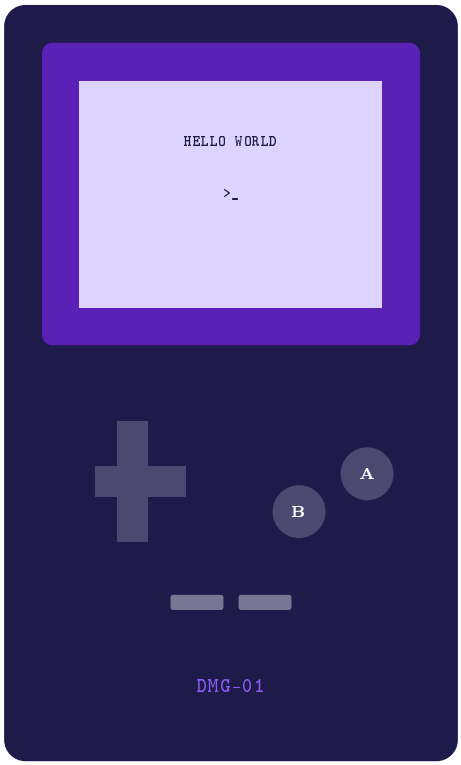


Écrire un Émulateur Game Boy

Du hardware au pixel, composant par composant

DMG (Game Boy classique) uniquement

KZ

Pour les iiens de l'association **PiiXeL**

Référence principale : *Pan Docs*
`https://gbdev.io/pandocs/`
C'est *la* doc. Garde-la ouverte. Tout le temps.

Table des matières

Comment lire ce guide

Ce guide est fait pour être lu **dans l'ordre**, du début à la fin. Chaque étape s'appuie sur la précédente. Si tu sautes des sections, tu vas manquer des trucs importants parce que les composants de la Game Boy sont interconnectés.

Les encadrés

Tu vas croiser plusieurs types d'encadrés dans ce guide. Chacun a un rôle précis :

▷ Pan Docs

Les encadrés **Pan Docs** te disent quoi lire dans la documentation officielle avant de commencer la section. Lis-les en premier, même en diagonale. C'est important d'avoir le contexte avant de lire mes explications.

Ce qu'il te faut

Les encadrés "Ce qu'il te faut" te donnent la **liste concrète** de ce que tu dois coder : les structures, les fonctions, les tableaux. C'est ta checklist d'implémentation pour chaque composant.

× Piège classique

Les "Piège classique" sont les trucs qui vont te faire perdre des heures si tu les connais pas. C'est les comportements du hardware qui sont contre-intuitifs ou mal documentés ailleurs. Lis-les attentivement, même si tu comprends pas tout de suite : tu y reviendras quand t'auras le bug.

△ Attention

Les encadrés "Attention" sont des warnings moins graves que les pièges, mais quand même importants. Des détails de timing, des ordres d'opérations, des trucs qui peuvent te bloquer.

Architecture

Les encadrés "Architecture" sont des conseils de design et de structure du code. C'est pas obligatoire de les suivre, mais ça t'évitera de réécrire des trucs plus tard.

Le workflow

Pour chaque étape, le workflow c'est toujours le même :

1. **Lire** la section Pan Docs indiquée (même rapidement)
2. **Lire** la section du guide pour comprendre les edge cases et les pièges
3. **Coder** en suivant l'encadré "Ce qu'il te faut"
4. **Tester** avec la ROM de test indiquée
5. Si ça passe pas, relire les pièges et les notes de la section. La réponse est souvent là.

À la fin du guide, il y a une **roadmap complète** (section 14) qui récapitule tout dans l'ordre : quoi coder, quel test lancer, quel résultat attendre, et où relire dans le doc si ça marche pas. Si t'es du genre à vouloir le plan d'ensemble avant de commencer, va la lire d'abord.

Pourquoi la Game Boy

En vrai, si tu veux apprendre comment un ordi fonctionne au niveau le plus bas, écrire un émulateur c'est probablement le meilleur exercice qui existe. Et la Game Boy c'est le point d'entrée parfait pour ça. Ce guide couvre la Game Boy originale, nom de code **DMG** (Dot Matrix Game), le modèle DMG-01 sorti en 1989.

Le hardware est simple : un CPU 8-bit, pas de pipeline, pas de cache, pas de MMU compliquée. Tout est documenté à mort grâce à la communauté. Et surtout, ya des ROMs de test pour vérifier que ton truc marche, donc tu peux avancer de manière itérative sans jamais être perdu.

Le but final : être capable de charger une ROM et d'y jouer. Et une fois que t'as ça, tu peux t'amuser avec du post-processing, des shaders, bref te faire plaisir sur la partie graphique.

Ce que tu vas construire

Ton émulateur c'est un programme qui va simuler le hardware de la Game Boy en logiciel. Concrètement, tu vas écrire du code qui :

- Lit les fichiers ROM (les jeux)
- Simule le processeur (CPU) instruction par instruction
- Simule la mémoire et le routing des adresses (Bus)
- Simule le chip graphique (PPU) qui dessine l'écran
- Simule le chip audio (APU) qui génère le son
- Simule le timer hardware
- Gère les entrées manette/clavier (Joypad)

Prérequis

Pas besoin d'être un expert. Par contre il te faut :

- Être à l'aise dans un langage (C, C++, Rust, Go, Java, peu importe)
- Comprendre le binaire et l'hexadécimal (si c'est pas le cas, passe 30 minutes dessus, c'est vite fait)
- Comprendre les opérations bit à bit : AND, OR, XOR, shift
- Avoir une base en assembleur, ça aide beaucoup (mais c'est pas bloquant)

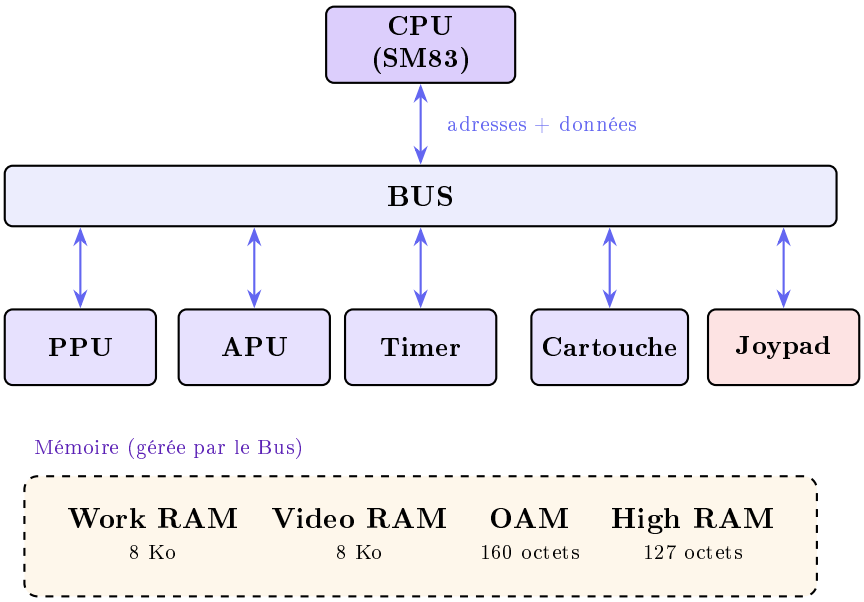
La doc : les Pan Docs

▷ Pan Docs

`https://gbdev.io/pandocs/`
C'est LA référence. Tout ce que tu as besoin de savoir sur le hardware de la Game Boy est dedans. Ce guide va te dire quoi lire et dans quel ordre, mais l'idée c'est que tu apprennes à naviguer dans les Pan Docs toi-même.
Un bon réflexe : quand t'as un doute sur un comportement, va chercher dans les Pan Docs avant de demander quelque part. 90% du temps la réponse y est.
Autre ressource utile : le *Game Boy CPU Manual* et le repo `gbdev/awesome-gbdev` sur GitHub.

Vue d'ensemble du hardware

Avant de coder quoi que ce soit, faut comprendre comment les composants de la Game Boy s'organisent entre eux. C'est pas compliqué, mais c'est important de voir le tableau complet.



Le CPU exécute les instructions du jeu. Chaque fois qu'il veut lire ou écrire quelque chose, il passe par le Bus. Le Bus regarde l'adresse demandée et redirige vers le bon composant : la ROM du jeu, la RAM de travail, les registres du PPU, etc.

Le PPU, l'APU et le Timer tournent en parallèle du CPU. À chaque cycle du CPU, tu fais aussi avancer ces composants.

Les specs en bref

Composant	Spec
CPU	Sharp SM83 (souvent appelé “Z80-like”), 4.19 MHz
Écran	160 × 144 pixels, 4 nuances de vert
VRAM (Video RAM)	8 Ko
Work RAM	8 Ko
OAM (Object Attribute Memory)	160 octets (40 sprites × 4 octets)
Cartouche	ROM variable + RAM optionnelle + MBC (Memory Bank Controller)
Audio	4 canaux (2 square, 1 wave, 1 noise)

Étape 0 : Architecture de ton projet

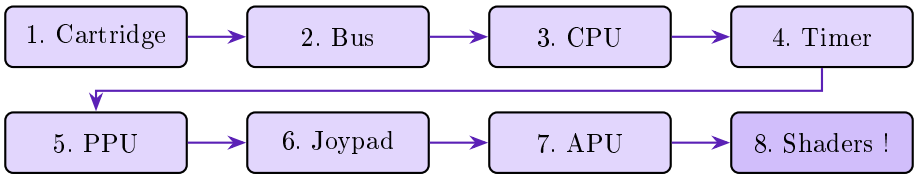
Avant d’écrire la moindre ligne, réfléchis à comment organiser ton code. Ton émulateur va avoir plusieurs modules qui correspondent aux composants hardware. Chaque module est relativement indépendant.

Architecture proposée

Tu vas avoir besoin de ces modules (un fichier ou une classe par module, selon ton langage) :

- **Cartridge** : Chargement et lecture de la ROM
- **Bus** : Routage mémoire (qui lit/écrit où)
- **CPU** : Le processeur, fetch-decode-execute
- **PPU** : Le chip graphique
- **APU** : Le chip audio
- **Timer** : Le timer hardware
- **Joypad** : Les entrées utilisateur
- **GameBoy** : Le module principal qui assemble tout

L’ordre dans lequel tu vas les implémenter est important. On va y aller comme ça :



Étape 1 : Charger la ROM (Cartridge)

▷ Pan Docs

Lis : **The Cartridge Header**
`https://gbdev.io/pandocs/The_Cartridge_Header.html`

Une ROM Game Boy c’est juste un fichier binaire. Un tableau d’octets. Ton premier boulot c’est de lire ce fichier et d’en extraire les infos du header.

Le header de la cartouche

Le header commence à l'adresse `0x0100` dans la ROM. C'est une zone fixe qui contient les métadonnées du jeu. Voici les champs importants :

Entry Point	Nintendo Logo (48 octets)	Titre du jeu
0x0100	0x0104–0x0133	0x0134–0x0143

Type	ROM sz	RAM sz	Checksums
0x0147	0x0148	0x0149	0x014D–0x014F

Ce qu'il te faut

- Une fonction qui **lit un fichier ROM** et stocke tout le contenu en mémoire (un tableau d'octets)
- Une fonction qui **parse le header** en extrayant : le titre (à partir de `0x0134`), le type de cartouche (`0x0147`), la taille ROM (`0x0148`), la taille RAM (`0x0149`)
- Une fonction qui **valide le logo Nintendo** : les 48 octets de `0x0104` à `0x0133` doivent correspondre à des valeurs précises (dans les Pan Docs). La vraie Game Boy refuse de booter si c'est pas bon.
- Une fonction qui **vérifie le header checksum** (`0x014D`) : c'est un checksum sur les octets `0x0134` à `0x014C`. La formule est dans les Pan Docs.

Le type de cartouche et les MBC

Le champ `0x0147` te dit quel type de cartouche c'est. Les plus simples ont juste de la ROM (type `0x00`), pas de bank switching. Mais la plupart des jeux utilisent un **MBC** (Memory Bank Controller) pour accéder à plus de 32 Ko de ROM.

△ Attention

Pour commencer, gère **uniquement le type** `0x00` (ROM only, pas de MBC). Ça suffit pour faire tourner les ROMs de test. Tu ajouteras le MBC1, MBC3, MBC5 plus tard quand tu voudras lancer de vrais jeux.

Les MBC les plus courants :

MBC	Types header	Ce que ça gère
Aucun	`0x00`	32 Ko ROM max, pas de RAM externe
MBC1	`0x01-0x03`	Jusqu'à 2 Mo ROM, 32 Ko RAM
MBC3	`0x0F-0x13`	2 Mo ROM, 32 Ko RAM, horloge temps réel
MBC5	`0x19-0x1E`	8 Mo ROM, 128 Ko RAM

Le principe du MBC c'est du bank switching : l'espace d'adressage de la Game Boy fait 64 Ko, mais la ROM peut être bien plus grosse. Le MBC permet de "mapper" différentes portions de la ROM dans la fenêtre visible par le CPU. Tu gères ça en interceptant les écritures dans la zone `0x0000-0x7FFF` (qui normalement est read-only). En fait ces écritures sont des commandes au MBC, pas de vraies écritures en ROM.

