Treball final de grau

Emulador GBC
Aprendre i jugar en la seva màxima expressió

Aleix Abengochea Molar

Director
Bernardino Casas Fernandez

Titulació
Grau en enginyeria informàtica

Departament
CS - Departament de Ciències de la Computació

20 d'octubre de 2021

Resum

Les videoconsoles antigues són una font perfecta d'informació i coneixement. S'han originat comunitats enormes al seu voltant, donant pas a un ecosistema que permet l'estudi de diferents camps de la computació amb un objectiu definit i sobretot interessant.

En aquest projecte, ens endinsem de ple en entendre el funcionament dels mecanismes interns d'un computador simple. Com tractar la programació a baix nivell des d'un llenguatge de programació modern i com dissenyar una aplicació que porti a la pràctica tots aquests conceptes.

L'objectiu d'aquest treball és aprendre com funcionen tots i cadascun dels components de la Gameboy de Nintendo per tal de dissenyar i implementar un emulador capaç d'executar alguns jocs.

Paraules clau

Emulador, Gameboy, Java, Unitat de gestió de memòria, Unitat de processament de píxels, CPU

Resumen

Las videoconsolas antiguas son una fuente perfecta de información y conocimiento. Se han originado comunidades enormes a su alrededor, dando paso a un ecosistema que permite el estudio de diferentes campos de la computación con un objetivo definido y sobre todo interesante.

En este proyecto, nos adentramos de lleno en entender el funcionamiento de los mecanismos internos de un computador simple. Como tratar la programación a bajo nivel desde un lenguaje de programación moderno y cómo diseñar una aplicación que lleve a la práctica todos estos conceptos.

El objetivo de este trabajo es aprender cómo funcionan todos y cada uno de los componentes de la Gameboy de Nintendo para diseñar e implementar un emulador capaz de ejecutar algunos juegos.

Palabras clave

Emulador, Gameboy, Java, Unidad de gestión de memoria, Unidad de procesamiento de píxeles, CPU

Abstract

Old video game consoles are an excellent source of information and knowledge. They allow the study of different fields of computing with a specific and interesting goal.

In this project, we delve fully into following the workings of the internal mechanisms of an ordinary computer. How to deal with low-level programming from a modern programming language and how to design an application that puts all these concepts into practice.

The objective of this work is to learn how each and every component of the Nintendo Gameboy works to design and implement an emulator capable of running some games.

Keywords

Emulador, Gameboy, Java, Memory Managment Unit, Pixel Processing Unit, CPU

Agraïments

En primer lloc m'agradaria agrair al professor Bernardino Casas per oferir-me l'oportunitat de fer aquest treball. Sense la seva dedicació i atenció aquest treball no hauria estat possible.

També vull agrair a la meva parella pel fet de recolzar-me i motivar-me en tot moment, als meus familiars per no perdre l'esperança i al meu pare al qual li hagués encantat el resultat.

Per últim incloure a tota la comunitat que recull i comparteix informació de manera altruista i fa que projectes com aquest siguin possibles.

Índex

Índex de figures

Índex de taules

Glossari

- **Apk:** Aplicació per al sistema operatiu Android.
- **Background:** Fons de la pantalla.
- **Clock:** Rellotge intern en l'àmbit de hardware. Normalment un tros de quars.
- **Cart:** Cartutx que conté la ROM del joc.
- **Cartridge:** Cart o cartutx.
- **CPU:** Central Processing Unit. Part principal de qualsevol computador.
- **Display:** Pantalla de la videoconsola.
- **Debugger:** Depurador. Eina que permet provar i trobar errors.
- **DMA:** Direct Memory Address.
- **DMG:** Nom intern que va rebre la versió de la Gameboy.
- **EDG:** Fil principal on s'executa els elements de la llibreria Swing.
- **Flags:** Indicadors del processador.
- **Framebuffer:** Búfer de memòria que conté les dades que representen tots els píxels d'un fotograma.
- **Factory:** Patró de disseny aplicat en la programació orientada a objectes.
- **Int:** Tipus de variable entera. En el context d'aquest treball variable de 32 bits.
- **I/O:** Entrada i sortida del programa.
- **Java:** Llenguatge de programació utilitzat en aquest treball.
- **Jump o jmp:** Instrucció de salt.
- **Joypad:** Controls d'usuari de la Gameboy.
- **LCDC:** LCD Control. Registre que permet configurar la LCD i la PPU.

- **LCDS:** LCD State. Registre que permet obtenir l'estat de la LCD i la PPU.
- **MBC:** Memory bank controller. Component que s'encarrega de gestionar tot l'espai adreçable del cartridge.
- **MMU:** Memory managment unit. Component que s'encarrega de gestionar tot l'espai adreçable.
- **Mockup:** Paraula que s'utilitza per definir un esbòs sobre una interficie d'usuari.
- **OAM:** Object Address Memory. Regió de memòria on es guarden els sprites.
- **Offset:** Desplaçament. En aquest text normalment es refereix a bits.
- **OOP:** Object Oriented Programming.
- **Overhead:** Computació extra que provoquen algunes operacions.
- **PPU:** Pixel Processing Unit. Component que s'encarrega de renderitzar els píxels per pantalla.
- **RAM:** Memòria volàtil.
- **Refactor:** Acció de repassar i millorar una part del codi.
- **ROM:** Memòria persistent, no modificable.
- **Short:** Tipus de variable entera. En el context d'aquest treball variable de 16 bits.
- **Singleton:** Patró de disseny aplicat en la programació orientada a objectes.
- **Sprites:** Objectes gràfics que modifiquen sovint la seva posició. Normalment són els jugadors, enemics, etc.
- **Swing:** Llibreria utilitzada al llenguatge Java per tal de crear interfícies d'usuari.
- **Threads:** Fil de processador.
- **Timer:** Comptador intern que té el processador per tal de sincronitzar processos.
- **Tile:** 8x8 píxels que formen una casella.
- **Tile Map:** Conjunt de tiles que normalment componen el background o el window.
- **Tile Set:** Conjunt de tiles que s'utilitzen per generar el tile map o els sprites.
- **UML:** Tipus de diagrama que representa l'arquitectura del codi.

- **Vertical Blank:** Moment en el qual s'ha acabat de renderitzar totes les línies horitzontals d'un frame.
- **Window:** Finestra fixa per sobre del background.

Capítol 1

Introducció

Aprendre a multiplicar números sense saber sumar pot ser útil en molts aspectes, però mai et portarà a plantejar-te certes coses o seràs incapaç de donar resposta a moltes preguntes. Crec amb molta convicció, però sense gaires proves que entendre la part més baixa de la piràmide que estudiem és fonamental per donar un fil lògic i explicació a tot el que creem.

Els computadors com a tal s'han complicat de manera exponencial amb el pas del temps. Tenen múltiples etapes, diferents nivells de memòria cau i fins i tot predictors de salt.

A mesura que evoluciona la tecnologia, tenir una idea general del funcionament de les coses és cada vegada més complicat. Per aquest motiu, penso que estudiar i recrear el comportament físic de la Gameboy es troba en el punt d'equilibri perfecte entre l'enteniment i la complexitat de molts conceptes.

La Gameboy ofereix un escenari de partida perfecte per tal d'assentar una base de com funcionen els microprocessadors en general. Té molt bona documentació, una comunitat sòlida, i fins i tot més versions que poden anar complicant i complementant les primeres.

1.1 Motivació

En l'àmbit personal, un dels camps de la informàtica que sempre m'ha interessat és la part lògica i programació a baix nivell. Quan vaig ser alumne de la Universitat Rovira i Virgili, vaig aprendre ARM a través de la Nintendo Ds (NDs). Això va fer créixer la meva motivació, ja que era el primer cop que veia una aplicació molt pràctica i real d'un àmbit que m'agradava.

A part de tot això, la Gameboy probablement va ser el primer component electrònic que vaig tenir a les mans i un dels que segurament li he dedicat més hores. Guardo bons records i d'alguna manera aquí es tanca el cicle.

1.2 Objectius

L'objectiu d'aquest treball és modelar i emular el comportament dels components interns de la Gameboy amb un llenguatge de programació modern. El qual ofereixi eines per entendre millor com funcionen les diferents parts.

També es vol donar a conèixer quins han estat els principals reptes i problemes que ha suposat endinsar-se en aquest món igual que intentar ajudar a qualsevol persona que ho estigui intentant.

Els objectius o fites inicials de cara a la implementació es poden definir com:

- **Prova de concepte:** Part fonamental per tal de poder mesurar quin és l'abast real del projecte.
- **Components bàsics:** Emular tots els components crítics per al funcionament bàsic de l'emulador.
- **Gameboy funcional:** Fita important del projecte on es posen en comú tots els components i s'aconsegueix un prototip inicial.
- **Gameboy Color:** Aconseguir fer funcionar un joc de la Gameboy Color a l'emulador.
- **Debugger:** Implementar eines que permetin als usuaris ser conscients que està passant en tot moment dins l'emulador. Si fos possible, permetre canviar informació de les memòries en temps d'execució així com guardar-la o comparar-la.

1.3 Grups d'interès

Aquest projecte va enfocat a totes aquelles persones que tinguin interès en les relíquies del passat, sobretot a aquelles que tinguin intenció d'implementar una versió emulada d'alguna d'elles.

La informació que es troba en aquest treball no dicta com s'ha de fer. Dona una visió general de quines idees s'han utilitzat per tal d'arribar a aconseguir-ho i quins han estat els reptes o problemes principals que es poden interpretar com a consell.

1.4 Estructura de la memòria

Aquest treball es divideix en dues parts principals. La primera posa en context com funciona la GameBoy i la segona explica quina ha estat la implementació que s'ha portat a terme.

Dins la implementació s'explica com s'ha pensat i programat cada part d'una manera més lògica que tècnica. Això simplifica la lectura, deixa de banda infinites parts del treball que haurien de portar codi i en general millora l'experiència. El codi font de

l'aplicació que s'entrega amb aquest treball està àmpliament documentat per si algun lector vol anar un pas més enllà.

Per acabar, es defineixen quins han estat els problemes principals del projecte, quin ha estat el seu progrés i un exemple de pressupost.

1.5 Eines

El llenguatge principal utilitzat per a implementar el projecte ha estat Java i s'ha utilitzat la llibreria Swing per a la creació de la interfície d'usuari. Els dos són open-source i multiplataforma per la qual cosa no hi haurà problemes utilitzant el programa en els diferents sistemes operatius més utilitzats (Windows, Linux i Mac).

Per tal de portar control sobre el desenvolupament i els canvis en el codi del projecte s'ha utilitzat Git.

Capítol 2

Planificació inicial i Pressupost

2.1 Planificació inicial

Per tal de tenir una idea i un fil conductor del projecte, s'ha portat a terme una primera planificació inicial.

Aquesta planificació s'ha basat en el nombre d'hores que normalment és dediquen a fer el TFG i a una primera estimació molt bàsica de quina feina comportaria el projecte. El diagrama de Gantt es pot veure a la figura 2.1 i a la figura 2.2.

- **Llegir/Planificació:** Inici del projecte. Començar a llegir informació i fer una primera recerca sobre quant de temps pot portar el projecte.
- **Documentació/Mapa conceptual:** Seguir amb la documentació, començar a fer diagrames sobre com es pot portar a terme la implementació.
- **Modelat en classes:** Definir un UML més concret sobre el problema.
- **Implementació:** Implementar l'UML en codi.
- **Afegir Swing:** Un cop acabada la implementació de la part lògica, afegir la interfície d'usuari utilitzant alguna llibreria.
- **Joc de proves:** Realitzar diferents proves per tal de comprovar el correcte funcionament de la implementació.
- **Redacció:** Redactar el TFG.
- **Revisió/Imprevistos:** Temps extra per tal de dedicar més o menys temps a cada apartat.

Diagrama de Gantt inicial

Inici projecte	1/3/21
Entrega projecte	14/6/21
Total Hores (6h dia, 5d setmana)	450

Tasques	Inici	Durada(Dies)	Done
Llegir/Planificació	1-mar.	5	8/3/21
Documentació / Mapa C	8-mar.	5	15/3/21
Modelat en classes	15-mar.	5	22/3/21
Implementació	22-mar.	30	3/5/21
Afegir swing/qt/windov	3-may.	5	10/5/21
Joc de proves	10-may.	5	17/5/21
Redacció	17-may.	10	31/5/21
Revisió/Imprevistos	31-may.	10	14/6/21

Figura 2.1: Diagrama de Gantt inicial

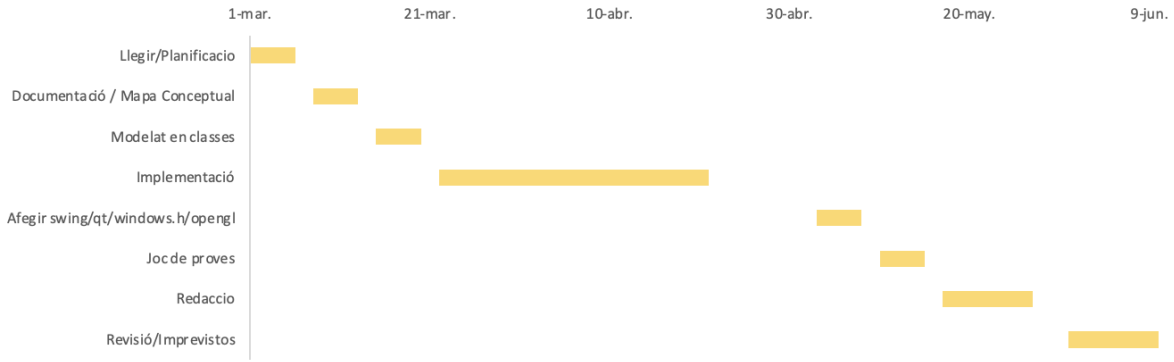

Figura 2.2: Línia temporal inicial

2.2 Pressupost

En aquest apartat tindrem en compte quins són els costos del projecte per tal de tenir una aproximació sobre el preu que costaria portar a terme el projecte.

Per tal de fer una estimació més real, suposarem el següent escenari:

- És un programador autònom que treballa per a si mateix.
- Per tal de tenir un entorn de treball separat del domicili paga mensualment per un espai coworking a Barcelona.

- Totes les eines utilitzades són de codi obert com en aquest treball.

