UNIVERSITÉ HASSAN 1er

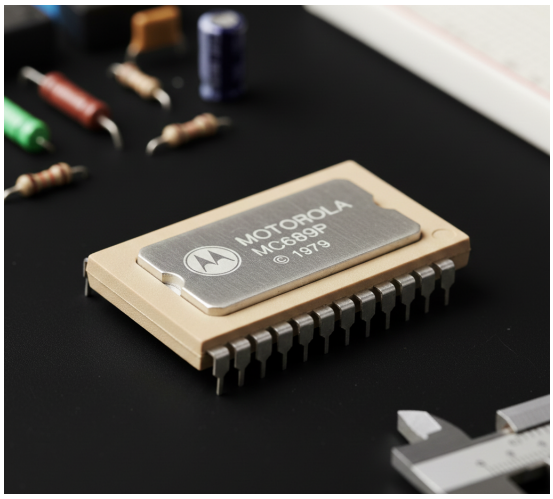
FACULTÉ DES SCIENCES ET TECHNIQUES DE SETTAT

Département : Génie Informatique (LST-GI)

Année universitaire : 2025 – 2026

Simulateur de Microprocesseur Motorola 6809

Documentation technique



Encadrant :
Prof. Benalla Hicham

Réalisé par :
Chritt Fadila
Barbay Hafsa

---

21 décembre 2025

TABLE DES MATIÈRES

1 Architecture globale du simulateur

1 Organisation générale du projet

Le simulateur du microprocesseur **Motorola 6809** est conçu selon une **architecture modulaire**, permettant une séparation claire des responsabilités et facilitant la compréhension, la maintenance et l'évolution du système.

Le projet est structuré en **quatre packages principaux** :

— `cpu`

— `app`

— `device`

— `gui`

Chaque package correspond à une partie précise de l'architecture d'un système à microprocesseur réel et joue un rôle bien défini dans le fonctionnement global du simulateur.

2 Séparation en packages

2.1 Package `cpu`

Le package `cpu` représente le **cœur du simulateur**. Il assure l'implémentation complète du microprocesseur Motorola 6809 et regroupe toutes les fonctionnalités liées à son fonctionnement interne, notamment :

— la gestion des registres internes,

— le cycle d'exécution des instructions (fetch, decode, execute),

— l'implémentation du jeu d'instructions,

— la gestion de la pile et des interruptions.

Ce package joue un rôle central, car toutes les opérations de calcul et de contrôle du simulateur dépendent directement de l'état du CPU.

2.2 Package `app`

Le package `app` assure la **logique applicative du simulateur**. Il agit comme un intermédiaire entre le cœur du processeur et les différentes interfaces utilisateur.

Il contient principalement :

— le chargement des programmes binaires en mémoire,

— le lancement et la configuration du simulateur,

— les classes de coordination entre le CPU et les interfaces (console ou graphique).

Ce package joue donc le rôle de **coordinateur** du système.

2.3 Package `device`

Le package `device` regroupe les **périphériques simulés** du système, tels que le clavier ou d'autres dispositifs d'entrée/sortie.

Les périphériques sont généralement **mappés en mémoire** et peuvent interagir avec le CPU à travers :

— des accès mémoire spécifiques,

— des interruptions simulées.

Ce package permet de reproduire le comportement des composants matériels externes dans un environnement logiciel.

2.4 Package `gui`

Le package `gui` correspond à l'**interface graphique du simulateur**, développée à l'aide de la bibliothèque **Swing**.

Il permet à l'utilisateur de :

— visualiser les registres et la mémoire,

— contrôler l'exécution du programme (step, run, reset),

— observer l'état interne du processeur et les mécanismes de débogage.

Ce package n'intervient pas directement dans l'exécution des instructions ; il se limite à l'observation et à l'affichage de l'état du système.

