



Simulateur de Microprocesseur Motorola 6809

Année Universitaire 2024 - 2025

Université Hassan 1er

Faculté des Sciences et Techniques Settat
Licence Sciences et Techniques en Génie Informatique

Encadré par :

Mr. BENALLA Hicham

Réalisé par :

LAMRISSI Bahaa-eddine

OUAZINI Mohammed

SALIHI Safwane

Lien Github du Projet

Table des matières

1	Introduction Générale	3
2	Étude du processeur Motorola 6809	4
2.1	Aperçu historique	4
2.2	Caractéristiques techniques	4
2.3	Impact industriel	4
3	Architecture et conception du simulateur	5
3.1	Description des composants clés	5
4	Fonctionnalités principales	7
4.1	Émulation précise du processeur Motorola 6809	7
4.2	Gestion mémoire	7
4.3	Débogage et visualisation	8
4.4	Contrôle de l'exécution	9
5	Mise en œuvre technique	10
5.1	Analyse des fichiers principaux (Main.Java)	10
5.1.1	Importations et Déclarations de Classes	10
5.1.2	Méthodes de Gestion de Fichiers	11
5.1.3	Méthode Main	12
5.1.4	Méthodes de Gestion des Instructions	14
5.1.5	Méthodes Utilitaires	18
5.1.6	Exécution des Instructions Spécifiques	20
5.2	Analyse des fichiers principaux (Instruction.java)	22
5.2.1	Déclaration de la Classe et des Variables	23
5.2.2	Constructeur de la Classe	23
5.2.3	Liste des Opcode Valides	23
5.2.4	Liste des Labels Valides	24
5.2.5	Méthode pour Détecter le Mode d'Adressage	24
5.2.6	Méthode pour Valider la Syntaxe de l'Instruction	24
5.2.7	Méthode pour Valider la Taille de l'Opérande	25
5.2.8	Méthode pour Vérifier si l'Opérande est en Hexadécimal	25
5.2.9	Méthode pour Valider l'Opérande	26
5.2.10	Méthode pour Obtenir l'Adresse	26
5.2.11	Méthode pour Obtenir le Code de l'Instruction	27
5.2.12	Méthode pour Filtrer l'Opérande	29
5.3	Présentation de classes Des interfaces graphiques	29

5.3.1	Les Fichiers RAM.java et ROM.java :	29
5.3.2	Les Fichiers Controler.java et Registre.java :	32
6	Tests	35
7	Défis et solutions	44
7.1	Incapacité à Encoder Directement les Valeurs dans la Mémoire Principale de la Machine	44
7.2	Manipulation Directe des Entrées de Chaînes de Caractères	44
7.3	Court Délai pour la Simulation de l'Ensemble d'Instructions Motorola 6809 . . .	44
8	Conclusion et perspectives	46
9	References	47

1. Introduction Générale

Le Motorola 6809 est un microprocesseur à 8 bits lancé par Motorola en 1978, marquant une avancée importante dans l'informatique. Conçu comme une version améliorée du Motorola 6800, il répondait aux besoins croissants en performance et flexibilité à l'époque. Ce processeur, intégré dans des ordinateurs personnels, des systèmes industriels et des applications embarquées, se distingue par son architecture avancée, ses instructions faciles à programmer et ses performances élevées. Bien qu'il n'ait pas connu un succès commercial aussi grand que certains concurrents, le 6809 a influencé les processeurs suivants et reste apprécié des passionnés d'informatique rétro et des éducateurs pour ses innovations techniques.

Objectifs globaux du projet

Les objectifs principaux sont les suivants :

- Simuler fidèlement le processeur Motorola 6809.
- Émuler ses instructions avec une gestion précise de la mémoire et des registres.
- Fournir une interface graphique simple pour interagir avec la simulation.
- Préserver numériquement les logiciels anciens et tester des applications historiques.

2. Étude du processeur Motorola 6809

2.1 Aperçu historique

Dans les années 1970, les microprocesseurs commençaient à révolutionner l'industrie informatique. Motorola avait déjà lancé le 6800 en 1974, mais face à une concurrence croissante de fabricants comme Intel (avec le 8080) et Zilog (avec le Z80), la société a développé le 6809 pour répondre à ces défis.

2.2 Caractéristiques techniques

Le Motorola 6809 est reconnu pour ses caractéristiques avancées et ses innovations techniques :

- **Jeu d'instructions avancé** : Le 6809 disposait d'un ensemble d'instructions riche et orthogonal, ce qui signifie que presque toutes les instructions pouvaient s'appliquer à n'importe quel mode d'adressage. Cela simplifiait considérablement la programmation.
- **Architecture double accumulateur** : Le processeur incluait deux accumulateurs (A et B) qui pouvaient être combinés en un registre 16 bits (D), permettant une plus grande flexibilité dans les opérations arithmétiques.
- **Modes d'adressage diversifiés** : Le 6809 offrait plusieurs modes d'adressage avancés, notamment des modes indexés puissants, rarement vus dans d'autres microprocesseurs 8 bits.
- **Conception efficace** : Par rapport à ses prédécesseurs, il nécessitait moins de cycles d'horloge pour exécuter les instructions, ce qui le rendait rapide et efficace.

2.3 Impact industriel

Adopté dans divers domaines, ce microprocesseur a marqué l'industrie informatique grâce à ses caractéristiques uniques et ses nombreuses applications :

- **Ordinateurs personnels** : Le 6809 a été utilisé dans des ordinateurs tels que le Tandy TRS-80 Color Computer (Coco) et le Dragon 32/64, contribuant à l'essor des ordinateurs personnels accessibles.
- **Systèmes embarqués** : Grâce à sa flexibilité et à sa puissance, il a trouvé des applications dans des systèmes embarqués, des équipements industriels et divers périphériques électroniques.

3. Architecture et conception du simulateur

3.1 Description des composants clés

Dans le cadre du développement d'un simulateur informatique complexe pour le processeur Motorola 6809, plusieurs composants logiciels jouent un rôle crucial pour assurer son fonctionnement harmonieux et efficace. Les fichiers **Main.java**, **Instruction.java**, **Motorola.java**, **RAM.java**, **ROM.java**, **Registre.java**, **Controller.java**, **ComponentsPanel.java**, **DarkUIScrollbar.java**, et **ShadowBorder.java** constituent les éléments fondamentaux de ce simulateur, chacun apportant une fonctionnalité spécifique et essentielle à l'ensemble du système. Ces fichiers, écrits en Java, un langage de programmation orienté objet et largement utilisé, sont conçus pour interagir ensemble dans le but de simuler le comportement du processeur Motorola 6809.

1. **Main.java** : Ce fichier est le point d'entrée principal du simulateur. Il orchestre l'initialisation des différents composants du simulateur, coordonne l'exécution du programme et contrôle l'interaction avec l'utilisateur. Ce fichier est essentiel pour démarrer l'application et gérer les processus du simulateur.
2. **Instruction.java** : Ce fichier est responsable de l'exécution des instructions spécifiques au processeur Motorola 6809. Il contient la logique qui décode et exécute les instructions, assurant ainsi que le simulateur fonctionne correctement selon les spécifications du processeur.
3. **Motorola.java** : Ce fichier modélise le processeur Motorola 6809. Il gère les registres, les cycles d'exécution des instructions, et les interactions avec la mémoire. Il est le cœur du simulateur, permettant d'émuler le comportement du processeur en temps réel.
4. **RAM.java** : Ce fichier simule la mémoire vive (RAM) du système. Il permet de stocker et de manipuler les données temporaires utilisées par le processeur. Ce fichier est essentiel pour comprendre comment la mémoire est utilisée et gérée dans le simulateur.
5. **ROM.java** : Ce fichier simule la mémoire morte (ROM) du système. Il contient les instructions du programme à exécuter et offre une interface pour accéder aux données stockées. Ce fichier joue un rôle clé dans la préservation de la structure du programme pendant l'exécution.
6. **Registre.java** : Ce fichier permet de simuler et de manipuler les registres du processeur. Il offre une vue en temps réel des valeurs des registres, permettant aux utilisateurs de suivre l'état du processeur au fur et à mesure de l'exécution des instructions.

7. **Controller.java** : Ce fichier est responsable de l'interaction entre l'utilisateur et le simulateur. Il gère les événements utilisateur comme les commandes de démarrage, d'arrêt, et de réinitialisation, tout en fournissant des mises à jour sur l'état de l'émulation.
8. **ComponentsPanel.java** : Ce fichier définit l'interface graphique des composants du simulateur. Il présente les différents éléments du simulateur (RAM, ROM, registres) dans une interface claire et interactive, facilitant l'interaction de l'utilisateur avec le simulateur.
9. **DarkUIScrollbar.java** et **ShadowBorder.java** : Ces fichiers sont utilisés pour personnaliser l'apparence graphique de l'interface utilisateur.
DarkUIScrollbar.java crée une barre de défilement au style sombre et moderne, tandis que **ShadowBorder.java** ajoute des bordures ombragées pour améliorer l'esthétique de l'interface.

4. Fonctionnalités principales

4.1 Émulation précise du processeur Motorola 6809

Le simulateur est conçu pour émuler avec précision le fonctionnement du processeur 6809. Cela inclut l'exécution des instructions dans leur ordre et cycle respectif, ainsi que la gestion des interruptions. Ce mode d'émulation permet à l'utilisateur de simuler de manière détaillée le comportement du processeur dans différents scénarios, ce qui est essentiel pour tester des programmes à bas niveau ou pour l'enseignement de l'architecture de processeurs.

Détails techniques

- Chaque instruction est traitée selon ses spécifications : certaines instructions, par exemple LD, ADD, ou SUB, modifient l'état des registres et des indicateurs en fonction du mode d'adressage utilisé (immédiat, direct, etc.).
- Le simulateur supporte l'exécution des instructions de manipulation des bits, comme les opérations AND, OR, EOR, permettant de tester des programmes plus complexes.
- Les interruptions, comme les interruptions matérielles ou logicielles (par exemple, SWI), sont simulées et peuvent être déclenchées par l'utilisateur ou automatiquement en fonction de l'état du processeur.

Cette émulation permet à l'utilisateur de vérifier l'exactitude de l'exécution de chaque instruction, un aspect fondamental dans les systèmes embarqués et la programmation bas niveau.

4.2 Gestion mémoire

Le simulateur gère deux types de mémoire : la RAM (mémoire vive) et la ROM (mémoire en lecture seule). Ces deux espaces mémoire sont cruciaux pour le bon fonctionnement du processeur et la gestion des programmes.

Détails techniques

- **RAM** : Elle est utilisée pour stocker les données temporaires et les variables pendant l'exécution des programmes. Le simulateur permet de manipuler directement la mémoire RAM en accédant à des adresses spécifiques, et il fournit une interface graphique pour visualiser et modifier les données en temps réel.
- **ROM** : Elle contient le programme à exécuter, ou le code machine. Le simulateur permet de charger un fichier `.asm` compilé et de l'exécuter directement dans la ROM.

Le simulateur assure la gestion correcte de ces deux types de mémoire. Par exemple, si un programme tente d'accéder à une adresse mémoire invalide, le système peut lever une exception

ou afficher un message d'erreur à l'utilisateur. Cela permet de tester des erreurs de mémoire comme des débordements ou des accès hors limites.

Simulation des erreurs mémoire

- Le simulateur peut générer des erreurs de mémoire en cas d'accès invalide, permettant ainsi de tester la stabilité du programme.
- Il est possible de configurer des scénarios de test pour observer le comportement du système face à des erreurs de mémoire, comme un dépassement de capacité de la pile ou un mauvais accès à la mémoire partagée.

4.3 Débogage et visualisation

Une des fonctionnalités clés du simulateur est son système de débogage. Il permet de suivre l'exécution des instructions et d'observer en temps réel l'état des registres, de la mémoire et des indicateurs de statut.

Détails techniques

- **Pas à pas (Step-by-step)** : Le simulateur peut exécuter les instructions une par une, ce qui permet à l'utilisateur de suivre de près le déroulement du programme. Cela est particulièrement utile pour le débogage, car l'utilisateur peut examiner chaque registre et chaque valeur de mémoire après l'exécution de chaque instruction.
- **Affichage des registres et des indicateurs** : L'état des registres (par exemple, les registres A, B, X, Y, PC, S, U, etc.) ainsi que les indicateurs de statut (Z, C, N, V) sont affichés en temps réel. Cela aide l'utilisateur à comprendre comment chaque instruction modifie l'état du processeur.
- **Suivi de la mémoire** : Le contenu de la RAM et de la ROM peut être visualisé pendant l'exécution du programme, et des modifications peuvent être apportées à la mémoire pour simuler des erreurs ou ajuster des variables.

Modes d'exécution et Débogage du Simulateur

Le simulateur offre plusieurs modes d'exécution pour mieux contrôler l'émulation du programme. Ces modes sont conçus pour aider l'utilisateur à tester et à déboguer les programmes efficacement.

Modes d'exécution

- **Exécution continue** : Le programme s'exécute jusqu'à ce qu'il atteigne un point d'arrêt, une erreur, ou la fin du programme. Ce mode est idéal pour observer l'évolution générale du programme sans interruption.
- **Exécution en boucle** : Ce mode est particulièrement utile pour tester des programmes longs ou des boucles. L'utilisateur peut arrêter l'exécution manuellement à tout moment, ce qui permet de vérifier l'état du programme après chaque itération.
- **Mode débogage interactif** : Le simulateur dispose d'un mode débogage interactif, permettant de modifier dynamiquement les registres ou la mémoire pendant l'exécution. Cela

permet d'observer l'impact des changements en temps réel et d'analyser plus précisément le comportement du programme.

Avantages de ces modes

Ces différents modes d'exécution et de débogage permettent aux utilisateurs de tester leurs programmes dans un environnement sécurisé. Ils facilitent la détection des bugs et des erreurs de logique en offrant une flexibilité maximale pour examiner et manipuler l'état du programme à chaque étape.

Outils de visualisation

- Le simulateur affiche graphiquement les valeurs des registres, des adresses mémoire, et des flags dans une interface utilisateur intuitive. Cette interface permet de suivre le déroulement du programme de manière claire et détaillée.
- Un tableau de mémoire permet de visualiser la mémoire sous forme de tableau, avec des valeurs hexadécimales et des valeurs binaires pour chaque adresse mémoire.

4.4 Contrôle de l'exécution

Le simulateur offre plusieurs modes d'exécution pour mieux contrôler l'émulation du programme. Cela inclut :

- **Exécution continue** : Le programme s'exécute jusqu'à ce qu'il atteigne un point d'arrêt, une erreur, ou la fin du programme.
- **Exécution en boucle** : Idéal pour tester des programmes longs ou des boucles, où l'utilisateur peut arrêter l'exécution manuellement à tout moment.
- **Mode débogage interactif** : Le débogueur permet de modifier dynamiquement les registres ou la mémoire pendant l'exécution pour observer l'impact des changements en temps réel.

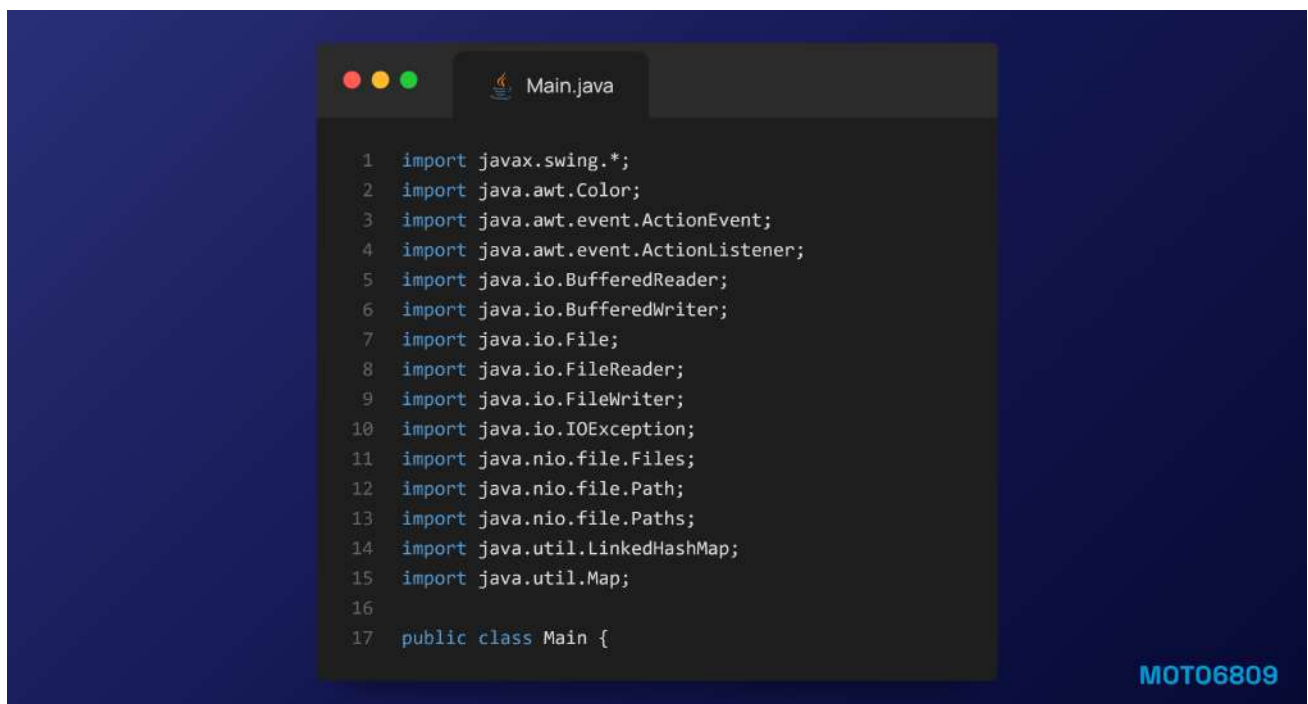
Ces modes d'exécution et de débogage permettent aux utilisateurs de tester des programmes dans un environnement sécurisé et de détecter plus facilement les bugs et erreurs de logique.

