



---

Université Hassan 1<sup>er</sup>  
Faculté des sciences et  
techniques  
Settat

جامعة الحسن الأول  
UNIVERSITÉ HASSAN 1<sup>er</sup>



---

# Projet architecture des ordinateurs : Simulateur de Microprocesseur Motorola 6809 en JAVA

---

Licence Sciences et Techniques en Génie Informatique (LST-GI)

Année universitaire : 2025/2026

Encadré par :

Mr. BENALLA HICHAM

Réalisé par :

FISSAL YOUSSEF

IRDA YOUNESS

## **Introduction Générale**



Le **Motorola 6809** est un microprocesseur 8 bits apparu à la fin des années 1970, conçu par Motorola comme une évolution majeure de la famille 6800. Il visait à offrir de meilleures performances, une architecture plus organisée et une plus grande flexibilité pour les programmeurs. Grâce à sa conception avancée, il a été utilisé aussi bien dans les micro-ordinateurs que dans les systèmes industriels et embarqués.

Ce processeur se distingue par une architecture moderne pour son époque, un jeu d'instructions clair et des mécanismes facilitant le développement de programmes complexes. Bien qu'il n'ait pas connu un succès commercial massif, le Motorola 6809 a marqué l'histoire des microprocesseurs et reste aujourd'hui une référence appréciée dans le domaine de la rétro-informatique et de l'enseignement.

### **Pour les objectifs généraux du projet**

Ce projet a pour but d'étudier et de mettre en valeur le fonctionnement du Motorola 6809 à travers une approche pédagogique. Il consiste notamment à :

- ✚ Simuler fidèlement le comportement interne du processeur.
- ✚ Émuler son jeu d'instructions avec une gestion précise de la mémoire et des registres.
- ✚ Développer une interface graphique simple pour suivre l'exécution des programmes.
- ✚ Préserver et analyser des logiciels anciens à des fins éducatives.

Ce travail contribue à la compréhension des bases de l'architecture des processeurs et à la valorisation du patrimoine informatique.

# **Une analyse pour Motorola 6809**

## Contexte historique et technologique

À partir des années 1970, l'informatique a connu une transformation majeure avec l'apparition des microprocesseurs. Cette période a été marquée par une forte compétition entre les grands constructeurs de semi-conducteurs. Motorola, déjà présent sur le marché avec le microprocesseur 6800 lancé en 1974, a rapidement compris la nécessité de proposer une solution plus performante et mieux adaptée aux nouveaux besoins. C'est dans ce contexte de rivalité technologique, notamment face aux processeurs Intel 8080 et Zilog Z80, que le Motorola 6809 a été conçu comme une évolution stratégique et ambitieuse.

## Architecture et spécificités techniques

Le Motorola 6809 se distingue par une architecture innovante qui le plaçait en avance sur de nombreux processeurs 8 bits de son époque. Son design interne a été pensé pour offrir à la fois puissance, flexibilité et simplicité de programmation, parmi ses éléments les plus remarquables, on trouve un jeu d'instructions cohérent et structuré, permettant une grande liberté dans l'utilisation des différents modes d'adressage, cette approche dite « orthogonale » facilitait l'écriture de programmes clairs et optimisés, le processeur intègre également deux accumulateurs distincts, pouvant fonctionner indépendamment ou être combinés afin de former un registre 16 bits. Cette particularité offrait des possibilités étendues pour le traitement des données et les calculs arithmétiques, en outre, le Motorola 6809 propose une large variété de modes d'adressage, notamment des modes indexés avancés, peu courants dans les architectures 8 bits concurrentes. Cette richesse fonctionnelle permettait une gestion efficace de la mémoire et une meilleure optimisation du code.

Enfin, grâce à une conception optimisée, de nombreuses instructions nécessitaient moins de cycles d'horloge, ce qui améliorait les performances globales du processeur tout en conservant une consommation maîtrisée.

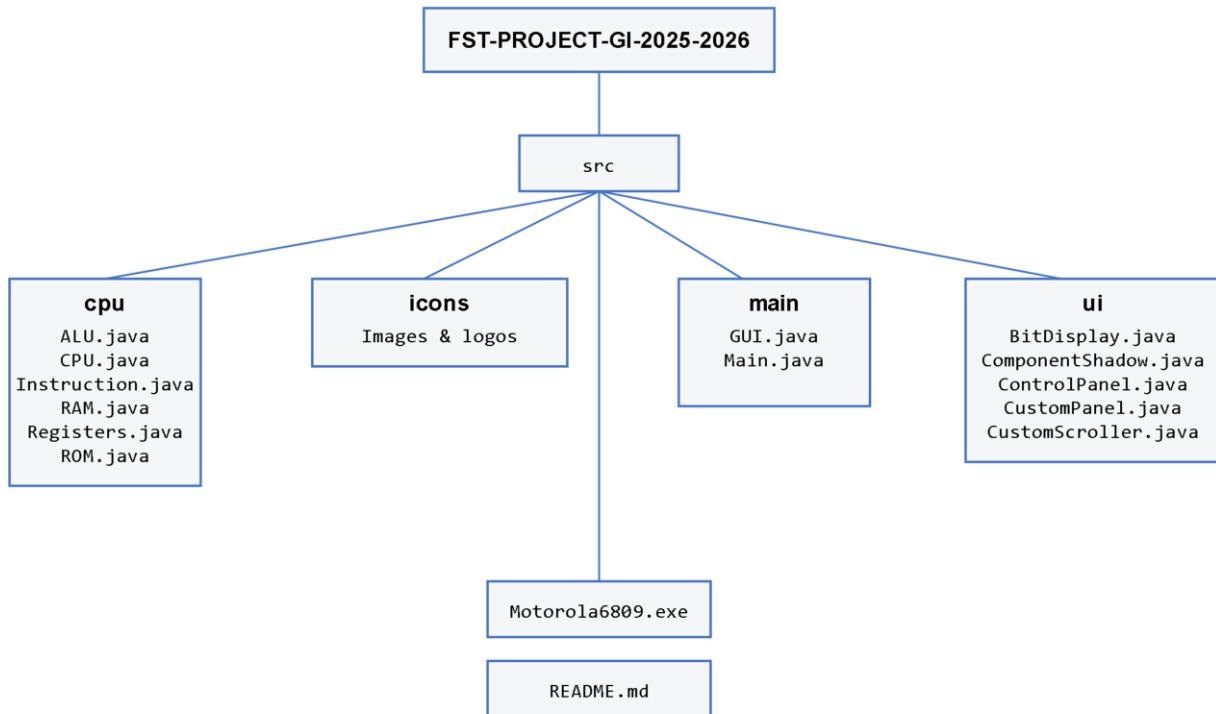
## Domaines d'utilisation et influence industrielle