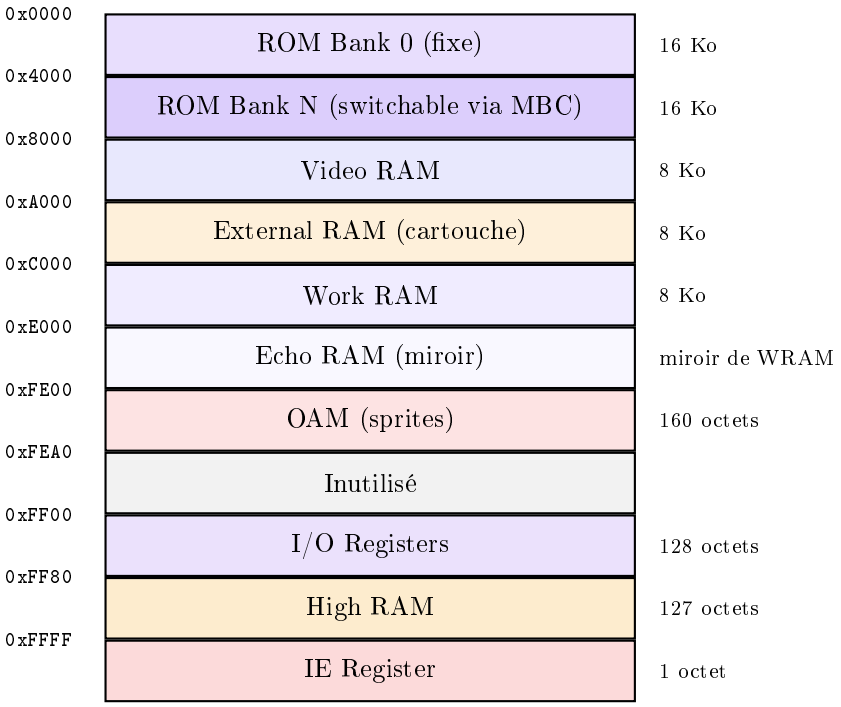
Étape 2 : La mémoire et le Bus

▷ Pan Docs

Lis : **Memory Map**
`https://gbdev.io/pandocs/Memory_Map.html`

Le CPU de la Game Boy voit un espace d'adressage de 16 bits, donc `0x0000` à `0xFFFF` (64 Ko). Mais c'est pas une grosse RAM linéaire : chaque plage d'adresses correspond à un composant différent.

La carte mémoire



Ce qu'il te faut

- Un module **Bus** avec deux fonctions principales : **Read(adresse)** qui retourne l'octet à cette adresse, et **Write(adresse, valeur)** qui écrit un octet.
- Dans ces fonctions, une série de conditions qui regarde dans quelle plage tombe l'adresse et redirige vers le bon composant.
- De la **Work RAM** : un tableau de 8192 octets.
- De la **High RAM** : un tableau de 127 octets.
- Un tableau pour les **I/O Registers** : 128 octets (`0xFF00-0xFF7F`).
- Un octet pour le registre **IE** (Interrupt Enable) à `0xFFFF`.

Concrètement, quand est-ce que tu fais ça ?

Tu as ta Cartridge qui sait lire la ROM. Maintenant il te faut un truc entre le CPU et la ROM pour que le CPU puisse dire "je veux l'octet à l'adresse `0x0150`" et que ça lui retourne le bon octet. C'est le Bus.

À ce stade, t'as pas encore de CPU. C'est pas grave. Tu vas quand même écrire le Bus parce que le CPU en aura besoin dès sa première instruction (il va faire un Read à l'adresse `0x0100` pour fetch

son premier opcode).

Ta fonction Read, au début, ressemble à ça (en pseudo-logique, pas du code) :

- Si l'adresse est entre `0x0000` et `0x7FFF`, tu lis dans la Cartridge (ta ROM)
- Si c'est entre `0xC000` et `0xDFFF`, tu lis dans la Work RAM
- Si c'est entre `0xFF80` et `0xFFFE`, tu lis dans la High RAM
- Pour tout le reste (VRAM, OAM, I/O, etc.), tu retournes `0xFF` pour l'instant. Tu brancheras les vrais composants quand tu les auras codés.

La fonction Write c'est pareil en miroir. Le Bus va grossir au fur et à mesure que tu ajoutes des composants (PPU, Timer, etc.), mais la structure ne change jamais.

Brancher les composants au fur et à mesure

Au début ton Bus retourne `0xFF` pour tout ce qu'il connaît pas encore. Au fil des étapes, tu vas remplacer ces `0xFF` par des vrais appels :

- **Après le Timer** : routage de `0xFF04-0xFF07` vers ton Timer (Read et Write)
- **Après le PPU** : `0x8000-0x9FFF` (VRAM) et `0xFE00-0xFE9F` (OAM) vers le PPU, plus les registres I/O `0xFF40-0xFF4B` (LCDC, STAT, SCY, etc.)
- **Après le Joypad** : `0xFF00` vers le Joypad
- **Après l'APU** : `0xFF10-0xFF3F` vers l'APU
- **Le registre IF** (`0xFF0F`) et **IE** (`0xFFFF`) : tu en as besoin dès les interruptions

Chaque fois que tu codes un nouveau composant, tu reviens dans ton Bus et tu ajoutes les lignes de routage. C'est mécanique : t'as juste un switch/if de plus à chaque fois.

Comment vérifier que ça marche

Avant même d'avoir un CPU, tu peux tester ton Bus :

- **Lire la ROM via le Bus** : appelle `Bus.Read(0x0104)` et vérifie que tu récupères bien le premier octet du logo Nintendo (qui vaut `0xCE`). Fais pareil avec `Bus.Read(0x0134)` pour le titre du jeu. Si les valeurs correspondent à ce que ta Cartridge avait parsé, ton routage ROM fonctionne.
- **Écrire puis relire la Work RAM** : écris une valeur quelconque à `0xC000` avec Write, puis relis-la avec Read. Tu dois retrouver exactement la même valeur. Teste à d'autres adresses dans la plage `0xC000-0xDFFF`.
- **Tester l'Echo RAM** : écris à `0xC000` et lis à `0xE000`. Tu dois obtenir la même valeur.
- **Tester la High RAM** : écris à `0xFF80` et relis. Même principe.
- **Les adresses non gérées** : lis à `0x8000` (VRAM que t'as pas encore). Ça doit retourner `0xFF` sans crasher.

Si tout ça passe, ton Bus est prêt. Tu peux passer au CPU.

L'Echo RAM

La zone `0xE000-0xFDFF` c'est un miroir de la Work RAM (`0xC000-0xDDFF`). Quand le CPU lit à `0xE000`, tu renvoies la même chose que `0xC000`. C'est un artefact hardware, pas un truc volontaire. Gère-le en soustrayant `0x2000` de l'adresse.

La zone inutilisée

`0xFEA0-0xFEFF` : retourne `0xFF` en lecture, ignore les écritures. Pas de prise de tête.

Étape 3 : Le CPU

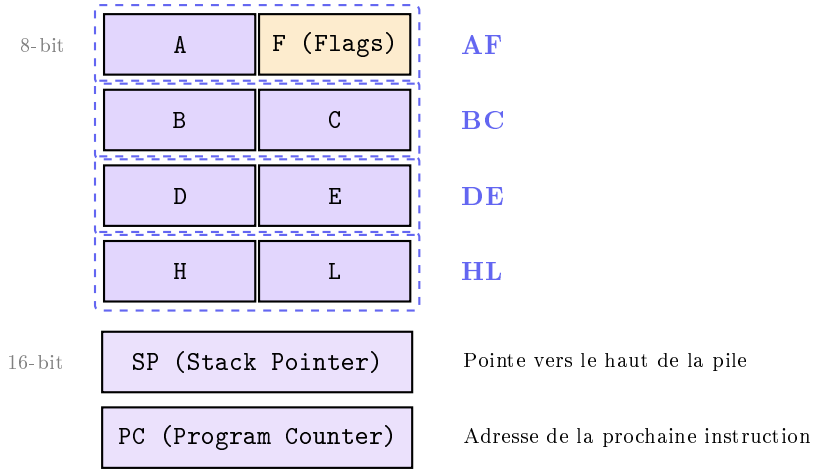
C'est le gros morceau. Le CPU de la Game Boy c'est un Sharp SM83. On l'appelle souvent "Z80-like" mais c'est pas exactement un Z80 : il a des instructions en moins et quelques-unes en plus. Fais attention à ça quand tu lis des docs, utilise les Pan Docs comme référence, pas une doc Z80.

▷ **Pan Docs**

Lis : **CPU Registers and Flags**
`https://gbdev.io/pandocs/CPU_Registers_and_Flags.html`
Puis : **CPU Instruction Set**
`https://gbdev.io/pandocs/CPU_Instruction_Set.html`
Et garde sous la main : `https://izik1.github.io/gbops/` (table d'opcodes interactive)

Les registres

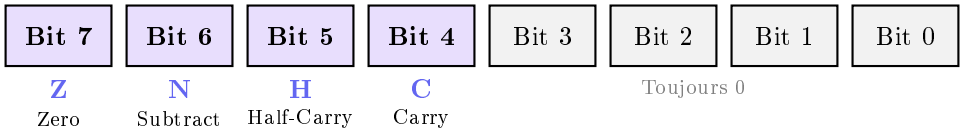
Le CPU a 8 registres 8-bit qui peuvent être utilisés par paires comme des registres 16-bit :



A c'est l'accumulateur : la plupart des opérations arithmétiques passent par lui. **HL** est souvent utilisé comme pointeur mémoire (quand une instruction dit `[HL]`, ça veut dire "l'octet à l'adresse pointée par HL").

Le registre Flags (F)

Le registre F n'est pas un registre normal. Tu n'y accèdes pas directement. Il contient 4 flags dans ses 4 bits de poids fort :



- **Z** (bit 7) : mis à 1 si le résultat d'une opération est zéro
- **N** (bit 6) : mis à 1 si la dernière opération était une soustraction

- **H** (bit 5) : Half-Carry, report du bit 3 au bit 4 (utile pour le BCD, Binary Coded Decimal : un format de nombre où chaque quartet code un chiffre 0-9)
- **C** (bit 4) : Carry, dépassement sur 8 bits (ou 16 bits pour certaines opérations)

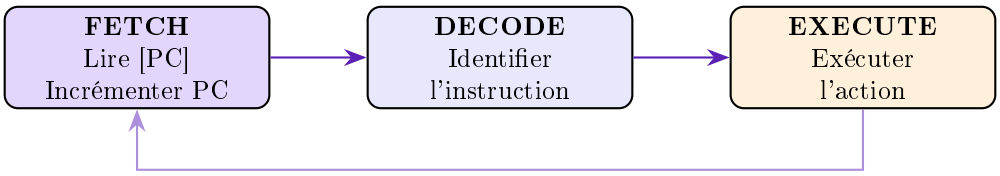
Les 4 bits de poids faible sont **toujours à 0**. Si tu fais un POP AF, masque les 4 bits bas.

Comment stocker les registres

Une approche classique : utilise une union (ou l'équivalent dans ton langage) pour pouvoir accéder aux registres individuellement (A, F) et par paire (AF) sans conversion. Si ton langage le permet pas, utilise des fonctions getter/setter qui combinent deux registres 8-bit en un 16-bit (et inversement).
Attention à l'endianness : la Game Boy est little-endian. Dans la paire AF, F est l'octet de poids faible et A l'octet de poids fort. Selon l'endianness de ta machine, l'ordre des champs dans ton union peut varier.

Le cycle Fetch-Decode-Execute

C'est le cœur de ton émulateur. Le CPU fait ça en boucle, indéfiniment :



Fetch : tu lis l'octet à l'adresse `PC` dans la mémoire (via le Bus), puis tu incrémentes `PC`. Cet octet c'est l'**opcode** : un nombre entre `0x00` et `0xFF` qui identifie l'instruction.

Decode : en fonction de la valeur de l'opcode, tu sais quelle instruction exécuter. Certaines instructions ont besoin d'un ou deux octets supplémentaires (les opérandes) que tu fetch aussi.

Execute : tu fais l'action. Ça peut être déplacer une valeur d'un registre à un autre, faire une addition, sauter à une adresse, etc.

