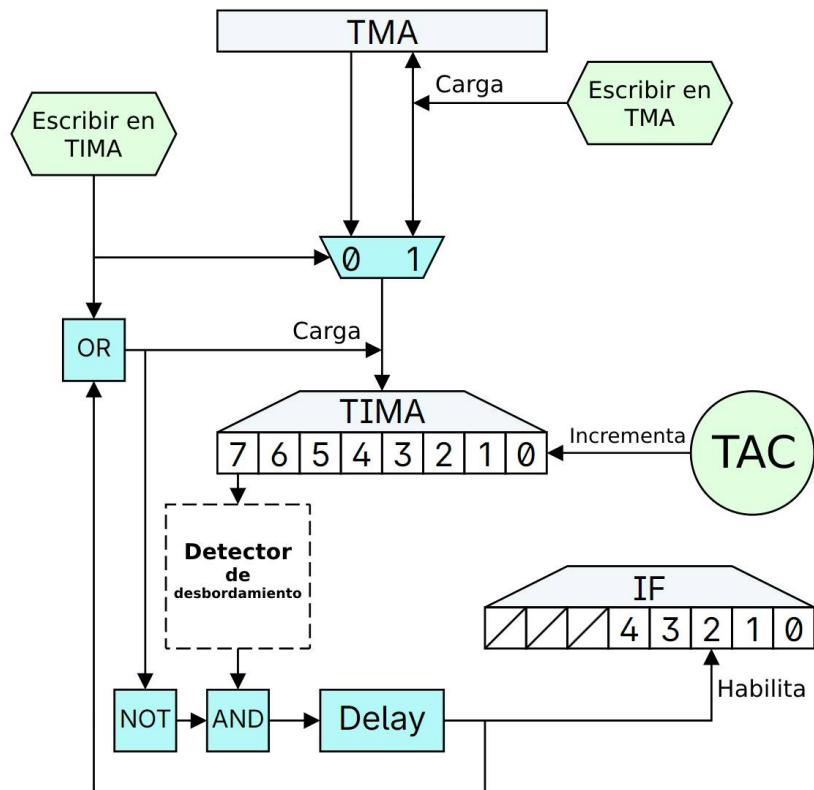


Figura 3.12: Diagrama - Interacción entre TMA y TIMA (?)

A continuación se explica el comportamiento del diagrama:

1. El temporizador TIMA se incrementa a la frecuencia especificada por el registro TAC. Cuando TIMA supera el valor 0xFF, se produce un desbordamiento, que es detectado por el sistema.
2. Al ocurrir un desbordamiento, se activa la interrupción correspondiente en el registro IF, y el valor de TIMA se restablece al valor presente en el registro TMA durante la señal de carga.
3. Si se escribe un valor en TIMA, este nuevo valor sobrescribe el actual y, además, impide que se produzca un desbordamiento en el mismo ciclo, como lo indica la operación NOT en la puerta AND.
4. Si se escribe un valor en TMA, TIMA tomará ese valor en el momento de un desbordamiento.
5. Entre TMA y TIMA se encuentra un multiplexor que controla qué valor se carga en el registro TIMA. Este multiplexor se activa al realizar una escritura en TIMA, y su funcionamiento es el siguiente:

- **0:** Carga el contenido de TMA en TIMA en caso de que se produzca un desbordamiento.
- **1:** Permite escribir un valor directamente en TIMA desde el bus de datos de la CPU.

3.7 I/O - Entrada / Salida

Los registros pertenecientes al rango de I/O (0xFF00-0xFF70) son los siguientes:

Inicio	Final	Primera aparición	Propósito
0xFF00		DMG	Joypad
0xFF01	0xFF02	DMG	Transferencia serie
0xFF04	0xFF07	DMG	Temporizador y divisor
0xFF0F		DMG	Interrupciones
0xFF10	0xFF26	DMG	Audio
0xFF30	0xFF3F	DMG	Patrón de onda
0xFF40	0xFF4B	DMG	Control LCD, estado, posición, desplazamiento y paletas
0xFF4F		CGB	Selección de banco de VRAM
0xFF50		DMG	Activar/Desactivar el boot de ROM
0xFF51	0xFF55	CGB	VRAM DMA
0xFF68	0xFF6B	CGB	Paletas de BG / OBJ
0xFF70		CGB	Selección de banco de WRAM

Tabla 3.13: Registros de I/O mapeados en memoria de la Game Boy.

3.7.1 Joypad: 0xFF00

Para el control de los botones en la Game Boy, se utiliza un único registro mapeado en memoria en la dirección 0xFF00, también conocido como el registro P1. Este registro encapsula el estado de los 8 botones (A, B, Start, Select, Arriba, Abajo, Izquierda, Derecha) en un único byte, estructurado como una matriz. Cada bit representa un botón o conjunto de botones, permitiendo identificar cuál ha sido presionado.

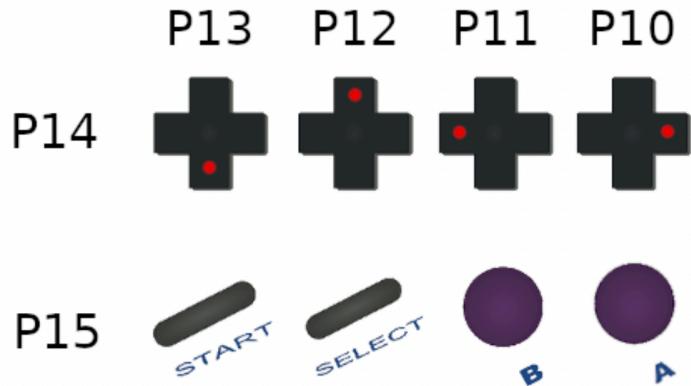


Figura 3.13: Distribución del Joypad (?)

La notación P1N en las especificaciones se refiere a “Player 1” seguido del número del bit correspondiente. Por ejemplo, si se desea consultar el estado de los botones A, B, Start o Select, es necesario configurar el bit 5 en 1 y el bit 4 en 0. Esto selecciona la línea correspondiente a estos botones en la matriz virtual que el registro simula.

Es importante destacar que el registro opera con lógica inversa: un botón presionado se representa con un valor 0, mientras que un botón sin presionar es un 1. Para trabajar de manera más intuitiva con estos valores, se puede realizar una operación complementaria, invirtiendo los bits, lo que facilita el manejo de la información en la lógica habitual (donde 1 indica botón presionado y 0 lo contrario).

3.7.2 Transferencia Serie: 0xFF01-0xFF02

Estos registros tienen relación directa con la comunicación entre dos consolas, la cual se realiza mediante un Cable Link, byte a byte. Se utiliza el protocolo SPI, en la que una de las consolas actua como ”maestro” y genera una señal de reloj interna, que controla el momento exacto en que ocurre el intercambio con el resto de consolas.

Si la consola esclava no ha cargado el siguiente byte de datos cuando comienza la transferencia, el último byte enviado será retransmitido. Por el contrario, si la consola esclava está lista para enviar un nuevo byte pero la transferencia anterior aún no ha concluido, no podrá transmitir hasta que el proceso en curso haya finalizado.



Figura 3.14: Cable Link (?)

3.7.2.1 SB (0xFF01): Transferencia de datos en serie

Antes de que comienza la transferencia de datos, contiene el byte que va a ser enviado.

Durante la transferencia, el registro se modifica en cada ciclo de ejecución. El bit más significativo (bit 7) es enviado, mientras que los bits restantes se desplazan a la izquierda, con el bit 6 convirtiéndose en el nuevo bit 7, siguiendo un proceso similar a un *shift left*. Simultáneamente, el bit menos significativo (bit 0) se reemplaza por el bit 7 del byte que la otra consola está transfiriendo. Este procedimiento se repite durante 8 ciclos hasta completar el intercambio del byte.

3.7.2.2 SC (0xFF02): Control de transmisión de datos

Bit	Descripción
7	Indica si la transferencia de datos está en progreso o no. (0 = transferencia completa, 1 = transferencia en progreso)
6 - 2	Sin uso.
1	Solo funciona en CGB. Habilita una velocidad serial mayor (+256kHz)
0	Selección de reloj. 0 = Reloj externo, 1 = Reloj interno de la consola maestro.

Tabla 3.14: Bits del registro SC (Serial Control) de la Game Boy

Se activa al configurar la transmisión de datos. La consola que funciona como maestro cargará un byte en el registro SB para posteriormente establecer este registro en 0x81 (0b10000001) el cual indica que se ha iniciado una transferencia y se utilizará el reloj interno de la consola maestro.

La otra consola deberá establecer el registro en 0x80 para habilitar el puerto. Al producirse la interrupción del serial cuando la transmisión de datos finalice, establecerá el bit 7 a 0.

3.7.2.2.1 Reloj Interno Las velocidad del reloj interno para DMG es de 8192Hz. Esto nos da una tasa de transferencia de 1KByte por segundo. En el modo CGB, disponemos de cuatro tasas de velocidad, dependiendo del bit 1 en el registro SC, y dependiendo de si se utiliza el modo *Double Speed*.

Frecuencia	Velocidad	Condiciones
8192Hz	1 KB/s	Bit 1 = 0, Velocidad normal.
16384Hz	2 KB/s	Bit 1 = 0, Velocidad doble.
262144Hz	32KB/s	Bit 1 = 1, Velocidad normal
524288Hz	64KB/s	Bit 1 = 1, Velocidad doble

Tabla 3.15: Frecuencias del Reloj Interno.

3.7.2.2.2 Reloj Externo El reloj externo de forma general lo aporta la otra consola, pero si se conectara la consola a un ordenador, la frecuencia podría ser cualquiera. Hay indicios de que el modelo DMG llega a reconocer frecuencias de 500 kHz. Los pulsos de reloj no necesitan ser regulares por ambos lados, ya que la consola va a esperar pacientemente hasta recibir el siguiente bit de información.

3.8 PPU

La GPU o comunmente conocida como PPU en la Game Boy es el componente encargado de gestionar la información que se muestra por pantalla. Maneja la representación de tiles para sprites y fondos, utilizando para ello una paleta de 4 colores. La resolución de la pantalla es de 160x144 píxeles, con una tasa de refresco de 59.7Hz. Para gestionar el dibujo de cada pixel, la PPU opera en diferentes modos, conocidos como *H-Blank* (Modo 0), *V-Blank* (Modo 1), *OAM Scan* (Modo 2) y *Drawing Pixels* (Modo 3, también conocido como *H-Draw*).

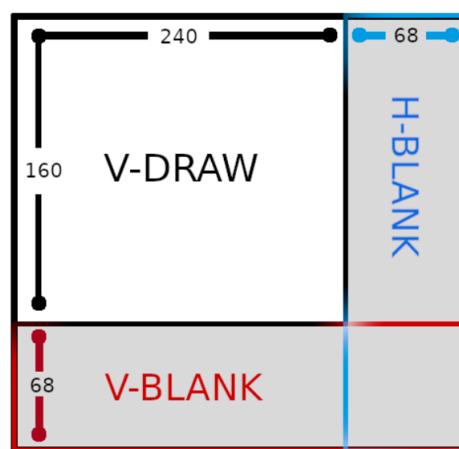


Figura 3.15: Modos internos de la PPU (?)

3.8.1 Tiles

Los tiles son la base de la representación gráfica en la Game Boy. Cada tile tiene un tamaño de 8x8 píxeles (16 bytes en total, 2 bytes por línea) y puede ser representado por un máximo de 4 colores. De esta forma, se evita procesar los píxeles de forma individual, lo que reduce la carga de trabajo de la CPU y permite una representación más eficiente de los gráficos.

El color de cada pixel se obtiene de un índice de 2 bits. Este índice se utiliza para acceder a la paleta de colores, que contiene un total de 4 colores. A este formato se le denomina comúnmente **2BPP**, el cual hace referencia a "2 bits por pixel". La paleta de colores es compartida por todos los tiles, lo que permite una mayor eficiencia en el uso de la memoria.

Para obtener el índice correcto para cada pixel, se deben leer de forma simultánea 2 líneas distintas de la memoria de tiles. La primera línea contiene los bits menos significativos, mientras que la segunda línea contiene los bits más significativos.

Tomando como ejemplo el siguiente tile: `$3C $7E $42 $42 $42 $42 $42 $42 $7E $5E $7E
$0A $7C $56 $38 $7C`

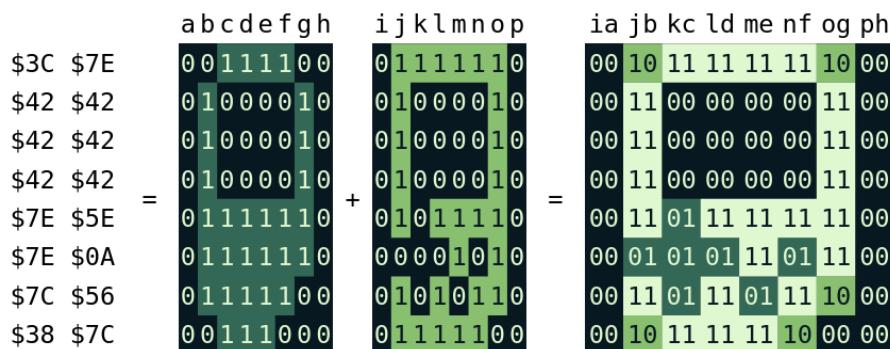


Figura 3.16: Obtención de colores para un tile (?)

3.8.2 Capas

La PPU de la Game Boy tiene **tres capas** diferentes: el **Background** o Fondo principal, el **Window** o HUD, y los **sprites**.

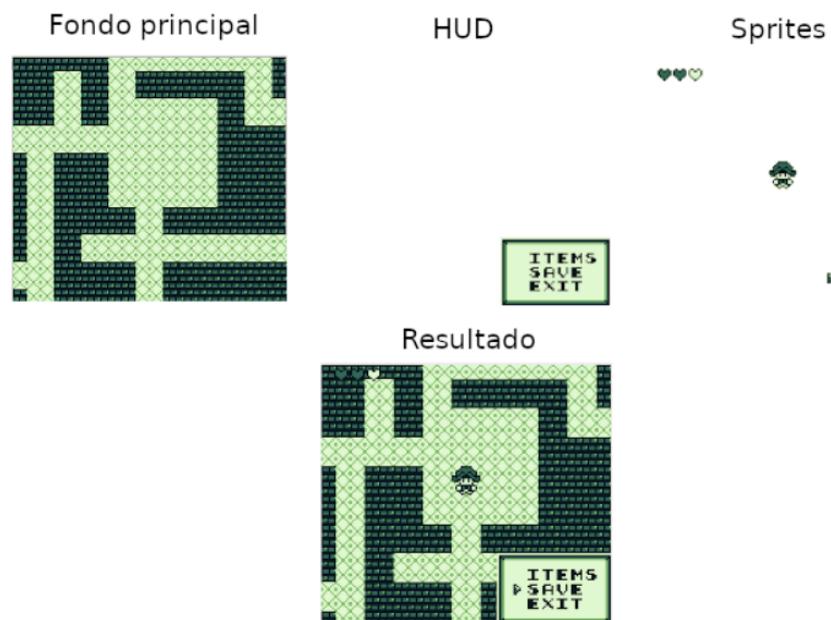


Figura 3.17: Capas de la PPU (?)

3.8.2.1 Background

La pantalla principal es la que sirve de fondo, es decir, cualquier cosa que no vaya a necesitar actualizarse de forma constante.

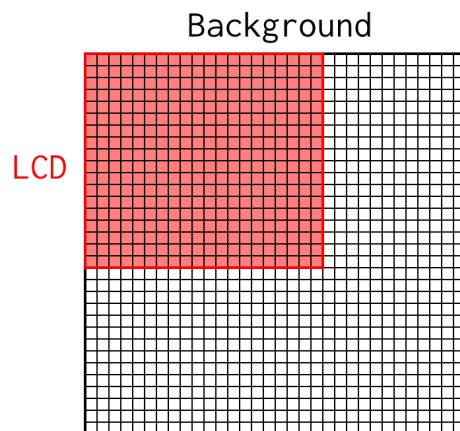


Figura 3.18: Capa de fondo de la PPU (?)

Está compuesto por 32x32 tiles (256x256 pixeles). De esos tiles totales, solamente una sección de 20x18 tiles (160x144 pixeles) puede ser renderizada en pantalla (*LCD* en la figura previa). Esa sección se puede mover utilizando los registros SCX y SCY, las cuales establecen un offset de la posición inicial (esquina superior izquierda).

3.8.2.1.1 SCY, SCX: 0xFF42 - 0xFF43: Estos registros definen un desplazamiento (*offset*) para la posición inicial de la pantalla. Los valores pueden superar los límites inferior y derecho, pero no el superior ni el izquierdo, ya que los valores no pueden salir del rango 0-255. Si la pantalla LCD excede estos límites, la imagen se repetirá en bucle desde la posición inicial.

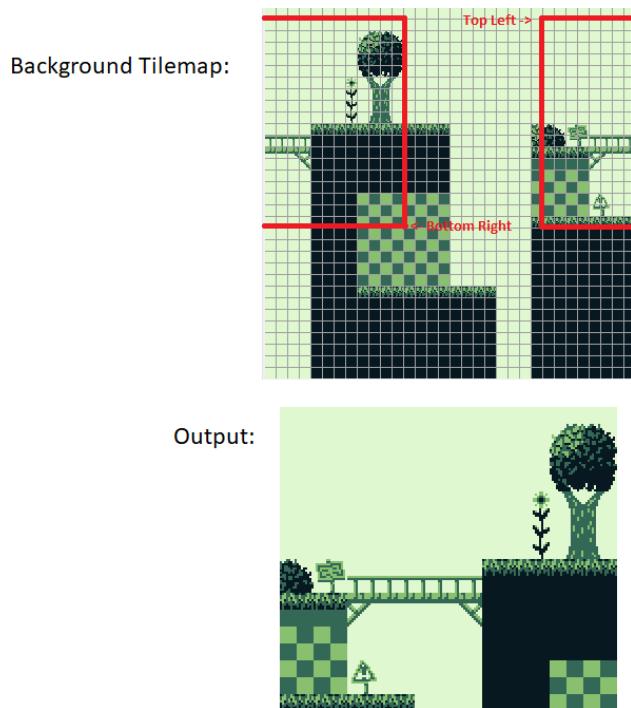


Figura 3.19: Diagrama de scroll del LCD (?)

3.8.2.2 Window

Inicialmente, la ventana (Window) será invisible ya que no tendrá ninguna información a mostrar. En los videojuegos, se utiliza comúnmente para mostrar el HUD (barras de estado, menús emergentes, etc.).

Su posición en pantalla está determinada por los registros WX y WY, donde WX incluye un desplazamiento de 7 píxeles (equivalente a un tile). Por lo tanto, si WX = 7, WY = 0, SCX = 0 y SCY = 0, la ventana se alinearán exactamente sobre el Background.

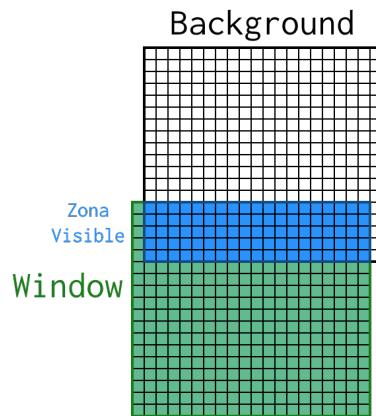


Figura 3.20: Capa de HUD de la PPU (?)

Si se analizan juegos como Pokémon Rojo/Azul, se puede apreciar mejor cómo se gestionan todas las ventanas de diálogos o de inventario.

Por un lado, está la pantalla de Background (Screen 1), donde se carga el tilemap. Por otro lado, la ventana (Screen 2) actúa como una copia de los tiles de fondo que el scroll permite visualizar en la primera. Esto permite generar una escena completamente idéntica, de modo que, al superponer la segunda pantalla sobre la primera, no se percibe ninguna diferencia.

La principal ventaja de este mecanismo es la posibilidad de mostrar ventanas emergentes sin necesidad de modificar el fondo ni añadir sprites, como se ilustra en la siguiente imagen.



Figura 3.21: Dibujado de Ventanas en Pokémon Rojo/Azul (?)

3.8.2.2.1 WY, WX: 0xFF4A - 0xFF4B : Especifican el offset de la ventana respecto de la esquina superior izquierda. Los valores que pueden tomar están dentro de los siguientes

rangos: 0-166 para WX y 0-143 para WY. Mencionado previamente, WX tiene un offset de 7 pixeles.

Hay bugs conocidos con los siguientes valores: si se establece en 0, la ventana tendrá un efecto de *temblor* horizontal cuando se modifique el registro SCX. Por otro lado, si se establece en 166, la ventana se dibujará en la siguiente línea del scanline.

3.8.2.3 Sprites

Los sprites de manera general son 8x8 pixeles (2x2 tiles) que no están limitados y se mueven de forma independiente del Background y el Window. Se pueden definir también como 8x16 pixeles (2x4 tiles). La información de ellos, como se ha explicado previamente, se almacena en la memoria OAM.

3.8.3 Registros

A continuación se detallan los registros principales que ajustan el comportamiento y proporcionan información importante de la PPU:

3.8.3.1 Control de LCD: 0xFF40

El registro principal que controla el LCD. Permite decidir qué elementos se muestran o no por pantalla.

- **Bit 7:** Habilita tanto la PPU como la pantalla LCD. Si se deshabilita, la pantalla se apaga y la PPU no renderiza nada. Deshabilitar la pantalla se debe hacer solamente en modo V-Blank, de otra forma puede estropear el hardware real.
- **Bit 6:** Selecciona qué tilemap va a utilizar el Window. Si está en 0, se utiliza el tilemap 0x9800-0x9BFF. Si está en 1, se utiliza el tilemap 0x9C00-0x9FFF.
- **Bit 5:** Habilita el Window. Si está en 0, el Window no se muestra. Si está en 1, el Window se muestra.
- **Bit 4:** Selecciona qué tileset va a utilizar el Background y el Window. Si está en 0, se utiliza el tileset 0x8800-0x97FF. Si está en 1, se utiliza el tileset 0x8000-0x8FFF.
- **Bit 3:** Selecciona qué tilemap va a utilizar el Background. Si está en 0, se utiliza el tilemap 0x9800-0x9BFF. Si está en 1, se utiliza el tilemap 0x9C00-0x9FFF.
- **Bit 2:** Selecciona el tamaño de los sprites. Si está en 0, los sprites son de 8x8 pixeles. Si está en 1, los sprites son de 8x16 pixeles.
- **Bit 1:** Habilita los sprites. Si está en 0, los sprites no se muestran. Si está en 1, los sprites se muestran.
- **Bit 0:** Habilita el Background y el Window. Si está en 0, el Background y el Window no se muestran. Si está en 1, el Background y el Window se muestran. En modo CGB, este bit se utiliza para habilitar el modo de prioridad de sprites.

3.8.3.2 Coordenada Y del LCD: 0xFF44

Nombrado generalmente como *LY*, indica la línea actual de renderizado o *scanline* que va a ser renderizada. Puede contener cualquier valor en el rango [0-153], donde los valores en el rango [144-153] forman parte del período de V-Blank.

3.8.3.3 Comparación de LY: 0xFF45

Internamente la consola compara constantemente el valor de LY con el de este registro, conocido de forma general como LYC. Cuando ambos registros coinciden (y si está habilitado), la interrupción de STAT es solicitada.

3.8.3.4 Estado del LCD: 0xFF41

Habilita ciertas interrupciones y ofrece información acerca del estado actual de la pantalla.