5. Mise en œuvre technique

5.1 Analyse des fichiers principaux (Main.Java)

Le fichier **Main.java** joue un rôle central dans le projet de simulation du microprocesseur Motorola 6809. Il sert de point d'entrée principal pour l'application et gère plusieurs aspects clés de la simulation, notamment l'interface utilisateur, la gestion des fichiers, et l'exécution des instructions assembleur. Voici une vue d'ensemble générale de ce fichier et de son rôle principal dans le projet :

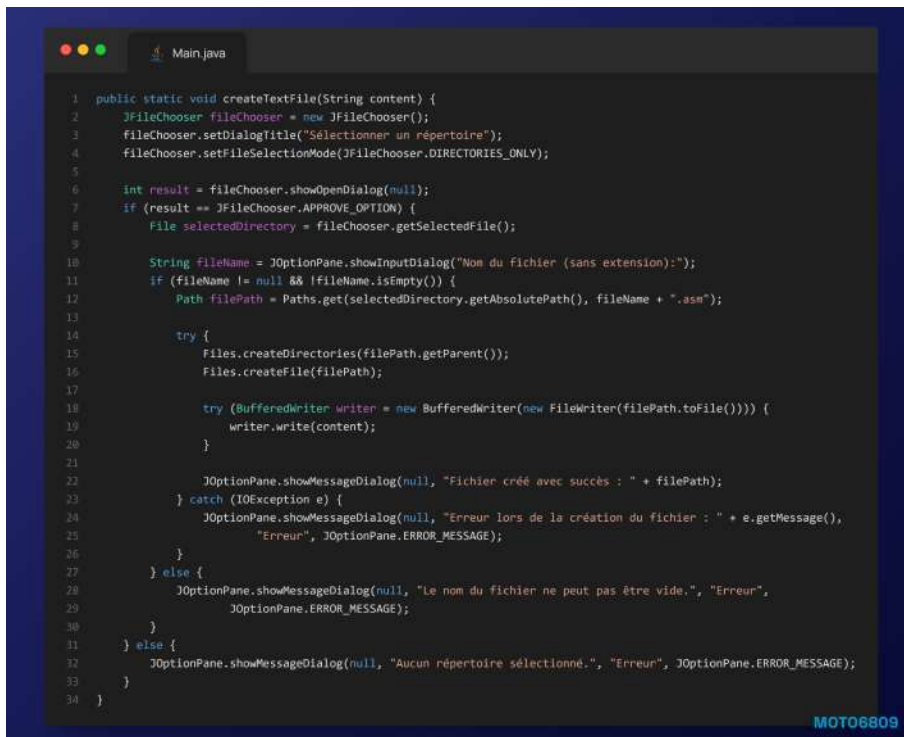
5.1.1 Importations et Déclarations de Classes



5.1.2 Méthodes de Gestion de Fichiers

Création de Fichiers Texte :

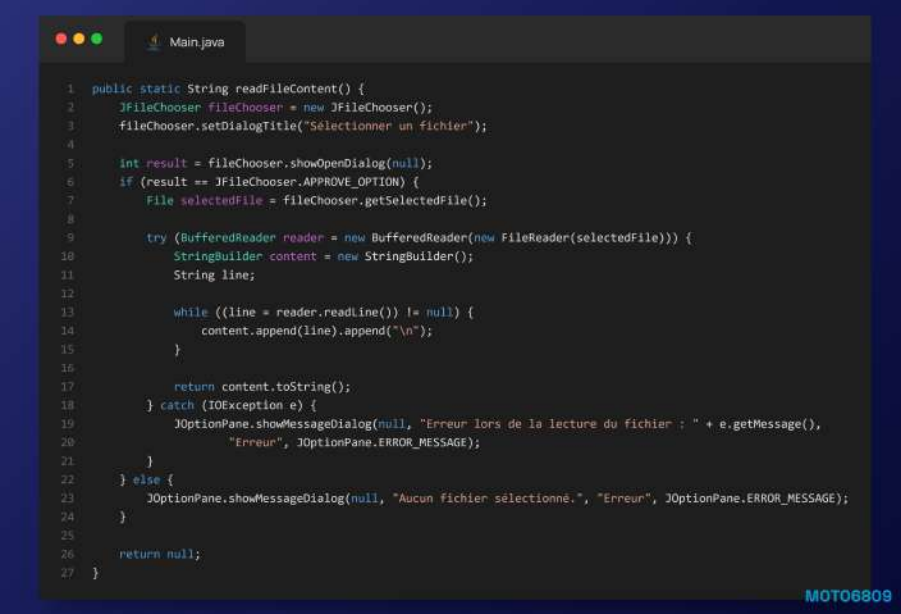
Ce segment de code contient la méthode `createTextFile`, qui permet à l'utilisateur de créer un fichier texte avec un contenu spécifique. La méthode utilise des composants graphiques de Swing pour interagir avec l'utilisateur et des classes de gestion de fichiers de Java pour créer et écrire dans le fichier. La méthode commence par initialiser un `JFileChooser` pour permettre à l'utilisateur de sélectionner un répertoire. Si l'utilisateur sélectionne un répertoire, une boîte de dialogue d'entrée demande le nom du fichier (sans extension). Si un nom de fichier valide est fourni, le chemin complet du fichier est construit en ajoutant l'extension `.asm`. Ensuite, la méthode tente de créer les répertoires nécessaires et le fichier spécifié. Si la création du fichier réussit, le contenu fourni est écrit dans le fichier et un message de confirmation est affiché. En cas d'erreur, un message d'erreur approprié est affiché. Si l'utilisateur ne fournit pas de nom de fichier valide ou annule la sélection de répertoire, des messages d'erreur correspondants sont affichés pour informer l'utilisateur.

A screenshot of a Java IDE window titled 'Main.java'. The code defines a static method `createTextFile(String content)`. It uses `JFileChooser` to select a directory, then `JOptionPane` to get a filename. It constructs a file path with the '.asm' extension, creates the directory and file, and writes the content using `BufferedWriter`. Error handling is implemented with `catch` blocks and `JOptionPane.showMessageDialog` calls. The code is numbered from 1 to 34. A 'MOTO6809' watermark is visible in the bottom right corner of the code editor.

```
1 public static void createTextFile(String content) {
2     JFileChooser fileChooser = new JFileChooser();
3     fileChooser.setDialogTitle("Sélectionner un répertoire");
4     fileChooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
5
6     int result = fileChooser.showOpenDialog(null);
7     if (result == JFileChooser.APPROVE_OPTION) {
8         File selectedDirectory = fileChooser.getSelectedFile();
9
10        String fileName = JOptionPane.showInputDialog("Nom du fichier (sans extension):");
11        if (fileName != null && !fileName.isEmpty()) {
12            Path filePath = Paths.get(selectedDirectory.getAbsolutePath(), fileName + ".asm");
13
14            try {
15                Files.createDirectories(filePath.getParent());
16                Files.createFile(filePath);
17
18                try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath.toFile()))) {
19                    writer.write(content);
20                }
21
22                JOptionPane.showMessageDialog(null, "Fichier créé avec succès : " + filePath);
23            } catch (IOException e) {
24                JOptionPane.showMessageDialog(null, "Erreur lors de la création du fichier : " + e.getMessage(),
25                    "Erreur", JOptionPane.ERROR_MESSAGE);
26            }
27        } else {
28            JOptionPane.showMessageDialog(null, "Le nom du fichier ne peut pas être vide.", "Erreur",
29                JOptionPane.ERROR_MESSAGE);
30        }
31    } else {
32        JOptionPane.showMessageDialog(null, "Aucun répertoire sélectionné.", "Erreur", JOptionPane.ERROR_MESSAGE);
33    }
34 }
```

Lecture de Fichiers Texte :

Ce segment de code définit une méthode statique `readFileContent` qui permet à l'utilisateur de sélectionner un fichier via une interface graphique et de lire son contenu. La méthode utilise `JFileChooser` pour ouvrir une boîte de dialogue de sélection de fichier. Si l'utilisateur sélectionne un fichier, le code ouvre ce fichier en mode lecture à l'aide de `BufferedReader` et lit son contenu ligne par ligne, en ajoutant chaque ligne à un `StringBuilder`. Si la lecture du fichier réussit, le contenu complet du fichier est retourné sous forme de chaîne de caractères. En cas d'erreur lors de la lecture du fichier ou si aucun fichier n'est sélectionné, un message d'erreur approprié est affiché à l'utilisateur via `JOptionPane`.

A screenshot of a Java IDE window titled 'Main.java'. The code is written in Java and implements a static method 'readFileContent()' that prompts the user to select a file and reads its content. The code is as follows:

```
1 public static String readFileContent() {
2     JFileChooser fileChooser = new JFileChooser();
3     fileChooser.setDialogTitle("Sélectionner un fichier");
4
5     int result = fileChooser.showOpenDialog(null);
6     if (result == JFileChooser.APPROVE_OPTION) {
7         File selectedFile = fileChooser.getSelectedFile();
8
9         try (BufferedReader reader = new BufferedReader(new FileReader(selectedFile))) {
10             StringBuilder content = new StringBuilder();
11             String line;
12
13             while ((line = reader.readLine()) != null) {
14                 content.append(line).append("\n");
15             }
16
17             return content.toString();
18         } catch (IOException e) {
19             JOptionPane.showMessageDialog(null, "Erreur lors de la lecture du fichier : " + e.getMessage(),
20                 "Erreur", JOptionPane.ERROR_MESSAGE);
21         }
22     } else {
23         JOptionPane.showMessageDialog(null, "Aucun fichier sélectionné.", "Erreur", JOptionPane.ERROR_MESSAGE);
24     }
25
26     return null;
27 }
```

The IDE has a dark theme. In the bottom right corner of the code editor, the text 'MOTO6809' is visible.

5.1.3 Méthode Main

Initialisation des Composants :

Ce segment de code représente la méthode main d'un programme Java qui initialise et configure divers composants d'une simulation ou d'une interface utilisateur graphique (GUI) pour un système informatique.

```
1 public static void main(String[] args) {
2     ComponentsPanel componentsPanel = new ComponentsPanel();
3     Color color = new Color(232, 20, 5);
4
5     LinkedHashMap<String, String> ramData = new LinkedHashMap<>();
6     ramData.put("FC00", "");
7     ramData.put("FC01", "");
8     ramData.put("FC02", "");
9     ramData.put("FC03", "");
10    ramData.put("FC04", "");
11    ramData.put("FC05", "");
12    RAM ram = new RAM(150, 230, 535, 30, ramData, color);
13
14    LinkedHashMap<String, String> romData = new LinkedHashMap<>();
15    romData.put("0001", "");
16    romData.put("0002", "");
17    romData.put("0003", "");
18    romData.put("0005", "");
19    romData.put("0006", "");
20    romData.put("0007", "");
21    ROM rom = new ROM(150, 230, 700, 30, romData, color);
22
23    Registre registreA = new Registre("A", "00", 120, 30, 130, 300, color);
24    Registre registreB = new Registre("B", "00", 120, 30, 260, 300, color);
25    Registre registreX = new Registre("X", "0000", 250, 30, 130, 340, color);
26    Registre registreY = new Registre("Y", "0000", 250, 30, 130, 380, color);
27    Registre registreU = new Registre("U", "0000", 250, 30, 130, 420, color);
28    Registre registreS = new Registre("S", "0000", 250, 30, 130, 460, color);
29    Registre registrePC = new Registre("PC", "FC00", 250, 30, 130, 500, color);
30    Registre registreDP = new Registre("DP", "00", 150, 30, 70, 200, color);
31
32    Registre flagN = new Registre("N", "0", 120, 30, 750, 330, color);
33    Registre flagZ = new Registre("Z", "1", 120, 30, 750, 370, color);
34    Registre flagV = new Registre("V", "0", 120, 30, 750, 410, color);
35    Registre flagC = new Registre("C", "0", 120, 30, 750, 450, color);
36    Registre flagH = new Registre("H", "0", 120, 30, 750, 490, color);
37
38    Controller registreI = new Controller("Instruction :", "", 250, 50, 30, 10, color);
39    Controller decodeur = new Controller("Décodeur", "", 250, 30, 10, 70, color);
40    Controller sequenceur = new Controller("Contrôleur séquenceur", "", 250, 30, 10, 110, color);
41
42    Binary binA = new Binary("00000000", 60, 12, registreA.x + 60, registreA.y - 12);
43    Binary binB = new Binary("00000000", 60, 12, registreB.x + 60, registreB.y - 12);
44
45    UAL ual = new UAL(138, 205, 530, 342, "", "", "", color, Color.WHITE);
```

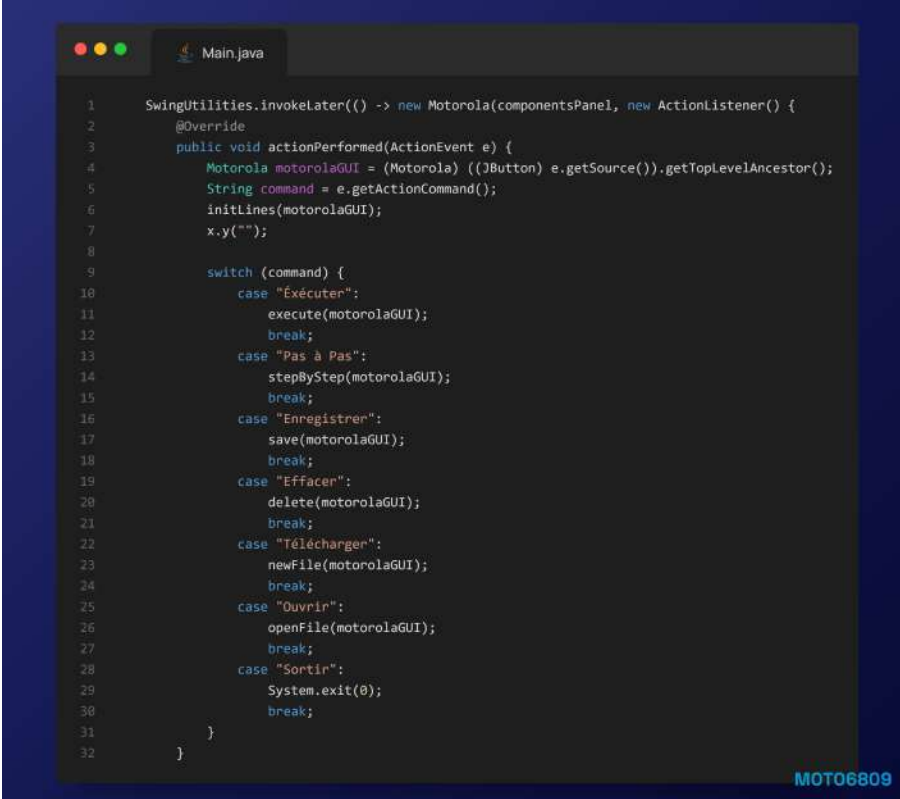
Ajout des Composants au Panneau :

ce segment de code ajoute tous les composants initialisés (RAM, ROM, registres, drapeaux, contrôleurs, binaires et UAL) au panneau componentsPanel en appelant leur méthode spawn(). Cela permet de les afficher et de les gérer dans l'interface graphique du programme.

```
1 componentsPanel.add(ram.spawn());
2 componentsPanel.add(rom.spawn());
3 componentsPanel.add(registreA.spawn());
4 componentsPanel.add(registreB.spawn());
5 componentsPanel.add(registreX.spawn());
6 componentsPanel.add(registreY.spawn());
7 componentsPanel.add(registreU.spawn());
8 componentsPanel.add(registreS.spawn());
9 componentsPanel.add(registrePC.spawn());
10 componentsPanel.add(registreDP.spawn());
11 componentsPanel.add(flagN.spawn());
12 componentsPanel.add(flagZ.spawn());
13 componentsPanel.add(flagV.spawn());
14 componentsPanel.add(flagC.spawn());
15 componentsPanel.add(flagH.spawn());
16 componentsPanel.add(registreI.spawn());
17 componentsPanel.add(decodeur.spawn());
18 componentsPanel.add(sequenceur.spawn());
19 componentsPanel.add(binA.spawn());
20 componentsPanel.add(binB.spawn());
21 componentsPanel.add(ual.spawn());
```

Initialisation de l'Interface Graphique :

ce segment de code initialise l'interface graphique en utilisant Swing et configure un ActionListener pour gérer les événements des boutons. Chaque bouton déclenche une action spécifique en fonction de sa commande, permettant ainsi de contrôler différentes fonctionnalités de l'application.



```
1 SwingUtilities.invokeLater(() -> new Motorola(componentsPanel, new ActionListener() {
2     @Override
3     public void actionPerformed(ActionEvent e) {
4         Motorola motorolaGUI = (Motorola) ((JButton) e.getSource()).getTopLevelAncestor();
5         String command = e.getActionCommand();
6         initLines(motorolaGUI);
7         x.y("");
8
9         switch (command) {
10             case "Exécuter":
11                 execute(motorolaGUI);
12                 break;
13             case "Pas à Pas":
14                 stepByStep(motorolaGUI);
15                 break;
16             case "Enregistrer":
17                 save(motorolaGUI);
18                 break;
19             case "Effacer":
20                 delete(motorolaGUI);
21                 break;
22             case "Télécharger":
23                 newFile(motorolaGUI);
24                 break;
25             case "Ouvrir":
26                 openFile(motorolaGUI);
27                 break;
28             case "Sortir":
29                 System.exit(0);
30                 break;
31         }
32     }
33 })
```

MOTO6809

5.1.4 Méthodes de Gestion des Instructions

Initialisation des Lignes :

cette méthode `initLines` traite le texte de l'éditeur en supprimant les commentaires, en convertissant le texte en majuscules, en mappant les étiquettes à leurs indices de ligne, et en supprimant les étiquettes des lignes. Cela prépare le texte pour une analyse ou une exécution ultérieure dans le contexte de l'application.

```

1      String[] lines;
2      boolean saved = false;
3      int index = 0, pointeur = 0;
4      LinkedHashMap<String, Integer> tikitat = new LinkedHashMap<>();
5
6      public void initLines(Motorola gui) {
7          lines = gui.textEditor.getText().replaceAll(".*?\n", "\n").toUpperCase().split("\\n+");
8          for(int i=0; i<lines.length; i++)
9              if(lines[i].contains(":"))
10                 tikitat.put(lines[i].split(":")[0], i);
11
12          for(int i=0; i<lines.length; i++) {
13              lines[i] = lines[i].contains(":") ? "" : lines[i];
14          }
15      }

```

Exécution des Instructions :

Ce segment de code définit deux méthodes, `execute` et `stepByStep`, qui sont utilisées pour exécuter des instructions dans une application graphique basée sur Swing. La méthode `execute` exécute toutes les instructions en appelant `stepByStep` pour chaque ligne. La méthode `stepByStep` exécute une instruction à la fois, en vérifiant d'abord si le texte a été sauvegardé, puis en traitant l'instruction courante, en mettant à jour les registres et la RAM, et en ajustant les indices pour passer à l'instruction suivante.

```

1      private String getRegistreVal(String reg) {
2          if (reg == null) {
3              return "";
4          }
5
6          switch (reg) {
7              case "A":
8                  return (registreA != null && registreA.vallabel != null) ? registreA.vallabel.getText() : "";
9              case "B":
10                 return (registreB != null && registreB.vallabel != null) ? registreB.vallabel.getText() : "";
11              case "X":
12                 return (registreX != null && registreX.vallabel != null) ? registreX.vallabel.getText() : "";
13              case "Y":
14                 return (registreY != null && registreY.vallabel != null) ? registreY.vallabel.getText() : "";
15              case "U":
16                 return (registreU != null && registreU.vallabel != null) ? registreU.vallabel.getText() : "";
17              case "S":
18                 return (registreS != null && registreS.vallabel != null) ? registreS.vallabel.getText() : "";
19              case "DP":
20                 return (registreDP != null && registreDP.vallabel != null) ? registreDP.vallabel.getText() : "";
21              default:
22                 return "";
23          }
24      }
25
26      private void setRegistreVal(String reg, String val) {
27          String binVal = String.format("%8s", Integer.toBinaryString(Integer.parseInt(val, 16))).replace(' ', '0');
28          val = val + "00000000000000000000000000000000";
29          if (reg.equals("A")) {
30              registreA.vallabel.setText(val.substring(0, 2));
31              binA.vallabel.setText(binVal);
32          }
33          if (reg.equals("B")) {
34              registreB.vallabel.setText(val.substring(0, 2));
35              binB.vallabel.setText(binVal);
36          }
37          if (reg.equals("U")) {
38              registreU.vallabel.setText(val.substring(0, 4));
39          }
40          if (reg.equals("S")) {
41              registreS.vallabel.setText(val.substring(0, 4));
42          }
43          if (reg.equals("X")) {
44              registreX.vallabel.setText(val.substring(0, 4));
45          }
46          if (reg.equals("Y")) {
47              registreY.vallabel.setText(val.substring(0, 4));
48          }
49          if (reg.equals("DP")) {
50              registreDP.vallabel.setText(val.substring(0, 2));
51          }
52      }

```


Sauvegarde des Instructions

La méthode `save` est responsable de la validation et de la sauvegarde des instructions dans la mémoire RAM de l'application. Elle commence par vérifier la syntaxe de chaque ligne de texte. Si une ligne contient une erreur de syntaxe, un message d'erreur est affiché et la méthode se termine. Si toutes les lignes sont valides, la méthode met à jour l'état de sauvegarde, active les boutons d'exécution et de pas à pas, et prépare les instructions pour être stockées en mémoire.

