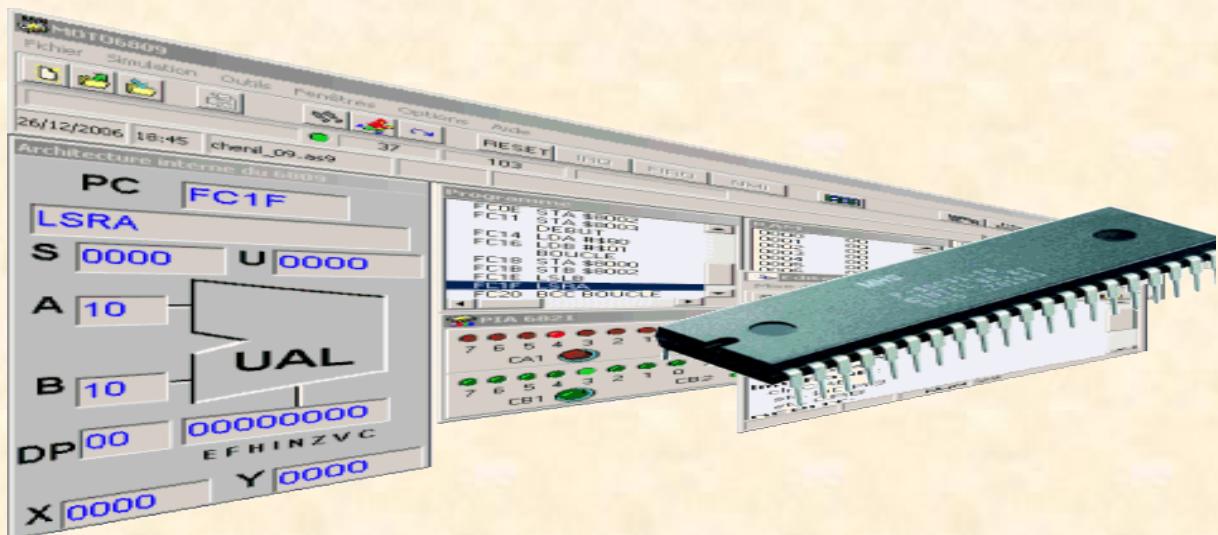


Rapport de Projet : Simulateur du Microprocesseur Motorola 6809

 RAPPORT DÉTAILLÉ DU PROJET MOTOROLA

Encadré par Mr Hicham BENALLA



➤ OBJECTIF :

objectif général du projet est de **modéliser le comportement matériel du Motorola 6809 sous forme logicielle**, en respectant son architecture, son jeu d'instructions, ses registres internes, ses modes d'adressage et son cycle d'exécution. Le simulateur doit permettre l'exécution correcte de programmes écrits en assembleur 6809 et offrir un environnement d'observation et d'analyse du fonctionnement du processeur.

I. L'architecture du programme :

```
src\motorola
  addressing
    AddressingMode.java
    AddressingModeType.java
  assembler
  cpu
    CPU.java
    Debugger.java
  decoder
  gui
  instructions
  memory
```

1. MEMOIRE :

La classe Memory constitue un composant fondamental du simulateur du Motorola 6809. Son implémentation respecte rigoureusement :

- la taille de l'espace d'adressage,
- l'organisation en octets,
- l'ordre big-endian,
- et les mécanismes d'accès mémoire du processeur réel

La classe Memory représente la **mémoire principale** du microprocesseur Motorola 6809 dans le simulateur. Elle modélise fidèlement l'espace d'adressage du processeur, qui est de **64 Ko**, soit des adresses comprises entre 0x0000 et 0xFFFF.

Cette taille correspond exactement au **bus d'adresses sur 16 bits** du Motorola 6809, ce qui garantit une conformité totale avec l'architecture matérielle réelle.

La mémoire est implémentée à l'aide d'un tableau de type `byte[]`, chaque élément représentant une cellule mémoire de 8 bits, conformément à l'organisation mémoire du processeur.

La méthode `reset()` met toutes les cellules mémoire à 0x00, simulant ainsi un état initial propre du système, équivalent à un démarrage à froid du microprocesseur.

