



Le Rapport du Projet

Simulateur du Microprocesseur Motorola 6809



Réalisée par :

- ✓ NEZHA TALEB
- ✓ CHAIMAE LAHRACH
- ✓ HAJAR CHAOUKI
- ✓ AMAL CHAFAOUI

Encadré par :

Mr. HICHAM BENALLA

Filière : Génie Informatique

Année : 2025-2026

Introduction

1

Contexte du projet

Objectifs

Présentation du notre Microprocesseur

2

Architecture générale

Instruction et modes d'adressage

Registre

Mémoire, RAM et ROM

CPU

Interface

Fonctionnement

3

Exécution du programme

Tests et réalisation du programme

4

Scenario

Résultats obtenus

Conclusion

5

Bilan du projet

Introduction

Dans ce projet, On présentera un simulateur du microprocesseur Motorola 6809 qui va nous aider de mieux comprendre l'architecture des ordinateurs.

Ce projet nous a permis de comprendre le comportement, les bases et le fonctionnement du microprocesseur à partir de ses instructions et ses modes d'adresses qu'on va les expliquer en détails dans la suite de notre présentation .

Avec ce simulateur, on peut exécuter des programmes écrits pour le 6809, suivre l'état du mémoire et des registres en direct et comprendre comment les instructions sont traitées.

Objectifs

- Reproduire l'architecture et le fonctionnement du microprocesseur Motorola 6809.
- Exécuter des programmes écrits pour le 6809.

- Enseigner l'architecture et la programmation du Motorola 6809.
- Implémenter des modes d'adressage.
- Vérifier les états des registres étape par étape.

2

Architecture générale

Les principales composantes sont :

- RAM et ROM : gestion de la mémoire.
- UAL : unité arithmétique et logique.
- Interface : liaison avec l'utilisateur.

Instruction

1. La classe instruction qu'on a utilisé dans notre projet :

Présentation de la classe Instruction :

La classe Instruction représente une instruction assembleur pour le microprocesseur Motorola 6809. Elle sert à représenter, analyser, vérifier et assembler une ligne d'instruction (opcode + opérande). Elle

détermine aussi son mode d'adressage et crée le code machine correspondant (sous forme de tableau d'octets). Cette classe est la base logique de l'assembleur dans le simulateur 6809, car elle fait le lien entre le langage assembleur compréhensible par l'humain et le code binaire exécutable par la machine.

Champs et constructeur : initialisation des données :

Les champs opcode et operand sont déclarés comme final, ce qui assure qu'ils ne peuvent pas être modifiés après la construction, ce qui est bien pour la sécurité. Le constructeur effectue un traitement de base : si l'opcode n'est pas nul, il est converti en majuscules avec .toUpperCase() pour que l'interprétation ne tienne pas compte de la casse (par exemple, « Ida » => « LDA »). L'opérande est débarrassé des espaces inutiles via .trim() s'il n'est pas nul, ou remplacé par une chaîne vide sinon. Cette façon de faire permet de gérer différentes entrées utilisateur tout en évitant les erreurs NullPointerException.

Détection du mode d'adressage : detecteMode() :

La méthode detecteMode() regarde la forme de l'opérande (et parfois de l'opcode) pour trouver le mode d'adressage utilisé, selon les règles du 6809. Elle suit une logique précise :

Une instruction END est vue comme une directive, pas comme une instruction exécutable.

Si l'opérande est absent (operand.isEmpty()), on a un mode inherent (par exemple : RTS, NOP).

Un opérande qui commence par # est immédiat.

S'il commence par \$, c'est une adresse : si elle a ≤ 2 chiffres hexadécimaux \rightarrow direct (\$1A), sinon \rightarrow étendu (\$1234).

Une virgule (A,X, 5,Y) indique un mode indexé ; si on a des crochets ([A,X]), on passe à indexé-indirect.

Un opérande entre crochets seul (par exemple, [10]) est vu comme étendu-indirect.

Enfin, certaines instructions comme BEQ, JMP, etc., utilisent toujours le mode relatif, même sans signe particulier.

Cette méthode couvre les principaux modes d'adressage du 6809, en donnant la priorité aux constructions les plus spécifiques (par exemple, # avant \$), pour éviter les confusions.