- **Bit 7:** No es utilizado.
- **Bit 6:** Si se habilita, se producirá una interrupción STAT si los valores de LY y LYC son iguales.
- **Bit 5:** Si se habilita, se producirá una interrupción STAT cuando el modo OAM de comienzo.
- **Bit 4:** Si se habilita, se producirá una interrupción STAT cuando el modo V-Blank de comienzo.
- **Bit 3:** Si se habilita, se producirá una interrupción STAT cuando el modo H-Blank de comienzo.
- **Bit 2:** Su valor es 1 en caso de que el valor de LYC sea idéntico al de LY.
- **Bit 1-0:** Indica el modo actual de la PPU: 00 = HBlank, 01 = V-Blank, 10 = OAM Scan, 11 = Drawing Pixels.

3.8.4 Modos

Los modos de PPU definen cómo se dibujan los píxeles en pantalla. Cada uno de ellos se activa en un momento específico del ciclo de renderizado, permitiendo que la PPU realice las operaciones necesarias para mostrar la información en pantalla.

Como se muestra en la figura, el renderizado completo de la pantalla tarda 70,224 ciclos de reloj, lo que equivale a 17,556 ciclos de máquina. Dado que la CPU de la Game Boy opera a 4.194304 MHz (4,194,304 ciclos de reloj por segundo), podemos calcular la tasa de fotogramas por segundo (FPS) de la siguiente manera:

$$\text{FPS} = \frac{4,194,304 \text{ ciclos por segundo}}{70,224 \text{ ciclos por frame}} = 59.8 \approx 60$$

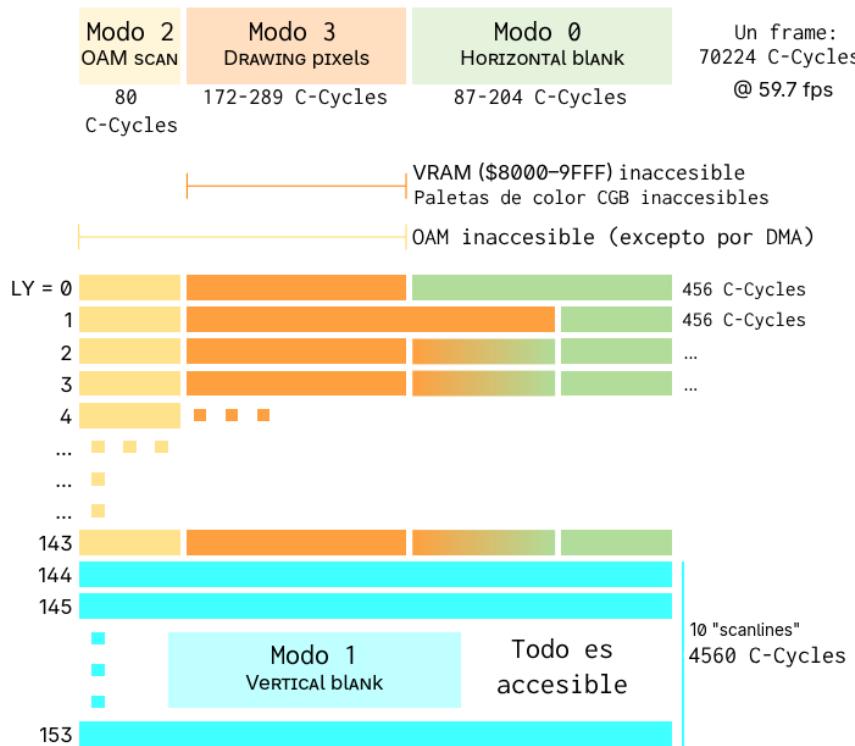


Figura 3.22: Proceso de renderizado de la PPU (?)

3.8.4.1 Modo 0: H-Blank

Ocupa el tiempo restante de la línea de renderizado después de que se hayan dibujado los píxeles. En este modo, la PPU no realiza ninguna operación, permitiendo que la CPU acceda a la memoria de vídeo sin interferencias.

Si el valor de LY alcanza 144 en la siguiente línea de renderizado, se generará automáticamente la interrupción de V-Blank (si está habilitada por el desarrollador).

3.8.4.2 Modo 1: V-Blank

Similar a H-Blank en el sentido de que la PPU no dibuja píxeles en pantalla. Sin embargo, es un período de tiempo más largo (4560 frente a un máximo de 204 ciclos de reloj).

De forma similar al final de la última línea (y si está habilitado), se producirá la interrupción que indica que el proceso de OAM da comienzo.

3.8.4.3 Modo 2: OAM Scan

Se activa al inicio de cada línea de renderizado (excepto durante el modo V-Blank), justo antes de que los píxeles comiencen a dibujarse en pantalla. Durante este modo, la PPU busca sprites en la memoria OAM que deban ser renderizados en la línea actual, guardándolos

posteriormente en un buffer.

Para decidir qué sprites se guardan en buffer se toman en cuenta las siguientes condiciones:

- La coordenada X del sprite debe ser mayor de 0.
- LY + 16 debe ser mayor o igual que la coordenada Y del sprite.
- LY + 16 debe ser menor o igual que la coordenada Y del sprite más su altura (8 o 16, según se haya configurado en el bit 2 de 0xFF40).
- La cantidad de sprites almacenados en buffer no puede ser mayor de 10.

3.8.4.4 Modo 3: Drawing Pixels

Modo durante el cual la PPU transfiere los colores correspondientes de cada píxel a la pantalla LCD. La duración varía según el número de sprites en la línea de renderizado actual, si se deben renderizar tiles de Window, etc.

3.8.5 Pixel FIFO

La Game Boy renderiza los píxeles de forma individual y los envía uno por uno a la pantalla LCD. Para lograrlo, utiliza dos Pixel FIFOs.

- **Background FIFO:** Almacena los píxeles del Background y del Window.
- **Sprite FIFO:** Almacena los píxeles de los sprites.

Estos dos FIFOs son independientes y no comparten información entre ellos. El único momento en el que ambos se utilizan de forma simultánea es al mezclar los píxeles extraídos para ser enviados a la pantalla LCD. Cada uno puede almacenar hasta 16 píxeles, y el proceso debe garantizar que ambos contengan siempre al menos 8 píxeles en cualquier momento.

La manipulación de los dos FIFOs solamente ocurre durante el modo 3.

En cada FIFO, los píxeles van acompañados de una serie de atributos o propiedades:

- **Color:** Índice de color del píxel.
 - **Paleta:** Paleta de colores a la que pertenece el píxel.
 - **Prioridad de Sprite:** Solo existe en modo CGB, indicando el índice OAM del sprite.
 - **Prioridad de Fondo:** Indica si el píxel es de fondo o de sprite.
-

3.8.5.1 Pixel FIFO Fetcher

El proceso de *Pixel Fetching* consta de 5 pasos:

1. Obtención de Tile
2. Obtención de datos bajos del Tile
3. Obtención de datos altos del Tile
4. Espera (Sleep) de 2 ciclos de reloj
5. Push

3.8.5.1.1 Obtención de Tile: Determina qué tile de Background o Window va a ser utilizado para obtener los píxeles. Por defecto el tilemap utilizado es el 0x9800. Las siguientes condiciones se deben tener en cuenta:

- Si el bit 3 del LCDC está habilitado y la coordenada X de la línea de renderizado actual no está dentro del Window, entonces el tilemap 0x9C00 es utilizado.
- Si el bit 6 del LCDC está habilitado y la coordenada X de la línea de renderizado actual está dentro del Window, entonces el tilemap 0x9C00 es utilizado.

El fetcher hace un seguimiento constante de las coordenadas X e Y del tile sobre el que está operando:

- **Coordenada X:**
 - Si el tile actual pertenece al Window, se usa directamente su coordenada X.
 - En caso contrario, se utiliza la siguiente fórmula:

$$((SCX/8) + \text{fetcherX})\&0x1F$$

- **Coordenada Y:**
 - Si el tile actual pertenece al Window, se usa directamente su coordenada Y.
 - En caso contrario, se utiliza la siguiente fórmula:

$$(LY + SCY)\&255$$

3.8.5.1.2 Obtención de datos bajos del Tile: Se verifica el bit 4 del LCDC para determinar qué tilemap se debe utilizar. En modo CGB, se debe determinar también el banco correspondiente de VRAM y si el tile esta volteado verticalmente.

Utilizando el numero de Tile obtenido en la operacion anterior, el fetcher obtiene el primer byte de la informacion en VRAM y la almacena. El offset a utilizar dependerá de si es de Background ($2 * ((LY + SCY)\%8)$) o de Window ($2 * ((LY - WY)\%8)$).

3.8.5.1.3 Obtención de datos altos del Tile: Este paso es el mismo que el anterior, con la diferencia de que se lee el siguiente byte de la dirección previamente recuperada.

3.8.5.1.4 Sleep: La consola espera durante dos ciclos de reloj sin realizar ninguna operación.

3.8.5.1.5 Push: Los datos de los 8 píxeles pertenecientes al tile obtenido previamente se decodifican y se cargan en el FIFO correspondiente.

Si en los atributos se indica que el tile está volteado horizontalmente, los píxeles se insertarán de LSB a MSB.

3.8.5.2 Administrar Píxeles a la Pantalla LCD

Cada ciclo de reloj, la PPU intenta enviar un píxel al LCD. Este proceso solo se inicia si el FIFO de Background contiene píxeles. Los píxeles del FIFO de sprites sirven para ser mezclados con los demás.

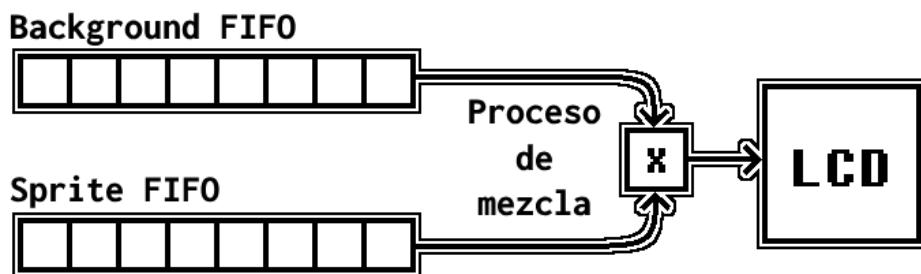


Figura 3.23: Proceso de mezcla de los dos FIFOs (?)

Para establecer qué pixel utilizar, se ejecutan las siguientes comprobaciones:

1. Si no hay pixel de sprite o el índice de color es 0 (invisible), se mantiene el pixel de fondo.
2. Si el índice de color del pixel de fondo no es 0, y la prioridad del pixel de sprite es 1, se mantiene el pixel de fondo.
3. En cualquier otro caso, se mantiene el pixel de sprite.

4 Metodología y Planificación

Este capítulo presenta la metodología seguida para el desarrollo del proyecto, así como la planificación temporal y técnica. En primer lugar, se expone el enfoque metodológico adoptado, justificando su idoneidad según el tipo de proyecto y sus objetivos. A continuación, se detallan las distintas etapas en las que se ha estructurado el desarrollo, desde la fase inicial de planificación hasta los resultados y conclusiones.

También se hace referencia al concepto de Mínimo Producto Viable, fundamental para establecer una primera versión funcional del emulador que pueda ejecutarse y evaluarse con éxito. Por último, se describen las herramientas empleadas a lo largo del proyecto para el desarrollo de la aplicación.

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 Etapas 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.

En la siguiente tabla se puede ver la planificación estimada con la que se pretende presentar y defender el proyecto a inicios de Junio de 2024 (C3). No se tienen en cuenta posibles retrasos por enfermedades, viajes u otros

Apartado	Tiempo estimado	Fecha límite
Agradecimientos, Objetivos y Justificación	1 semana	25 Agosto
Planificación y Metodología	2 semanas	8 Septiembre
Diseño	4 semanas	6 Octubre
Aprendizaje	1 semana	13 Octubre
Marco teórico	3 semanas	3 Noviembre
Desarrollo	5 semanas	8 Diciembre
Pruebas y validación	2 semanas	22 Diciembre
Resultados	1 semana	29 Diciembre
Conclusiones y trabajo futuro	1 semana	5 Enero
Bibliografía y Referencias	1 semana	12 Enero

Tabla 4.1: Planificación de contenidos y fechas límite

Entrando en detalle en algunas etapas del proyecto:

- **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.

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.

4.4 Herramientas utilizadas

El desarrollo de la aplicación se apoyará del uso de las siguientes tecnologías:

- **Android Studio:** Se utilizará como editor de código principal, permitiendo la creación, prueba y depuración de la aplicación mediante emuladores de dispositivos Android.
- **Visual Studio Code:** Herramienta clave para la redacción y visualización de documentos en LaTeX, además de proporcionar un entorno gráfico para gestionar el control de versiones mediante Git.
- **Photoshop:** Herramienta principal de diseño utilizada para la creación de la interfaz gráfica de la aplicación.
- **Procreate:** Aplicación de diseño gráfico que se empleará para la creación de ilustraciones y recursos visuales específicos de la interfaz, complementando a Photoshop en tareas artísticas y detalladas.



Figura 4.1: Herramientas utilizadas en el proyecto

5 Revisión del Mercado de Aplicaciones

Antes de abordar el desarrollo del emulador, resulta fundamental analizar el panorama actual de aplicaciones similares disponibles en el mercado. Este capítulo tiene como objetivo estudiar y comparar distintas soluciones existentes que permiten emular juegos de Game Boy en dispositivos Android, identificando sus características, puntos fuertes y limitaciones.

La revisión se centra en emuladores reconocidos como My OldBoy! y GBCC, prestando especial atención a aspectos como el rendimiento, la interfaz de usuario, las opciones de configuración y el sistema de monetización.

Este análisis no solo proporciona una visión general del estado de la tecnología en este ámbito, sino que también permite identificar oportunidades de mejora y justificar las decisiones de diseño adoptadas en el presente proyecto.

5.1 Referencias

En esta sección se encuentran aquellas aplicaciones que, tras una revisión de los mismos, se cree que han sido de utilidad por su parecido a la hora de desarrollar este proyecto:

5.1.0.1 My OldBoy!

My OldBoy! es un emulador de Game Boy y Game Boy Color diseñado específicamente para dispositivos Android, que destaca por su precisión en la emulación de casi todos los aspectos del hardware original. Además de simular las consolas, soporta características especiales como el uso del cable link, la vibración y el sensor de inclinación.

El emulador es altamente valorado por su interfaz intuitiva y las múltiples opciones de personalización, que incluyen la posibilidad de añadir colores a juegos monocromáticos y modificar la disposición y el tamaño de los controles en pantalla. Entre las ventajas que ofrece, están la alta compatibilidad con juegos y la capacidad de ajustar la velocidad del juego, tanto para avanzar rápidamente como para ralentizarlo en momentos difíciles.

Sin embargo, tiene algunas limitaciones. Por ejemplo, puede generar archivos de guardado duplicados, carece de un sistema de autoguardado periódico y no ha recibido actualizaciones recientes para las versiones más nuevas de Android, lo que puede afectar su compatibilidad en dispositivos más modernos.



Figura 5.1: My OldBoy! - Logotipo.

La pantalla principal de la aplicación presenta un explorador de archivos que muestra, en formato de lista, los documentos almacenados en la carpeta especificada de la memoria interna del dispositivo. Los archivos se organizan automáticamente por orden alfabético, ofreciendo una visualización clara y estructurada para facilitar su navegación y selección. Dispone de dos botones, uno para recargar la carpeta actual y otro para acceder a los ajustes de sistema.

En los ajustes, podremos modificar distintos aspectos agrupados en las categorías de video, audio, datos de entrada, disposición, varios y avanzado. En vídeo, por ejemplo, podremos modificar el tamaño de la pantalla de juego, la orientación predeterminada de la aplicación, o la paleta de colores.

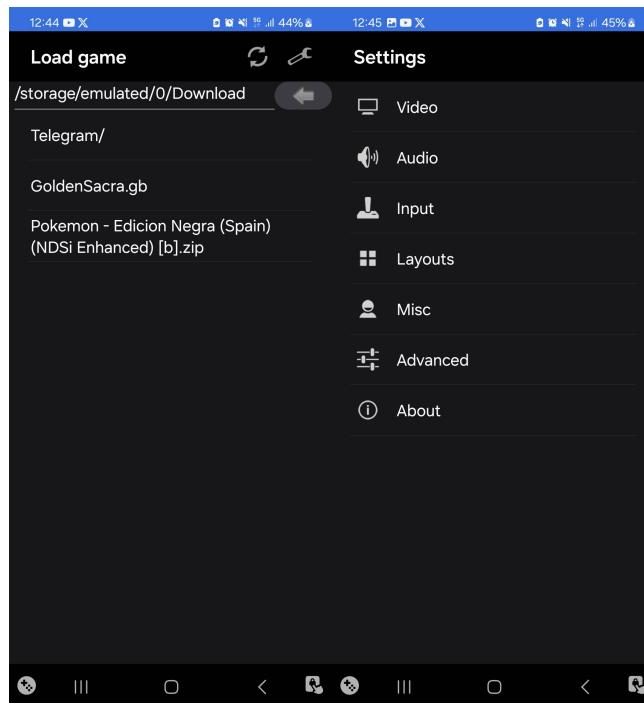


Figura 5.2: My OldBoy! - Menú de inicio y pantalla de ajustes.

La pantalla de juego presenta un diseño más simplificado en comparación con otros emuladores destacados. Su principal ventaja radica en la facilidad para ajustar la posición y tamaño de los botones, permitiendo al usuario personalizar su experiencia de juego. Sin embargo, algunos usuarios consideran que, dado que se trata de una aplicación de pago, el aspecto gráfico podría beneficiarse de mejoras adicionales para estar a la altura de sus competidores.

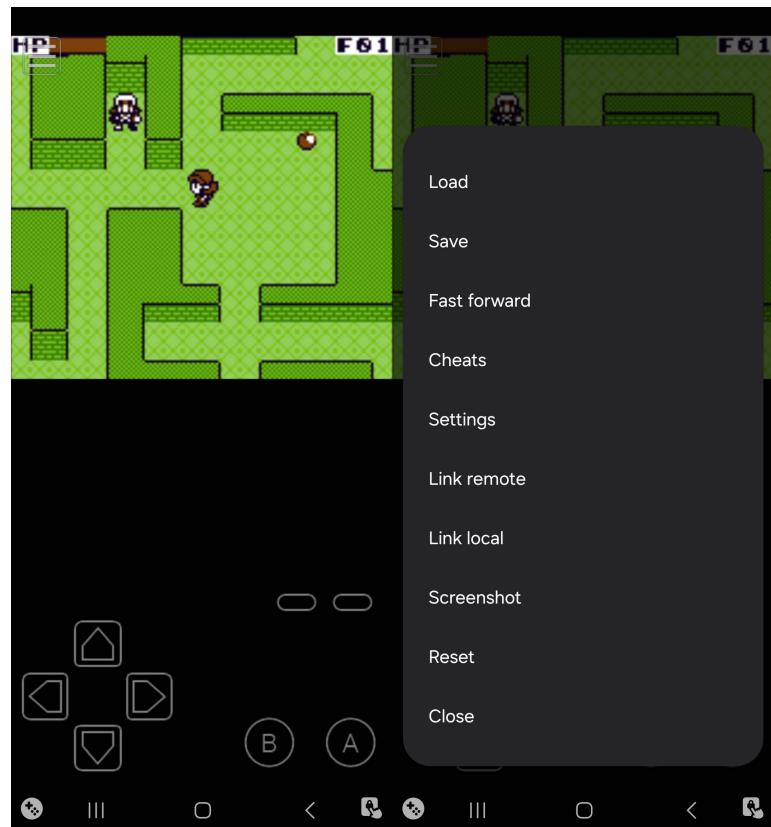


Figura 5.3: My OldBoy! - Pantalla y opciones de juego.



Figura 5.4: My OldBoy! - Pantalla de juego en orientación horizontal.

5.1.0.2 GBCC

GBCC es un emulador multiplataforma de Game Boy y Game Boy Color desarrollado en lenguaje C. Está disponible en Linux, Windows, Android y macOS.

Entre sus funcionalidades se incluyen soporte para GUIs basadas en SDL y GTK, paletas personalizadas para juegos DMG, estados de guardado automáticos, y una función de turbo ajustable. También soporta shaders como Dot-Matrix y Subpixel, los cuales permiten una reproducción precisa de los colores en CGB. Además, ofrece integración con hardware como el Rumble Pak, acelerómetros y la Game Boy Printer, aunque algunos periféricos como el IR Port y el Pocket Sonar no están implementados.

También soporta diversos controladores de memoria: MBC1, MBC2, MBC3 y MBC5, pero aún carece de soporte para MBC6 y MBC7, entre otros. Además, el emulador tiene la capacidad de simular el cable link, aunque es una conexión *fake* consigo mismo.

El proyecto se destaca por su compatibilidad con shaders y la posibilidad de visualizar los datos de VRAM en una ventana separada, lo cual es útil para desarrolladores y aficionados que quieran inspeccionar los gráficos en detalle.



Figura 5.5: GBCC - Logotipo.

La pantalla de juego se caracteriza por su diseño minimalista y adaptable. Dependiendo de si se carga una ROM de Game Boy (DMG) o de Game Boy Color (CGB), la interfaz ajusta su apariencia para emular de manera más fiel las características visuales de la consola correspondiente. El diseño de los controles también se adapta dinámicamente a las dimensiones del dispositivo: en modo vertical, la separación entre los botones y la escala de la pantalla gráfica varían según el ancho y el alto disponibles. En el modo horizontal, el diseño se ajusta automáticamente, manteniendo la proporción y los colores, sin perder la coherencia visual. Además, se pueden observar características adicionales como el botón de turbo y la aplicación de shaders como el Dot-Matrix para mejorar la experiencia visual.

Por último, GBCC permite personalizar el fondo de pantalla que simula la consola original, brindando la opción de cambiar el color según las preferencias del usuario, lo que añade un toque personal a la experiencia de emulación. Esta característica es muy apreciada por quienes buscan una mayor personalización en la interfaz.



Figura 5.6: GBCC - Pantalla de juego.



Figura 5.7: GBCC - Pantalla de juego en orientación horizontal.

5.1.0.3 iGBA

El emulador iGBA fue una aplicación diseñada para dispositivos iOS, disponible brevemente en la App Store antes de ser retirada en abril de 2024. Permitía a los usuarios emular juegos de Game Boy, Game Boy Color y Game Boy Advance en sus iPhones, ofreciendo características como guardado de estado, avance rápido y soporte para ROMs legalmente adquiridas.

Entre sus características principales destacaban la capacidad de guardar estados de juego, permitiendo a los usuarios guardar y retomar partidas en cualquier momento, además de la función de avance rápido (igual que GBCC), que permitía saltar rápidamente escenas o partes lentas del juego. Sin embargo, un detalle a mejorar era que esta velocidad del avance era excesivamente rápida y no permitía ajustes más finos.



Figura 5.8: iGBA - Logotipo.

La interfaz era sencilla, con un diseño limpio que simulaba los botones de las consolas originales. Sin embargo, generó controversia por dos razones: incluir anuncios que interrumpían la experiencia de juego y ser una copia no autorizada de GBA4iOS, un emulador desarrollado por *Riley Testut*. Este conflicto con el código de GBA4iOS, junto con la preocupación por la privacidad y la recopilación de datos de los usuarios, llevó a críticas dentro de la comunidad de emuladores.

Los controles virtuales incluían botones de hombro para los juegos de Game Boy Advance, aunque algunos usuarios mencionaron que estos botones eran algo pequeños. Además, ofrecía un menú de opciones en la esquina inferior izquierda con funciones como guardado rápido, carga de estados, códigos de trucos y avance rápido.

A pesar de sus problemas, algunos usuarios apreciaron lo fácil que era configurarlo y su capacidad para emular juegos retro sin retrasos significativos.