Un cop definit l'escenari, podem tenir control sobre quin tipus de costos ens tocarà amortitzar per tal de fer el projecte rendible:

- **Costos de personal:** Preu actual d'un programador Java a la ciutat de Barcelona. Fixarem un sou de 20 euros encara que pot fluctuar molt. Informació extreta de Glassdoor [1].
- **Costos de maquinari:** Aquí inclourem l'amortització de l'ordinador, el qual és l'únic element necessari per a portar a terme el projecte. S'ha utilitzat un preu base de 1000 amb l'amortització especificada per l'estat [2].
- **Costos de programari:** Totes les eines utilitzades són gratuïtes, per la qual cosa no tindrem cap cost de programari.
- **Costos generals:** Aquí inclourem tots els costos no relacionats de forma directa amb el projecte. En aquest cas hem simplificat bastant el projecte escollint un espai coworking ja que totes les despeses estan incloses. En aquest exemple: l'empresa Aureacoworking [3]. També s'haurà d'incloure la nostra generosa contribució a l'estat a través de la quota d'autònoms.
- **Costos de contingència:** Fan referència al cost extra que ens permet establir un marge econòmic de seguretat per tal de fer front a imprevistos. El cost de contingència en els projectes de programari acostuma a ser d'un 15 per cent sobre la suma dels costos de personal, material (maquinari i programari) i generals.

A la figura 2.3 podem observar quin seria el preu total que seria convenient cobrar a l'empresa per tal de ser solvents com a programador autònom. En aquest cas el pressupost inicial està marcat amb un preu de 14564 euros.

Pressupost final

Temps total (hores)		450
Preu hora (euros)		20
Quota d'autònoms (euros)		289
Coworking (euros)		149
Mesos totals invertits		2,8125
Costs de personal	Preu hora	Total
Programador	20	9000
Costos de programari		Total
		0
Costos de maquinari	Preu inicial	Total
	1000	234,375
Costos generals	Preu mensual	Total
Quota autònoms	289	812,8125
Coworking	149	419,0625
Total	438	1231,875
Total		
		10466,25
Costos de contingència	Percentatge aplicat	
	0,15	1569,9375
Cost Total	12036,1875	

	Cost	IVA	Total
Preu a facturar	12036,1875	0,21	14563,78688

Figura 2.3: Pressupost final

Capítol 3

Gameboy

La Gameboy és una videoconsola llençada al mercat l'any 1989 per la companyia Nintendo [4]. No va ser la videoconsola més innovadora de l'època ni la que tenia les millors prestacions, però en general, va acabar sent un èxit mundial [5]. Va tenir diverses versions i millores, però en aquesta documentació ens centrarem en la primera, també coneguda com a DMG.

En aquest escrit, explicarem els diferents components de la Gameboy per separat. Cal remarcar que tots aquests components realment estan compresos dins del mateix xip físic, el LR35902. Es pot observar l'esquema físic a la Figura 3.1 i l'esquema intern a la Figura 3.2.

A priori, que els components estiguin en un mateix xip pot no semblar rellevant, però més endavant dona resposta a preguntes com: per què hi ha regions de memòria accessibles en certs moments i altres no?

3.1 CPU

Quan parlem de processador, estem parlant concretament de la CPU core. Només aquella part que s'encarrega d'obtenir, descodificar i executar instruccions sense parar.

El processador que porta la Gameboy és de 8 bits. És un híbrid entre els processadors de l'època Intel8080 i el Zilog 80 [8]. En general comparteixen moltes funcions, però canvien aspectes claus de la configuració interna que fa impossible compartir codi entre ells.

El processador comparteix la memòria tant per dades com per instruccions, inclús per accedir o comunicar-se amb qualsevol altre component. Utilitza 16 bits per a adreçar la memòria, el qual permet un total de 64KB direccions útils.

3.2 Registres i Flags

En general, els registres són la memòria més ràpida que es pot trobar en qualsevol processador i fan la funció de variables temporals que permeten modificar el seu valor

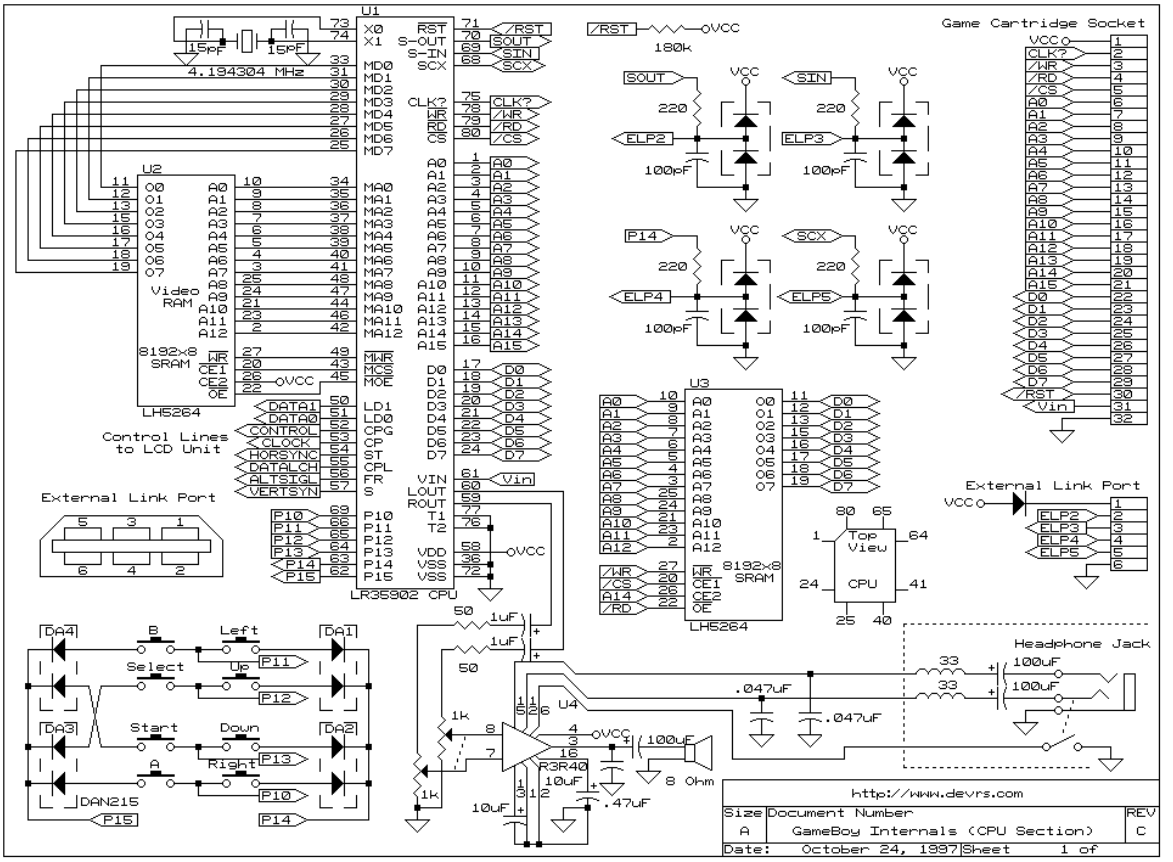

Figura 3.1: Esquema físic [6]

intern i així poder portar a terme les diferents operacions.

La Gameboy té un conjunt peculiar de registres. Té un total de 4 registres d'ús general que es poden accedir com a 1 registre de 16 bits o com a 2 registres de 8 bits. A part d'aquests 4, també disposa del PC (Program Counter) que conté la instrucció a executar i el SP (Stack Pointer) que serveix per gestionar la pila veure (Taula 3.1).

El registre F o més aviat els 8 bits baixos del registre AF, està dedicat a guardar les Flags del sistema. Les Flags són un mecanisme útil que permet saber una part del resultat d'una operació aritmètica sense necessitat d'accedir al registre o de tornar a calcular-lo. Per exemple, una de les més utilitzades, és saber si el resultat de l'operació ha estat 0. Veure (Taula 3.2)

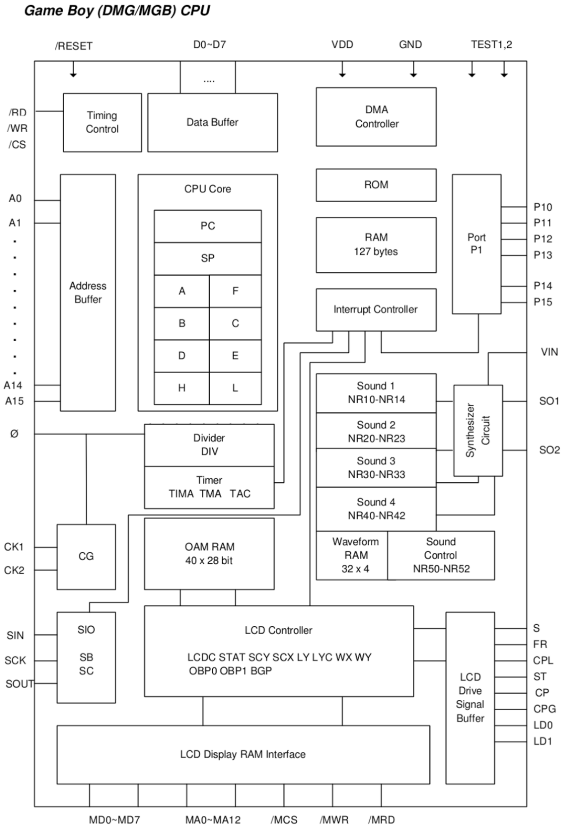

Figura 3.2: Esquema intern [7]

3.3 Operacions

Com hem comentat anteriorment, estem davant d'un processador de 8bits i això vol dir que tenim 256 possibles combinacions per tal de codificar instruccions. Això no és totalment cert, ja que la Gameboy té una instrucció (0xCB) la qual actua com a prefix i executa la següent instrucció de manera diferent. Per tant, tenim un total de 255 instruccions base i 256 esteses.

Com la majoria de processadors, podem dividir el set d'instruccions en les següents categories:

- Load/Store: El qual permet operacions tant de 8 com de 16 bits.
- Arithmetic/Logic: Les quals gairebé totes les operacions nomes es poden realitzar sobre el registre A.
- Rotate and shift: Aquestes es troben al prefix 0xCB
- Single bit: Operacions que es realitzen a nivell de bit: set, reset, test

16-bit	High	Low	Funció
AF	A	-	Acumulador i Flags
BC	B	C	BC
DE	D	E	DE
HL	H	L	HL
SP	-	-	Stack Pointer
PC	-	-	Conté la següent instrucció a executar

Taula 3.1: Registres

Bit	Nom	Funció
7	z	Serà 1 si el resultat ha estat 0.
6	n	Permet saber si la ultima operació ha estat una resta
5	h	Permet saber si la ultima operació ha tingut carry als 4 bits baixos
4	c	Permet saber si un resultat ha tingut carry o no.

Taula 3.2: Flags

- CPU control: Petit set d'instruccions que permeten la gestió del processador: nop, halt, stop, habilitar/deshabilitar interrupcions...
- Jump

3.4 Memòria

Quan ens referim a memòria, parlem sobretot de l'espai adreçable pel bus de 16 bits del processador. Dins d'aquest espai, es troben tots els components accessibles, des de la memòria ROM fins als dispositius d'entrada i sortida.

És una manera senzilla i eficaç de comunicar-se amb diferents components sense haver d'implementar alguna espècie de protocol.

Per tal de tenir més clars tots els components i com el processador accedeix a ells, definirem el mapa de memòria amb les seves direccions i funcions reals.

Podem veure com està dividint tot l'espai adreçable (Figura 3.3) i quines són les adreces amb les qual podem accedir. Veure Taula 3.3.

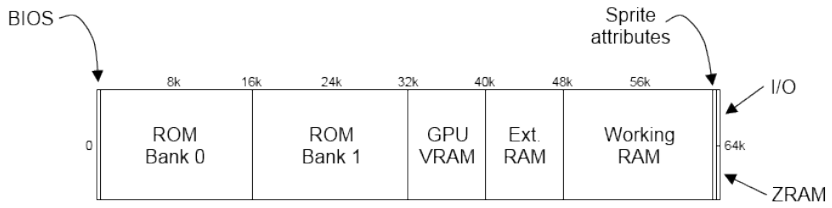

Figura 3.3: Distribució de la memòria (Imran Nazar) [9]

Rang	Descripció	Observacions
0000-3FFF	16KB ROM	Cartutx
4000-7FFF	16KB ROM	Cartutx
8000-9FFF	8KB VRAM	
A000-BFFF	8KB ExRAM	Cartutx pot tenir xip de RAM
C000-CFFF	4KB WRAM	
D000-DFFF	4KB WRAM	
E000-FDFF	Copia de la WRAM	Prohibit per Nintendo
FE00-FE9F	OAM	
FEA0-FEFF	-	Prohibit per Nintendo
FF00-FF7F	Registres I/O	
FF80-FFFE	HRAM	
FFFF-FFFF	IE	

Taula 3.3: Mapa de memòria

3.5 Gràfics

Com hem vist a l'apartat d'especificacions, la Gameboy té una pantalla de 160x144 píxels. Veure Figura 3.4 .Té una profunditat de color de 2 bits, és a dir, és capaç de reproduir 4 colors diferents, que en aquest cas es representa amb una escala de grisos.

Per tal de gestionar tots els gràfics, la videoconsola té un xip de VRAM, el qual té la mateixa capacitat que la RAM (8 Kb) i treballa al doble de la seva freqüència (2 Mhz).

Tota la informació referent a gràfics s'ha de guardar a la VRAM. Això es deu al fet que el component que s'encarrega de renderitzar els elements per pantalla (PPU) només té accés a aquesta. A excepció del DMA i la OAM.

Tant la CPU com la PPU treballen en paral·lel i això origina problemes d'accés a memòria. El processador i el PPU són capaços d'accedir a la VRAM i a la OAM així que s'han de poder sincronitzar.

En els següents apartats, descriurem el funcionament de tots els components i estructures principals i per últim els dos registres que permeten controlar i sincronitzar-se amb el processador.

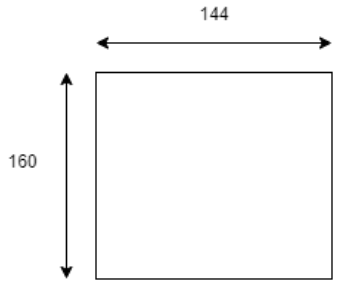

Figura 3.4: Mides LCD

3.5.1 Tiles

Com la majoria de les videoconsoles de l'època, la Gameboy no tenia la memòria suficient per tenir un framebuffer i poder mantenir tots els píxels. Per tal de reduir el pes al màxim possible i poder gestionar aquest volum de dades, s'utilitza un sistema basat en tiles. Cada una de les tiles està formada per un subconjunt de 8x8 píxels, els quals es codifiquen amb 2 bits per píxel. Veure Figura 3.5.