3 Architecture interne du simulateur

Cette section décrit le fonctionnement interne du simulateur Motorola 6809, en mettant l'accent sur les mécanismes fondamentaux qui permettent de reproduire le comportement réel du microprocesseur. L'architecture interne repose principalement sur la simulation du CPU, de la mémoire et des périphériques, ainsi que sur la gestion du cycle d'exécution des instructions.

3.1 Composants internes principaux

Le simulateur est basé sur plusieurs composants internes essentiels :

— le processeur central (CPU),

— la mémoire principale,
— les périphériques simulés,
— le système de contrôle de l'exécution.

3.2 Cycle d'exécution des instructions

Le cycle d'exécution des instructions (Fetch–Decode–Execute), introduit par John von Neumann, décrit le fonctionnement fondamental d'un processeur. Ce cycle commence au démarrage de la machine et se répète continuellement jusqu'à son arrêt.

Il est composé de trois phases principales :

Phase Fetch (chargement)

Lors de la phase de chargement, le processeur lit l'instruction située à l'adresse indiquée par le compteur ordinal (PC) depuis la mémoire, puis incrémente ce registre afin de préparer l'instruction suivante.

Cette phase est implémentée dans le simulateur par la méthode suivante :

```
public int fetchByte() {
 int b = readByte(reg.PC);
 reg.PC = (reg.PC + 1) & 0xFFFF;
 return b;
}
```

La lecture effective de la mémoire est assurée par la méthode `readByte()`, qui gère également les accès aux périphériques mémoire-mappés.

—

Phase Decode (décodage)

Durant la phase de décodage, l'opcode de l'instruction chargée est analysé afin d'identifier l'opération à exécuter et les ressources nécessaires (registres, mémoire ou unité arithmétique).

Dans le simulateur, le décodage repose sur une table de correspondance entre les opcodes et leurs implémentations :

```
public Instruction get(int opcode) {
 return instructions.get(opcode);
}
```

Les instructions étendues du Motorola 6809, identifiées par les préfixes `0x10` et `0x11`, sont prises en charge directement dans le cycle d'exécution :

```
if (inst == null && opcode == 0x10)
inst = iset.get(0x100 | fetchByte());
else if (inst == null && opcode == 0x11)
inst = iset.get(0x1100 | fetchByte());
```

—

Phase Execute (exécution)

Lors de la phase d'exécution, l'instruction décodée est exécutée. Cette opération peut correspondre à un calcul, un accès mémoire, un saut ou un branchement conditionnel.

L'exécution est déclenchée par l'appel suivant :

```
inst.execute(this);
```

L'exemple suivant illustre l'exécution d'une instruction réelle du simulateur (chargement immédiat dans l'accumulateur A) :

```
add(0x86, "LDA #", cpu -> {
 cpu.reg.A = cpu.fetchByte() & 0xFF;
 setNZ8(cpu, cpu.reg.A);
});
```

Cette instruction met à jour le registre A ainsi que les indicateurs d'état (flags) du processeur.

—

À la fin de l'exécution, le compteur ordinal est mis à jour et le cycle recommence. Des interruptions peuvent temporairement suspendre ce cycle afin de traiter des événements externes. Dans les processeurs modernes, des techniques comme le pipeline permettent d'améliorer les performances en exécutant plusieurs instructions en parallèle.

3.3 Architecture des registres du Motorola 6809

Le microprocesseur Motorola 6809 dispose d'une architecture de registres avancée pour un processeur 8 bits. Ces registres jouent un rôle fondamental dans l'exécution des instructions, le stockage temporaire des données, la gestion de la mémoire et le contrôle de l'état du processeur.