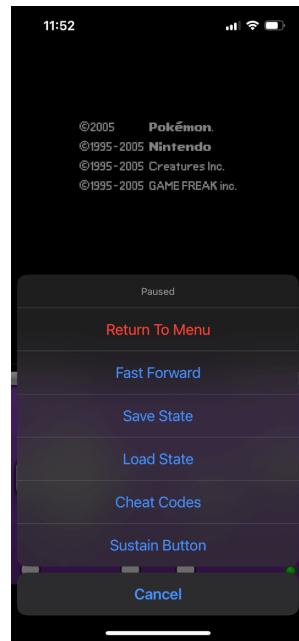


Figura 5.9: iGBA - Ajustes de juego.



Figura 5.10: iGBA - Pantalla de juego.

6 Análisis y Diseño

Este capítulo recoge el proceso de análisis y diseño previo al desarrollo del emulador, donde se definen los requisitos que debe cumplir la aplicación y se plantean las primeras decisiones estructurales y visuales del proyecto.

En primer lugar, se detallan los requisitos funcionales y no funcionales, los cuales especifican tanto las funcionalidades que debe ofrecer la aplicación como las restricciones de calidad, rendimiento o usabilidad que debe respetar. A continuación, se presentan los casos de uso, acompañados de su correspondiente diagrama, para ilustrar de manera clara la interacción entre el usuario y el sistema.

Finalmente, se incluyen una serie de mockups que representan visualmente la interfaz de usuario en sus distintas pantallas, ayudando a anticipar la experiencia del usuario final y facilitando una primera validación del diseño.

6.1 Requerimientos

A continuación se detallan los requerimientos funcionales y no funcionales que rigen el diseño y la implementación del proyecto. Estos requerimientos han sido definidos a partir del análisis de las necesidades del usuario final y de las restricciones técnicas:

6.1.1 Requerimientos Funcionales

Estos requerimientos son las capacidades específicas que la aplicación debe cumplir para garantizar que su funcionamiento sea acorde a los objetivos establecidos:

- **Carga de ROMs:** El usuario debe poder seleccionar y cargar archivos .gb o .gbc desde su dispositivo.
- **Emulación:** Debe replicar fielmente el comportamiento del hardware original.
- **Interfaz táctil:** Proveer botones virtuales que simulen los originales.
- **Audio:** Emular el sonido original de la consola.
- **Gestión de estado:** Guardar y cargar partidas.
- **Compatibilidad:** Dar soporte para ROMs de DMG y CGB.
- **Configuraciones:** Permitir al usuario ajustar las características del emulador, como la velocidad y los controles.

6.1.2 Requerimientos No Funcionales

Estos requerimientos se enfocan en aspectos de calidad de la aplicación, como el rendimiento o la usabilidad, con el objetivo de asegurar una experiencia fluida y eficiente.

- **Rendimiento:** La emulación debe ir a la frecuencia original de la consola en la mayoría de los dispositivos Android.
- **Compatibilidad:** Funcionar en dispositivos Android 8.0 o superior.
- **Eficiencia:** Consumo de batería optimizado durante la ejecución.
- **Usabilidad:** La interfaz debe ser intuitiva y accesible para el usuario.
- **Mantenimiento:** Código modular, que sea fácil de escalar y el cual disponga de una buena documentación para facilitar futuras mejoras.

6.2 Casos de Uso

Los casos de uso permiten identificar y documentar cómo los usuarios interactuarán con el sistema, destacando las funcionalidades clave y sus flujos de ejecución.

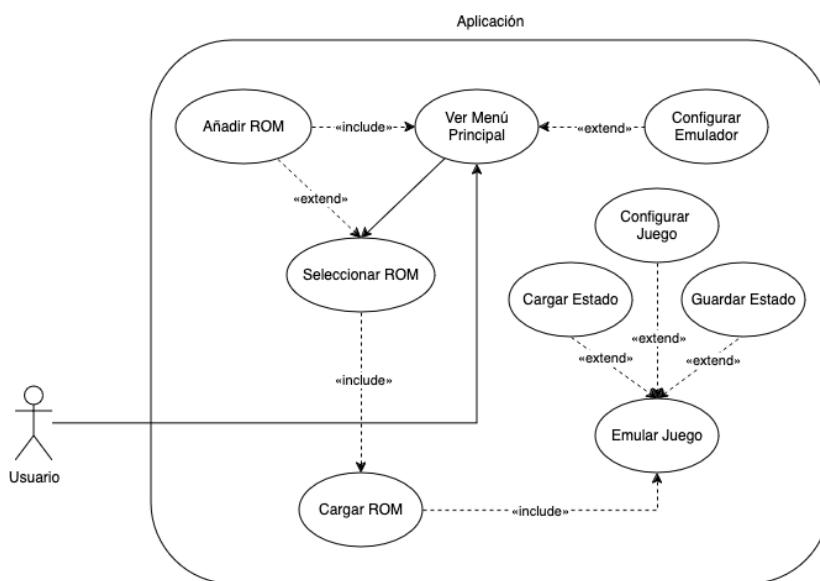


Figura 6.1: Diagrama de Casos de Uso

El usuario tiene acceso directo al menú principal al entrar en la aplicación. Desde ahí, el usuario puede realizar diferentes acciones como añadir o seleccionar una ROM o configurar de forma general el emulador. La selección de una ROM incluye los procesos de carga y emulación. Además, durante la emulación, el usuario puede cargar o guardar el estado y configurar el juego. Las relaciones entre los casos de uso se estructuran con inclusiones para acciones necesarias y extensiones para funcionalidades opcionales.

6.3 Mockup

La aplicación seguirá un diseño minimalista, tomando como referencia principal el emulador GBCC. En su versión inicial, el menú principal presentará una lista en formato de cuadrícula que mostrará todas las ROMs agregadas por el usuario. Cada uno de los elementos de la cuadrícula contará con una imagen predeterminada, representando la consola original, ya sea DMG o CGB, junto con el nombre de la ROM. Estas se ordenarán alfabéticamente para facilitar la navegación. Este enfoque busca mantener una interfaz limpia y funcional, con un acceso directo a los juegos.

En la barra de navegación se mostrarán dos íconos. El primero, representado por un símbolo de suma ("+"), permitirá al usuario agregar nuevas ROMs a la aplicación. El segundo ícono, con la forma de un engranaje, brindará acceso a la configuración general del emulador, que incluirá opciones relacionadas con el video, audio, disposición de controles, entre otros. Estas configuraciones tendrán un impacto en todos los juegos de manera uniforme.

La imagen de cada juego podrá ser personalizada por el usuario a través de un gesto táctil de "hold". Al realizar este gesto sobre una cuadrícula específica, el usuario tendrá la capacidad de acceder a los ajustes individuales del juego, donde podrá reemplazar la imagen predeterminada por una de su elección o modificar el nombre del juego.



Figura 6.2: Menú de inicio.

En cuanto a la pantalla de juego, se busca crear un diseño que evoque la estética de la consola original. Se incluirá un botón que brindará acceso a los ajustes del juego, el cual se ubicará, de manera provisional, en la parte superior izquierda de la pantalla. El layout será

minimalista, permitiendo a los usuarios ajustar el tamaño, color y disposición de los botones según sus preferencias. La imagen de fondo tendrá un color amarillo por defecto, pero los usuarios también podrán personalizarla a su gusto. Además, se ofrecerá la opción de subir imágenes propias para reemplazar los botones o el fondo, lo que facilitará una experiencia de juego más personalizada y adaptada a los gustos individuales.

Aunque no se representa en los mockups, se planea implementar un shader Dot-Matrix, similar al utilizado en GBCC. Esta función podrá desactivarse desde los ajustes generales de la aplicación.

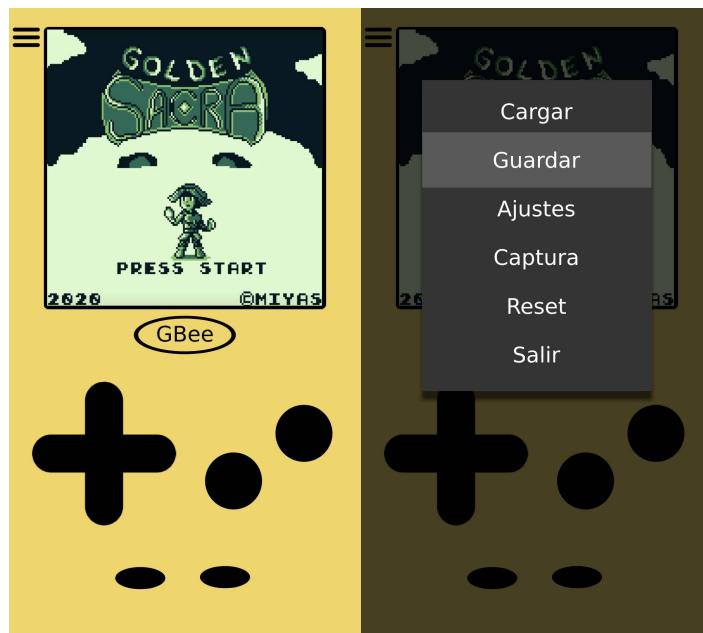


Figura 6.3: Pantalla de juego.

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 = 0xFFE // 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 ↪
3          ↪ -> A
4      val intC = C.toInt() and 0xFF // Conversión de Byte a Integer sin signo ↪
5          ↪ -> C
6      val result = intA + intC // Se suman ambos valores (ADD)
7      A = (result and 0xFF).toByte() // Al resultado se le quita el signo por ↪
8          ↪ precaución y se convierte a Byte
9
10 updateAddOperationFlags(intA, intC, result) // Se actualizan los Flags ↪
11     ↪ correspondientes
12
13 return CYCLES_4 // Devolvemos los ciclos que la CPU debe tardar en ↪
14     ↪ ejecutar la instrucción
15 }

```

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            0xFA -> ld_a_nn() // LD A, [NN]
13            0xFB -> ei() // EI
14            0xFE -> cp_n() // CP N
15            0xFF -> rst(0x0038) // RST 38H
16            else -> throw IllegalArgumentException("Instruction not supported: $←
17                           ↪ {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 llegaría en caso de que en este primero encontrremos 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

```

1   fun fetch(): Byte {
2       val byte = Memory.getByteOnAddress(PC)
3
4       if(!cpu_halt_bug){
5           PC = (PC + 1) and 0xFFFF
6       }else{
7           cpu_halt_bug = false
8       }
9
10      return byte
11  }
12
13  fun fetch16(): Int {
14      val low = fetch().toInt() and 0xFF
15      val high = fetch().toInt() and 0xFF
16      return return get_16bit_address(high, low)
17  }
18
19  fun get_16bit_address(high: Byte, low: Byte): Int{
20      return ((high.toInt() and 0xFF) shl 8) or (low.toInt() and 0xFF)
21  }
22
23  fun set_16bit_address_value(high: Byte, low: Byte, value: Byte){
24      val address = get_16bit_address(high, low)
25      Memory.writeByteOnAddress(address, value)
26  }

```

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

```

1   fun ld_c_l(): Int{
2       C = L
3       return CYCLES_4
4   }
5
6   fun ld_c_hl(): Int{
7       val hl = get_16bit_address(H, L)
8       C = Memory.getByteOnAddress(hl)
9       return CYCLES_8
10  }
11
12  fun ld_hl_b(): Int{
13      set_16bit_address_value(H, L, B)
14      return CYCLES_8
15  }
16
17  fun ld_hl_nn(): Int{
18      val low = fetch().toInt() and 0xFF
19      val high = fetch().toInt() and 0xFF
20      H = high.toByte()
21      L = low.toByte()
22
23      return CYCLES_12
24  }

```

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 **LittleEndian**, 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

```

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

```

```

2     updateFlag(FLAGS_Z, result == 0)
3     clearFlag(FLAGS_N)
4     updateFlag(FLAGS_H, (val1 and 0xF) + (val2 and 0xF) > 0xF)
5     updateFlag(FLAGS_C, result > 0xFF)
6 }
7
8     fun add_a_b(): Int{
9         val intA = A.toInt() and 0xFF
10        val intB = B.toInt() and 0xFF
11        val result = intA + intB
12        A = (result and 0xFF).toByte()
13
14        updateAddOperationFlags(intA, intB, result)
15
16        return CYCLES_4
17    }
18
19    fun adc_a_b(): Int{
20        val carry = if (flagIsSet(FLAGS_C)) 1 else 0
21        val intA = A.toInt() and 0xFF
22        val intB = B.toInt() and 0xFF
23        val result = intA + (intB + carry)
24        A = (result and 0xFF).toByte()
25
26        updateAddOperationFlags(intA, intB + carry, result)
27
28        return CYCLES_4
29    }

```

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) ?? como referencia.

Código 7.8: Operaciones INC y DEC

```

1     fun inc_8bit_register(register: Byte): Byte{
2         val toReturn = (register.toInt() + 1).toByte()
3         updateFlag(FLAGS_Z, toReturn.toInt() == 0x00)
4         clearFlag(FLAGS_N)
5         updateFlag(FLAGS_H, ((register.toInt() and 0xF) + 1) and 0x10 != 0x00)
6         return toReturn
7     }
8
9     fun dec_8bit_register(register: Byte): Byte{
10        val toReturn = (register.toInt() - 1).toByte()

```

```

11     updateFlag(FLAGS_Z, toReturn.toInt() == 0x00)
12     setFlag(FLAGS_N)
13     updateFlag(FLAGS_H, (register.toInt() and 0xF == 0x00))
14     return toReturn
15 }
16
17 fun inc_bc(): Int{
18     val oldValue = get_16bit_address(B, C)
19     val newValue = (oldValue + 1) and 0xFFFF
20     B = (newValue shr 8).toByte()
21     C = newValue.toByte()
22     return CYCLES_8
23 }
24
25 fun dec_b(): Int{
26     B = dec_8bit_register(B)
27     return CYCLES_4
28 }
29
30 fun inc_b(): Int{
31     B = inc_8bit_register(B)
32     return CYCLES_4
33 }
```

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

```

1 fun xor_b(): Int{
2     A = (((A.toInt() and 0xFF) xor (register.toInt() and 0xFF)).toByte()
3
4     updateFlag(FLAGS_Z, (A.toInt() and 0xFF) == 0)
5     clearFlag(FLAGS_N)
6     clearFlag(FLAGS_H)
7     clearFlag(FLAGS_C)
8
9     return CYCLES_4
10 }
```

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

```

1  fun call_nz_nn(): Int{
2      val address = fetch16()
3
4      if (!flagIsSet(FLAGS_Z)) {
5          SP = (SP - 1) and 0xFFFF
6          Memory.writeByteOnAddress(SP, (PC ushr 8).toByte()) // Alto
7          SP = (SP - 1) and 0xFFFF
8          Memory.writeByteOnAddress(SP, (PC and 0xFF).toByte()) // Bajo
9
10         PC = address
11         return CYCLES_24
12     }
13
14     return CYCLES_12
15 }
```

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 última 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

```

1  fun jr_n(): Int{
2      val offset = fetch()
3      PC += offset.toInt()
4      return CYCLES_12
5  }
6
7  fun jp_nn(): Int{
8      val address = fetch16()
9      PC = address
10     return CYCLES_16
11 }
```

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

```

1   fun executeRetOperation(){
2       val low = Memory.getByteOnAddress(SP).toInt() and 0xFF
3       SP = (SP + 1) and 0xFFFF
4       val high = Memory.getByteOnAddress(SP).toInt() and 0xFF
5       SP = (SP + 1) and 0xFFFF
6
7       PC = (high shl 8) or low
8   }
9
10  fun ret(): Int{
11      executeRetOperation()
12      return CYCLES_16
13  }
14
15  fun reti(): Int{
16      executeRetOperation()
17      Interrupt.enableInterrupts(true)
18      return CYCLES_16
19  }

```

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

```

1   fun cp_b(): Int{
2       val intA = A.toInt() and 0xFF
3       val intRegister = B.toInt() and 0xFF
4
5       val result = (intA - intRegister) and 0xFF
6
7       updateFlag(FLAGS_Z, result == 0)
8       setFlag(FLAGS_N)

```

```

9     updateFlag(FLAGS_H, (intA and 0xF) < (intRegister and 0xF))
10    updateFlag(FLAGS_C, intA < intRegister)
11
12    return CYCLES_4
13 }
```

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

```

1  fun cpl(): Int{
2
3      A = (A.toInt() xor 0xFF).toByte()
4
5      setFlag(FLAGS_N)
6      setFlag(FLAGS_H)
7
8      return CYCLES_4
9 }
```

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

```

1  fun ccf(): Int{
2
3      val newCarry = ((F.toInt() and 0xFF) and FLAGS_C) == 0
4      updateFlag(FLAGS_C, newCarry)
5      setFlag(FLAGS_N)
6      setFlag(FLAGS_H)
7
8      return CYCLES_4
9 }
```

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

```

1  fun daa(): Int{
2
3      var result = A.toInt() and 0xFF
4
5      if (!flagIsSet(FLAG_N)) { // Addition
6
7          if ((result and 0x0F) > 9 || flagIsSet(FLAG_H)) // Lower nibble
8              result += 0x06
9
10         if ((result and 0xF0) > 0x90 || flagIsSet(FLAG_C)) // Higher nibble
11             result += 0x60
12
13     }else{ // Subtraction
14         if (flagIsSet(FLAG_H)) // Lower nibble
15             result -= 0x06
16
17         if (flagIsSet(FLAG_C)) // Higher nibble
18             result -= 0x60
19     }
20
21     updateFlag(FLAG_Z, (result and 0xFF) == 0x00)
22     clearFlag(FLAG_H)
23     updateFlag(FLAG_C, result > 0xFF)
24
25     result = result and 0xFF
26     A = result.toByte()
27
28     return CYCLES_4
29 }
```

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

```

1  fun scf(): Int{
2
3      clearFlag(FLAG_N)
4      clearFlag(FLAG_H)
5      setFlag(FLAG_C)
6
7      return CYCLES_4
8 }
```

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

```

1   fun rl_b(): Int{
2       val bByte = B.toInt() and 0xFF
3       val oldCarry = if (flagIsSet(FLAG_C)) 1 else 0
4       val newCarry = (bByte ushr 7) and 0x1
5
6       B = ((bByte shl 1) or oldCarry).toByte()
7
8       updateFlag(FLAG_Z, B == 0.toByte())
9       clearFlag(FLAG_N)
10      clearFlag(FLAG_H)
11      updateFlag(FLAG_C, newCarry == 1)
12
13      return CYCLES_8
14  }
15
16  fun rr_b(): Int{
17      val bByte = B.toInt() and 0xFF
18      val oldCarry = if (flagIsSet(FLAG_C)) 1 else 0
19      val newCarry = (bByte ushr 7) and 0x1
20
21      B = ((bByte shr 1) or (oldCarry shl 7)).toByte()
22
23      updateFlag(FLAG_Z, B == 0.toByte())
24      clearFlag(FLAG_N)
25      clearFlag(FLAG_H)
26      updateFlag(FLAG_C, newCarry == 1)
27
28      return CYCLES_8
29  }

```

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

```

1   fun rlc_b(): Int{
2       val bByte = B.toInt() and 0xFF
3       val carry = (bByte ushr 7) and 0x1
4       B = ((bByte shl 1) or carry).toByte()
5
6       updateFlag(FLAG_Z, B == 0.toByte())
7       clearFlag(FLAG_N)

```

```

8     clearFlag(FLAGS_H)
9     updateFlag(FLAGS_C, carry == 1)
10
11    return CYCLES_8
12 }
13
14 fun rrc_b(): Int{
15     val bByte = B.toInt() and 0xFF
16     val carry = bByte and 0x1
17     B = ((bByte shr 1) or (carry shl 7)).toByte()
18
19     updateFlag(FLAGS_Z, B == 0.toByte())
20     clearFlag(FLAGS_N)
21     clearFlag(FLAGS_H)
22     updateFlag(FLAGS_C, carry == 1)
23
24     return CYCLES_8
25 }
```

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

```

1 fun sla_b(): Int{
2     val bByte = B.toInt() and 0xFF
3     val newCarry = (bByte ushr 7) and 0x1
4     B = ((bByte shl 1) and 0xFE).toByte()
5
6     updateFlag(FLAGS_Z, B == 0.toByte())
7     clearFlag(FLAGS_N)
8     clearFlag(FLAGS_H)
9     updateFlag(FLAGS_C, newCarry == 1)
10
11    return CYCLES_8
12 }
13
14 fun sra_b(): Int{
15     val bByte = B.toInt() and 0xFF
16     val oldBit7 = bByte and 0x80
17     val newCarry = bByte and 0x1
18
19     B = ((bByte shr 1) or oldBit7).toByte()
20
21     updateFlag(FLAGS_Z, B == 0.toByte())
22     clearFlag(FLAGS_N)
23     clearFlag(FLAGS_H)
24     updateFlag(FLAGS_C, newCarry == 1)
25
26     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(FLAGS_Z, B == 0.toByte())
35         clearFlag(FLAGS_N)
36         clearFlag(FLAGS_H)
37         updateFlag(FLAGS_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(FLAGS_Z, B == 0.toByte())
8         clearFlag(FLAGS_N)
9         clearFlag(FLAGS_H)
10        clearFlag(FLAGS_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(FLAGS_Z, result)
3         clearFlag(FLAGS_N)
4         setFlag(FLAGS_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) ←
26                           ← and bit) == 0
27                cyclesToReturn = CYCLES_16
28            }
29            8 -> bitZero = ((A.toInt() and 0xFF) and bit) == 0
30        }
31
32        updateBitOperationFlags(bitZero)
33        return cyclesToReturn
34    }

```

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()

```

```

12         3 -> D = ((D.toInt() and 0xFF) and bit).toByte()
13         4 -> E = ((E.toInt() and 0xFF) and bit).toByte()
14         5 -> H = ((H.toInt() and 0xFF) and bit).toByte()
15         6 -> L = ((L.toInt() and 0xFF) and bit).toByte()
16         7 -> {
17             val address = get_16bit_address(H, L)
18             val result = ((Memory.getByteOnAddress(address).toInt() and 0xFF) ←
19                           ← and bit).toByte()
20             Memory.writeByteOnAddress(address, result)
21             cyclesToReturn = CYCLES_16
22         }
23     }
24
25     return cyclesToReturn
26 }
```

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

```

1 fun set_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
8
9     when (register) {
10         1 -> B = ((B.toInt() and 0xFF) or bit).toByte()
11         2 -> C = ((C.toInt() and 0xFF) or bit).toByte()
12         3 -> D = ((D.toInt() and 0xFF) or bit).toByte()
13         4 -> E = ((E.toInt() and 0xFF) or bit).toByte()
14         5 -> H = ((H.toInt() and 0xFF) or bit).toByte()
15         6 -> L = ((L.toInt() and 0xFF) or bit).toByte()
16         7 -> {
17             val address = get_16bit_address(H, L)
18             val result = ((Memory.getByteOnAddress(address).toInt() and 0xFF) ←
19                           ← or bit).toByte()
20             Memory.writeByteOnAddress(address, result)
21             cyclesToReturn = CYCLES_16
22         }
23         8 -> A = ((A.toInt() and 0xFF) or bit).toByte()
24     }
```

```

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

```

1   fun push_bc(): Int{
2       SP = (SP - 1) and 0xFFFF
3       Memory.writeByteOnAddress(SP, B) // high
4       SP = (SP - 1) and 0xFFFF
5       Memory.writeByteOnAddress(SP, C) // low
6
7       return CYCLES_16
8   }
9
10  fun pop_bc(): Int{
11
12      C = Memory.getByteOnAddress(SP)
13      SP = (SP + 1) and 0xFFFF
14      B = Memory.getByteOnAddress(SP)
15      SP = (SP + 1) and 0xFFFF
16
17      return CYCLES_12
18  }

```

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

