

Simulateur de microprocesseur Motorola 6809 : Documentation technique

Introduction

Ce document présente la **documentation technique** du simulateur Motorola 6809 développé dans le cadre de ma 3^e année licence Génie Informatique à la FST Settat. L'objectif du projet est de concevoir un simulateur logiciel capable d'exécuter un programme assembleur 6809 dans un environnement contrôlé, tout en offrant des outils d'observation et de débogage similaires à ceux d'un vrai système (visualisation mémoire, registres, flags, exécution pas à pas, breakpoints...).

Le simulateur est organisé en modules (packages) séparés afin de garder une architecture claire et maintenable. Chaque module couvre une responsabilité précise : lecture des tables CSV des instructions, assemblage du code, décodage des opcodes, exécution CPU, gestion des registres, gestion mémoire (RAM/ROM), gestion du matériel mappé mémoire (PIA), journalisation et remontée d'erreurs, puis enfin l'interface graphique JavaFX qui permet d'utiliser le simulateur facilement. Cette séparation facilite aussi le test et l'évolution du projet, puisque chaque partie peut être améliorée sans impacter directement les autres.

Dans ce document, je détaille le rôle de chaque package, la responsabilité des classes principales, et les méthodes les plus importantes, en expliquant comment les modules collaborent pour réaliser le cycle complet :

programme assembleur → assemblage → chargement ROM → reset → exécution → observation/débogage.

Package com.simulator.moto6809.Memory

1. Rôle général du package

Le package **Memory** représente l'espace mémoire du simulateur Motorola 6809, c'est-à-dire **64 Ko adressables** (de \$0000 à \$FFFF). Il fournit les opérations de base nécessaires au CPU et à l'interface :

- lecture/écriture **octet** et **mot 16 bits** (big-endian, comme sur 6809),
- séparation logique **RAM / ROM** (ROM protégée en écriture),
- chargement de programmes/ROM,
- notifications vers l'UI (ex: vue mémoire) via des **listeners**,
- traces via le logger (utile pendant le debug).

Ce package est donc au cœur du simulateur : toutes les instructions finissent par lire ou écrire en mémoire, directement ou indirectement.

2. Structure interne et composants

2.1 Memory

Classe principale qui contient le **tableau mémoire** :

- private final byte[] memory = new byte[0x10000]; → 64 Ko
- bornes RAM/ROM :
 - par défaut : **RAM \$0000–\$DFFF, ROM \$E000–\$FFFF**
- protection ROM : toute écriture dans [ROMstart..ROMend] est ignorée (avec warning).

En plus, Memory gère :

- un ILogger logger pour les messages (INFO/WARNING/DEBUG),

- une liste de MemoryListener pour informer l'UI lors des écritures et des resets.

2.2 MemoryBus

Classe utilitaire qui fournit une API d'accès mémoire plus "bus CPU".

Elle travaille directement sur un byte[] (souvent celui de Memory) et garde une copie locale de la plage ROM (romStart, romEnd) pour bloquer les écritures ROM.

En pratique, MemoryBus est utile quand on veut :

- une interface d'accès mémoire simple (read/write/readWord/writeWord),
- moins de logique UI/logging,
- synchroniser la protection ROM à partir de Memory.

2.3 MemoryListener

Interface d'événements mémoire :

- onMemoryWrite(int address, int value) : appelée après une écriture d'octet
- onMemoryReset() : appelée après un flush/reset/clear

C'est typiquement consommé par l'interface graphique (table mémoire, coloration, rafraîchissement, etc.).

3. Détails techniques des classes et méthodes

3.1 Classe Memory

3.1.1 Initialisation et configuration

- Memory(ILogger logger)
 - initialise la mémoire avec la configuration par défaut (RAM/ROM) via initializeResetMemory().
- Memory(int ramStart, int ramEnd, int romStart, ILoggger logger)
 - permet une configuration personnalisée des bornes.
- initializeResetMemory()
 - fixe la configuration standard :
 - RAM \$0000–\$DFFF
 - ROM \$E000–\$FFFF
- setMemoryBoundaries(int ramStartAddress, int ramEndAddress)
 - enregistre les bornes RAM avec masquage & 0xFFFF.
 - écrit un log INFO de la plage RAM.
- setROMStartAddress(int romStartAddress)
 - fixe ROMstart, et ROMend = 0xFFFF.
 - écrit un log INFO de la plage ROM.

3.1.2 Protection ROM

- isReadonly(int address)

- retourne true si l'adresse est dans la zone ROM.
- utilisé avant toute écriture.

Cette règle évite qu'un programme "écrase" la ROM pendant l'exécution normale.

3.1.3 Lecture mémoire

- `readMem(int address)`
 - lecture d'un octet : retourne `memory[a] & 0xFF` (valeur non signée 0..255).
- `readMemWord(int address)`
 - lecture d'un **mot 16 bits big-endian** :
 - `high = readMem(a)`
 - `low = readMem(a+1)`
 - `résultat = (high << 8) | low`

3.1.4 Écriture mémoire

- `writeToMem(int address, byte newValue)`
 - si ROM → écriture ignorée + log WARNING.
 - sinon :
 - écrit l'octet
 - appelle `notifyWrite(...)` pour l'UI.
- `writeWordToMem(int address, int newValue)`
 - écrit 2 octets big-endian.
 - si une des deux adresses tombe en ROM → écriture ignorée (word complet rejeté), log WARNING.
 - sinon, écrit via `writeToMem` pour garder les notifications.

3.1.5 Gestion (flush / reset)

- `flushRamOnly()`
 - remet à zéro uniquement la RAM (préserve la ROM).
 - déclenche `notifyReset()`.
 - log DEBUG.
- `flushAll()`
 - remet à zéro toute la mémoire (RAM + ROM).
 - déclenche `notifyReset()`.
 - log DEBUG.
- `resetMemory()`
 - fait `flushAll()`, puis remet la configuration standard RAM/ROM via `initializeResetMemory()`.

- déclenche notifyReset().
- log INFO.

3.1.6 Chargement de programmes / ROM

- loadBytes(int startAddress, byte[] data, boolean allowROMWrite)
 - écrit un tableau d'octets à partir d'une adresse.
 - si allowROMWrite == false et l'adresse est en ROM → skip + WARNING.
 - sinon écrit directement dans memory[addr] et déclenche notifyWrite.

Cette méthode est utilisée typiquement pour :

- charger un programme assemblé en RAM,
- charger une ROM en zone ROM (dans ce cas allowROMWrite = true).

3.1.7 Accès brut au tableau mémoire

- byte[] getMemory()
 - retourne **la référence directe** du tableau interne (pas de clone).

C'est pratique pour la performance (ex: bus CPU), mais il faut faire attention : n'importe quel module qui reçoit cette référence peut modifier la mémoire sans passer par la protection ROM ou les notifications.

3.1.8 Nettoyage ROM avec option “vecteurs”

- clearRom(boolean keepVectors)
 - si keepVectors == false : efface entièrement la ROM.
 - si keepVectors == true : sauvegarde la zone \$FFF2..\$FFFF (vecteurs/fin ROM), efface le reste, puis restaure cette zone.

L'idée ici est de pouvoir nettoyer la ROM sans casser les vecteurs d'interruptions/reset (souvent placés en fin d'espace mémoire sur 6809).

3.1.9 Listener mémoire

- addListener(MemoryListener listener)
- notifyWrite(int address, int value) (privée)
- notifyReset() (privée)

Le listener sert à synchroniser l'affichage mémoire (table hex) avec les écritures faites par le CPU.

3.2 Classe MemoryBus

3.2.1 Construction

- MemoryBus(byte[] initialMemory)
 - exige un tableau de taille exacte 64 Ko.
 - par défaut, la ROM protection est désactivée (romStart > romEnd).
- MemoryBus(Memory mem)

- récupère mem.getMemory() (même référence)
- copie la plage ROM depuis Memory via syncRomRangeFrom(mem).

3.2.2 Synchronisation ROM

- syncRomRangeFrom(Memory mem)
 - met à jour romStart et romEnd du bus.
 - important si on change les bornes ROM dynamiquement.

3.2.3 Accès byte / word

- read(int address) / write(int address, int value)
- readWord(int address) / writeWord(int address, int value)

Même logique big-endian, et écriture ignorée si adresse en ROM.

3.2.4 Effacer RAM seulement

- clearRamOnly()
 - remet à zéro tout sauf la zone ROM (selon romStart..romEnd).

3.2.5 Accès brut

- getRawMemory()
 - retourne la référence directe.

4. Points importants :

Espace mémoire 64 Ko conforme au 6809.

- **Big-endian** sur les mots (readWord/writeWord).
- **Protection ROM** (écritures ignorées) : utile pour simuler une ROM réelle.
- **Listeners mémoire** : indispensable pour une UI réactive (affichage mémoire qui se met à jour).
- **Deux niveaux d'accès** :
 - Memory = plus “riche” (logging, listeners, chargements...)
 - MemoryBus = plus “proche CPU” (lecture/écriture simple et rapide)

Package com.simulator.moto6809.Registers

1. Rôle général du package

Ce package encapsule l'état interne des registres CPU Motorola 6809 et fournit une API centralisée pour :

- **Lire / écrire** les registres 8-bit et 16-bit
- Gérer le registre **CC (Condition Code)** via des **flags**
- **Notifier** l'interface (ou d'autres modules) quand une valeur de registre change

Responsabilités

- Stocker l'état des registres : A, B, D, X, Y, S, U, PC, DP, CC
- Assurer la **cohérence** entre registres liés (notamment **D \leftrightarrow A/B**)
- Appliquer le **masquage** 8/16 bits (0xFF / 0xFFFF)
- Fournir une gestion propre des **flags** et mises à jour **NZ**

Dépendances / Interactions

- Dépend de ILogger et LogLevel (package Logger) pour tracer les accès (lecture/écriture).
- Expose un mécanisme de **listeners** (RegisterListener) typiquement consommé par l'UI (vue registres) et/ou le debugger.

2. Composants (classes / enums / interfaces)

2.1 enum Flag

Rôle : Représente les bits du registre **CC** (Condition Code).

Valeurs :

- E Entire flag
- F Masque FIRQ
- H Half carry
- I Masque IRQ
- N Negative
- Z Zero
- V Overflow
- C Carry

Remarque technique : le mapping bit est défini par flagBit() dans RegisterFunctions.

2.2 enum Register

Rôle : Déclare tous les registres du 6809 manipulables dans le simulateur.

- Registres 8-bit : A, B, DP, CC
- Registres 16-bit : X, Y, S, U, PC
- Registre virtuel : D (concaténation A:B)

Remarque importante : D n'est pas stocké séparément : il est calculé depuis A et B à la lecture, et ré-écrit sur A/B à l'écriture.