Création du code opcode hexadécimal :
getOpcodeHex() :

La méthode getOpcodeHex() convertit l'instruction en sa version hexadécimale, en tenant compte de l'opcode et du mode d'adressage. Elle utilise une structure switch organisée comme ceci :

D'abord par opcode (LDA, BEQ, RTS, etc.),
Ensuite, pour les opcodes polymorphes (qui dépendent du mode), un autre switch sur le mode détecté.

Par exemple, LDA a quatre codes machine différents : 86 (immédiat), 96 (direct), B6 (étendu), A6 (indexé). De même, LDY a besoin d'un préfixe d'opcode à deux octets (10 8E, etc.), ce qui est géré directement. Les cas non pris en charge renvoient une chaîne vide, ce qui indique une instruction non compatible. Cette table de

correspondance est basée sur la documentation officielle du 6809.

Vérification de la syntaxe : `estSyntaxValide()` :

Avant de créer le code, la méthode `estSyntaxValide()` vérifie que l'instruction respecte les règles de syntaxe du langage. Elle renvoie false si :

Le mode est Inconnu ou l'opcode est vide.

Pour les opérandes immédiats, la valeur doit être une constante hexadécimale de 1 ou 2 octets (#\$A, #\$F5, #\$1234), précédée de #.

Pour les modes direct et étendu, l'opérande doit commencer par \$, suivi d'une valeur hexadécimale de 1–2 caractères (direct) ou 1–4 caractères (étendu), sans être trop grand ni avoir de caractères incorrects.

Les modes indexés doivent avoir une virgule.

Les instructions inherent ne doivent pas avoir d'opérande.

Cette vérification précoce permet de détecter les erreurs de frappe (par exemple, LDA \$G1, STA #10) et d'éviter des problèmes lors de l'assemblage.

Diagnostic d'erreur : getMessageErreur() :

La méthode getMessageErreur() donne un message clair à l'utilisateur. En cas d'erreur de syntaxe (détectée par estSyntaxeValide()), elle renvoie un message indiquant l'opcode et l'opérande tels qu'écrits par l'utilisateur (par exemple : Erreur : syntaxe incorrecte → 'LDA \$G1'). Sinon, elle confirme OK. C'est important pour un simulateur pédagogique, car ça aide l'utilisateur à corriger ses erreurs.

Assemblage binaire : assemble() :

La méthode assemble() crée le code machine sous forme de tableau d'octets (byte[]), prêts à être chargés en mémoire ou exécutés. Après vérification, elle :

Convertit d'abord l'opcode hexadécimal en un ou plusieurs octets (en gérant les préfixes comme 10 pour LDY).

Récupère la valeur utile de l'opérande en enlevant les symboles (\$, #, ,, [,]) avec une expression régulière.

Crée les octets d'opérande selon le mode :

Immédiat : 1 octet pour 8 bits (#\$FF → FF), 2 octets big-endian pour 16 bits (#\$1234 → 12 34).

Direct / Étendu : toujours big-endian ; les adresses courtes (\$0A) sont étendues à 2 octets (00 0A).

Indexé : seul le décalage (offset) est codé (par exemple, 5,X → 05) ; les cas spéciaux comme ,X → 84 sont codés directement.

Relatif : pour simplifier, l'offset est fixé à 00 (à remplacer par un calcul dans une version plus poussée).

À noter que END renvoie un tableau vide, car c'est une directive d'assemblage. Cette méthode assure que la syntaxe, le sens et la version binaire soient cohérents.

2. Les instructions qu'on a utilisé dans notre projet :

LD (Load) : Charge une valeur dans un registre ou une adresse mémoire.

ST (Store) : Sauvegarde une valeur dans une adresse mémoire.

ADD : Additionne les valeurs de deux registres et stocke le résultat.

SUB : Soustrait les valeurs de deux registres et stocke le résultat.

AND : Effectue une opération logique ET entre deux registres.

OR : Effectue une opération logique OU entre deux registres.

CMP (Compare) : Compare deux registres en soustrayant leur contenu sans stocker le résultat.

JMP (Jump) : Effectue un saut inconditionnel vers une autre adresse de code.