```

1   fun ldh_n_a(): Int{
2
3       val byte = fetch().toInt() and 0xFF
4       val address = (0xFF00 + byte) and 0xFFFF
5       Memory.writeByteOnAddress(address, A)
6
7       return CYCLES_12
8   }
9
10  fun ld_cn_a(): Int{
11      val address = (0xFF00 + (C.toInt() and 0xFF)) and 0xFFFF
12      Memory.writeByteOnAddress(address, A)
13
14      return CYCLES_8
15  }

```

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 = 0x0OFF
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 = 0xFEAO
14 const val IO_START = 0xFF00
15 const val HRAM_START = 0xFF80
16 const val HRAM_END = 0xFFFFE
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

```

1  fun writeByteOnAddress(address: Int, value: Byte){
2      if(address < VRAM_START) { // ROM DATA
3          ROM.writeToROM(address, value)
4      }else if(address < EXTERNAL_RAM_START) { // VRAM DATA
5          PPU.writeToVRAM(address, value)
6      }else if(address < WRAM_START) { // EXTERNAL / CARTRIDGE RAM DATA
7          ROM.writeToROM(address, value)
8      }else if(address < ECHO_RAM_START){ // WRAM
9          RAM.writeToWRAM(address, value)
10     }else if(address < OAM_START) { // ECHO RAM -- CANT BE USED !
11         return
12     }else if(address < RESERVED_MEM_START) { // OAM DATA
13         if(!DMA.transferring())
14             PPU.writeToOAM(address, -1, value)
15     }else if(address < IO_START) { // RESERVED MEMORY - CANT BE USED !
16         return
17     }else if(address < HRAM_START) { // IO DATA
18         IO.writeToIO(address, value)
19     }else if(address < IE){ // HRAM DATA
20         RAM.writeToHRAM(address, value)
21     }else if(address == IE){ // IE FLAG DATA
22         write(address, value)
23     }
24 }

25
26 fun getByteOnAddress(address: Int): Byte{
27     if(address < VRAM_START) { // ROM DATA
28         return ROM.readFromROM(address)
29     }else if(address < EXTERNAL_RAM_START) { // VRAM DATA
30         return PPU.readFromVRAM(address)
31     }else if(address < WRAM_START) { // EXTERNAL / CARTRIDGE RAM DATA
32         return ROM.readFromROM(address)
33     }else if(address < ECHO_RAM_START){ // WRAM
34         return RAM.readFromWRAM(address)
35     }else if(address < OAM_START) { // ECHO RAM -- CANT BE USED !
36         return 0
37     }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() ←
3          ↪ (), 0xD0.toByte(), 0x00.toByte(), 0xB0.toByte(),
4      0x03.toByte(), 0x73.toByte(), 0x00.toByte(), 0x83.toByte(), 0x00.toByte() ←
5          ↪ (), 0xC0.toByte(), 0x00.toByte(), 0x0D.toByte(),
6      0x00.toByte(), 0x08.toByte(), 0x11.toByte(), 0x1F.toByte(), 0x88.toByte() ←
7          ↪ (), 0x89.toByte(), 0x00.toByte(), 0x0E.toByte(),
8      0xDC.toByte(), 0xCC.toByte(), 0x6E.toByte(), 0xE6.toByte(), 0xDD.toByte() ←
9          ↪ (), 0xDD.toByte(), 0xD9.toByte(), 0x99.toByte(),
0      0xBB.toByte(), 0xBB.toByte(), 0x67.toByte(), 0x63.toByte(), 0x6E.toByte() ←
1      ↪ (), 0x0E.toByte(), 0xEC.toByte(), 0xCC.toByte(),
2      0xDD.toByte(), 0xDC.toByte(), 0x99.toByte(), 0x9F.toByte(), 0xBB.toByte() ←
3          ↪ (), 0xB9.toByte(), 0x33.toByte(), 0x3E.toByte()
4  )
5
```

```
10  private val bootstrapRom = byteArrayOf(
11      0x31.toByte(), 0xFE.toByte(), 0xFF.toByte(), // LD SP, 0xFFFFE
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() ←
114     ↪ (), 0xA5.toByte(), 0x42.toByte(), 0x3C.toByte(),
115     0x21.toByte(), 0x04.toByte(), 0x01.toByte(),
116     0x11.toByte(), 0xA8.toByte(), 0x00.toByte(),
117     0x1A.toByte(), // LD A, [DE]
118     0x13.toByte(), // INC DE
119     0xBE.toByte(), // CP [HL]
120     0x20.toByte(), 0xFE.toByte(), // JR NZ, PC + 0xFE
121     0x23.toByte(), // INC HL
122     0x7D.toByte(), // LD A, L
123     0xFE.toByte(), 0x34.toByte(), // CP 0x34
124     0x20.toByte(), 0xF5.toByte(), // JR NZ, PC + 0xF5
125     0x06.toByte(), 0x19.toByte(), // LD B, 0x19
126     0x78.toByte(), // LD A, B
127     0x86.toByte(), // ADD A, [HL]
128     0x23.toByte(), // INC HL
129     0x05.toByte(), // DEC B
130     0x20.toByte(), 0xFB.toByte(), // JR NZ, PC + 0xFB
131     0x86.toByte(), // ADD A, [HL]
132     0x20.toByte(), 0xFE.toByte(), // JR NZ, 0xFE
133     0x3E.toByte(), 0x01.toByte(), // LD A, 0x01
134     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(←
5      ↪ ActivityResultContracts.OpenDocument()) { uri ->
6      uri?.let {
7
8          val intent = Intent(this, EmuActivity::class.java).apply {
9              putExtra(ROM_URI_EXTRA, it.toString())
10         }
11         startActivity(intent)
12     }
13 }
14
15 override fun onCreate(savedInstanceState: Bundle?) {
16     super.onCreate(savedInstanceState)
17
18     binding = ActivityMainBinding.inflate(layoutInflater)
19     setContentView(binding.root)
20
21     selectRomButton = binding.selectRomButton
22
23     selectRomButton.setOnClickListener {
24         openFilePicker()
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`.

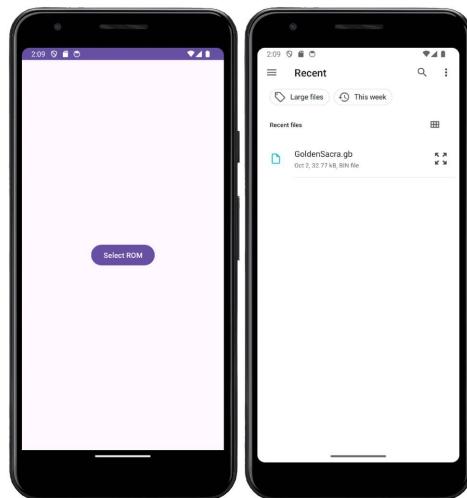


Figura 7.1: Selección de archivo desde la memoria SD del dispositivo.

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.

```

1   fun load_rom(romBytes: ByteArray): Boolean{
2
3       try {
4           if(romBytes.isNotEmpty()){
5               val cartSize = min(romBytes.size, ROM_END - ROM_START + 1)
6
7               for (i in 0 until cartSize) {
8                   Memory.write(ROM_START + i, romBytes[i])
9               }
10
11           return rom_init(romBytes)
12       }
13   }catch (ex: Exception){
14       println("Error loading ROM: $ex")
15   }
16
17   return false
18 }
```

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.

```

1   enum class CONSOLE_TYPE(val value : Int){
2       DMG(0),
3       DMG_CGB(0x80),
4       CGB(0xC0),
5       UNKNOWN(-1);
6
7       companion object {
8           fun fromValue(value: Int): CONSOLE_TYPE {
9               return entries.find { it.value == value } ?: UNKNOWN
10          }
11      }
12  }
13
14  private var bootSection = ByteArray(0xFF)
15
16  private var cartTitle : String = "Unknown"
17  private var licenseCode : String = "None"
18  private var cartType : Int = -1
19  private var romSize : Int = -1
20  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 ↫
67             ↪ } // Filter only ASCII characters
68         return String(title.toByteArray(),Charsets.US_ASCII)
69     }
70
71     fun rom_init(romBytes: ByteArray): Boolean{
```

```

71     // Compare Cartridge Header with the Boot fixed one
72
73     // Get header section from the romBytes
74     val bootByteArray = Memory.getNintendoLogo()
75     val cartByteArray = extractByteArray(romBytes, N_LOGO_START, N_LOGO_END, ←
76         ↪ true) // Nintendo Logo on Cartridge goes from 0x104 to 0x133
77
78     if(memcmp(Memory.getNintendoLogo(), cartByteArray, bootByteArray.size) ←
79         ↪ != 0){
80         return false
81     }
82
83     cartTitle = convertBytesToString(extractByteArray(romBytes, TITLE_START, ←
84         ↪ TITLE_END, true))
85
86     licenseCode = if((extractByte(romBytes, OLD_LCNS_CODE).toInt() and 0xFF) ←
87         ↪ == NEW_LICENSE_CODE){
88         getNewLicenseNameFromIndex(convertBytesToString(extractByteArray(←
89             ↪ romBytes, LCNS_CODE_START, LCNS_CODE_END, true)))
90     }else{
91         getOldLicenseNameFromIndex(extractByte(romBytes, OLD_LCNS_CODE).←
92             ↪ toInt() and 0xFF)
93     }
94
95     cartType = extractByte(romBytes, CART_TYPE).toInt() and 0xFF
96     romSize = 32 * (1 shl extractByte(romBytes, ROM_SIZE).toInt() and 0xFF) ←
97         ↪ // Value in KiB
98     ramSize = extractByte(romBytes, RAM_SIZE).toInt() and 0xFF
99     romVersion = extractByte(romBytes, ROM_V_NUM).toInt() and 0xFF
100    console = CONSOLE_TYPE.fromValue(extractByte(romBytes, TITLE_END).toInt←
101        ↪ () and 0xFF)
102
103    println("ROM Loaded Successfully!")
104
105    bootSection = extractByteArray(romBytes, 0x00, 0xFF, true) // Save ←
106        ↪ portion of code where the boot is going to load
107
108    return true
109 }

```

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.

```

1 --- Super Marioland
2 Cart Title: SUPER MARIO LAND
3 License: Nintendo
4 Cart Type: MBC1
5 ROM Size: 64
6 ROM Version Number: 0
7 RAM Size: 0
8 Console Type: DMG
9
10 --- Pokemon Azul
11 Cart Title: POKEMON BLUE
12 License: Nintendo R&D1
13 Cart Type: MBC5+RAM+BATTERY
14 ROM Size: 1024
15 ROM Version Number: 0
16 RAM Size: 3
17 Console Type: DMG
18
19 --- Harry Potter y la Piedra Filosofal
20 Cart Title: HARRY POTTER BHVE
21 License: EA (Electronic Arts)
22 Cart Type: MBC5+RAM+BATTERY
23 ROM Size: 4096
24 ROM Version Number: 0
25 RAM Size: 2
26 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.

```

1 0000: E1 CD 9D 31 E9 87 87 87 85 6F 3E 00 8C 67 7E C9 | ...1.....o>..g~.
2 0010: 7A BC C0 7B BD C9 FF FF 3E 01 E0 A6 C9 FF FF FF | z...{....>.....
3 0020: AF E0 A6 3E 02 E0 A8 C9 7C E0 51 7D E0 52 7A E0 | ...>....|.Q}.Rz.
4 0030: 53 7B E0 54 79 E0 55 C9 40 F5 D1 7B EA BA CD C9 | Sf.Ty.U.0..{....
5 0040: C3 80 36 FF FF FF FF C3 CE C0 C3 AC 23 FF FF | ..6.....#..
6 0050: FB C3 AC 39 FF FF FF FF C3 A3 38 FF FF FF FF FF | ...9.....8.....
7 0060: D9 D7 38 04 D5 54 5D E1 CB 2A CB 1B CB 2A CB 1B | ..8..T]...*....*..
8 0070: CB 2A CB 1B 19 19 19 C9 CB 7C C8 F5 7D 2F C6 01 | .*.....|...}../.
9 0080: 6F 7C 2F CE 00 67 F1 C9 F5 79 2F C6 01 4F 78 2F | o!/.g...y..0x/
10 0090: CE 00 47 F1 C9 CB 7A C8 F5 7B 2F C6 01 5F 7A 2F | ..G...z..{/_z/
11 00A0: CE 00 57 F1 C9 C5 06 08 AF 87 CB 15 30 04 84 30 | ..W.....0..0
12 00B0: 01 2C 05 20 F4 65 6F C1 C9 AF B5 20 03 65 37 C9 | ..eo....eo...
13 00C0: C5 D5 06 08 4D 2E 00 CB 14 CB 15 5D 7D 91 6F 3F | ....M.....].o?

```

```

14 00D0: 38 01 6B 05 20 F1 CB 14 B7 D1 C1 C9 C5 4D 44 3E | 8.k. ....MD>
15 00E0: 0F 21 00 00 CB 23 CB 12 30 01 09 29 3D 20 F5 CB | .!...#..0..)=...
16 00F0: 7A 28 01 09 C1 C9 19 3E 02 EA 00 20 5E 23 56 C9 | z(....>... ^#V.
17 0100: 00 C3 50 01 CE ED 66 66 CC 0D 00 0B 03 73 00 83 | ..P...ff.....s..
18 0110: 00 0C 00 0D 00 08 11 1F 88 89 00 0E DC CC 6E E6 | .....n....
19 0120: DD DD D9 99 BB BB 67 63 6E 0E EC CC DD DC 99 9F | .....gcn.....
20 0130: BB B9 33 3E 48 41 52 52 59 50 4F 54 54 45 52 42 | ..3>HARRYPOTTERB
21 0140: 48 56 45 C0 36 39 00 1B 07 02 01 33 00 D7 B4 78 | HVE.69....3...x
22 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

7.4.2.0.1 Implementación de las Acciones de Lectura y Escritura en ROM Para implementar las operaciones, es necesario tener en cuenta cómo cada MBC gestiona de manera específica las diferentes regiones de memoria y los registros asociados. Cada MBC presenta un mecanismo particular para el acceso y la selección de bancos de ROM y RAM, lo que afecta directamente la forma en que el emulador debe interactuar con ellos.

Con el objetivo de lograr un emulador completo, es imprescindible implementar todos los tipos de controladores, desde el MBC1 hasta el MBC7, así como otros controladores especiales. No obstante, en este apartado se detallará exclusivamente la implementación del MBC1, que abarca una amplia gama de títulos, incluyendo juegos como *Tetris DX* y *Mario & Yoshi*.

La estructura propuesta, de forma que se faciliten la integración de los distintos tipos de MBC, es la siguiente:

- Se generará una interfaz llamada MBCInterface, el cual tendrá dos funciones a implementar de lectura y escritura.
- Una clase abstracta MBC la cual implementa la interfaz y que contendrá datos como el modo de banca, el banco actual de ROM o los bancos de RAM.
- El módulo de ROM contendrá una variable de clase de tipo MBCInterface que por defecto será nulo y que obtendrá valor en la lectura inicial de ROM.
- El tipo NoMBC implementará directamente el MBCInterface, mientras que MBC1 heredará de la clase abstracta MBC.
- A la hora de escribir o leer en ROM, simplemente se trasladará la responsabilidad de la tarea a la variable guardada.

Código 7.39: Inicialización del MBC.

```

1  private var mbcInterface: MBCInterface? = null
2  [...]
3  private fun initMBC(romBytes: ByteArray){
4      mbcInterface = when(getCartTypeIndex()){
5          0 -> NoMBC(romBytes)
6          1 -> MBC1(romBytes)
7          [...] // Resto de controladores
8          else -> null
}
```

```

9      }
10 }
```

Como ya ha sido explicado, dependiendo de las regiones de memoria en las que se intente escribir, se deben actualizar unos registros específicos.

Código 7.40: Lectura en regiones ROM.

```

1   override fun read(address: Int): Byte {
2     when (address) {
3       in ROM_START..< ROM_SW_START -> { // READ FROM ROM FIXED BANK
4         return when(bankingMode){
5           BankingMode.MODE_0 -> Memory.read(address)
6           BankingMode.MODE_1 -> romData?.get(getROMOBankAddr(address)) ←
7             ↪ ?: 0xFF.toByte()
8         }
9       in ROM_SW_START..ROM_END -> return romData?.get(getROMSwBankAddr(←
10          ↪ address)) ?: 0xFF.toByte() // READ FROM SWITCHABLE ROM BANK
11       in EXTERNAL_RAM_START..< WRAM_START -> { // READ FROM EXTERNAL RAM
12         return if(ramEnabled) {
13           ramBanks!![currentRamBank] [address - EXTERNAL_RAM_START]
14         } else 0xFF.toByte()
15     }
16     return 0xFF.toByte()
17   }
18 }
```

Desglosemos el código:

- Si la dirección de memoria a leer se encuentra en el rango 0x0000–0x3FFF (ROM fija o estática), primero se comprobará el modo de banco. En el caso del modo básico, la lectura se redirigirá al módulo de memoria. Para el modo avanzado, la lectura se realizará directamente sobre los bytes cargados del fichero .gb/.gbc. La función `getROMOBankAddr(address: Int)` verifica la cantidad de bancos ROM que contiene el cartucho, aplicando, si es necesario, una máscara y utilizando el registro de dos bits de la región 0x4000–0x5FFF de manera complementaria.
- Si la dirección se encuentra en la región de ROM dinámica, 0x4000–0x7FFF, los datos se obtendrán directamente del fichero .gb/.gbc. La función `getROMSwBankAddr(address: Int)` implementará una lógica similar a la de `getROMOBankAddr`, con la diferencia de que, en este caso, se sumará el resultado de la multiplicación entre el número de banco ROM actual y el tamaño de un banco (16 KiB) a la dirección indicada.
- Si la dirección se encuentra dentro del rango de la RAM externa, se comprobará primero que esta esté activada, para proceder a leer los valores correspondientes de la variable de clase.

Veamos ahora la escritura:

Código 7.41: Escritura en regiones ROM.

```

1   override fun write(address: Int, value: Byte) {
2       val valueInt = value.toInt() and 0xFF
3
4       when {
5           address <= ENABLE_RAM_END -> { // ENABLE RAM
6               if(cartHasRam())
7                   ramEnabled = valueInt and 0xF == 0xA
8           }
9           address in (ENABLE_RAM_END + 1)..ROM_BANK_NUMBER_END -> { // ROM ↪
10              ↪ BANK SELECTION
11              var currentRomBank = valueInt and romBankMask
12              if (currentRomBank == 0) currentRomBank += 1
13          }
14           address in (ROM_BANK_NUMBER_END + 1)..RAM_BANK_NUMBER_END -> { // ↪
15              ↪ RAM BANK SELECTION
16              if (bankingMode == BankingMode.MODE_1 && saveNeeded) {
17                  saveExRAMToFile()
18              }
19              currentRamBank = valueInt and 0b11
20          }
21           address in (RAM_BANK_NUMBER_END + 1)..RAM_BANK_MODE_END -> { // ↪
22              ↪ BANKING MODE
23              bankingMode = if (valueInt and 0b1 != 0) BankingMode.MODE_1 else ↪
24                  BankingMode.MODE_0
25
26              if(bankingMode == BankingMode.MODE_1 && saveNeeded){
27                  saveExRAMToFile()
28              }
29          }
30           address in EXTERNAL_RAM_START..< WRAM_START -> { // WRITE TO ↪
31              ↪ EXTERNAL RAM
32              if(ramEnabled){
33                  ramBanks!![currentRamBank] [address - EXTERNAL_RAM_START] = ↪
34                      value
35
36                  if(cartHasBattery()) saveNeeded = true
37              }
38          }
39      }
40  }

```

Desglosemos nuevamente el código:

- Si la dirección de memoria se encuentra en el rango 0x0000 hasta 0x1FFF, se realiza la habilitación de RAM externa.
 - Si el cartucho incluye RAM, se habilita la RAM cuando los 4 bits inferiores del valor escrito (`value`) son iguales a 0xA.
- Si la dirección está en el rango desde 0x2000 hasta 0x3FFF, se selecciona el banco de ROM.

- Se determina el banco de ROM actual utilizando una máscara `romBankMask` y aplicándola a `value`.
- Si el banco resultante es el banco 0 (que está reservado), se ajusta el valor a 1 para evitar acceder a la ROM reservada.
- Si la dirección está en el rango desde 0x4000 hasta 0x5FFF, se selecciona el banco de RAM.
 - Si el modo de banca está en básico y es necesario guardar el estado, se llama a la función `saveExRAMToFile()` para almacenar los datos en el archivo de RAM externa.
- Si la dirección está en el rango desde 0x6000 hasta 0x7FFF, se define el modo de banca.
 - Se selecciona el modo 1 si el bit menos significativo es 1, y modo 0 en caso contrario.
 - Si el modo seleccionado es el básico y se requiere guardar, se realiza nuevamente el guardado con `saveExRAMToFile()`.
- Si la dirección cae en el rango de RAM externa, 0xA000 hasta 0xBFFF, se realiza una escritura en la RAM externa.
 - Si la RAM está habilitada, el valor se escribe en el banco de RAM actual, ajustando la dirección para obtener la posición correcta.
 - Si el cartucho tiene batería, se establece un flag para indicar que se requiere guardar.

Con esto ya tendríamos lo básico para que nuestro controlador funcione. Partiendo de este ejemplo, se deben implementar el resto de los tipos existentes para asegurar que cualquier juego funcione.

7.5 Interrupciones

Las interrupciones deben ser procesadas en todos los ciclos de CPU, justo antes de ejecutar la siguiente instrucción de ROM. En la siguiente implementación, se puede ver el inicio del método `tick()`:

Código 7.42: Interrupciones en el método principal de la CPU.

```

1  fun tick(): Boolean{
2
3      // Interrupciones
4      if(!pendingEI){
5          handleInterrupts()
6      }else{
7          pendingEI = false
8          Interrupt.enableInterrupts(true)
9      }
10
11     [...]

```

El flag *pendingEI* es una variable que se activa en el momento que un opcode EI es ejecutado. Como se explicó previamente, esta instrucción no habilita las interrupciones de forma inmediata, si no que toma efecto en el siguiente ciclo de ejecución.

En el método `handleInterrupts()`, se va a comprobar que las interrupciones estén habilitadas y que se deba procesar al menos una. En caso de que ambas condiciones se cumplan, se delegará el trabajo al módulo específico de las interrupciones.