En general, aquests 2 bits per píxel permetran 4 colors diferents i generaran 16 bytes per cada tile.

Després es podran construir el background, el window i els sprites utilitzant referències a les diferents tiles. Això provoca que es reutilitzi molta memòria, ja que només hem guardat una vegada els 64 píxels de cada tile, però els podem referenciar tantes vegades com vulguem.

La videoconsola té assignada una regió de memòria per tal de guardar les diferents tiles que després podrem utilitzar. Aquesta àrea de la VRAM està compresa entre 8000-9700 donant un total de 6 KB d'espai. Això permet emmagatzemar un total de 384 tiles diferents.

Com ja es pot intuir, només podem adreçar 256 tiles amb 8 bits. Així que encara que sigui la mateixa regió, amb el LCD Control, explicat més endavant, podem triar a quin subconjunt fa referència el nostre índex. A la Taula 3.4 podem observar aquesta subdivisió i com s'indexa.

Subconjunt	VRAM	Idx Sprites	Idx Background/Window
0	8000-87FF	0-127	0-127 Si el 4 bit del registre LCDControl == 1
1	8800-8FFF	128-255	128-255
2	9000-97FF	No es pot utilitzar	0-127 Si el 4 bit del registre LCDControl == 0

Taula 3.4: Distribució de tiles a la VRAM

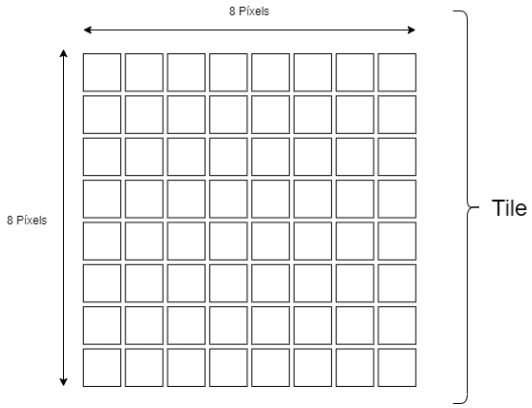

Figura 3.5: Representació del píxels formant una Tile

3.5.2 Paletes

El color de cada píxel, encara que té el mateix pes de 2bits, no està definit directament a les tiles. Això permet canviar la paleta de colors sense la necessitat de modificar tot el tilemap sencer.

En general, només es poden representar 4 colors encara que tinguem diferents paletes. El que permet canviar de paleta és bescanviar els colors per tal d'aconseguir diferents efectes interessants. Per exemple, podem alterar contínuament les paletes per tal de generar un efecte de flashing.

Els colors són 4 escales de gris. Tenint en compte el color inicial del material de la LCD, és normal representar els colors amb una tonalitat verda com es pot apreciar a la Figura 3.6.

Figura 3.6: Paleta d'exemple

3.5.3 Tile Map

Per tal de poder representar el background, la Gameboy reserva dos espais en memòria. Aquestes regions tenen una dimensió fixada de 32x32 tiles i es troben en les adreces 9800-9BFF i 9C00-9FFF respectivament.

Aquests mapes estan generats com un collage de les diferents tiles i s'indexen amb 1 byte, la qual cosa permet un total de 256 tiles diferents.

3.5.4 Background

Com es pot observar en els apartats anteriors, el Tile Map és més gran que la pantalla. El Tile Map té 32x32 tiles que són 256x256 pixels en total. En canvi, la pantalla físicament només és capaç de representar 160x144 píxels.

Aquí és on entra el concepte de background. Aquest actua com si fos un viewport i permet fer un moviment horitzontal i vertical per tal de modificar el que veiem per pantalla. Podem veure un esquema a la Figura 3.7

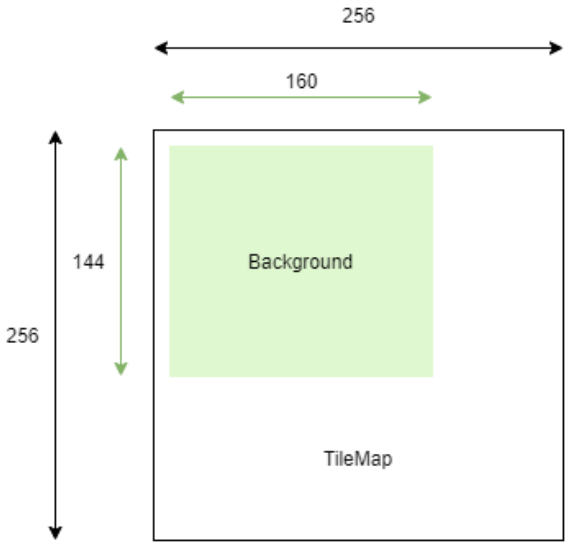

Figura 3.7: Diferència visible entre Background i TileMap

3.5.5 Window

Molts videojocs, a part de tenir un fons i uns objectes que es mouen, tenen algun tipus d'interfície d'usuari que acostuma a tenir una posició permanent. Per exemple, el Tetris té una part fixa a la pantalla i una altra on es col·loquen les peces. Molts jocs de plataformes tenen algun tipus d'icona per identificar les vides que falten, les monedes que es recullen, etc. Per tal d'implementar aquests gràfics que són fixes independentment del fons, s'implementen utilitzant el Window.

A diferència del background, el contingut no es pot desplaçar a dreta i esquerra i només es pot controlar utilitzant dos registres que indiquen la seva posició a la pantalla. Veure Taula 3.5.

Per tal de crear el panell, s'utilitza el mateix mapa i sistema que el del background. Un cop els valors dels registres estiguin dins dels límits, es mostrarà per pantalla.

Adreça	Nom	Descripció
0xFF4A	WY	Determina la Y del punt superior esquerra on començara la window, pot prendre els valors de 0-143
0xFF4B	WX	Determina la X del punt superior esquerra on començara la window, pot prendre els valors de 0-166 (Té un offset de +7)

Taula 3.5: Registres del Window

3.5.6 Sprites

Tenir un background és útil per tal de tenir un fons i elements que com a màxim es mouen en conjunt. En canvi, és molt poc pràctic a l'hora de definir petits objectes que es mouen de manera independent. Per tal de poder implementar-ho, tenim els objectes o sprites.

La Gameboy pot tenir fins a 40 sprites mostrant-se alhora per pantalla, però a causa de limitacions del hardware, només es poden mostrar 10 sprites per cada scanline. Les caselles tenen el mateix format de 8x8 que les caselles del background, però la taula que les representa es troba al 8000-8FFF. Els sprites es guarden a una part de la memòria anomenada OAM (Object Address Memory), que comprèn FE00-FE9F (160 bytes). En haver-hi 40 sprites com a màxim, a cada un li corresponen 4bytes. 40 x 4 = 160

La informació que contenen els 4 bytes de cada sprite es pot observar a la Taula 3.6.

3.5.7 DMA

El DMA o Direct Memory Acces, és una característica del processador que permet transferir dades entre components sense ser el processador l'encarregat de fer el treball. El DMA de la Gameboy s'utilitza principalment per tota la gestió dels sprites, ja que acostuma a necessitar molta informació i es modifica amb molta freqüència.

Té les següents propietats:

- Copiarà sempre un total de 160bytes (40 sprites)

<table>
<tr><th>Byte</th><th>Descripció</th></tr>
<tr><td>0</td><td>Posició X, indica la posició x de la pantalla amb un offset de 8.</td></tr>
<tr><td>1</td><td>S'incrementa a la freqüència especificada. Genera interrupció 0x50 quan passa de 255 a 0</td></tr>
<tr><td>2</td><td>És l'índex de la casella que representa el sprite a la taula 8000-8FFF. Hi ha que tenir en compte que si el LCDC està en 16x8, s'agafaran 2 tiles i no una.</td></tr>
<tr><td>3</td><td>
<table>
<tr><th>Bits</th><th>Descripció</th></tr>
<tr><td>7</td><td>Fons i finestra per sobre del OBJ (0 = no, 1=fons, 1-3 = colors de la finestra per sobre de OBJ)</td></tr>
<tr><td>6</td><td>Indica si està girada sobre les Y (0 = normal, 1=Mirall vertical)</td></tr>
<tr><td>5</td><td>Mateix que el 5 en horitzontal</td></tr>
<tr><td>4</td><td>Paleta, (0 = OBP0, 1 = OBP1)</td></tr>
<tr><td>3</td><td>Només gameboy color (Bank, la gameboy te 2 banks de VRAM es pot triar quin dels dos)</td></tr>
<tr><td>2-0</td><td>Només gameboy color (Paleta, (0-7 OBP)</td></tr>
</table>
</td></tr>
</table>

Taula 3.6: Sprites

- Podem obtenir les dades de la regió de memòria compresa entre 0000 - DFFF que és la ROM o la RAM
- Sempre es copiarà a la OAM. (FE00-FE8F)
- Single bit: Operacions que es realitzen a nivell de bit: set, reset, test
- Tardarà 160ms = 671 cicles

Un detall que s'ha de tenir en compte, és que mentre s'està executant el DMA, el processador només pot accedir a la HRAM (FF80-FFFE).

La manera d'executar el DMA és bastant senzilla i únicament hem d'escriure al registre FF46 la adreça de memòria inicial del segment de 160bytes que volem llegir. Des del moment que escrivim el registre, automàticament començarà la còpia de dades.

3.5.8 LCD Control

Un cop definit tots els components que poden tenir els gràfics, es necessita una manera de gestionar tot això. Aquí és on entra el registre FF40, el qual habilita una manera de comunicar-se amb la LCD i configurar els diferents paràmetres.

Com tots els registres de la Gameboy, es tracta de 8 bits i les diferents responsabilitats es poden observar a la Taula 3.7.

Bit	Descripció
7	Habilitar/deshabilitar tant la LCD com la PPU de manera conjunta
6	Indicar quin dels dos Tile Map s'utilitzara al Window
5	Habilitar/deshabilitar que aparegui el Window a pantalla
4	Indicar quina regió de tiles utilitzarem pel Background i el Window
3	Indicar quin dels dos Tile Map s'utilitza al Background
2	Indicar quin format utilitzarem per als sprites. 8x8 o 16x8
1	Habilitar/deshabilitar la renderització dels sprites
0	Habilitar/deshabilitar el només renderitzar sprites.

Taula 3.7: Control de la LCD i PPU

3.5.9 LCD Status

Com hem vist en l'apartat anterior, gràcies al registre LCDC podem modificar la configuració de la LCD i de la PPU. Per tal de poder sincronitzar la CPU amb la PPU és necessari saber que està fent en cada moment.

És important per al processador saber en quin estat es troba la PPU en un moment concret, ja que depenen de com estigui no podrem accedir a certes zones de la memòria.

Per tal d'entendre millor com funcionen els estats, definirem una mica com es renderitzen els píxels: La Gameboy segueix el mateix esquema que els televisors antics. No renderitzen tots els píxels de cop, sinó que renderitzen línia a línia fins a generar tota la pantalla. Comença al píxel superior esquerre de la pantalla, renderitza la línia de manera horitzontal i repeteix amb les següents 144 línies. Això genera 4 estats possibles:

- **Fase 2:** Llegir OAM per calcular si hi ha sprites per pintar en aquesta línia.
- **Fase 3:** Llegir VRAM i OAM per tal de dibuixar la línia.
- **Fase 0:** Final de línia horitzontal.
- **Fase 1:** Final de línia vertical.

Les fases 2,3 i 0 es repeteixen 144 cops sempre en el mateix ordre. Un cop finalitzades totes les línies es produeix la fase 1, el vertical blank. És important saber que en la fase

2, el processador no pot accedir a la OAM perquè s'està llegint per tal de calcular els sprites. Tanmateix, a la fase 3 no es pot llegir ni la OAM ni la VRAM, ja que la PPU les està llegint per tal de generar la imatge.

Es poden utilitzar tant el temps de fase 0 com el temps de fase 1 per modificar la memòria de vídeo, però la majoria dels videojocs utilitzen el vertical blank perquè té una durada fixa i superior a les altres.

Fase	Durada	Memòria Accessible
2	20 cicles	VRAM
3	42-72 cicles	-
0	21-52 cicles	VRAM, OAM
1	1140 cicles	VRAM, OAM

Taula 3.8: Durada de les diferents fases de la PPU

Per a no ser l'excepció, per tal de saber l'estat de la PPU haurem de llegir el registre 0xFF41. Hi ha dues maneres de consultar l'estat: de manera activa, preguntat directament cada cop que volem saber com es troba, o habilitant una interrupció per tal que ens avisi quan s'ha entrat a l'estat que volem.

A la Taula 3.9 podem observar el propòsit dels diferents bits del registre.

3.6 I/O

3.6.1 Interrupts

Les interrupcions de qualsevol processador són una part crucial que molts cops passa desapercebuda. Permeten trencar el fil d'execució quan algun factor normalment extern vol alterar el comportament seqüencial del programa. Per tal de poder portar a terme els interrupts hi ha aspectes de software i hardware a tenir en compte:

Pel que fa a hardware, el processador ha de deixar d'executar el que estava fent en el moment que rep una interrupció i posar-se a executar el ïnterrupt handler".

El processador comprova a cada cicle si ha arribat algun senyal d'interrupció. Si ha arribat alguna, inicia una funció per tal de gestionar quina interrupció s'ha activat i que cal fer.

En la majoria de processadors moderns, el interrupt handler o manegador d'interrupcions, s'implementa com un vector. Aquest vector conté totes les interrupcions que poden passar i quina subrutina s'ha de cridar.

En el cas de la Gameboy no és així, però tampoc té una única interrupció. Aquí les interrupcions executen directament (modifiquen el registre PC) una adreça de memòria que és sempre la mateixa i que els programadors de jocs han de tenir en compte.

Bit	Descripció
6	Habilitar/deshabilitar interrupt quan arribi a una línia en concret
5	Habilitar/deshabilitar la interrupció quan entra a la fase 2
4	Habilitar/deshabilitar la interrupció quan entra a la fase 1
3	Habilitar/deshabilitar la interrupció quan entra a la fase 0
2	Quan la linia horitzontal es igual a la esperada 1, sinó 0
1-0	Indica la fase en la que es troba: 00: Fase 0 01: Fase 1 10: Fase 2 11: Fase 3

Taula 3.9: Registre LCD Status

Per tal de gestionar totes les interrupcions, tenim 3 registres i algunes operacions de processador.