2. CPU :

La classe CPU constitue le **cœur du simulateur du microprocesseur Motorola 6809**. Elle modélise le comportement interne du processeur en regroupant :

- les registres matériels,
- l'état d'exécution,
- le cycle de traitement des instructions,
- la gestion des piles,
- et la communication avec la mémoire.

Cette classe joue un rôle équivalent à celui du **microprocesseur réel** dans un système matériel.

Le CPU du simulateur du microprocesseur Motorola 6809 fonctionne selon le **cycle classique Fetch – Decode – Execute**, qui constitue le principe fondamental de l'exécution des instructions dans un microprocesseur.

Dans un premier temps (**Fetch**), le CPU lit l'instruction située à l'adresse indiquée par le compteur ordinal (PC) dans la mémoire, puis met à jour ce compteur afin de pointer vers l'instruction suivante.

Ensuite (**Decode**), l'instruction lue est analysée afin d'identifier l'opération à effectuer ainsi que le mode d'adressage associé. Cette étape permet au processeur de déterminer les ressources nécessaires à l'exécution de l'instruction.

Enfin (**Execute**), l'instruction est exécutée. Le CPU effectue les opérations prévues, met à jour les registres concernés, modifie éventuellement la mémoire et ajuste les indicateurs du registre d'état.

Ce cycle est répété de manière séquentielle pour chaque instruction du programme, permettant une exécution correcte et ordonnée du code assembleur. Dans le simulateur, l'implémentation de ce cycle garantit une reproduction fidèle du fonctionnement du Motorola 6809 et constitue un élément central de la valeur pédagogique du projet.

Le CPU du simulateur est responsable de l'initialisation et de la gestion des **registres internes** et des **flags du processeur Motorola 6809**.

Les registres initialisés sont :

- les **accumulateurs 8 bits** A et B,
- l'**accumulateur 16 bits D** (formé par A et B),
- les **registres d'index X et Y**,
- les **pointeurs de pile S** (System Stack Pointer) et U (User Stack Pointer),
- le **compteur ordinal PC** (Program Counter),
- le **registre de page directe DP** (Direct Page Register),
- le **registre des codes de condition CC**.

Le registre CC regroupe les **flags d'état du processeur**, à savoir :

- **N (Negative)** : indique un résultat négatif,
- **Z (Zero)** : indique un résultat nul,
- **V (Overflow)** : indique un dépassement arithmétique,
- **C (Carry)** : indique une retenue ou un emprunt,
- **I et F** : contrôlent le masquage des interruptions,
- **H** : utilisé pour certaines opérations arithmétiques,
- **E** : utilisé lors des interruptions étendues.

L'initialisation de ces registres et flags garantit un **fonctionnement cohérent et conforme à l'architecture du Motorola 6809**, constituant une étape essentielle avant l'exécution des instructions.

3. DEBUGER :

La classe Debugger permet d'ajouter des fonctionnalités de **débogage** au simulateur du microprocesseur Motorola 6809. Elle offre à l'utilisateur un contrôle précis de l'exécution des programmes, facilitant ainsi l'analyse et la compréhension du fonctionnement du processeur simulé.

Le débogueur est directement associé au CPU et agit comme un **outil d'observation et de contrôle**, sans modifier le comportement interne des instructions.

Le débogueur conserve l'adresse de la **dernière instruction exécutée** et fournit une représentation synthétique de l'état interne du CPU. Cette information est utilisée notamment par l'interface graphique pour afficher les valeurs des registres en temps réel.

4. ADDRESSINGMODETYPE :

La classe AddressingMode a pour rôle de **calculer les adresses mémoire effectives** en fonction des **différents modes d'adressage du microprocesseur Motorola 6809**. Elle agit comme une **classe utilitaire**, appelée par les instructions, afin de traduire les opérandes en adresses mémoire concrètes.

Cette séparation entre le décodage des modes d'adressage et l'exécution des instructions permet :

- une meilleure lisibilité du code,
- une architecture modulaire,
- et une fidélité accrue au fonctionnement du processeur réel.

Modes d'adressage implémentés

1. Mode immédiat (Immediate)

Les méthodes immediate8 et immediate16 permettent de lire directement une valeur intégrée dans le code de l'instruction.

Dans ce mode, l'opérande ne correspond pas à une adresse mémoire, mais à une **valeur immédiate**, lue directement après l'opcode.

Ce mode est utilisé pour charger des constantes dans les registres.