La lógica principal del módulo es la siguiente:

Código 7.43: Lógica principal del módulo Interrupt.

```

1   fun flush(){
2       val activeInterrupts = getPendingInterrupts()
3
4       if ((activeInterrupts and InterruptType.VBLANK.getInterruptMask()) != 0) ←
5           ↪ return handleInterrupt(VBLANK_PTR, InterruptType.VBLANK)
6       else if ((activeInterrupts and InterruptType.LCD_STAT.getInterruptMask() ←
7           ↪ ) != 0) return handleInterrupt(LCD_STAT_PTR, InterruptType.←
8           ↪ LCD_STAT)
9       else if ((activeInterrupts and InterruptType.TIMER.getInterruptMask()) ←
10          ↪ != 0) return handleInterrupt(TIMER_PTR, InterruptType.TIMER)
11      else if ((activeInterrupts and InterruptType.SERIAL.getInterruptMask()) ←
12          ↪ != 0) return handleInterrupt(SERIAL_PTR, InterruptType.SERIAL)
13      else if ((activeInterrupts and InterruptType.JOYPAD.getInterruptMask()) ←
14          ↪ != 0) return handleInterrupt(JOYPAD_PTR, InterruptType.JOYPAD)
15  }
16
17  private fun handleInterrupt(address: Int, type: InterruptType){
18      enableInterrupts(false) // IME se deshabilita para prevenir que otras ←
19          ↪ interrupciones se ejecuten
20
21      // Desactivar interrupciones
22      val bit = (type.getInterruptMask()).inv()
23      val ifValue = (Memory.getByteOnAddress(IF).toInt() and 0xFF)
24      set_IF(ifValue and bit)
25
26      CPU.executeInterrupt(address)
27  }

```

Primero se ejecuta la función `flush()`, la cual se encarga de verificar las interrupciones pendientes y, si se detecta alguna, manejarla de manera adecuada:

1. Se obtienen las interrupciones activas mediante la función `getPendingInterrupts()`, que retorna una máscara de bits representando las interrupciones pendientes.
2. Se evalúa si cada tipo de interrupción está activa utilizando operaciones AND en combinación con las máscaras de interrupción.
3. Si se detecta una interrupción activa para el tipo VBLANK, se llama a la función `handleInterrupt()` con la dirección de interrupción (0x0040) y el tipo correspondiente,

finalizando así la ejecución de flush. Para el resto de tipos se hace lo mismo, siguiente la prioridad de bit.

Por otro lado, la función `handleInterrupt(address: Int, type: InterruptType)` gestiona el proceso de atender una interrupción específica, asegurándose de que no se produzcan interrupciones adicionales durante su ejecución.

1. Se deshabilitan las interrupciones globalmente llamando a `enableInterrupts(false)`. Esto evita que otras interrupciones se ejecuten mientras se atiende la actual.
2. Se lee el valor actual del registro IF, se realiza una operación bit a bit and con la máscara invertida para desactivar el bit correspondiente a la interrupción actual, y se actualiza en memoria.
3. Finalmente, se ejecuta la rutina de servicio de interrupción llamando al método `executeInterrupt(address)`. En este se desactiva el HALT de la CPU (en caso de que lo estuviese), se pushea PC al stack pointer, y se sobreescribe PC con la dirección obtenida por parámetro.

Lo que ocurría a continuación depende de la decisión que haya tomado el desarrollador del juego que se ejecute.

7.6 Temporizadores / Timers

La implementación debe simular el comportamiento de los temporizadores en el hardware original. Para ello se comenzará declarando unas constantes que representen las direcciones de memoria reales y se generarán las funciones de lectura y escritura:

Código 7.44: Lectura y escritura en registros de Timers.

```

1  const val DIV = 0xFF04 // Divider Register (16 bit Register, but only the ↪
2      ↪ upper 8 bit (0-7) are public to the developer)
3  const val TIMA = 0xFF05 // Timer Counter
4  const val TMA = 0xFF06 // Timer Module
5  const val TAC = 0xFF07 // Timer Control
6
7  [...]
8
9  fun readFromTimer(address: Int): Byte{
10     when(address){
11         DIV -> return ((div16 shr 8) and 0xFF).toByte()
12         TIMA -> return Memory.read(TIMA)
13         TMA -> return Memory.read(TMA)
14         TAC -> return Memory.read(TAC)
15     }
16
17     return 0
18 }
19
20 fun writeToTimer(address: Int, value: Byte){

```

```

20     when(address){
21         DIV -> {
22             div16 = (div16 and 0xFF00)
23         }
24         TIMA -> Memory.write(TIMA, value)
25         TMA -> Memory.write(TMA, value)
26         TAC -> Memory.write(TAC, value)
27     }
28 }
```

A la hora de interactuar con el registro DIV, es importante considerar que internamente consta de 16 bits, aunque externamente solo se exponen los 8 bits más altos. Para manejarlo, se utiliza una variable de clase llamada *div16*, que se inicializa siempre en 0x0000.

Cuando se realiza una operación de escritura en este registro, únicamente se reinician a 0 los 8 bits menos significativos, preservando el estado de los 8 bits más altos. Por otro lado, al leer el registro, solo se devuelven los 8 bits más altos del valor contenido.

Por último, se debe implementar la función que actualice en cada ciclo de ejecución el temporizador. Los pasos que debe seguir son:

- Comparar el valor actual de DIV y verificar si ha llegado el momento de incrementar el contador TIMA basado en el valor de TAC (registro de control del temporizador).
- Usar el valor de TAC para determinar la frecuencia de incremento de DIV. Dependiendo de los dos primeros bits de TAC, el temporizador se incrementará a una velocidad diferente (256, 4, 16 o 64 ciclos de máquina).
- Si TIMA llega a 0xFF, se recarga con el valor de TMA y se genera una interrupción de temporizador.

Código 7.45: Lectura y escritura en registros de Timers.

```

1 fun tick(){
2     val prevDIV = div16
3     div16 += 1
4
5     var timerUpdate = false
6     val tacValue = Memory.read(TAC).toInt() and 0xFF
7
8     when(tacValue and 0b11){
9         0b00 -> timerUpdate = (prevDIV and (1 shl 9) != 0) && (div16 and (1 ←
10           ↘ shl 9) == 0)
11         0b01 -> timerUpdate = (prevDIV and (1 shl 3) != 0) && (div16 and (1 ←
12           ↘ shl 3) == 0)
13         0b10 -> timerUpdate = (prevDIV and (1 shl 5) != 0) && (div16 and (1 ←
           ↘ shl 5) == 0)
14         0b11 -> timerUpdate = (prevDIV and (1 shl 7) != 0) && (div16 and (1 ←
           ↘ shl 7) == 0)
15     }
16 }
```

```

14
15     if(timerUpdate && (tacValue and (1 shl 2) != 0)){
16         var timaValue = Memory.read(TIMA).toInt() and 0xFF
17         timaValue++
18         Memory.write(TIMA, timaValue.toByte())
19
20         if(timaValue == 0xFF){
21             Memory.write(TIMA, Memory.read(TMA))
22             Interrupt.requestInterrupt(Interrupt.InterruptType.TIMER.←
23                 ← getByteMask())
24         }
25     }

```

7.7 RAM

En la Game Boy, el término RAM puede hacer referencia a distintas áreas de memoria: WRAM (Working RAM), HRAM (High RAM), VRAM (Video RAM) o la RAM externa presente en algunos cartuchos. Este módulo se centrará en la implementación de WRAM y HRAM, ya que la VRAM está estrechamente vinculada al funcionamiento del módulo PPU (Processing Pixel Unit), mientras que la RAM externa está gestionada por los controladores de memoria de los cartuchos (MBC, Memory Bank Controllers).

La implementación de la **HRAM**, siendo ésta un espacio fijo de memoria, se delegará directamente al módulo Memory, que ya se encarga de manejar lecturas y escrituras en memoria. Debido a la simplicidad de su gestión y a que no requiere lógica específica adicional, no se desarrollará un apartado independiente para explicar su implementación en esta documentación.

7.7.1 WRAM

La WRAM en modo DMG es fija y se maneja directamente, mientras que en modo CGB incluye un banco fijo y hasta siete bancos comutables adicionales, permitiendo que la memoria sea más dinámica. Estos bancos se almacenan en un array (*wramBanks*) y el banco activo se controla mediante la variable *wramBank*. Las lecturas y escrituras en la WRAM consideran si se está accediendo a un banco comutable o al espacio fijo, ajustando automáticamente la dirección de memoria según sea necesario.

Código 7.46: Lectura y escritura en Work RAM.

```

1   private var wramBanks: Array<ByteArray> = Array(CGB_SW_BANKS) { ByteArray(←
2       ← ECHO_RAM_START - SWITCHABLE_WRAM_START) } // Not used if not in CGB ←
3       ← mode
4   private var wramBank = 0
5
6   fun readFromWRAM(address: Int) : Byte{
7       return if(!ROM.isCGB() || address < SWITCHABLE_WRAM_START){
8           Memory.read(address)
9       }
10      else
11          wramBanks[wramBank].get(address)
12  }
13
14  fun writeToWRAM(address: Int, value: Byte){
15      if(!ROM.isCGB() || address < SWITCHABLE_WRAM_START){
16          Memory.write(address, value)
17      }
18      else
19          wramBanks[wramBank].set(address, value)
20  }

```

```

7     }else{
8         wramBanks[wramBank][address - SWITCHABLE_WRAM_START]
9     }
10    }
11
12    fun writeToWRAM(address: Int, value: Byte){
13        if(!ROM.isCGB() || address < SWITCHABLE_WRAM_START){
14            Memory.write(address, value)
15        }else{
16            wramBanks[wramBank][address - SWITCHABLE_WRAM_START] = value
17        }
18    }

```

7.7.2 VRAM

Al estar estrechamente vinculada al módulo de la PPU, será ahí donde declaremos la variable. Únicamente se llegará a utilizar en modo CGB, ya que de otra forma acudiremos al módulo de memoria. Su implementación no tiene mucho más misterio: debe tener las dimensiones correctas y, en caso de estar en modo CGB, obtener o escribir los valores en ella:

Código 7.47: Lectura y escritura en Video RAM.

```

1  private var vRam : ByteArray = ByteArray((VRAM_END - VRAM_START) + 1)
2
3  fun readFromVRAM(address: Int) : Byte{
4      return if(!ROM.isCGB()){
5          Memory.read(address)
6      }else{
7          vRam[address - VRAM_START]
8      }
9  }
10
11 fun writeToVRAM(address: Int, value: Byte){
12     if(!ROM.isCGB()){
13         Memory.write(address, value)
14     }else{
15         vRam[address - VRAM_START] = value
16     }
17 }

```

7.7.3 RAM Externa

La RAM Externa, como se ha visto, reside en los cartuchos y es gestionada por los controladores de memoria. En este caso, se ha implementado un controlador MBC1, por lo que la RAM externa se manejará a través de él. La lectura y escritura en la RAM externa se delegan al controlador MBC1, que se encarga de gestionar los bancos de RAM y de redirigir las operaciones a la RAM externa.

La implementación ya se ha proporcionado en el apartado de lectura y escritura en ROM.

7.8 DMA

Como se ha descrito en el apartado teórico, definiremos una función que inicie el proceso de DMA. Por ahora no será utilizada, ya que para ello antes se debe implementar la PPU.

En esta función de inicio daremos por activado el proceso, calcularemos la dirección de memoria sobre la que comenzar a iterar, y desactivaremos todas las interrupciones para que otra no pueda interrumpir la ejecución.

Código 7.48: Inicio del proceso DMA.

```

1  private var active : Boolean = false
2  private var offset : Int = 0
3  private var startAddress : Int = 0
4  private var startDelay : Byte = 0
5
6  fun start(value: Byte){ // Value: Indica el rango (0xXX00 - 0xXX9F)
7      active = true
8      offset = 0
9      startDelay = 2
10     startAddress = (value.toInt() and 0xFF) shl 8 // Si el valor es 0xC00, la
11         ↪ dirección será 0xC000
12     Interrupt.enableInterrupts(false) // Deshabilitamos las interrupciones
13 }
14 }
```

En el método *tick()* comprobaremos si hay o no un delay para comenzar la ejecución. En caso negativo obtendremos la dirección de inicio a la que le sumaremos el offset actual, y copiaremos el byte desde la RAM a la OAM. Al terminar de copiar los 160 bytes, desactivaremos el proceso y reactivaremos las interrupciones para continuar con la ejecución normal.

Código 7.49: Proceso DMA.

```

1  fun tick(){
2      if (active){
3          if(startDelay.toInt() == 0){
4              val address = startAddress + offset
5              PPU.writeToOAM(address, startAddress, Memory.getByteOnAddress(
6                  ↪ address))
7              offset++
8              active = (offset and 0xFF) < TOTAL_BYTES_TO_COPY
9
10             if(!active)
11                 Interrupt.enableInterrupts(true)
12             }else{
13                 startDelay--
14             }
15 }
```

7.9 PPU

En la PPU debemos antes de nada definir bastantes objetos con los que luego podamos trabajar de forma eficiente:

- Enumeradores para obtener los bits individuales del LCDC, LCD Status, modo de PPU, paletas de color seleccionadas y atributos de un sprite.
- Un mapa que represente las distintas paletas y sus códigos de color con los que podamos trabajar en Android.
- Todas las variables de clase que necesitemos, como el contador de sprites por línea de renderizado, array de bytes para la VRam y la OAM, frame actual, paleta de color seleccionada, etc...

Una vez definidos todos estos objetos, podremos pasar a inicializar la PPU:

Código 7.50: Inicialización de la PPU.

```

1  fun init() {
2      Memory.write(LCD_STAT, 0x81.toByte())
3      Memory.write(LCDC_ADDR, 0x91.toByte())
4      Memory.write(SCY, 0)
5      Memory.write(SCX, 0)
6      Memory.write(LY_ADDR, 0)
7      Memory.write(LYC_ADDR, 0)
8      Memory.write(WY, 0)
9      Memory.write(WX, 0)
10     val consoleType = ROM.getConsole()
11
12     if(consoleType == ROM.CONSOLE_TYPE.DMG || consoleType == ROM.←
13         ↪ CONSOLE_TYPE.DMG_CGB) {
14         Memory.write(BGP, 0xFC.toByte())
15         Memory.write(OBP0, 0xFF.toByte())
16         Memory.write(OBP1, 0xFF.toByte())
17     }else{
18         //TODO - CGB
19     }
20
21     handleLCDC(0x91.toByte())
22 }
23
24 [...]
25
26 private fun handleLCDC(value: Byte){
27     // Bit 7
28     ppuEnabled = LCDCObj.LCDC_ENABLE.get(value) != 0
29     lcdEnabled = ppuEnabled
30     // Bit 6
31     winTilemapAddr = if(LCDCObj.WIN_TILEMAP.get(value) == 0) TM_1_START else←
32         ↪ TM_2_START
33     // Bit 5
34 }
```

```

32     enabledWindow = LCDCObj.WINDOW_ENABLE.get(value) != 0
33     // Bit 4
34     addrModeAddr = if(LCDCObj.ADDRESS_MODE.get(value) == 0) ←
35         ↪ SIGNED_TILE_REGION else VRAM_START
36     // Bit 3
37     bgTilemapAddr = if(LCDCObj.BG_TILEMAP.get(value) == 0) TM_1_START else ←
38         ↪ TM_2_START
39     // Bit 2
40     objSize = if(LCDCObj.OBJ_SIZE.get(value) == 0) 8 else 16
41     // Bit 1
42     objEnabled = LCDCObj.OBJ_ENABLE.get(value) != 0
43     // Bit 0
44     bgWinEnabled = LCDCObj.MASTER_ENABLE.get(value) != 0
45 }
```

Primero se configura la pantalla LCD:

- **LCD Status (LCD_STAT)**: Se escribe el valor `0x81` (10000001 en binario). Este valor activa la interrupción cuando la línea de escaneo coincide con el valor de LYC y asegura que la pantalla empiece en un estado adecuado.
- **LCD Control (LCDC)**: Se escribe el valor `0x91` (10010001 en binario), que habilita la pantalla, permite el uso de sprites y selecciona un modo de dirección de tiles.

Ahora configuramos los registros de desplazamiento y posición:

- **SCY y SCX**: Se establecen en 0 para que la imagen de fondo no esté desplazada.
- **LY y LYC**: Se inician en 0 para que la pantalla empiece desde la primera línea de escaneo.
- **WY y WX**: Controlan la posición de la ventana en pantalla, comenzando en 0.

Configuramos la paleta de colores detectando el tipo de consola:

- Si es modo DMG o en modo compatibilidad (DMG_CGB):
 - **BGP** se establece en `0xFC` (11111100 en binario), configurando la paleta de colores del fondo.
 - **OBP0 y OBP1** se establecen en `0xFF`, dejando los sprites con su configuración predeterminada.
- Para el modo CGB, aún falta implementar la configuración específica.

Finalmente, se llama a `handleLCDC(0x91)` para aplicar la configuración del registro LCDC

Al igual que el DMA y la CPU, la PPU también dispondrá de su propio método `tick()`. En él, comprobaremos que la PPU esté activa y ejecutaremos el código de los distintos modos según venga indicado en el LCD Stat.

Código 7.51: Lógica principal de la PPU.

```
1   fun tick(){
```

```

2     if(moduleIsActive()) {
3         lineTicks++
4         val stat = Memory.getByteOnAddress(LCD_STAT)
5
6         when (StatObj.PPU_MODE.get(stat)) {
7             PPUMode.HBlank.number -> hBlankMode(stat) // MODE 0
8             PPUMode.VBlank.number -> vBlankMode(stat) // MODE 1
9             PPUMode.OAM.number -> oamMode(stat) // MODE 2
10            PPUMode.DRAW_LCD.number -> drawLCDMode(stat) // MODE 3
11        }
12    }
13 }
```

7.9.1 Modo 0: H-Blank

El proceso verifica si se ha alcanzado el número total de ciclos (LINE_TOTAL_TICKS) para la línea actual y, si es así, se actualiza el valor del registro LY. Si la línea actual ha alcanzado o superado la resolución vertical de la Game Boy (GB_Y_RESOLUTION), significa que la PPU ha terminado de procesar todas las líneas visibles y entra en el modo V-Blank. En este caso, el método actualiza el estado de LCD_STAT para reflejar este cambio y solicita una interrupción de V-Blank. Además, incrementa el contador de fotogramas y llama a calculateFPS() para actualizar la métrica de rendimiento.

Por otro lado, si LY no ha alcanzado este límite, significa que la PPU aún está dentro del proceso de renderizado de la pantalla y debe volver al modo OAM. En este caso, el método actualiza LCD_STAT para reflejar este cambio y, si la interrupción de estado del LCD para OAM está habilitada, solicita la interrupción correspondiente. Finalmente, restablece lineTicks a cero para comenzar el procesamiento de la nueva línea.

Código 7.52: Lógica del proceso de H-Blank.

```

1  private fun hBlankMode(stat: Byte){
2      if(lineTicks >= LINE_TOTAL_TICKS){
3          increment_LY()
4
5          val ly = (Memory.getByteOnAddress(LY_ADDR).toInt()) and 0xFF
6
7          if(ly >= GB_Y_RESOLUTION){ // ENTER VBLANK MODE
8              Memory.write(LCD_STAT, StatObj.PPU_MODE.set(stat, PPUMode.VBlank.←
9                           ↩ number))
10
11         Interrupt.requestInterrupt(Interrupt.InterruptType.VBLANK.←
12                           ↩ getByteMask()) // ASK FOR VBLANK INTERRUPT
13
14         if(StatObj.VBLANK_INTERRUPT.get(stat) != 0)
15             Interrupt.requestInterrupt(Interrupt.InterruptType.LCD_STAT.←
16                           ↩ getByteMask()) // ASK FOR LCD STAT INTERRUPT IF ←
17                           ↩ LCD_STAT HAS THE VBLANK BIT ACTIVATED
18
19         currentFrame++
20     }
21 }
```

```

16
17     calculateFPS()
18
19 }else{ // RETURN TO OAM MODE
20     Memory.write(LCD_STAT, StatObj.PPU_MODE.set(stat, PPUMode.OAM.←
21     ↪ number))
22
23     if(StatObj.OAM_INTERRUPT.get(stat) != 0)
24         Interrupt.requestInterrupt(Interrupt.InterruptType.LCD_STAT.←
25         ↪ getByteMask()) // ASK FOR LCD STAT INTERRUPT IF ←
26         ↪ LCD_STAT HAS THE OAM BIT ACTIVATED
27 }
28
29     lineTicks = 0
30 }
31 }
```

7.9.2 Modo 1: V-Blank

El proceso verifica si se ha alcanzado el número total de ciclos para la línea actual y, si es así, incrementa el valor del registro LY. Luego, obtiene el valor actualizado y comprueba si ha alcanzado o superado el número total de líneas. Si esto ocurre, significa que el periodo de V-Blank ha finalizado y la PPU debe volver al modo OAM para comenzar el renderizado del siguiente cuadro.

En este caso, el método actualiza el estado de LCD_STAT para reflejar el cambio al modo OAM y restablece LY_ADDR a 0, indicando el inicio de un nuevo fotograma. Además, si la interrupción de estado del LCD para OAM está habilitada, solicita la interrupción. Al igual que en H-Blank, lineTicks se restablece a cero para comenzar el procesamiento de la nueva línea.

Código 7.53: Lógica del proceso de V-Blank.

```

1  private fun vBlankMode(stat: Byte){
2      if(lineTicks >= LINE_TOTAL_TICKS){
3          increment_LY()
4
5          val ly = (Memory.getByteOnAddress(LY_ADDR).toInt()) and 0xFF
6
7          if(ly >= TOTAL_LINES){ // RETURN TO OAM MODE
8              Memory.write(LCD_STAT, StatObj.PPU_MODE.set(stat, PPUMode.OAM.←
9                  ↪ number))
10             Memory.write(LY_ADDR, 0)
11
12             if(StatObj.OAM_INTERRUPT.get(stat) != 0)
13                 Interrupt.requestInterrupt(Interrupt.InterruptType.LCD_STAT.←
14                     ↪ getByteMask()) // ASK FOR LCD STAT INTERRUPT IF ←
15                     ↪ LCD_STAT HAS THE OAM BIT ACTIVATED
16     }
17 }
```

```

15         lineTicks = 0
16     }
17 }
```

7.9.3 Modo 2: OAM Scan

Aquí gestionamos el modo OAM Scan, el cual se encarga de escanear los sprites visibles en la línea actual antes de proceder al renderizado.

Si el contador lineTicks ha alcanzado el número de ciclos necesarios para el modo OAM, la PPU cambia al modo de dibujo (DRAW_LCD), actualizando el estado de LCD_STAT y reiniciando los parámetros del FIFO fetcher, que se encarga de obtener los píxeles a renderizar. Sin embargo, en el primer ciclo, se inicializa el conteo de sprites en la línea y se llama a loadLineSprites(), que carga los sprites visibles en la línea actual para que puedan ser procesados durante la fase de dibujo.

Código 7.54: Lógica del proceso de OAM Scan.

```

1  private fun oamMode(stat: Byte){ // MODE 2
2      if(lineTicks >= OAM_CYCLES){ // ENTER DRAWING PIXELS MODE
3          Memory.write(LCD_STAT, StatObj.PPU_MODE.set(stat, PPUMode.DRAW_LCD.←
4              ↪ number))
5          fifoFetcher.resetParams() // Reset FIFO Fetcher
6      }else if(lineTicks == 1){ // Scanning -> Read OAM on the first tick
7          lineSpriteCount = 0
8          loadLineSprites()
9      }
}
```

El método `loadLineSprites()` se encarga de seleccionar y ordenar los sprites que deben renderizarse en la línea de escaneo actual de la pantalla. Esta función solo se ejecuta si los sprites están habilitados o si la consola es un modelo CGB, ya que en este último caso siempre se procesan los sprites.

Primero, se obtiene la línea de escaneo actual, sumándole un desplazamiento para compensar la posición real de los sprites. Luego, se determina el tamaño de los sprites, que puede ser de 8 o 16 píxeles. A continuación, se inicializa el array `objsFetched` con valores nulos y se resetea el contador que indica el número de sprites obtenidos por línea. Por último, se recorre la memoria OAM en bloques de 4 bytes (siendo cada uno en orden: posición Y, posición X, índice de tile y atributos). Si se han alcanzado el número máximo de sprites por línea, el bucle se detiene.

Para cada sprite, se verifican las coordenadas x e y para determinar si es visible en la línea actual. Si lo es, se almacena en el array `objsFetched`, incrementando el contador a posteriori. Finalmente, se ordena la lista de sprites por su posición x, asegurando que aquellos con menor x se rendericen primero.