El registre principal que habilita o deshabilita totes les interrupcions s'anomena IME. Aquest registre és l'únic que no es pot modificar escrivint directament al registre. Per tal de modificar-lo hem d'utilitzar una de les instruccions de la Taula 3.10.

INS	Descripció
EI	Habilitar les interrupcions
DI	Deshabilitar les interrupcions
RETI	Habilitar les interrupcions i fer un return

Taula 3.10: Instruccions per modificar el IME

El registre que habilita o deshabilita cada una de les interrupcions s'anomena IE. Aquest registre és l'últim de l'espai de memòria (0xFFFF). Per tal d'habilitar una interrupció hem de posar el bit a 1. La Taula 3.11 mostra els diferents bits i la seva interrupció associada.

Bit	Descripció	Instrucció que executarà
0	Habilita les interrupcions del Vertical Blank	0x0040
1	Habilita les interrupcions del LCD Stat	0x0048
2	Habilita les interrupcions del Timer	0x0050
3	Habilita les interrupcions del port serial	0x0058
4	Habilita les interrupcions del Joypad	0x0060

Taula 3.11: Bit de cada interrupció del registre IE

3.6.2 Timers

Des dels primers computadors, una de les funcions bàsiques ha estat mantenir d'alguna manera una unitat de temps per tal de poder coordinar accions. Qualsevol joc, inclús el més simple, necessita una unitat de temps. Per exemple, el Tetris, necessita el temps per tal de definir la velocitat de les peces o fins i tot per fer que les peces siguin aleatòries.

Per tal de gestionar tots aquests problemes de temps, tota videoconsola i tot processador té algun tipus de timer que permet coordinar totes aquestes accions.

La Gameboy té uns registres que automàticament s'incrementen basats en uns paràmetres programables. En aquest apartat, descriuré com funciona aquest timer i quina és la seva estructura.

Com es pot observar en les especificacions inicials, la CPU de la gameboy funciona amb un clock de 4 MHz, amb dues mesures de temps per tal d'executar les diferents instruccions: El T-clock, el qual s'incrementa amb cada tick del rellotge, i el M-clock, el qual s'incrementa a un quart de la velocitat (1 Mhz). Aquests clocks són els que s'utilitzen com a base per a fer funcionar el timer, el qual funciona a un quart del M-clock, és a dir, 256Khz.

La Gameboy proporciona quatre registres separats per tal de fer funcionar els dos timers, divider i counter. En general, el sistema incrementa el valor del timer a una freqüència determinada. El divider o divisor, s'incrementa de manera permanent a 1/16 del rellotge base (16Khz). Com els registres són només d'un byte, el valor va de 0-255 de manera infinita.

El counter és més versàtil, ja que es pot programar amb 4 velocitats: el clock base dividit per 1, 4, 16 o 64. També es pot programar perquè no comenci a 0 cada cop que omple el registre, sinó que permet afegir un valor base anomenat Modulo. A part, el sistema llença una interrupció cada vegada que el comptador s'omple, és a dir, quan arriba al 255.

En total hi ha 4 registres d'un byte dedicats als timers. Veure Taula 3.12.

<table>
<tr><th>Adreça</th><th>Registre</th><th>Descripció</th></tr>
<tr><td>0xFF04</td><td>Divider</td><td>S'incrementa amb una freqüència de 16384Hz; Es reinicia cada cop que escrivim aquest registre</td></tr>
<tr><td>0xFF05</td><td>Counter</td><td>S'incrementa a la freqüència especificada. Genera interrupció 0x50 quan passa de 255 a 0</td></tr>
<tr><td>0xFF06</td><td>Modulo</td><td>Quan el Counter sobrepassa el 0, es reinicia amb aquest valor.</td></tr>
<tr><td>0xFF07</td><td>Control</td><td>
<table>
<tr><th>Bits</th><th>Funció</th><th>Descripció</th></tr>
<tr><td>0-1</td><td>Velocitat</td><td>00: 4096Hz
01: 262144Hz
10: 65536Hz
11: 16384Hz</td></tr>
<tr><td>2</td><td>Execució</td><td>1 per activar, 0 per desactivar</td></tr>
<tr><td>3-7</td><td colspan="2">Sense funció</td></tr>
</table>
</td></tr>
</table>

Taula 3.12: Timers

3.6.3 Joypad

La Gameboy té un total de 8 botons, 4 per les direccions, 2 per als botons A i B, i uns altres 2 per als botons de Select i Start.

Com tots els altres perifèrics de la videoconsola, per tal de llegir el seu valor hem de consultar un registre. Exactament el FF00. Encara que el registre sigui de 8 bits i tinguem 8 botons, no s'ha codificat assignant un bit a cada botó. Si no que s'ha subdividit el conjunt en botons de direcció i botons d'acció. Per tal de poder llegir el valor, primer hem de triar quin dels dos subconjunts volem conèixer, de tal manera que el registre quedaria com a la Taula 3.13.

Per tal de no haver d'estar preguntant constantment al registre, com hem vist al apartat d'interrupcions, hi ha un interrupció que salta cada vegada que un dels botons ha estat premut.

3.6.4 Cartridge

El cartridge o cartutx de la Gameboy color és l'element extern que conté la ROM del joc. Es podria pensar que només tenen una memòria on guarden la informació, però realment fan molt més que guardar dades.

Com s'ha vist en altres apartats, la capacitat del processador per accedir a l'espai adreçable és molt limitada. Només pot accedir a 64 KB de memòria o a 65.535 adreces

Bit	Descripció
5	Seleccionar botons d'acció
4	Seleccionar botons de direcció
3	Direcció sud o Acció Start
2	Direcció nord o Acció Select
1	Direcció oest o Acció B
0	Direcció est o Acció A

Taula 3.13: Registre FF00

i més de la meitat de l'espai s'utilitza de manera interna. Això vol dir que només ens quedarien 32 KB per tal de guardar i llegir tot el contingut d'un joc.

Aquest problema es pot solucionar afegint més lògica i responsabilitat al cartridge. El sistema consisteix a tenir més memòria que els 32 KB i tenir una espècie de selector que ens permeti triar quina regió de la memòria total volem veure.

A més a més, hi ha jocs que necessiten més memòria RAM per tal de poder emmagatzemar més informació. Per exemple qualsevol joc de la saga Pokémon, necessita molta informació per tal de poder representar tots els diferents gràfics. Per tal de solucionar aquest problema, hi ha videojocs que inclouen més memòria RAM a dins el cartutx. Aquesta memòria RAM també es pot fer persistent si s'implementa una font d'alimentació externa com vindria a ser una pila, cosa que permet guardar estats persistents del joc.

Per últim comentar que hi ha múltiples versions i configuracions diferents, a part tenim tota una gamma de cartutxos modificats que permeten funcionalitats especials. Alguns dels exemples més extravagants: càmera, jocs amb moviment per giroscopi i acceleròmetres, jocs amb plaques fotovoltaiques, i al final qualsevol cosa que puguis adaptar amb lectura i escriptura per registres.

Aquí explicarem dos casos i deixarem tota la resta de banda, encara que tots funcionen de manera molt similar:

No MBC

Aquest és el cas més senzill, ja que el joc no sobrepassa els 32KB i estan mapejats directament a la memòria. Seria el cas del Tetris, i tota la informació es pot accedir directament utilitzant el rang 0000-7FFF com es pot veure a la Taula 3.3.

MBC1

El MBC (Memory Bank Controller) és l'encarregat de gestionar quina memòria volem escriure en cada moment. Hi ha diferents tipus de xip i aquest és el primer el qual permet 2 MB de memòria ROM i 32 KB de RAM.

Nosaltres des del punt de vista del processador, només podem seguir accedint a la regió 0000-7FFF. El truc està en el fet que no només podem llegir, sinó que podem escriure per tal de configurar el MBC i modificar el que veiem a través d'aquella regió.

Sense entrar en molt detall, a la Taula 3.14, podem veure quin comportament modifiquem escrivint en les diferents adreces.

Rang	Descripció
0x0000-1FFF	Habilita la memòria ram externa(ExRAM)
0x2000-3FFF	Selecciona quina regió de la ROM es vol adreçar
0x4000-5FFF	Selecciona quina regio de la RAM es vol adreçar.
0x6000-7FFF	Selecciona el mode de MBC que volem utilitzar

Taula 3.14: Selector del MBC

3.7 Parts no explicades

En aquest apartat s'han explicat les parts més rellevants de la Gameboy. Queden molts detalls que no han estat comentats, al igual que alguns apartats s'han obviat totalment de la documentació (So, Port Serie), ja que no s'implementaran en aquest TFG.

Capítol 4

Implementació

Aquest és un apartat descriptiu on explico tot el procés que he seguit per tal de traduir tota la documentació i aconseguir una implementació funcional. He decidit dividir la documentació en 6 etapes diferents. Aquestes etapes van des d'una definició de prototip inicial fins a la completa implementació del projecte. A la pràctica, no s'ha seguit exactament aquesta evolució perquè hi ha hagut moltes modificacions al llarg de la implementació. Tanmateix, la manera com s'exposa aquí és la més senzilla per entendre-ho i explicar-ho. Al mateix temps, és la manera natural d'implementar-ho.

Encara que no s'entrarà en detalls tècnics de la implementació, és útil saber el context amb el qual s'ha implementat. La implementació s'ha dissenyat utilitzant programació orientada a objectes amb l'última versió de Java disponible (15.0.1). També s'han aplicat patrons de disseny, encapsulació i un model vista controlador. Aquestes aplicacions no s'expliquen en aquest treball, però donen lloc a algunes abstraccions que per si soles no tenen gaire sentit.

4.1 Sistema bàsic

La primera etapa es basa en una prova de concepte. A causa de la dificultat de comprensió del funcionament i lectura de la documentació, la prova de concepte permet entendre el funcionament general d'una forma simplificada.

La prova de concepte es basa a emular una CPU mínima amb una arquitectura Von-Neuman. És a dir, llegir instruccions i modificar el resultat partint de la mateixa memòria. La CPU és per definició una màquina de Turing, per tant, podríem dir que és una caixa negra on entra un input i s'obté un output. Per tal de poder llegir instruccions (input) i guardar resultats (output), definim el nostre escenari com: una CPU que realitza X funcions bàsiques i una memòria RAM que conté les instruccions a executar i on es guarda la informació.

4.1.1 RAM

Com que és una fase inicial, la implementació més senzilla de memòria RAM és utilitzar un vector. Primerament, vaig utilitzar les mateixes unitats que la Gameboy, on la memòria RAM era un vector de 64 KB, és a dir, un vector de tipus byte de 65.535 posicions. D'aquesta manera s'utilitzen 16 bits per a indicar l'adreça de memòria a la qual es vol accedir.

4.1.2 CPU

La CPU consta de diversos components i perquè funcioni la implementació inicial, com a mínim necessitem les operacions i els registres. Per tal de simplificar el concepte d'operacions, podem dividir el seu funcionament en tres fases: fetch, decode i execute. Quan la CPU està en funcionament, aquestes tres fases s'executen de manera indefinida. En aquesta implementació, també vaig introduir els flags de la CPU, ja que en un principi no deixen de ser un registre i en general no compliquen la implementació.

Operacions

Per emular un conjunt d'operacions bàsiques, hem de definir com a mínim un conjunt d'operacions. En el nostre cas i per tal de tocar una mica de tot, definirem unes 6 instruccions permeten com a mínim fer: una operació aritmètica, llegir i escriure de memòria, i un jump per tal de modificar la seqüència del programa.

Per tal d'identificar l'operació utilitzaré 8 bits per a seguir les unitats de la Gameboy encara que ho podríem fer amb 3 bits, ja que implementarem unes 6 operacions.

Un cop ja tenim definit el plantejament del problema, podem tenir un esquema de com serà la nostra configuració de les instruccions. Veure taula 4.1.

Codi	Instrucció	Descripció
0x00	`Nop`	Instrucció que no fa absolutament res
0x01	`Load A, nn`	Guardem al registre A, el numèric n
0x02	`Load B, A`	Guardem al registre B el valor de A
0x03	`Add A, B`	A = A + B
0x04	`Save A, nn`	Guardem A a la adreça nn
0x05	`Jmp nn`	Saltem a la adreça nn

Taula 4.1: Instruccions de la fase 1.

Registres

Per tal de recrear la nostra fase inicial, hem de definir els diferents registres. Hi ha diverses maneres d'implementar això, però la més senzilla és tenir una variable tipus byte per a cada registre.

En aquesta fase inicial i com es pot veure al set d'instruccions, només tindrem 3 registres: A, B i PC. Els registres A i B seran de 8 bits, però el PC ha de ser de 16 bits per tal de poder adreçar la memòria RAM que acabem de definir. Al principi el registre PC l'inicialitzarem a 0x0000 i serà la direcció on guardarem les diferents instruccions a executar.

Flag

En la prova de concepte, només definirem un flag per tal de no complicar les coses. Utilitzarem el flag 0 perquè és el més senzill de realitzar.

Després podrem utilitzar la instrucció jmp i fer-la condicional per tal que només salti quan es compleix que el flag és zero.

Fetch

Fetch és la fase on el processador obté la instrucció a executar, hem de llegir la memòria RAM en la posició del PC (Program counter). Després incrementarem el PC per tal que el següent cop que entrem carreguem la pròxima instrucció.

```
funcio fetch():
        retornar memoria[registrePC]
        registrePC++
```

Decode

Decode és la fase en la qual el processador interpreta la instrucció que hem obtingut a l'etapa anterior.

Aquest apartat es pot implementar de diferents maneres. Ja que el nostre cas inicial té poques instruccions i no segueixen cap lògica, el que hem fet és implementar un switch-case o un if-else per tal d'executar una instrucció o un altre depenen de la instrucció.

A manera d'exemple amb pseudocodi, quedaria de la següent manera:

```
funcio decode(instruccio tipus enter):
        si (instruccio):
                es 0x00: no fer res
                es 0x01: fer load a, nn
                es 0x02: fer load b, a
                ....
        si no es ninguna:
```

```
retornar error: instruccio no trobada
```

Execute

Aquesta etapa és l'encarregada d'executar la funció que s'hagi escollit a la fase anterior. Aquest apartat es pot implementar conjuntament amb l'anterior per tal d'executar la funció directament. També es pot definir de manera separada i que l'etapa decode retorni una funció.

A la figura 4.1 podem observar com quedaria l'esquema simplificat amb aquests elements.