2. Mode direct (Direct)

Le mode direct est implémenté par la méthode direct.

Il combine :

- un **offset sur 8 bits** contenu dans l'instruction,
- avec le **registre DP (Direct Page Register)**.

L'adresse finale est formée en concaténant DP (octet de poids fort) et l'offset (octet de poids faible).

Ce mode permet un accès rapide à une zone mémoire fréquemment utilisée, conformément à l'architecture du 6809.

3. Mode étendu (Extended)

Le mode étendu permet d'accéder à n'importe quelle adresse mémoire sur 16 bits. La méthode extended lit directement une adresse complète sur deux octets depuis la mémoire.

Ce mode est utilisé lorsque l'adresse de l'opérande ne peut pas être exprimée sur 8 bits.

4. Mode étendu indirect (Extended Indirect)

Le mode étendu indirect introduit un **niveau d'indirection supplémentaire**. L'instruction contient l'adresse d'un pointeur mémoire, lequel contient lui-même l'adresse finale de l'opérande.

Ce mode est implémenté par la méthode extendedIndirect et permet de simuler des mécanismes avancés comme les tables de pointeurs.

5. Mode relatif (Relative)

Les méthodes relative8 et relative16 sont utilisées pour les instructions de branchement.

L'adresse cible est calculée en ajoutant un **déplacement signé** au compteur ordinal (PC).

Ce mode permet des branchements avant ou arrière dans le code, ce qui est essentiel pour les boucles et les structures conditionnelles.

6. Mode indexé (Indexed)

Le mode indexé est l'un des plus puissants du Motorola 6809.

Il permet de calculer une adresse à partir d'un **registre d'index** (X, Y, U, S ou PC) et d'un **post-octet** décrivant précisément le mode utilisé.

La classe implémente :

- les offsets courts et longs,
- les incrémentations et décrémentations automatiques,
- l'indexation par accumulateur,
- l'indexation relative au PC.

La méthode centrale decodeIndexedAddress reproduit fidèlement le **format du postbyte du 6809**, ce qui garantit une compatibilité étroite avec la documentation officielle du processeur.

5. DECODER :

Le décodeur d'instructions a pour rôle de **traduire l'opcode lu en mémoire en une instruction exécutable** par le CPU.

Il identifie l'instruction correspondante, ainsi que sa variante (mode d'adressage, taille, page), puis fournit au processeur l'objet instruction approprié pour son exécution.

6. ASSEMBLER ;

L'**assembleur** a pour rôle de **traduire le code assembleur Motorola 6809**

(mnémotechniques, opérandes, labels) en **code machine** (suite d'octets) compréhensible par le CPU.

Il analyse les instructions, détermine le **mode d'adressage**, calcule les **opcodes**, résout les **labels**, puis génère le programme binaire à charger en mémoire pour être exécuté par le processeur.

7. LES INSTRUCTIONS :

1. Transfert de données

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

2. Opérations arithmétiques

- ADD, ADC, SUB, SBC
- MUL

3. Opérations logiques

- AND, OR, EOR (XOR)
- COMA, COMB, COM
- ANDA, ANDB, ORA, ORB, EORA, EORB

4. Comparaisons

- CMP A/B, CMPX, CMPY, CMPPU, CMPS
- TST A/B, TSTA, TSTB

5. Contrôle de flux

- BRA, BRN, BEQ, BNE, BGT, BLT, BGE, BLE
- BCC, BCS, BVC, BVS, BMI, BPL, BHI, BLS
- JSR, JMP, RTS, RTI

6. Manipulation de pile

- PSH A/B, PUL A/B
- PSHS, PULS, PSHU, PULU

7. Spécialisées

- LEA (Load Effective Address), CLR ,INC ,DEC,NOP

8. GUI (swing/awt) :

L'interface graphique (GUI) joue un rôle central dans l'utilisation du simulateur du microprocesseur Motorola 6809. Elle constitue le **point de liaison entre l'utilisateur et les composants internes du simulateur**, notamment le CPU, la mémoire, l'assembleur et le débogueur.

Le GUI permet tout d'abord de **charger et assembler des programmes écrits en assembleur 6809**. L'utilisateur peut ainsi saisir ou importer un programme, lancer le processus d'assemblage, puis charger le code machine généré directement en mémoire pour exécution par le CPU.