Código 7.55: Obtención de sprites en modo OAM Scan.

```

1  private fun loadLineSprites(){
2
3      if(objEnabled || ROM.isCGB()) { // CGB Ignores this condition
4
5          val currentY = (Memory.getByteOnAddress(LY_ADDR).toInt() and 0xFF) + ↵
6              ↪ OAM_Y_OFFSET
7          val lcdc = Memory.getByteOnAddress(LCDC_ADDR)
8          val objSize = LCDCOBJ.OBJ_SIZE.get(lcdc)
9          val spriteHeight = if (objSize == 0) 8 else 16
10
11         objsFetched.fill(null)
12         lineSpriteCount = 0
13
14         for (i in oamRam.indices step 4) {
15
16             if (lineSpriteCount >= MAX_OBJ_PER_SCANLINE) {
17                 break
18             }
19
20             val y = oamRam[i].toInt() and 0xFF
21             val x = oamRam[i + 1].toInt() and 0xFF
22             val tile = oamRam[i + 2]
23             val flags = oamRam[i + 3]
24
25             if (x == 0 || x >= (GB_X_RESOLUTION + OAM_X_OFFSET)) { // Sprite ↵
26                 ↪ not visible
27                 continue
28             }
29
30             if (y <= currentY && (y + spriteHeight) > currentY) { // Sprite ↵
31                 ↪ on current line
32                 objsFetched[lineSpriteCount] = OAMObj(y.toByte(), x.toByte(), ↵
33                     ↪ tile, flags)
34                 lineSpriteCount++
35             }
36
37         }
38
39         objsFetched.sortBy { it?.x ?: Byte.MAX_VALUE} // Sort by X position
40     }
41 }
```

7.9.4 Modo 3: Drawing Pixels

El modo de dibujo es el proceso en el que la PPU renderiza los píxeles en la pantalla. En este modo, se obtienen los datos de los tiles y se procesan para generar la imagen final. La función `drawLCDMode()` gestiona este proceso:

Código 7.56: Dibujado de píxeles en Modo 3.

```

1  private fun drawLCDMode(stat: Byte){
2 }
```

```

3     fifoFetcher.process()
4
5     if(fifoFetcher.getPushedPixels() >= GB_X_RESOLUTION){ // ENTER HBLANK ↪
6         ↪ MODE
7
8     fifoFetcher.clear()
9
10    Memory.write(LCD_STAT, StatObj.PPU_MODE.set(stat, PPUMode.HBlank.↪
11        ↪ number))
12
13    if(StatObj.HBLANK_INTERRUPT.get(stat) != 0)
14        Interrupt.requestInterrupt(Interrupt.InterruptType.LCD_STAT.↪
15            ↪ getByteMask()) // ASK FOR LCD STAT INTERRUPT IF LCD_STAT ↪
16            ↪ HAS THE HBLANK BIT ACTIVATED
17
18 }
19 }
```

El código propio del FIFO vendrá documentado más adelante, pero por ahora se debe entender que su función es transferir la información de los píxeles al búfer de la pantalla.

Después del proceso, se comprueba que el número total de píxeles (correspondientes a la línea actual) que se han añadido sea mayor o igual a la resolución horizontal de la pantalla. Si estamos en este caso, reiniciaremos el proceso volviendo al modo H-Blank, y lanzando una interrupcion del Stat si estuviese activo.

7.9.5 FIFO Fetcher

Como viene definido en la parte teórica, la GB disponía de dos FIFO's, los cuales trabajan de manera independiente para transferir los píxeles de Background y Window por un lado, y los Sprites por otro. El proceso original finalizaba haciendo una mezcla de la información proporcionada por ambos.

En este proyecto se va a utilizar una implementación algo más sencilla, tanto de implementar como de entender, basada en la guía de programación de ?.

Como variables de clase, se necesita guardar:

- Estado del fetcher (las distintas etapas del procesado).
- Coordenada X del tile en VRAM del que se está obteniendo información.
- Coordenada X del scanline que se va a dibujar.
- El propio FIFO, del cual se realizará una implementación propia.
- El búfer de video, que será un array de enteros, con un tamaño fijo equivalente a `Resolución_Y * Resolución_X`.
- Un array de 3 bytes donde se irá insertando información específica del tile sobre el que se trabajará.

- Un array de 3 objetos OAM (definido más adelante) se guardarán los metadatos de los sprites.
- Un array de 6 bytes que almacene los datos de píxeles crudos de los sprites.
- La coordenada Y de la línea actual sobre la que se deben procesar los tiles.
- Coordenadas globales X e Y en el mapa de tiles para determinar qué tile de VRAM debe ser obtenido y procesado.

La lógica principal de procesado se ha implementado de la siguiente manera:

Código 7.57: Código principal del FIFO Fetcher.

```

1  fun process(){
2      val scx = Memory.getByteOnAddress(SCX).toInt() and 0xFF
3      val scy = Memory.getByteOnAddress(SCY).toInt() and 0xFF
4      val ly = Memory.getByteOnAddress(LY_ADDR).toInt() and 0xFF
5
6      mapY = scy + ly
7      mapX = scx + fetchX
8      tileY = (mapY % 8) * 2
9
10     if(PPU.getLineTicks() % 2 == 0){
11         fetch()
12     }
13     pushPixelsToBuffer() // Push pixels to pipeline
14 }
15
16     private fun fetch(){
17         when(state){
18             FetcherState.OBTAIN_TILE -> getTile() // Fetch the current tile ↪
19                         ↪ identification in the BG Tilemap
20             FetcherState.LOW_DATA_TILE -> getTileLowData() // Fetch the low byte ↪
21                         ↪ of the tile
22             FetcherState.HIGH_DATA_TILE -> getTileHighData() // Fetch the high ↪
23                         ↪ byte of the tile
24             FetcherState.SLEEP -> sleepState()
25             FetcherState.PUSH -> pushState()
26         }
27     }

```

En primer lugar, `process()` accede a los registros de memoria SCX, SCY y LY. Utilizando estos registros, calcula las coordenadas absolutas (`mapX`, `mapY`) en el espacio del mapa de tiles. Por ejemplo, `mapY = SCY + LY` permite ajustar la renderización vertical según el desplazamiento del fondo, mientras que `mapX = SCX + fetchX` hace lo mismo para el eje horizontal. La variable `tileY` se deriva de `mapY % 8` para identificar la línea específica dentro del tile actual, esencial para acceder a los datos de píxeles correctos en VRAM.

Para evitar conflictos con el PPU, en la función se verifica el estado de los ciclos de renderizado mediante `PPU.getLineTicks() % 2 == 0`. Esta condición asegura que `fetch()` solo se

ejecute cada 2 ciclos de máquina, simulando el comportamiento del hardware real, donde el acceso a VRAM está restringido durante ciertas fases del renderizado.

Finalmente, `pushPixelsToBuffer()` transfiere los píxeles desde el FIFO al búfer de video. Esta etapa es crítica para mantener un flujo constante de datos hacia la pantalla, asegurando que no haya retrazos en la visualización.

La función `fetch()` implementa una máquina de estados, en el que cada estado representa una etapa específica del pipeline. Los estados son los siguientes:

- **OBTAIN_TILE**: donde se determina qué tile debe ser renderizado.
- **LOW_DATA_TILE**: donde se obtienen los datos de píxeles bajos del tile.
- **HIGH_DATA_TILE**: donde se obtienen los datos de píxeles altos del tile.
- **SLEEP**: donde se espera a que el ciclo de renderizado esté listo para continuar.
- **PUSH**: donde se transfieren los píxeles al búfer de video.

7.9.5.1 OBTAIN_TILE

El fetcher determina el tile que debe ser renderizado. Accede al mapa de tiles (ubicado en \$9800-\$9BFF o \$9C00-\$9FFF, dependiendo de los flags del registro LCDC) para leer el índice del tile correspondiente a las coordenadas mapX y mapY. Este acceso a memoria es fundamental, ya que define los datos que se procesarán en los estados siguientes.

Código 7.58: FIFO Fetcher - Obtención de Tile.

```

1  private fun getTile(){
2      if(PPU.lcdIsEnabled())
3          getBGTile()
4
5      if(PPU.objsAreEnabled() && PPU.getFetchedSpriteEntries().isNotEmpty()) {
6          fetchedSprites = 0
7          getSpriteTile()
8      }
9
10     state = FetcherState.LOW_DATA_TILE
11     fetchX += 8
12 }
```

En primer lugar, la función `getBGTile()` realiza una lectura de los registros que controlan el desplazamiento y posicionamiento de los elementos gráficos. Estos incluyen el LY, las coordenadas de la ventana (WY y WX) y los valores de scroll (SCX y SCY).

Posteriormente, la función realiza una evaluación de las posiciones relativas y los estados de los flags de control, para establecer si el tile que está siendo procesado pertenece al Background o al Window. Esta distinción determina qué tilemap debe ser consultado, alternando entre las zonas reservadas 0x9800 y 0x9C00 según corresponda.

Código 7.59: FIFO Fetcher - Obtención de Tile de Background.

```

1  private fun getBGTile(){
2      val ly = Memory.getByteOnAddress(LY_ADDR).toInt() and 0xFF
3      val wy = Memory.getByteOnAddress(WY).toInt() and 0xFF
4      val scx = Memory.getByteOnAddress(SCX).toInt() and 0xFF
5      val wx = Memory.getByteOnAddress(WX).toInt() and 0xFF - WIN_X_OFFSET
6
7      // Obtain tilemap to use (BG or WIN)
8      val windowTile = isWindowTile()
9      val tilemapToUse = if(windowTile) PPU.getWinTilemapAddr() else PPU.←
10         ↪ getBGTilemapAddr()
11
12      val xCoordinate = if (windowTile) ((fetchX - wx) / 8) else (mapX / 8) ←
13         ↪ and 0x1F
14      val yCoordinate = if (windowTile) ((ly - wy) / 8) else (mapY / 8)
15
16      val address = tilemapToUse + ((xCoordinate + (yCoordinate * ←
17         ↪ GB_X_TOTAL_TILES)) and 0x3ff)
18      var tile = Memory.getByteOnAddress(address) // 1 Tile == 8 Pixels
19
20      if(PPU.getAddrModeAddr() == SIGNED_TILE_REGION){
21          tile = ((tile.toInt() and 0xFF) + 128).toByte() // Signed Region ←
22             ↪ [-128, 128] --> Transform to [0, 255]
23      }
24      tileData[0] = tile
25  }

```

Se transforman las coordenadas globales, expresadas en píxeles, en índices específicos de tiles, teniendo en cuenta que cada elemento gráfico ocupa un área de 8x8 píxeles. Este cálculo considera tanto la posición absoluta en pantalla como también la incorporación de los efectos de desplazamiento producidos por el scroll.

Si el modo de direccionamiento de tiles indica que se trata de una región con direcciones firmadas (índices que pueden ir de -128 a +127), se ajusta el valor sumándole 128 para convertirlo a un índice positivo válido. El identificador final del tile se guarda en el array `tileData`, dejándolo listo para ser usado en la siguiente fase.

Por otro lado, la función `getSpriteTile()` se encarga de obtener el índice del tile correspondiente a los sprites visibles en la línea actual. Esta función es similar a la anterior, pero se centra en los sprites y su posición relativa en la pantalla.

Código 7.60: FIFO Fetcher - Obtención de Tile de Sprites.

```

1  private fun getSpriteTile(){
2
3      val fetchedObjs = PPU.getFetchedSpriteEntries()
4
5      for (obj in fetchedObjs) {
6          if(obj != null) {
7              val scx = Memory.getByteOnAddress(SCX).toInt() and 0xFF

```

```

8     val sprX = ((obj.x.toInt() and 0xFF) - OAM_X_OFFSET) + (scx % ↪
9         ↪ PIXELS_PER_TILE)
10
11    if ((sprX >= fetchX && sprX < fetchX + OAM_X_OFFSET) ||
12        ((sprX + OAM_X_OFFSET) >= fetchX && (sprX + OAM_X_OFFSET) < ↪
13            ↪ fetchX + OAM_X_OFFSET)) {
14        objTileData[fetchedSprites] = obj
15        fetchedSprites++
16    }
17
18    if (fetchedSprites >= 3) break
19}

```

La función recorre la lista proporcionada por la PPU `PPU.getFetchedSpriteEntries()` y comprueba, para cada sprite no nulo, su posición horizontal en pantalla. Para ello, obtiene el valor `SCX` desde memoria, y calcula la posición horizontal del sprite ajustando el valor original del sprite en el OAM, restando una constante que representa el desfase de coordenadas (`OAM_X_OFFSET`) y sumando un pequeño ajuste basado en el módulo del desplazamiento.

Con la posición horizontal calculada, se evalúa si dicho sprite se encuentra dentro del rango de píxeles que se están a punto de dibujar, comparándolo con `fetchX`. Se realiza una doble condición para cubrir tanto la coincidencia exacta como los solapes de píxeles con el ancho de un sprite. Si el sprite cumple estas condiciones de visibilidad, se almacena en el array `objTileData`, y se incrementa el contador `fetchedSprites` que indica el número de sprites seleccionados.

Este proceso continúa hasta que se han encontrado tres sprites coincidentes, momento en el cual el bucle se interrumpe. Esta limitación viene explicada en el marco teórico.

7.9.5.2 LOW_DATA_TILE - HIGH_DATA_TILE

En ambas funciones lo primero que se realiza es el cálculo del offset vertical dentro del tile. Posteriormente, se accede a la VRAM para recuperar el primer o el segundo byte de datos de línea de ese tile, lo que representa los bits bajos o altos de color para los 8 píxeles de esa fila, respectivamente.

La dirección a la que accede se compone usando el identificador del tile, el modo de direccionamiento actual, y la posición vertical dentro del tile. Una vez obtenido ese primer byte, se llama a `loadSpriteData()` con el desplazamiento correspondiente, para que se vayan cargando también los datos de los sprites superpuestos. Finalmente, se cambia el estado del fetcher al siguiente paso, `HIGH_DATA_TILE` o `SLEEP`, dependiendo del estado actual en el que se encuentre el fetcher.

Código 7.61: FIFO Fetcher - Obtención de los Bits Bajos o Altos del Tile.

```

1  private fun getTileLowData(){
2      val offset = calculeTileDataOffset()

```

```

3     tileData[1] = Memory.getByteOnAddress(PPU.getAddrModeAddr() + ((tileData←
4         ↪ [0].toInt() and 0xFF) * 16) + offset)
5     loadSpriteData(0)
6     state = FetcherState.HIGH_DATA_TILE
7
8     private fun getTileHighData(){
9         val offset = calculeTileDataOffset()
10        tileData[2] = Memory.getByteOnAddress(PPU.getAddrModeAddr() + ((tileData←
11            ↪ [0].toInt() and 0xFF) * 16) + (offset + 1))
12        loadSpriteData(1)
13        state = FetcherState.SLEEP
14    }

```

Finalmente, `loadSpriteData()` es la función auxiliar que realiza la lectura de los datos gráficos de los sprites que se van a dibujar sobre la línea actual. Por cada sprite, se calcula la línea dentro del tile que corresponde al valor actual de LY, considerando si el sprite está volteado verticalmente. También se ajusta el índice del tile en caso de sprites de 16 píxeles. Luego, se accede a VRAM para obtener el byte correspondiente a esa línea de píxeles dentro del tile del sprite, y se guarda en el bufer `objFetchedData`.

Es una de las funciones que deberían estar separadas en una lógica independiente, pero que por eliminar cierta complejidad se ha incluido en lo que sería el fetcher de background.

Código 7.62: FIFO Fetcher - Obtención de Información de un Sprite en la Línea Actual.

```

1  private fun loadSpriteData(offset: Int){
2      val ly = Memory.getByteOnAddress(LY_ADDR).toInt() and 0xFF
3      val spriteHeight = if (LCDCObj.OBJ_SIZE.get(Memory.getByteOnAddress(←
4          ↪ LCDC_ADDR)) == 1) 16 else 8
5
6      for(i in 0 until fetchedSprites){
7          if(objTileData[i] != null) {
8              val flags = objTileData[i]!!.flags
9              val spriteY = (objTileData[i]!!.y.toInt() and 0xFF) - ←
10                 ↪ OAM_Y_OFFSET
11              var tileIndex = objTileData[i]!!.tile.toInt() and 0xFF
12
13              if (spriteHeight == 16)
14                  if ((ObjFlags.Y_FLIP.get(flags) == 0 && (ly - spriteY) >= 8) ←
15                     ↪ ||
16                      (ObjFlags.Y_FLIP.get(flags) == 1 && (ly - spriteY) < 8)) {
17                          tileIndex += 1
18
19                      }
20
21              val baseAddress = VRAM_START + (tileIndex * 16)
22              val tileLine = (ly - spriteY) % 8
23              val address = baseAddress + (tileLine * 2)
24
25              objFetchedData[(i * 2) + offset] = Memory.getByteOnAddress(←
26                 ↪ address + offset)
27          }
28      }
29  }

```

```

22         }
23     }
24 }
```

7.9.5.3 SLEEP

La implementación del estado SLEEP no va a tener nada relevante, simplemente va a dar paso al siguiente estado. Este método se utiliza para simular y representar los tiempos reales del hardware.

Código 7.63: FIFO Fetcher - Sleep.

```

1  private fun sleepState(){
2      state = FetcherState.PUSH
3 }
```

7.9.5.4 PUSH

En la función `pushState()` se intenta introducir nuevos píxeles en el FIFO. Si el intento de empujar píxeles al FIFO tiene éxito (lo que significa que la cola tiene espacio suficiente), el estado del fetcher se reinicia para comenzar la obtención del siguiente tile.

Código 7.64: FIFO Fetcher - Push.

```

1  private fun pushState(){
2      val bgPush = pushBGPixelsToFifo()
3
4      if(bgPush){
5          state = FetcherState.OBTAIN_TILE
6      }
7 }
```

La lógica principal ocurre en `pushBGPixelsToFifo()`, que primero comprueba si el FIFO tiene suficiente espacio libre para recibir una nueva tanda de píxeles, que son ocho por cada tile. Si está lleno, simplemente se detiene y devuelve false, esperando a que se vacíe en futuras iteraciones.

Si hay espacio, la función calcula la posición horizontal real de dibujo teniendo en cuenta el SCX y una corrección por el offset de los sprites. A continuación, se procesan los píxeles del tile que acaba de ser cargado. Para cada uno, se extraen los bits correspondientes desde los datos de tile almacenados previamente en `objTileData`, combinando los bits bajos y altos de cada línea para determinar el índice de color del píxel.

Este índice puede cambiar si los sprites están habilitados en la configuración del sistema. En tal caso, se llama a `obtainSpriteColor`, que se encarga de comprobar si algún sprite visible debe sobrescribir el color de fondo en esa posición. Esta función recorre los sprites previamente extraídos para el scanline actual y, si alguno se solapa con la posición del píxel en cuestión, evalúa si el sprite tiene prioridad sobre el fondo. También tiene en cuenta atributos como la inversión horizontal y la transparencia. Si se cumplen las condiciones, el color del

sprite sobrescribe el del fondo.

Nuevamente, esta última parte donde se obtienen los valores de los sprites y se hace una mezcla de valores para determinar el color, debería estar de forma independiente y ser añadido a un segundo FIFO.

Por último se inserta el color en el FIFO de fondo, que es lo que usará el PPU para representar visualmente los píxeles en pantalla. Esto se repite para los ocho píxeles del tile, tras lo cual el método finaliza con éxito

Código 7.65: FIFO Fetcher - Push de Píxeles al FIFO.

```

1  private fun pushBGPixelsToFifo(): Boolean{
2      if(backgroundFifo.getSize() >= 8)
3          return false // Fifo is full
4
5      val scx = Memory.getByteOnAddress(SCX).toInt() and 0xFF
6      val x = fetchX - (OAM_X_OFFSET - (scx % PIXELS_PER_TILE))
7
8      for(i in 0..7){
9          val bit = 7 - i
10
11         val low = (((tileData[1].toInt() and 0xFF) shr bit) and 1)
12         val high = (((tileData[2].toInt() and 0xFF) shr bit) and 1) shl 1
13         var color = if(PPU.bgWinIsEnabled()) PPU.getColorIndex(high or low) ←
14             ↪ else PPU.getColorIndex(0) // Pixel Color
15
16         if(PPU.objsAreEnabled())
17             color = obtainSpriteColor(color)
18
19         if(x >= 0){
20             backgroundFifo.push(color)
21             fifoX++
22         }
23
24     return true
25 }
```

Los colores vienen definidos como propiedad de clase en la PPU, tratando de simular todas las posibles paletas de color que existen tanto en la GB original como en la CGB. Por defecto, se iniciará con la paleta que imita los colores verdosos del primer modelo DMG.

Código 7.66: PPU - Paletas de Color.

```

1  // Colors follows as: White - Light Gray - Dark Gray - Black
2  private val palettes: HashMap<PALETTE_TYPE, IntArray> = hashMapOf(
3      Pair(PALETTE_TYPE.BASIC_PL, intArrayOf(0xFFFFFFFF.toInt(), 0xFFAAAAAA.←
4          ↪ toInt(), 0xFF555555.toInt(), 0xFF000000.toInt())),
5      Pair(PALETTE_TYPE.GREENER_PL, intArrayOf(0xFFE0F8D0.toInt(), 0xFF88C070.←
6          ↪ toInt(), 0xFF346856.toInt(), 0xFF081820.toInt())),
7      [...]
```

```

6      )
7
8  private var selectedPalette = PALETTE_TYPE.GREENER_PL

```

7.10 Game Surface View

El `GameSurfaceView` es la clase encargada de gestionar la representación gráfica de la consola. Esta clase se encarga de crear un canvas y dibujar los píxeles en él, utilizando el búfer de video proporcionado por la PPU. La implementación de esta clase es bastante sencilla, ya que solo se encarga de crear un canvas y dibujar los píxeles en él.

La clase hereda de `SurfaceView` y está diseñada para funcionar de forma independiente gracias a un hilo (`GameThread`) que se encarga del renderizado continuo. Además implementa `SurfaceHolder.Callback`, lo cual permite controlar cuándo se crea, cambia o destruye el canvas.

Código 7.67: Creación del Game Surface View.

```

1  class GameSurfaceView @JvmOverloads constructor(
2      context: Context,
3      attrs: AttributeSet? = null
4  ) : SurfaceView(context, attrs), SurfaceHolder.Callback {
5
6      private var scale : Float = 0f
7      var debugMode = false
8      private var gameThread: GameThread? = null
9
10     init {
11         holder.addCallback(this)
12         gameThread = GameThread(holder, this)
13     }
14
15     [...]

```

El método `onMeasure()` se encarga de calcular las dimensiones basándose en la resolución original de la GB (160x144 píxeles) y en el tamaño disponible en pantalla. La idea es mantener la relación de aspecto original sin deformar la imagen. En función del ancho y alto del dispositivo, se calcula un factor de escala que se utilizará para transformar los píxeles al tamaño físico del dispositivo Android.

Código 7.68: Medición y Escalado del Canvas.

```

1  override fun onMeasure(widthMeasureSpec: Int, heightMeasureSpec: Int) {
2      val width = MeasureSpec.getSize(widthMeasureSpec) // Android phone total ←
3          ↪ width
4      val height = MeasureSpec.getSize(heightMeasureSpec) // Android phone ←
5          ↪ total height
6
7      val aspectRatio = GB_X_RESOLUTION.toFloat() / GB_Y_RESOLUTION.toFloat() ←
8          ↪ // GB Aspect Ratio

```

```

6
7     val newWidth: Int
8     val newHeight: Int
9
10    if (width / height > aspectRatio) { // Landscape
11        newWidth = (height * aspectRatio).toInt()
12        newHeight = height
13        scale = height / GB_Y_RESOLUTION.toFloat()
14    } else { // Straight
15        newWidth = width
16        newHeight = (width / aspectRatio).toInt()
17        scale = width / GB_X_RESOLUTION.toFloat()
18    }
19
20    setMeasuredDimension(newWidth, newHeight)
21 }
```

Durante el renderizado, el método `render()` decide si debe dibujarse el bufer final donde la PPU guarda los colores o si se debe entrar en el modo depuración en el que se muestran los tiles disponibles en memoria. Si el LCD está deshabilitado, simplemente se pinta la pantalla de negro.