Registres 8 bits

Les registres 8 bits sont principalement utilisés pour le traitement des données et la gestion de l'état du processeur :

- **Accumulateur A** : registre 8 bits utilisé pour les opérations arithmétiques et logiques.
- **Accumulateur B** : registre 8 bits également dédié aux opérations arithmétiques et logiques. La majorité des instructions applicables à l'accumulateur A peuvent aussi être utilisées avec l'accumulateur B.
- **Accumulateur D** : accumulateur 16 bits résultant de la combinaison des registres A et B. Le registre A contient l'octet de poids fort, tandis que le registre B contient l'octet de poids faible.
- **Registre DP (Direct Page)** : contient l'octet de poids fort de la page mémoire utilisée pour l'adressage direct. Après une réinitialisation du processeur, ce registre est initialisé à la valeur `00h`.
- **Registre CC (Condition Code)** : contient les indicateurs d'état (flags) reflétant le résultat des opérations exécutées par le processeur.

Certaines instructions sont spécifiques à un accumulateur précis, comme `DAA` qui s'applique uniquement à l'accumulateur A, ou `ABX` qui utilise l'accumulateur B.

Registres 16 bits

Les registres 16 bits sont destinés à la gestion des adresses mémoire, de l'indexation et des piles :

- **Registres X et Y** : registres d'index utilisés principalement pour les modes d'adressage indexé.
- **Registre PC (Program Counter)** : registre 16 bits pointant vers l'adresse de la prochaine instruction à exécuter. Il peut également être utilisé avec certains modes d'adressage indexé.
- **Registre S (System Stack Pointer)** : pointeur de pile système. Cette pile est utilisée pour stocker l'état du processeur lors des appels de sous-programmes et des interruptions.
- **Registre U (User Stack Pointer)** : pointeur de pile utilisateur, généralement employé pour le passage de paramètres entre sous-programmes et le stockage temporaire de données.

Lorsqu'une donnée est empilée dans la pile système ou utilisateur, le pointeur de pile correspondant est décrémenté de deux octets avant l'écriture en mémoire. Les registres **S** et **U** peuvent également être utilisés comme registres d'index.

Implémentation logicielle des registres

Dans le simulateur, l'ensemble des registres du Motorola 6809 est implémenté dans une classe dédiée, permettant de représenter fidèlement l'état interne du processeur.

```
public class Registers {
 public int PC;   // 16-bit Program Counter
 public int A;    // 8-bit Accumulator A
 public int B;    // 8-bit Accumulator B
 public int X;    // 16-bit Index Register X
 public int Y;    // 16-bit Index Register Y
 public int SP;   // 16-bit Hardware Stack Pointer (S)
 public int U;    // 16-bit User Stack Pointer (U)
 public int CC;   // 8-bit Condition Code Register
 public int DP;   // 8-bit Direct Page Register

 // Accumulateur 16 bits D (A:B)
 public int D() {
  return ((A << 8) | B) & 0xFFFF;
 }

 public void setD(int val) {
  A = (val >> 8) & 0xFF;
  B = val & 0xFF;
 }
}
```

Cette implémentation permet de manipuler facilement les registres du processeur et de reproduire le comportement réel du Motorola 6809, notamment la gestion de l'accumulateur 16 bits **D**, formé par la combinaison des registres **A** et **B**.

4 Système de mémoire et adressage

Cette section décrit l'organisation de la mémoire du simulateur Motorola 6809 ainsi que les mécanismes d'adressage utilisés pour accéder aux données et aux instructions. L'objectif est de reproduire fidèlement le fonctionnement du microprocesseur réel tout en conservant une implémentation claire et pédagogique.

4.1 Organisation de la mémoire

Le Motorola 6809 dispose d'un espace d'adressage de **64 Ko**, correspondant à des adresses codées sur 16 bits, allant de `0000h` à `FFFFh`.

Dans le simulateur, la mémoire principale est implémentée dans la classe `Memory.java` sous la forme d'un tableau de 65 536 octets :

```
public class Memory {
 private final byte[] mem = new byte[65536];  // 64 Ko
 public byte readByte(int address) {
  return mem[address & 0xFFFF];  // Masquage 16 bits
 }
}
```

4.2 Modes d'adressage implémentés

Le Motorola 6809 propose un ensemble riche de modes d'adressage. Dans le cadre de ce simulateur, un sous-ensemble représentatif de ces modes a été implémenté afin de couvrir les cas d'utilisation les plus courants, tout en conservant une complexité adaptée à un simulateur pédagogique.