L'interface graphique offre également une **visualisation en temps réel de l'état du processeur**. Elle affiche les valeurs des registres internes (A, B, D, X, Y, U, S, PC, DP),

ainsi que le registre des codes de condition (CC). Cette visualisation permet de suivre précisément l'évolution de l'état du CPU au cours du cycle Fetch–Decode–Execute.

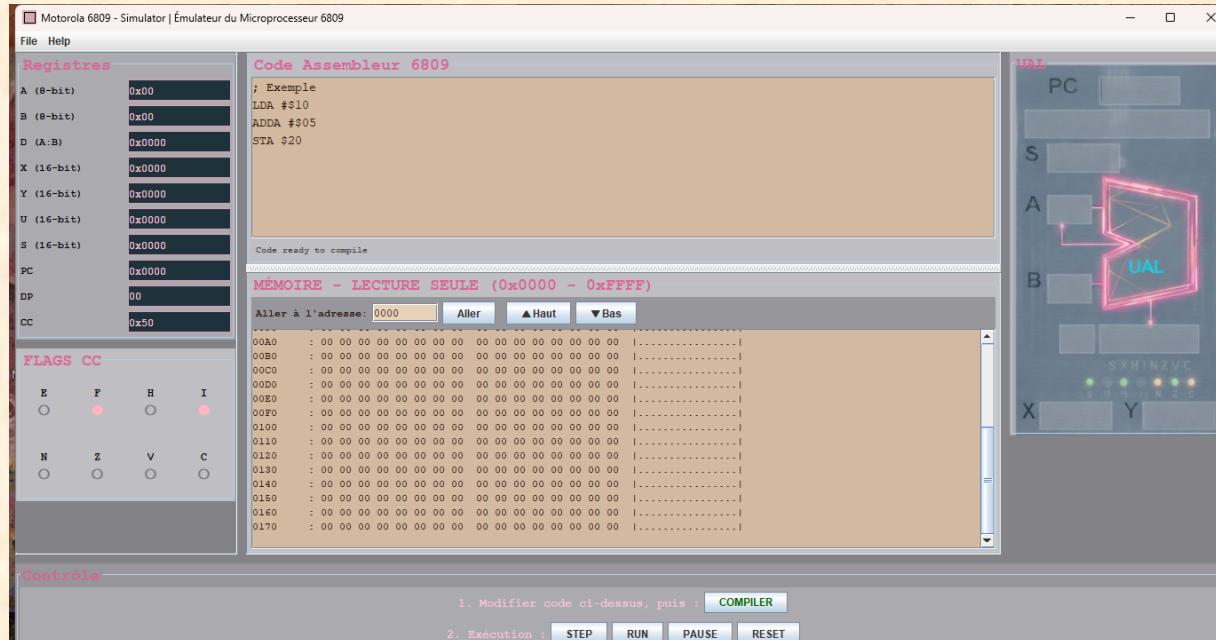
En lien avec la classe Debugger, le GUI permet de **contrôler le déroulement de l'exécution** du programme. L'utilisateur peut :

- exécuter le programme en continu,
- activer le mode pas à pas,
- mettre l'exécution en pause,
- définir et supprimer des points d'arrêt (breakpoints).

Ces fonctionnalités facilitent l'analyse du comportement des instructions et des modes d'adressage.

Le GUI permet aussi d'**observer les accès mémoire**, en affichant le contenu de certaines zones mémoire ou les adresses calculées par les modes d'adressage (direct, étendu, indexé, relatif). Cela renforce la compréhension des mécanismes d'adressage propres au Motorola 6809.

Enfin, l'interface graphique améliore considérablement la **dimension pédagogique du projet**. Elle rend visibles des concepts internes normalement abstraits, tels que l'évolution du compteur ordinal, l'effet des instructions sur les registres et les flags, ou encore l'influence des modes d'adressage sur les accès mémoire.



Après avoir présenté le rôle et le fonctionnement de chaque classe du projet, nous abordons maintenant les **méthodes d'écriture en assembleur associées à chaque mode d'adressage**, afin de montrer comment ces modes sont utilisés concrètement lors de la programmation du microprocesseur Motorola 6809.