Ensuite, la méthode parcourt chaque ligne de texte, ignore les lignes vides, et traite chaque instruction. Pour chaque instruction, elle extrait l'opcode et l'opérande, puis génère le code correspondant. Si l'opérande contient des caractères spéciaux (comme `#`, `$`, ou `,`), il est filtré et traité en conséquence. Les instructions sont ensuite mappées à des adresses mémoire spécifiques.

Finalement, la méthode met à jour la mémoire RAM avec les instructions mappées et ajuste le registre PC (Program Counter) pour refléter l'adresse de la prochaine instruction à exécuter. Cette méthode assure que les instructions sont correctement validées, traitées et stockées en mémoire, prêtes pour l'exécution.

```
1 private void save(Motorola gui) {
2     for (int i = 0; i < lines.length; i++) {
3         String line = lines[i].trim();
4         String[] parts = line.split("\\s+");
5         String opcode = parts[0];
6         String operand = parts.length > 1 ? parts[1] : null;
7         Instruction instr = new Instruction(opcode, operand);
8         if (operand != null && !Instruction.isValidSyntax(instr)) {
9             JOptionPane.showMessageDialog(null,
10                "Mauvaise Syntaxe : (dans la ligne : " + (i + 1) + " " + Instruction.errMsg, "Erreur",
11                JOptionPane.ERROR_MESSAGE);
12             return;
13         }
14     }
15
16     saved = true;
17     gui.saveButton.setText("Effacer");
18     gui.stepButton.setEnabled(true);
19     gui.executeButton.setEnabled(true);
20     int k = 0;
21     LinkedHashMap<String, String> map = new LinkedHashMap();
22
23     for (int i = 0; i < lines.length; i++) {
24         if (lines[i].equals(""))
25             continue;
26         String line = lines[i].trim();
27         String[] parts = line.split("\\s+");
28         String opcode = parts[0];
29         String operand = parts.length > 1 ? parts[1] : null;
30         map.put(Instruction.getAddress(k++), Instruction.getCode(new Instruction(opcode, operand)));
31         operand = String.format("%s", operand);
32         if (operand != null && (operand.contains("#") || operand.contains("$") || operand.contains(","))) {
33             operand = Instruction.getFilter(operand);
34             if (operand.length() > 2) {
35                 if (operand.contains(",") {
36                     if (operand.equals("TFR") || operand.equals("EXG")) {
37                         // cas de TFR et EXG
38                         map.put(Instruction.getAddress(k++), "89");
39                     }
40                     else {
41                         // cas de L'indexé
42                     }
43                 } else {
44                     map.put(Instruction.getAddress(k++), operand.substring(0, 2));
45                     map.put(Instruction.getAddress(k++), operand.substring(2, 4));
46                 }
47             }
48             if (operand.length() <= 2) {
49                 map.put(Instruction.getAddress(k++), operand);
50             }
51         }
52     }
53     ram.setRAM(map);
54     registrePC.valLabel.setText(Instruction.getAddress(index));
55 }
```

MOTO6809

Suppression des Instructions

Ce segment de code définit plusieurs méthodes pour gérer différentes actions dans une application graphique basée sur Swing, ainsi qu'une méthode pour exécuter des instructions spécifiques.

```
1 private void save(Motorola gui) {
2     for (int i = 0; i < lines.length; i++) {
3         String line = lines[i].trim();
4         String[] parts = line.split("\\s+");
5         String opcode = parts[0];
6         String operand = parts.length > 1 ? parts[1] : null;
7         Instruction instr = new Instruction(opcode, operand);
8         if (operand != null && !Instruction.goodSyntax(instr)) {
9             JOptionPane.showMessageDialog(null,
10                "Mauvaise Syntaxe : (dans la ligne : " + (i + 1) + " " + "\n" + Instruction.errMsg, "Erreur",
11                JOptionPane.ERROR_MESSAGE);
12             return;
13         }
14     }
15     private void delete(Motorola gui) {
16         saved = false;
17         index = pointeur = 0;
18         gui.saveButton.setText("Enregistrer");
19         gui.stepButton.setEnabled(false);
20         gui.executeButton.setEnabled(false);
21         gui.textEditor.setText("");
22         ram.setRAM(ramData);
23         rom.resetROM();
24         ual.setUAL("", "", "");
25         registreA.vallabel.setText("00");
26         registreB.vallabel.setText("00");
27         registreU.vallabel.setText("0000");
28         registreS.vallabel.setText("0000");
29         registreX.vallabel.setText("0000");
30         registreY.vallabel.setText("0000");
31         registreI.vallabel.setText("");
32         registrePC.vallabel.setText("");
33         flagC.vallabel.setText("0");
34         flagH.vallabel.setText("0");
35         flagZ.vallabel.setText("1");
36         flagH.vallabel.setText("0");
37         flagV.vallabel.setText("0");
38         return;
39     }
40     saved = true;
41     gui.saveButton.setText("Effacer");
42     gui.stepButton.setEnabled(true);
43     gui.executeButton.setEnabled(true);
44     int k = 0;
45     LinkedHashMap<String, String> map = new LinkedHashMap();
46
47     for (int i = 0; i < lines.length; i++) {
48         if (lines[i].equals(""))
49             continue;
50         String line = lines[i].trim();
51         String[] parts = line.split("\\s+");
52         String opcode = parts[0];
53         String operand = parts.length > 1 ? parts[1] : null;
54         map.put(Instruction.getAddress(k++), Instruction.getCode(new Instruction(opcode, operand)));
55         operand = String.format("%s", operand);
56         if (operand != null && (operand.contains("#") || operand.contains("$") || operand.contains(","))) {
57             operand = Instruction.getFilter(operand);
58             if (operand.length() > 2) {
59                 if (operand.contains(",") {
60                     if (operand.equals("TFR") || operand.equals("EXG")) {
61                         // cas de TFR et EXG
62                         map.put(Instruction.getAddress(k++), "89");
63                     }
64                     else {
65                         // cas de L'indexé
66                     }
67                 } else {
68                     map.put(Instruction.getAddress(k++), operand.substring(0, 2));
69                     map.put(Instruction.getAddress(k++), operand.substring(2, 4));
70                 }
71             }
72             if (operand.length() <= 2) {
73                 map.put(Instruction.getAddress(k++), operand);
74             }
75         }
76     }
77     ram.setRAM(map);
78     registrePC.vallabel.setText(Instruction.getAddress(index));
79 }
```

MOTO6809

Création et Ouverture de Fichiers

La méthode `newFile` permet de créer un nouveau fichier texte en utilisant le contenu actuel de l'éditeur de texte. Elle appelle la méthode `createTextFile` pour effectuer cette opération. La méthode `openFile` permet d'ouvrir un fichier texte existant et de charger son contenu dans l'éditeur de texte. Elle appelle la méthode `readFileContent` pour lire le contenu du fichier et le charger dans l'éditeur de texte.



5.1.5 Méthodes Utilitaires

Offset Shifter

La méthode `offsetshifter` prend en entrée une chaîne de caractères représentant un opérande au format *value, register* ou *value, register*. Elle divise cette chaîne en deux parties : le décalage (*offset*) et le registre (*register*) ou en décimal. La méthode convertit ensuite le décalage en un entier et récupère la valeur du registre spécifié en utilisant la méthode `getRegisterVal`.

La méthode vérifie que le registre est l'un des registres valides (X, Y, U, S) et convertit la valeur du registre en un entier. Ensuite, elle additionne le décalage et la valeur du registre pour obtenir une adresse finale. Cette adresse est convertie en une chaîne hexadécimale de 4 caractères, en ajoutant des zéros de remplissage si nécessaire.

Si le format de l'opérande est incorrect ou si une exception de format de nombre se produit, la méthode lève une exception `IllegalArgumentException` avec un message d'erreur approprié.

```

1  public String offsetshifter(String Operand) {
2      // Split the operand into two parts
3      String[] Split = Operand.split(",");
4      if (Split.length != 2) {
5          throw new IllegalArgumentException(
6              "Operand must be in the format '$value,register' or 'value,register'.");
7      }
8
9      try {
10         // Parse the first part (the offset in hex or decimal)
11         String firstPart = Split[0].trim();
12         int value1 = 0;
13
14         if (firstPart.startsWith("$")) {
15             // Convert from hex string to integer
16             value1 = Integer.parseInt(firstPart.substring(1), 16); // Remove the "$" and parse as hex
17         } else {
18             // Parse as a decimal number
19             value1 = Integer.parseInt(firstPart); // Parse directly as decimal
20         }
21
22         // Parse the second part (register offset as string)
23         String register = Split[1].trim();
24         String registerValue = getRegistreVal(Split[1]); // Default string value for the register
25
26         switch (register) {
27             case "X":
28                 registerValue = getRegistreVal("X"); // Default value for X
29                 break;
30             case "Y":
31                 registerValue = getRegistreVal("Y"); // Default value for Y
32                 break;
33             case "U":
34                 registerValue = getRegistreVal("U"); // Default value for U
35                 break;
36             case "S":
37                 registerValue = getRegistreVal("S"); // Default value for S
38                 break;
39             default:
40                 throw new IllegalArgumentException("Invalid register: " + register);
41         }
42
43         // Convert registerValue string to integer for addition
44         int registerInt = Integer.parseInt(registerValue, 16);
45
46         // Add the two values
47         int result = value1 + registerInt;
48
49         // Convert the result to a 4-digit hex string
50         String hexResult = Integer.toHexString(result).toUpperCase();
51
52         // Ensure the result is padded to 4 characters
53         while (hexResult.length() < 4) {
54             hexResult = "0" + hexResult;
55         }
56
57         return hexResult; // Return the final 16-bit hex address
58     } catch (NumberFormatException e) {
59         throw new IllegalArgumentException("Invalid number format in operand: " + Operand, e);
60     }
61 }

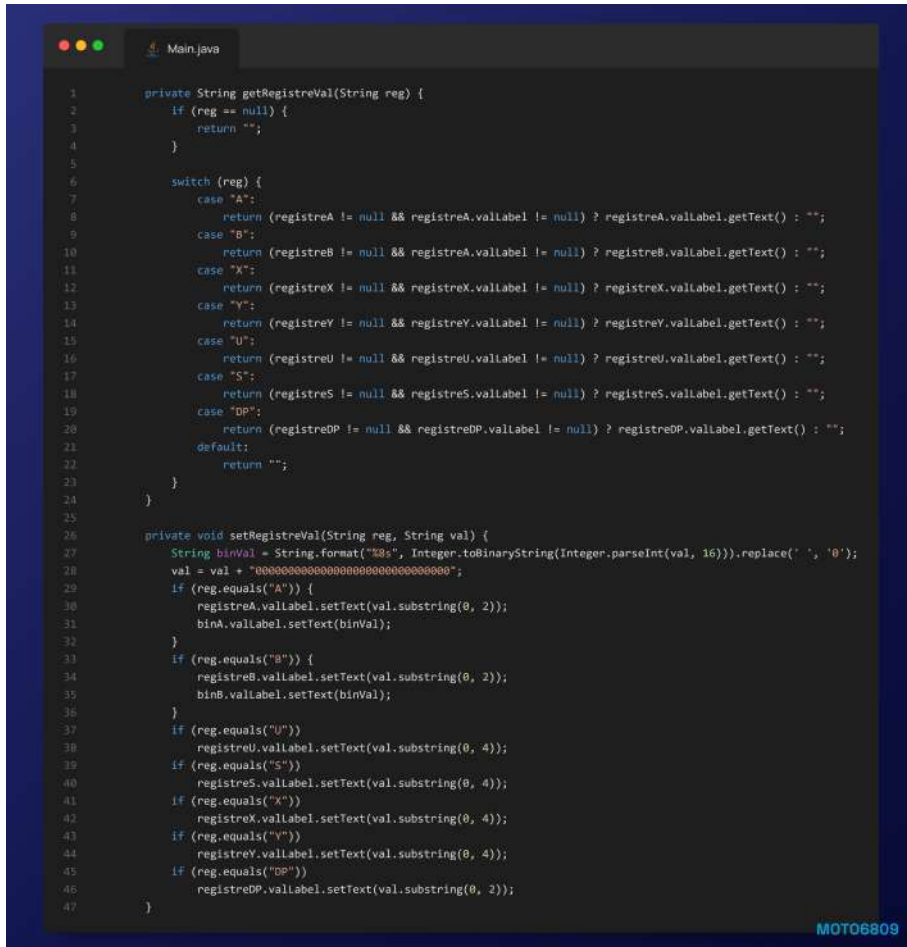
```

MOT06809

Get and Set Registre Values

définir les valeurs des registres, ce qui est crucial pour le fonctionnement correct du simulateur.

La méthode `getRegistreVal` prend en entrée le nom d'un registre (sous forme de chaîne de caractères) et retourne la valeur actuelle de ce registre. Elle utilise une structure de contrôle `switch` pour vérifier quel registre est demandé et retourne la valeur correspondante. Si le registre ou son étiquette de valeur (`valLabel`) est nul, la méthode retourne une chaîne vide. Cette méthode est utilisée pour récupérer les valeurs des registres A, B, X, Y, U, S, et DP, qui sont des composants essentiels du microprocesseur.



```

1 private String getRegistreVal(String reg) {
2     if (reg == null) {
3         return "";
4     }
5
6     switch (reg) {
7         case "A":
8             return (registreA != null && registreA.vallabel != null) ? registreA.vallabel.getText() : "";
9         case "B":
10            return (registreB != null && registreB.vallabel != null) ? registreB.vallabel.getText() : "";
11        case "X":
12            return (registreX != null && registreX.vallabel != null) ? registreX.vallabel.getText() : "";
13        case "Y":
14            return (registreY != null && registreY.vallabel != null) ? registreY.vallabel.getText() : "";
15        case "U":
16            return (registreU != null && registreU.vallabel != null) ? registreU.vallabel.getText() : "";
17        case "S":
18            return (registreS != null && registreS.vallabel != null) ? registreS.vallabel.getText() : "";
19        case "DP":
20            return (registreDP != null && registreDP.vallabel != null) ? registreDP.vallabel.getText() : "";
21        default:
22            return "";
23    }
24 }
25
26 private void setRegistreVal(String reg, String val) {
27     String binVal = String.format("%32s", Integer.toBinaryString(Integer.parseInt(val, 16))).replace(' ', '0');
28     val = val + "00000000000000000000000000000000";
29     if (reg.equals("A")) {
30         registreA.vallabel.setText(val.substring(0, 2));
31         binA.vallabel.setText(binVal);
32     }
33     if (reg.equals("B")) {
34         registreB.vallabel.setText(val.substring(0, 2));
35         binB.vallabel.setText(binVal);
36     }
37     if (reg.equals("U")) {
38         registreU.vallabel.setText(val.substring(0, 4));
39     }
40     if (reg.equals("S")) {
41         registreS.vallabel.setText(val.substring(0, 4));
42     }
43     if (reg.equals("X")) {
44         registreX.vallabel.setText(val.substring(0, 4));
45     }
46     if (reg.equals("Y")) {
47         registreY.vallabel.setText(val.substring(0, 4));
48     }
49     if (reg.equals("DP")) {
50         registreDP.vallabel.setText(val.substring(0, 2));
51     }
52 }

```

5.1.6 Exécution des Instructions Spécifiques

Exécution d'une Instruction

Ces méthodes sont conçues pour exécuter différentes instructions d'un processeur simulé, chacune effectuant une opération spécifique en fonction de l'opcode de l'instruction.