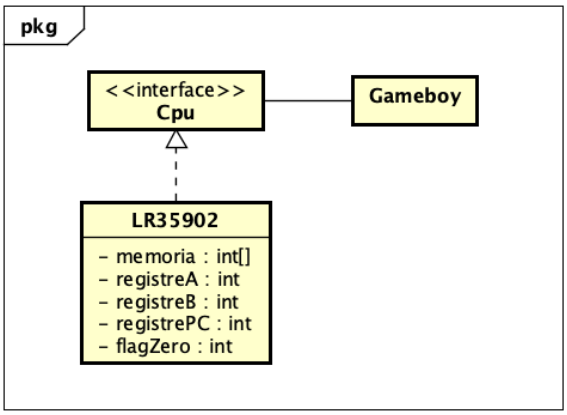

Figura 4.1: Diagrama de classes del sistema bàsic

4.2 Memòria

Un cop sabem que el prototip de processador funciona, és un bon moment per separar responsabilitats i intentar fer més abstracte i independent cada una de les parts. Aquí no introduirem funcionalitat nova, només aïllarem i encapsularem els diferents components per tal de facilitar la implementació final i ser capaços de reutilitzar codi.

El processador i la memòria és l'element principal a separar. Durant tot el projecte, hi ha diferents tipus de memòria la qual el processador pot accedir i hauria de ser transparent. Per tal d'aconseguir això, podem definir la memòria externa a la CPU i ser el processador el que fa lectures i escriptures sobre aquesta.

A diferència de la memòria, els registres i els flags sí que són totalment dependents del processador. La CPU és l'únic component lògic que pot llegir i escriure sobre aquests. Però pensant en una implementació final, ens serà útil separar aquests elements per tal de poder tenir un logger o poder fer modificacions independents a la lògica del programa. També és útil separar-los, per tal de poder definir alguns mètodes que ens facilitin manegar els registres dobles i altres funcionalitats.

A part de tot això és útil aplicar una interfície per tal d'estandarditzar una manera d'accedir als diferents futurs elements.

A la figura 4.2 podem veure la representació sobre l'abstracció de memòria.

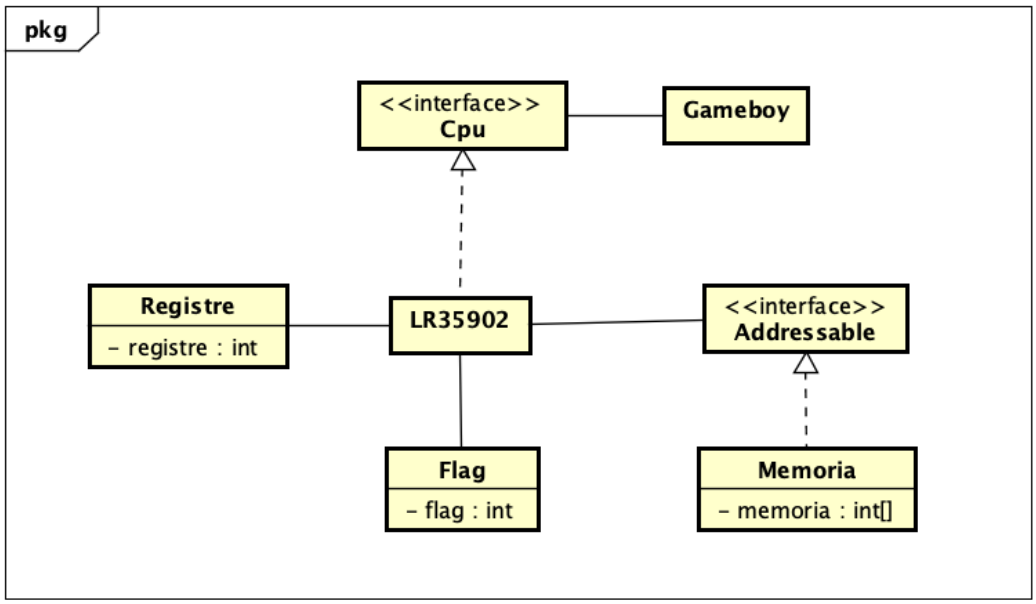

Figura 4.2: Diagrama de classes amb la separació de memòria

4.3 MMU

Un cop separats tots els components, podem fer altres modificacions per tal de poder incloure amb facilitat altres components. Un dels components més clars per tal de separar és la memòria. El processador es comunica amb els altres components per adreces de memòria. Fins ara la implementació només utilitza la memòria per a llegir instruccions i modificar la memòria. Per tal de poder incloure diferents components o diferents tipus de memòria i que sigui transparent per al processador, podem definir el concepte de MMU. El MMU o el Memory Managment Unit, serà un component que diferenciarà quina direcció o conjunt de direccions accedeixen a quin element. Així que aquest contindrà un llistat amb tots els diferents processos que el processador és capaç d'accedir: Memòria Ram, Cartutx, VRAM, OAM, Joypad, Timers i PPU. Només podrem llegir i escriure posicions de memòria i serà responsabilitat de cada element prendre les decisions adients.

La implementació més fàcil per a la MMU és un llistat de if-else que redirigeixi la petició en funció del rang d'adreces. Un exemple simple de pseudocodi seria el següent: La nova representació i diagrama UML serà el següent:

```
funcio readByte(addr tipus enter):
```

```
si (addr <= 0x7FFF):
        llegir rom
si (addr <= 0x9FFF):
        llegir ram
....
si no es ninguna:
        retornar error: direccio no trobada
```

A la figura 4.3 podem observar una iteració de l'esquema amb el MMU incorporat. També s'introdueix la interface Addressable per tal de que tots els elements que s'accedeixen per adreces de memòria tinguin 3 funcions en comú.

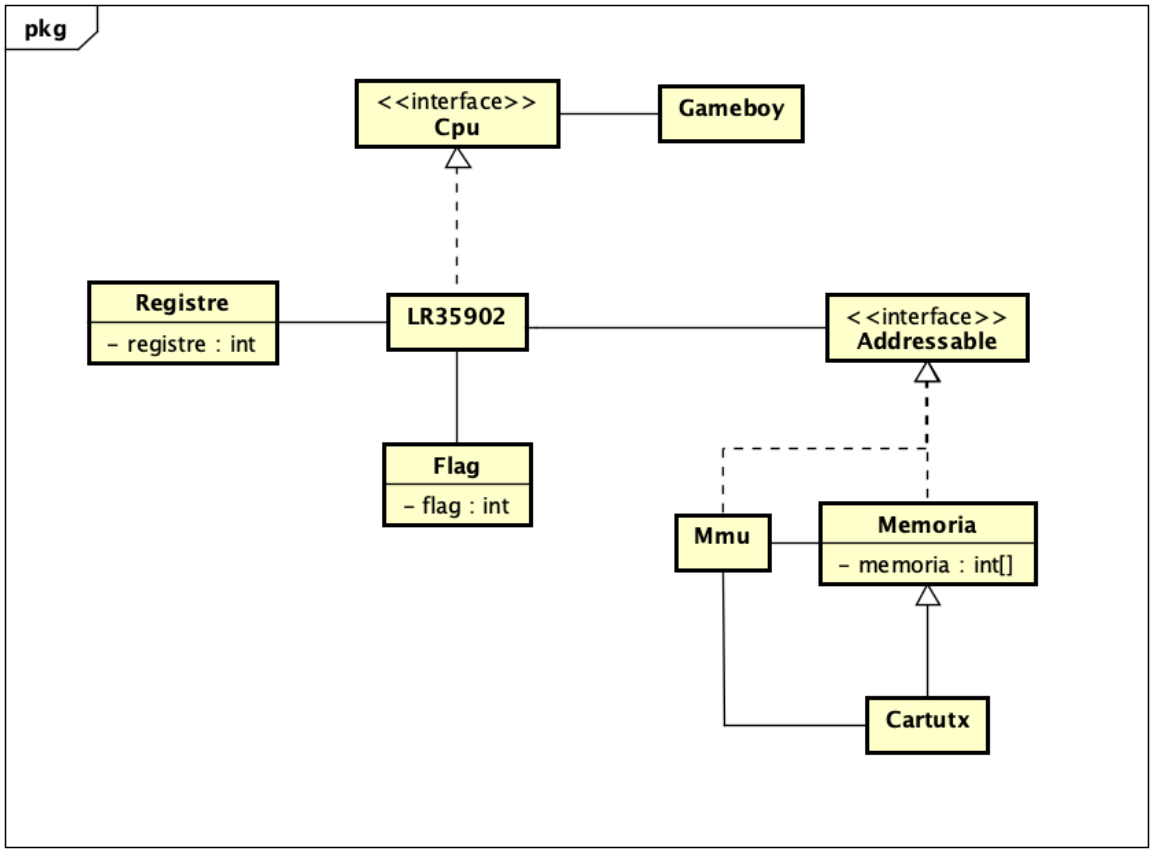

Figura 4.3: Diagrama de classes afegint el MMU

4.4 Cart

La implementació més senzilla per tal de poder provar una ROM oficial, és utilitzar un vector i carregar el contingut del fitxer. Hem de ser conscients que aquest mètode només funcionarà amb els jocs més antics i més simples. Els jocs només poden tenir

memòria ROM i d'una mida màxima de 32KB. Un bon exemple de joc que compleix aquestes característiques és el Tetris.

Per tal de fer aquesta primera implementació, podem inicialitzar un vector amb el contingut del fitxer que tinguem i només accedir aquest vector quan llegim les posicions de memòria 0x0000-0x8000.

4.5 Interrupcions

Per tal de poder comunicar-nos necessitem implementar les interrupcions. La lògica és senzilla. Per tal d'executar una interrupció s'han de complir dues condicions. Que el flag o registre d'interrupcions actives (0xFFFF) estigui activat i que el registre (0xFF0F) que marca els diferents tipus d'interrupcions tingui el bit == 1.

Un cop donades aquestes dues condicions, el processador guarda a la pila l'estat actual i salta a la línia de codi corresponent a la interrupció executada. Aquesta bé donada pel registre 0xFF0F.

Les dues coses més significatives, és que la modificació del registre 0xFFFF que activa i desactiva totes les interrupcions només es pot modificar per operacions de la CPU i no modificant directament el registre. L'altre és que un cop s'inicia una subrutina d'una interrupció es deshabiliten automàticament modificant el registre 0xFFFF.

4.6 CPU estès

Un cop sabem quin és el funcionament bàsic de les diferents parts internes, podem implementar tot el conjunt d'operacions que té la GameBoy. Aquesta implementació es pot portar a terme de diferents maneres: la més senzilla i aplicada en aquest projecte és utilitzar un switch o if-else per cadascuna de les operacions diferents. L'altra manera més compacta però menys visual i clara és fer-ho de manera algorítmica.

S'ha de remarcar que moltes de les operacions són repeticions de si mateixes però amb diferents registres com a arguments. Una manera efectiva de simplificar això sense implementar una manera algorítmica de fer el decode de les instruccions, és definir funcions principals i agrupar el subconjunt d'operacions semblants, per tal de no repetir el codi que comparteixen.

Aquesta esdevé una de les parts més tedioses i propensa a errors que s'ha implementat durant aquest projecte. Per tal de poder comprovar els errors, es poden aplicar diverses tècniques: utilitzar un emulador ja existent per a comprovar el resultat esperat, escriure tests per tal de comprovar el correcte funcionament de les instruccions...

La implementació es pot portar a terme comprovant a cada cicle de CPU quin és l'estat de les interrupcions i executant la subrutina pertinent.

4.7 Display

En aquest apartat definirem com serà l'output del programa. L'únic output que obtenim de la gameboy és el que es mostra per pantalla, en el nostre cas, una imatge de 160x144 píxels que teòricament es renderitzarà 60 cops per segon.

Per tal de fer més senzilla la implementació dividirem el cas en dues parts. La primera serà mostrar una finestra i representar un vector de píxels amb els 4 colors possibles. La segona s'encarregarà de convertir la informació en un vector i establir quan i com s'ha de pintar.

Display

El display serà l'encarregat de crear un frame i representar el vector. La implementació és senzilla, ja que utilitzarem una llibreria o framework per tal de pintar en pantalla i només haurem de fer la correlació entre l'índex i el color corresponent i transformar un vector en posicions X i Y per poder identificar el píxel de la pantalla.

Un exemple de com renderitzar el vector:

```
funcio pintar(vector tipus enter):
  per i = 0 fins i = 144 fer:
    per j = 0 fins j = 160 fer:
          color = obtenir_color ( vector[160 * i + j] )
          renderitzar(i , j , color )
        fi
  fi
```

Pixel Processing Unit

El PPU serà el component encarregat de gestionar diverses funcions:

- Gestionar les diferents crides de pantalla a través dels registres LCD Control i LCD Status mencionats al capítol anterior.
- Accedir a la VRAM i la OAM per tal de crear els elements Background, Sprites i Window.
- Generar un vector amb els píxels a dibuixar.
- Portar un rellotge intern per tal de renderitzar la pantalla 60 imatges per segon i sincronitzar-se amb els altres components.

Per tal d'entendre millor el funcionament del dos elements i de com es comuniquen entre ells ens podem guiar per la figura 4.4.

A la figura 4.5 podem observar el esquema simplificat amb la PPU i el Display.

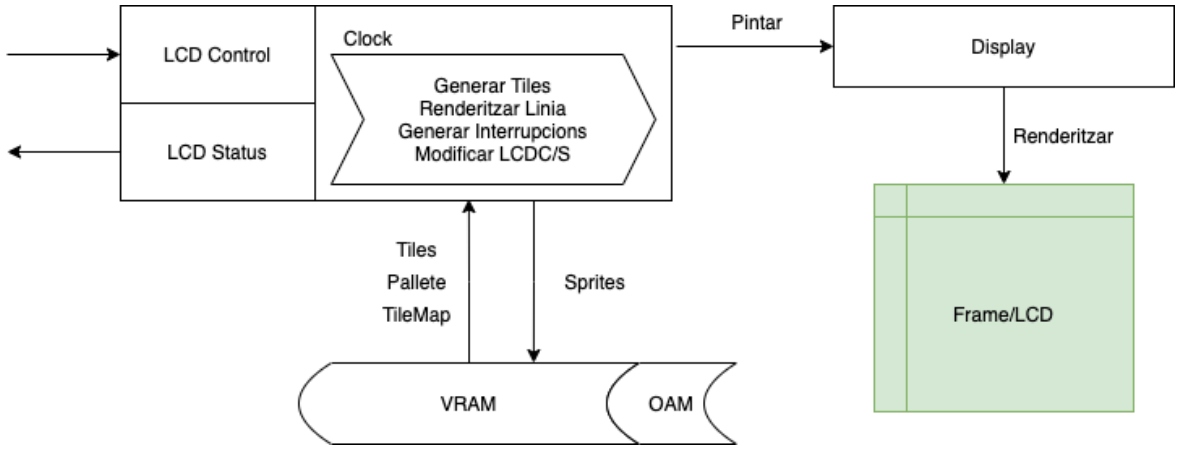

Figura 4.4: Funcionament intern Ppu

4.8 Input

4.8.1 Joypad

Per tal de poder tenir algun tipus d'interacció amb l'exterior, en aquesta etapa definirem com serà l'input del sistema. Implementarem una classe, la qual serà la responsable d'enviar els diferents senyals als altres components. Com ja s'ha comentat en altres apartats, l'única manera de compartir informació entre components és utilitzant l'espai de memòria. En el nostre cas, utilitzarem el Memory Managment Unit que hem implementat anteriorment.