2.3 interface RegisterListener

Rôle : Contrat d'observation : permet de réagir aux changements d'un registre/flag.

```
void onRegisterChanged(Register register, int newValue);
```

Usage typique :

- UI : rafraîchir affichage des registres

- Debugger : mettre à jour panneaux, watch, flags, etc.

2.4 class RegisterFunctions

Rôle : Classe centrale du package.

Elle implémente le **stockage**, les **accès** (read/write), la **gestion CC/flags**, et la **notification**.

Données internes (attributs principaux)

- 8-bit : A, B, DP, CC
- 16-bit : X, Y, S, U, PC
- List<RegisterListener> listeners
- ILogger logger

3. API publique (méthodes essentielles)

3.1 Gestion des listeners

- addListener(RegisterListener listener)
 - Ajoute un observateur (si non-null).
- notifyListeners(Register reg, int value) (*privée*)
 - Appelle onRegisterChanged(...) sur tous les listeners.

But technique : découpler CPU/logiciel interne de l'UI : le CPU met à jour les registres, l'UI écoute.

3.2 Lecture des registres

- int getRegister(Register register, boolean notify)
- int getRegister(Register register) (notify = true)

Comportement :

- Masquage automatique :
 - 8-bit : & 0xFF
 - 16-bit : & 0xFFFF
- D est reconstruit : ((A & 0xFF) << 8) | (B & 0xFF)
- Si notify == true, un log DEBUG est produit ("Register X was read").

Objectif : garantir des valeurs propres même si l'état interne est stocké en int.

3.3 Écriture des registres

- void setRegister(Register register, int regValue, boolean notify)
- void setRegister(Register register, int regValue) (notify = true)

Comportement :

1. Détermine si le registre est 16-bit via is16BitRegister()
2. Applique masquage (0xFFFF ou 0xFF)
3. Affecte la valeur :

- si register == D : écrit dans A et B
4. Log DEBUG si demandé : "Register %s updated to \$%04X"

5. Notifications (cohérence)

- notifie toujours le registre écrit
- si A ou B change → notifie D
- si D change → notifie A et B

Point clé : cette logique de notification évite les incohérences d'affichage : si l'UI montre A, B, D, elle reste synchronisée.

4. Gestion du registre CC et des flags

4.1 Mise à jour d'un flag

- setFlag(Flag flag, boolean value)

Comportement :

- Calcule la position du bit (0..7) via flagBit(flag)
- Met ou enlève le bit dans CC
- Notifie ensuite Register.CC

Important : CC est considéré comme un registre standard pour l'observation UI.

4.2 Lecture d'un flag

- boolean getFlag(Flag flag)

Comportement :

- Retourne l'état du bit correspondant dans CC.

4.3 Mapping bit CC

- flagBit(Flag f) (privée)

Mapping actuel :

- E=7, F=6, H=5, I=4, N=3, Z=2, V=1, C=0

5. Mise à jour automatique des flags NZ

5.1 updateNZ(int value, boolean is16)

Met à jour :

- N : bit de signe (0x80 ou 0x8000)
- Z : vrai si résultat == 0 (8 ou 16 bits)

Surcharges :

- updateNZ(int value) (8-bit)
- updateNZ16(int value) (16-bit)

Utilisation : appelée typiquement par les instructions arithmétiques/logiques après calcul d'un résultat.

6. Utilitaires de typage registre

- boolean is16BitRegister(Register reg)
 - true pour {X, Y, S, U, PC, D}
 - false sinon

But : unifier le comportement de masquage (et limiter les bugs de dépassement).

Package com.simulator.moto6809.Resource

1. Rôle du package

Le package **Resource** est responsable du **chargement des ressources CSV** utilisées par le simulateur (tables d'instructions). L'objectif principal est de centraliser la lecture des fichiers CSV (placés dans les ressources du projet) afin d'obtenir, sous forme de structures Java, les informations nécessaires à l'assembleur et/ou au moteur d'exécution : **opcode, cycles, taille** selon le mode d'adressage.

2. Ressources manipulées

Les fichiers CSV sont recherchés dans le chemin de base suivant :

- BASE_PATH = "/com/simulator/moto6809/"

Trois tables sont chargées :

- InstructionOpcode.csv
- InstructionCycle.csv
- InstructionSize.csv

Chaque fichier contient des lignes structurées par **mnémonique** (clé principale) et des colonnes correspondant aux modes d'adressage :

- imm : Immediate
- drt : Direct
- idx : Indexed
- etd : Extended
- inh : Inherent
- rlv : Relative / Long Relative

3. Classes du package

3.1 InstructionCsvLoader

Rôle

InstructionCsvLoader est la classe qui lit les fichiers CSV depuis les ressources du projet et construit une structure Map<String, InstructionCsvRow> indexée par **mnémonique** (ex : "LDA", "ADDA", etc.).

Attributs importants

- BASE_PATH : chemin racine des ressources CSV.

- ILogger logger : utilisé pour tracer le chargement et signaler les erreurs (INFO/WARNING/ERROR).

API publique

- loadOpcodeTable()
Charge InstructionOpcode.csv.
- loadCycleTable()
Charge InstructionCycle.csv.
- loadSizeTable()
Charge InstructionSize.csv.

Toutes ces méthodes déleguent au cœur commun :

- loadTable(String fileName) (méthode privée)

Algorithme de chargement (principe)

1. Construire le chemin complet : BASE_PATH + fileName
2. Ouvrir le fichier via getResourceAsStream(fullPath)
 - si la ressource est introuvable → exception IllegalStateException("Missing CSV resource ...")
3. Lire ligne par ligne avec BufferedReader
4. Nettoyer et filtrer :
 - ignorer lignes vides
 - ignorer commentaires (lignes commençant par # ou //)
 - ignorer l'en-tête si la première ligne ressemble à une ligne "header" (commence par INST_ ou INSTRUCTION)
5. Parser la ligne par split(", ", -1)
 - -1 est important : il conserve les colonnes vides en fin de ligne
6. Contrôles :
 - si parts.length < 7 → ligne considérée mal formée, log WARNING, puis skip
 - si mnémonique vide → log WARNING, puis skip
7. Construire un InstructionCsvRow avec les 7 champs
8. Insérer dans la Map avec table.put(mnemonic, row)
 - si le même mnémonique apparaît deux fois → la dernière occurrence écrase la précédente
9. Log INFO du nombre total d'entrées chargées

Gestion des erreurs

- IOException pendant la lecture : log ERROR avec message.
- ressource manquante : IllegalStateException (erreur bloquante, utile pour détecter un mauvais packaging des ressources).

3.2 InstructionCsvRow

Rôle

Classe immuable (attributs final) qui représente une ligne d'un CSV associée à un mnémonique. Elle stocke les valeurs correspondant aux modes d'adressage.

Resource

Attributs

- mnemonic
- imm, drt, idx, etd, inh, rlv (tous en String)

Resource

Getters

Méthodes simples :

- mnemonic(), imm(), drt(), idx(), etd(), inh(), rlv()

Helpers importants

Ces méthodes permettent de savoir si un mode est disponible pour l'instruction :

- hasImmediate(), hasDirect(), hasIndexed(), hasExtended(), hasInherent(), hasRelative()
Elles retournent true si la colonne correspondante n'est pas vide.

toString()

Fourni pour faciliter le debug (affichage complet de la ligne).

4. Utilisation typique dans le simulateur

Ce package est généralement utilisé comme "source de vérité" pour :

- **Assembler** : valider qu'une instruction supporte un mode d'adressage, et récupérer l'opcode correspondant.
- **Exécution CPU** : récupérer le nombre de cycles, ou la taille d'instruction (si ces infos sont utilisées côté runtime ou debugger).

Debugger / UI : éventuellement, afficher des informations sur les instructions ou aider à la désassemblage.
(Le package Resource reste indépendant : il ne dépend pas de l'assembleur ou du CPU, il fournit seulement les données.

Package com.simulator.moto6809.Debugger

Ce package regroupe les classes nécessaires pour gérer le **debug d'exécution** du simulateur, principalement :

- la création/gestion des **breakpoints** (points d'arrêt par adresse),
- le contrôle de l'état d'exécution (RUN / PAUSE / STOP),
- le mode **STEP** (exécuter exactement une instruction puis s'arrêter).

1. Objectif du package

L'objectif principal est de fournir une couche simple entre :

- le **CPU** (qui exécute des instructions),
- et l'**interface utilisateur** (boutons Run / Pause / Stop / Step + gestion de la liste de breakpoints).

Ce package ne dépend pas directement de l'UI : il expose une logique "métier" que l'UI peut appeler.

2. Classes principales

2.1 Breakpoint

Rôle

Représente un point d'arrêt à une **adresse mémoire précise** (16 bits) avec un état **activé/désactivé**.

Attributs

- address : adresse du breakpoint (forcée sur 16 bits avec & 0xFFFF)
- enabled : indique si le breakpoint est actif (par défaut true)

Méthodes importantes

- getAddress() : retourne l'adresse du breakpoint.
- isEnabled() : retourne l'état du breakpoint.
- setEnabled(boolean enabled) : active ou désactive le breakpoint sans le supprimer.

Remarque technique

Le masquage & 0xFFFF garantit que l'adresse reste toujours dans l'espace adressable du 6809.

2.2 BreakpointManager

Rôle

Gère l'ensemble des breakpoints du simulateur.

Il offre les opérations classiques : ajouter, supprimer, vider, lister et vérifier l'existence d'un breakpoint actif à une adresse.

Structure interne

- Map<Integer, Breakpoint> breakpoints = new HashMap<>();
La clé est l'adresse (16 bits), donc un breakpoint par adresse (pas de doublons).

Méthodes importantes

- add(int address)
Ajoute un breakpoint à l'adresse donnée (écrase l'ancien si déjà présent).
- remove(int address)
Supprime le breakpoint à cette adresse.
- clear()
Supprime tous les breakpoints.
- hasEnabledAt(int address)
Retourne true si un breakpoint existe à cette adresse **et** est activé.
- all()
Retourne Collection<Breakpoint> : utile pour afficher la liste dans l'UI.

Remarque technique

L'utilisation d'une Map rend la vérification de breakpoint très rapide (accès en O(1)), ce qui est important car le CPU peut vérifier à chaque instruction.

2.3 DebugController

Rôle

Contrôle l'état d'exécution global : **STOPPED / RUNNING / PAUSED** et gère le mécanisme du **STEP**.