Ce qu'il te faut

- Une structure/classe **CPU** qui contient tous les registres (AF, BC, DE, HL, SP, PC, les flags, IME, c'est-à-dire Interrupt Master Enable, le master switch des interruptions, et un flag "EI pending" pour le délai d'activation)
- Une fonction **Fetch** qui lit l'octet à [PC] via le Bus et incrémente PC
- Une fonction **Fetch16** qui fetch deux octets consécutifs et les combine en une valeur 16-bit little-endian (l'octet de poids faible d'abord, puis l'octet de poids fort). Beaucoup d'instructions en ont besoin (tous les loads 16-bit, les jumps absolus, les calls).
- Trois fonctions d'accès mémoire : **BusRead(adresse)** qui lit un octet et avance d'un M-cycle, **BusWrite(adresse, valeur)** qui écrit et avance d'un M-cycle, et **Tick()** pour les délais internes du CPU (certaines instructions ont des cycles "vides" où le CPU calcule sans toucher à la mémoire)
- Une fonction **Step** qui fetch un opcode puis l'exécute (un gros switch/match sur l'opcode)
- Le CPU a une référence vers le Bus pour pouvoir lire/écrire en mémoire

Les fonctions helper qui vont te sauver

Les opcodes utilisent un encodage régulier pour les registres. Tu vas en avoir besoin partout.
GetReg(index) et **SetReg(index, valeur)** : fonctions qui prennent un index 3-bit (0=B, 1=C, 2=D, 3=E, 4=H, 5=L, 6=[HL], 7=A) et retournent/modifient la bonne valeur. Le cas 6 est spécial : c'est un accès mémoire à l'adresse pointée par HL (donc un BusRead ou BusWrite).
GetReg16(index) et **SetReg16(index, valeur)** : même principe pour les paires. L'encodage 2-bit c'est 0=BC, 1=DE, 2=HL, 3=SP (ou AF pour PUSH/POP).
CheckCondition(cc) : les sauts et appels conditionnels encodent la condition dans les bits 4-3 de l'opcode. 0=NZ (pas Zero), 1=Z (Zero), 2=NC (pas Carry), 3=C (Carry).
Ces helpers rendent le gros switch d'opcodes beaucoup plus compact.

C'est quoi un opcode, concrètement

Un opcode c'est juste un nombre. Par exemple, l'opcode `0x3E` veut dire "charge une valeur immédiate dans A" (c'est un `LD A, n8`). Quand le CPU fetch `0x3E`, il sait qu'il doit fetch un deuxième octet (la valeur à charger), puis la mettre dans A.

Autre exemple : `0x47` c'est `LD B, A`, copier A dans B. Pas d'opérande supplémentaire, c'est juste un fetch et une copie.

Les valeurs initiales

Au démarrage (sans boot ROM), les registres ont des valeurs précises :

Registre	Valeur initiale (DMG)
AF	`0x01B0`
BC	`0x0013`
DE	`0x00D8`
HL	`0x014D`
SP	`0xFFFE`
PC	`0x0100`
IME	0 (désactivé)

`0x0100` c'est le point d'entrée : c'est là que la ROM commence à s'exécuter.

Lire une instruction

Dans les tables d'opcodes (Pan Docs, gbops, ce guide), tu vas voir des abréviations dans les instructions. C'est la notation standard qu'on retrouve partout dans la doc :

Notation	Ce que ça veut dire
`r`	Un registre 8-bit (A, B, C, D, E, H, L)
`rr`	Une paire de registres 16-bit (BC, DE, HL, SP ou AF)
`n8`	Un octet immédiat (la valeur est dans la ROM, juste après l'opcode)
`n16`	Deux octets immédiats (une valeur 16-bit, fetché avec Fetch16)
`a16`	Une adresse 16-bit (même format que n16, mais utilisé comme adresse)
`e8`	Un offset signé 8-bit (−128 à +127, pour les sauts relatifs)
`cc`	Une condition : NZ, Z, NC ou C
`[HL]` ou `(HL)`	L'octet en mémoire à l'adresse pointée par HL
`b`	Un numéro de bit (0 à 7)

Quand tu vois `LD A, n8`, ça veut dire "fetch l'opcode, puis fetch un octet de plus (le n8), et mets cette valeur dans A". Quand tu vois `JP a16`, c'est "fetch l'opcode, puis fetch 2 octets (l'adresse 16-bit), et saute à cette adresse".

Commence par ces opcodes

Implémente pas les 256 opcodes d'un coup. Commence par ceux-là, dans cet ordre. L'idée c'est de faire tourner les premières instructions d'une ROM de test.

Il existe des ROMs spécialement faites pour tester les émulateurs. Les plus connues c'est les **ROMs de test de Blargg**, en particulier `cpu_instrs` : 11 sous-tests qui vérifient chaque groupe d'instructions (`01-special.gb`, `06-ld r,r.gb`, `09-op r,r.gb`, etc.). On va les mentionner souvent dans ce guide. Tu les trouves ici : `https://github.com/retrio/gb-test-roms`. Plus de détails dans la section "Tester le CPU" plus loin.

Les NOP et les loads simples

Opcode	Instruction	Ce que ça fait
`0x00`	`NOP`	Rien. Avance juste PC.
`0x06`	`LD B, n8`	Charge l'octet suivant dans B
`0x0E`	`LD C, n8`	Charge l'octet suivant dans C
`0x16`	`LD D, n8`	Charge l'octet suivant dans D
`0x1E`	`LD E, n8`	Charge l'octet suivant dans E
`0x26`	`LD H, n8`	Charge l'octet suivant dans H
`0x2E`	`LD L, n8`	Charge l'octet suivant dans L
`0x36`	`LD (HL), n8`	Charge l'octet suivant à l'adresse [HL]
`0x3E`	`LD A, n8`	Charge l'octet suivant dans A
`0x47`	`LD B, A`	Copie A dans B
`0x4F`	`LD C, A`	Copie A dans C
`0x7F`	`LD A, A`	Copie A dans A (NOP en pratique)

Les LD r, n8 suivent un pattern : le registre destination est dans les bits 5-3 de l'opcode. Il y en a 8 (un par registre, y compris [HL]).

Le pattern des LD r, r'

Regarde bien la table d'opcodes (sur `https://izik1.github.io/gbops/`). Les opcodes de `0x40` à `0x7F` (sauf `0x76` qui est HALT) suivent un pattern régulier :

> Les bits 5-3 de l'opcode encodent le registre **destination**, et les bits 2-0 encodent le registre **source**.
> L'encodage des registres est toujours le même : 0=B, 1=C, 2=D, 3=E, 4=H, 5=L, 6=[HL], 7=A.
> Donc au lieu de coder 63 cas individuels, tu peux faire :

1. Extraire `destination = (opcode » 3) & 0x07`
2. Extraire `source = opcode & 0x07`
3. Lire la valeur du registre source (si c'est 6, lire depuis [HL] en mémoire)
4. Écrire dans le registre destination (si c'est 6, écrire à [HL])

Les opérations ALU (Arithmetic & Logic Unit)

Même logique pour les opcodes `0x80-0xBF` :

Opcodes	Opération	Description
0x80-0x87	ADD A, r	A = A + r, met à jour Z/N/H/C
0x88-0x8F	ADC A, r	A = A + r + carry
0x90-0x97	SUB r	A = A − r
0x98-0x9F	SBC A, r	A = A − r − carry
0xA0-0xA7	AND r	A = A & r
0xA8-0xAF	XOR r	A = A ^ r
0xB0-0xB7	OR r	A = A | r
0xB8-0xBF	CP r	Compare A avec r (comme SUB mais sans stocker)

Le pattern : `opération = (opcode » 3) & 0x07`, `source = opcode & 0x07`. Tu peux gérer tout ce bloc avec une seule fonction par opération.

Il y a aussi les **versions avec immédiat** : les opcodes `0xC6`, `0xCE`, `0xD6`, `0xDE`, `0xE6`, `0xEE`, `0xF6`, `0xFE` font la même chose que les opérations ci-dessus mais avec un octet immédiat (fetché après l'opcode) au lieu d'un registre. Même pattern : bits 5-3 donnent l'opération (ADD, ADC, SUB, SBC, AND, XOR, OR, CP). Tu peux réutiliser tes fonctions ALU, il suffit de changer la source.

Les flags pour chaque opération

C'est **crucial** de bien mettre les flags. C'est la source de bug #1 dans un émulateur Game Boy. Il te faut une fonction pour chaque opération (Add, Adc, Sub, Sbc, And, Or, Xor, Cp) qui modifie A et les flags correctement. Les Pan Docs et la table d'opcodes donnent les flags pour chaque instruction.
Détail :

- **ADD** : Z=(résultat==0), N=0, H=((A & 0x0F) + (val & 0x0F) > 0x0F), C=(résultat > 0xFF)
- **ADC** : pareil que ADD mais tu ajoutes aussi l'ancien carry dans le calcul de H et C. C'est `A + val + carry_flag` pour tout.
- **SUB** : Z=(résultat==0), N=1, H=((A & 0x0F) < (val & 0x0F)), C=(A < val)
- **SBC** : pareil que SUB mais tu soustrais aussi l'ancien carry. Attention à l'ordre du calcul pour H et C.
- **AND** : Z=(résultat==0), N=0, **H=1 toujours**, C=0. Le H=1 c'est une particularité du hardware, c'est pas un "vrai" half-carry.
- **XOR, OR** : Z=(résultat==0), N=0, H=0, C=0
- **CP** : exactement les mêmes flags que SUB, mais A n'est pas modifié

Les jumps et calls

Opcode	Instruction	Ce que ça fait
0xC3	JP a16	PC = adresse 16-bit (utilise Fetch16)
0xE9	JP HL	PC = HL (pas d'accès mémoire, instantané)
0x18	JR e8	PC = PC + offset signé (1 octet suivant)
0x20, 0x28, 0x30, 0x38	JR cc, e8	Saute si condition vraie
0xC2, 0xCA, 0xD2, 0xDA	JP cc, a16	Saute si condition vraie
0xCD	CALL a16	Push l'adresse de retour sur la pile, saute à l'adresse
0xC4, 0xCC, 0xD4, 0xDC	CALL cc, a16	CALL si condition vraie
0xC9	RET	Dépile une adresse dans PC
0xD9	RETI	RET + active IME immédiatement
0xC0, 0xC8, 0xD0, 0xD8	RET cc	RET si condition vraie

Les conditions sont encodées dans les bits 4-3 de l'opcode : 0=NZ, 1=Z, 2=NC, 3=C. C'est là que

ta fonction **CheckCondition** entre en jeu.

× Piège classique

`JR` utilise un offset **signé** sur 8 bits (−128 à +127). Fais gaffe au cast signé dans ton langage. Et l'offset est **relatif à l'adresse après le fetch de l'opérande**, pas au début de l'instruction.

△ Attention

Les sauts et calls conditionnels n'ont pas le même timing selon que la condition est vraie ou fausse. Si la condition est fausse, l'instruction prend moins de cycles (pas de saut, pas de push sur la pile). Vérifie le nombre de cycles dans la table d'opcodes, il y a souvent deux valeurs indiquées.

Les loads 16-bit, PUSH/POP, INC/DEC

Opcodes	Pattern	Ce que ça fait
`0x01`, `0x11`, `0x21`, `0x31`	`LD rr, n16`	Charge 16 bits dans une paire
`0x03`, `0x13`, `0x23`, `0x33`	`INC rr`	Incrémente une paire de registres
`0x0B`, `0x1B`, `0x2B`, `0x3B`	`DEC rr`	Décrémente une paire de registres
`0xC5`, `0xD5`, `0xE5`, `0xF5`	`PUSH rr`	Pousse une paire sur la pile
`0xC1`, `0xD1`, `0xE1`, `0xF1`	`POP rr`	Dépile dans une paire

Les bits 5-4 de l'opcode encodent la paire (0=BC, 1=DE, 2=HL, 3=SP ou AF selon le contexte).

Les instructions `LD rr, n16` utilisent ta fonction **Fetch16** pour lire l'opérande 16-bit. `INC rr` et `DEC rr` ne touchent à **aucun flag** (contrairement à INC/DEC 8-bit). `PUSH` décrémente SP de 2 puis écrit la paire (octet de poids fort d'abord, puis poids faible). `POP` fait l'inverse. N'oublie pas que PUSH et POP font chacun 2 accès mémoire (donc 2 M-cycles en plus du fetch).

ADD HL, rr et les opérations 16-bit spéciales

Opcodes	Instruction	Ce que ça fait
`0x09`, `0x19`, `0x29`, `0x39`	`ADD HL, rr`	HL = HL + paire (BC/DE/HL/SP)
`0xE8`	`ADD SP, e8`	SP = SP + offset signé 8-bit
`0xF8`	`LD HL, SP+e8`	HL = SP + offset signé 8-bit
`0x08`	`LD (a16), SP`	Écrit SP en mémoire (2 octets, low puis high)
`0xF9`	`LD SP, HL`	SP = HL

× Piège classique

Les flags de `ADD HL, rr` sont particuliers : Z n'est **pas modifié**, N=0, et H/C opèrent sur les 16 bits (H = report du bit 11 au bit 12, C = dépassement sur 16 bits).
`ADD SP, e8` et `LD HL, SP+e8` sont encore plus tordus : Z=0, N=0, et H/C sont calculés sur les **8 bits de poids faible** de SP + l'offset (comme si c'était une addition 8-bit). C'est un cas bizarre du hardware. Si tu galères sur `01-special.gb`, c'est sûrement à cause de ça.