❖ **N.B : il faut insérer « ; » après chaque ligne de commandes**

1. Mode immédiat (Immediate)

Écriture assembleur

```
LDA    #$10  
LDD    #$1234
```

Description

Le symbole # indique que l'opérande est une **valeur immédiate** codée directement dans l'instruction. Le symbole \$ précise que la valeur est exprimée en **hexadécimal**.

Ce qui se passe

- L'assembleur reconnaît le symbole #.
- Il génère un opcode suivi de la valeur \$10.
- À l'exécution, le CPU lit cette valeur directement après l'opcode.
- La valeur est chargée dans le registre sans accès mémoire.

L'opérande est une valeur, pas une adresse.

2. Mode direct (Direct)

Écriture assembleur

```
LDA    <$20    LDA    $20  
STA    <$30    STA    $30
```

Description

Le symbole < force l'utilisation du **mode direct**. L'adresse effective est construite à partir du registre **DP** et de l'octet de poids faible fourni. Ou on insère l'adresse direct après la mnémonique sans < .

Ce qui se passe

- L'assembleur force le mode direct grâce à <.
- Il encode uniquement l'offset \$20.
- À l'exécution, le CPU combine :
 - le registre DP (octet haut),
 - l'offset \$20 (octet bas).
- L'adresse mémoire finale est calculée et utilisée.

Accès rapide à une zone mémoire prédefinie.

3. Mode étendu (Extended)

Écriture assembleur

```
LDA  >$4000  
STA  >$8000
```

Description

Le symbole > force le **mode étendu**. L'instruction contient une adresse complète sur 16 bits.

Ce qui se passe

- L'assembleur force le mode étendu avec >.
- Il encode une adresse complète sur 16 bits.
- À l'exécution, le CPU lit cette adresse et accède directement à la mémoire.

Accès libre à toute la mémoire.

4. Mode étendu indirect (Extended Indirect)

Écriture assembleur

```
LDA  [>$4000]
```

Description

L'adresse indiquée pointe vers une **adresse mémoire intermédiaire**, qui contient l'adresse finale de l'opérande.

Ce qui se passe

- L'assembleur encode une adresse sur 16 bits.
- À l'exécution :
 1. le CPU lit l'adresse \$4000,
 2. il lit le contenu mémoire à cette adresse,
 3. ce contenu devient l'adresse finale.
- L'opérande est ensuite lu à cette adresse finale.

Double accès mémoire (indirection).

6. Mode indexé (Indexed) – sans offset

Écriture assembleur

```
LDA    , X  
LDA    , Y
```

Description

L'adresse effective correspond à la valeur contenue dans le registre d'index.

Ce qui se passe

- L'assembleur encode un post-octet indiquant l'utilisation de X.
- À l'exécution, le CPU utilise directement la valeur de X comme adresse mémoire.

Accès indirect via un registre.

7. Mode indexé avec offset constant

Offset 5 bits

```
LDA    3,X  
LDA   -2,Y
```

Offset 8 bits

```
LDA    $20,X
```

Offset 16 bits

LDA \$1000, X

Description

L'offset est ajouté au registre d'index pour calculer l'adresse finale.

Ce qui se passe

- L'assembleur encode l'offset et le registre X.
- À l'exécution, le CPU ajoute l'offset à X.
- Le résultat donne l'adresse finale.

Accès à un élément d'un tableau.

8. Mode indexé avec accumulateur

Écriture assembleur

LDA A, X
LDA B, Y

Description

L'accumulateur A ou B fournit l'offset utilisé dans le calcul de l'adresse.

Ce qui se passe

- L'assembleur encode un mode spécial.
- À l'exécution, le CPU prend la valeur du registre A.
- Cette valeur est ajoutée à X pour former l'adresse finale.

Indexation dynamique.

9. Mode indexé avec auto-incrément / auto-décrément

Post-incrément

LDA , X+

LDA , X++

Pré-décrément

LDA , -X
LDA , --X

Description

Le registre d'index est automatiquement modifié avant ou après l'accès mémoire.

Ce qui se passe

- Le CPU utilise l'adresse contenue dans X.
- Après l'accès mémoire, X est automatiquement incrémenté.decrimenté

Parcours séquentiel automatique.

10. Mode indexé indirect

Écriture assembleur

LDA [, X]
LDA [5, X]

Description

L'adresse calculée est utilisée comme **pointeur** vers l'adresse final