NB : Dans le fichier Main.java, toutes les méthodes des instructions sont présentes, mais nous avons implémenté dans l'image uniquement deux d'entre elles à des fins d'explication, car elles reposent sur le même principe et la même logique.

```

Main.java
1      public void executeOne(Instruction instr) {
2          String str = instr.opcode.toUpperCase();
3          if (str.startsWith("LD"))
4              executeLD(instr);
5          if (str.startsWith("ADD"))
6              executeADD(instr);
7          if (str.startsWith("SUB"))
8              executeSUB(instr);
9          if (str.startsWith("ST"))
10             executeST(instr);
11         if (str.startsWith("INC"))
12             executeINC(instr);
13         if (str.startsWith("DEC"))
14             executeDEC(instr);
15         if (str.startsWith("LSL"))
16             executeLSL(instr);
17         if (str.startsWith("LSR"))
18             executeLSR(instr);
19         if (str.startsWith("ROL"))
20             executeROL(instr);
21         if (str.startsWith("ROR"))
22             executeROR(instr);
23         if (str.startsWith("NOP"))
24             executeNOP(instr);
25         if (str.startsWith("CLR"))
26             executeCLR(instr);
27         if (str.startsWith("SWI"))
28             executeSWI(instr);
29         if (str.startsWith("END"))
30             executeEND(instr);
31         if (str.startsWith("COM"))
32             executeCOM(instr);
33         if (str.startsWith("CMP"))
34             executeCMP(instr);
35         if (str.startsWith("EXG"))
36             executeEXG(instr);
37         if (str.startsWith("TFR"))
38             executeTFR(instr);
39         if (str.startsWith("JMP"))
40             executeJMP(instr);
41         if (str.startsWith("RTS"))
42             executeRTS(instr);
43         if (str.startsWith("BEQ"))
44             executeBEQ(instr);
45
46         if (str.startsWith("BEQ"))
47             executeBEQ(instr);
48         if (str.startsWith("BNE"))
49             executeBNE(instr);
50         if (str.startsWith("BMI"))
51             executeBMI(instr);
52         if (str.startsWith("BPL"))
53             executeBPL(instr);
54         if (str.startsWith("BCC"))
55             executeBCC(instr);
56         if (str.startsWith("BCS"))
57             executeBCS(instr);
58         if (str.startsWith("BVC"))
59             executeBVC(instr);
60         if (str.startsWith("BVS"))
61             executeBVS(instr);
62         if (str.startsWith("BRA"))
63             executeBRA(instr);
64     }

```

MOTO6809

Exécution des Instructions Spécifiques



```
1 public void executeCMP(Instruction instr) {
2     String reg = instr.opcode.substring(3);
3     String value = getRegistreVal(reg);
4     String operand = instr.operand;
5     String mode = Instruction.detectMode(instr);
6     int currentValue = Integer.parseInt(value, 16);
7     switch (mode) {
8         case "immediat":
9             value = operand.replace("#", "").replace("$", "");
10            break;
11        case "direct":
12            String adresseDirecte = operand.replace("$", "");
13            value = rom.map.getOrDefault(registreOP.vallabel.getText() + adresseDirecte, "00");
14            break;
15        case "etendu":
16            String adresseEtendue = operand.replace("$", "");
17            value = rom.map.getOrDefault(adresseEtendue, "00");
18            break;
19        default:
20            break;
21    }
22
23    int Comp = Integer.parseInt(value, 16);
24    if (currentValue - Comp == 0) {
25        flagN.vallabel.setText((currentValue - Comp) < 0 ? "1" : "0");
26        flagZ.vallabel.setText((currentValue - Comp) == 0 ? "1" : "0");
27    }
28    if (currentValue - Comp < 0) {
29        flagN.vallabel.setText((currentValue - Comp) < 0 ? "1" : "0");
30        flagZ.vallabel.setText((currentValue - Comp) == 0 ? "1" : "0");
31    }
32    if (currentValue - Comp == 0) {
33        flagN.vallabel.setText((currentValue - Comp) < 0 ? "1" : "0");
34        flagZ.vallabel.setText((currentValue - Comp) == 0 ? "1" : "0");
35    }
36    }
37
38    public void executeD(Instruction instr) {
39        String reg = instr.opcode.substring(2);
40        String operand = instr.operand;
41        String mode = Instruction.detectMode(instr);
42        String value = "";
43
44        switch (mode) {
45            case "immediat":
46                value = operand.startsWith("#$") ? operand.substring(2) : operand.substring(1);
47                break;
48
49            case "direct":
50                String adresseDirecte = operand.replace("$", "");
51                value = rom.map.getOrDefault(adresseDirecte + registreOP.vallabel.getText(), adresseDirecte + "00");
52                break;
53
54            case "etendu":
55                String adresseEtendue = operand.replace("$", "");
56                value = rom.map.getOrDefault(adresseEtendue, "00");
57                break;
58
59            case "indexe":
60                value = rom.map.getOrDefault(offsetshifter(operand), "00");
61                break;
62
63            default:
64                throw new IllegalArgumentException("Mode d'adressage inconnu : " + mode);
65        }
66
67        if (value == null || value.isEmpty()) {
68            throw new IllegalArgumentException("Valeur introuvable pour l'adresse ou l'opérande : " + operand);
69        }
70
71        setRegistreVal(reg, value);
72    }
```

5.2 Analyse des fichiers principaux (Instruction.java)

Le fichier Instruction.java représente et manipule les instructions du processeur Motorola 6809. Il contient des méthodes pour valider, détecter le mode d'adressage, et générer le code machine pour chaque instruction. La classe déclare des variables pour l'opcode, l'opérande, le mode d'adressage, la taille de l'instruction, et les messages d'erreur. Le constructeur initialise l'opcode et l'opérande. La classe inclut des listes de codes et de labels valides. La méthode detectMode détermine le mode d'adressage, goodSyntaxe valide la syntaxe, operandSizeValide vérifie la taille de l'opérande, isHexadecimal vérifie le format hexadécimal, operandValide vérifie

l'opérande pour l'opcode, getAddress génère une adresse, getCode retourne le code hexadécimal, et getFilter filtre l'opérande. Ces méthodes permettent de valider, analyser et générer le code machine pour les instructions du processeur.

5.2.1 Déclaration de la Classe et des Variables

Déclare les variables membres pour stocker l'opcode, l'opérande, le mode d'adressage, la taille de l'instruction et les messages d'erreur.

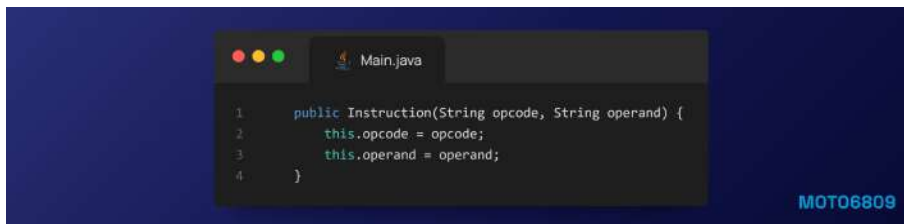


```
1 public class Instruction {
2     public String opcode;
3     public String operand;
4     public String mode;
5     public String size;
6     public static String errMsg = "";
```

MOTO6809

5.2.2 Constructeur de la Classe

Initialise les variables opcode et operand avec les valeurs passées en paramètres.

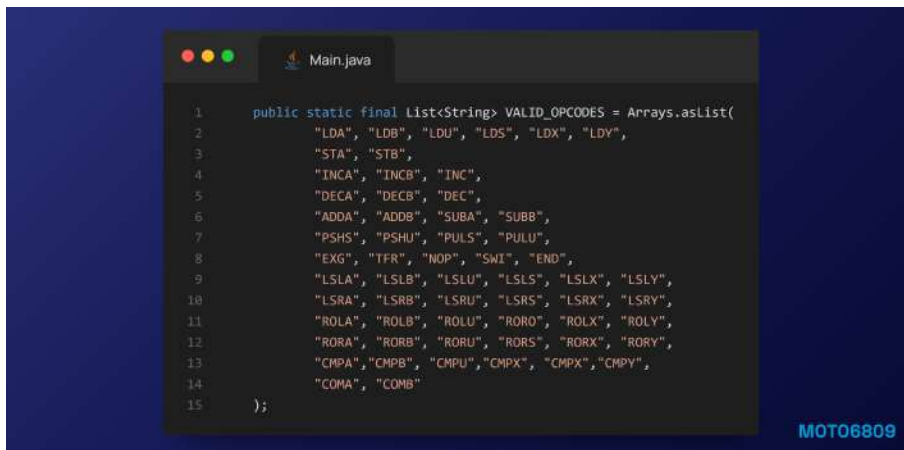


```
1 public Instruction(String opcode, String operand) {
2     this.opcode = opcode;
3     this.operand = operand;
4 }
```

MOTO6809

5.2.3 Liste des Opcode Valides

Contient la liste des opcodes valides pour le processeur Motorola 6809, utilisée pour valider les opcodes dans les instructions.




```
1 public static final List<String> VALID_OPCODES = Arrays.asList(
2     "LDA", "LDB", "LDU", "LDS", "LDX", "LDY",
3     "STA", "STB",
4     "INCA", "INCB", "INC",
5     "DECA", "DECB", "DEC",
6     "ADDA", "ADDB", "SUBA", "SUBB",
7     "PSHS", "PSHU", "PULS", "PULU",
8     "EXG", "TFR", "NOP", "SWI", "END",
9     "LSLA", "LSLB", "LSLU", "LSLS", "LSLX", "LSLY",
10    "LSRA", "LSRB", "LSRU", "LSRS", "LSRX", "LSRY",
11    "ROLA", "ROLB", "ROLU", "RORO", "ROLX", "ROLY",
12    "RORA", "RORB", "RORU", "RORS", "RORX", "RORY",
13    "CMPA", "CMPB", "CMPU", "CMPX", "CMPY",
14    "COMA", "COMB"
15 );
```

MOTO6809

5.2.4 Liste des Labels Valides

Contient la liste des labels valides pour les instructions de branchement, utilisée pour valider les labels dans les instructions.




```
1 public static final List<String> VALID_LABELS = Arrays.asList(  
2     "JMP", "RTS", "BRA",  
3     "BEQ", "BNE",  
4     "BMI", "BPL",  
5     "BCC", "BCS",  
6     "BVC", "BVS"  
7 );
```

MOTO6809

5.2.5 Méthode pour Détecter le Mode d'Adressage

Détermine le mode d'adressage de l'instruction en fonction de l'opérande.

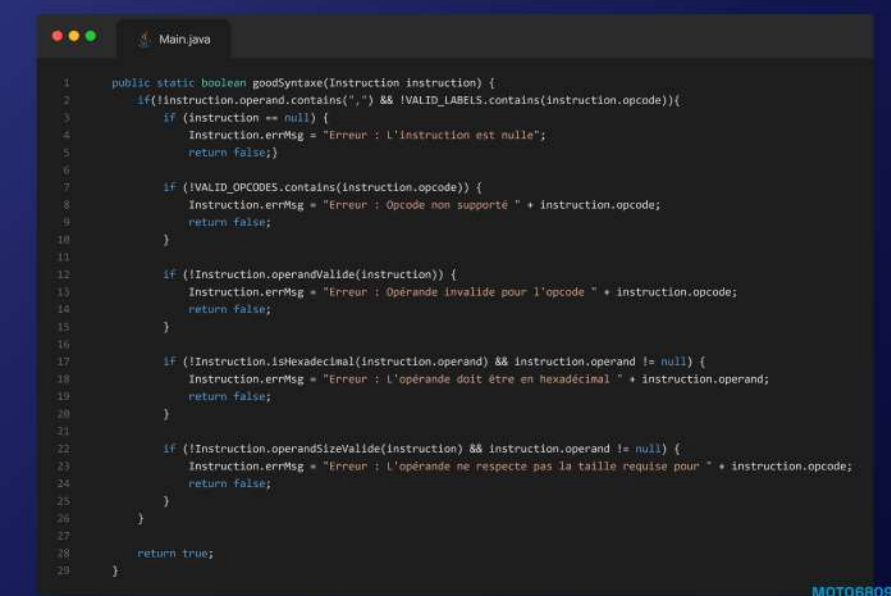


```
1 public static String detectMode(Instruction instr) {  
2     String o = instr.operand;  
3     if (o == null || o.isEmpty())  
4         return "inherent";  
5     if (o.startsWith("#"))  
6         return "immediate";  
7     if (o.startsWith("$")) {  
8         if (o.replace("$", "").length() == 2)  
9             return "direct";  
10        else  
11            return "extended";  
12    }  
13    if (o.contains(","))  
14        return "indexed";  
15    return "branch";  
16 }
```

MOTO6809

5.2.6 Méthode pour Valider la Syntaxe de l'Instruction

Valide la syntaxe de l'instruction en vérifiant plusieurs conditions : validité de l'opcode, présence et validité de l'opérande, format hexadécimal et taille requise.



```

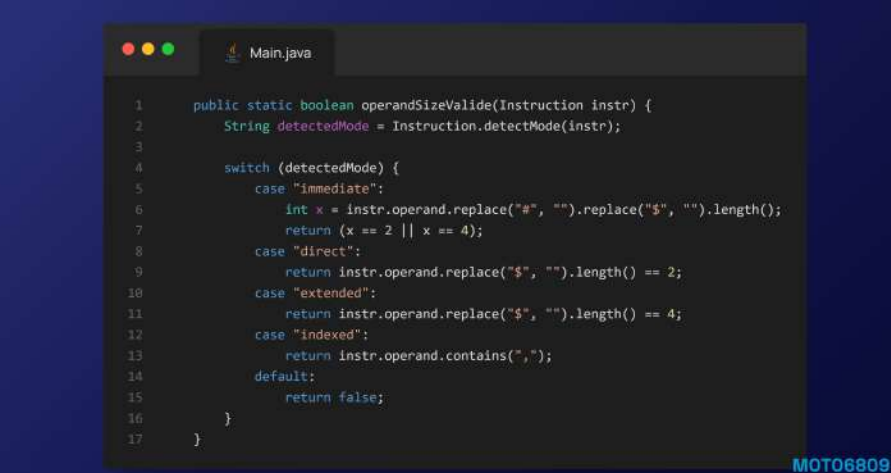
1 public static boolean goodSyntax(Instruction instruction) {
2     if(!instruction.operand.contains(",") && !VALID_LABELS.contains(instruction.opcode)){
3         if (instruction == null) {
4             instruction.errMsg = "Erreur : L'instruction est nulle";
5             return false;
6         }
7
8         if (!VALID_OPCODES.contains(instruction.opcode)) {
9             instruction.errMsg = "Erreur : Opcode non supporté " + instruction.opcode;
10            return false;
11        }
12
13        if (!instruction.operandValide(instruction)) {
14            instruction.errMsg = "Erreur : Opérande invalide pour l'opcode " + instruction.opcode;
15            return false;
16        }
17
18        if (!instruction.isHexadecimal(instruction.operand) && instruction.operand != null) {
19            instruction.errMsg = "Erreur : L'opérande doit être en hexadécimal " + instruction.operand;
20            return false;
21        }
22
23        if (!instruction.operandSizeValide(instruction) && instruction.operand != null) {
24            instruction.errMsg = "Erreur : L'opérande ne respecte pas la taille requise pour " + instruction.opcode;
25            return false;
26        }
27    }
28    return true;
29 }

```

MOTO6809

5.2.7 Méthode pour Valider la Taille de l'Opérande

Vérifie si la taille de l'opérande est valide en fonction du mode d'adressage détecté.



```

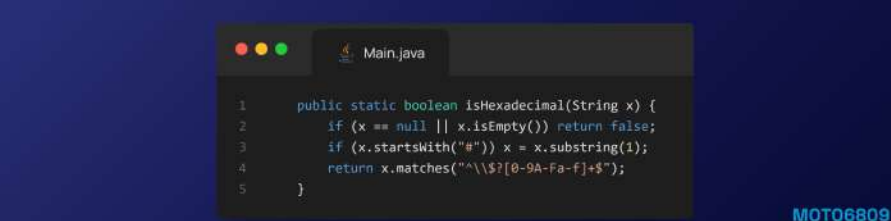
1 public static boolean operandSizeValide(Instruction instr) {
2     String detectedMode = Instruction.detectMode(instr);
3
4     switch (detectedMode) {
5         case "immediate":
6             int x = instr.operand.replace("#", "").replace("$", "").length();
7             return (x == 2 || x == 4);
8         case "direct":
9             return instr.operand.replace("$", "").length() == 2;
10        case "extended":
11            return instr.operand.replace("$", "").length() == 4;
12        case "indexed":
13            return instr.operand.contains(",");
14        default:
15            return false;
16    }
17 }

```

MOTO6809

5.2.8 Méthode pour Vérifier si l'Opérande est en Hexadécimal

Vérifie si une chaîne de caractères est en format hexadécimal.



```

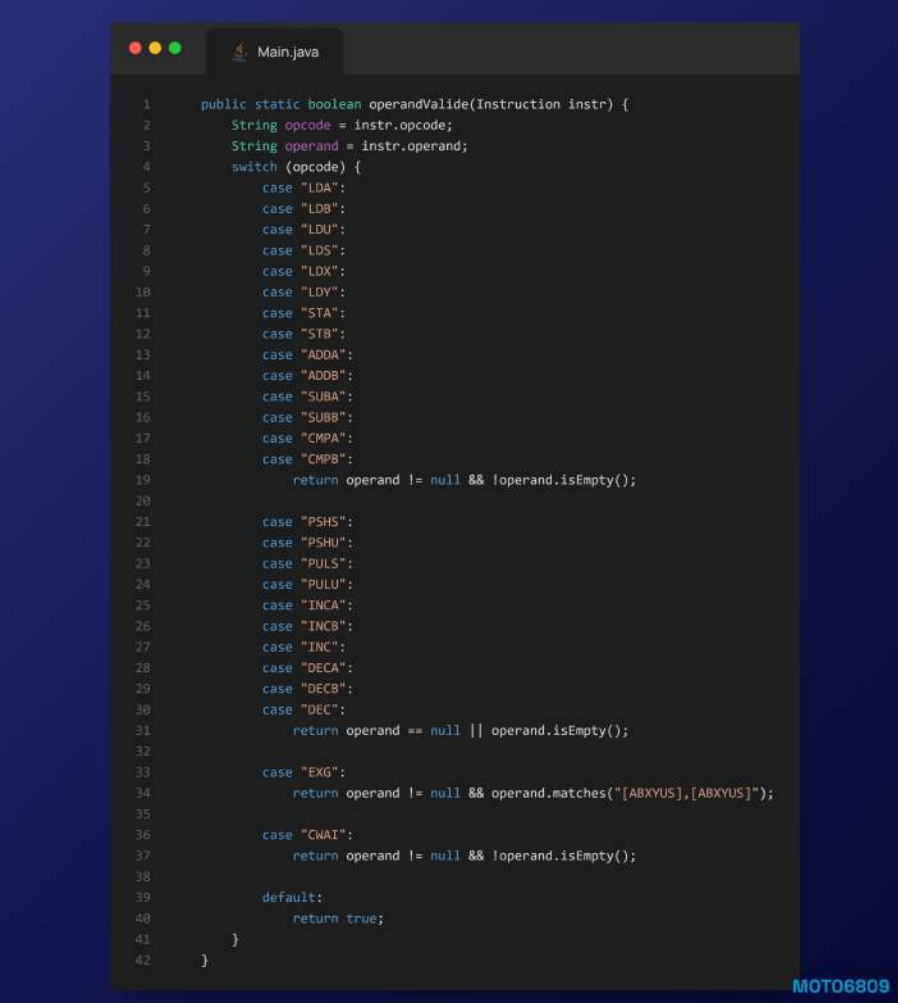
1 public static boolean isHexadecimal(String x) {
2     if (x == null || x.isEmpty()) return false;
3     if (x.startsWith("#")) x = x.substring(1);
4     return x.matches("^\\$?[0-9A-Fa-f]+$");
5 }

```

MOTO6809

5.2.9 Méthode pour Valider l'Opérande

Vérifie si l'opérande est valide pour l'opcode donné.