INC r et DEC r (8-bit)

Groupe d'opcodes souvent oublié. Les INC 8-bit c'est `0x04`, `0x0C`, `0x14`, `0x1C`, `0x24`, `0x2C`, `0x34`, `0x3C`. Les DEC c'est `0x05`, `0x0D`, `0x15`, `0x1D`, `0x25`, `0x2D`, `0x35`, `0x3D`.

Le pattern : le registre cible est dans les bits 5-3 de l'opcode (même encodage que d'habitude). Le bit 0 distingue INC (0) de DEC (1).

△ Attention

Les flags de INC et DEC 8-bit sont piégeux :

- Z est mis à jour normalement
- N = 0 pour INC, 1 pour DEC
- H = half-carry (report du bit 3)
- **C n'est PAS modifié**. C'est un piège classique : INC et DEC ne touchent pas au carry.

Les loads mémoire spéciaux

Opcode	Instruction
`0x02`	`LD (BC), A` (écrit A à l'adresse pointée par BC)
`0x12`	`LD (DE), A` (écrit A à l'adresse pointée par DE)
`0x0A`	`LD A, (BC)` (lit l'octet à l'adresse BC dans A)
`0x1A`	`LD A, (DE)` (lit l'octet à l'adresse DE dans A)
`0x22`	`LD (HL+), A` (écrit A à [HL] puis incrémente HL)
`0x32`	`LD (HL-), A` (écrit A à [HL] puis décrémente HL)
`0x2A`	`LD A, (HL+)` (lit [HL] dans A puis incrémente HL)
`0x3A`	`LD A, (HL-)` (lit [HL] dans A puis décrémente HL)
`0xE0`	`LDH (n8), A` (écrit A à l'adresse `0xFF00`+n8)
`0xF0`	`LDH A, (n8)` (lit `0xFF00`+n8 dans A)
`0xE2`	`LDH (C), A` (écrit A à `0xFF00`+C)
`0xF2`	`LDH A, (C)` (lit `0xFF00`+C dans A)
`0xEA`	`LD (a16), A` (écrit A à une adresse 16-bit, utilise Fetch16)
`0xFA`	`LD A, (a16)` (lit une adresse 16-bit dans A)

Les instructions LDH (aussi notées LD avec `0xFF00`) sont très utilisées pour accéder aux registres I/O. C'est la façon principale dont le jeu communique avec le hardware.

Les rotations (hors CB)

Opcode	Instruction	Ce que ça fait
`0x07`	`RLCA`	Rotation gauche de A (bit 7 va dans Carry et bit 0)
`0x17`	`RLA`	Rotation gauche à travers Carry
`0x0F`	`RRCA`	Rotation droite de A (bit 0 va dans Carry et bit 7)
`0x1F`	`RRA`	Rotation droite à travers Carry

× Piège classique

Ces 4 rotations mettent **toujours Z=0, N=0, H=0**. Seul C est mis à jour (le bit qui sort). C'est différent des versions CB (RLC, RL, RRC, RR) qui mettent Z à 1 si le résultat est zéro. C'est une source de bugs subtils.

Instructions spéciales

Opcode	Instruction	Description
0x27	DAA	Ajustement décimal de A (correction BCD après ADD/SUB)
0x2F	CPL	Inverse tous les bits de A (N=1, H=1, Z et C inchangés)
0x37	SCF	Set Carry Flag (C=1, N=0, H=0, Z inchangé)
0x3F	CCF	Complement Carry Flag (C = !C, N=0, H=0, Z inchangé)
0x10	STOP	Arrête le CPU (utilisé pour le speed switch en CGB)
0xF3	DI	Désactive les interruptions (IME = 0, immédiat)
0xFB	EI	Active les interruptions (IME = 1, avec un délai)

`DAA` est la plus compliquée. C'est de l'arithmétique BCD (Binary Coded Decimal) : elle corrige le résultat d'une addition ou soustraction pour que chaque quartet (4 bits) représente un chiffre 0-9. L'algorithme dépend des flags N, H et C en entrée. En sortie : Z = (A == 0), N inchangé, H=0, C est mis à 1 s'il y a eu une correction au-delà de 99. Cherche "DAA implementation" dans la communauté gbdev, c'est un classique.

`STOP` (`0x10`) est une instruction 2 octets : le CPU lit aussi l'octet suivant (normalement `0x00`). Sur DMG c'est surtout utilisé pour mettre le CPU en veille. Tu peux juste le traiter comme un NOP au début, ça posera pas de problème pour les tests.

Les RST

Les opcodes `0xC7`, `0xCF`, `0xD7`, `0xDF`, `0xE7`, `0xEF`, `0xF7`, `0xFF` sont les instructions RST (Restart). C'est un CALL vers une adresse fixe encodée dans l'opcode (bits 5-3 × 8 : `0x00`, `0x08`, `0x10`, `0x18`, `0x20`, `0x28`, `0x30`, `0x38`). Plus rapide qu'un CALL normal car pas besoin de fetch l'adresse.

△ Attention

Il y a 11 opcodes "illégaux" qui n'existent pas sur le SM83 : `0xD3`, `0xDB`, `0xDD`, `0xE3`, `0xE4`, `0xEB`, `0xEC`, `0xED`, `0xF4`, `0xFC`, `0xFD`. Si le CPU tombe dessus, le vrai hardware se bloque. Dans ton émulateur, tu peux juste les ignorer ou afficher un warning. Si tu en rencontres un, c'est que t'as un bug ailleurs.

Le préfixe CB

L'opcode `0xCB` est spécial : c'est un préfixe. Quand le CPU fetch `0xCB`, il fetch un **deuxième octet** qui identifie l'instruction parmi 256 instructions supplémentaires. Ce sont des opérations bit à bit.

▷ Pan Docs

Regarde la table CB sur `https://izik1.github.io/gbops/` (onglet CB).

Les 256 opcodes CB se décomposent en 4 groupes de 64 :

Opcodes CB	Groupe	Description
0x00-0x07	RLC r	Rotation gauche (bit 7 va dans C et bit 0)
0x08-0x0F	RRC r	Rotation droite (bit 0 va dans C et bit 7)
0x10-0x17	RL r	Rotation gauche à travers Carry
0x18-0x1F	RR r	Rotation droite à travers Carry
0x20-0x27	SLA r	Shift gauche arithmétique (bit 7 dans C, bit 0 = 0)
0x28-0x2F	SRA r	Shift droite arithmétique (bit 0 dans C, bit 7 conservé)
0x30-0x37	SWAP r	Échange les 4 bits hauts et bas
0x38-0x3F	SRL r	Shift droite logique (bit 0 dans C, bit 7 = 0)
0x40-0x7F	BIT b, r	Teste le bit b du registre (Z = !bit, N=0, H=1, C inchangé)
0x80-0xBF	RES b, r	Met le bit b à 0
0xC0-0xFF	SET b, r	Met le bit b à 1

△ Attention

Pour toutes les rotations et shifts CB : Z = (résultat == 0), N=0, H=0, C = le bit qui est sorti. Exception : `SWAP` met C=0 (pas de bit sorti). Et `SRA` conserve le bit 7 (le signe), alors que `SRL` met le bit 7 à 0.
`BIT` ne modifie pas le registre, c'est juste un test. `RES` et `SET` ne touchent à aucun flag.

Les patterns de la table d'opcodes

△ Attention

Tu peux essayer de trouver les patterns tout seul en regardant la table d'opcodes. C'est un très bon exercice et c'est comme ça qu'on apprend le mieux. Mais si tu veux pas réfléchir ou si tu veux vérifier tes patterns, les voici.

La table d'opcodes de la Game Boy est très régulière. Quasiment tout peut se déduire de quelques bits :

Encodage des registres 8-bit (bits 2-0 ou 5-3 selon le contexte) : 0=B, 1=C, 2=D, 3=E, 4=H, 5=L, 6=[HL], 7=A. C'est toujours le même ordre, partout.

Encodage des paires 16-bit (bits 5-4) : 0=BC, 1=DE, 2=HL, 3=SP. Sauf pour PUSH/POP où 3=AF au lieu de SP.

Encodage des conditions (bits 4-3) : 0=NZ, 1=Z, 2=NC, 3=C.

Les blocs de la table principale :

- `0x00-0x3F` : les instructions "diverses". C'est le bloc le moins régulier, mais on retrouve quand même des patterns. Les colonnes x0/x8 c'est les LD rr/JR, x1 c'est LD rr n16, x2/xA les loads mémoire, x3/xB les INC/DEC rr, x4/xC les INC r, x5/xD les DEC r, x6/xE les LD r n8, x7/xF les rotations et DAA/CPL/SCF/CCF, x9 les ADD HL.
- `0x40-0x7F` : les LD r, r'. Destination = bits 5-3, source = bits 2-0. Exception : `0x76` c'est HALT, pas LD [HL], [HL].
- `0x80-0xBF` : les opérations ALU. Opération = bits 5-3 (ADD, ADC, SUB, SBC, AND, XOR, OR, CP), source = bits 2-0.

- `0xC0-0xFF` : les jumps, calls, returns, RST, et les cas spéciaux. Moins de patterns réguliers ici, mais les conditions (bits 4-3) et les paires (bits 5-4) sont là.

La table CB :

- Bits 7-6 : type d'opération (0 = rotates/shifts, 1 = BIT, 2 = RES, 3 = SET)
- Bits 5-3 : pour le groupe 0 c'est le type de rotation (RLC, RRC, RL, RR, SLA, SRA, SWAP, SRL). Pour les groupes 1-3 c'est le numéro de bit (0 à 7).
- Bits 2-0 : registre cible (toujours le même encodage)

Le timing (M-cycles)

Chaque instruction prend un certain nombre de **M-cycles** (Machine Cycles). Un M-cycle = 4 T-cycles (T = fréquence du cristal à 4.19 MHz). Les instructions prennent entre 1 et 6 M-cycles.

Le truc important : **chaque accès mémoire prend 1 M-cycle**. Donc une instruction qui fetch un opcode + lit un opérande + écrit un résultat prend au minimum 3 M-cycles.

Comment gérer le timing

À chaque M-cycle du CPU, tu dois aussi faire avancer le Timer, le PPU et l'APU. L'approche la plus simple :

- Ta fonction de lecture/écriture via le Bus fait aussi avancer les autres composants de 4 T-cycles (= 1 M-cycle)
- Les "cycles internes" (quand le CPU calcule sans accéder à la mémoire) font aussi un tick du Bus

Tu peux aussi simplement compter les cycles et faire tourner les autres composants après chaque instruction, mais c'est moins précis. Pour passer les tests de timing, il faut la première approche.

Les interruptions

▷ Pan Docs

Lis : **Interrupts**
`https://gbdev.io/pandocs/Interrupts.html`

Les interruptions c'est un mécanisme qui permet au hardware de dire au CPU "hé, ya un truc qui s'est passé, gère-le". Le CPU peut être interrompu par 5 sources :

Bit	Source	Adresse	Registre IF/IE
0	VBlank (PPU)	`0x0040`	Bit 0
1	LCD STAT (PPU)	`0x0048`	Bit 1
2	Timer overflow	`0x0050`	Bit 2
3	Serial (liaison série)	`0x0058`	Bit 3
4	Joypad (bouton pressé)	`0x0060`	Bit 4

Deux registres contrôlent les interruptions :

- **IF** (`0xFF0F`) : Interrupt Flag. Un bit à 1 = cette interruption est **demandée**
- **IE** (`0xFFFF`) : Interrupt Enable. Un bit à 1 = cette interruption est **autorisée**
- **IME** : flag interne du CPU (pas mappé en mémoire). C'est le master switch.

Le traitement des interruptions à chaque Step du CPU :

1. Si IME == 1 et que (IF & IE & `0x1F`) != 0 :
2. Désactiver IME
3. Pousser PC sur la pile
4. Mettre PC à l'adresse de l'interruption (la plus prioritaire = le bit le plus bas)
5. Effacer le bit correspondant dans IF
6. Tout ça prend 5 M-cycles

`EI` (opcode `0xFB`) active IME mais **avec un délai d'une instruction**. Concrètement, quand tu exécutes EI, tu mets un flag "EI pending" à vrai. C'est seulement *après* avoir exécuté l'instruction suivante que tu actives vraiment IME. Ça permet de faire `EI / RET` sans risquer d'être interrompu entre les deux.

Dans ta boucle CPU, ça donne :

1. Vérifier les interruptions (si IME == 1 et IF & IE != 0, traiter)
2. Si "EI pending" est vrai, mettre IME = 1 et "EI pending" = faux
3. Fetch-Decode-Execute l'instruction suivante (c'est là que EI met le flag pending)

L'ordre est important : tu résous le pending *avant* de fetch, mais *après* le check d'interruptions. Comme ça, quand EI est exécuté, une instruction complète s'exécute encore avant que IME passe à 1.