Les modes d'adressage pris en charge dans cette version du simulateur sont définis comme suit :

- **Adressage inhérent (Inherent)** : la valeur de données ou l'adresse des données est implicitement associée à l'instruction. Ce mode est utilisé par des instructions ne nécessitant aucun opérande explicite, telles que `NOP`, `SYNC` ou `RTS`.
- **Adressage immédiat (Immediate)** : une donnée sur 8 bits ou 16 bits est directement fournie dans l'instruction. L'opérande est donc lu immédiatement après l'opcode, sans accès mémoire supplémentaire.
- **Adressage par registre (Register)** : l'instruction fait référence directement au contenu d'un registre ou d'une paire de registres. Ce mode est notamment utilisé pour les opérations impliquant les accumulateurs **A**, **B** ou le registre combiné **D**.
- **Adressage direct (Direct)** : un opérande sur un octet contenu dans l'instruction spécifie une adresse mémoire située dans la page directe. Cette page correspond à une zone de 256 octets définie par le registre **DP**. Après un reset, le registre **DP** pointe par défaut vers la page mémoire `0000h`.
- **Adressage étendu (Extended)** : un opérande sur deux octets contenu dans l'instruction spécifie directement l'adresse mémoire complète sur 16 bits où les données sont stockées.

— **Adressage indexé (Indexed)** : l'adresse effective est obtenue en ajoutant un décalage à la valeur contenue dans un registre d'index ou de pointeur (**X**, **Y**, **U** ou **SP**). Dans le simulateur, ce mode est implémenté sous forme d'un adressage indexé simple avec décalage constant.

— **Adressage relatif (Relative)** : un décalage sur un ou deux octets est ajouté au compteur ordinal (**PC**). Le résultat correspond à l'adresse de destination vers laquelle le processeur transfère le contrôle, principalement pour les instructions de branchement conditionnel ou inconditionnel.

Les modes d'adressage indirects (étendu indirect, relatif indirect, indexé indirect et leurs variantes avancées telles que l'auto-incrémentation ou l'auto-décrémentation) sont présents dans l'architecture complète du Motorola 6809, mais ne sont pas implémentés dans cette version du simulateur.

4.3 Mémoire mappée pour les périphériques

Dans les systèmes réels, certains périphériques (clavier, écran, disques) sont accessibles via des adresses mémoire spécifiques. Notre simulateur reproduit ce mécanisme avec un clavier virtuel.

Fonctionnement

1. L'utilisateur appuie sur une touche dans l'interface
2. La classe `Keyboard6809` stocke le caractère
3. Le caractère est écrit à l'adresse `0xFF00`
4. Le statut "touche disponible" est écrit à `0xFF01`
5. Le programme assembleur peut lire ces adresses

5 Jeu d'instructions implémenté

Le simulateur implémente un sous-ensemble représentatif du jeu d'instructions du microprocesseur Motorola 6809. Ce sous-ensemble couvre les principales catégories fonctionnelles du processeur et permet l'exécution de programmes assembleur simples à intermédiaires, tout en conservant une architecture logicielle claire et extensible.

Les instructions sont centralisées dans la classe `InstructionSet`, située dans le package `cpu`. Chaque instruction est identifiée par son opcode et associée à une fonction Java implémentant son comportement.

5.1 Catégories d'instructions

Conformément à la classification présentée dans le cours sur le Motorola 6809, les instructions implémentées sont regroupées en plusieurs catégories fonctionnelles.

Instructions arithmétiques

Les instructions arithmétiques réalisent des opérations mathématiques sur les registres du processeur. Elles mettent à jour les indicateurs d'état du registre **CC** (Negative, Zero, Carry, Overflow, Half-Carry).