A screenshot of a Java IDE window titled 'Main.java'. The code defines a static method 'operandValide' that takes an 'Instruction' object and returns a boolean. It uses a switch statement to check the opcode against a list of valid instructions. For most opcodes, it checks if the operand is not null and not empty. For 'EXG', it checks if the operand matches a specific format. For 'CMAI', it checks if the operand is not null and empty. The default case returns true.

```
1 public static boolean operandValide(Instruction instr) {
2     String opcode = instr.opcode;
3     String operand = instr.operand;
4     switch (opcode) {
5         case "LDA":
6         case "LDB":
7         case "LDU":
8         case "LDS":
9         case "LDX":
10        case "LDY":
11        case "STA":
12        case "STB":
13        case "ADD":
14        case "ADB":
15        case "SUB":
16        case "SDB":
17        case "CMP":
18        case "CPB":
19            return operand != null && !operand.isEmpty();
20
21        case "PSHS":
22        case "PSHU":
23        case "PULS":
24        case "PULU":
25        case "INCA":
26        case "INCB":
27        case "INC":
28        case "DECA":
29        case "DECB":
30        case "DEC":
31            return operand == null || operand.isEmpty();
32
33        case "EXG":
34            return operand != null && operand.matches("[ABXYUS],[ABXYUS]");
35
36        case "CMAI":
37            return operand != null && !operand.isEmpty();
38
39        default:
40            return true;
41    }
42 }
```

MOTO6809

5.2.10 Méthode pour Obtenir l'Adresse

Génère une adresse en fonction d'un index k.

A screenshot of a Java IDE window titled 'Main.java'. The code defines a static method 'getAddress' that takes an integer 'k' and returns a string representing an address. The address is calculated as 'F' followed by the value of 'k' plus 600.

```
1 public static String getAddress(int k) {
2     return "F" + (600 + k);
3 }
```

MOTO6809

5.2.11 Méthode pour Obtenir le Code de l'Instruction

Retourne le code hexadécimal de l'instruction en fonction de l'opcode et du mode d'adressage détecté.

```
1 public static String getCode(Instruction instr) {
2     String opcode = instr.opcode;
3     String mode = Instruction.detectMode(instr);
4     String code = "";
5
6     switch (mode) {
7         case "inherent":
8             switch (opcode) {
9                 case "INCA": code = "4C"; break;
10                case "INCB": code = "5C"; break;
11                case "INC": code = "7C"; break;
12                case "DECA": code = "4A"; break;
13                case "DECB": code = "5A"; break;
14                case "DEC": code = "7A"; break;
15                case "PSHS": code = "34"; break;
16                case "PSHU": code = "36"; break;
17                case "PULS": code = "35"; break;
18                case "PULU": code = "37"; break;
19                case "EXG": code = "1E"; break;
20                case "TFR": code = "1F"; break;
21                case "NOP": code = "12"; break;
22                case "SWI": code = "3F"; break;
23                case "END": code = "39"; break;
24                case "LSLA": code = "48"; break;
25                case "LSLB": code = "58"; break;
26                case "LSRA": code = "44"; break;
27                case "LSRB": code = "54"; break;
28                case "ROLA": code = "49"; break;
29                case "ROLB": code = "59"; break;
30                case "RORA": code = "46"; break;
31                case "RORB": code = "56"; break;
32                case "COMA": code = "43"; break;
33                case "COMB": code = "53"; break;
34            }
35            break;
36    }
```

MOTO6809

```

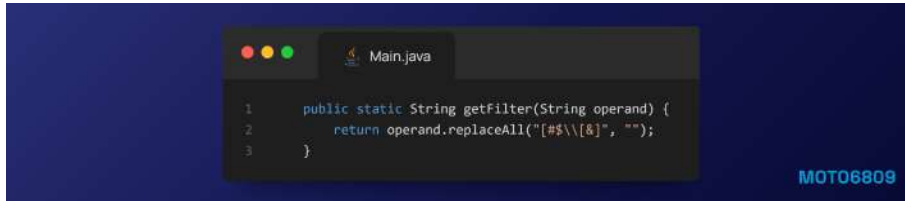
1  case "immediate":
2      switch (opcode) {
3          case "LDA": code = "86"; break;
4          case "LDB": code = "C6"; break;
5          case "LDU": code = "CE"; break;
6          case "LDS": code = "8E"; break;
7          case "LDX": code = "8E"; break;
8          case "LDY": code = "CE"; break;
9          case "ADDA": code = "8B"; break;
10         case "ADDB": code = "CB"; break;
11         case "SUBA": code = "80"; break;
12         case "SUBB": code = "C0"; break;
13         case "CMPA": code = "81"; break;
14         case "CMPB": code = "C1"; break;
15     }
16     break;
17
18     case "direct":
19         switch (opcode) {
20             case "LDA": code = "96"; break;
21             case "LDB": code = "D6"; break;
22             case "LDU": code = "DE"; break;
23             case "LDS": code = "9E"; break;
24             case "LDX": code = "9E"; break;
25             case "LDY": code = "DE"; break;
26             case "STA": code = "97"; break;
27             case "STB": code = "D7"; break;
28             case "CMPA": code = "91"; break;
29             case "CMPB": code = "D1"; break;
30         }
31         break;
32
33     case "extended":
34         switch (opcode) {
35             case "JMP": code = "7E"; break;
36             case "LDA": code = "B6"; break;
37             case "LDB": code = "F6"; break;
38             case "LDU": code = "FE"; break;
39             case "LDS": code = "BE"; break;
40             case "LDX": code = "BE"; break;
41             case "LDY": code = "FE"; break;
42             case "STA": code = "B7"; break;
43             case "STB": code = "F7"; break;
44         }
45         break;
46
47     case "indexed":
48         switch (opcode) {
49             case "LDA": code = "A6"; break;
50             case "LDB": code = "E6"; break;
51             case "LDX": code = "AE"; break;
52             case "LDY": code = "10"; break;
53             case "STA": code = "A7"; break;
54             case "STB": code = "E7"; break;
55             case "CMPA": code = "A1"; break;
56             case "CMPB": code = "E1"; break;
57         }
58         break;
59
60     case "branch":
61         switch (opcode) {
62             case "JMP": code = "A6"; break;
63             case "RTS": code = "E6"; break;
64             case "BEQ": code = "AE"; break;
65             case "BNE": code = "10"; break;
66             case "BPL": code = "A7"; break;
67             case "BCC": code = "E7"; break;
68             case "BCS": code = "A1"; break;
69             case "BVC": code = "E1"; break;
70             case "BVS": code = "E3"; break;
71             case "BRA": code = "G9"; break;
72         }
73         break;
74
75     default:
76         throw new IllegalArgumentException("Mode d'adressage inconnu : " + mode);
77 }
78
79 return code;
80 }

```

MOTO6809

5.2.12 Méthode pour Filtrer l'Opérande

Filtre l'opérande en supprimant les caractères spéciaux comme #, \$, [, et &.



```
1 public static String getFilter(String operand) {
2     return operand.replaceAll("#$\\[&", "");
3 }
```

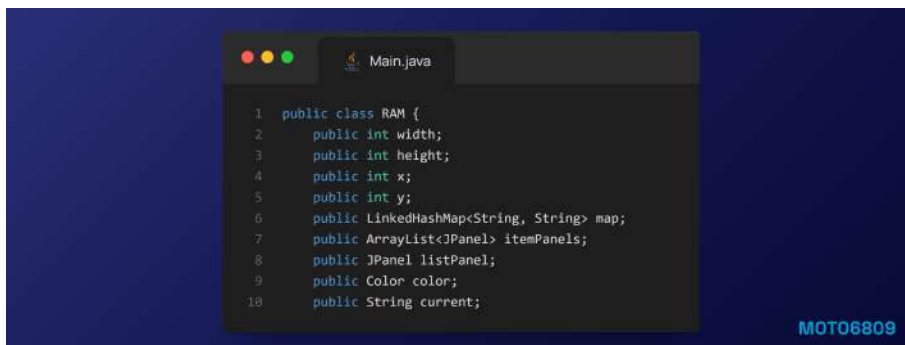
MOTO6809

5.3 Présentation de classes Des interfaces graphiques

5.3.1 Les Fichiers RAM.java et ROM.java :

Dans les fichiers RAM et ROM, ils ont la même logique de codage, telle que les classes de chaque fichier. En effet, on va montrer ci-dessous le fichier RAM.java pour détailler comment ça marche, et le fichier ROM n'est qu'une copie de ce fichier, sauf les labels et les attributs liés à chaque type de mémoire.

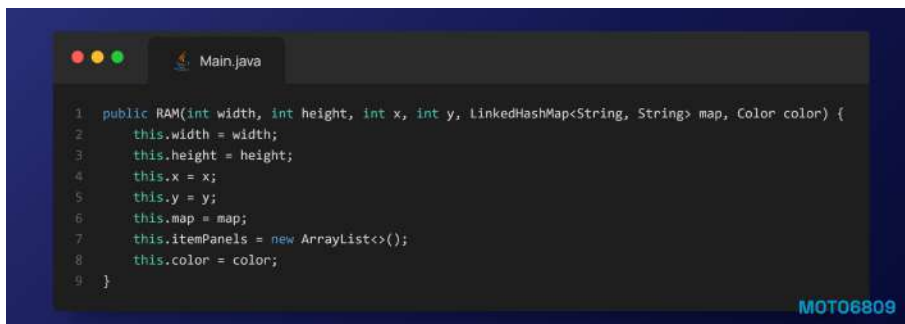
Déclaration de la Classe et des Attributs



```
1 public class RAM {
2     public int width;
3     public int height;
4     public int x;
5     public int y;
6     public LinkedHashMap<String, String> map;
7     public ArrayList<JPanel> itemPanels;
8     public JPanel listPanel;
9     public Color color;
10    public String current;
11 }
```

MOTO6809

Constructeur



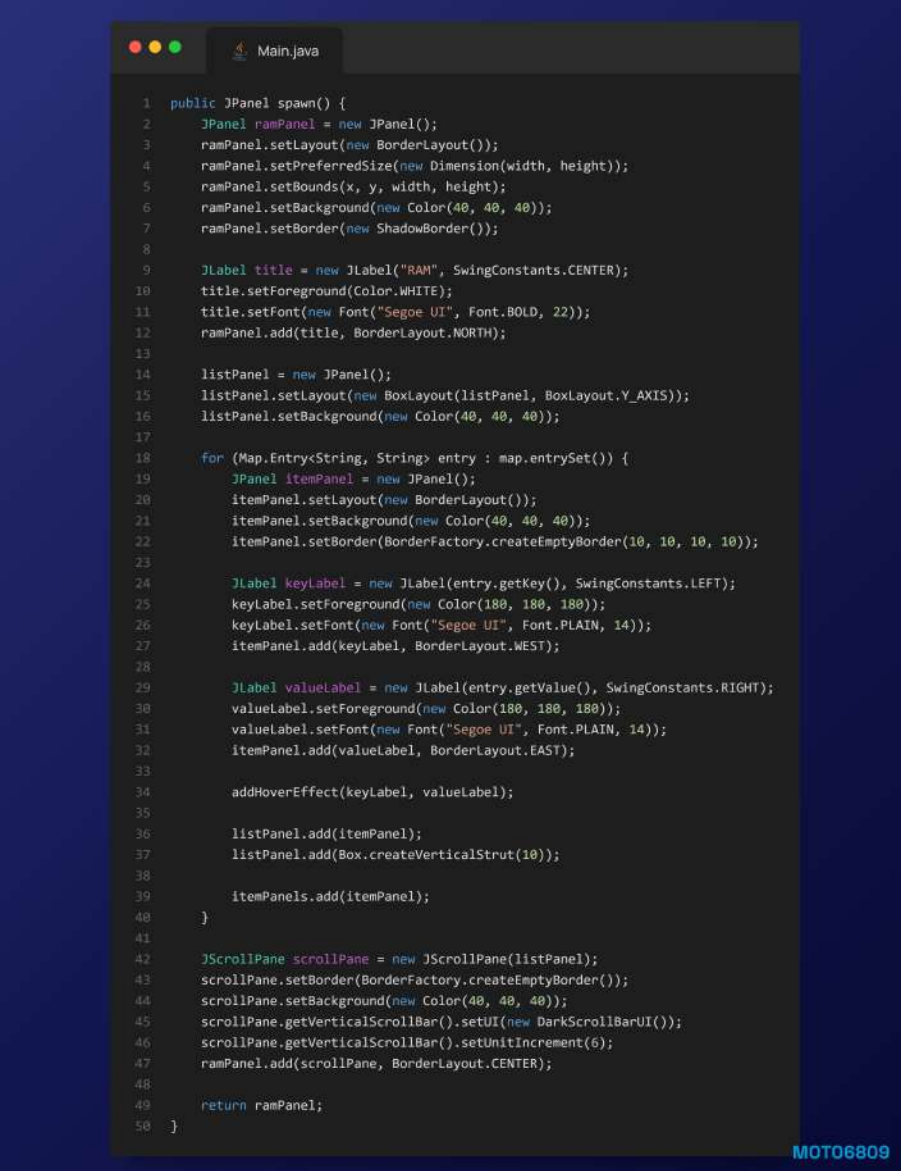
```
1 public RAM(int width, int height, int x, int y, LinkedHashMap<String, String> map, Color color) {
2     this.width = width;
3     this.height = height;
4     this.x = x;
5     this.y = y;
6     this.map = map;
7     this.itemPanels = new ArrayList<>();
8     this.color = color;
9 }
```

MOTO6809

Méthode spawn

Cette méthode crée et retourne un JPanel contenant la liste des éléments. Elle configure le panneau principal (ramPanel), ajoute un titre, et crée un JScrollPane pour permettre le défilement de la liste des éléments. Chaque élément est représenté par un JPanel contenant deux

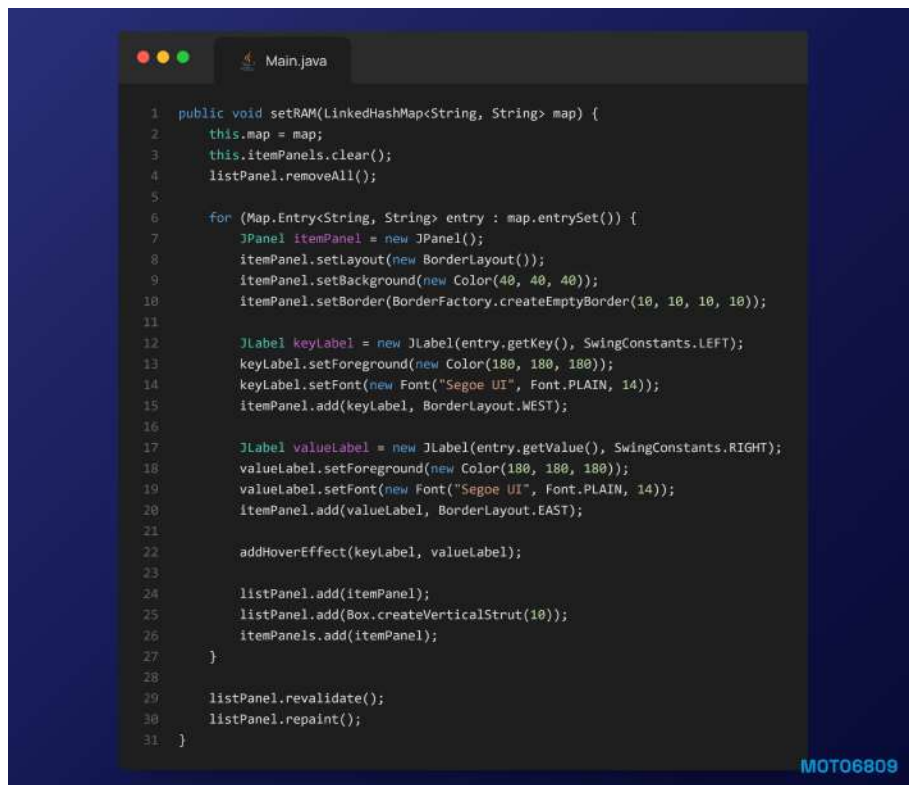
JLabel pour la clé et la valeur.