Enum interne Mode

- STOPPED : CPU arrêté (pas d'exécution)
- RUNNING : CPU en exécution continue
- PAUSED : CPU suspendu (attente)

Attributs

- mode : mode courant (par défaut STOPPED)
- stepRequested : booléen indiquant si un STEP a été demandé
- breakpoints : référence au BreakpointManager utilisé

Méthodes principales

- mode() : retourne le mode actuel.
- run() : passe en RUNNING et annule une demande de step (exécution continue).
- pause() : passe en PAUSED.
- stop() : passe en STOPPED.
- requestStep()
Demande l'exécution **d'une seule instruction** :
 - stepRequested = true
 - mode = RUNNING (le CPU doit exécuter une instruction puis se remettre en pause)
- stepRequested() / clearStepRequest()
Permet au CPU de savoir si un step est demandé et de “consommer” la demande après exécution.
- shouldBreakAt(int pc)
Retourne true si un breakpoint activé existe à l'adresse pc.
C'est l'appel typique effectué par le CPU juste avant (ou juste après) l'exécution d'une instruction.
- breakpoints()
Retourne le manager (utile pour UI : add/remove/list).

3. Fonctionnement global (logique d'intégration CPU)

Dans un cycle d'exécution, le CPU suit généralement cette logique :

1. Lire le PC
2. Vérifier debugController.shouldBreakAt(PC)
 - si true → passer en PAUSED
3. Si mode == RUNNING :

- exécuter une instruction
4. Si stepRequested == true :
- après l'instruction : pause() puis clearStepRequest()

Cette organisation permet :

- des breakpoints “classiques” (arrêt automatique sur adresse),
- un step stable (une instruction exactement),
- une séparation claire entre UI et CPU.

Package com.simulator.moto6809.Execution.CPU

1. Rôle général

Le package **Execution.CPU** contient le **cœur d'exécution** du simulateur Motorola 6809. Il regroupe :

- la classe **CPU** (cycle “fetch → decode → execute”),
- la gestion des **cycles**,
- la logique **interruptions** (IRQ/FIRQ/NMI + SWI),
- un **snapshot** d'état pour l'UI/le debugger,
- un exécuteur qui délègue l'exécution aux **groupes d'instructions** (Load/Store, Arithmetic, Branch, etc.).

2. Énumérations

2.1 CpuMode

Définit l'état global du CPU pendant l'exécution :

- RUNNING : exécution normale
- HALTED : CPU arrêté (stop définitif côté simulateur)
- WAIT_SYNC : état d'attente après instruction **SYNC** (le CPU attend une interruption)
- WAIT_CWAI : état d'attente après **CWAI** (attente interruption, comportement spécial)

2.2 InterruptType

Liste les types d'interruptions reconnues :

- RESET, NMI, IRQ, FIRQ, SWI, SWI2, SWI3

3. Classes principales

3.1 CPU

Rôle

La classe **CPU** orchestre l'exécution réelle : elle lit le PC, décode l'instruction, exécute le comportement 6809 via un exécuteur, met à jour le PC si besoin, comptabilise les cycles, et traite les interruptions + breakpoints.

Dépendances (composants internes)

- MemoryBus bus : accès mémoire (lecture/écriture, mots 16 bits)

- RegisterFunctions regs : registres CPU + flags
- Decoder decoder : décodage des opcodes en DecodedInstruction
- InstructionExecutor executor : exécution via groupes d'instructions
- CycleCounter cycles : compteur global de cycles
- InterruptController interrupts : lignes IRQ/FIRQ/NMI + SWI
- ILogger logger : journalisation (info/debug)
- DebugController debug : gestion run/pause/step + breakpoints
- CpuMode mode : état courant du CPU
- DecodedInstruction lastInstruction : dernière instruction exécutée

Vecteurs d'interruptions

Le CPU utilise les vecteurs standard situés en fin de mémoire (ex : RESET à \$FFFE, IRQ à \$FFF8, etc.).

Méthodes importantes

reset()

- remet le CPU en état RUNNING,
- reset cycles et interruptions,
- charge le **PC** à partir du **vecteur RESET** (bus.readWord(\$FFFE)),
- log INFO du nouveau PC.

halt(), isHalted(), mode()

- halt() force l'arrêt (HALTED) + log WARNING.
- isHalted() renvoie si le CPU est stoppé.

snapshot()

Retourne un CpuStateSnapshot (valeurs registres + flags + cycles + dernière instruction), utilisé côté UI/debugger.

3.1.1 Cycle “une instruction” : stepOnce()

C'est la méthode la plus importante : **exécuter exactement une étape CPU**. La logique est organisée de manière “6809-correct” :

1. Conditions d'arrêt

- si HALTED → retourne 0

2. États d'attente (SYNC/CWAI)

- si WAIT_SYNC ou WAIT_CWAI :
 - si aucune interruption pending → retourne 0 (CPU bloqué)
 - sinon → prend l'interruption (takeInterrupt())

3. Breakpoint

- lit PC

- si debug.shouldBreakAt(PC) :

- log “Breakpoint hit...”
- debug.pause()
- retourne 0

le test est fait **avant** l'exécution de l'instruction à cette adresse.

4. Interruptions asynchrones

- si une interruption est pending (interrupts.next(regs)) :

- takeInterrupt(pending) (avant decode/execute)

5. Decode

- decoder.decodeAt(bus, pc) → produit DecodedInstruction

6. Execute

- executor.execute(instr) → retourne cycles consommés par l'instruction

7. Cycles

- cycles.add(used)

8. Mise à jour du PC

- si l'instruction n'a pas modifié le PC (PC inchangé) :

- PC = instr.nextPc()

9. Gestion SYNC / CWAI

- si mnemonic = SYNC → mode = WAIT_SYNC
- si mnemonic = CWAI → mode = WAIT_CWAI

10. SWI/SWI2/SWI3

- après exécution, si instruction = SWI/SWI2/SWI3 :

- le CPU “request” l'interruption logiciel puis la prend immédiatement (entrée vecteur).

3.1.2 Exécution continue : run(int maxInstructions)

Boucle d'exécution “Run” utilisée par l'UI :

- met debug.run() si debug activé,
- boucle tant que pas HALTED,
- s'arrête si debug passe en PAUSED ou STOPPED,
- exécute stepOnce() à chaque itération,
- si debug.stepRequested() devient vrai : consomme la demande + met pause,
- s'arrête aussi si maxInstructions est atteint (protection anti-boucle infinie).

3.2 InstructionExecutor

Rôle

InstructionExecutor est le **dispatcher** : il reçoit une DecodedInstruction et choisit quel **groupe d'instructions** doit l'exécuter (Load/Store, Arithmetic, Branch...). Il retourne le nombre de cycles consommés.

Dépendances

- RegisterFunctions registers
- MemoryBus memory

Méthode clé : execute(DecodedInstruction instr)

1. Récupère le mnemonic : instr.mnemonic()
2. Teste chaque groupe via supports(m)
3. Appelle Group.execute(instr, registers, memory)
4. Si addressing mode = INDEXED :
 - ajoute une pénalité cycles via IndexedCycleCalculator.computePenalty(instr)
5. Si instruction non trouvée :
 - IllegalStateException("Instruction not implemented...")
6. En cas d'exception runtime :
 - enveloppe dans une exception avec contexte (mnemonic, mode, opcodeBytes).

Groupes d'instructions utilisés (dans cette partie)

- LoadStoreInstructions
- ArithmeticInstructions
- LogicalInstructions
- ShiftRotateInstructions
- UnaryInstructions
- CompareInstructions
- BranchInstructions
- JumpInstructions
- StackInstructions
- RegisterTransferInstructions
- ControlInstructions

Remarque : les détails internes de ces groupes sont dans le package Execution.Instructions (on pourra les documenter séparément, groupe par groupe).

3.3 IndexedCycleCalculator

Rôle

Calcule des **cycles supplémentaires** pour certaines formes du mode **INDEXED** (postbyte).

La méthode principale est : computePenalty(DecodedInstruction instr).

Principe

- si mode ≠ INDEXED → 0
- analyse le postbyte (selon opcodeByteCount())
- distingue :
 - offset 5 bits (bit7=0) → 0
 - sinon (bit7=1) → pénalité selon le nibble bas
- si forme indirecte → +3 cycles (sauf mode 0xF déjà indirect).

3.4 CycleCounter

Rôle

Compteur simple du nombre total de cycles exécutés :

- reset()
- add(int cycles) (sécurise cycles négatifs)
- getTotalCycles()

3.5 CpuStateSnapshot

Rôle

Objet immuable représentant un **instantané** de l'état CPU, destiné au debugger/UI. Il contient :

- valeurs des registres A, B, D, X, Y, S, U, PC, DP, CC
- flags E,F,H,I,N,Z,V,C
- totalCycles
- lastInstruction

Construction

- constructeur privé
- fabrique statique :


```
CpuStateSnapshot.from(RegisterFunctions regs, CycleCounter cycles, DecodedInstruction last)
```

Elle lit les registres (sans notify) et lit les flags depuis regs.

3.6 InterruptController

Rôle

Gère les demandes d'interruptions (lignes latched), et décide quelle interruption doit être prise selon la priorité et les masques flags.

États internes

- interruptions matérielles latched : nmi, irq, firq
- interruptions logicielles : swi, swi2, swi3

Méthodes importantes

- requestNMI() / requestIRQ() / requestFIRQ()
- requestSWI() / requestSWI2() / requestSWI3()
- clearAll()
- next(RegisterFunctions regs) :
 - priorité : NMI > FIRQ > IRQ > SWI/SWI2/SWI3
 - FIRQ bloquée si flag F=1
 - IRQ bloquée si flag I=1
- acknowledge(InterruptType t) : reset la ligne correspondante.

4. Entrée interruption : takeInterrupt(InterruptType type) (dans CPU)

Cette méthode applique l'entrée interruption “réaliste” :

1. interrupts.acknowledge(type)
2. si CPU était en WAIT (SYNC/CWAI) → retour RUNNING
3. empilement + flags via ControllInstructions.enterInterrupt(type, regs, bus)
4. charge PC depuis le vecteur correspondant (readVector(vectorAddress(type)))
5. ajoute les cycles d'entrée
6. log INFO “INTERRUPT ... vector ... PC=...”.

Package com.simulator.moto6809.Execution.Instructions

1. Rôle du package

Le package **Execution.Instructions** regroupe l'implémentation des **familles d'instructions** du Motorola 6809.

Le CPU (via InstructionExecutor) ne code pas chaque opcode directement : il délègue à ces classes “groupes”, qui :

- vérifient si un mnémonique est supporté (supports()),
- exécutent l'instruction (execute()),
- mettent à jour **registres, mémoire et flags CC**,
- retournent le nombre de cycles (généralement instr.cycles()).

Ce découpage rend le projet plus lisible : chaque famille (Load/Store, Branch, Stack, etc.) reste isolée et testable.

2. Conventions communes dans ce package

- **Entrées principales** : DecodedInstruction instr, RegisterFunctions regs, MemoryBus mem.
- **Adressage** : lorsqu'il faut accéder à un opérande mémoire (DIRECT/EXTENDED/INDEXED), les classes utilisent AddressingHelpers.computeEffectiveAddress(instr, regs, mem).
- **Flags** : mises à jour via regs.setFlag(...) et regs.updateNZ(...).