BNE (Branch if Not Equal) : Effectue un saut si les registres comparés ne sont pas égaux.

BEQ (Branch if Equal) : Effectue un saut si les registres comparés sont égaux.

INC (Increment) : Incrémente une valeur dans un registre.

DEC (Decrement) : Décrémente une valeur dans un registre.

Modes d'adressage

6.

1

2

3

4

5

Adressage immédiat :

Le symbole « # » signifie immédiat, ce type d'adressage permet de charger les registres internes du microprocesseur avec la valeur de l'opérande.

5.

1

2

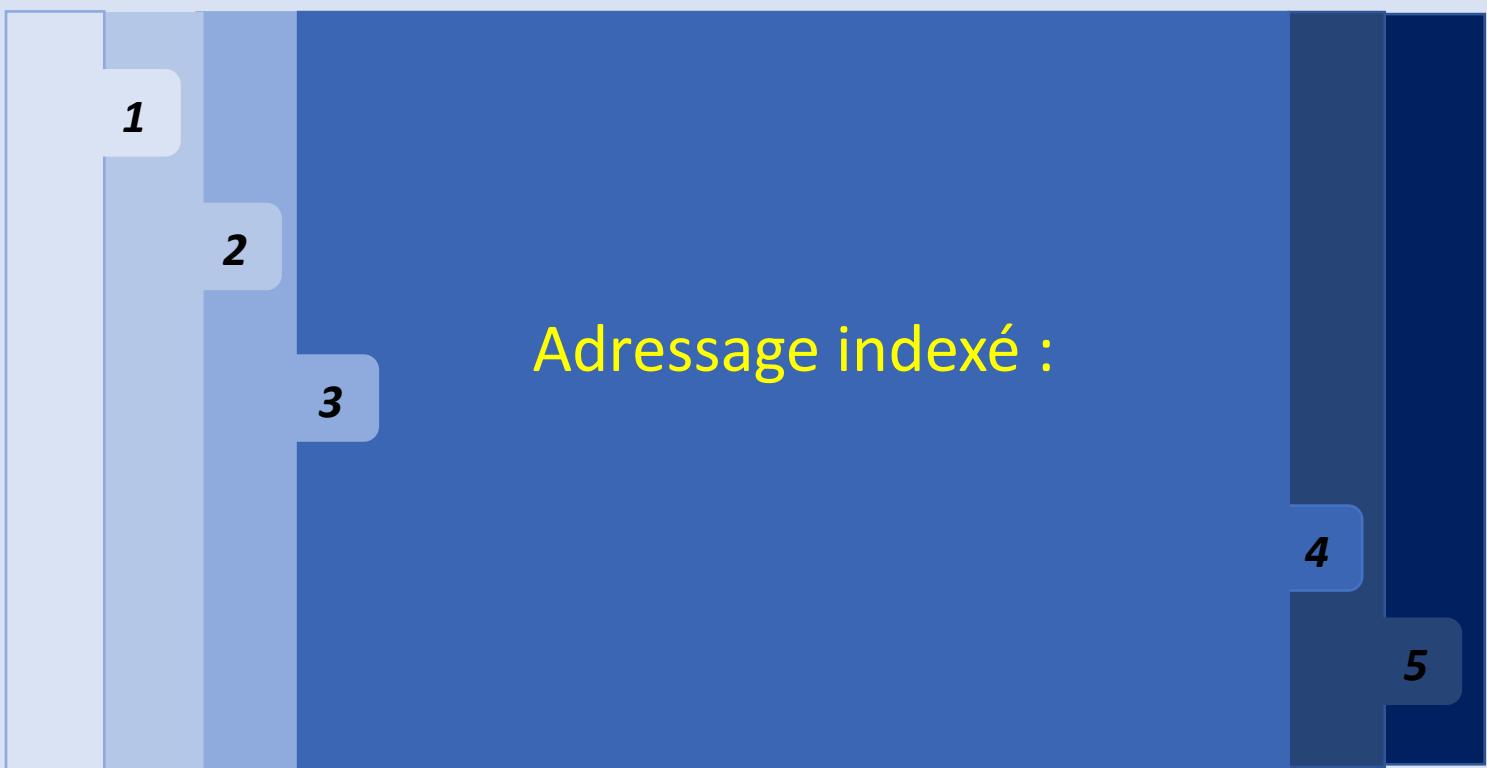
3

4

5

Adressage inhérent :

4.