A screenshot of a Java IDE window titled 'Main.java'. The code defines a 'spawn()' method for a 'JPanel'. It creates a 'ramPanel' with a 'BorderLayout', sets its preferred size, bounds, background color (40, 40, 40), and border (ShadowBorder). A 'JLabel' titled 'RAM' is added to the north position. Then, a 'listPanel' is created with a 'BoxLayout'. A loop iterates over a map of entries, creating 'itemPanel's. Each 'itemPanel' has a 'BorderLayout', background color, and a thin border. It contains two 'JLabel's: 'keyLabel' (left-aligned, white font) and 'valueLabel' (right-aligned, white font). A hover effect is added between the labels. The 'itemPanel's are added to the 'listPanel' with vertical struts. Finally, a 'JScrollPane' is created with the 'listPanel' and added to the center of 'ramPanel'.

```
1 public JPanel spawn() {
2     JPanel ramPanel = new JPanel();
3     ramPanel.setLayout(new BorderLayout());
4     ramPanel.setPreferredSize(new Dimension(width, height));
5     ramPanel.setBounds(x, y, width, height);
6     ramPanel.setBackground(new Color(40, 40, 40));
7     ramPanel.setBorder(new ShadowBorder());
8
9     JLabel title = new JLabel("RAM", SwingConstants.CENTER);
10    title.setForeground(Color.WHITE);
11    title.setFont(new Font("Segoe UI", Font.BOLD, 22));
12    ramPanel.add(title, BorderLayout.NORTH);
13
14    listPanel = new JPanel();
15    listPanel.setLayout(new BoxLayout(listPanel, BoxLayout.Y_AXIS));
16    listPanel.setBackground(new Color(40, 40, 40));
17
18    for (Map.Entry<String, String> entry : map.entrySet()) {
19        JPanel itemPanel = new JPanel();
20        itemPanel.setLayout(new BorderLayout());
21        itemPanel.setBackground(new Color(40, 40, 40));
22        itemPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
23
24        JLabel keyLabel = new JLabel(entry.getKey(), SwingConstants.LEFT);
25        keyLabel.setForeground(new Color(180, 180, 180));
26        keyLabel.setFont(new Font("Segoe UI", Font.PLAIN, 14));
27        itemPanel.add(keyLabel, BorderLayout.WEST);
28
29        JLabel valueLabel = new JLabel(entry.getValue(), SwingConstants.RIGHT);
30        valueLabel.setForeground(new Color(180, 180, 180));
31        valueLabel.setFont(new Font("Segoe UI", Font.PLAIN, 14));
32        itemPanel.add(valueLabel, BorderLayout.EAST);
33
34        addHoverEffect(keyLabel, valueLabel);
35
36        listPanel.add(itemPanel);
37        listPanel.add(Box.createVerticalStrut(10));
38
39        itemPanels.add(itemPanel);
40    }
41
42    JScrollPane scrollPane = new JScrollPane(listPanel);
43    scrollPane.setBorder(BorderFactory.createEmptyBorder());
44    scrollPane.setBackground(new Color(40, 40, 40));
45    scrollPane.getVerticalScrollBar().setUI(new DarkScrollBarUI());
46    scrollPane.getVerticalScrollBar().setUnitIncrement(6);
47    ramPanel.add(scrollPane, BorderLayout.CENTER);
48
49    return ramPanel;
50 }
```

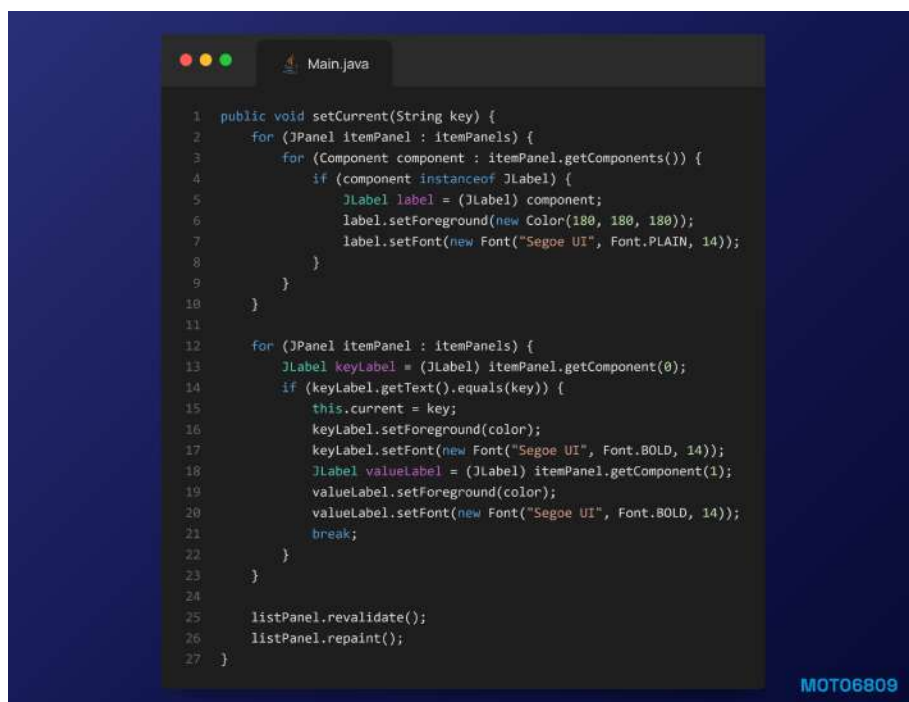
MOTO6809

Méthode setRAM



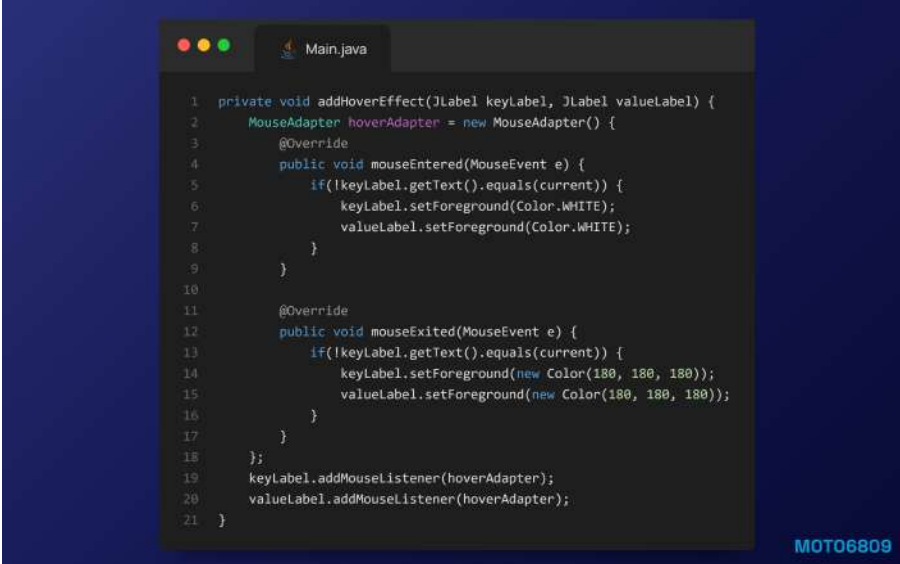
Méthode setCurrent

Cette méthode met en évidence l'élément actuel en changeant la couleur et la police des JLabel correspondants. Elle réinitialise d'abord tous les éléments, puis met en évidence l'élément spécifié par la clé key.



Méthode addHoverEffect

Cette méthode ajoute un effet de survol aux JLabel. Lorsque la souris entre dans la zone d'un label, sa couleur change en blanc, et lorsque la souris sort, la couleur revient à la normale, sauf si l'élément est l'élément actuel.

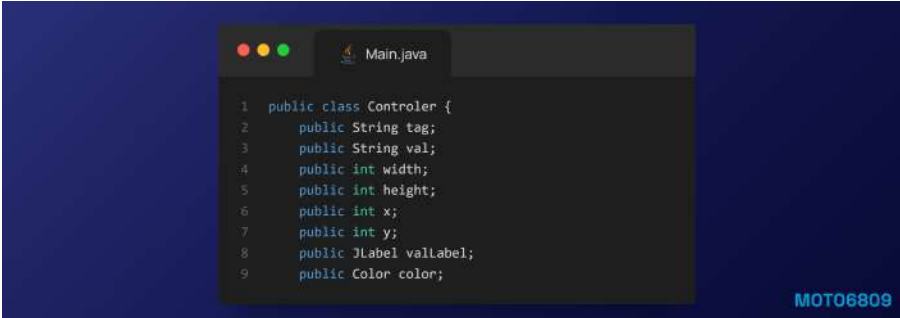


```
1 private void addHoverEffect(JLabel keyLabel, JLabel valueLabel) {
2     MouseAdapter hoverAdapter = new MouseAdapter() {
3         @Override
4         public void mouseEntered(MouseEvent e) {
5             if(!keyLabel.getText().equals(current)) {
6                 keyLabel.setForeground(Color.WHITE);
7                 valueLabel.setForeground(Color.WHITE);
8             }
9         }
10
11         @Override
12         public void mouseExited(MouseEvent e) {
13             if(!keyLabel.getText().equals(current)) {
14                 keyLabel.setForeground(new Color(180, 180, 180));
15                 valueLabel.setForeground(new Color(180, 180, 180));
16             }
17         }
18     };
19     keyLabel.addMouseListener(hoverAdapter);
20     valueLabel.addMouseListener(hoverAdapter);
21 }
```

MOTO6809

5.3.2 Les Fichiers Controler.java et Registre.java :

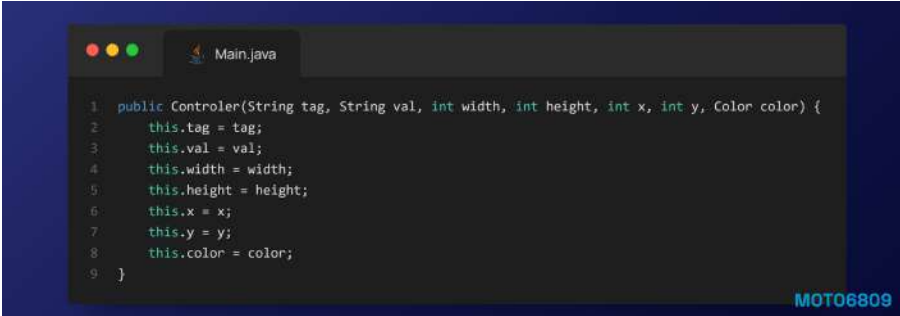
Déclaration de la Classe et des Attributs



```
1 public class Controller {
2     public String tag;
3     public String val;
4     public int width;
5     public int height;
6     public int x;
7     public int y;
8     public JLabel vallabel;
9     public Color color;
10 }
```

MOTO6809

Constructeur

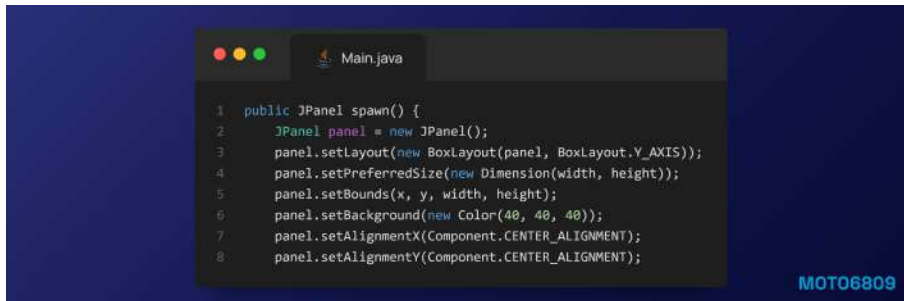


```
1 public Controller(String tag, String val, int width, int height, int x, int y, Color color) {
2     this.tag = tag;
3     this.val = val;
4     this.width = width;
5     this.height = height;
6     this.x = x;
7     this.y = y;
8     this.color = color;
9 }
```

MOTO6809

Méthode spawn

Cette méthode crée et retourne un JPanel contenant l'étiquette et la valeur. Elle configure le panneau principal (panel) avec une disposition verticale (BoxLayout), définit ses dimensions et sa position, et configure son arrière-plan et son alignement.

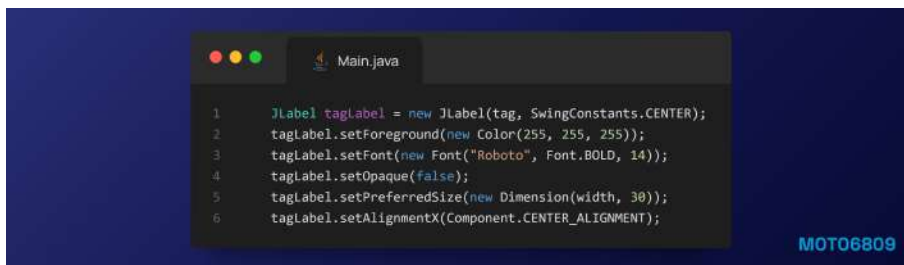
A screenshot of a Java IDE window titled 'Main.java' with a dark background. The code defines a public static method 'spawn()' that creates and configures a JPanel. The code is as follows:

```
1 public JPanel spawn() {
2     JPanel panel = new JPanel();
3     panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
4     panel.setPreferredSize(new Dimension(width, height));
5     panel.setBounds(x, y, width, height);
6     panel.setBackground(new Color(40, 40, 40));
7     panel.setAlignmentX(Component.CENTER_ALIGNMENT);
8     panel.setAlignmentY(Component.CENTER_ALIGNMENT);
9 }
```

The IDE has standard window controls (red, yellow, green buttons) in the top left. The username 'MOTO6809' is visible in the bottom right corner.

Ajout des Étiquettes

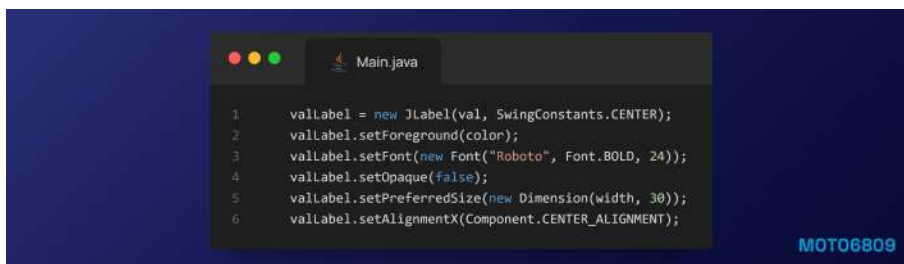
Ce bloc de code crée un JLabel pour l'étiquette (tagLabel). Il configure la couleur du texte, la police, la transparence, les dimensions préférées, et l'alignement.

A screenshot of a Java IDE window titled 'Main.java' with a dark background. The code creates a JLabel object named 'tagLabel' and configures its appearance. The code is as follows:

```
1 JLabel tagLabel = new JLabel(tag, SwingConstants.CENTER);
2 tagLabel.setForeground(new Color(255, 255, 255));
3 tagLabel.setFont(new Font("Roboto", Font.BOLD, 14));
4 tagLabel.setOpaque(false);
5 tagLabel.setPreferredSize(new Dimension(width, 30));
6 tagLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
```

The IDE has standard window controls in the top left. The username 'MOTO6809' is visible in the bottom right corner.

Ce bloc de code crée un JLabel pour la valeur (valLabel). Il configure la couleur du texte, la police, la transparence, les dimensions préférées, et l'alignement.

A screenshot of a Java IDE window titled 'Main.java' with a dark background. The code creates a JLabel object named 'valLabel' and configures its appearance. The code is as follows:

```
1 valLabel = new JLabel(val, SwingConstants.CENTER);
2 valLabel.setForeground(color);
3 valLabel.setFont(new Font("Roboto", Font.BOLD, 24));
4 valLabel.setOpaque(false);
5 valLabel.setPreferredSize(new Dimension(width, 30));
6 valLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
```

The IDE has standard window controls in the top left. The username 'MOTO6809' is visible in the bottom right corner.

Ajout des Étiquettes au Panneau

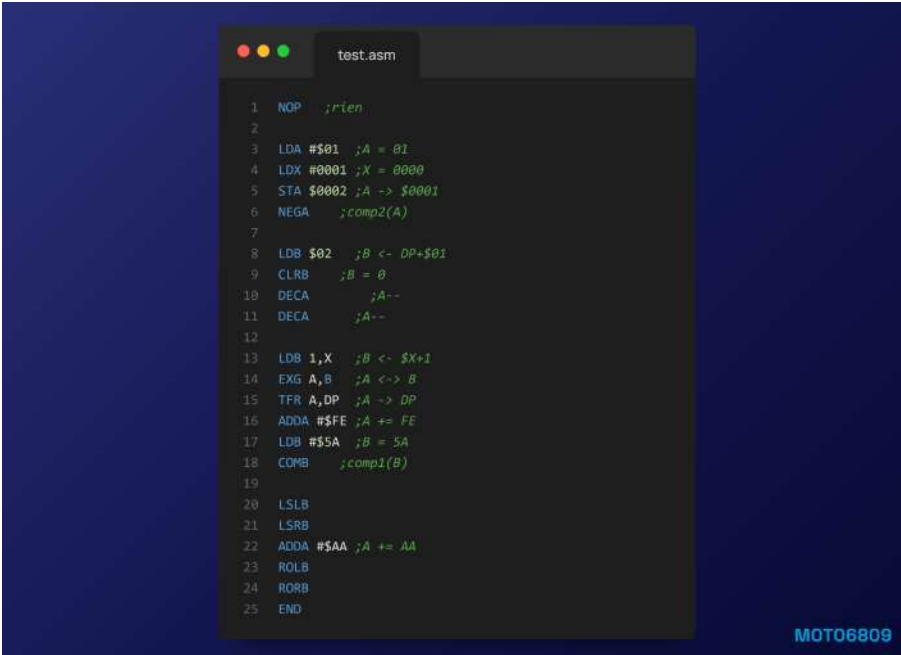
Ces lignes ajoutent les JLabel (tagLabel et valLabel) au panneau principal (panel), puis retournent le panneau.



6. Tests

Dans ce fichier, nous allons explorer plusieurs instructions d'assemblage qui utilisent différents modes d'adressage. Chaque instruction sera accompagnée d'une brève explication de son fonctionnement et de son mode d'adressage spécifique. Cela permettra de mieux comprendre comment les données sont manipulées et stockées dans la mémoire, ainsi que les différentes manières d'accéder à ces données.