- **Registre cible** : plusieurs familles utilisent Mnemonics.getMnemonicRegister(mnemonic) pour retrouver le registre lié au mnémonique (ex : LDA → A, CMPX → X...).

A. LoadStoreInstructions (chargement / stockage)

Rôle

Implémente les instructions de chargement et stockage :

- LD* : charge une valeur vers un registre
- ST* : écrit un registre en mémoire

Mnemonics supportés

LDA, LDB, LDD, LDX, LDY, LDU, LDS, STA, STB, STD, STX, STY, STU, STS.

Méthodes importantes

- supports(String mnemonic) : test d'appartenance à l'ensemble SUPPORTED.
- execute(instr, regs, mem) :
 - détecte si c'est un store (mnemonic.startsWith("ST"))
 - appelle load(...) ou store(...)
 - retourne instr.cycles().

Détails d'exécution

- load(...) :
 - IMMEDIATE : utilise directement instr.operand()
 - DIRECT/EXTENDED/INDEXED : calcule EA, puis lit mem.read() ou mem.readWord() selon 8/_toggle_toggle_16 bits
 - écrit dans le registre et met à jour **N/Z** via regs.updateNZ(value, is16).
- store(...) :
 - calcule EA, puis écrit mem.write() ou mem.writeWord() selon 8/16 bits.
 - (dans cette version, STORE ne modifie pas les flags).

B) ArithmeticInstructions (addition / soustraction / DAA / MUL)

Rôle

Gère les opérations arithmétiques sur accumulateurs et registre D.

Mnemonics supportés

ADDA, ADDB, ADDD, ADCA, ADCB, SUBA, SUBB, SUBD, SBCA, SBCB, MUL, DAA.

Méthode principale

- execute(instr, regs, mem) : redirige vers add(...), sub(...), mul(...), daa(...).

Détails d'exécution

- add(...) / sub(...)

- récupère $a = \text{regs.getRegister}(reg)$
- récupère l'opérande via `fetchOperand(...)` (IMMEDIATE ou mémoire via EA)
- prend en compte le carry si `withCarry=true`
- écrit le résultat masqué (8 ou 16 bits)
- met à jour les flags via `updateFlagsAdd(...)` / `updateFlagsSub(...)`.
- Flags calculés :
 - C : dépassement (add) / emprunt (sub)
 - V : overflow signé
 - N/Z : via `regs.updateNZ(...)`
 - H : half-carry uniquement en 8 bits (A/B).
- MUL :
 - $D = A * B$, met Z, et met C selon un bit du résultat (implémentation actuelle : test 0x80).
- DAA :
 - ajuste A en BCD selon H et C (correction 0x06 et/ou 0x60)
 - met C, N/Z.

C) Logical Instructions (AND / OR / EOR / BIT / ANDCC / ORCC)

Rôle

Implémente les opérations logiques sur accumulateurs et opérations directes sur CC.

Mnemonics supportés

ANDA, ANDB, ORA, ORB, EORA, EORB, BITA, BITB, ANDCC, ORCC.

Détails d'exécution

- ANDCC / ORCC :
 - opèrent directement sur Register.CC (sans passer par flags individuels) :
 - $CC = CC \& mask$ ou $CC = CC | mask$.
- Instructions sur accumulateurs :
 - operand IMMEDIATE : `instr.operand()` & 0xFF
 - operand mémoire : EA puis `mem.read(ea)`
 - opérations :
 - AND / OR / EOR modifient l'accumulateur
 - BIT : ne modifie pas l'accumulateur, seulement flags sur acc & operand
 - flags :
 - N/Z mis à jour

- V forcé à false pour ces instructions (dans cette implémentation).

D) ShiftRotateInstructions (décalages / rotations)

Rôle

Gère les instructions de décalage et rotation sur un registre (A/B) ou sur une case mémoire.

Mnemonics supportés

ASL/ASLA/ASLB, LSL/LSLA/LSLB, ASR/ASRA/ASRB, LSR/LSRA/LSRB, ROL/ROLA/ROLB, ROR/RORA/RORB.

Fonctionnement

- Détermine si l'opération est sur registre (mode INHERENT) ou mémoire (autre mode).
- Lit la valeur :
 - registre : regs.getRegister(reg) & 0xFF
 - mémoire : EA puis mem.read(ea) & 0xFF
- Calcule carryOut et result selon l'instruction (ASL/LSR/ASR/ROL/ROR).
- Écrit le résultat :
 - registre : regs.setRegister(reg, result)
 - mémoire : mem.write(ea, result)
- Flags :
 - C = carryOut
 - N/Z mis à jour
 - V = N XOR C (comme dans le code actuel).

E) CompareInstructions (CMP*)

Rôle

Compare un registre avec un opérande sans modifier le registre (uniquement flags).

Mnemonics supportés

CMPA, CMPB, CMPD, CMPX, CMPY, CMPPU, CMPS.

Fonctionnement

- Déduit le registre via Mnemonics.getMnemonicRegister(...).
- Calcule result = (regVal - operand) & mask.
- Met à jour :
 - N/Z sur le résultat
 - C comme “borrow” : regVal < operand
 - V overflow signé (formule basée sur XOR).

F) BranchInstructions (branchements courts et longs)

Rôle

Gère les branchements conditionnels et inconditionnels, y compris versions longues (LBxx).

Mnemonics supportés

BRA, BRN, ... , BLE et LBRA, LBRN, ... , LBLE.

Fonctionnement

- Calcule takeBranch selon les flags C/Z/N/V (ex : BHI = !C && !Z, BGE = N==V, etc.).
- Si branchement pris : PC = instr.relativeTargetAddress()
- Sinon : PC = instr.nextPc()
- Retourne instr.cycles().

G) JumpInstructions (JMP/JSR/BSR)

Rôle

Implémente sauts et appels de sous-programmes.

Mnemonics supportés

JMP, JSR, BSR, LBSR.

Détails

- JMP : PC = EA
- JSR :
 - empile instr.nextPc() sur la pile S (pushWordS)
 - puis PC = EA
- BSR/LBSR :
 - empile instr.nextPc()
 - PC = instr.relativeTargetAddress()

H) StackInstructions (PSH/PUL sur S et U)

Rôle

Implémente les instructions de pile :

- PSHS / PULS : pile système (S)
- PSHU / PULU : pile utilisateur (U)

Mnemonics supportés

PSHS, PULS, PSHU, PULU.

Principe

- Utilise un masque 8 bits mask = instr.operand() & 0xFF :
 - bits haut : PC, U/S, Y, X

- bits bas : DP, B, A, CC
- PSH* : pousse dans un ordre défini via StackHelpers.pushByte* / pushWord*
- PUL* : tire (pull) dans l'ordre inverse via StackHelpers.pullByte* / pullWord*
- Les getRegister(..., false) / setRegister(..., false) sont utilisés pour éviter des notifications UI inutiles pendant la séquence.

I) RegisterTransferInstructions (TFR / EXG / ABX)

Rôle

Gère transfert et échange de registres, plus une instruction de calcul simple.

Mnemonics supportés

TFR, EXG, ABX.

Détails

- ABX :
 - $X = X + B$ (B sur 8 bits)
 - **n'affecte pas les flags**
 - cycles fixes : 3.
- TFR :
 - lit un postbyte : source = nibble haut, destination = nibble bas
 - vérifie même taille (8 \leftrightarrow 8 ou 16 \leftrightarrow 16) via ensureSameSize
 - copie la valeur src \rightarrow dst
 - cycles : 6.
- EXG :
 - même décodage que TFR
 - échange les valeurs
 - cycles : 8.

J) ControlInstructions (NOP, SYNC, CWAI, SWI, RTS, RTI + entrée interruption)

Rôle

Gère les instructions de contrôle du processeur (retours, interruptions logicielles, modes d'attente).

Mnemonics supportés

NOP, SYNC, CWAI, SWI, SWI2, SWI3, RTS, RTI.

Exécution (execute)

- NOP : aucun effet.
- RTS : PC = pullWordS().
- RTI :

- dépend du flag E :
 - si E=1 : restauration “entire state”
 - sinon : restauration “minimal state”.
- SWI/SWI2/SWI3 :
 - forcent E=1
 - push “entire state”
 - SWI masque aussi I et F
 - le chargement du PC via vecteur est fait ensuite par le CPU.
- CWAI :
 - CC = CC & imm
 - force E=1
 - le CPU doit passer en état WAIT (géré dans CPU).
- SYNC :
 - ne change pas les registres, mais déclenche l'état WAIT côté CPU.

Entrée interruption (enterInterrupt)

La méthode enterInterrupt(InterruptType type, regs, mem) est appelée par le CPU **avant** de charger PC depuis le vecteur :

- IRQ/NMI : push entire state, met I=1, cycles ~19
- FIRQ : push minimal state avec E=0, met F=1, cycles ~10
- SWI/SWI2/SWI3 : déjà traitées dans execute() (ici retourne 0)

K) UnaryInstructions

Rôle

UnaryInstructions implémente les instructions qui opèrent sur :

- soit un registre A ou B (formes INHERENT : INCA, DECB...),
- soit une adresse mémoire (formes DIRECT/EXTENDED/INDEXED : INC <ea>, etc.).

Instructions couvertes :

- INC, DEC, NEG, COM, CLR, TST et leurs variantes A/B.

5.1 Détection registre vs mémoire

- Si instr.addressingMode() == INHERENT :
 - registre = Mnemonics.getMnemonicRegister(instr.mnemonic())
 - valeur = regs.getRegister(reg) & 0xFF
- Sinon :
 - EA = AddressingHelpers.computeEffectiveAddress(instr, regs, mem)

- valeur = mem.read(ea) & 0xFF

5.2 Comportement par instruction + flags

INC

- résultat = value + 1
- flags :
 - N/Z via regs.updateNZ(result)
 - V = (result == 0x80)

DEC

- résultat = value - 1
- flags :
 - N/Z
 - V = (result == 0x7F)

NEG

- résultat = -value
- flags :
 - N/Z
 - C = (value != 0)
 - V = (result == 0x80)

COM

- résultat = ~value
- flags :
 - N/Z
 - C = true
 - V = false

CLR

- résultat = 0
- flags :
 - N = false
 - Z = true
 - V = false
 - C = false

TST

- ne modifie pas l'opérande (pas de write-back)
- flags :
 - N/Z sur la valeur testée
 - V = false

5.3 Write-back (registre ou mémoire)

Une méthode write(...) applique le résultat :

- registre : regs.setRegister(reg, result)
- mémoire : mem.write(ea, result)
Puis retourne instr.cycles().

L) AddressingHelpers

Rôle

AddressingHelpers centralise le calcul de **l'adresse effective (EA)** pour les modes d'adressage qui pointent vers la mémoire.

C'est une classe très importante, parce que toutes les instructions de type STA, LDA, INC <ea>, etc., ont besoin d'un calcul EA identique pour être correct.