`DI` (`0xF3`) désactive IME immédiatement. `RETI` (`0xD9`) c'est comme RET mais active aussi IME immédiatement (pas de délai).

HALT

`HALT` (`0x76`) met le CPU en pause. Il arrête d'exécuter des instructions jusqu'à ce qu'une interruption soit demandée (IF & IE != 0). Ça consomme 1 M-cycle par itération en attendant.

× Piège classique

Le **HALT bug** : si on exécute HALT avec IME == 0 et qu'il y a déjà une interruption pending (IF & IE != 0), le CPU ne halt pas, mais le prochain octet fetché est lu deux fois (PC n'est pas incrémenté pour le premier fetch). C'est un bug hardware réel. Les ROMs de test le vérifient.

Tester le CPU

▷ Pan Docs

ROMs de test : `Blargg's cpu_instrs`
`https://github.com/retrio/gb-test-roms`

Les ROMs de test de Blargg c'est le graal. En particulier `cpu_instrs` qui a 11 sous-tests. Chaque sous-test vérifie un groupe d'instructions.

Ces ROMs envoient le résultat sur le port série. Ton Bus a un registre série à `0xFF01` (SB, données) et `0xFF02` (SC, contrôle). Quand le jeu écrit à `0xFF02` avec le bit 7 et le bit 0 à 1, il envoie l'octet de SB. Accumule ces octets en string et check si tu vois "Passed" ou "Failed".

Ordre de test recommandé

1. `01-special.gb` : les instructions spéciales (DAA, CPL, etc.)
2. `03-op sp,hl.gb` : opérations sur SP et HL
3. `04-op r,imm.gb` : opérations registre/immédiat
4. `05-op rp.gb` : opérations sur les paires de registres
5. `06-ld r,r.gb` : les loads registre à registre
6. `07-jr,jp,call,ret,rst.gb` : les sauts et appels
7. `08-misc instrs.gb` : instructions diverses
8. `09-op r,r.gb` : opérations registre/registre
9. `10-bit ops.gb` : les instructions CB (bit)
10. `11-op a,(hl).gb` : opérations A avec [HL]
11. `02-interrupts.gb` : les interruptions (fais celui-ci quand tu as le Timer)

Tu ne passeras probablement pas tout du premier coup. C'est normal. Chaque test qui échoue te dit quelle instruction pose problème. Corrige, relance, itère.

Débugger avec des logs de référence

Un moyen très efficace de trouver tes bugs : il existe des logs de référence pour les tests Blargg, générés par des émulateurs connus (comme Gameboy Doctor : `https://github.com/robert/gameboy-doctor`). L'idée c'est de loguer l'état de ton CPU à chaque instruction (PC, SP, registres, flags) et de comparer ligne par ligne avec le log de référence. Dès qu'il y a un mismatch, tu sais à quel moment l'état a divergé. Attention : le bug n'est pas forcément à cette instruction, il peut venir d'une instruction précédente dont le résultat incorrect n'a eu de conséquence visible que plus tard. Mais c'est un excellent point de départ pour remonter la piste.

Étape 4 : Le Timer

▷ Pan Docs

Lis : **Timer and Divider Registers**
`https://gbdev.io/pandocs/Timer_and_Divider_Registers.html`

Le Timer a l'air innocent mais il cache pas mal de comportements subtils. Tu en as besoin pour passer le test d'interruptions de Blargg, et les tests de timing vont vérifier les edge cases décrits ici.

Les registres

Adresse	Nom	R/W	Description
`0xFF04`	DIV	R/(W=reset)	Registre diviseur. S'incrémente à 16384 Hz. Écrire n'importe quoi le remet à 0.
`0xFF05`	TIMA	R/W	Timer counter. S'incrémente à la fréquence définie par TAC.
`0xFF06`	TMA	R/W	Timer modulo. Quand TIMA overflow, il est rechargé avec cette valeur.
`0xFF07`	TAC	R/W	Timer control. Bit 2 = enable, bits 1-0 = fréquence.

Comment ça marche en interne

En interne, c'est un compteur 16-bit qui s'incrémente à chaque T-cycle. DIV c'est les 8 bits de poids fort de ce compteur. TIMA s'incrémente quand un signal spécifique fait un falling edge (passe de 1 à 0).

Ce signal c'est un AND entre deux choses : le **bit sélectionné** du compteur interne et le **bit enable** de TAC (bit 2). Formellement : `signal = counter_bit AND tac_enable`. TIMA s'incrémente quand ce signal passe de 1 à 0.

TAC bits 1-0	Bit du compteur interne surveillé
00	Bit 9 (incrémente TIMA tous les 256 M-cycles)
01	Bit 3 (incrémente TIMA tous les 4 M-cycles)
10	Bit 5 (incrémente TIMA tous les 16 M-cycles)
11	Bit 7 (incrémente TIMA tous les 64 M-cycles)

Ce qu'il te faut

- Un compteur interne 16-bit qu'on incrémente à chaque T-cycle (donc 4 fois par M-cycle)
- DIV = compteur » 8
- Écrire à DIV remet le compteur à 0
- À chaque incrémentation : calculer le signal `counter_bit AND tac_enable` **avant** et **après** le changement. Si ça passe de 1 à 0 (falling edge), incrémenter TIMA.
- Si TIMA overflow (passe de `0xFF` à `0x00`), ne pas recharger immédiatement. Voir la section "Overflow delay" ci-dessous.

Les falling edges piégeux

× Piège classique

Puisque le signal est `counter_bit AND tac_enable`, un falling edge peut arriver dans plusieurs situations inattendues :

- **Écrire à DIV** remet le compteur interne à 0. Si le bit surveillé était à 1 et que le timer était activé, le signal passe de 1 à 0 : falling edge, TIMA s'incrémente.
- **Désactiver le timer** (écrire à TAC avec bit 2 = 0) alors que le bit surveillé est à 1 : le signal passe de `1 AND 1` à `1 AND 0` = 0. Falling edge, TIMA s'incrémente.
- **Changer la fréquence dans TAC** (bits 1-0) peut changer quel bit du compteur est surveillé. Si l'ancien bit était à 1 (et le timer activé) et que le nouveau bit est à 0, ça fait un falling edge. Par exemple, si tu passes de la fréquence 01 (bit 3) à 00 (bit 9) et que le bit 3 est à 1 mais le bit 9 à 0, TIMA s'incrémente.

L'overflow delay (le piège le plus vicieux)

Quand TIMA passe de `0xFF` à `0x00`, le rechargement depuis TMA et la demande d'interruption ne se font **pas immédiatement**. Il y a un **délai de 1 M-cycle** (4 T-cycles). Pendant ces 4 T-cycles, TIMA vaut `0x00`.

Ce délai crée plusieurs edge cases :

× Piège classique

- **Écrire à TIMA pendant le délai** (les 4 T-cycles entre l'overflow et le reload) **annule** le rechargement ET l'interruption. Ta valeur écrite reste dans TIMA.
- **Écrire à TMA pendant le délai** : la nouvelle valeur de TMA sera celle qui est chargée dans TIMA à la fin du délai (pas l'ancienne).
- **Écrire à TIMA le même cycle que le reload** (le cycle exact où TMA est copié dans TIMA) : c'est le reload qui gagne, ta valeur est écrasée.

Pour implémenter ça, tu as besoin d'un flag "overflow pending" et d'un compteur de cycles pour le délai. À chaque tick du timer :

1. Si le compteur de délai arrive à 0 et que "overflow pending" est vrai : copier TMA dans TIMA, demander l'interruption (IF bit 2 = 1), remettre le flag à faux.
2. Incrémenter le compteur interne, vérifier les falling edges, incrémenter TIMA si nécessaire.
3. Si TIMA vient d'overflow : mettre "overflow pending" à vrai, démarrer le délai de 4 T-cycles. TIMA reste à `0x00` en attendant.

Étape 5 : Le PPU (Graphiques)

▷ Pan Docs

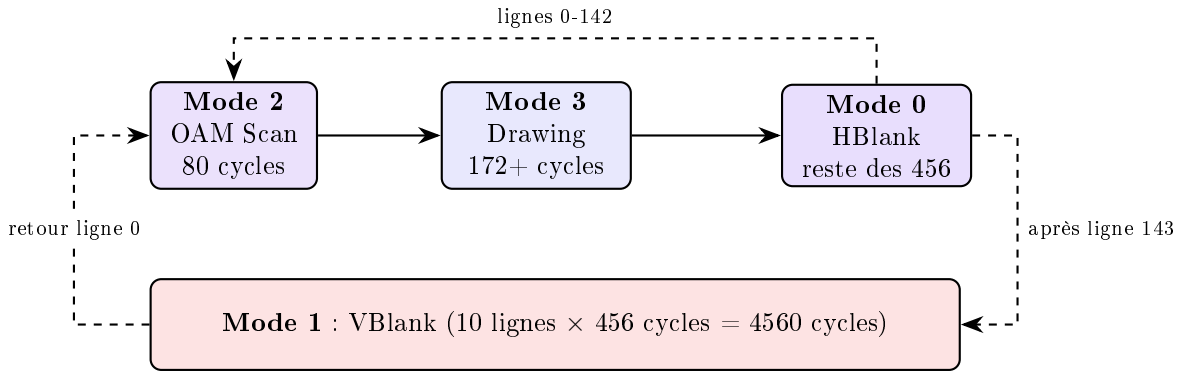
Lis dans l'ordre :

1. **Pixel Processing Unit** : `https://gbdev.io/pandocs/pixel_processing_unit.html`
2. **LCDC** : `https://gbdev.io/pandocs/LCDC.html`
3. **STAT** : `https://gbdev.io/pandocs/STAT.html`
4. **Tile Data** : `https://gbdev.io/pandocs/Tile_Data.html`
5. **Tile Maps** : `https://gbdev.io/pandocs/Tile_Maps.html`
6. **OAM** : `https://gbdev.io/pandocs/OAM.html`

Le PPU (Pixel Processing Unit) c'est le chip qui dessine l'écran. L'écran fait 160×144 pixels, et le PPU dessine une ligne (scanline) à la fois, de haut en bas. C'est le composant le plus complexe de la Game Boy et celui qui a le plus d'edge cases.

Les modes du PPU

Le PPU passe par 4 modes pour chaque scanline. Le total fait toujours 456 cycles par ligne :



- **Mode 2 (OAM Scan)** : 80 cycles fixes. Le PPU parcourt l'OAM pour trouver quels sprites sont sur cette ligne (max 10).
- **Mode 3 (Drawing)** : durée **variable** (172 à ~289 cycles). Le PPU dessine les pixels. Plus il y a de sprites sur la ligne et plus SCX % 8 est grand, plus c'est long.
- **Mode 0 (HBlank)** : ce qui reste pour arriver à 456. Le CPU peut accéder librement à la VRAM et l'OAM.
- **Mode 1 (VBlank)** : lignes 144-153. Le frame est prêt, le CPU peut tout faire.

Total : 456 × 154 = 70224 cycles par frame ≈ 59.7 FPS.

× Piège classique

La durée du Mode 3 n'est **pas fixe**. Elle dépend de : SCX % 8 (ajoute 0-7 cycles de penalty au début), le nombre de sprites sur la scanline (chaque sprite coûte 6-11 cycles en plus selon son alignement), et si la Window est active sur cette ligne (coûte 6 cycles de plus). Le Mode 0 absorbe le reste pour garder 456 au total. Si tu hardcodes le Mode 3 à 172 cycles, la plupart des jeux marcheront mais certains effets mid-scanline (comme les raster effects) casseront.

Accès VRAM et OAM selon le mode

C'est un point que beaucoup d'émulateurs ignorent au début, mais qui compte pour la précision :

Mode	VRAM	OAM
Mode 0 (HBlank)	accessible	accessible
Mode 1 (VBlank)	accessible	accessible
Mode 2 (OAM Scan)	accessible	**bloqué**
Mode 3 (Drawing)	**bloqué**	**bloqué**

Quand un composant est bloqué, les lectures retournent `0xFF` et les écritures sont ignorées. Tu peux commencer sans implémenter ça, mais certains jeux en dépendent (ils attendent le HBlank pour écrire en VRAM).

Le registre LCDC, LCD Control (`0xFF40`)

C'est LE registre qui contrôle tout le PPU. Chaque bit active/désactive un truc :

Bit	Nom	Effet
7	LCD Enable	0 = écran off, 1 = écran on
6	Window Tile Map	0 = `0x9800`, 1 = `0x9C00`
5	Window Enable	0 = window off, 1 = on
4	BG & Window Tile Data	0 = `0x8800-0x97FF` (signé), 1 = `0x8000-0x8FFF` (non signé)
3	BG Tile Map	0 = `0x9800`, 1 = `0x9C00`
2	OBJ Size	0 = 8×8, 1 = 8×16
1	OBJ Enable	0 = sprites off, 1 = on
0	BG Enable	0 = BG blanc, sprites toujours visibles

× Piège classique

- **Éteindre le LCD** (bit 7 passe de 1 à 0) : sur le vrai hardware, tu es censé ne faire ça que pendant le VBlank, sinon tu risques d'endommager l'écran. Quand le LCD s'éteint,

LY est remis à 0, le mode passe à 0, et les cycles sont reset. Quand tu le rallumes, le PPU redémarre depuis le début.