Cuando se renderiza la VRAM, se recorre la matriz de píxeles que la PPU ha rellenado y se dibujan los píxeles uno por uno. Para cada tile, se leen 16 bytes de memoria. A partir de estos bytes, se reconstruye el color de cada píxel del tile utilizando la lógica de combinación de bits, y se pinta como un pequeño rectángulo en el canvas.

Código 7.69: Medición y Escalado del Canvas.

```

1  private fun renderVRam(canvas: Canvas){
2      val paint = Paint()
3
4      for (lineNum in 0 until GB_Y_RESOLUTION) {
5          for (x in 0 until GB_X_RESOLUTION) {
6              val left = Math.round(x * scale).toFloat()
7              val top = Math.round(lineNum * scale).toFloat()
8              val right = left + ceil(scale.toDouble()).toFloat()
9              val bottom = top + ceil(scale.toDouble()).toFloat()
10
11              val color = PPU.getBufferPixelFromIndex(x + (lineNum * ↪
12                  ↪ GB_X_RESOLUTION))
13              paint.color = color
14
15              canvas.drawRect(left, top, right, bottom, paint)
16      }
17  }
```

Por último, el `GameThread` estará constantemente renderizando mientras que el flag `running` esté activado. Para hacer este render, primero bloquea el Canvas para ejecutar el método de renderizado del `GameSurfaceView` y finalmente libera el Canvas para que el sistema lo dibuje

en pantalla.

Código 7.70: Renderizado en el GameThread.

```

1   override fun run() {
2       while (running) {
3           var canvas: Canvas? = null
4           try {
5               canvas = surfaceHolder.lockCanvas()
6               if (canvas != null) {
7                   synchronized(surfaceHolder) {
8                       gameSurfaceView.render(canvas)
9                   }
10              }
11          }catch (ex: Exception){
12              ex.printStackTrace()
13          } finally {
14              if (canvas != null) {
15                  surfaceHolder.unlockCanvasAndPost(canvas)
16              }
17          }
18      }
19  }

```

7.10.1 Layout

Ahora el EmuActivity contendrá el `GameSurfaceView` como variable privada de clase. Para hacerlo funcionar, se necesita también añadir la vista al layout y enlazarla con la variable en el `onCreate()`. Se puede insertar, por ejemplo, dentro de un `FrameLayout`:

Código 7.71: GameSurfaceView en Layout.

```

1 <FrameLayout
2     android:id="@+id/frameLayout"
3     app:layout_constraintTop_toTopOf="parent"
4     app:layout_constraintStart_toStartOf="parent"
5     app:layout_constraintEnd_toEndOf="parent"
6     android:layout_width="match_parent"
7     android:layout_centerInParent="true"
8     android:padding="16dp"
9     android:layout_height="wrap_content" >
10
11     <es.atm.gbee.views.GameSurfaceView
12         android:id="@+id/gameSurface"
13         android:layout_width="wrap_content"
14         android:layout_height="wrap_content"
15         app:layout_constraintTop_toTopOf="parent"
16         app:layout_constraintStart_toStartOf="parent"
17         app:layout_constraintEnd_toEndOf="parent"
18     />
19 </FrameLayout>

```

De manera automática, se puede comenzar a ver resultados por pantalla. Aquí distintos resultados obtenidos según se han ido depurando errores, ordenados de forma cronológica:



Figura 7.2: Renderizado de Tiles en SurfaceView

7.11 I/O

Al igual que con el resto de módulos, se debe implementar el encargado de escribir y leer los registros de entrada y salida. Se definirán también un enumerador que indique a qué bit va asignado cada botón del joypad y un objeto que facilite hacer un seguimiento de cuáles están pulsados.

Código 7.72: Estados de los Botones en el Módulo IO.

```

1  enum class GamePadBits(val bit: Int) {
2      A_BUTTON(0),
3      RIGHT_PAD(0),
4      B_BUTTON(1),
5      LEFT_PAD(1),
6      SELECT_BUTTON(2),
7      UP_PAD(2),
8      START_BUTTON(3),
9      DOWN_PAD(3),
10     SELECT_DPAD(4),
11     SELECT_BUTTONS(5);
12
13     fun get(value: Byte): Int {
14         return (value.toInt() and 0xFF) and (0b1 shl bit)
15     }
16 }
17
18 object IO {
19
20     private var serialData = ByteArray(2)
21
22     data class GamepadState(
23         var start: Boolean = false,
24         var select: Boolean = false,
25         var a: Boolean = false,

```

```

26     var b: Boolean = false,
27     var up: Boolean = false,
28     var down: Boolean = false,
29     var left: Boolean = false,
30     var right: Boolean = false
31   )
32
33     private val gamePadState = GamepadState()
34
35   [...]

```

Los métodos de lectura y escritura son muy similares al resto de módulos que ya se han explicado previamente, por lo que se van a omitir en esta explicación para evitar redundancias. Los puntos que se deben tener en cuenta son:

- Si se escribe en el rango [0xFF400xFF40], se delega a la PPU su comportamiento. En general, los casos especiales serán el inicio del DMA o cambios de comportamiento en el LCD.
- Si se escribe en el rango [0xFF040xFF07], se delega al Timer su comportamiento, el cual ya se ha visto en su apartado correspondiente.

El caso importante que se debe controlar en el propio módulo, es el del Joypad. La lógica a seguir es la siguiente:

1. El `EmuActivity` indica que el jugador ha pulsado o dejado de pulsar un botón definido en el layout.
2. Se actualiza la propiedad de la variable de clase con el nuevo valor.
3. Se actualiza el valor en la memoria interna del emulador teniendo en cuenta el funcionamiento matricial original.

A la hora de leer el joypad, simplemente se devuelve el valor almacenado en la memoria interna.

En cuanto a la escritura, la función `updateJoyPadReadOnlyValues(value: Byte)` es el principal encargado. Empieza extrayendo los bits 4 y 5 del valor de entrada; estos bits indican qué grupo de botones se quiere consultar: los botones de dirección (arriba, abajo, izquierda, derecha) o los botones de acción (A, B, Start, Select). Esta selección se hace a través de los bits `SELECT_DPAD` y `SELECT_BUTTONS`.

Si ambos selectores son iguales, no se selecciona ningún grupo concreto, y se ponen todos los bits de botones como "no presionados".

Si solo uno de los dos selectores está activo, se determina cuál y se leen los estados de los botones direccionales del objeto `gamePadState`.

Código 7.73: Escritura de los Estados de los Botones del Joypad

```

1  private fun updateJoyPadReadOnlyValues(value: Byte): Byte{
2      var toReturn = value.toInt() and 0x30
3
4      // 0 == Button Pressed
5      // 1 == Button Released
6
7      val selectDPad = GamePadBits.SELECT_DPAD.get(value)
8      val selectButtons = GamePadBits.SELECT_BUTTONS.get(value)
9
10     if(selectDPad == selectButtons){ // Both selectors are either 0 or 1
11         toReturn = toReturn or 0x0F
12     }else{
13         if(selectDPad == 0){ // DPad selected
14             if(!gamePadState.up) toReturn = toReturn or (0b1 shl GamePadBits.UP_PAD.bit)
15             if(!gamePadState.down) toReturn = toReturn or (0b1 shl GamePadBits.DOWN_PAD.bit)
16             if(!gamePadState.left) toReturn = toReturn or (0b1 shl GamePadBits.LEFT_PAD.bit)
17             if(!gamePadState.right) toReturn = toReturn or (0b1 shl GamePadBits.RIGHT_PAD.bit)
18         }else{ // Buttons selected
19             if(!gamePadState.select) toReturn = toReturn or (0b1 shl GamePadBits.SELECT_BUTTON.bit)
20             if(!gamePadState.start) toReturn = toReturn or (0b1 shl GamePadBits.START_BUTTON.bit)
21             if(!gamePadState.a) toReturn = toReturn or (0b1 shl GamePadBits.A_BUTTON.bit)
22             if(!gamePadState.b) toReturn = toReturn or (0b1 shl GamePadBits.B_BUTTON.bit)
23         }
24     }
25
26     //println(Integer.toBinaryString(toReturn))
27     return toReturn.toByte()
28 }
```

7.12 Joypad

Con la implementación del módulo de entrada/salida, es posible incorporar los controles en la actividad. Lo primero es añadir los botones al layout. Además, se deben tener en cuenta los distintos tamaños de pantalla de los dispositivos, así como la orientación del mismo, ya sea en modo *portrait* (vertical) o *landscape* (horizontal). En otras palabras, el layout debe ser *responsive*.

Se trata de una tarea principalmente de diseño, que puede implicar una gran extensión a nivel de código. Por ello, en este apartado se presentará únicamente un diseño inicial, repre-

sentado de forma gráfica, sobre el cual se puede seguir trabajando y refinando.



Figura 7.3: Diseño inicial en modo Portrait



Figura 7.4: Diseño inicial en modo Landscape

En la actividad se implementará una función llamada `configureButtons()`, cuya finalidad será, por un lado, enlazar cada botón del *layout* con una constante que identifique qué botón representa y, por otro, delegar en la función `setButton(button: View, name: String)` la inicialización de los correspondientes *listeners*.

Código 7.74: Enlace y Lógica de los Botones en la Actividad

```

1  private fun configureButtons(){
2      setButton(binding.dpadUp, UP_DPAD)
3      setButton(binding.dpadDown, DOWN_DPAD)
4      setButton(binding.dpadLeft, LEFT_DPAD)
5      setButton(binding.dpadRight, RIGHT_DPAD)
6      setButton(binding.buttonA, A_BUTTON)
7      setButton(binding.buttonB, B_BUTTON)
8      setButton(binding.buttonSelect, SELECT_BUTTON)
9      setButton(binding.buttonStart, START_BUTTON)
10     dbgButton = binding.switchButton
11 }
12
13 @SuppressLint("ClickableViewAccessibility")
14 private fun setButton(button: View, name: String){
15     button.setOnTouchListener { v, event ->

```

```

16     when (event.action) {
17         MotionEvent.ACTION_DOWN -> {
18             //println("Button pressed: $name")
19             IO.setButtonPressed(name, true)
20             true
21         }
22         MotionEvent.ACTION_UP -> {
23             //println("Button released: $name")
24             IO.setButtonPressed(name, false)
25             v.performClick()
26             true
27         }
28         else -> false
29     }
30 }
31 }
```

Ahora, utilizando nuevamente el juego *Golden Sacra*, al pulsar el botón de Start, el emulador debe dar paso del menú principal a la pantalla de juego:

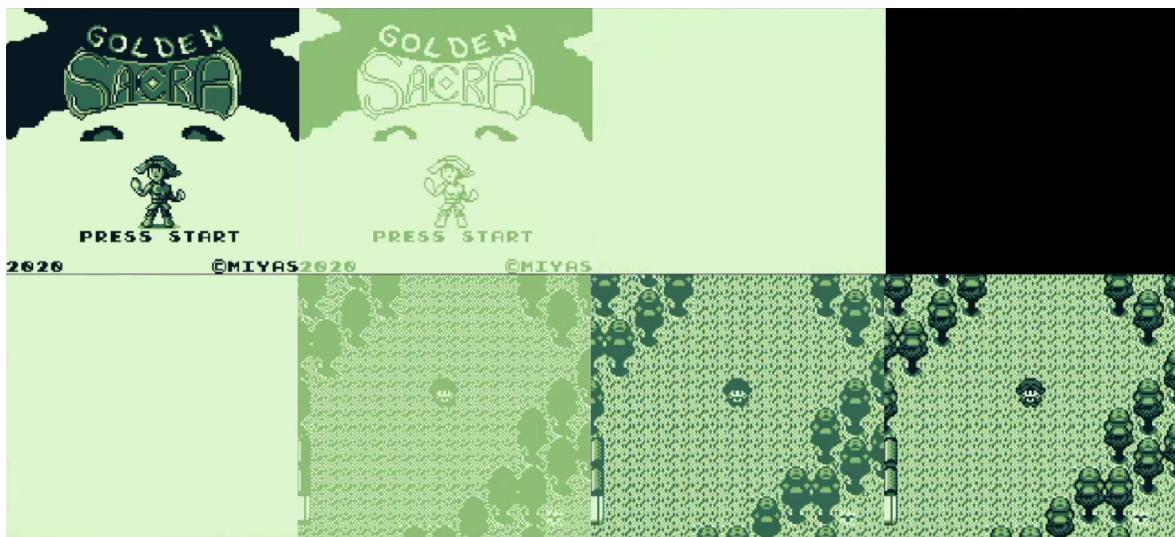


Figura 7.5: Paso desde la Pantalla de Título a Juego.

7.13 MainActivity

A partir de este apartado se enfocará la memoria en lo que es la parte púramente de Android a nivel de aplicación. Se dará comienzo por lo primero que se ve al iniciar la aplicación: el *MainActivity*.

Haciendo uso del diseño original, se procede a generar las siguientes clases, estructuras o modificaciones:

1. Añadir un Toolbar al layout que tenga dos botones, uno para añadir una ROM y otro

para entrar a la configuración.

2. Añadir un RecyclerView al layout, el cual quedará configurado en el `onCreate()`.
3. Generar un Adapter para el RecyclerView, que contendrá la lógica de cómo se van a mostrar las ROMs en pantalla.
4. Crear un layout para cada elemento del RecyclerView, que contendrá una imagen y el nombre de la ROM.
5. Aadir dos *listeners*, el primero (clics corto) que indicará que un usuario quiere iniciar una ROM, y otro (clics largos) que indicará que el usuario quiere hacer algo (eliminar o modificar) esa ROM.
6. Añadir un *listener* al botón de añadir ROM, que abrirá un explorador de archivos para que el usuario seleccione la ROM que quiere añadir.
7. Copiar y guardar las ROMs añadidas por el usuario en la memoria interna de la aplicación, para que se mantengan entre sesiones, independientemente de si el usuario la ha borrado de la carpeta original a posteriori.
8. Generar un modelo de SQLite para guardar la información de las ROMs en la base de datos. De esta forma el usuario podrá modificar el título y la imagen utilizada como carátula.
9. Añadir un *listener* al botón de configuración, que abrirá una nueva actividad donde se podrá modificar la configuración del emulador.

En la configuración del `RecyclerView` se debe establecer el `LayoutManager`, que es el encargado de gestionar la disposición de los elementos en pantalla. En este caso, se utilizará un `GridLayoutManager` para mostrar las ROMs en una cuadrícula. También se debe establecer el `Adapter`, que es el encargado de proporcionar los datos al `RecyclerView`. Para este último punto, se creará una clase `RomAdapter` que extenderá de `RecyclerView.Adapter`.

A continuación se muestra el código de la actividad principal, donde se inicializa el `RecyclerView` y se configuran los *listeners*:

Código 7.75: MainActivity - Configuración del RecyclerView.

```
1  private fun createRecyclerView(){  
2  
3      RomManagement.loadROMSFromDBIfNeeded(this)  
4  
5      val spanCount = if (resources.configuration.orientation == Configuration.orientatio  
6          ↪ .ORIENTATION_LANDSCAPE) {  
7          3  
8      } else {  
9          2  
10     }  
11     binding.romGrid.layoutManager = GridLayoutManager(this, spanCount)
```

```

12     binding.romGrid.itemAnimator = DefaultItemAnimator()
13
14     romAdapter = ROMAdapter(ROMDataSource.roms, spanCount)
15
16     // Click Listener
17     romAdapter.setOnItemClickListener { romPosition ->
18
19         val selectedROMTitle = romAdapter.romList[romPosition].title
20
21         if (selectedROMTitle != null) {
22             val romEntity =
23                 SQLManager.getDatabase(this).romDAO().getROMByTitle(←
24                     ↩ selectedROMTitle)
25
26             if (romEntity != null)
27                 startActivity(
28                     Intent(this, EmuActivity::class.java)
29                     .putExtra(ROM_PATH_EXTRA, romEntity.filePath)
30                 )
31         }
32     }
33
34     romAdapter.setOnLongItemClickListener { romPosition, v ->
35         showPopupMenu(v, romPosition)
36         true
37     }
38
39     binding.romGrid.adapter = romAdapter
}

```

Gracias al `GridLayoutManager`, podemos determinar cuántas ROMs mostrar por fila. Se han definido 2 para el modo *portrait* y 3 para el *landscape*. También se asigna un animador de ítems por defecto para gestionar animaciones al añadir, eliminar o mover elementos.

Posteriormente, se instancia el adaptador `ROMAdapter` pasando la lista de ROMs y el número de columnas. Como se ha expuesto antes, se configuran los *listeners* para los dos tipos de clic. Cuando se selecciona una ROM, se obtiene su título y, si está disponible, se consulta a la base de datos para recuperar la entidad correspondiente. Si se encuentra la ROM, se lanza el intent para pasar al `EmuActivity` y se le pasa la ruta del archivo como extra.

A continuación, se configura el *listener* para el clic largo. En este caso, se muestra un menú emergente relacionado con la ROM seleccionada.

Finalmente, se asigna el adaptador al `RecyclerView`, completando así la configuración.

Por otro lado, se deben implementar los *listeners* de los botones que permiten añadir ROMs y entrar a la pantalla de ajustes. Al mismo tiempo que configuramos el `RecyclerView`, enlazaremos los dos botones definidos en el layout con dos *launchers*.

El launcher del botón de añadir, comprobará primero si el fichero ya existe en la memoria interna mediante la función `RomManagement.fileAlreadyExists(context: Context, uri: Uri)`. En caso negativo, copiará y añadirá el fichero a la memoria privada de la aplicación y, si esta operación tiene éxito, la añadirá también al *data source* y a la base de datos. Por último, notificará al *adapter* de que un nuevo objeto se ha insertado para que la vista se actualice correctamente.

El launcher del botón de ajustes simplemente generará un *intent* que delegará el trabajo en otra actividad, la cual se implementará posteriormente.

7.13.1 SQL ROM Data

Para guardar, obtener y listar todas las ROMs y sus datos, se van a generar los siguientes archivos:

- **ROM**: *Data Class* que va a contener un entero como identificador, una cadena de texto para el título y una segunda cadena de texto para la imagen.
- **ROMDataSource**: Contiene el listado de todas las ROMs en tiempo de ejecución.
- **ROMManagement**: El manejador de las ROMs, contiene toda la lógica para guardar, eliminar y obtener las ROMs de la memoria privada. Se implementará como un **singleton**.
- **ROMDao**: Un *Data Access Object* que define la interfaz SQL. Contendrá funciones para insertar, actualizar, obtener y eliminar ROMs.
- **ROMEntity**: Un *Data Class* parecido al de ROM, pero con propiedades para la ruta donde se almacena el fichero original, la ruta donde se almacene el archivo de guardado, la fecha en la que la ROM fue añadida, y el tipo (DMG o CGB).

7.13.1.1 ROM Management

A continuación se explica el `ROMManagement` más en profundidad:

La función principal es `addROM()`, que permite registrar una nueva ROM. Este método limpia el nombre del archivo recibido, eliminando extensiones y caracteres indeseados para generar un título más legible. A continuación, se crea una entidad `ROMEntity` que representa la ROM con su ruta, título, tipo y fecha de añadido. Esta entidad se guarda en la base de datos y se añade a la lista de ROMs en memoria mediante `saveROMToDatabaseAndDataSource()`.

Código 7.76: ROM Management - Añadir una ROM.

```

1   fun addROM(context: Context, romPath: String, romTitle: String) {
2
3     val title = romTitle.replace(Regex("\\.gbc?"), "").replace(Regex("_"), " ")
4     val type = if(romTitle.contains(".gbc")) "gbc" else "gb"
5
6     val newROM = ROMEntity(

```

```

7         filePath = romPath,
8         title = title,
9         imageRes = null,
10        saveFilePath = null,
11        addedDate = System.currentTimeMillis(),
12        type = type
13    )
14
15    saveROMToDatabaseAndDataSource(newROM, context)
16 }
17
18 private fun saveROMToDatabaseAndDataSource(rom: ROMEntity, context: Context
19     ↪ ) {
20     val db = SQLManager.getDatabase(context)
21     val id = db.romDAO().insertROM(rom)
22
23     ROMDataSource.addROM(ROM(title = rom.title, id = id.toInt()))
24 }
```

En cuanto a la eliminación de ROMs, la función `deleteRom()` permite borrarlas tanto de la base de datos como del almacenamiento y las preferencias asociadas. Si se elimina correctamente, también se actualiza la fuente de datos en memoria y se informa al usuario con un mensaje emergente. Internamente, este proceso utiliza `deleteROMFile()`, que se encarga de borrar físicamente los archivos relacionados y actualizar la base de datos.

Existe una clase llamada `FileManager`, que podría definirse como una capa inferior en cuanto al mantenimiento de archivos. Contiene una implementación bastante común, por lo que se va a omitir la explicación su código.

Código 7.77: ROM Management - Eliminar una ROM.

```

1  fun deleteRom(context: Context, rom: ROM, position: Int): Boolean{
2      val dao = SQLManager.getDatabase(context).romDAO()
3      val romEntity = dao.getROMByTitle(rom.title ?: "")
4      if(romEntity != null) {
5          // Delete private file and database entry
6          if(deleteROMFile(context, romEntity)) {
7
8              // Delete from Shared Preferences
9              val preferences = context.getSharedPreferences(romEntity.id. ↪
10                  ↪ toString(), Context.MODE_PRIVATE)
11              val editor = preferences.edit()
12              editor.clear()
13              editor.apply()
14
15              // Delete from Data Source
16              ROMDataSource.roms.removeAt(position)
17
18              Toast.makeText(context, "${rom.title} has been deleted", Toast. ↪
19                  ↪ LENGTH_SHORT).show()
20          }
21      }
22
23      return true
24 }
```

```

19         }
20     }
21     Toast.makeText(context, "An error has occurred", Toast.LENGTH_SHORT).  
    ↪ show()
22     return false
23 }
24
25 private fun deleteROMFile(context: Context, rom: ROMEntity?) : Boolean {
26     val romDao = SQLManager.getDatabase(context).romDAO()
27
28     // Delete file from private storage
29     if(rom != null) {
30         val filepath = rom.filePath
31         val imagePath = rom.imageRes
32
33         if (filepath.isNotEmpty()) {
34             val file = File(filepath)
35             if (file.exists()) {
36                 file.delete()
37             }
38         }
39         // Delete from DB
40         romDao.deleteROM(rom)
41
42         // Delete game cover if needed
43         if(imagePath != null)
44             FileManager.deleteFileFromPrivateStorage(imagePath)
45
46         return true
47     }
48
49     return false
50 }
```

Por último, la función `loadROMSFromDBIfNeeded()` sincroniza la fuente de datos en memoria con el contenido de la base de datos. Si el número de ROMs en memoria no coincide con el de la base de datos, las recarga todas, teniendo en cuenta también posibles valores almacenados en `SharedPreferences`, como portadas personalizadas o nombres editados.