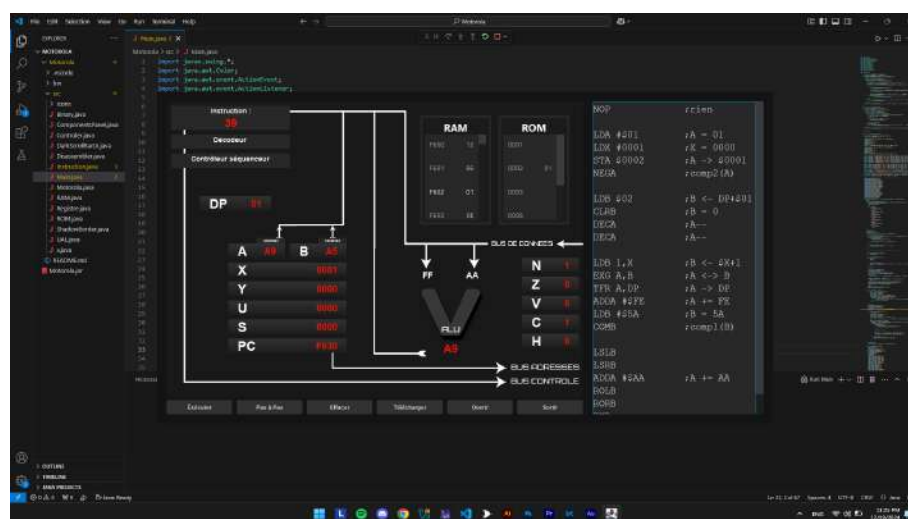
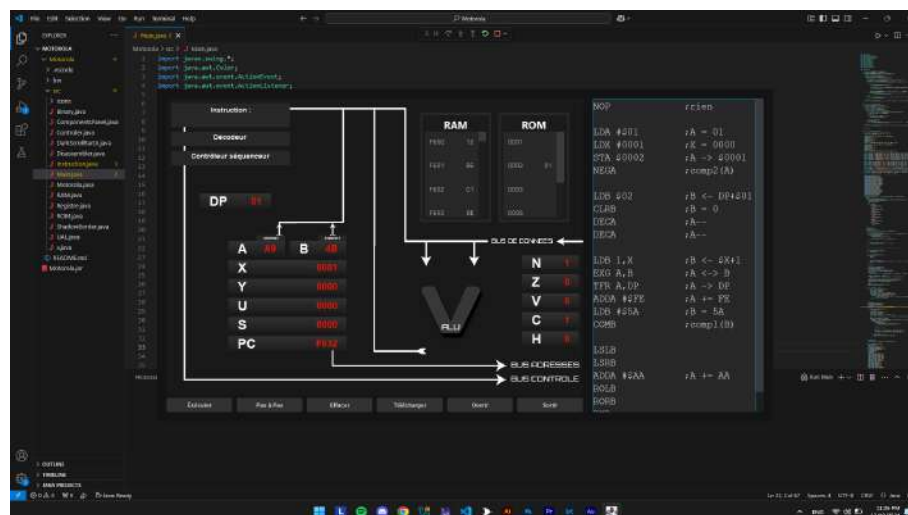
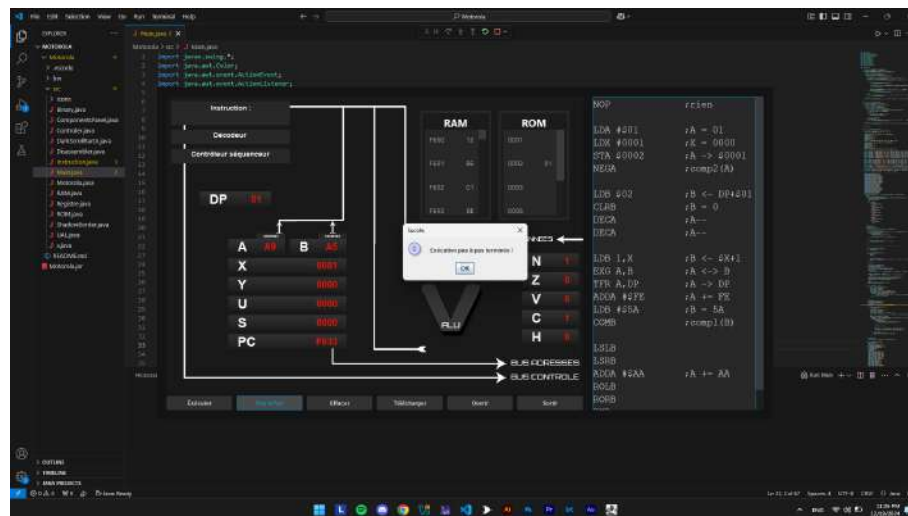
Nous allons commencer par examiner chaque instruction et son mode d'adressage associé, en fournissant des commentaires détaillés pour chaque étape. Cela nous permettra de voir comment les différents modes d'adressage sont utilisés dans un programme réel et comment ils contribuent à la flexibilité et à l'efficacité du code assembleur.

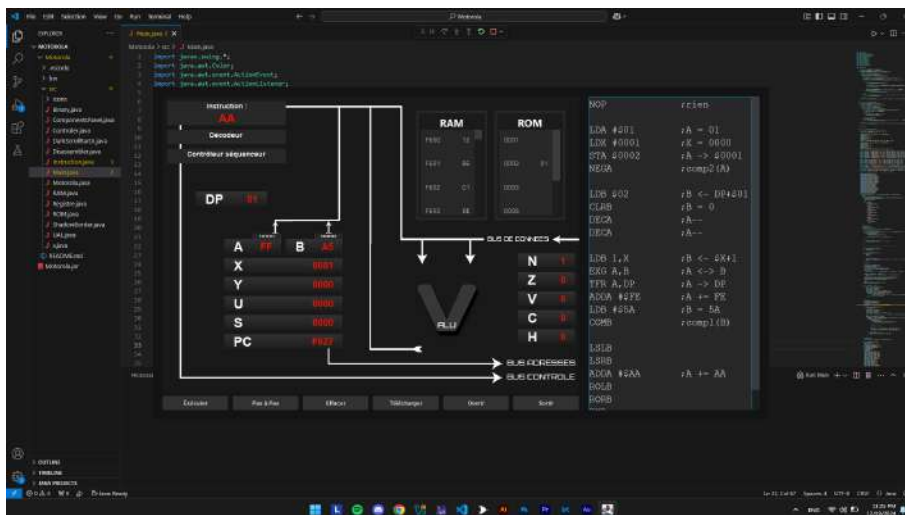
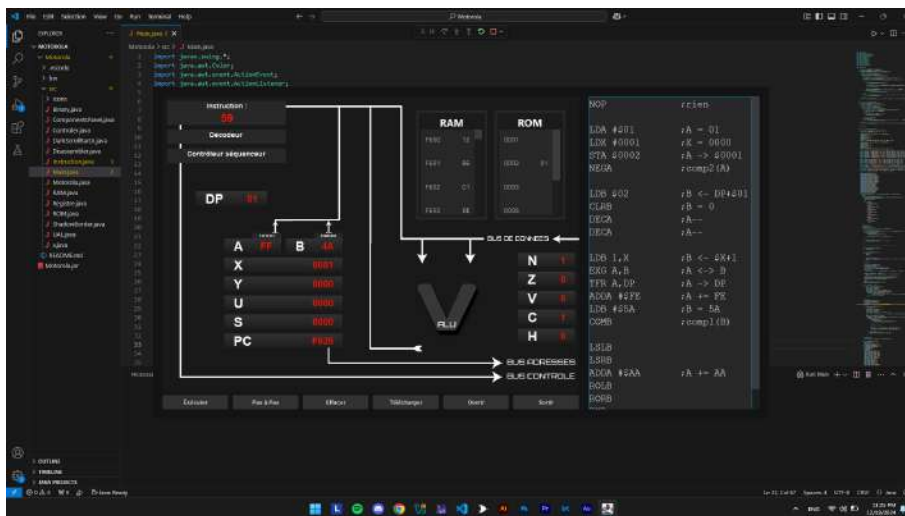
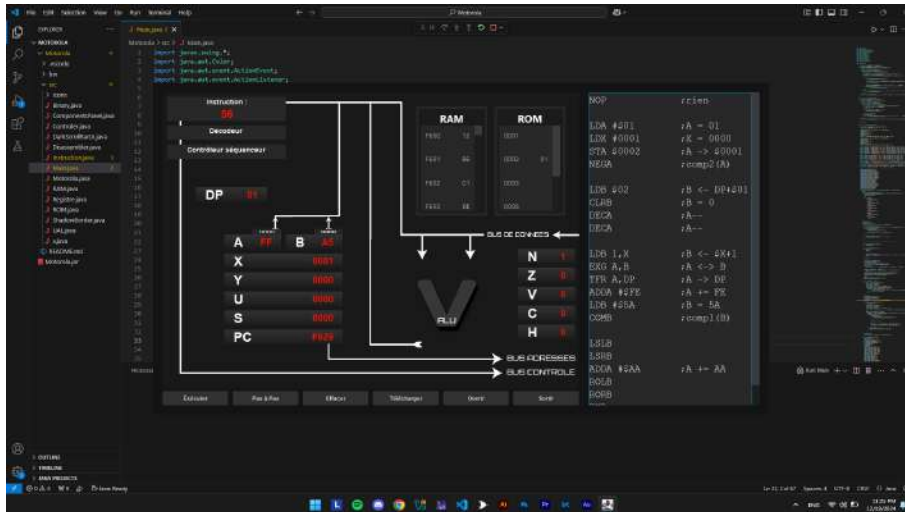


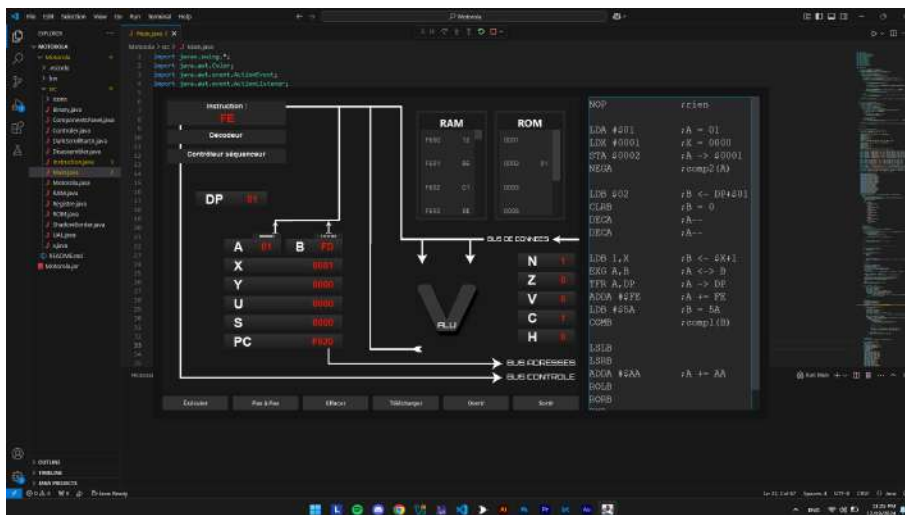
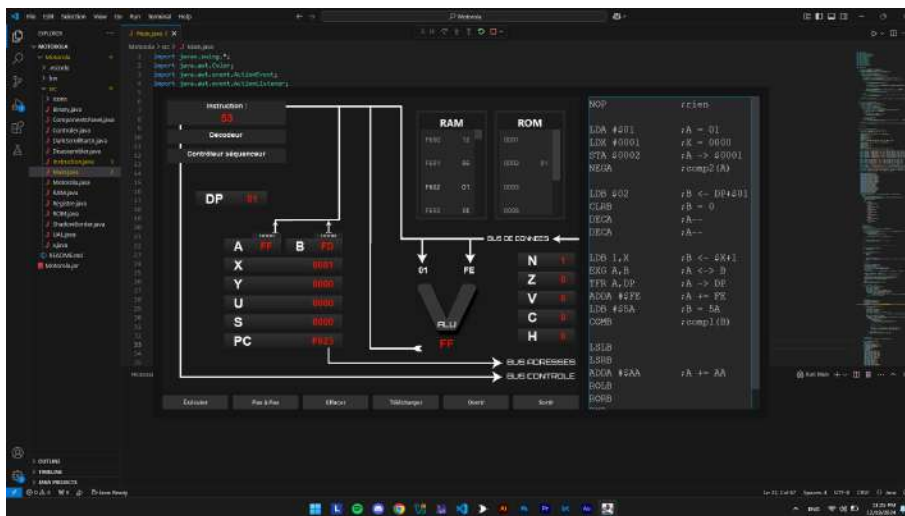
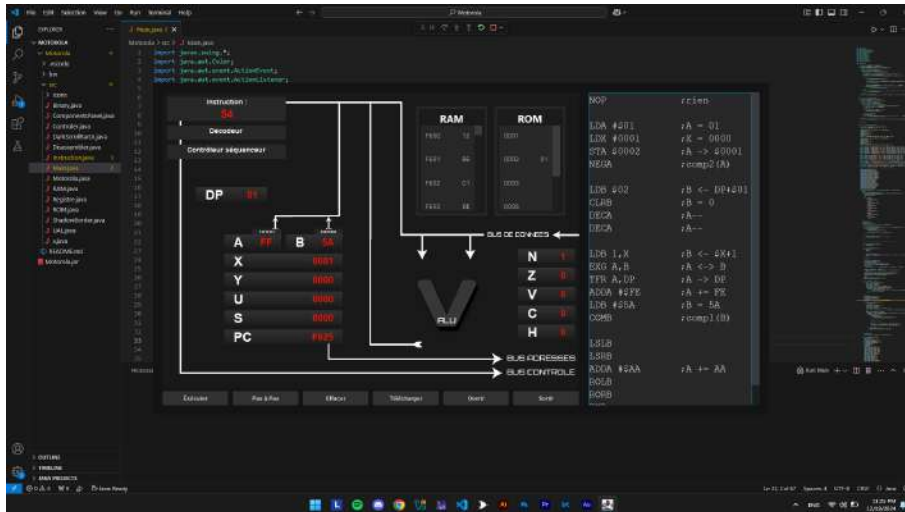
```
1  NOP    ;rien
2
3  LDA #01    ;A = 01
4  LDX #0001  ;X = 0000
5  STA $0002  ;A -> $0001
6  NEGA      ;comp2(A)
7
8  LDB $02    ;B <- DP+$01
9  CLRB      ;B = 0
10 DECA      ;A--
11 DECA      ;A--
12
13 LDB 1,X    ;B <- $X+1
14 EXG A,B   ;A <-> B
15 TFR A,DP   ;A -> DP
16 ADDA #$FE  ;A += FE
17 LDB #$5A   ;B = 5A
18 COMB      ;comp1(B)
19
20 LSLB
21 LSRB
22 ADDA #$AA  ;A += AA
23 ROLB
24 RORB
25 END
```

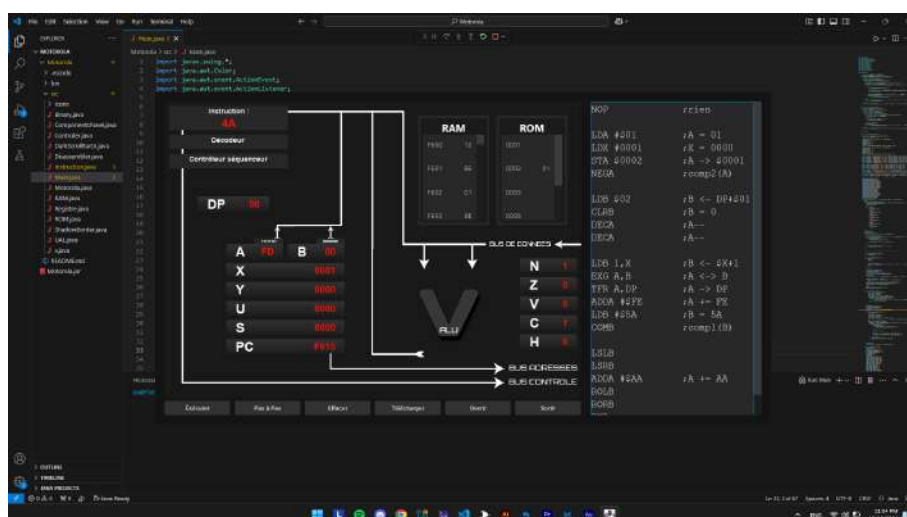
MOTO6809

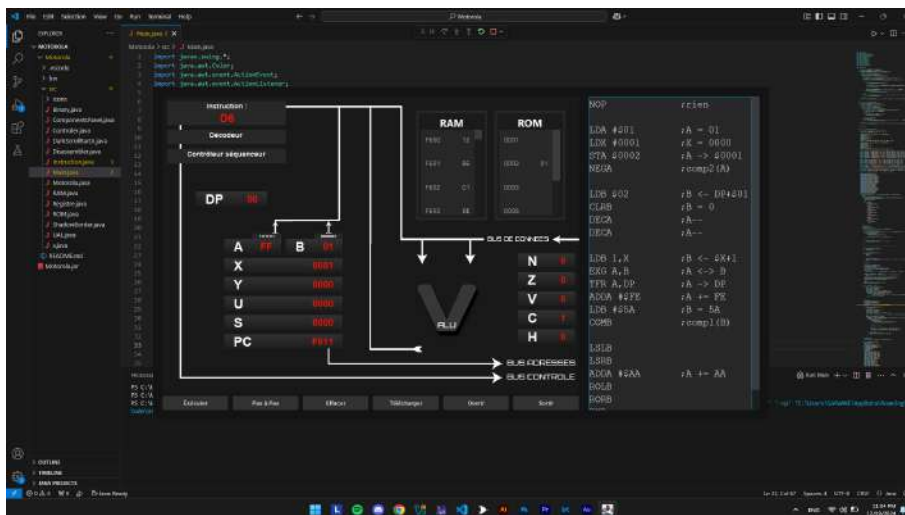
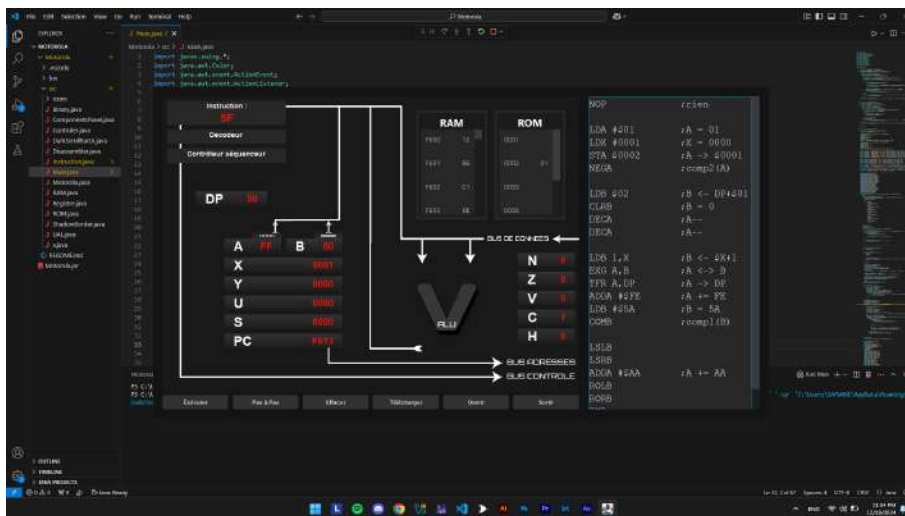
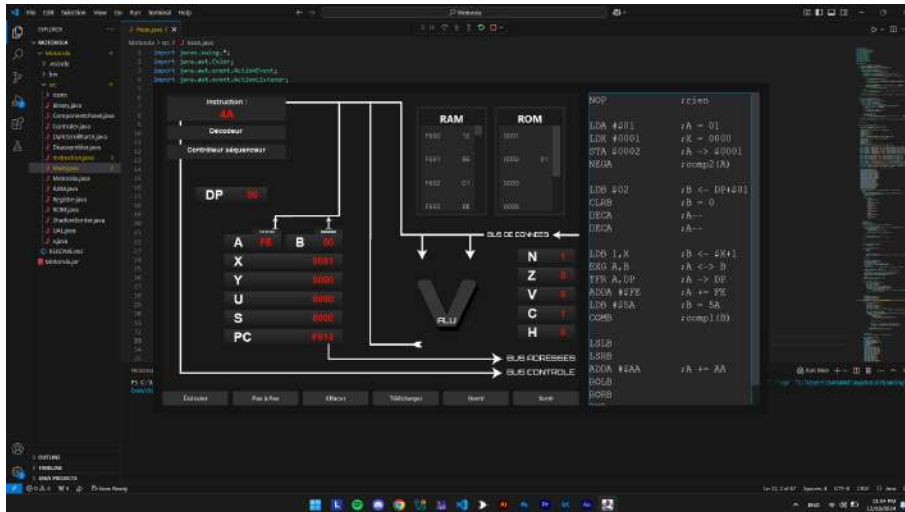
dans la suite des Screenshots, ils montrent l'exécution des instructions pas à pas du fichier test ci-dessus ;