La GameBoy té un total de 8 botons diferents. Les 4 fletxes de direcció, els botons d'acció A, B i els generals Start i Select.

La manera com estan implementats per hardware és utilitzant un multiplexor i dedicant 6 bits en total. S'utilitzen 4 per identificar el botó i 2 per al selector de canal.

Per tal de fer la implementació en codi utilitzarem les mateixes llibreries del llenguatge per a saber quina tecla hem polsat i assignarem cada una d'elles als diferents bits del registre.

Aquest l'actualitzarem de manera instantània utilitzant les mateixes interrupcions del llenguatge i activarem la interrupció interna corresponent per tal de comunicar a tot el sistema el canvi efectuat.

A partir d'aquí qualsevol element podrà accedir al registre 0xFF00 per tal de llegir el valor de les tecles polsades.

A la figura 4.6 podem veure un esquema de com s'interpretaria un esdeveniment de teclat.

4.8.2 Timer

Per una altre banda tenim el comptador o timer. Aquest element s'encarrega de portar un petit comptador el qual activarà una interrupció en el moment que arribi al valor

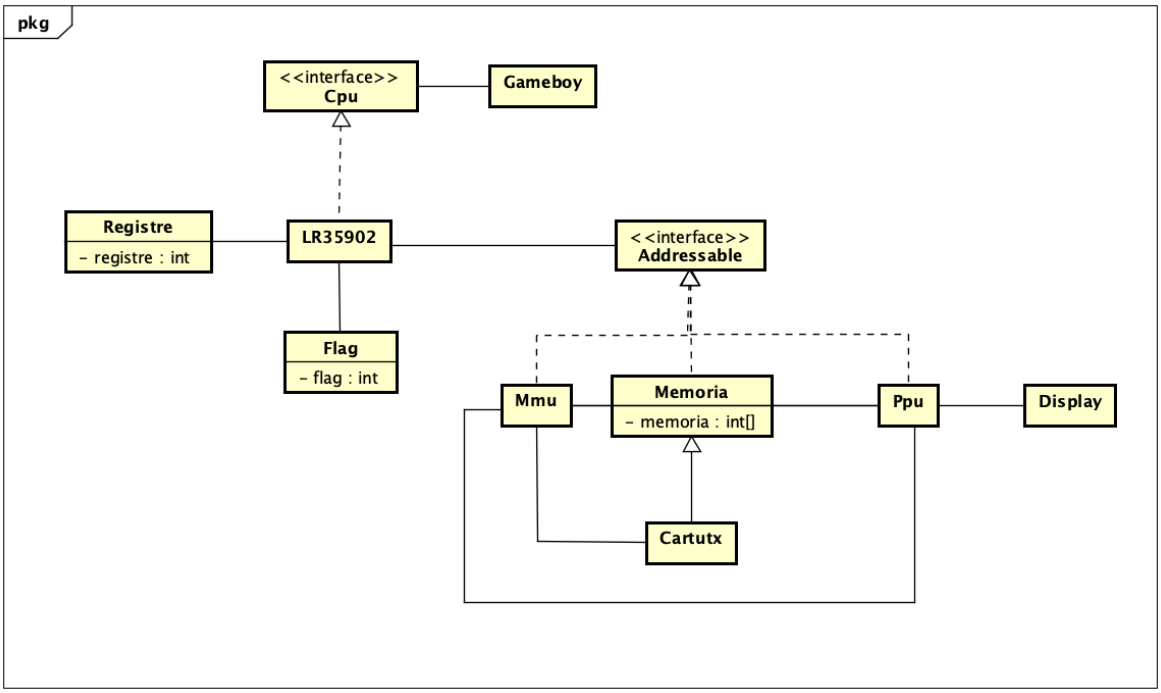

Figura 4.5: Diagrama de classes afegint el element PPU i Display

0xFF. Aquest timer serà un subscriptor del nostre clock per la qual cosa incrementara el seu valor a cada iteració de la CPU.

A la figura 4.7 tenim l'esquema amb els elements de l'input.

4.9 Clock

El concepte del clock en general és senzill, encara que la seva implementació a causa de limitacions del hardware i el llenguatge fan que la implementació hagi de ser una mica enrevessada.

El clock s'ha implementat seguint el patró de disseny Observer. Aquest és basa en una llista de subscriptors que en el nostre cas seran tots els elements síncrons (timer, cpu, ppu).

Cada cop que la CPU realitzi cert número d'accions, el clock informa a tots els seus subscriptors per tal de que realitzin el mateix número d'accions i així generar un tipus de sincronisme amb tots els elements. Per altra banda, podem limitar que el propi clock tingui una freqüència màxima si el relacionem amb el temps que tarda un fotograma en la Gameboy original (16.7ms).

A partir d'aquí podem controlar el flux del emulador. Podem parar, continuar o alterar la velocitat del rellotge per tal de modificar el comportament general del programa.

A la figura 4.8 podem observar l'esquema amb el Clock incorporat.

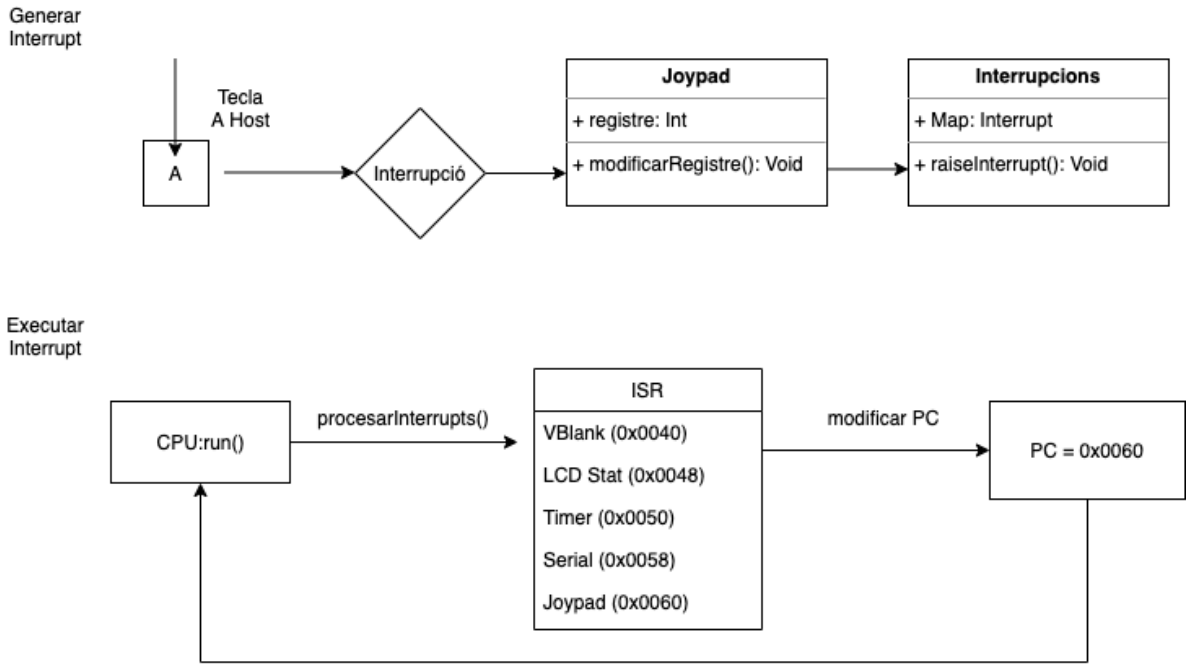

Figura 4.6: Cicle interrupcions del input

4.10 MBC

El cart o cartutx és la part de la Gameboy que conté tota la informació sobre el joc a executar. Com hem vist en l'apartat d'especificacions sobre la Gameboy, és un dels components que més configuracions pot arribar a implementar. Tot depèn de quines versions o configuracions de cartutx volem acceptar en el nostre emulador.

El més senzill és la versió que implementen jocs com el Tetris on només tenim una memòria ROM la qual té la mateixa mida que la direcció d'adreces (32KB). Això es va complicant fins que tenim versions amb molta més memòria ROM, xips de RAM al mateix cartutx o fins i tot sensors o rellotges.

Com totes aquestes funcionalitats estan dins el cartutx i només podem accedir a aquest a través dels 32KB, s'ha d'implementar una lògica extra per tal de gestionar quina regió i quin tipus de memòria estem accedint en cada moment.

En la part pràctica d'aquest treball, només s'ha implementat la versió on tenim ROM i el MBC1,3,5 encara que no funcionen completament. Ja que característiques com la RAM persistent o el rellotge no estan implementades.

La primera implementació que es va portar a terme va ser la creació de dues classes. Una anomenada Cartutx, la qual contenia tota la informació rellevant al cart i l'inicialitzava segons els headers del cartutx. L'altra classe anomenada MBC, la qual contenia tota la lògica de tots els tipus de cart i s'utilitzava d'interfície per a comunicar-se amb la MMU. El problema d'aquesta implementació radica amb la quantitat de carts diferents que existeixen al mercat. A mesura que intentaves introduir un nou tipus de

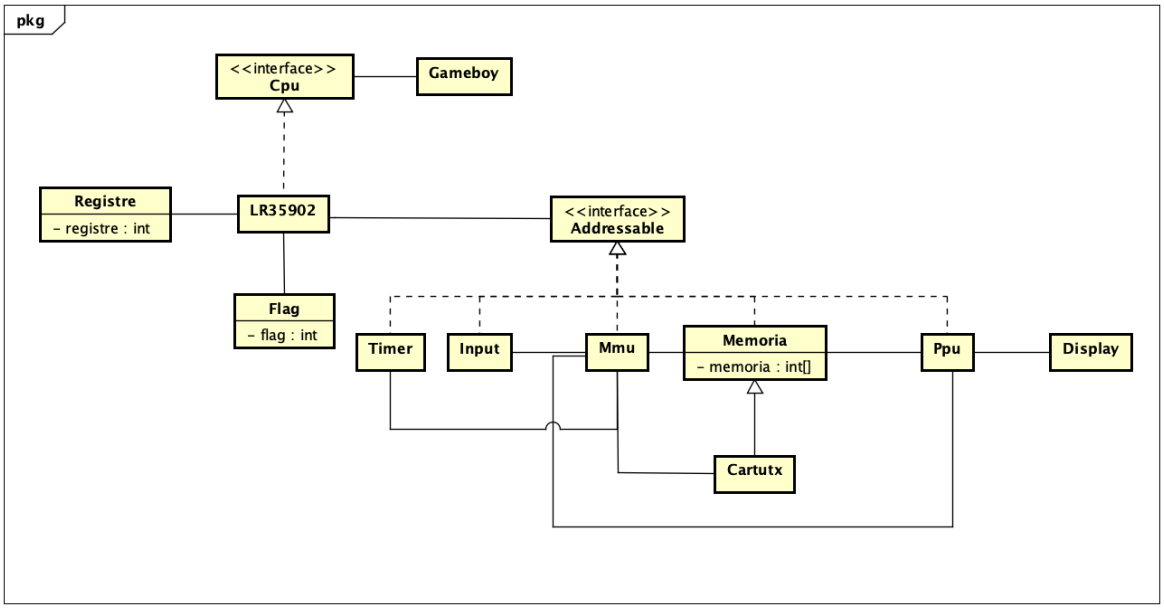

Figura 4.7: Diagrama de classes afegint els Joypad i Timer

MBC, s'havien de modificar les dues classes i creixien en complexitat i mida. Cada cop que es llegia o s'escrivia en el cartutx, s'havia de passar per tota la lògica de la classe MBC. En acabar d'implementar els 3 MBC, la classe MBC era molt extensa i feixuga de modificar, igual que lenta a l'hora d'executar.

Un exemple de com s'efectuava cada lectura i escriptura:

```
funcio llegirByte(adresa tipus enter):
 Si el tipus de cart es ROM:
   retorna rom[adresa]
 Si el tipus de cart es MBC1:
  Si la adresa < 4000:
   retornar rom [adresa]
  Si la adresa es > 4000 i < 8000:
   adresaReal = (bancRom * 0x4000 ) + ( adresa - 0x4000 )
   retornar rom[adresaReal]
  ...

 Si el  tipus de cart es MBC2:
  ...
 Si el tipus de cart es MBC3:
  ...
 Si el tipus de cart es MBC5:
  ...
fi
```

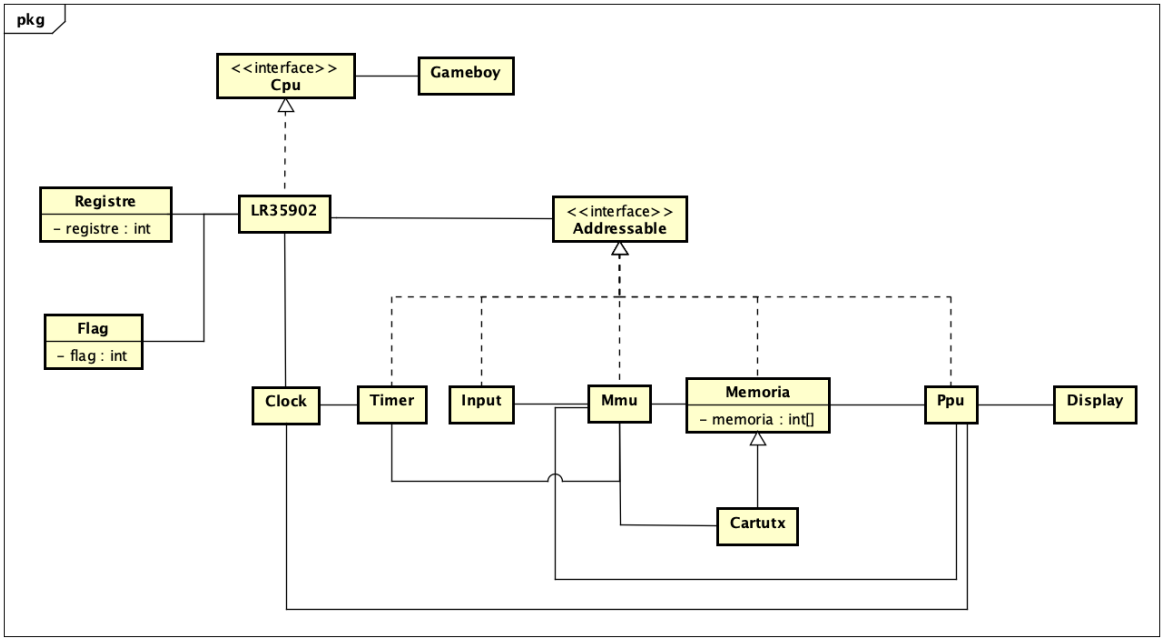

Figura 4.8: Diagrama de classes de la etapa 6

Per millorar aquesta implementació, es van abstreure i subdividir les classes per tal que la incorporació d'un nou tipus de cartutx no modifiqués les classes existents i que la lògica del cartutx fos única segons la instància que es tingués en cada moment.