3.



2.

1

2

Adressage direct :

3

4

5

1.

1

2

Adressage indirect :

3

4

5

Registre

1.Les registres qu'on a utilisé dans notre projet :

Registres

Taille

Description

Accumulateurs A,B.

8bits.

les calculs arithmétiques et les instructions de comparaison .

Registres d'index X,Y.

16 bits.

stockage des résultats intermédiaires et pointeur de données (adresses).

Pointeurs de pile S,U.

16 bits.

S:Pointeur vers la pile système.
U:le passage des paramètres .

Registre compteur programme PC.

16 bits.

Pointeur vers la prochaine instruction à exécution.

Registre de page DP.

8 bits.

Les drapeaux de l'état du processeur.

1.La classe Register qu'on a utilisé dans notre projet :

Dans la classe Register, on a déclaré les registres A, B, X, Y, PC, SP, CC, u, s, dp ainsi que flags de 8 bits de condition (C, V, Z, N, I, H, F, E) qui sont utilisés pour stocker les résultats des opérations arithmétiques et logiques, chaque flag occupe un bit spécifique dans le registre CC.

Les registres A et B ce sont des registres de 8 bits qui servent d'accumulateurs, les registres X et Y sont des registres de 16 bits qui servent d'index, le registre PC est le compteur de programme qui contient l'adresse de la prochaine instruction à exécuter, le registre SP est le pointeur de pile qui contient l'adresse de la pile.

Les flags de condition sont utilisés pour indiquer les résultats des opérations,

tels que:

-Débordement (Overflow OV ou V)

-Nullité (Zéro Z)

-Négativité (Sign S)

-Retenue (Carry C)

-Retenue intermédiaire (Auxiliary-Carry H)

-Parité (Parity P)

-Etat de sauvegarde (E)

-Masque des interruptions (IRQ I)

-Masque des interruptions (FIRQ F)

Nous avons également implémenté les méthodes get() et set() pour accéder et modifier ces registres et flags,

les méthodes update() pour mettre à jour les flags en fonction des résultats des opérations, la méthode reset() pour réinitialiser les registres et les flags et finalement la méthode printState() pour afficher l'état des registres et des flags.

RAM, ROM et Mémoire

1.RAM :

On a présenté une classe RAM dans notre projet comme un tableau de 1024 octets qui représentera la mémoire RAM du notre simulateur. Chaque élément du tableau représente une adresse mémoire du 0000 à 03FF .

Elle contient deux méthodes :

- read(adresse) :lire une valeur à une adresse dans la mémoire RAM.

- `write(adresse,valeur)` : écrire une valeur à une adresse dans la mémoire RAM.

2.ROM :

Il prend un tableau aussi de 1024 octets de FC00 à FFFF qui représente le programme à charger dans la mémoire ROM.

Elle contient une seule méthode :

- `read(adresse)` : lire une valeur à une adresse dans la mémoire ROM.

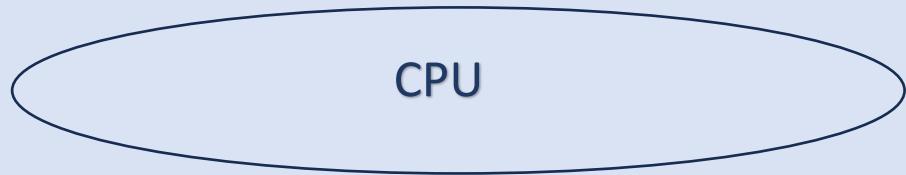
3.Memoire :

La mémoire est représentée par la classe memoire. Elle fait l'appel à RAM et ROM.

Elle contient deux méthodes :

- `read(adresse)` : lire une valeur à une adresse dans la mémoire mémoire. Elle vérifie si l'adresse se trouve dans l'intervalle de la mémoire RAM ou ROM et fait l'appel à la méthode `read` correspondante.
- `write(adresse,valeur)` : écrire une valeur à une adresse dans la mémoire. Elle vérifie si l'adresse se trouve dans l'intervalle de la mémoire RAM et fait l'appel à la méthode `write`. Si l'adresse est dans l'intervalle de la mémoire ROM, elle ne fait pas

l'appel car la mémoire ROM est en lecture seulement.