- **Bit 0 de LCDC sur DMG** : quand il est à 0, le background est rendu entièrement blanc (color 0 partout), mais les sprites sont toujours visibles et dessinés normalement. Les sprites ne vérifient plus la priorité BG puisque tout est color 0.

Les autres registres du PPU

Adresse	Nom	Rôle
`0xFF41`	STAT	Status du PPU : mode courant (bits 1-0, lecture seule), coincidence LY==LYC (bit 2, lecture seule), et 4 flags d'interruption STAT (bits 3-6, lecture/écriture)
`0xFF42`	SCY	Scroll Y : décalage vertical du background
`0xFF43`	SCX	Scroll X : décalage horizontal du background
`0xFF44`	LY	Numéro de la scanline courante (0-153). Lecture seule.
`0xFF45`	LYC	LY Compare : quand LY == LYC, le bit 2 de STAT est mis à 1
`0xFF46`	DMA	OAM DMA : écrire ici lance un transfert (voir section OAM DMA)
`0xFF47`	BGP	Palette du background
`0xFF48`	OBP0	Palette des sprites groupe 0 (color 0 = transparent)
`0xFF49`	OBP1	Palette des sprites groupe 1 (color 0 = transparent)
`0xFF4A`	WY	Position Y de la Window
`0xFF4B`	WX	Position X de la Window (position réelle = WX − 7)

Le registre STAT et ses interruptions

STAT (`0xFF41`) est plus subtil qu'il en a l'air. Les bits 3-6 permettent de déclencher une interruption STAT (interruption LCD, bit 1 de IF) dans 4 situations :

Bit	Condition	Quand
3	Mode 0 (HBlank)	à chaque passage en HBlank
4	Mode 1 (VBlank)	au début du VBlank
5	Mode 2 (OAM Scan)	au début de chaque OAM Scan
6	LY == LYC	quand la coincidence se produit

L'interruption STAT est levée quand une de ces conditions **passe de faux à vrai** (front montant). Un signal interne combine toutes les conditions en un seul OR : si le résultat global passe de 0 à 1, l'interruption est demandée.

× Piège classique

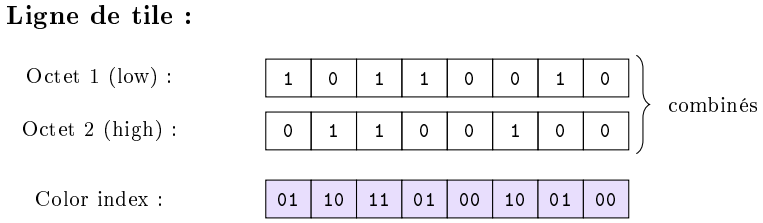
- **Le STAT bug (DMG uniquement)** : sur le vrai DMG, écrire à STAT (`0xFF41`) peut déclencher une fausse interruption STAT, même si aucune condition n'est vraie. C'est parce que l'écriture cause un glitch momentané sur la ligne d'interruption. Certains jeux

en dépendent. Pour émuler ça : quand le CPU écrit à STAT et que le LCD est allumé, lève une interruption STAT si le mode courant est 0 ou 1.
- **LY == LYC au changement de LY** : la comparaison LY == LYC est évaluée à chaque fois que LY change, pas à chaque cycle. Quand LY s'incrémente, le bit 2 de STAT est mis à jour et l'interruption est potentiellement levée. Si tu écris à LYC, la comparaison est aussi réévaluée.

Les tiles

L'écran de la Game Boy c'est pas un framebuffer (un tableau de pixels qu'on affiche directement) pixel par pixel. C'est un système basé sur des **tiles** : des blocs de 8×8 pixels. Le background est une grille de 32×32 tiles (256×256 pixels, dont seulement 160×144 sont visibles).

Chaque tile est encodé sur 16 octets (2 octets par ligne de 8 pixels). Chaque pixel a un index de couleur de 0 à 3, encodé sur 2 bits répartis entre deux octets :



Le bit de poids fort vient de l'octet 2, le bit de poids faible de l'octet 1. Le pixel le plus à gauche correspond au bit 7 (MSB).

Les deux modes d'adressage (bit 4 de LCDC) :

- **Mode non signé** (bit 4 = 1) : les tile index vont de 0 à 255, l'adresse de base est `0x8000`. Adresse = `0x8000` + tileIndex × 16.
- **Mode signé** (bit 4 = 0) : les tile index vont de −128 à 127 (signé), l'adresse de base est `0x9000`. Adresse = `0x9000` + `(S8)`tileIndex × 16. Le tile index 0 pointe vers `0x9000`, pas `0x8000`.

Les palettes (DMG)

Les color index (0-3) ne sont pas directement des couleurs. Ils passent par une palette. Le registre BGP (`0xFF47`) mappe chaque index vers une des 4 nuances :

Bits de BGP	Index	Nuance
1-0	Color 0	Typiquement le plus clair (“blanc”)
3-2	Color 1	Clair
5-4	Color 2	Foncé
7-6	Color 3	Le plus foncé (“noir”)

OBP0 et OBP1 ont le même format, mais le color 0 est toujours transparent pour les sprites (peu importe ce que la palette dit).

Le scrolling

Les registres SCX (`0xFF43`) et SCY (`0xFF42`) définissent le coin haut-gauche de la zone visible dans le background de 256×256 pixels. Le scrolling wrappe naturellement (si SCX + 160 dépasse 256, ça reprend au début).

Pour chaque pixel de la scanline courante :

1. Calculer la position dans le background : `bgX = (SCX + pixelX) & 0xFF`, `bgY = (SCY + LY) & 0xFF`
2. Trouver la tile : `tileX = bgX / 8`, `tileY = bgY / 8`
3. Lire l'index de tile dans la tile map (`0x9800` ou `0x9C00` selon LCDC bit 3)
4. Lire les données de la tile (selon le mode d'adressage, LCDC bit 4) et extraire le color index du pixel
5. Appliquer la palette BGP

> **△ Attention**
>
> SCX et SCY sont lus **au moment du rendu de chaque scanline**. Certains jeux changent SCX ou SCY mid-frame (entre les scanlines) pour faire des effets comme le "wobble" ou le split-screen.

Les sprites (OBJ)

L'OAM (Object Attribute Memory, `0xFE00-0xFE9F`) contient 40 entrées de sprites, chacune sur 4 octets :

Octet	Nom	Description
0	Y Position	Position Y + 16 (donc Y=16 = haut de l'écran, Y=0 = sprite caché au-dessus)
1	X Position	Position X + 8 (donc X=8 = bord gauche, X=0 = sprite caché à gauche)
2	Tile Index	Numéro de tile (en mode 8×16, le bit 0 est ignoré : les tiles vont par paires)
3	Attributes	Voir ci-dessous

Les bits d'attributs (octet 3) :

Bit	Effet
7	BG priority : si 1, le sprite est caché derrière les pixels BG de couleur 1-3
6	Y flip
5	X flip
4	Palette : 0 = OBP0, 1 = OBP1
3-0	Inutilisés sur DMG

Le processus de sélection des sprites (Mode 2, OAM Scan) :

Pendant les 80 cycles du Mode 2, le PPU parcourt les 40 entrées de l'OAM **dans l'ordre** (entrée 0 d'abord) et sélectionne les sprites dont la position Y chevauche la scanline courante (LY >= spriteY − 16 et LY < spriteY − 16 + spriteHeight). Il s'arrête dès qu'il en a trouvé **10**. Les sprites au-delà sont ignorés pour cette ligne, même s'ils devraient être visibles.

Priorité entre sprites (DMG) : les sprites sélectionnés sont triés par X croissant. Celui avec le X le plus petit est dessiné en dernier (donc au-dessus). Si deux sprites ont le même X, celui avec l'index OAM le plus petit gagne.

× Piège classique

Mode 8×16 (LCDC bit 2 = 1) : le tile index du sprite est masqué avec `0xFE` (le bit 0 est forcé à 0). Le sprite utilise deux tiles consécutives : tileIndex pour la moitié haute, tileIndex + 1 pour la moitié basse. Avec le Y flip, c'est inversé.

Algorithme de rendu des sprites dans DrawScanline :

1. Parcourir les 40 entrées OAM dans l'ordre. Pour chaque sprite dont la position Y chevauche LY, l'ajouter à ta liste (max 10).
2. Trier la liste par X croissant (en DMG). Si même X, l'index OAM le plus petit gagne.
3. Dessiner les sprites en **ordre inverse** (le dernier de la liste d'abord, le premier en dernier) pour que les plus prioritaires écrasent les autres.
4. Pour chaque sprite, pour chaque pixel (8 pixels de large) :
   - Calculer la ligne dans la tile : `row = LY - (spriteY - 16)`. Si Y flip, inverser : `row = spriteHeight - 1 - row`.
   - En mode 8×16, si row >= 8, utiliser tileIndex + 1 et row − 8.
   - Lire les 2 octets de la tile à cette ligne (adresse = tileIndex × 16 + row × 2)
   - Extraire le color index du pixel. Si X flip, inverser l'ordre des bits.
   - Si color index == 0, le pixel est transparent : ne rien dessiner.
   - Si le bit de priorité BG est à 1 et que le BG à cette position n'est pas color 0, ne pas dessiner (le sprite passe derrière).
   - Sinon, appliquer la palette (OBP0 ou OBP1 selon le bit 4 des attributs) et écrire dans le framebuffer.

La Window

La Window c'est comme un deuxième layer de background, mais elle ne scrolle pas avec SCX/SCY. Elle a sa propre position définie par WX (`0xFF4B`) et WY (`0xFF4A`). WX est décalé de 7 : la position réelle à l'écran est WX − 7.

La Window a son propre **compteur de ligne interne** (souvent appelé "window line counter"). C'est un détail critique :

- Le compteur ne s'incrémente que sur les scanlines où la Window est **effectivement rendue** (LY >= WY, la Window est activée dans LCDC, et WX − 7 < 160).
- Si la Window est désactivée puis réactivée mid-frame, le compteur reprend là où il s'était arrêté, pas à 0.
- Si WY est changé mid-frame pour une valeur <= LY, la Window ne s'affiche pas tant qu'on n'a pas atteint un nouveau frame où LY >= WY naturellement.

× Piège classique

WX = 0 à 6 : la position réelle serait négative (WX − 7 < 0), ce qui cause des glitchs sur le vrai hardware. Les quelques premiers pixels de la Window sont perdus. Certains jeux utilisent WX = 7 (position 0) pour couvrir tout l'écran avec la Window (typiquement pour les menus ou les HUD).

OAM DMA (0xFF46)

Écrire une valeur V à `0xFF46` lance un transfert de 160 octets depuis l'adresse V × `0x100` vers l'OAM (`0xFE00-0xFE9F`). Par exemple, écrire `0xC0` copie `0xC000-0xC09F` dans l'OAM.

△ Attention

Le transfert prend 160 M-cycles (640 T-cycles). Pendant ce temps, le CPU ne peut accéder qu'à la High RAM (`0xFF80-0xFFFE`). C'est pour ça que les jeux placent une petite routine de DMA en High RAM : ils écrivent à `0xFF46` depuis la High RAM puis attendent avec une boucle. Pour commencer, tu peux implémenter le DMA comme un transfert instantané (copie les 160 octets d'un coup). Ça marchera pour la grande majorité des jeux.

Afficher l'écran

Pour afficher le framebuffer, tu as besoin d'une bibliothèque graphique. SDL2 (Simple DirectMedia Layer) est le choix classique : tu crées une fenêtre, une texture, et tu copies ton framebuffer dedans à chaque VBlank.

L'idée c'est : ta boucle principale fait tourner le CPU jusqu'à ce qu'un frame soit prêt (le PPU signale un VBlank), puis tu affiches le framebuffer et tu recommences.

△ Attention

Tu dois **forcer le rythme à ∼59.7 FPS**. Sans ça, ton émulateur va tourner aussi vite que possible (des milliers de FPS) et le jeu sera injouable. Deux approches :

- **VSync** : active la synchronisation verticale dans SDL (`SDL_RENDERER_PRESENTVSYNC`). Si ton écran est à 60 Hz, c'est quasi parfait (59.7 vs 60 Hz, la différence est négligeable).
- **Sleep manuel** : mesure le temps écoulé par frame et fais un `SDL_Delay` pour compléter les ∼16.74 ms (1/59.7). Moins précis mais marche sur tous les écrans.

Si tu ne fais ni l'un ni l'autre, l'audio sera aussi accéléré et tout sera désynchronisé.