A trets generals, la implementació es basa a dividir al màxim les classes per simplificar cada una d'elles considerablement, utilitzant alguns patrons de disseny com la factory o abstract facotry per tal de facilitar la creació del cartutx basat en el seu tipus. Podem veure el diagrama a la figura 4.9

4.11 Interficie d'usuari

La interfície d'usuari es basa en quatre pantalles principals i una barra de menú que permet accedir als diferents elements. No s'ha implementat cap mockup previ ja que la interficie era molt senzilla.

La pantalla principal es mostra en iniciar l'aplicació. Veure figura 4.10.

Utilitzant el submenú File podem carregar, parar i continuar un joc, o sortir de l'aplicació. En seleccionar un arxiu compatible, l'aplicació s'inicia i carrega la pantalla de joc. Veure figura 4.11.

Per tal de poder saber quines tecles s'utilitzen per controlar l'emulador podem anar a la secció Keybindings utilitzant el menú. Un cop allà obtindrem la figura 4.12 la qual ens mostra quines tecles corresponent a cada botó.

Per últim podem obtenir la informació sobre l'aplicació anant a la pantalla about. Aquesta mostrarà per pantalla diferent informació com podem veure a la figura 4.13.

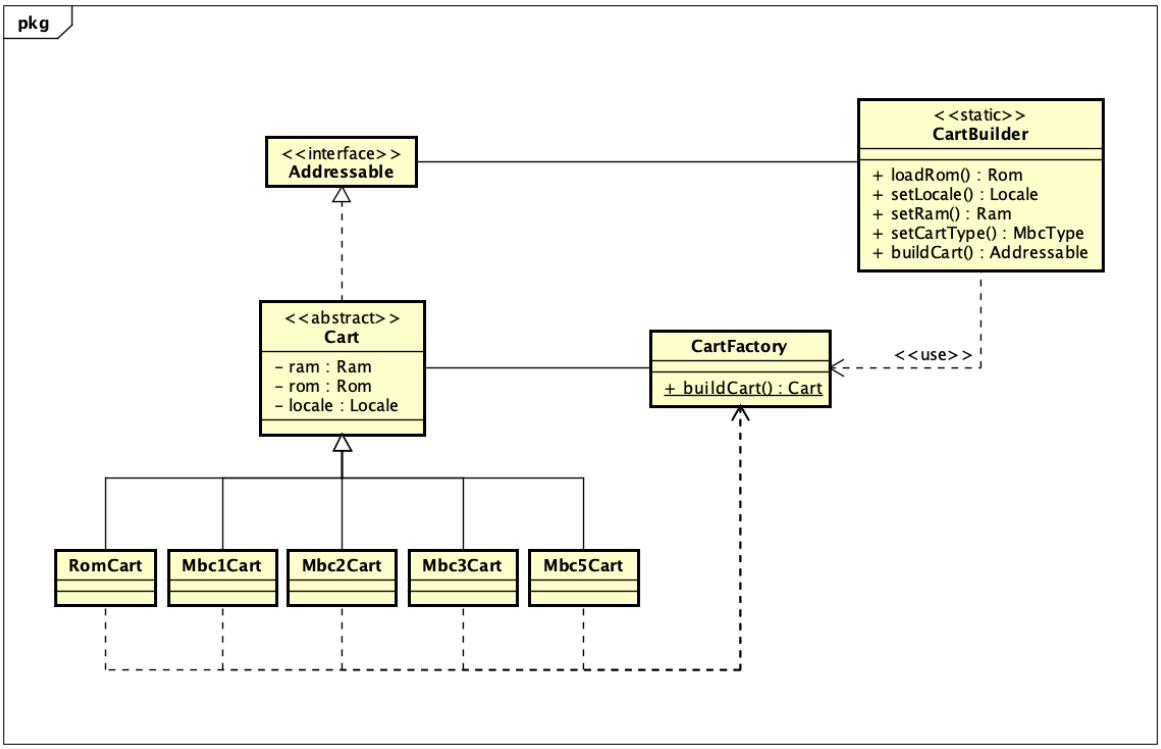

Figura 4.9: Diagrama del Cartutx

El submenú Size permet triar la mida amb la qual es representen els píxels, per tal de poder fer més gran o petita la interfície d'usuari.

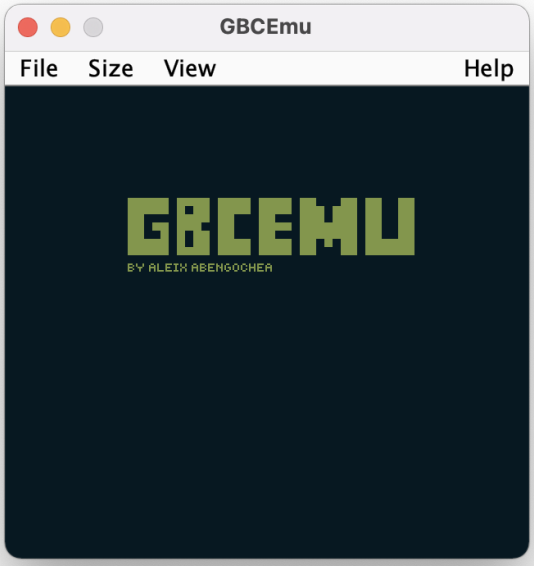

Figura 4.10: Pantalla inicial

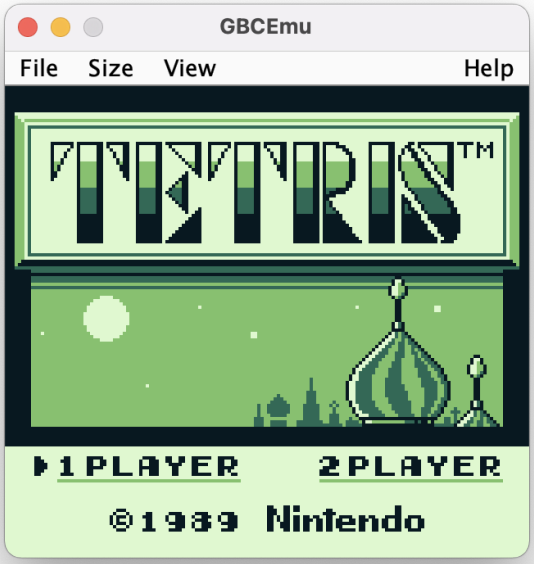

Figura 4.11: Execució d'un joc

Figura 4.12: Disposició de les tecles

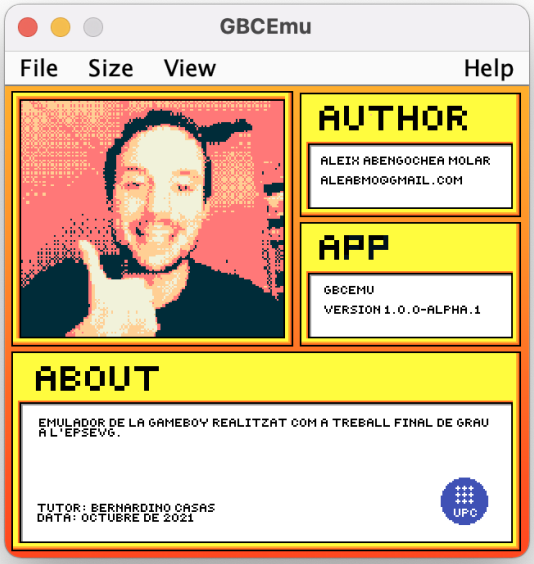

Figura 4.13: Informació sobre l'aplicació

Capítol 5

Problemes

Ja que tant la teoria com la implementació de l'emulador és un tema feixuc i extens, trobo que és encertat dedicar un apartat a explicar els diferents problemes que s'han anat trobant durant el desenvolupament. És una manera de fer més entretinguda l'explicació i probablement l'apartat que serà més útil a aquelles persones que estiguin interessades a intentar afrontar el mateix repte.

En aquest apartat s'explicarà amb detall els problemes més importants que s'han anat afrontant a mesura que s'ha desenvolupat el treball. Estarà centrat en la implementació.

5.1 Tipus primitius en Java

El problema que més modificacions ha provocat és el fet d'haver triat Java com a llenguatge per a la implementació.

Per tal de ser rigorosos amb la implementació, en un principi, es van triar diferents tipus primitius de variable per tal que tinguessin la mateixa mida que la seva analogia real. Per exemple:

Per a totes les adreces de memòria i opcodes de la CPU, s'utilitzava el tipus short de Java perquè té una capacitat de 16 bits.

Per a totes les configuracions de memòria i registres, s'utilitzava el byte, ja que té una capacitat de 8 bits.

El problema ve donat quan intentes operar amb aquest tipus en Java. Java no realitza les operacions amb el tipus que hagis definit, sinó que fa una conversió interna al tipus int. Això genera diferents problemes:

El primer i més important, és que realitza una extensió de signe per defecte. Un valor que tingui el bit de més pes a 1 com el 0xFFFF, quan es transforma a un int passa a ser el valor 0x8000FFFF. I no hi ha cap manera de saltar aquesta funció i no existeixen els primitius sense signe. La solució que es va fer al principi és intentar fer comprovacions per cada operació que es portava a terme, però al final es va decidir tornar a implementar tot el projecte utilitzant int com el tipus per defecte. Això és

possible, ja que cap dada omple els 32 bits i mai es té el problema del signe.

L'altre problema era que després de fer una operació, el resultat havia de ser guardat en una variable int o realitzar un casting al tipus en qüestió. Ja que en Java una operació entre dos bytes genera un int. La solució inicial va ser fer casting després de cada operació, però al final es va resoldre aquest problema quan es va tornar a implementar amb ints.

5.2 Rellotge de 8Mhz

Per tal de definir el temps dins el nostre emulador, es va intentar definir una classe clock que portés el temps dins el programa i organitzés els altres components. Per exemple, aquesta classe clock tindria una funció run, que s'encarregaria d'activar la funció tick o step a tots els altres elements. Així aconseguiríem que tots els elements tinguessin un sincronisme entre ells. Aquesta idea d'implementació es basa en com funciona el hardware.

El problema d'aquesta idea inicial és una limitació del llenguatge i segurament de hardware. El processador de la Gameboy té una velocitat de rellotge de 8 Mhz, això vol dir que hem de ser capaços de fer tick cada 125 nanosegons.

És impossible fer una implementació que ens deixi executar una funció cada 125 nano segons. Hi ha una funció en Java que ens retorna el temps en aquesta magnitud, però a la documentació ja indica que té una imprecisió molt gran. Per altra banda hi ha massa overhead si intentem fer el càlcul de 125 ns i intentem executar tots els elements.

La solució a aquesta implementació va estar donar el control a la CPU. Executar una instrucció, portar un recompte de quants cicles teòrics comporta això i executar el mateix nombre de cicles a la resta de components.

5.3 Refactor Cartutx

Aquest va ser un refactor necessari perquè el codi es pogués entendre. La primera implementació recarregava la classe Cart de tal manera que qualsevol lectura o escriptura havia de passar per moltes comprovacions per tal d'identificar quin tipus de MBC havia d'utilitzar.

La solució ha estat dividir en múltiples classes per tal que futures implementacions de diferents tipus de cartutx no impliqui modificar totes les classes un altre cop. Ara en carregar, s'identifica el tipus i només es carrega l'algoritme necessari.

5.4 Display Jswing

Més que un problema ha estat un falta de comprensió de com funcionen els gràfics o el Swing a Java. Probablement si algú es troba en aquest problema, el millor consell

és que utilitzi una llibreria o un framework que no sigui enfocat a finestres o entendre com funciona el EDG de Swing.

La llibreria Swing de Java genera un thread principal que serà l'encarregat de gestionar tots els elements gràfics del programa. El problema sorgeix quan intentem executar el nostre emulador i a la vegada tot el sistema de finestres sense utilitzar threads nous.

El display és una imatge que es refresca cada 16.7 mil·lisegons. El sistema no és capaç d'executar tot a la vegada. Estem obligats a generar un thread per a la nostra lògica de l'emulador i un per al funcionament principal de l'aplicació.

5.5 Debugger

La implementació de la CPU és molt propensa a errors. És important tenir alguna espècie de debugger o test que ens permeti saber que està passant a l'emulador en cada moment.

En aquesta implementació a causa de la falta de temps no s'ha inclòs una explicació detallada d'aquest comportament. Al codi font del programa s'adjunten diferents classes que es van utilitzar per a mostrar el comportament dels diferents components.

Capítol 6

Planificació final i cost total

6.1 Planificació final

A mesura que el projecte va anar avançant es van produir molts canvis. El fet d'entendre millor l'abast del projecte va anar modificant enormement les fites programades. No hi ha una planificació final acurada perquè moltes de les tasques s'havien de portar en paral·lel o es desenvolupaven en situacions diferents.

A causa de la feina extra que va comportar, l'entrega del projecte es va ajornar per tal de deixar més marge de temps.

S'ha realitzat una planificació final per tal de poder comparar el temps inicial amb el final. Veure figura 6.1.

Al final hem vist incrementada la feina inicial programada amb unes 90 hores més, fent un total de 540 hores dedicades al projecte.