Méthode principale

- computeEffectiveAddress(DecodedInstruction instr, RegisterFunctions regs, MemoryBus bus)

Cette méthode :

- lit instr.addressingMode(),
- calcule l'EA selon le mode :

Mode	Comportement
DIRECT	EA = (DP << 8) + opérande 8-bit
EXTENDED	EA = opérande 16-bit
RELATIVE	EA = instr.relativeTargetAddress()
INDEXED	EA calculée à partir du postbyte + registre index + offsets

IMMEDIATE / INHERENT **pas d'EA** → exception (appel interdit)

Important : le choix d'émettre une exception pour IMMEDIATE/INHERENT évite les bugs silencieux (EA incorrecte).

1.1 Détail du calcul INDEXED (postbyte)

L'indexed du 6809 est le plus riche. L'algorithme suit les règles du postbyte :

Étape 1 : récupérer le postbyte

- Le postbyte est dans instr.bytes() à l'index opcodeByteCount().
- Si bytes == null ou longueur insuffisante → exception (decode invalide).

Étape 2 : choisir le registre index (bits 6-5)

- $00 \rightarrow X, 01 \rightarrow Y, 10 \rightarrow U, 11 \rightarrow S$
Base = valeur du registre index (16 bits).

Étape 3 : deux cas

Cas A — offset signé 5 bits (bit7 = 0)

- off5 = post & 0x1F
- extension de signe si bit4=1
- EA = base + off5

Cas B — indexed “complexe” (bit7 = 1)

- indirectFlag = bit4 (0x10)
- mode = nibble bas (0x0..0xF)
- plusieurs sous-formes : ,R+, ,--R, 8-bit offset, 16-bit offset, PC-relative, etc.

Exemples importants :

- ,R+ et ,R++ : EA = base puis **le registre est incrémenté** (post-increment).
- ,-R et ,--R : le registre est décrémenté avant utilisation (pre-decrement).
- 8-bit offset : lit bytes[postIndex+1] comme signed.
- 16-bit offset : lit 2 octets comme signed short.
- D,R : offset = D (signé 16 bits).

Cas spécial 0xF (extended indirect)

- [nn] : lit une adresse 16 bits dans les bytes, puis lit un pointeur en mémoire (2 octets) pour obtenir l'EA finale.
- le code empêche de ré-appliquer indirectFlag pour 0xF (car déjà indirect par définition).

Indirect général

Si indirectFlag == true, l'EA calculée devient un pointeur :

- EA = mem[EA] + mem[EA+1] (lecture 16 bits en mémoire).

M) Mnemonics

Rôle

Mnemonics fournit :

1. une liste centralisée des mnemonics supportés (ensemble mnemonics)
2. des ensembles spécialisés pour identifier rapidement les instructions qui agissent sur **A** ou **B**
3. une méthode pour associer un mnemonic à un registre (getMnemonicRegister)
4. un validateur (isMnemonic) utilisé côté assembleur/decoder ou UI.

Méthodes importantes

- mnemonics() : retourne l'ensemble global

- AccaMnemonics() / AccbMnemonics() : ensembles mnemonics qui utilisent A ou B
- isMnemonic(String mnemonic) : vérifie null/blank puis membership set
- getMnemonicRegister(String mnemonic) : retourne le registre ciblé le plus probable

Exemples de mapping :

- LDA → A, LDB → B
- LDD/ADDD/CMPD → D
- LDX/CMPX → X
- PSHS/PULS/CMPS → S
- ANDCC/ORCC → CC

Execution.Instructions_classes_...

❖ Cette classe permet d'écrire des exécutions génériques dans les groupes (ex : UnaryInstructions, CompareInstructions, etc.) sans recopier la logique “mnemonic → registre”.

N) EAResult

Rôle

EAResult est un record qui transporte :

- ea : adresse effective calculée
- extraCycles : cycles supplémentaires associés au calcul

Cette structure est utile quand un helper doit renvoyer l'EA et en même temps indiquer une pénalité de cycles. Dans la version actuelle, on voit surtout que c'est préparé pour une gestion plus fine des cycles.

O) StackHelpers

Rôle

StackHelpers centralise toutes les opérations sur les piles du 6809 :

- pile système **S**
- pile utilisateur **U**

Le 6809 utilise des piles **descendantes** : un push décrémente le pointeur, un pull l'incrémente.

Méthodes sur pile **S**

- pushByteS(...), pushWordS(...)
- pullByteS(...), pullWordS(...)

Détails importants :

- pushWordS pousse **high puis low** (big-endian).
- pullWordS recompose correctement en lisant **low puis high** (inverse logique du push).

Sauvegarde/restauration d'état CPU (interruptions)

- pushEntireStateS(...) : pousse **PC,U,Y,X,DP,B,A,CC**
- pushMinimalStateS(...) : pousse seulement **PC,CC** (cas FIRQ)
- pullEntireStateS(...) : restaure dans l'ordre inverse **CC,A,B,DP,X,Y,U,PC**
- pullMinimalStateS(...) : restaure **CC,PC**

Ces méthodes sont utilisées par ControllInstructions et le CPU lors de RTI et des entrées interruption.

Méthodes sur pile U

Même logique que S :

- pushByteU, pushWordU, pullByteU, pullWordU
Mais elles manipulent Register.U.

Package com.simulator.moto6809.Decoder

Le package **Decoder** est responsable de transformer une séquence d'octets en mémoire (opcode + opérande) en un objet exploitable par le CPU : une **instruction décodée** (DecodedInstruction). Il s'appuie sur une base de données d'instructions (InstructionSet) construite à partir des fichiers CSV (opcode/cycles/size).

1. AddressingMode

Rôle

Enum qui décrit **comment** une instruction accède à son opérande. C'est utilisé dans plusieurs parties du projet : InstructionSet, Decoder, DecodedInstruction, et les helpers d'exécution.

Valeurs

- IMMEDIATE : opérande dans le flux d'instruction (#)
- DIRECT : adresse sur 8 bits combinée avec DP
- INDEXED : adressage indexé basé sur postbyte (X/Y/U/S + offsets)
- EXTENDED : adresse 16 bits complète
- INHERENT : pas d'opérande (ex : RTS, NOP)
- RELATIVE : branches (offset 8 ou 16 bits)

Méthodes utilitaires

- hasOperand() : vrai sauf INHERENT
- isRelative() : vrai uniquement pour RELATIVE
- accessesMemory() : faux pour IMMEDIATE et INHERENT, vrai sinon

2. DecodedInstruction

Rôle

Objet immuable qui représente **une instruction 6809 déjà décodée**. Il contient tout ce dont le CPU et l'exécuteur ont besoin : identité, mode d'adressage, taille, cycles, opérande, et bytes originaux (utile pour debug).

Champs principaux

- pc : PC au début de l'instruction (snapshot)

- opcode : opcode complet (8 bits ou préfixé 0x10xx / 0x11xx)
- mnemonic : ex "LDA"
- addressingMode
- cycles : cycles de base (hors pénalités indexed)
- size : taille totale en octets
- operand : opérande brut (big-endian), sans le postbyte indexed
- effectiveAddress : champ prévu pour stocker une EA (peut rester null)
- bytes : tableau complet des octets de l'instruction (obligatoire ici)

Méthodes importantes

- isPrefixedOpcode() : vrai si opcode page2/page3 (0x10xx ou 0x11xx)
- opcodeByteCount() : 1 ou 2
- operandByteCount() : size - opcodeByteCount()
- signedRelativeOffset() : convertit operand en offset signé (8 ou 16 bits)
- relativeTargetAddress() : calcule la cible (base = PC+size)
- nextPc() : PC après instruction (PC+size)
- toString() : format debug lisible (PC, mnemonic, mode, opcode, size, cycles, operand, ea, bytes)

Builder

Le Builder force des contrôles de cohérence :

- bytes non null et bytes.length == size
- mnemonic non vide
- addressingMode non null
- size > 0

3. InstructionDefinition

Rôle

Représente la définition d'une instruction au niveau "base de données".

Une même instruction (ex : LDA) peut exister dans plusieurs modes d'adressage, donc InstructionDefinition garde des maps séparées :

- opcode par mode,
- cycles par mode,
- size par mode.

Structure

- mnemonic
- opcodeMap : Map<AddressingMode, Integer>

- cycleMap : Map<AddressingMode, Integer>
- sizeMap : Map<AddressingMode, Integer>

Méthodes importantes

- addOpcode(mode, opcode)
- addCycles(mode, cycles)
- addSize(mode, size)
- supports(mode)
- getOpcode(mode) / getCycles(mode) / getSize(mode)
→ lèvent une IllegalStateException si la valeur n'existe pas (instruction non supportée dans ce mode).

4. InstructionSet

Rôle

C'est la **base centrale** de toutes les instructions 6809. Elle est construite au démarrage à partir des CSV via InstructionCsvLoader.

Elle fournit deux accès :

- recherche par mnemonic (LDA),
- recherche par opcode (ex : \$86 ou \$108E).

Champs

- byMnemonic : Map<String, InstructionDefinition>
- byOpcode : Map<Integer, InstructionDefinition>
- ILogger logger

Méthodes publiques

- getByMnemonic(String mnemonic) : case-insensitive (toUpperCase)
- getByOpcode(int opcode)
- containsMnemonic(String mnemonic)

Construction load()

1. Charger trois tables :
 - opcodeTable, cycleTable, sizeTable
2. Pour chaque mnemonic :
 - récupérer les 3 lignes CSV correspondantes
 - si cycle/size manquante → log WARNING et skip
3. Créer InstructionDefinition def
4. Enregistrer pour chaque mode : register(def, mode, opcodeStr, cyclesStr, sizeStr)
5. Si aucune cellule valide (ligne invalide / HD6309 only) → skip

6. Ajouter dans byMnemonic, et chaque opcode dans byOpcode
7. Log INFO : nombre de mnemonics / opcodes chargés

Méthode register(...)

- parseOpcodeHex(opcodeStr) accepte :
 - "86", "108E", "10 8E", "0x108E"
 - ignore ce qui n'est pas hex (ex : (HD6309 only))
- parseCycles(...) : extrait le début numérique ("5+" → 5, "6(7)" → 6)
- parseInt(...) : tolérant ("2+" → 2, "" → 0)
- normalizeSizeForPrefix(...) : point important :
 - certains opcodes préfixés (page 2/3) ont une taille CSV qui semble compter l'opcode comme 1 byte
 - la méthode corrige en ajoutant 1 si nécessaire
 - cas particulier : immediate 16-bit "3" devient "4" pour opcode préfixé.

5. Decoder

Rôle

Decoder lit les octets à partir d'un PC donné, gère les opcodes préfixés, détermine le mode d'adressage réel, calcule la taille finale (surtout pour indexed), puis construit un DecodedInstruction.