Código 7.78: ROM Management - Recarga de ROMs en Memoria.

```

1  fun loadROMSFromDBIfNeeded(context: Context){
2      val romDao = SQLManager.getDatabase(context).romDAO()
3      val allRoms = romDao.getAllROMs()
4      if(ROMDataSource.roms.size != allRoms.size){
5          ROMDataSource.roms.clear()
6          allRoms.forEach {
7              val preferences = context.getSharedPreferences(it.id.toString(),  
    ↪ Context.MODE_PRIVATE)
8              val prefTitle = preferences.getString(TITLE_KEY, it.title)
9              val prefImage = preferences.getString(COVER_KEY, it.imageRes)
10             ROMDataSource.addROM(ROM(id = it.id, title = prefTitle, imageRes ↪
```

```

11         ↪ = prefImage))
12     }
13 }
```

7.13.2 ROM Adapter

Como se ha explicado ya, el `ROMAdapter` es el encargado de gestionar la forma en que se muestran las ROMs en el `RecyclerView`. Este adaptador extiende de `RecyclerView.Adapter` y se inicializa con una lista de ROMs y el número de columnas que se van a mostrar.

Contiene una clase interna llamada `ViewHolder`, que se encarga de gestionar la vista de cada elemento. Esta clase contiene una referencia a la vista de la ROM y a su imagen, así como los métodos para enlazar los datos de la ROM con la vista.

A cada ROM, por defecto, se le asigna una imagen predeterminada que se encuentra en la carpeta de recursos. Esta imagen se puede cambiar posteriormente a través de un *dialog* que permite al usuario seleccionar una imagen de su galería.

A cada view se le hace el inflado mediante un layout que define cómo representar cada objeto de forma individual. Este layout se encuentra en la carpeta de recursos y contiene un `ImageView` y un `TextView` para mostrar la imagen y el título de la ROM, respectivamente. El layout se infla en el método `onCreateViewHolder()` del adaptador.

Código 7.79: ROM Adapter.

```

1  class ROMAdapter(val romList: MutableList<ROM>, private val spanCount: Int) ←
2      ↪ :
3      RecyclerView.Adapter<ROMAdapter.ViewHolder?>() {
4
5      [...]
6
6  override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ←
7      ↪ ViewHolder {
8      val v: View = LayoutInflater.from(parent.context).inflate(R.layout.←
9          ↪ rom_grid_item, parent, false)
10
10 [...]
11
11 class ViewHolder(v: View) : RecyclerView.ViewHolder(v) {
12     private var name: TextView
13     private var icon: ImageView
14     private val view = v
15
16     fun bind(it: ROM) {
17         name.text = it.title
18
19         icon.setImageResource(R.drawable.rom_icon)
20 }
```

```

21         if(it.imageRes != null) {
22             val file = File(it.imageRes!!)
23             if (file.exists())
24                 icon.setImageURI(Uri.fromFile(file))
25         }
26     }
27
28     init {
29         name = view.findViewById(R.id.rom_title)
30         icon = view.findViewById(R.id.rom_image)
31     }
32 }
```

Una vez implementados los puntos anteriores, la aplicación debe permitir añadir y visualizar las ROMs en la página principal:

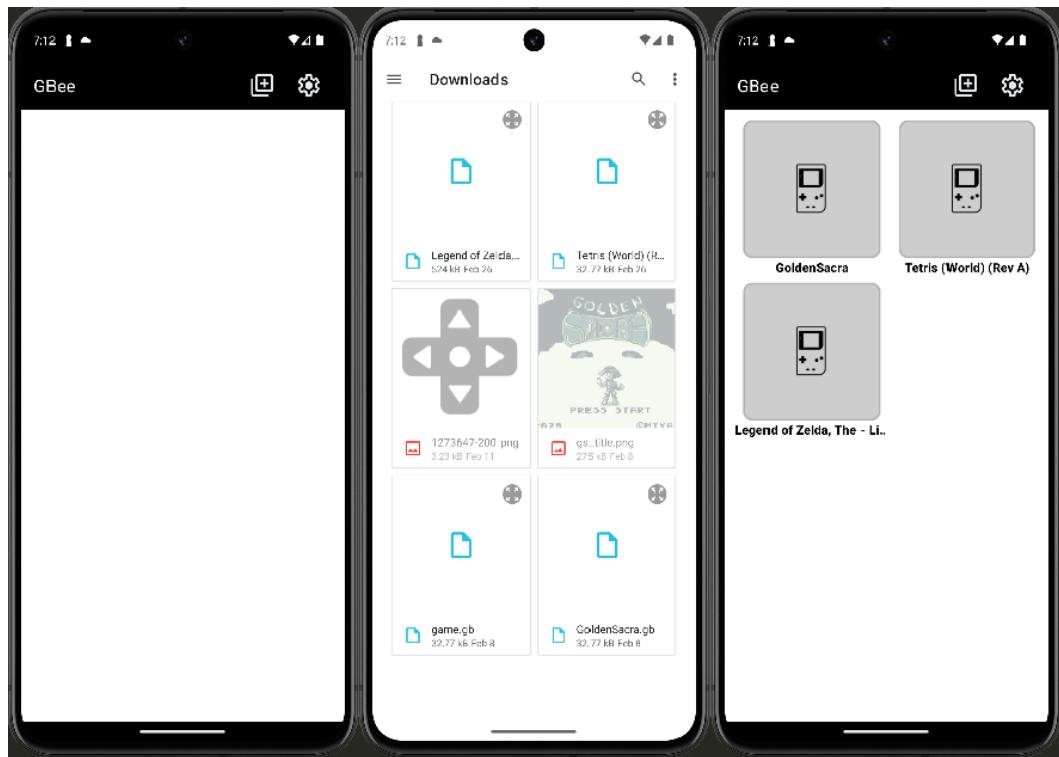


Figura 7.6: MainActivity - Añadir ROMs al Emulador.

7.14 SettingsActivity

La actividad de configuración es la encargada de gestionar la configuración del emulador. Esta actividad se lanza desde el `MainActivity` y permite al usuario modificar distintas opciones del emulador, como la paleta de colores o modificar la interfaz de juego. Implementa la interfaz `OnPreferenceStartFragmentCallback`, lo que permite mostrar fragmentos secundarios de configuración según las interacciones del usuario.

En el método `onCreate()` se incluye una barra de herramientas y un contenedor de fragmentos. La barra se configura para permitir la navegación hacia atrás. Por otro lado, se obtiene el valor del identificador de juego a través del *intent* que ha lanzado la actividad anterior. Este identificador determina si se deben mostrar las preferencias asociadas a un juego específico (`GameSettingsFragment`) o las preferencias generales de la aplicación (`SettingsFragment`). Si el identificador es distinto de -1, se interpreta que la configuración está vinculada a un juego concreto.

Código 7.80: SettingsActivity - Inicialización.

```

1   class SettingsActivity : AppCompatActivity(), PreferenceFragmentCompat.OnPreferenceStartFragmentCallback { ↪
2       ↪ OnPreferenceStartFragmentCallback {
3
4       private lateinit var binding: ActivitySettingsBinding
5       private var gameId = -1
6
7       override fun onCreate(savedInstanceState: Bundle?) {
8           super.onCreate(savedInstanceState)
9
10          binding = ActivitySettingsBinding.inflate(layoutInflater)
11          setContentView(binding.root)
12          setSupportActionBar(binding.toolbar)
13
14          supportActionBar?.apply {
15              setDisplayHomeAsUpEnabled(true)
16          }
17
18          gameId = intent.getIntExtra(GAME_ID, -1)
19
20          if (gameId != -1) {
21              supportFragmentManager
22                  .beginTransaction()
23                  .replace(R.id.fragment_container, GameSettingsFragment(gameId))
24                  .commit()
25          } else if (savedInstanceState == null) {
26              supportFragmentManager
27                  .beginTransaction()
28                  .replace(R.id.fragment_container, SettingsFragment())
29                  .commit()
30
31          [...]

```

La función `onPreferenceStartFragment()` se ejecuta cuando el usuario selecciona una opción de configuración que abre un nuevo fragmento. En este método se instancia dinámicamente el nuevo fragmento a partir de la clase especificada en la preferencia. Además, se le pasan los extras necesarios, como el identificador visto previamente. El fragmento se añade a la pila de retroceso para que el usuario pueda navegar hacia atrás de forma natural.

Código 7.81: SettingsActivity - Selección de Otro Fragmento.

```

1  override fun onPreferenceStartFragment(
2      caller: PreferenceFragmentCompat,
3      pref: Preference
4  ): Boolean {
5      val fragment = supportFragmentManager.fragmentFactory.instantiate(
6          classLoader,
7          pref.fragment!!
8      )
9
10     pref.extras.putInt(GAME_ID, intent.getIntExtra(GAME_ID, -1))
11
12     fragment.arguments = pref.extras
13
14     supportFragmentManager.beginTransaction()
15         .replace(R.id.fragment_container, fragment)
16         .addToBackStack(null)
17         .commit()
18
19     return true
}

```

El `SettingsFragment` que se utiliza por defecto en la actividad, utiliza un archivo XML de preferencias, donde vienen definidos y enlazados con sus respectivas clases todos los sub-fragmentos a los que se puede navegar. Además, utilizar fragmentos ofrece la posibilidad de añadir toda la lógica que se necesite según qué situación.

Código 7.82: SettingsActivity - Archivo de Preferencias.

```

1  <PreferenceScreen
2      xmlns:android="http://schemas.android.com/apk/res/android">
3
4      <Preference
5          android:key="video_settings"
6          android:title="Video"
7          android:icon="@drawable/video_settings_vector"
8          android:summary="Adjust video settings"
9          android:fragment="es.atm.gbee.core.fragments.VideoSettingsFragment" ↵
10         ↵ />
11
12      <Preference
13          android:key="audio_settings"
14          android:title="Audio"
15          android:icon="@drawable/music_note_vector"
16          android:summary="Adjust audio settings"
17          android:fragment="es.atm.gbee.core.fragments.AudioSettingsFragment" ↵
18         ↵ />
19
20      [...]

```

Por último, se ha implementado el método `onSupportNavigateUp()`, el cual se encarga de devolver el control a la actividad anterior cuando el usuario pulsa el botón de retroceso en la barra de herramientas. Se crea un *intent* con el identificador de juego actual para devolverlo

como resultado, y se lanza `onBackPressedDispatcher` para que la navegación hacia atrás se gestione correctamente.

El resultado es el que se puede ver en las siguientes figuras:

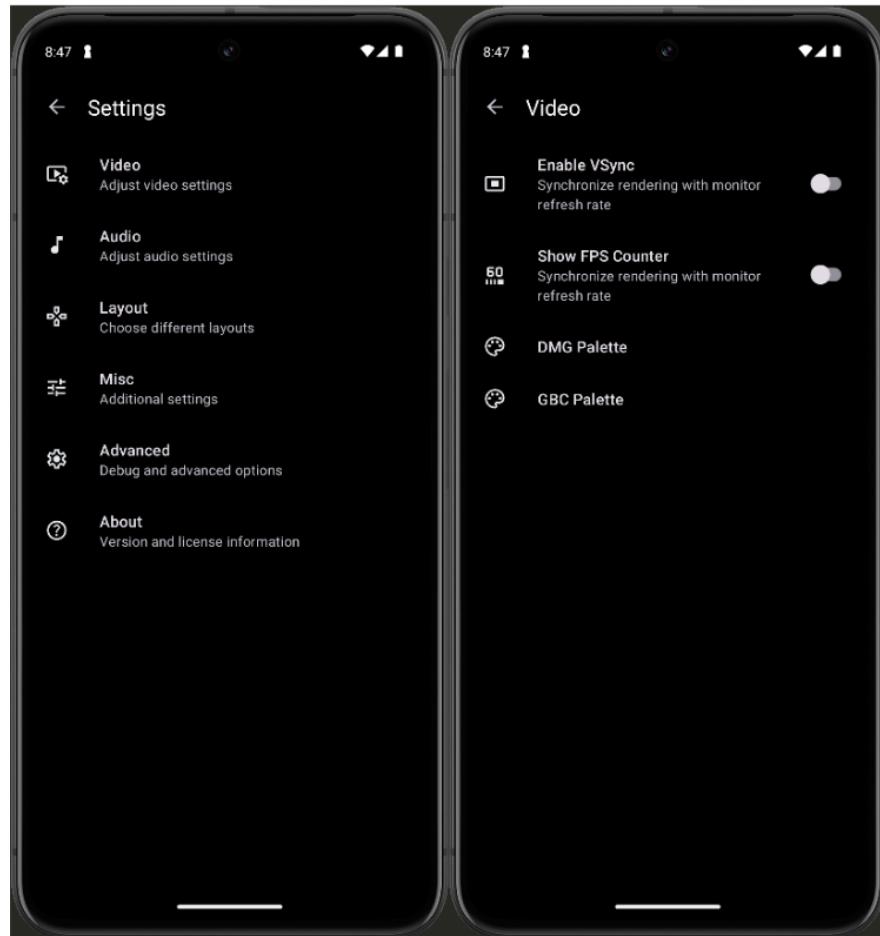


Figura 7.7: Transición entre Fragmentos en la Pantalla de Ajustes.

Los ajustes que el usuario tome, tanto de forma general como de forma individual por cada juego, se guardarán mediante *SharedPreferences*. Existen excepciones como la carátula que el usuario quiera configurar a cada juego, que se deben guardar en memoria privada.

A continuación se muestra un ejemplo en el que el usuario modifica la portada de un juego:

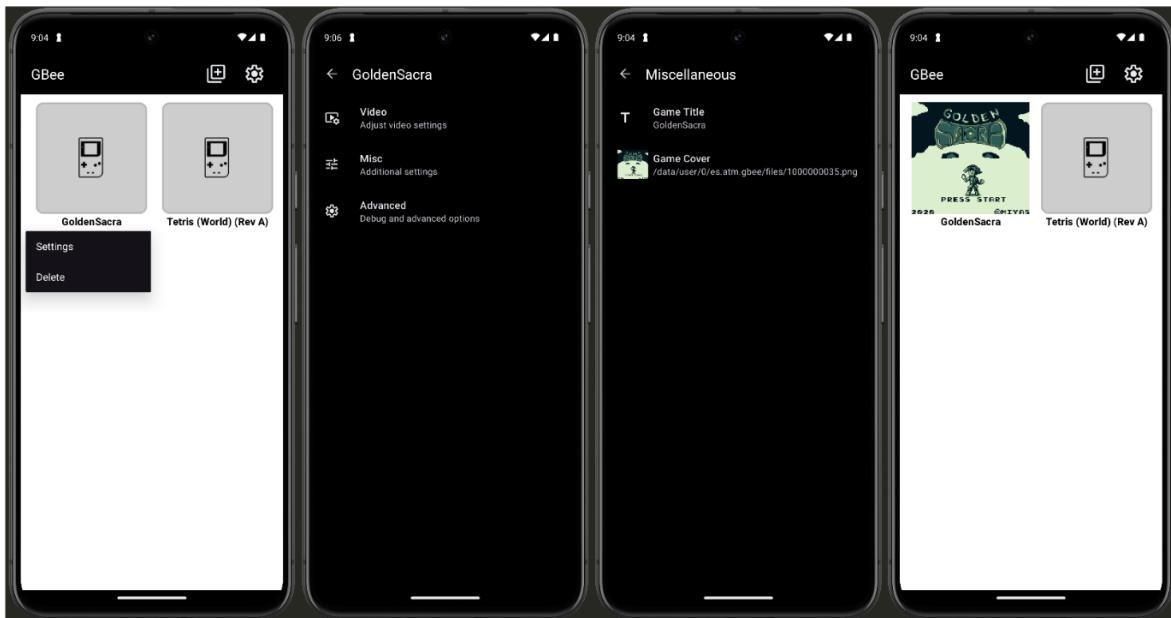


Figura 7.8: Modificación de la Portada de un Juego.

7.14.1 Skins

El emulador permite al usuario cambiar la apariencia de la interfaz de juego. Para ello, se ha implementado un sistema de *skins* que permite personalizar la apariencia del emulador. Cada skin se define mediante una entidad SQL que contiene los colores, imágenes y estilos de los distintos elementos de la interfaz.

El sistema de *skins* se basa en el uso de `SharedPreferences` para almacenar la configuración de la skin seleccionada por el usuario. Al iniciar el emulador, se carga la skin seleccionada y se aplican los estilos correspondientes a los distintos elementos de la interfaz.

De la misma manera que se ha hecho con la página principal que muestra todas las ROMs, se ha generado las siguientes clases y estructuras:

- **Skin:** *Data Class* que contiene un entero como identificador, una cadena de texto para el título, un color para el fondo, y un array de bytes para cada uno de los botones.
- **SkinDataSource:** Contiene el listado de todas las skins en tiempo de ejecución.
- **SkinManagement:** El manejador de las skins, contiene toda la lógica para guardar, eliminar y obtener las skins de la base de datos. Se implementa también como un **singleton**.
- **SkinDao:** Un *Data Access Object* que define la interfaz SQL. Contiene funciones para insertar, actualizar, obtener y eliminar skins.
- **SkinEntity:** Un *Data Class* muy similar al modelo de Skin.

- **SkinAdapter:** Un adaptador para el `RecyclerView` que muestra las skins disponibles. Contiene la lógica de cómo se van a mostrar las skins en pantalla.
- **Skin Item:** Un layout para cada elemento del `RecyclerView`, que contendrá una imagen y el nombre de la skin.
- **CustomSkinsActivity:** Una actividad que permite al usuario seleccionar una skin personalizada. Contiene un `RecyclerView` que muestra todas las skins disponibles y permite al usuario seleccionar una de ellas.
- **CreateCustomSkinActivity:** Una actividad que permite al usuario crear una skin personalizada. Muestra un layout por defecto en el que el usuario puede seleccionar los colores y las imágenes de los distintos elementos de la interfaz.

La diferencia con el `ROMAdapter` es que no se va a utilizar un `GridLayoutManager`, ya que se van a listar en forma de lista. Además, siempre aparecerá una skin por defecto, que vendrá preconfigurada con la aplicación y que los usuarios pueden utilizar pero no eliminar ni editar.

La lógica del manejador de skins es muy similar a la de las ROMs, con la diferencia de que no se están manejando ficheros, solamente entidades SQL. Por otro lado, la configuración del `RecyclerView` y los *listeners* también sigue el mismo patrón. Por ello, vamos a obviar la parte de código para evitar redundancias.

El emulador comenzará utilizando la skin predeterminada, pero el usuario podrá cambiarla en cualquier momento tanto por otra skin que genere como por el diseño original que utiliza botones comunes de Android.

7.14.2 Diseño de Skin Predeterminada

Para la skin predeterminada se ha optado por un diseño sencillo, que no sature al usuario y que permita una buena visibilidad de los elementos. Se ha utilizado un fondo amarillo con un color negro para los botones. Dar las gracias a Carla Maciá Díez por la ayuda proporcionada con todos los elementos. A continuación se muestra el diseño de todos los elementos de la skin predeterminada:

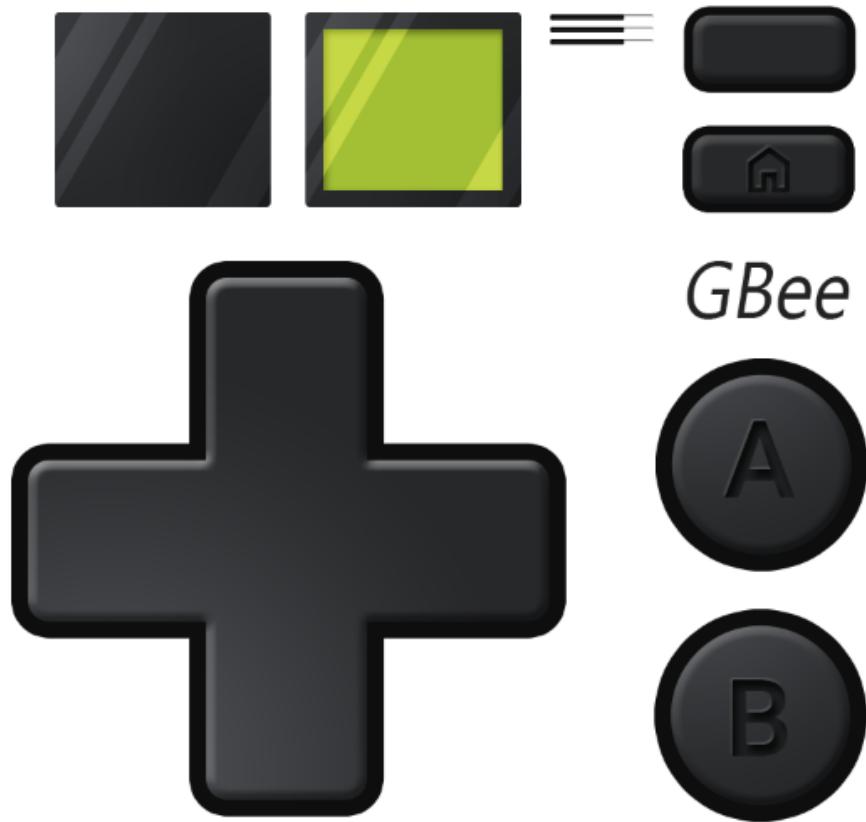


Figura 7.9: Nuevos Diseños para los Elementos de la Interfaz de Juego.

Bibliografía

- Antonio Niño Díaz. (2015). *The Cycle-Accurate Game Boy Docs.* <https://raw.githubusercontent.com/geaz/emu-gameboy/master/docs/The%20Cycle-Accurate%20Game%20Boy%20Docs.pdf>.
- Bulbapedia. (2024). *Game Link Cable.* https://bulbapedia.bulbagarden.net/wiki/Game_Link_Cable.
- Computing History. (2013). *Nintendo Game Boy.* <http://wwwcomputinghistory.org.uk/det/4033/Nintendo-Game-Boy/>.
- Gekkio. (2024). *Game Boy hardware database.* <https://gbhwdb.gekkio.fi/>.
- Hacktix, Brian Jia. (2023). *GBEDG - Gameboy Emulator Development Guide.* <https://hacktix.github.io/GBEDG/>.
- Low Level Devel. (2021). *Gameboy Emulator Development.* https://www.youtube.com/watch?v=e87qKixKFME&list=PLVxiWMqQvhg_yk4qy2cSC3457wZJga_e5&index=2.
- Marat Fayzullin, P. R., Pascal Felber, y Korth, M. (1998). *Pan Docs.* <http://bgb.bircd.org/pandocs.htm>.
- Marc Rawer. (1999). *Game Boy CPU Manual.* <http://marc.rawer.de/Gameboy/Docs/GBCPUman.pdf>.
- New Pan Docs.* (2020). <https://gbdev.io/pandocs/Specifications.html>.
- Nintendo. (1999). *Manual Oficial de Programación para Game Boy.*
- nitro2k01. (2008). *Game Boy Development Wiki.* https://gbdev.gg8.se/wiki/articles/Main_Page.
- Pastraiser. (s.f.). *Gameboy CPU (LR35902) instruction set.* <http://marc.rawer.de/Gameboy/Docs/GBCPUman.pdf>.
- Peter Johnson. (2022). *C Game Boy Emulator.* <https://github.com/zid/gameboy>.
- Ryan Levick. (2020). *DMG-01: How to Emulate a Game Boy.* <https://rylev.github.io/DMG-01/public/book/introduction.html>.
- System of Levers. (2023). *The Window Layer - How GameBoy Graphics Work.* <https://www.youtube.com/watch?v=8TVgN16DrEU>.
- Wikiguru. (2007). *z80 Heaven.* <http://z80-heaven.wikidot.com/>.

Wikipedia. (2023). *Serial Peripheral Interface*. https://es.wikipedia.org/wiki/Serial_Peripheral_Interface.

Ángel Jesús Terol Martínez. (2020). *Golden Sacra - Videojuego en ensamblador para Game Boy*. <https://rua.ua.es/dspace/handle/10045/109317>.