Exemples d'instructions implémentées :

— `ADDA`, `ADDB`, `ADCA` : addition avec ou sans retenue
— `SUBA`, `SBCA` : soustraction avec ou sans retenue
— `CMPA`, `CMPB`, `CMPX`, `CMPY` : comparaison
— `MUL` : multiplication des accumulateurs A et B vers l'accumulateur D
— `DAA` : ajustement décimal après addition

Instructions logiques

Les instructions logiques effectuent des opérations bit à bit sur les données et mettent à jour les indicateurs **N** et **Z**.

Exemples d'instructions implémentées :

— `ANDA`, `ANDB`
— `ORA`
— `EORA`
— `BITA`

Instructions de décalage et rotation

Ces instructions permettent de décaler ou de faire pivoter les bits d'un registre, souvent utilisées pour les traitements binaires.

Exemples d'instructions implémentées :

— `ASLA`, `ASRA`
— `LSRA`
— `ROLA`, `RORA`

Instructions d'incrémentation et décrémentation

Ces instructions modifient les registres de manière unitaire et mettent à jour les indicateurs d'état.

Exemples d'instructions implémentées :

— `INCA`, `DECA`

— `INCB`, `DECB`
— `CLRA`, `CLRB`
— `COMA`, `NEGA`
— `TSTA`, `TSTB`

Instructions de chargement et stockage

Ces instructions permettent de transférer des données entre la mémoire et les registres internes du processeur.

Exemples d'instructions implémentées :

— `LDA`, `LDB`, `LDX`, `LDY`, `LDU`, `LDD`
— `STA`, `STX`, `STY`, `STU`, `STD`

Instructions de transfert entre registres

Ces instructions permettent le transfert ou l'échange de valeurs entre registres internes.

Exemples d'instructions implémentées :

— `TFR` : transfert de registre
— `EXG` : échange de registres

Instructions de pile

Les instructions de pile sont utilisées pour la sauvegarde et la restauration du contexte du processeur, notamment lors des appels de sous-programmes et des interruptions.

Exemples d'instructions implémentées :

— `PSHS`
— `PULS`

Instructions de Sauts et du Branchements

Ces instructions modifient la valeur du compteur ordinal (**PC**) afin de réaliser des sauts, des branchements conditionnels ou des appels de sous-programmes.

Exemples d'instructions implémentées :

— `JMP`, `JSR`, `RTS`
— `BRA`, `BEQ`, `BNE`, `BCC`, `BCS`
— `BMI`, `BPL`, `BGE`, `BLT`, `BGT`, `BLE`
— `BSR`, `BRN`

Instructions d'interruption et de synchronisation

Ces instructions permettent la gestion des interruptions logicielles et matérielles, ainsi que la synchronisation du processeur.

Exemples d'instructions implémentées :

— `SWI`, `SWI2`, `SWI3`

— `RTI`

— `SYNC`

— `CWAI`

5.2 Table des opcodes supportés

Les instructions sont stockées dans une structure de type `Map<Integer, Instruction>` associant chaque opcode à son comportement. La table suivante illustre quelques opcodes représentatifs pris en charge par le simulateur :

Opcode	Instruction	Catégorie
0x8B	ADDA #	Arithmétique
0x86	LDA #	Chargement
0x97	STA direct	Stockage
0x20	BRA	Branchement
0x39	RTS	Contrôle
0x3F	SWI	Interruption

6 Gestion des interruptions

6.1 Définition des interruptions

Une interruption est un mécanisme qui permet au microprocesseur d'interrompre temporairement l'exécution normale d'un programme afin de traiter un événement particulier. Cet événement peut être d'origine matérielle ou logicielle.

Lorsqu'une interruption se produit, le processeur sauvegarde le contexte courant (notamment le compteur ordinal et certains registres), puis exécute une routine spécifique appelée routine d'interruption. À la fin du traitement, le processeur reprend l'exécution du programme initial.

Ce mécanisme est largement utilisé pour la gestion des périphériques, des événements externes et des appels système.