Dépendances

- InstructionSet instructionSet
- ILogger logger
- MemoryBus bus (argument de decodeAt)

Méthode principale : decodeAt(MemoryBus bus, int pc)

Étapes :

1. b0 = bus.read(pc)
2. Si b0 == 0x10 ou 0x11 :
 - lire b1 = bus.read(pc+1)
 - opcode = (b0<<8) | b1
 - opcodeBytes = 2
 - sinon :
 - opcode = b0
 - opcodeBytes = 1
3. Chercher définition : def = instructionSet.getByOpcode(opcode)
 - si null → log ERROR + exception "Unknown/illegal opcode"
4. Résoudre le mode :

- resolveModeFromDefinition(def, opcode)
- garde : vérifie def.supports(mode) et que l'opcode correspond
- 5. Récupérer size & cycles depuis la table
- 6. Si mode == INDEXED :
 - lire postbyte juste après opcode
 - calculer extra = indexedExtraBytes(postbyte)
 - size += extra
- 7. Lire tous les bytes (taille finale) dans un tableau bytes[]
- 8. Parser operand via parseOperand(mode, bytes, opcodeBytes)
 - important : en INDEXED, bytes[opcodeBytes] est le postbyte, donc l'opérande commence après
- 9. Construire et retourner DecodedInstruction via builder

Méthodes internes importantes

- resolveModeFromDefinition(def, opcode) :
 - parcourt tous les AddressingMode et retourne celui dont def.getOpcode(mode) correspond à l'opcode.
- parseOperand(mode, bytes, opcodeBytes) :
 - INHERENT → operand = 0
 - INDEXED → start = opcodeBytes + 1
 - concatène le reste des bytes en big-endian
- indexedExtraBytes(postbyte) :
 - si bit7 = 0 → 0 extra (offset 5 bits)
 - sinon dépend du nibble bas :
 - 0x8 / 0xC → +1
 - 0x9 / 0xD / 0xF → +2
 - sinon → 0

Package com.simulator.moto6809Assembler

1) Rôle du package

Le package **Assembler** transforme un programme assembleur 6809 (texte) en **octets machine** prêts à être chargés en mémoire (ROM).

Il gère :

- l'encodage des instructions (opcode + opérandes),
- les **modes d'adressage** (immédiat, direct, étendu, indexé, relatif...),
- les **labels** (symboles) et les références avant/après définition,
- les **directives** (ORG, RMB, FCB, FDB, EQU/SET, END),

- la génération d'un **listing** utile pour l'interface "Programme".

Ce package s'appuie fortement sur InstructionSet / InstructionDefinition (tables CSV) pour garantir la **bonne taille** d'instruction, surtout pour les opcodes préfixés et les immédiats 8/16 bits.

2) Vue d'ensemble du fonctionnement

Approche globale

Le simulateur utilise une stratégie **2 passes** (dans AssemblerProgram) :

- **Pass 1** : calculer les adresses, enregistrer les labels, et générer des **placeholders** (octets "formes" correctes) en respectant la taille réelle des instructions.
- **Pass 2** : remplacer les placeholders par les valeurs finales (offsets de branches, valeurs d'immédiats, adresses de labels, PC-relative, etc.), puis produire :
 - une liste d'octets linéaire,
 - une image mémoire sparse (adresse → byte),
 - un listing structuré pour l'UI.

3) Classes du package

3.1 Assembler

Rôle

Assembler assemble **une seule ligne** (instruction) en octets, en gérant :

- INHERENT,
- IMMEDIATE,
- INDEXED (y compris indirect [...]),
- DIRECT / EXTENDED,
- et les BRANCHES (RELATIVE) sous forme de placeholder (offset patché plus tard).

Dépendances

- OpcodeSelector opcodeSelector : choisit l'opcode selon (mnemonic, addressingMode) via la table CSV opcode.
- InstructionSet instructionSet : utilisé pour calculer le nombre exact d'octets d'opérande (requiredOperandBytes), donc la taille réelle de l'instruction.

Méthode principale : assembleLine(String line)

Algorithme (résumé) :

1. Nettoyer la ligne, séparer mnemonic et operandText.
2. Si mnémonique de branchemen (BRA, BEQ, ... ou LB..) :
 - opcode RELATIVE
 - produire opcode + **0x00 placeholders** pour l'offset (1 ou 2 octets selon la taille).
3. Si operand vide → INHERENT : produire uniquement l'opcode.

4. Si operand commence par # (et pas de virgule) → IMMEDIATE :
 - encoder la valeur si numérique, sinon placeholders (label).
5. Sinon → tester INDEXED :
 - accepte formes avec virgule , et/ou indirect [...],
 - encode le postbyte + extra bytes (via IndexedEncoder),
 - cas spécial [\$nnnn] : postbyte 0x9F + adresse 16-bit.
6. Sinon → DIRECT / EXTENDED :
 - si label → placeholders (choix EXTENDED par défaut si possible),
 - si numérique : DIRECT si \leq 0xFF sinon EXTENDED.

Helpers importants

- requiredOperandBytes(mnemonic, mode, opcode) : calcule le nombre d'octets d'opérande à partir de InstructionDefinition.size(mode) et du nombre d'octets opcode (1 ou 2 si préfixé).
- emitOpcode(...) : écrit correctement 0x10/0x11 + opcode low byte si préfixé.
- parseNumber(...) : supporte \$hex, %bin, @oct, décimal, avec signe.

3.2 AssemblerProgram (driver 2-pass)

Rôle

C'est la classe "haut niveau" qui assemble un **fichier complet** (liste de lignes). Elle gère :

- labels (avec : ou forme "LABEL INSTRUCTION ..."),
- directives,
- calcul PC,
- patching final,
- génération d'un listing.

Structures internes

- SymbolTable symbols : table des labels/constants (case-insensitive).
- LineEntry : structure interne qui mémorise pour chaque ligne :
 - type (INSTRUCTION, ORG, RMB, FCB/FDB, EQU/SET, END...),
 - label/keyword/opérande,
 - bytes placeholders,
 - pcBefore/pcAfter,
 - items de données (FCB/FDB).

API publique

- assemble(lines, origin) : retourne une liste d'octets **linéaire** (concaténation des bytes générés).
- assembleToMemory(lines, origin) : retourne un Map<address, byte> (respecte ORG/RMB).

- `assembleListing(lines, origin)` : retourne une liste de ListingRow pour l'UI (adresse avant/après + bytes + source).
- `assembleToRom(Memory memory, lines, defaultOrigin, writeResetVectorIfMissing)` :
 - assemble,
 - vérifie que les bytes générés sont **dans la zone ROM**,
 - charge dans la ROM (`memory.loadBytes(..., allowROMWrite=true)`),
 - écrit le **RESET vector** `$FFFE/$FFFF` si absent, en pointant vers l'entry point (première adresse émise).

Directives supportées

- ORG addr : change PC.
- RMB n : réserve n bytes (avance PC, n'émet pas d'octets).
- FCB a,b,c : émet des bytes.
- FDB w1,w2 : émet des mots 16-bit big-endian.
- EQU/SET : définit une constante.
 - EQU interdit redéfinition,
 - SET autorise (comportement “variable”).
- END : stop l'assemblage.

[Pass 1 : production de placeholders corrects](#)

Le point important est `buildPlaceholdersPass1(...)` :

- Sans InstructionSet, les lignes avec labels risquent d'avoir une taille incorrecte → le code bloque avec une exception pour éviter une génération fausse.
- Pour les branches : taille connue → opcode + offset placeholder.
- Pour l'indexed :
 - gère [LABEL] (extended indirect) en forçant la forme 0x9F + 16-bit placeholder,
 - gère LABEL,PC en forçant la forme 16-bit si forward label (évite “too far” après).

[Pass 2 : patchIfNeeded\(...\)](#)

Corrige les placeholders selon les symboles :

- **RELATIVE** : calcule offset = target - nextPc puis vérifie la plage (8-bit ou 16-bit).
- **IMMEDIATE label** : remplace les octets de l'immédiat par la valeur du symbole.
- **INDEXED PC-relative label** : remplace les extra bytes (après postbyte) par l'offset signé.
- **[LABEL]** : patch l'adresse 16-bit après le postbyte 0x9F.
- **DIRECT/EXTENDED label** : patch la/les bytes finales d'adresse.

3.3 IndexedEncoder + IndexedOperand

Rôle

Ces classes encodent **uniquement** la partie indexed :

- le **postbyte**,
- et les “extra bytes” (offset 8/16 bits ou adresse).

IndexedOperand

Conteneur :

- postbyte (0..255),
- extra (null, 8-bit ou 16-bit).

Encodages gérés (principaux)

- ,R et [,R] : zero offset
- ,R+, ,R++, ,-R, ,--R
- A,R / B,R / D,R
- n,R :
 - -16..+15 → forme 5-bit (si non indirect)
 - sinon 8-bit (post 0x08) ou 16-bit (post 0x09)
- n,PC ou [n,PC] : postbytes 0x8C/0x8D (+0x10 si indirect)
- [\$nnnn] : 0x9F + 16-bit address

3.4 OpcodeSelector

Rôle

Sélectionne l'opcode correspondant à un couple (**mnemonic**, **AddressingMode**) en lisant une ligne InstructionCsvRow (table opcode).

Méthode clé : select(mnemonic, mode)

- normalise la clé en uppercase,
- lit la colonne associée au mode (imm/drt/idx/etd/inh/r1v),
- parse l'opcode (2 à 4 digits hex, accepte “10 8E”, “0x108E”, etc.),
- si pas supporté → exception “Addressing mode ... not supported”.

3.5 SymbolTable

Rôle

Table des symboles **case-insensitive** (labels et constantes), avec stockage en 16 bits.

Méthodes principales

- define(label, value) : ajoute/écrase.
- definelfAbsent(label, value) : utile pour règles EQU.
- contains(label)
- resolve(label) : retourne la valeur ou erreur “Undefined label”.

- `snapshot()` : copie non modifiable (utile debug/UI).
- `clear()` : réinitialise.

4) Points importants à mettre dans la documentation globale

- L'assembleur est **fiable pour les labels** grâce au mécanisme 2-pass et au fait qu'il force parfois des formes "stables" (ex : PC-relative 16-bit pour forward label).
- `assembleToMemory` respecte ORG/RMB (image sparse), alors que `assemble` renvoie juste une concaténation des bytes.
- `assembleToRom` sécurise l'écriture : le programme ne doit pas émettre hors de la plage ROM, et le vecteur RESET peut être ajouté automatiquement si manquant.

Package com.simulator.moto6809.Bootstrap

1) Rôle du package

Le package **Bootstrap** contient la classe qui sert de **point d'initialisation** du simulateur.

Son rôle est de créer et relier toutes les briques principales (mémoire, bus, registres, instruction set, CPU, debug, assembleur), puis d'exposer une API simple pour l'interface utilisateur (charger un programme, reset, run/step, breakpoints, clear RAM/ROM...).