Diagrama de Gantt final

Inici projecte	1/3/21
Entrega projecte	
Total Hores (6h dia, 5d setmana)	540

Tasques	Durada(Dies)
Llegir/Planificació	10
Documentació / Mapa Conceptual	5
Modelat en classes	10
Implementació/Refactor	30
Afegir swing	10
Joc de proves	5
Redacció	10
Revisió/Imprevistos	10

Figura 6.1: Diagrama de Gantt final

6.2 Cost d'implementació

Els costos d'implementacions són molt variables en funció de les circumstàncies en les quals es troba el projecte. Probablement la variable més important en aquesta fórmula complicada seria quina és la quantitat d'informació sobre la GameBoy que tenim en el moment de dur a terme el projecte. Sense tota la documentació que hi ha escrita el cost seria incalculable, no és tant el temps d'implementació en codi sinó entendre el funcionament de tots els components.

Per altra banda, si ho calculem amb el supòsit que som al mateix moment que s'ha acomplert aquest projecte i que el programador té la mateixa habilitat podem fer una estimació de quin seria el pressupost del projecte.

Gràcies als costs de contingència, podem veure que el preu final no es desvia tant del inicial. Al final un increment d'un 20 per cent en un projecte on la majoria de despeses són de personal, incrementa molt el preu final.

A la figura 6.2 podem observar quin seria el preu total que seria convenient cobrar a l'empresa per tal de ser solvents com a programador autònom. En aquest cas es tracta de 15197 euros, els quals es poden variar molt en funció del sou que tinguem com a programadors.

Pressupost final

Temps total (hores)	540
Preu hora (euros)	20
Quota d'autònoms (euros)	289
Coworking (euros)	149
Mesos totals invertits	3,375

Costs de personal	Preu hora	Total
Programador	20	10800

Costos de programari	Total
	0

Costos de maquinari	Preu inicial	Total
	1000	281,25

Costos generals	Preu mensual	Total
Quota autònoms	289	975,375
Coworking	149	502,875
Total	438	1478,25

Total
12559,5

	Cost	IVA	Total
Preu a facturar	#REF!	0,21	15196,995

Figura 6.2: Cost Total

Capítol 7

Conclusions i Treball futur

7.1 Conclusions

La principal dificultat que m'he trobat a l'hora de dur a terme el projecte, ha estat el fet de planificar i saber valorar la quantitat de feina que suposava la implementació de l'emulador. Hi ha seccions que van requerir més temps del plantejat inicialment i han quedat com a implementacions futures.

A continuació, defineixo un conjunt de conceptes per tal de tenir una visió més concreta de quins eren els objectius plantejats inicialment i quin és l'estat en que es troben.

- **Prova de concepte:** Aquesta va ser la primera fase. La qual intentava aclarir dubtes sobre la planificació.
- **Components bàsics:** Es va donar prioritat als components crítics per tal de poder jugar a dos o més jocs. Aquí manquen apartats com el so, la connexió sèrie i més implementacions de tipus de joc. .
- **Gameboy funcional:** Aquesta era una fita que havia d'estar si o sí. Presentar un projecte sense aquest apartat suposava directament no mostrar un mínim producte viable. Aquesta és la situació actual de la implementació.
- **Gameboy Color:** Aquesta fase es va proposar en el moment d'escollir el títol del treball. Encara que la implementació no dista gaire de la Gameboy, era molt ambiciós i gairebé al principi del treball es va decidir prioritzar tota la resta.
- **Debugger:** Aquesta fase no s'ha acabat implementant. Una bona part del codi té classes enfocades a aquest apartat. El fet d'haver utilitzat singletons i alguna altra decisió sobre la visibilitat de les instàncies, va fer deixar enrere aquest apartat, ja que requeria més temps del que veritablement es disposava.

Encara que és més fàcil dir-ho que aplicar-ho, crec que no precipitar-se en la implementació i dedicar més temps a documentar-se i dissenyar afavoreix i simplifica el fet de programar l'aplicació.

Un altre aspecte complicat del treball ha estat que no és un tema molt nou. Hi ha moltíssimes implementacions ja realitzades i en gairebé tots els llenguatges i paradigmes que es puguin imaginar.

Una de les premisses de les quals partia aquest treball era realitzar una implementació totalment pròpia. Per tal d'aconseguir això és crucial el fet d'haver implementat una versió inicial de la CPU i dels components bàsics perquè defineixen una estructura molt marcada que després és impossible de canviar.

7.2 Valoració personal

Encara que no s'han pogut assolir tots el objectius que s'havien plantejat inicialment, el resultat del treball és totalment satisfactori. No s'ha aconseguit implementar el funcionament de la Gameboy Color ni de les opcions de debugger, però el simple fet de veure executar el Tetris per primer cop fa que tot l'esforç hagi valgut la pena.

Per tot el que resta puc dir que fer aquest projecte t'ensenya moltíssim. Des de com lidiar amb operacions una mica més baix nivell fins a abstraccions enrevessades. Estic orgullós de poder-me definir com una persona que té per afició el mateix que ha estudiat. El fet d'haver fet aquest projecte em proporciona un escenari ideal per fer mil proves i implementacions esbojarrades que queden massa lluny de la línia general del projecte per a ser incloses en implementacions futures. Un altre aspecte important és com és d'interessant el projecte. En l'àmbit personal m'ha motivat molt i sé que qualsevol altre projecte m'hagués cansat o avorrit molt abans.

Recomano a qualsevol persona interessada una mica amb el tema al fet que s'animi a intentar-ho. Hi ha moltíssimes variants, molt bona documentació i ofereix moltes idees noves que es poden aplicar.

7.3 Treball futur

Moltes característiques han quedat fora de l'abast d'aquest projecte. Algunes per complexitat i moltes per la falta de temps. En aquest apartat definirem els objectius principals que es voldrien aconseguir en una segona iteració d'aquest projecte.

7.3.1 Més configuracions de MBC

Hi ha jocs que no funcionen en aquest emulador perquè no estan implementats tots els tipus de cartutx que existeixen. Només estan implementats els principals i més senzills. Una de les implementacions que es vol aconseguir tenir en una futura versió és tot l'espectre de MBC fins al 5. El qual comprèn jocs com el Pokémon.

La implementació no és gaire complicada, però té elements nous com guardar el joc de manera persistent en una memòria RAM alimentada per bateria o un rellotge real.

7.3.2 Sistema de so

No tenir sistema de so va ser una de les primeres decisions que es van prendre a l'hora de dur a terme aquest projecte. Era factible no implementar res del sistema de so, ja que no interactua amb cap altre component de l'emulador.

En una segona iteració d'aquest projecte, seria un dels principals objectius a assolir, ja que és una part fonamental per a l'experiència d'usuari.

7.3.3 Cpu algorítmica

Des d'un principi es valorava la possibilitat que les diferents instruccions de la CPU es generessin de manera automàtica. Utilitzant un algoritme que seguís la lògica real que té el disseny del hardware.

No va ser fins que ja estava molt avançat el projecte que vaig trobar la informació de com es podia portar a terme. Aquesta implementació no afectarà l'usuari final, però simplificarà moltíssim la classe CPU i permetrà entendre de manera més senzilla quin és el seu funcionament.

7.3.4 Debugger i visió interna

Un dels objectius principals del projecte era la possibilitat que la gent aprengués i entengués com funcionen els diferents components. En un principi es volia tenir una opció per desenvolupadors que presentés d'una manera senzilla la informació de les diferents memòries, dels registres i de com es formen els gràfics.

Es marca com a objectiu el fet de crear una interfície per a tots els components interns per tal de visualitzar-los i modificar-los en temps d'execució.

7.3.5 Serial i connexió remota

Una de les connexions de la Gameboy és una connexió sèrie que permet connectar diferents videoconsoles entre elles. Això permet jugar o compartir dades en funció del joc.

En una segona versió del projecte es vol implementar aquesta lògica per tal d'intentar connectar dues instàncies de l'emulador en remot.

També seria interessant intentar crear un adaptador per tal de connectar una Gameboy física amb una emulada.

7.3.6 Gameboy Color

En un principi, aquest treball contemplava la implementació d'un emulador de la Gameboy Color. Al final només s'ha emulat la seva versió base, però és un dels objectius per a futures versions.

7.3.7 Guardar l'estat del emulador

Guardar l'estat del joc actual és una de les opcions que tenen gairebé tots els emuladors. En aquest treball no es va contemplar aquesta funció des d'un principi i la implementació requereix temps. S'han de serialitzar tots els objectes que guarden informació (Ram i registres) i guardar-los en un fitxer.

Per a una pròxima versió seria una bona funcionalitat a tenir en compte.

7.3.8 Iniciar un altre joc

La manera d'implementar alguns components utilitzant Singletons ha provocat que reiniciar tots els components no sigui trivial. Per la qual cosa la versió actual de l'emulador no permet iniciar un segon joc.

Es preveu reformular aquests components o implementar alguna funció reset per tal de resoldre el problema.

7.3.9 Android

L'única cosa bona que podem treure d'haver utilitzat Java com a llenguatge és que podem fer la portabilitat a Android amb relativa facilitat. Únicament haurem de reformular l'input i la interfície d'usuari, ja que tota la part lògica hauria de ser equivalent.

En futures versions es contemplaria l'opció de generar una apk.

Bibliografia

[1] Glassdoor. "Glassdoor." (2021), adr.: https://www.glassdoor.es/Sueldo.

[2] A. Tributaria. "Amortizaciones." (2021), adr.: https://www.agenciatributaria.es/AEAT.internet/Inicio/_Segmentos_/Empresas_y_profesionales/Empresas/Impuesto_sobre_Sociedades/Periodos_impositivos_a_partir_de_1_1_2015/Base_imponible/Amortizacion/Tabla_de_coeficientes_de_amortizacion_lineal_.shtml.

[3] Aureacoworking. "Aureacoworking precios." (2021), adr.: https://www.aureacoworking.com/precios-coworking/.

[4] author. "Game Boy." (), adr.: https://es.wikipedia.org/wiki/Game_Boy.

[5] L. S. Vailshery. "Video game console sales worldwide for products total lifespan as of September 2021." (jul. de 2021), adr.: https://www.statista.com/statistics/268966/total-number-of-game-consoles-sold-worldwide-by-console-type/.

[6] J. Frohwein. "DMG Schematics." (), adr.: https://gbdev.gg8.se/wiki/articles/DMG_Schematics.

[7] author. "DMG." (), adr.: http://dot-matrix-game.blogspot.com/2014/01/.

[8] M. Steil. "The Ultimate Game Boy Talk [slides]." (abr. de 2019), adr.: https://www.pagetable.com/?p=1099.

[9] I. Nazar. "GameBoy Emulation in Javascript: Memory." (ag. de 2010), adr.: http://imrannazar.com/GameBoy-Emulation-in-JavaScript:-Memory.

[10] A. N. Díaz, A. Vivace, Beannaich, E. ". H. Cory Sandlin, Elizafox, Furrtek, Gekkio, J. Frohwein, J. Harrison, L. ". Halphon, Mantidactyle, M. Fayzullin, M. ". Korth, P. of ATX, P. Felber, P. Robson, T4g1, TechFalcon, endrift, exezin, jrra, kOOPa, mattcurrie, nitro2k01, pinobatch, P. Fagan i A. Burnett. "Pan Docs." (), adr.: https://gbdev.io/pandocs/.

[11] A. N. Díaz. "The Cycle-Accurate GameBoy Docs." (), adr.: https://github.com/AntonioND/giibiiadvance/blob/master/docs/TCAGBD.pdf.

[12] A. N. Díaz, A. Vivace, Beannaich, E. ". H. Cory Sandlin, Elizafox, Furrtek, Gekkio, J. Frohwein, J. Harrison, L. ". Halphon, Mantidactyle, M. Fayzullin, M. ". Korth, P. of ATX, P. Felber, P. Robson, T4g1, TechFalcon, endrift, exezin, jrra, kOOPa, mattcurrie, nitro2k01, pinobatch, P. Fagan i A. Burnett. "Game Boy CPU (SM83) instruction set." (), adr.: https://gbdev.io/gb-opcodes/optables/.

[13] Gekkio. "Complete Technical Reference." (), adr.: https://gekkio.fi/files/gb-docs/gbctr.pdf.

[14] author. "Swing." (), adr.: https://docs.oracle.com/.

[15] ——, "Memory Bank Controllers." (), adr.: https://gbdev.gg8.se/wiki/articles/Memory_Bank_Controllers.

[16] ——, "title." (), adr.: ttps://stackoverflow.com/.

[17] I. Nazar. "GameBoy Emulation in JavaScript: Interrupt." (), adr.: http://imrannazar.com/GameBoy-Emulation-in-JavaScript:-Interrupts.

[18] T. Rekawek. "Why did I spend 1.5 months creating a Gameboy emulator?" (), adr.: http://blog.rekawek.eu/2017/02/09/coffee-gb/.

[19] ——, "Coffee-gb." (), adr.: https://github.com/trekawek/coffee-gb.

[20] S. Mansell. "DECODING Gameboy Z80 OPCODES." (), adr.: https://gb-archive.github.io/salvage/decoding%20gbz80%20opcodes/Decoding%20Gamboy%20Z80%20Opcodes.html.

[21] M. Steil. "The Ultimate Game Boy Talk." (), adr.: https://media.ccc.de/v/33c3-8029-the_ultimate_game_boy_talk.

[22] javidx9. "NES Emulator." (), adr.: https://www.youtube.com/channel/UC-yuWVUplUJZvieEligKBkA.

[23] Mattbruv. "Programming A Gameboy Color Emulator." (), adr.: https://mattbruv.github.io/gameboy-crust/.

[24] M. Fayzullin, P. Felber, P. Robson i M. Korth. "Pan Docs." (), adr.: https://bgb.bircd.org/pandocs.htm.

[25] author. "The Gameboy Emulator Development Guid." (), adr.: https://hacktixme.ga/GBEDG/ppu/.

[26] ISSOtm. "GBZ80." (), adr.: https://rgbds.gbdev.io/docs/v0.5.1/gbz80..

[27] "DMG-01." (), adr.: http://dot-matrix-game.blogspot.com/2014/01/.

Apèndix A

Guia d'execució

Per tal d'executar l'aplicació s'ha d'utilitzar la versió de Java 8 o superior. Aquest treball s'ha desenvolupat amb la versió 11.0.4 del JDK (Java Development Kit) i també s'ha provat amb la 16.

Podem utilitzar la següent comanda per a compilar el projecte

```
javac ./src/*/*.java -d classes
```

I la següent comanda per tal d'executar-lo

```
java -cp classes logic.Gameboy
```

En aquest treball s'inclou un arxiu .gb de mostra fet per la comunitat, ja que els jocs originals tenen drets d'autor i no es poden distribuir. Hi ha moltes pàgines web que distribueixen còpies dels jocs, com per exemple: https://www.romsgames.net/roms/gameboy/