6.2 Types d'interruptions du Motorola 6809

Selon le cours d'architecture des ordinateurs et d'assembleur, le microprocesseur Motorola 6809 définit plusieurs types d'interruptions :

— **IRQ (Interrupt Request)** : interruption matérielle masquable, généralement déclenchée par un périphérique externe.

— **FIRQ (Fast Interrupt Request)** : interruption matérielle rapide, avec une sauvegarde minimale du contexte.
— **NMI (Non Maskable Interrupt)** : interruption non masquable, utilisée pour des événements critiques.
— **SWI (Software Interrupt)** : interruption logicielle déclenchée par une instruction du programme.
— **SWI2 et SWI3** : interruptions logicielles étendues, offrant des vecteurs supplémentaires.

6.3 Interruptions utilisées dans le simulateur

Dans le cadre de ce projet, le mécanisme des interruptions est utilisé pour reproduire le comportement réel du microprocesseur Motorola 6809.

Les interruptions permettent notamment :
— d'interrompre l'exécution normale du programme,
— de sauvegarder le contexte du processeur dans la pile,
— de transférer l'exécution vers une adresse spécifique appelée vecteur d'interruption,
— puis de reprendre l'exécution normale après traitement.

La gestion de la pile joue un rôle central dans ce mécanisme, comme vu dans le cours, où le pointeur de pile (SP) est utilisé pour sauvegarder et restaurer le contexte du processeur.

La gestion des interruptions est réalisée dans les classes liées au processeur et au jeu d'instructions.

Par exemple, les interruptions logicielles sont définies dans la classe qui gère le jeu d'instructions du processeur. Chaque interruption est associée à un opcode et à une action spécifique.

Lorsqu'une interruption logicielle est déclenchée :
— le compteur ordinal (PC) est sauvegardé dans la pile,
— le processeur charge l'adresse du vecteur d'interruption,
— l'exécution est redirigée vers la routine correspondante.

Le retour d'interruption s'effectue grâce à l'instruction `RTI`, qui permet de restaurer le contexte sauvegardé et de reprendre l'exécution du programme principal.

6.4 Composants de l'interface graphique

L'interface graphique du simulateur est composée de plusieurs panneaux permettant de visualiser et de contrôler l'exécution du microprocesseur Motorola 6809 de manière simple et intuitive.

Affichage des registres et des indicateurs

Le panneau des registres permet d'afficher en temps réel l'état des principaux registres du processeur (A, B, X, Y, PC). Les valeurs sont affichées en hexadécimal et mises à jour après chaque instruction.

Les indicateurs d'état (flags N, Z, V, C) sont également affichés afin de suivre l'effet des instructions sur le processeur.

La mise à jour des informations est réalisée à partir des données contenues dans la classe `CPU`.

Affichage de la mémoire

Un panneau mémoire permet de visualiser le contenu de la mémoire sous forme hexadécimale. Cette vue facilite la compréhension des accès mémoire effectués lors de l'exécution des instructions.

Contrôle de l'exécution

L'interface graphique propose plusieurs boutons de contrôle :

— **STEP** : exécution d'une seule instruction,

— **RUN** : exécution continue du programme,

— **STOP** : arrêt de l'exécution,

— **RESET** : réinitialisation du simulateur.

Le mode pas à pas permet de suivre précisément l'évolution des registres et de la mémoire.

Interaction avec le processeur

À chaque action de l'utilisateur, le processeur exécute les instructions correspondantes, puis l'interface graphique met à jour automatiquement les différents panneaux.

Le lien entre l'interface et le processeur est assuré par des appels directs aux méthodes de la classe `CPU`.

Organisation générale

L'interface graphique suit une organisation simple de type Modèle–Vue–Contrôleur :

— le **modèle** représente le processeur et la mémoire,

— la **vue** correspond aux panneaux graphiques,

— le **contrôleur** gère les actions de l'utilisateur.

Cette organisation rend le code plus clair et facilite son évolution.