Ce qu'il te faut pour le PPU

- 8 Ko de **VRAM** (`0x8000-0x9FFF`)
- 160 octets d'**OAM** (`0xFE00-0xFE9F`)
- Tous les registres listés plus haut (LCDC, STAT, SCY, SCX, LY, LYC, DMA, BGP, OBP0, OBP1, WY, WX)
- Un compteur de cycles, un état de mode (0-3), et le **window line counter**
- Une fonction **Tick** qui avance le PPU et gère les transitions de mode
- Une fonction **DrawScanline** appelée à la fin du mode Drawing
- Un framebuffer de 160×144 pixels
- La gestion des interruptions VBlank et STAT (avec le signal combiné)
- Un tableau de 160 "color index BG" par scanline pour la priorité sprites/BG

L'algorithme du Tick

Ta fonction Tick reçoit le nombre de M-cycles écoulés et fait avancer le PPU. Voici la logique concrète :

1. Si le LCD est éteint (LCDC bit 7 = 0) : juste compter les cycles pour le frame timing (70224 cycles par frame) et signaler FrameReady. Pas de changement de mode, pas d'interruption. Return.

2. Ajouter les cycles au compteur interne.
3. Selon le mode courant :
   - **Mode 2 (OAM Scan)** : quand le compteur atteint 80, passer en Mode 3. L'interruption STAT Mode 2 est levée à *l'entrée* en Mode 2, si le bit 5 de STAT est actif.
   - **Mode 3 (Drawing)** : à 80 + durée du drawing, appeler `DrawScanline` et passer en Mode 0. STAT interrupt si bit 3 actif.
   - **Mode 0 (HBlank)** : à 456 cycles, remettre à 0, incrémenter LY. Si LY == 144 : Mode 1, VBlank interrupt + STAT si bit 4. Sinon : Mode 2, STAT si bit 5.
   - **Mode 1 (VBlank)** : quand le compteur atteint 456, remettre à 0 et incrémenter LY. Si LY > 153, remettre LY à 0, le window line counter à 0, passer en Mode 2, signaler FrameReady.
4. Mettre à jour les bits 0-1 de STAT avec le mode courant.
5. Vérifier LY == LYC : si oui, mettre le bit 2 de STAT à 1. Si le bit 6 de STAT est actif, lever l'interruption STAT. Si LY != LYC, remettre le bit 2 à 0.

△ Attention

Pour les interruptions STAT, rappel : tu dois détecter le **front montant** du signal combiné (le OR de toutes les conditions actives). Si tu lèves bêtement une interruption à chaque check, tu vas en lever trop. L'approche simple : garde en mémoire l'ancien état du signal. Si l'ancien est 0 et le nouveau est 1, là tu lèves l'interruption.

Tester le PPU

Pas de Blargg pour le PPU (ses tests PPU existent mais sont moins utilisés). Les meilleures ROMs de test :

ROMs de test pour le PPU

- **dmg-acid2** : `https://github.com/mattcurrie/dmg-acid2`. C'est LE test visuel. Il affiche une image de référence, et si ton PPU est correct, tu dois voir exactement la même chose. Il teste le BG, la Window, les sprites, les priorités, les flips, le mode 8×16, les palettes. Si ya un pixel qui diffère, un truc est cassé. Le repo donne l'image de référence pour comparer.
- **Un jeu simple en premier** : avant les tests formels, charge Tetris (pas de MBC). Si tu vois le background du menu, t'es sur la bonne voie. Les sprites viendront après.

Debug visuel du PPU

Un truc très utile : affiche un "tile viewer" à côté de l'écran. C'est un deuxième rendu qui montre toutes les tiles en VRAM (les 384 tiles) dans une grille. Ça te permet de voir si les tiles sont bien décodées avant même de t'occuper de la tile map et du scrolling. Si les tiles sont correctes mais l'écran est cassé, le problème est dans ton scrolling ou ta tile map. Si les tiles sont déjà n'importe quoi, le problème est dans ton décodage 2bpp.

Étape 6 : Le Joypad

▷ Pan Docs

Lis : **Joypad Input**
`https://gbdev.io/pandocs/Joypad_Input.html`

Le joypad c'est le plus simple de tous les composants. Un seul registre à `0xFF00`.

Le principe : la Game Boy a 8 boutons (A, B, Start, Select, Haut, Bas, Gauche, Droite) mais seulement 4 bits de données. Donc les boutons sont divisés en 2 groupes :

- **Groupe direction** (sélectionné quand bit 4 = 0) : Right, Left, Up, Down
- **Groupe action** (sélectionné quand bit 5 = 0) : A, B, Select, Start

Le jeu écrit à `0xFF00` pour sélectionner quel groupe lire, puis lit `0xFF00` pour avoir l'état des boutons. Un bouton pressé = 0, relâché = 1 (logique inversée).

Ce qu'il te faut

- Un état interne qui stocke quels boutons sont pressés (8 flags)
- Quand le jeu écrit à `0xFF00` : stocker la valeur (bits 4-5 = sélection du groupe)
- Quand le jeu lit `0xFF00` : retourner l'état des boutons du groupe sélectionné
- Mapper les touches du clavier (ou les boutons d'une manette) vers les boutons Game Boy

Étape 7 : L'APU (Audio)

▷ Pan Docs

Lis : **Audio**
`https://gbdev.io/pandocs/Audio.html`
Et chaque sous-page pour les canaux individuels.

L'APU c'est optionnel pour commencer (tu peux jouer sans son), mais ça ajoute beaucoup à l'expérience. La Game Boy a 4 canaux audio :

Canal	Type	Registres	Description
1	Square + Sweep	NR10-NR14	Onde carrée avec pitch sweep (la fréquence monte ou descend auto.)
2	Square	NR21-NR24	Onde carrée simple
3	Wave	NR30-NR34	Forme d'onde custom (32 samples 4-bit)
4	Noise	NR41-NR44	Bruit (LFSR)

Plus 3 registres de contrôle global : NR50 (volume master), NR51 (panning gauche/droite), NR52 (power on/off et status des canaux).

Comment marche un canal square

C'est le plus simple à comprendre, et les canaux 1 et 2 sont quasi identiques (le canal 1 a le sweep en plus).

Un canal square génère une onde carrée à une fréquence donnée. Le **duty cycle** détermine la forme de l'onde, c'est-à-dire quelle proportion du cycle est "haute" :

Duty	Pattern (8 steps)
0 (12.5%)	0 0 0 0 0 0 0 1
1 (25%)	0 0 0 0 0 0 1 1
2 (50%)	0 0 0 0 1 1 1 1
3 (75%)	1 1 1 1 1 1 0 0

Le canal a un **frequency timer** qui se décrémente à chaque T-cycle. Quand il atteint 0, il est rechargé avec `(2048 − frequency) × 4` et la position dans le pattern avance d'un step ($0 \to 1 \to 2 \to \ldots \to 7 \to 0$). La sortie du canal c'est le bit courant du pattern (0 ou 1) multiplié par le volume courant (0-15).

Le Frame Sequencer

C'est un timer central qui cadence les composants lents de l'APU. Il tourne à 512 Hz (incrémenté tous les 8192 T-cycles) et il a 8 steps (0-7). Chaque step déclenche des actions :

Step	Action
0	Length counter
1	(rien)
2	Length counter + Sweep
3	(rien)
4	Length counter
5	(rien)
6	Length counter + Sweep
7	Envelope

Length counter : un compteur qui, quand il atteint 0, éteint le canal. Il est décrémenté sur les steps pairs (0, 2, 4, 6) si le bit "length enable" est actif dans NRx4.

Envelope : modifie le volume du canal progressivement. À chaque tick d'envelope (step 7), si le period est non-nul, un compteur interne se décrémente. Quand il atteint 0, le volume monte ou descend de 1 (selon la direction dans NRx2) et le compteur est rechargé. Le volume est clampé entre 0 et 15.

Sweep (canal 1 seulement) : modifie la fréquence du canal automatiquement. À chaque tick de sweep (steps 2 et 6), la fréquence est recalculée : `newFreq = freq ± (freq » shift)`. Si newFreq dépasse 2047, le canal est coupé.

Le canal Wave (canal 3)

Il lit des samples dans une table de 32 entrées 4-bit stockées dans la Wave RAM (`0xFF30-0xFF3F`, 16 octets, chaque octet contient 2 samples). Le frequency timer avance la position dans la table. Le volume est contrôlé par un shift (NR32 bits 6-5) : 0 = muet, 1 = 100%, 2 = 50% (shift 1), 3 = 25% (shift 2).

Le canal Noise (canal 4)

Il utilise un LFSR (Linear Feedback Shift Register) de 15 bits. À chaque tick du frequency timer, le LFSR avance : XOR les bits 0 et 1, injecter le résultat dans le bit 15 (et optionnellement dans le bit 7 si le mode "narrow" est actif, ce qui donne un son plus métallique). La sortie c'est le bit 0 inversé, multiplié par le volume.

Générer un sample audio

Tu dois produire un sample audio à intervalles réguliers pour alimenter ton backend audio. À 44100 Hz, tu génères un sample environ tous les 95 T-cycles. Avec SDL, `SDL_QueueAudio` fait le boulot.

L'algorithme de mixage pour chaque sample :

1. Pour chaque canal actif, récupérer sa sortie (0-15)
2. Appliquer le panning : NR51 dit quels canaux vont à gauche et/ou à droite
3. Sommer les canaux pour chaque côté (gauche et droite séparément)
4. Appliquer le volume master (NR50, bits 6-4 pour gauche, 2-0 pour droite, valeur 0-7)
5. Convertir en sample flottant (-1.0 à 1.0) ou en entier 16-bit selon ton backend

△ Attention

La synchronisation audio est le vrai défi. Si tu génères trop de samples, le buffer audio explose et le son prend du retard. Pas assez, ça craquèle. L'approche la plus simple : accumule un compteur de T-cycles. Chaque fois qu'il dépasse 95 (environ), génère un sample et push-le dans un buffer. Quand le frame est fini, envoie le buffer à SDL_QueueAudio. Tu peux aussi utiliser la taille du buffer audio comme signal de timing : si le buffer est trop plein, ralentis l'émulation, s'il est presque vide, accélère.

Ce qu'il te faut

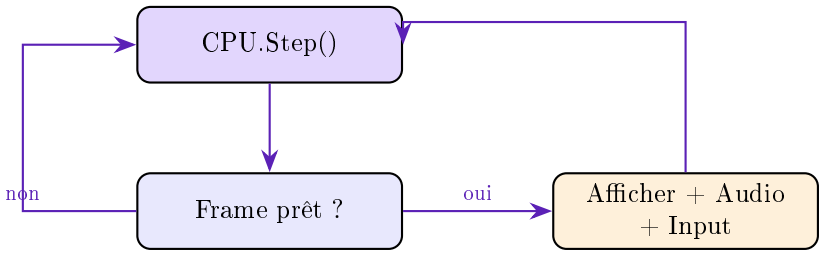
- Une structure par canal avec ses registres et son état interne (frequency timer, duty position, volume, envelope counter, length counter, LFSR pour le noise)
- Le **Frame Sequencer** : un compteur qui tourne à 512 Hz et dispatche les length/envelope/sweep
- Une fonction **Tick** globale de l'APU appelée à chaque M-cycle, qui fait avancer les frequency timers des 4 canaux et le frame sequencer
- Un compteur de **sample** : tous les ∼95 T-cycles, mixer les 4 canaux et pousser un sample dans le buffer
- Un buffer audio à envoyer au backend (SDL_QueueAudio)
- La gestion du **trigger** : quand le jeu écrit le bit 7 de NRx4, réinitialiser le canal (timer, volume, length counter, etc.)

Étape 8 : Assembler le tout

La boucle principale

Ton module principal (GameBoy) instancie tous les composants et les connecte :

1. Créer : Cartridge, Timer, PPU, APU, Bus, CPU
2. Le Bus a des références vers la Cartridge, le Timer, le PPU, l'APU
3. Le CPU a une référence vers le Bus
4. La boucle principale :
   - **CPU.Step()** : exécute une instruction complète (fetch + decode + execute, qui ticks le Bus à chaque M-cycle)
   - Vérifier si un frame est prêt (le PPU signale FrameReady)
   - Si oui : afficher le framebuffer, envoyer l'audio, traiter les événements clavier
   - Recommencer



Les MBC (pour les vrais jeux)

Une fois que les tests Blargg passent, tu voudras lancer des vrais jeux. La plupart utilisent un MBC.

Implémenter MBC1

C'est le plus courant. Le principe :

- `0x0000-0x1FFF` : écrire `0x0A` active la RAM externe, autre chose la désactive
- `0x2000-0x3FFF` : sélectionne le numéro de ROM bank (5 bits bas). Si la valeur est 0, elle devient 1.
- `0x4000-0x5FFF` : sélectionne le bank RAM (2 bits) ou les bits hauts du bank ROM
- `0x6000-0x7FFF` : sélectionne le mode banking (0 = ROM, 1 = RAM)

Quand le CPU lit dans `0x4000-0x7FFF`, tu retournes les données de la ROM au bon offset : `romBank * 0x4000 + (adresse - 0x4000)`.