La classe Cpu représente le processeur Motorola 6809 dans notre émulateur. Elle gère:

- la lecture des instructions.
- leur exécution.
- la manipulation des registres et de la mémoire.

La classe contient des méthodes pour initialiser, exécuter et contrôler l'état du CPU, ainsi que des fonctions auxiliaires pour les opérations arithmétiques et logiques.

1. Attributs principaux

- reg : une instance de la classe Register qui contient tous les registres du CPU (A, B, X, Y, PC, SP, DP, et les flags conditionnels).
- mem : une instance de la classe Memoire représentant la mémoire du CPU.

-halted : un booléen indiquant si le CPU est arrêté (true) ou prêt à exécuter (false).

Ces attributs permettent au CPU de suivre l'état interne et de savoir quelles instructions exécuter.

2. Initialisation et contrôle

reset() : remet le CPU à l'état initial. Il réactive le CPU (halted = false) et réinitialise tous les registres et flags via reg.reset().

isHalted() : retourne l'état du CPU, permettant de vérifier s'il est arrêté ou non.

Ces méthodes sont essentielles pour démarrer ou redémarrer l'exécution d'un programme.

3. Lecture des instructions

fetch8() : lit un octet depuis la mémoire à l'adresse pointée par le compteur de programme (PC), puis incrémente le PC.

fetch16() : lit deux octets consécutifs (format big-endian) et les combine en un entier 16 bits. Cette opération est utilisée pour les instructions nécessitant une adresse ou une valeur sur deux octets.

Ces méthodes permettent de décoder correctement les instructions et leurs opérandes.

4. Exécution des instructions

-step() : exécute une seule instruction pointée par le PC.
Elle lit l'opcode, puis effectue le traitement correspondant via un switch sur l'opcode.

Les instructions couvertes incluent :

Chargement / stockage : LDA, LDB, LDX, LDY, STA, STB, STX (avec modes direct, étendu et indexé).

Opérations arithmétiques : ADD, SUB, MUL, INCA, DECA.

Opérations logiques : ANDA, ANDB, COMA, COMB, CLRA, CLR.B.

Comparaisons : CMP A, CMP B.

Branches et sauts : JMP, BRA, BNE, BEQ, BMI, BPL.

Arrêt du CPU : HALT.

-run() : exécute le programme jusqu'à ce que le CPU soit arrêté (halted = true), en appelant step() de manière répétée.

5. Méthodes auxiliaires

Pour simplifier le traitement des instructions, la classe contient des méthodes privées telles que :

- Opérations arithmétiques : addA(), addB(), subA(), subB(). Elles mettent également à jour les flags conditionnels (N, Z, H, C).
- Comparaisons : cmpA(), cmpB(). Elles ne modifient pas les registres mais mettent à jour les flags pour les instructions de branchement.
- Branches conditionnelles : branchIf(boolean condition) permet de sauter à une adresse relative en fonction d'une condition (utile pour BNE, BEQ, BMI, etc.).

6. Gestion de la mémoire et des registres

La classe utilise les registres et flags via reg pour :

Calculer les adresses mémoire (par exemple pour le mode indexé).

Mettre à jour les flags conditionnels après chaque opération.

Lire et écrire des valeurs dans la mémoire via mem.write() et mem.read().

Interface

Le code présenté crée une interface graphique complète pour un simulateur du microprocesseur Motorola 6809, entièrement développé en Java. Il utilise deux bibliothèques graphiques de base du langage : AWT (Abstract Window Toolkit) et Swing. AWT fournit les classes nécessaires pour gérer les fenêtres, les événements et le dessin graphique de base, notamment via les packages `java.awt` et `java.awt.event`. Swing, qui fait partie de `javax.swing`, enrichit cette base avec des composants plus modernes, flexibles et personnalisables, comme `JFrame`, `JTextArea`, `JButton` et `JToolBar`. Cette combinaison permet de créer une interface réactive et modulaire, tout en ajoutant des éléments graphiques personnalisés, comme les lignes qui relient les registres à l'UAL. Ces lignes sont dessinées à l'aide de la méthode `paintComponent` héritée de la classe `JPanel`, qui utilise le canevas graphique d'AWT via `Graphics`. L'ensemble est conçu selon un modèle d'événements strict, garantissant une interaction fluide entre l'utilisateur, l'éditeur de code, le moteur d'assemblage et le simulateur du processeur.