2) Classe principale : Bootstrap

2.1 Objectif

Bootstrap joue le rôle de "classe centrale" (service locator).

L'UI ou un contrôleur principal n'a pas besoin de construire manuellement chaque composant : Bootstrap s'en charge une seule fois, et fournit des getters.

2.2 Composition (objets internes)

Dans le constructeur, Bootstrap instancie dans cet ordre :

- `ILogger logger`
- `Memory memory`
- `MemoryBus bus`
- `InstructionSet instructionSet`
- `RegisterFunctions registers`
- `BreakpointManager breakpointManager`
- `DebugController debugController` (initialisé en STOPPED)
- `CPU cpu` (lié à bus + registers + instructionSet + debug)
- `AssemblerProgram assemblerProgram` (construit via `buildAssemblerProgram`)

❖ L'ordre est important :

- le CPU dépend du bus, registres, instruction set et debug,
- AssemblerProgram dépend de Assembler + InstructionSet.

2.3 API publique exposée à l'UI

Accès aux composants

- `memory()` : accès direct à l'objet mémoire (RAM/ROM).
- `bus()` : accès au bus mémoire (utilisé par CPU).
- `instructionSet()` : base des instructions (chargée depuis CSV).
- `registers()` : registres CPU + flags CC.
- `debug()` : accès au DebugController (pause/run/step + breakpoints).
- `cpu()` : accès au CPU lui-même.
- `assemblerProgram()` : accès aux fonctions d'assemblage + listing.

Ces getters permettent à l'UI d'afficher l'état interne (registres, mémoire, cycles, etc.) et d'appeler les actions (run/step/reset).

Chargement d'un programme en ROM

- `loadAsmToRom(List<String> asmLines, Integer defaultOrigin, boolean writeResetVectorIfMissing)`

Fonctionnement :

1. Calcule origin :
 - si `defaultOrigin != null` : utilise `defaultOrigin & 0xFFFF`
 - sinon : utilise `memory.getROMstart()` (adresse ROM de départ)
2. Synchronise les bornes ROM avec le bus : `bus.syncRomRangeFrom(memory)`
3. Appelle `assemblerProgram.assembleToRom(...)`

Résultat :

- retourne l'**entry point** (adresse de départ du programme chargé).

Ce choix est pratique pour l'UI : après assemblage, on peut positionner PC/affichage sur l'entry point.

Reset et exécution CPU

- `resetCpu()`
 - resynchronise la plage ROM dans le bus,
 - appelle `cpu.reset()` (PC chargé depuis vecteur RESET).
- `run(int maxInstructions)`
 - lance l'exécution continue en appelant `cpu.run(maxInstructions)`.
- `stepOnce()`
 - exécute une seule instruction (utile pour le mode STEP).

Gestion des breakpoints

- `addBreakpoint(int address)`
- `removeBreakpoint(int address)`
- `clearBreakpoints()`

Les adresses sont masquées & 0xFFFF pour rester dans l'espace 6809.

Nettoyage mémoire

- clearRam() : appelle bus.clearRamOnly() (préserve ROM).
- clearRom(boolean keepVectors) :
 - memory.clearRom(keepVectors)
 - resynchronise bus avec memory.

Listing (pour mapping PC ↔ ligne source)

- assembleListing(List<String> asmLines, int origin)

Retourne une liste de AssemblerProgram.ListingRow.

Cette méthode est importante pour l'UI : elle permet d'associer une adresse PC aux lignes affichées dans l'éditeur (highlight de la ligne en cours d'exécution).

2.4 Construction de l'assembleur : buildAssemblerProgram(...)

Méthode interne qui met en place la chaîne :

1. InstructionCsvLoader loader
2. opcodeTable = loader.loadOpcodeTable()
3. OpcodeSelector selector = new OpcodeSelector(opcodeTable)
4. Assembler assembler = new Assembler(selector, instructionSet)
5. return new AssemblerProgram(assembler, instructionSet)

Ici, InstructionSet est déjà instancié dans Bootstrap, donc on utilise la même source de vérité entre **assembleur** et **décodage**.

3) Logger fallback interne : StdoutLogger

Si Bootstrap reçoit logger == null, il utilise un logger minimal interne (StdoutLogger) :

- ERROR vers System.err
- le reste vers System.out
- pas de fichier log
- clear() vide

Ce fallback est utile pour des tests en console.

4) Chargement depuis un fichier .asm

Deux méthodes utilitaires :

- loadAsmFileToRom(Path asmFile, Integer defaultOrigin, boolean writeResetVectorIfMissing)
 - lit toutes les lignes via Files.readAllLines
 - appelle loadAsmToRom(...)
- loadAsmFileToRom(Path asmFile) : version simplifiée (origin null, writeResetVectorIfMissing=true)

Conclusion (pour ta documentation globale)

Bootstrap est la classe qui relie tout le simulateur :

- il centralise l'initialisation,
- expose une API minimale et claire pour l'UI,
- garantit la cohérence entre ROM, bus, assembleur, CPU et debug,
- fournit le listing nécessaire au lien **PC ↔ source**.

Package com.simulator.moto6809.UI

Ce package contient toute la partie **interface graphique JavaFX** du simulateur : l'éditeur assembleur, l'affichage CPU (registres flags/PC/cycles), la console de logs, les vues mémoire (RAM/ROM), la table “Program listing”, et la gestion des breakpoints côté UI. L'UI ne fait pas l'exécution elle-même : elle pilote la couche centrale (Bootstrap) via CentralController, et elle récupère l'état CPU sous forme de snapshots.

1) Architecture UI : séparation “contrôle” / “vue”

1.1 CentralController (couche logique UI)

CentralController est la classe qui sert de **pont** entre :

- les actions utilisateur (cliquer sur Run, Step, Assemble/Load...),
- et les services du simulateur (Bootstrap, CPU, DebugController, mémoire, assembleur).

Il expose des **properties JavaFX** (PC, cycles, registres, flags) directement bindables dans l'interface, et il gère aussi la console.

UI

1.2 MainView (couche vue)

MainView construit la fenêtre :

- une barre d'outils (actions principales),
- un éditeur de code assembleur,
- des tabs RAM/ROM/Program,
- un panneau à droite pour l'état CPU,
- une console en bas.

MainView ne fait pas d'exécution CPU : il appelle uniquement le CentralController.

2) CentralController — gestion CPU + synchronisation UI

2.1 Gestion du thread CPU

L'exécution CPU est faite dans un **thread dédié** via :

```
ExecutorService cpuExec = Executors.newSingleThreadExecutor(...)
```

Objectif : éviter de bloquer le thread JavaFX (sinon l'interface freeze).

Toutes les actions lourdes passent par submitCpuTask(Runnable r) qui :

- empêche les doubles clics (guard busyFlag),

- met à jour une property busy utilisable pour désactiver les boutons,
- exécute la tâche sur cpu-thread.

2.2 Console UI

La console est un ObservableList<String> (bindé dans MainView).

Les logs viennent d'un logger interne UiLogger : chaque message est envoyé sur le thread UI via Platform.runLater(...).

2.3 Snapshots CPU (communication thread-safe)

Le CPU publie des snapshots (CpuStateSnapshot) grâce à un listener :

- hookCpuListener() → boot.cpu().setListener(snap -> pendingSnapshot.set(snap))
- côté UI : pumpUi() lit pendingSnapshot.getAndSet(null) puis met à jour les properties.

Comme ça, l'UI récupère l'état CPU **sans accéder directement** au CPU depuis le thread JavaFX.

2.4 Properties exposées à l'UI

CentralController garde des propriétés pour :

- PC, cycles, dernière instruction,
- registres A, B, D, X, Y, S, U, DP, CC,
- flags E F H I N Z V C (boolean).

Ces propriétés sont mises à jour dans refreshFromSnapshot(...). L'intérêt est que toute l'UI peut se binder dessus (labels, textfields, indicateurs de flags...).

3) Chargement programme : assembleur + mapping PC ↔ lignes source

3.1 assembleAndLoad(String asmText, Integer defaultOrigin)

C'est la fonction principale "Assemble / Load". Elle :

1. stop debug, clear breakpoints, reset état "programLoaded"
2. découpe le texte en lignes
3. choisit l'origin (par défaut ROM start)
4. appelle boot.loadAsmToRom(...) (écrit en ROM + reset vector si besoin)
5. récupère le listing : boot.assembleListing(lines, origin)
6. construit :
 - pcToLine et lineToPc (pour highlight et breakpoints),
 - loadedRomMask (masque des octets ROM réellement générés).
7. remplit programRows (affichage dans l'onglet Program)
8. réinstalle les breakpoints qui étaient cochés dans l'éditeur
9. reset CPU (PC depuis RESET vector).

3.2 Table "Program listing" (ProgramRow)

Chaque ligne affichée contient :

- lineIndex (ligne source),
- pcBefore,
- bytesHex,
- source (texte d'origine).

Ça sert à montrer clairement “ligne assembleur → octets machine → adresse”.

4) Breakpoints côté UI : lignes → adresses

4.1 Stockage breakpoints

- breakpointLines : set de lignes (ce que l'utilisateur clique dans l'éditeur)
- installedBreakpointAddrs : set d'adresses réellement installées dans le CPU

4.2 toggleBreakpointLine(int lineIndex)

Quand on clique sur une ligne :

- si déjà présent → remove
- sinon → add
Puis applyEditorBreakpointsToCpu(copy) est appelé dans le thread CPU.

4.3 Résolution “ligne → adresse”

resolveBreakpointAddressForLine(lineIndex) :

- si la ligne n'émet pas d'octets (label / ORG / vide), la fonction cherche la **prochaine ligne exécutable** en avançant.
Ça évite le cas où un breakpoint est posé sur une ligne qui n'a pas de PC réel.

5) Règle “fin de programme” via loadedRomMask

Le controller construit un tableau boolean[65536] qui marque tous les octets ROM réellement chargés.

Après step() ou run(), la méthode stopIfPcOutsideLoadedRom() :

- vérifie si PC est dans la zone ROM,
- si le byte à PC n'est pas dans le mask → le programme est considéré terminé,
- puis fait boot.debug().pause() et boot.cpu().halt().

C'est une solution pratique pour éviter une exécution infinie quand le PC “sort” de la zone du programme.

6) MainView — construction de la fenêtre

6.1 Layout global

- ToolBar en haut : Assemble/Load, Run, Pause, Step, Reset CPU, Clear RAM, Clear Console, champs Origin/Max.
- SplitPane en 3 colonnes :
 1. éditeur (CodeArea),

2. tabs (RAM/ROM/Program),
 3. panneau CPU state.
- ListView console en bas.

6.2 Éditeur assembleur (RichTextFX)

CodeArea + VirtualizedScrollPane :

- style monospace,
- line numbers via LineNumberFactory,
- texte par défaut chargé par defaultProgram().