Les sauvegardes

Les jeux avec batterie (type `0x03`, `0x13`, etc.) ont de la RAM externe qui persiste quand on éteint. Pour émuler ça, quand le jeu quitte, sauvegarde le contenu de la RAM externe dans un fichier `.sav`. Au prochain lancement, recharge-le.

Les save states

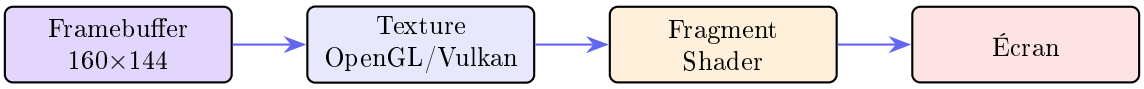
Bonus : tu peux sauvegarder l'état complet de l'émulateur (tous les registres, toute la RAM, l'état du PPU, etc.) dans un fichier. Ça permet de reprendre exactement où t'en étais. C'est juste une sérialisation de toutes tes structures.

Étape 9 : Les shaders !

Une fois que ton émulateur marche et affiche le framebuffer dans une texture, t'es dans la position parfaite pour ajouter des shaders en post-processing.

Le principe

Le PPU produit une image de 160×144 pixels. Tu la mets dans une texture. Normalement tu affiches cette texture directement. Avec des shaders, tu la passes d'abord par un (ou plusieurs) fragment shaders qui modifient chaque pixel.



Idées de shaders

LCD Grid : simule la grille de pixels de l'écran Game Boy original. Tu assombris les bords de chaque pixel pour créer l'effet de séparation.

CRT Scanlines : ajoute des lignes horizontales sombres, un léger barrel distortion, du vignetting.

Color Correction : transforme les 4 nuances de vert en d'autres palettes. Sépia, noir et blanc, palette Game Boy Pocket, palette personnalisée.

Glow / Bloom : ajoute un halo lumineux autour des pixels clairs.

Upscaling : des algorithmes comme xBR, HQx, ou Scale2x qui font de l'upscaling intelligent pour adoucir les pixels.

Aberration chromatique : décale légèrement les canaux R, G, B pour un effet rétro.

Setup technique

Si tu utilises SDL2 pour l'affichage, tu peux soit passer à SDL2 + OpenGL (créer un contexte GL dans la fenêtre SDL), soit utiliser un framework comme GLFW directement. L'idée :

1. Crée un quad plein écran (deux triangles)
2. Upload le framebuffer comme texture
3. Écris un vertex shader basique (juste passe les coordonnées)
4. Écris ton fragment shader qui sample la texture et applique l'effet
5. Tu peux chaîner plusieurs shaders avec des FBO (Frame Buffer Objects) pour des effets multiples

Tu peux aussi regarder comment RetroArch gère ses shaders (format Slang / GLSL, le langage de shaders d'OpenGL) pour t'inspirer. Y a des centaines de shaders open source que tu peux adapter.

Roadmap complète

C'est le plan de route de A à Z. Chaque étape te dit quoi coder, quel test lancer, et où relire dans le doc si ça marche pas. Suis cet ordre, c'est celui qui minimise les allers-retours.

Phase 1 : Les fondations (pas d'affichage)

#	Quoi coder	Test / Validation	Si ça marche pas
1.1	Charger une ROM et parser le header	Affiche le titre du jeu et le type de cartouche en console	Relis la section Header (Étape 1). Vérifie les offsets `0x0134` et `0x0147`.
1.2	Le Bus : Read et Write avec routage basique	Écris à `0xC000`, relis : même valeur. Lis `0x0104` : `0xCE` (premier octet du logo)	Relis la carte mémoire (Étape 2). Vérifie les bornes de tes conditions.
1.3	Le CPU : registres, Fetch, et les premiers opcodes (NOP, LD r/n8, LD r/r')	Lance une ROM de test, log chaque instruction. Compare avec un log de référence (Gameboy Doctor).	Relis les patterns d'encodage (Étape 3). Vérifie GetReg / SetReg.
1.4	Le CPU : ALU (ADD, SUB, AND, OR, XOR, CP)	`09-op r,r` et `04-op r,imm` passent	C'est les flags à 99%. Vérifie le half-carry.
1.5	Le CPU : jumps, calls, returns, RST	`07-jr,jp,call, ret,rst` passe	Relis le JR signé et les conditions. Vérifie le timing conditionnel.
1.6	Le CPU : CB prefix (rotations, BIT, SET, RES)	`10-bit ops.gb` passe	Relis la section CB. Vérifie que les rotations CB mettent Z correctement (contrairement à RLCA/RRA).
1.7	Le CPU : toutes les instructions spéciales (DAA, HALT, PUSH/POP, etc.)	Les 11 sous-tests de `cpu_instrs` passent tous	Relis DAA et POP AF (masquer les 4 bits bas). Les logs de référence te montrent où l'état diverge, remonte à partir de là.
1.8	Le port série (SB/SC) pour le debug	Tu vois "Passed" en texte dans la console quand un test Blargg réussit	Quand `0xFF02` reçoit une écriture avec bit 7 = 1, lis `0xFF01` et concatène les caractères.

△ Attention

Si tu bloques sur un test Blargg, la méthode la plus efficace c'est le **log de référence**. Log l'état de ton CPU (PC, SP, AF, BC, DE, HL) à chaque instruction, compare ligne par ligne avec le log de Gameboy Doctor. La première divergence te montre où l'état a dévié. C'est pas forcément l'instruction buggée elle-même (le vrai bug peut venir de plus haut), mais c'est ton meilleur point de départ pour remonter la piste.

Phase 2 : Timer et Interruptions

#	Quoi coder	Test / Validation	Si ça marche pas
2.1	Les interruptions : IME, IF, IE, le dispatch, le EI pending	`02-interrupts.gb` de Blargg (il va fail tant que t'as pas le Timer, c'est normal)	Relis la section Interruptions (Étape 3). Vérifie l'ordre : check interrupts, résoudre EI pending, fetch.
2.2	Le Timer : compteur 16-bit, DIV, TIMA, TMA, TAC, le modèle AND gate	`02-interrupts.gb` passe	Relis le modèle AND gate (Étape 4). Vérifie que DIV est bien les 8 bits hauts du compteur interne.
2.3	Brancher le Timer dans le Bus	Écrire à `0xFF04` remet DIV à 0. Lire `0xFF04` retourne une valeur qui change.	Ajoute le routage `0xFF04-0xFF07` dans ton Bus.

Phase 3 : L'écran (PPU)

C'est là que ça devient visuel. Prends ton temps, c'est le composant le plus complexe.

#	Quoi coder	Test / Validation	Si ça marche pas
3.1	La VRAM, l'OAM, les registres PPU, et le branchement dans le Bus	Rien de visible encore, mais les jeux ne crashent plus en accédant à `0x8000-0x9FFF`	Routage Bus : `0x8000-0x9FFF` (VRAM), `0xFE00-0xFE9F` (OAM), `0xFF40-0xFF4B` (registres).
3.2	Le Tick du PPU : les 4 modes, les transitions, LY qui s'incrémente, l'interruption VBlank	LY s'incrémente de 0 à 153 en boucle. L'interruption VBlank se lève à LY == 144.	Relis l'algorithme du Tick (Étape 5). Vérifie que tu remets le compteur de cycles à 0 à chaque fin de ligne.
3.3	Le rendu du background : décodage 2bpp des tiles, lecture de la tile map, scrolling SCX/SCY	Lance Tetris : tu dois voir le background du menu (même déformé ou avec des mauvaises couleurs, c'est un bon signe)	Relis le décodage des tiles et l'adressage (Étape 5). Astuce : affiche un tile viewer pour isoler le problème.
3.4	Les palettes BGP, OBP0, OBP1	Les couleurs sont correctes (4 nuances de gris/vert)	La palette c'est 4 paires de 2 bits dans un octet. Color 0 = bits 1-0, color 1 = bits 3-2, etc.
3.5	Le rendu des sprites	Tu vois les sprites dans Tetris (les pièces qui tombent)	Relis l'algorithme de rendu des sprites (Étape 5). Vérifie l'OAM scan (max 10) et la transparence (color 0 = invisible).
3.6	La Window	L'interface du jeu s'affiche (score, menus).	Relis le window line counter (Étape 5). Vérifie WX − 7.
3.7	Le DMA OAM (`0xFF46`)	Les sprites marchent dans tous les jeux (certains ne marchent qu'avec DMA)	Copie instantanée de 160 octets, c'est suffisant pour commencer.
3.8	dmg-acid2	L'image rendue correspond pixel pour pixel à l'image de référence du repo.	Compare avec la référence. Chaque défaut visuel correspond à un comportement précis (flips, priorité, 8×16, etc.).

△ Attention

Si ton background est tout noir ou tout blanc mais que le CPU tourne correctement (les tests Blargg passaient), le problème est probablement dans le décodage des tiles. Commence par vérifier que LCDC bit 7 est bien à 1 (LCD allumé) et que tu lis la bonne tile map (LCDC bit 3).

Phase 4 : Jouer

#	Quoi coder	Test / Validation	Si ça marche pas
4.1	Le Joypad (`0xFF00`)	Tu peux naviguer dans les menus de Tetris et jouer une partie	Relis l'Étape 6. Vérifie la logique inversée (0 = pressé) et la sélection du groupe (bits 4-5).
4.2	Le frame pacing (VSync ou sleep)	Le jeu tourne à vitesse normale, pas accéléré	Relis l'Attention box dans "Afficher l'écran". Active VSync ou fais un SDL_Delay de ~16.74 ms.
4.3	MBC1	Zelda, Pokémon, Super Mario Land marchent	Relis MBC1 (Étape 8). Vérifie que le bank 0 est mappé en 1 quand on écrit 0.
4.4	Les sauvegardes (.sav)	Tu peux sauvegarder dans Pokémon et retrouver ta partie au prochain lancement	Écris la RAM externe dans un fichier .sav à la fermeture, relis-la au lancement.

Phase 5 : Le son et le polish

#	Quoi coder	Test / Validation	Si ça marche pas
5.1	L'APU : les canaux square (1 et 2) avec frequency timer et duty cycle	Tu entends les mélodies (même si c'est pas parfait)	Relis "Comment marche un canal square" (Étape 7). Vérifie le reload : `(2048 − freq) × 4`.
5.2	Le Frame Sequencer, Length, Envelope, Sweep	Les notes s'arrêtent au bon moment, le volume fade correctement	Relis la table du Frame Sequencer (Étape 7).
5.3	Le canal Wave et le canal Noise	L'audio est complet. Tu entends les basses (wave) et les percussions (noise).	Wave : vérifie la lecture de la Wave RAM. Noise : vérifie le LFSR.
5.4	Timing précis : `instr_timing.gb` et `mem_timing.gb`	Les deux tests passent	Vérifie que chaque M-cycle fait bien avancer Timer et PPU. Le timing conditionnel des jumps/calls est souvent faux.
5.5	Les edge cases du Timer (falling edges, overflow delay)	`timer` tests de Mooneye passent	Relis les pièges du Timer (Étape 4). C'est le modèle AND gate qui compte.
5.6	Shaders !	L'écran est *stylé*.	Relis l'Étape 9. C'est du bonus, amuse-toi.

Et après ?

Si t'en es là, félicitations, t'as un émulateur Game Boy fonctionnel. Quelques pistes pour aller plus loin :

- **MBC3 et MBC5** : pour supporter plus de jeux. MBC3 a une horloge temps réel en plus.
- **Game Boy Color (CGB)** : double la VRAM, ajoute des palettes couleur, un mode double-speed. C'est un bon projet d'extension.

- **Le PPU pixel-accurate** : au lieu de dessiner scanline par scanline, émule le pixel FIFO du PPU. C'est beaucoup plus complexe mais ça passe tous les tests de timing.
- **Le debugger intégré** : breakpoints, step-by-step, visualisation mémoire, tile viewer, register inspector. C'est un projet en soi et c'est super utile.
- **Le multijoueur** : émuler le câble link pour connecter deux instances.
- **Le rewind** : sauvegarder l'état à chaque frame pour pouvoir "rembobiner" le jeu.

Ressources

- **Pan Docs** (LA référence) : `https://gbdev.io/pandocs/`
- **Table d'opcodes interactive** : `https://izik1.github.io/gbops/`
- **Game Boy CPU Manual** (PDF) : cherche "Game Boy CPU Manual" sur le web
- **ROMs de test Blargg** : `https://github.com/retrio/gb-test-roms`
- **awesome-gbdev** : `https://github.com/gbdev/awesome-gbdev`
- **The Ultimate Game Boy Talk** (conférence vidéo) : cherche sur YouTube, c'est un excellent overview du hardware en 30 minutes
- **Shaders RetroArch** : `https://github.com/libretro/slang-shaders`