L'architecture générale est basée sur une séparation claire entre la logique de simulation (classes Cpu,

Register, Mémoire, RAM, ROM, Instruction) et l'interface utilisateur. La méthode main sert de point d'entrée et initialise d'abord les composants essentiels de la simulation. Cela inclut une instance de mémoire avec une ROM de 1024 octets, qui est vide au départ, une RAM conforme à l'architecture cible, et un registre et un processeur associés. Le processeur est immédiatement réinitialisé via la méthode reset() pour établir un état de départ connu. Cette initialisation garantit que chaque session de simulation commence dans des conditions identiques.

L'interface principale se compose de plusieurs fenêtres indépendantes mais synchronisées. La fenêtre principale affiche une barre de menus complète (Fichier, Simulation, Outils, Fenêtres, Options, Aide) et une barre d'outils graphique intuitive. Cette barre d'outils offre des raccourcis visuels pour des opérations courantes : ouverture et enregistrement de fichiers, réinitialisation du processeur, déclenchement d'interruptions (IRQ, FIRQ, NMI), exécution continue, arrêt et exécution pas à pas. L'exécution continue s'effectue dans un thread dédié pour ne pas bloquer l'interface graphique. Les mises à jour de l'affichage des registres et de la mémoire se font de manière thread-safe grâce à SwingUtilities.invokeLater. Cela assure une

utilisation fluide même lors de longues boucles d'exécution.

La fenêtre des registres représente visuellement l'architecture interne du 6809. Elle affiche les huit registres principaux : PC (compteur de programme), A et B (accumulateurs 8 bits), X et Y (registres d'index 16 bits), S et U (pointeurs de pile système et utilisateur), et DP (registre de page directe 8 bits). De plus, les huit bits d'état du registre de flags (H, N, Z, V, C, F, I, E) sont affichés en binaire. Une zone graphique centrale représente l'Unité Arithmétique et Logique (UAL) et montre visuellement les liens entre les registres A, B et DP. Ces liens sont tracés dynamiquement à l'aide d'une surcharge de paintComponent dans un JPanel personnalisé. Cette approche utilise efficacement AWT pour le dessin et Swing pour la gestion des conteneurs. Cette représentation aide à comprendre le flux de données dans le processeur.