6.3 Désactivation intelligente des boutons

Les boutons Run/Step sont désactivés si :

- aucun programme n'est chargé (programLoadedProperty().not()),
- ou si le simulateur est occupé (busyProperty()).

7) Onglets centraux : RAM / ROM / Program

7.1 MemoryGridPane

C'est une table mémoire 16 colonnes (00..0F) + une colonne base "R/C".

Deux usages :

- **RAM** : éditable (poke activé)
- **ROM** : lecture seule (poke null)

Fonctions importantes :

- navigation (Prev/Next), refresh,
- champ "Base" (adresse) et "Rows" (nombre de lignes affichées),
- mise en forme responsive (largeur colonnes + hauteur lignes + font).

Édition hex en RAM (HexCell)

- autorise 1 ou 2 digits hex,
- commit sur Enter ou perte de focus,
- refuse l'écriture hors range RAM.

Highlight mémoire

- En RAM : highlight si byte ≠ 0 (visualisation rapide des zones modifiées)
- En ROM : highlight si c'est un octet du programme (via controller.isRomProgramByte(addr)).

7.2 ProgramPane

Table simple qui affiche la liste programRows (Line, PC, Bytes, Source).

8) Panneau CPU State (droite)

8.1 Affichage PC + cycles

- PC affiché en \$%04X (read-only)
- cycles affichés en label large.

8.2 Cartes registres éditables

Chaque registre est un TextField avec :

- un formatter qui impose \$ + digits hex,
- un commit automatique quand le champ perd le focus,
- appel controller.setRegister(reg, value).

8.3 Flags (E F H I N Z V C)

Affichés avec un point circulaire vert/gris selon le boolean property.

8.4 “Last instruction” enrichi

La dernière instruction est une string issue de DecodedInstruction.toString().

MainView parse cette string pour extraire :

- mnemonic, opcode, mode,
- operand, bytes,
- EA, size, cycles,
et les affiche sous forme de “chips”.

9) Highlight ligne PC + breakpoints dans l’éditeur

9.1 Gutter breakpoints

Dans la marge gauche, chaque ligne affiche :

- le numéro de ligne,
- un cercle rouge cliquable (breakpoint on/off).

9.2 Styles de lignes

applyLineStyle(lineIndex) applique des classes CSS :

- bp-line si la ligne a un breakpoint,
- pc-line si la ligne correspond au PC courant.

10) Boucle de rafraîchissement UI : startUiPump()

Un AnimationTimer tourne en continu :

- ~60fps : controller.pumpUi() + highlight ligne courante
- toutes les ~250ms : refresh RAM/ROM (si pas en édition)

Ça donne une UI fluide sans surcharger le rendu mémoire.

11) Classe Application (entry point JavaFX)

- crée MainView,
- applique BreakPoint.css,

- fixe taille minimale fenêtre,
- démarre view.startUiPump(),
- stop propre : controller.shutdown() à la fermeture.

Package com.simulator.moto6809.Hardware

1. Objectif du package

Le package **Hardware** représente la partie “matérielle” du simulateur, c'est-à-dire les **périphériques mappés mémoire** (memory-mapped I/O).

L'idée est la suivante : au lieu que le CPU écrive uniquement dans la RAM/ROM, certaines adresses peuvent correspondre à un composant externe (ex : PIA). Dans ce cas, une lecture/écriture à ces adresses déclenche une opération sur ce périphérique (changer l'état d'un port, lire un registre de contrôle, etc.).

2. Architecture générale

Le package est organisé autour de trois éléments :

1. **Device** : interface générique pour tout périphérique mappé mémoire.
2. **DeviceListener** : interface d'observation (UI/Debugger) pour être notifié quand l'état d'un périphérique change.
3. **MC6821PIA** : première implémentation d'un périphérique réel : le **PIA Motorola MC6821**, utilisé pour la gestion des ports d'E/S.

3. Détails des composants

3.1 Interface Device

Rôle

Définit un contrat minimal pour intégrer n'importe quel périphérique dans le système.

Méthodes

- boolean handles(int address)
Retourne true si le périphérique gère l'adresse mémoire donnée (plage d'adresses du device).
- int read(int address)
Lecture d'un registre/port du périphérique (retourne une valeur 8 bits sous forme int).
- void write(int address, int value)
Écriture sur un registre/port du périphérique (valeur 8 bits).

Remarque

Cette interface permet d'ajouter facilement d'autres périphériques plus tard (ACIA, timers, etc.) sans changer la logique du CPU.

3.2 Interface DeviceListener

Rôle

Permet à l'interface graphique ou au debugger de détecter les modifications d'état d'un périphérique.

Méthode

- void onDeviceStateChanged(Device device)
Appelée à chaque fois qu'un device change d'état suite à un write() ou un reset().

3.3 Classe MC6821PIA

3.3.1 Rôle

MC6821PIA simule le **Peripheral Interface Adapter 6821**, un composant classique utilisé avec les CPU Motorola. Il fournit deux ports d'entrées/sorties (**Port A** et **Port B**) ainsi que des registres de contrôle.

Dans ce simulateur, il est implémenté comme un **device mappé mémoire de 4 octets**.

3.3.2 Mapping mémoire

Le PIA occupe 4 adresses consécutives à partir de baseAddress :

- base + 0 : Port A / DDRA
- base + 1 : CRA (Control Register A)
- base + 2 : Port B / DDRB
- base + 3 : CRB (Control Register B)

Important : dans cette version, la sélection “Port vs DDR” est simplifiée (le commentaire indique *Port A / DDRA* et *Port B / DDRB*, mais l'implémentation actuelle garde ddra/ddrb comme registres séparés et read/write sur offset 0/2 utilisent uniquement readPortA/writePortA et readPortB/writePortB).

Hardware

3.3.3 Attributs internes

Registres 8 bits simulés par des int masqués sur 0xFF :

- portA, portB : valeur actuelle des ports
- ddra, ddrb : Data Direction Registers (1 = sortie, 0 = entrée)
- cra, crb : Control Registers A/B
- listeners : liste des DeviceListener (pour UI/Debugger)

3.3.4 Méthodes principales

Constructeur

- MC6821PIA(int baseAddress)
Initialise l'adresse de base (masquage 16 bits) puis appelle reset().

Hardware

Vérification d'adresse

- handles(int address)
Retourne true si address est dans [baseAddress, baseAddress+3].

Lecture

- read(int address)
Calcule l'offset et retourne :

- offset 0 : readPortA()
- offset 1 : cra
- offset 2 : readPortB()
- offset 3 : crb

Écriture

- write(int address, int value)

Écrit selon l'offset :

- offset 0 : writePortA(value)
 - offset 1 : cra = value
 - offset 2 : writePortB(value)
 - offset 3 : crb = value
- puis appelle notifyListeners().

3.3.5 Gestion des ports (version simplifiée)

- readPortA() retourne portA & ddra
- readPortB() retourne portB & ddrb

Cela signifie que seuls les bits configurés en sortie (DDR=1) sont visibles en lecture.

C'est une version minimale correcte pour une première intégration, mais elle ne modélise pas complètement les entrées externes.

- writePortA(int value) : met à jour uniquement les bits où ddra=1
- writePortB(int value) : met à jour uniquement les bits où ddrb=1

3.3.6 Reset

- reset() met tous les registres à 0x00 et notifie les listeners.

Hardware

3.3.7 Support UI / Debugger

Le PIA expose :

- addListener(DeviceListener listener)
- getters pour tous les registres
- setters manuels (ex : setPortA, setDDRA ...) pour permettre à l'UI ou au debugger de modifier l'état sans passer par le bus mémoire.

4. Scénario typique d'utilisation

1. Le CPU exécute une instruction qui écrit en mémoire (ex : STA \$8000).
2. Le système vérifie si \$8000 correspond à un périphérique (device.handles(address)).
3. Si oui → appel device.write(address, value).
4. Le PIA met à jour ses registres internes et notifie DeviceListener.

5. L'UI (fenêtre PIA) se met à jour automatiquement.

5. Limites actuelles (à signaler dans ton rapport)

- Le modèle présent est indiqué comme “squelette fonctionnel” :
 - logique IRQ (IRQ A/B, FIRQ) non implémentée,
 - handshaking CA1/CA2/CB1/CB2 non implémenté,
 - la sélection Port vs DDR via CRA/CRB (comportement réel du 6821) n'est pas entièrement reproduite, mais ddra/ddrb existent déjà pour une extension future.

Conclusion : Logique de fonctionnement

Le simulateur suit une logique proche du fonctionnement réel d'un système basé sur le Motorola 6809. Le point de départ est un programme assembleur saisi dans l'éditeur. Lors de l'étape **Assemble/Load**, le module **Assembler** réalise un assemblage en deux passes : la première construit le mapping des labels et la taille exacte des instructions, puis la deuxième applique les patchs (branches, labels, PC-relative) et produit les octets machine. Ces octets sont ensuite chargés en **ROM** dans le module **Memory**, avec protection contre l'écriture normale, et un vecteur RESET peut être ajouté automatiquement pour garantir un démarrage correct.

Après chargement, le **CPU** est réinitialisé : il lit le vecteur RESET en mémoire pour placer le **PC** à l'adresse de démarrage. Ensuite, l'exécution se fait par étapes : à chaque instruction, le CPU lit l'opcode en mémoire, le module **Decoder** construit un objet **DecodedInstruction** (*mnemonic, addressing mode, taille, cycles, bytes, operand...*), puis l'**InstructionExecutor** délègue l'exécution au groupe correspondant dans **Execution.Instructions** (Load/Store, Arithmetic, Branch, Stack...). Les opérations modifient les registres via **RegisterFunctions**, mettent à jour les flags du registre CC, et accèdent à la mémoire via **MemoryBus**. Les cycles sont comptabilisés, et les interruptions (IRQ/FIRQ/NMI + SWI) sont gérées en respectant les masques et les vecteurs.

Le débogage est assuré par le package **Debugger** : les breakpoints sont installés par adresse, vérifiés avant l'exécution de l'instruction située au PC, et l'exécution peut être contrôlée avec **Run/Pause/Step**. L'interface JavaFX pilote tout cela via **CentralController**, qui exécute le CPU sur un thread séparé, récupère des snapshots (**CpuStateSnapshot**) pour mettre à jour l'affichage, et met à disposition des vues RAM/ROM et du listing du programme. La console affiche les informations importantes grâce au package **Logger**, et les erreurs/problèmes sont remontés via les structures standardisées (**Response**, **Problem**, **CompilationProblems**).

En résumé, le logiciel fonctionne comme une chaîne cohérente : **ressources CSV → assembleur → mémoire ROM → CPU (decode/execute) → mise à jour registres/mémoire → UI + debug**. Cette organisation modulaire garantit une bonne lisibilité du code, une séparation claire des responsabilités, et une utilisation simple lors de la démonstration (assemblage, chargement, run/step, breakpoints, observation en temps réel).