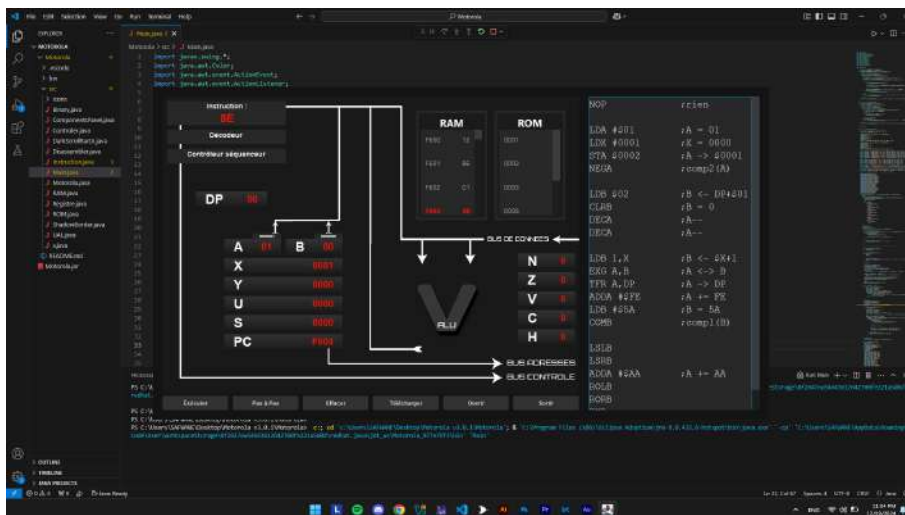
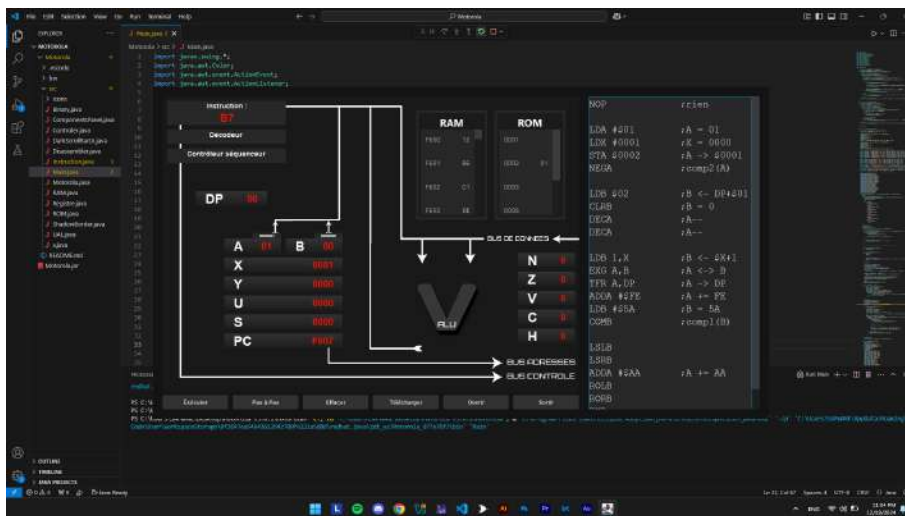
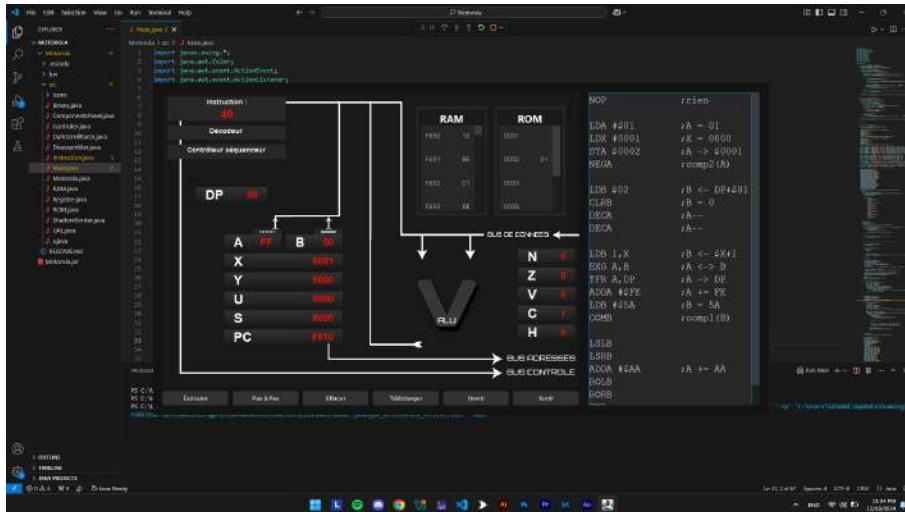


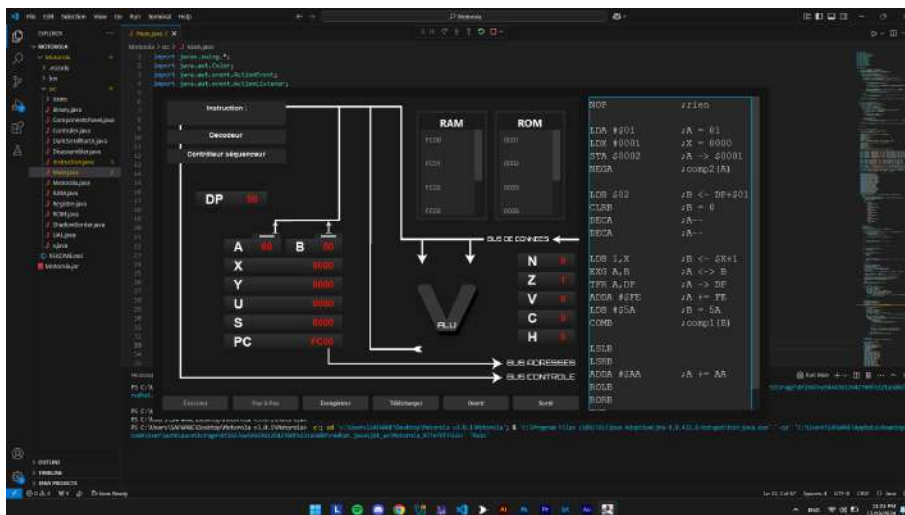
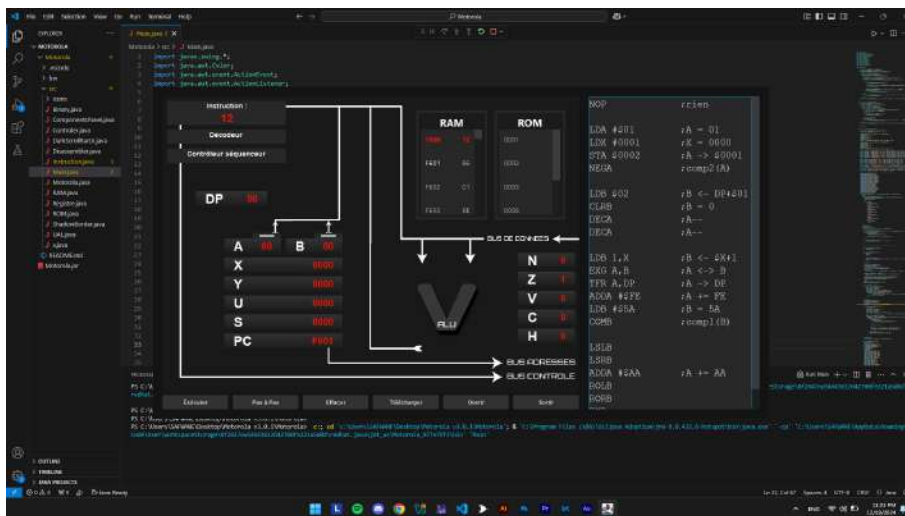
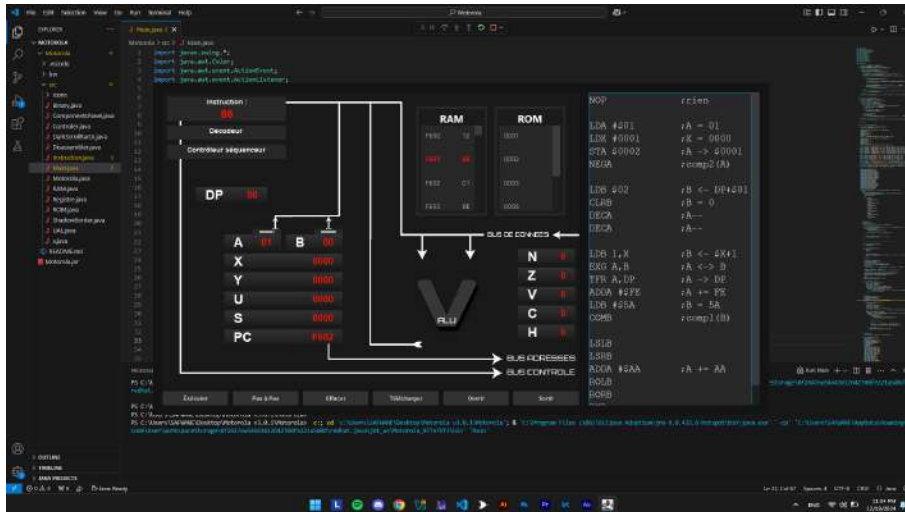


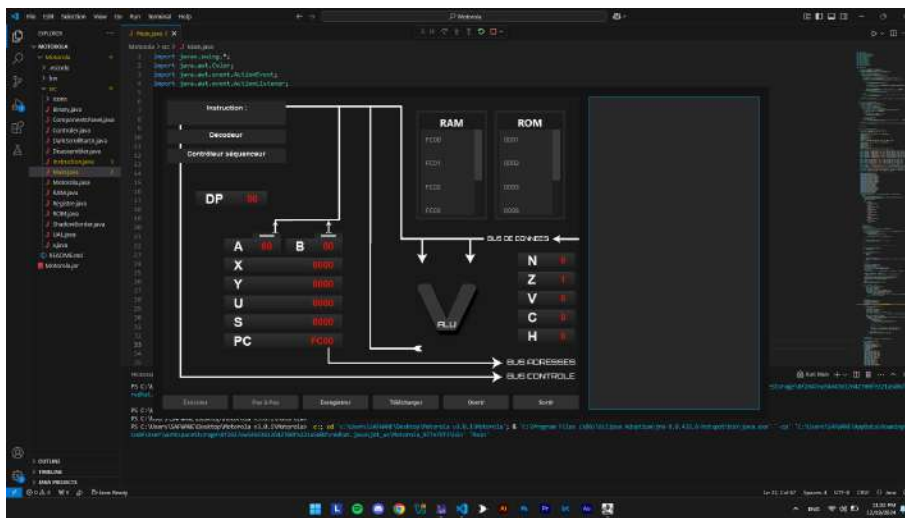
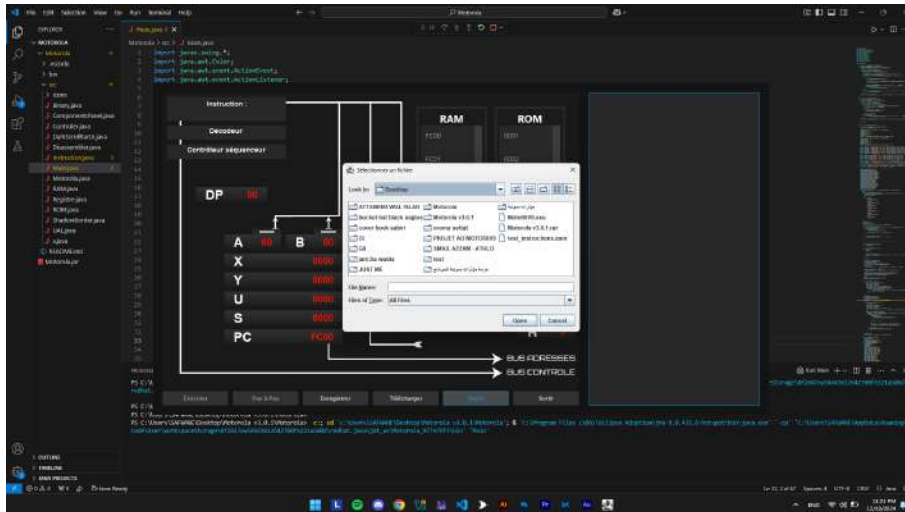












7. Défis et solutions

7.1 Incapacité à Encoder Directement les Valeurs dans la Mémoire Principale de la Machine

L'un des défis significatifs que nous avons rencontrés lors du développement de notre projet Java était l'incapacité à encoder directement les valeurs dans la mémoire principale de la machine. Cette limitation découle de l'architecture de Java, qui compile le bytecode pour être exécuté par la Java Virtual Machine (JVM) plutôt que directement par le matériel. La JVM abstrait le matériel sous-jacent, offrant un environnement indépendant de la plateforme, mais cette abstraction s'accompagne de certaines contraintes.

Plus précisément, la JVM gère l'allocation et la désallocation de la mémoire via son mécanisme de garbage collection. Bien que ce processus automatisé soit bénéfique pour la gestion de la mémoire et la prévention des fuites de mémoire, il signifie également que les développeurs ont un contrôle limité sur les adresses mémoire et la manipulation directe de la mémoire. Cela posait un problème pour nous car notre application nécessitait un contrôle précis des emplacements mémoire pour des opérations critiques en termes de performance et des interactions système de bas niveau.

7.2 Manipulation Directe des Entrées de Chaînes de Caractères

Un autre défi significatif que nous avons rencontré était la nécessité de manipuler directement les entrées de chaînes de caractères en raison des limitations de la gestion de la mémoire de Java et du mécanisme de garbage collection de la JVM. Notre application nécessitait un contrôle précis des emplacements mémoire pour des opérations critiques en termes de performance et des interactions système de bas niveau, ce qui n'était pas réalisable avec les pratiques standard de gestion de la mémoire de Java.

7.3 Court Délai pour la Simulation de l'Ensemble d'Instructions Motorola 6809

L'un des défis les plus pressants que nous avons rencontrés lors du développement de notre projet Java était le court délai alloué pour livrer une simulation complète de l'ensemble d'instructions Motorola 6809. Le Motorola 6809 est un microprocesseur complexe avec un ensemble riche d'instructions, et simuler son comportement avec précision nécessitait une compréhension approfondie de son architecture et une implémentation méticuleuse.

Pour respecter ce délai, nous avons dû abandonner le mode d'adressage indirect, ce qui a ajouté une couche de complexité supplémentaire à notre tâche. Nous avons dû prioriser les tâches, gérer notre temps efficacement et nous assurer que les progrès sur la simulation n'affectaient pas nos autres projets. Cela a nécessité un niveau élevé de dévouement, une gestion efficace du temps et un travail d'équipe efficace pour livrer la simulation Motorola 6809 à temps tout en remplissant nos autres responsabilités. Cette expérience a souligné l'importance de la planification, de la priorisation et de l'adaptabilité dans la gestion de projets complexes avec des délais serrés.

8. Conclusion et perspectives

Le projet de simulation du microprocesseur Motorola 6809 a été une entreprise ambitieuse et enrichissante, visant à reproduire fidèlement le comportement d'un processeur historique tout en offrant des fonctionnalités modernes de débogage et de visualisation. Ce projet a permis de mieux comprendre les principes fondamentaux de l'architecture des microprocesseurs, ainsi que les défis et les solutions associés à la simulation de systèmes complexes.

Les objectifs principaux du projet, tels que la simulation précise du processeur, l'émulation des instructions, la gestion de la mémoire, et la fourniture d'une interface graphique interactive, ont été atteints avec succès. Le simulateur développé offre une plateforme robuste pour tester des programmes à bas niveau, déboguer des applications historiques, et enseigner les concepts de l'architecture des processeurs.

Cependant, le projet a également rencontré plusieurs défis, notamment l'incapacité à encoder directement les valeurs dans la mémoire principale de la machine en raison des limitations de la JVM, la manipulation directe des entrées de chaînes de caractères, et le court délai pour la simulation de l'ensemble d'instructions Motorola 6809. Ces défis ont nécessité des solutions créatives et une gestion efficace du temps et des ressources.

En conclusion, ce projet a non seulement permis de développer un simulateur fonctionnel et précis du Motorola 6809, mais il a également offert une expérience précieuse en matière de gestion de projet, de résolution de problèmes, et de collaboration. Les perspectives futures incluent l'amélioration continue du simulateur, l'ajout de nouvelles fonctionnalités, et l'exploration de nouvelles méthodes pour surmonter les limitations actuelles. Ce projet constitue une base solide pour de futurs développements dans le domaine de la simulation de microprocesseurs et de l'enseignement de l'informatique.

9. References

Le développement de ce projet a bénéficié de diverses ressources en ligne qui ont été essentielles pour surmonter les défis techniques et améliorer la qualité du code. Nous avons utilisé des plateformes d'intelligence artificielle comme ChatGPT et Mistral AI pour obtenir des conseils et des suggestions sur le développement du code. W3Schools a servi de référence précieuse pour les concepts de programmation en Java. Overleaf a été notre outil de choix pour la création et la mise en forme du fichier LaTeX, tandis que Code to Image Converter nous a permis de convertir le code en images pour une meilleure visualisation dans le document. De plus, le site 6809.uk a été une ressource inestimable pour la simulation du microprocesseur Motorola 6809. Voici les références détaillées de ces outils :

ChatGPT, <https://chat.openai.com/>.

Mistral AI, <https://mistral.ai/>.

W3Schools, <https://www.w3schools.com/>.

Overleaf, <https://www.overleaf.com/>.

Code to Image Converter, <https://10015.io/tools/code-to-image-converter>.

6809.uk, <https://6809.uk/>.