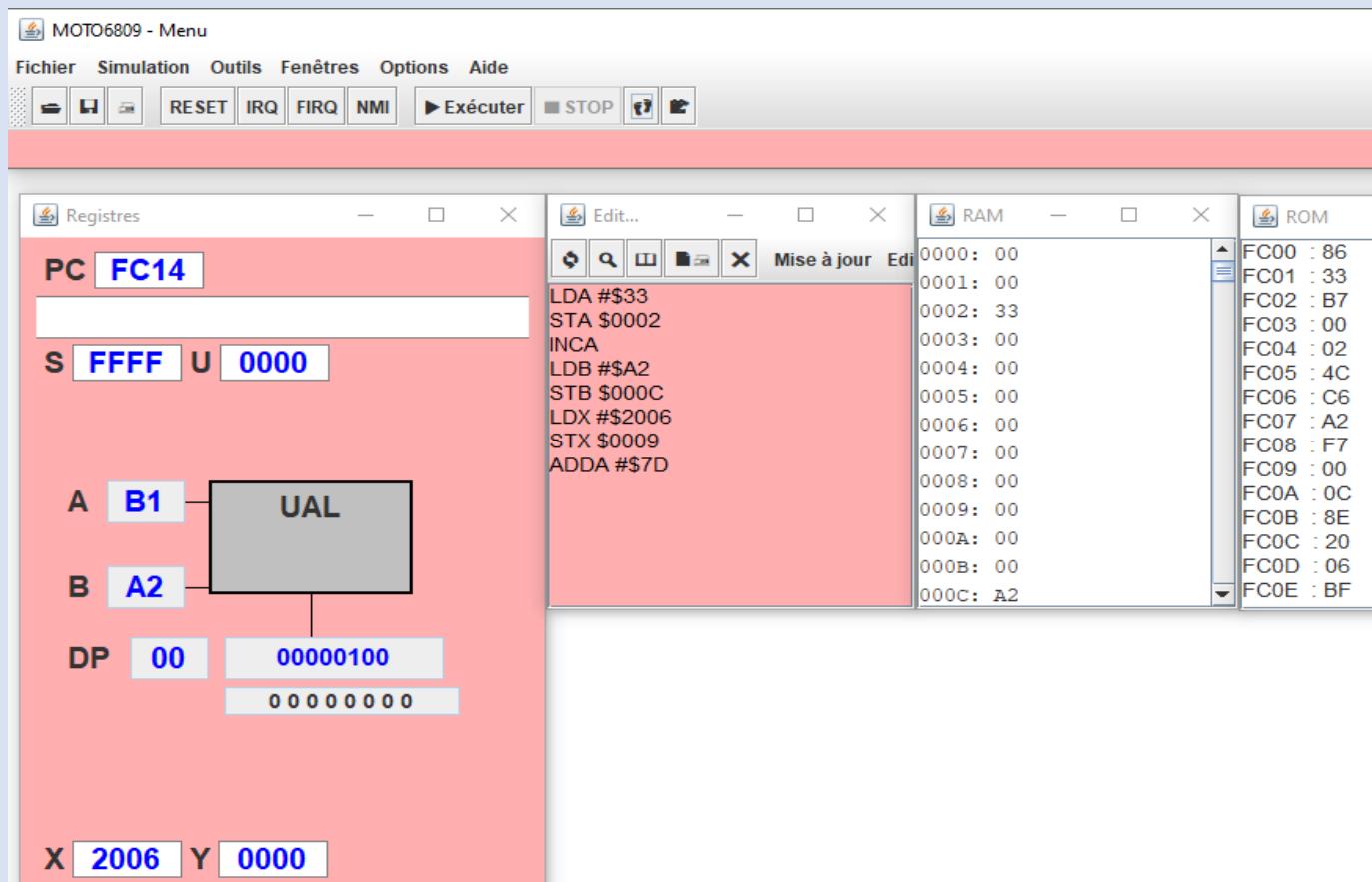
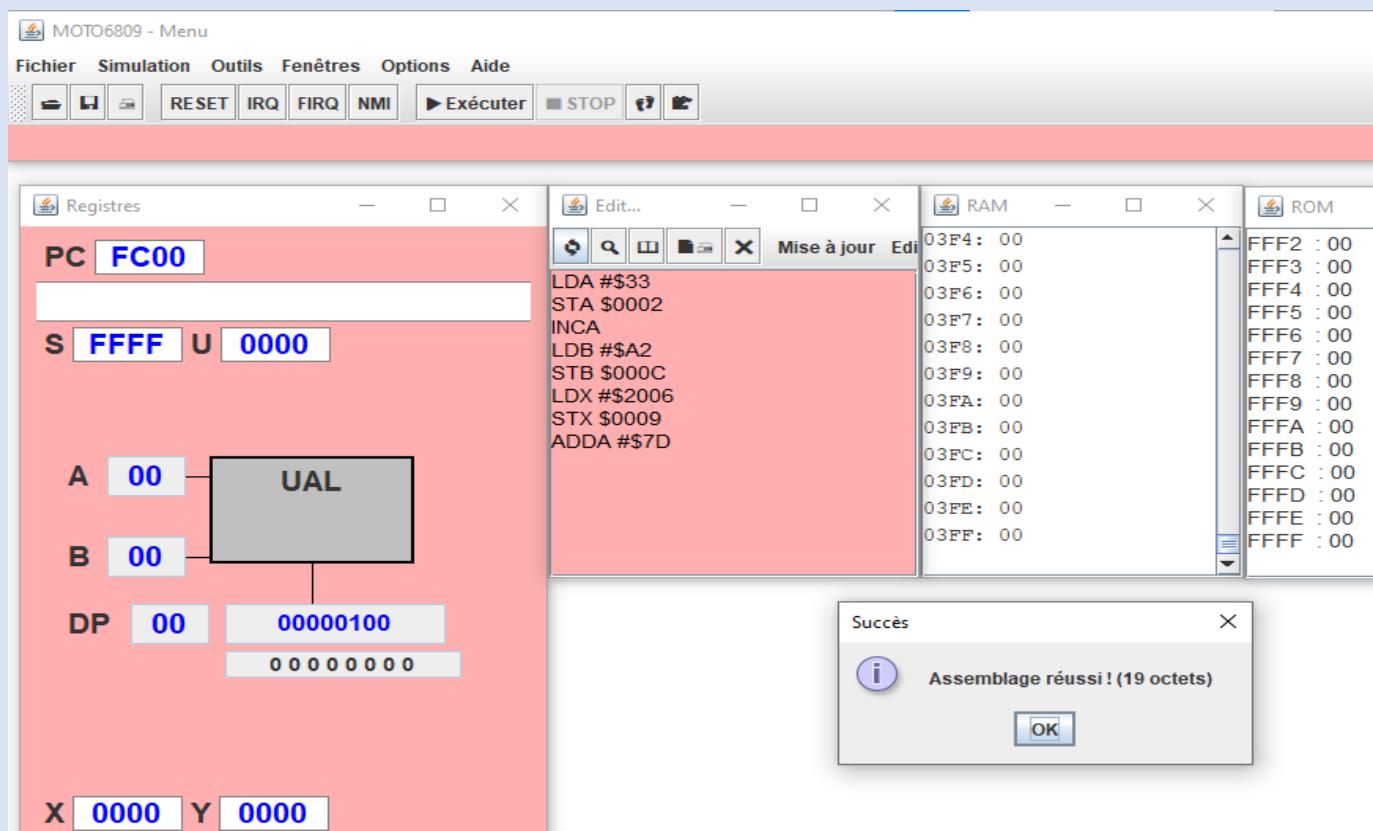
L'éditeur de code source est le cœur de l'interaction utilisateur. Il permet de saisir du code assembleur dans une zone de texte multiligne (JTextArea), avec une police appropriée et une mise en page claire. Un bouton déclenche la compilation du code via la

méthode assemblerCode(). Cette méthode analyse le contenu ligne par ligne, ignore les commentaires (qui commencent par ;) et les lignes vides, et interprète chaque instruction à l'aide d'une classe Instruction, qui n'est pas incluse ici mais qui est supposée être bien implémentée. En cas d'erreur syntaxique ou de dépassement de la capacité mémoire (1024 octets), un message d'erreur apparaît via JOptionPane. À la fin d'un assemblage réussi, la nouvelle image binaire est chargée dans la ROM, le processeur est réinitialisé, et toutes les fenêtres d'affichage sont mises à jour automatiquement. Cette boucle de rétroaction immédiate est essentielle pour un outil d'apprentissage efficace.

Enfin, deux fenêtres dédiées affichent le contenu de la RAM (adresses 0x0000 à 0x03FF) et de la ROM (adresses 0xFC00 à 0xFFFF). Pour améliorer la lisibilité, seules les cases mémoire dont la valeur diffère de 0xFF (valeur par défaut non initialisée) sont montrées. Cela évite l'encombrement visuel et se concentre sur les données importantes modifiées par l'exécution du programme. Cette approche est particulièrement utile lors de démonstrations de programmes courts ou de débogage d'algorithmes simples.

En somme, cette interface graphique offre un environnement de simulation complet, rigoureux et éducatif, parfaitement adapté à l'enseignement de l'architecture des microprocesseurs et de la programmation en assembleur. Elle combine une présentation claire des concepts fondamentaux (registres, mémoire, exécution, interruptions) avec des outils pratiques (édition, assemblage, visualisation dynamique), rendant l'apprentissage du Motorola 6809 accessible et engageant. L'utilisation habile d'AWT pour le dessin de bas niveau et de Swing pour les composants interactifs montre une bonne maîtrise des outils graphiques de Java dans un contexte éducatif exigeant.





|k