Grâce à ses qualités techniques, le Motorola 6809 a été adopté dans plusieurs secteurs de l'industrie informatique et électronique. Il a notamment joué un rôle important dans le développement des premiers micro-ordinateurs destinés au grand public, en équipant des machines telles que le TRS-80 Color Computer ou les ordinateurs Dragon, contribuant ainsi à la démocratisation de l'informatique personnelle, par ailleurs, sa robustesse et sa flexibilité ont favorisé son intégration dans de nombreux systèmes embarqués et applications industrielles, incluant des dispositifs de contrôle, des équipements électroniques spécialisés et divers périphériques intelligents., et même si sa diffusion est restée plus limitée que celle de certains concurrents, le Motorola 6809 a exercé une influence durable sur l'évolution des architectures de processeurs et demeure aujourd'hui une référence technique étudiée dans le cadre de la rétro-informatique et de l'enseignement des systèmes numériques.

# **Architecture logicielle et conception du simulateur**

## Organisation générale et modules principaux

La conception du simulateur du processeur **Motorola 6809** repose sur une architecture logicielle modulaire, pensée pour assurer à la fois la clarté du code, la maintenabilité et la précision de la simulation. Chaque fonctionnalité du processeur est représentée par un ensemble de composants logiciels indépendants mais étroitement interconnectés, formant un système cohérent et robuste.



**Main** : Initialise l’interface graphique, crée tous les composants CPU (RAM, ROM, registres, ALU) et relie les boutons aux actions (exécuter, sauvegarder, gérer le programme).

**GUI** : Gère la fenêtre principale de l’émulateur Moto6809, affiche l’éditeur assembleur, les panneaux CPU, les boutons de contrôle, configure les couleurs, raccourcis clavier et interactions utilisateur.

**ALU** : Représente graphiquement l’Unité Arithmétique et Logique, permet d’afficher et mettre à jour opérandes et résultats, et d’exécuter des opérations hexadécimales.

**CPU** : Simule le processeur Motorola 6809, gère registres, mémoire et ALU, permet de charger, exécuter et contrôler un programme, en mettant à jour registres, drapeaux et mémoire.

**Instruction** : Représente une instruction du 6809, contient opcode et opérande, détecte le mode d’adressage, obtient le code hexadécimal, gère les registres, analyse les modes indexés et vérifie la syntaxe.

**RAM** : Gère l’affichage et manipulation de la mémoire vive, stocke les données clé-valeur, crée un panel pour chaque cellule, permet de générer, mettre à jour, réinitialiser la RAM et gérer les effets visuels au survol.

**ROM** : Gère l’affichage et manipulation des données ROM, crée un panel par cellule, permet de mettre à jour, surligner une cellule et ajouter des effets visuels au survol.

**Registers** : Gère l’affichage des registres, crée un panneau pour chaque registre avec nom et valeur, définit couleur et style, fournit un panneau prêt à afficher.

**BitDisplay** : Crée un panneau graphique pour afficher une valeur binaire, gère taille, position, style et fournit un JLabel centré.

**ComponentShadow** : Bordure personnalisée pour Swing avec ombre douce, définit marges, transparence et dessine un rectangle arrondi semi-transparent.

**ControlPanel** : Panneau de contrôle pour un registre ou contrôleur, affiche un label et une valeur avec JPanel stylisé (fond noir, bordure blanche).

**CustomPanel** : JPanel personnalisé affichant une image de fond, utilise paintComponent pour le dessin, gère transparence et couleur par défaut si l'image est absente.

**CustomScroller** : Personnalise JScrollPane, change couleurs du curseur et du track, supprime boutons par défaut et fixe une taille spécifique pour le curseur

## **Fonctionnalités principales**

### **Émulation du processeur Motorola 6809**

Le simulateur implémente une émulation fonctionnelle du processeur Motorola 6809 en interprétant et exécutant les instructions assembleur ligne par ligne.

Il gère les registres principaux (A, B, X, Y, U, S, PC, DP), les indicateurs d'état (N, Z, V, C, H) ainsi que l'ALU, permettant d'observer précisément l'évolution de l'état du processeur durant l'exécution.

L'exécution peut se faire pas à pas ou de manière continue, avec mise à jour en temps réel du compteur ordinal (PC), du registre d'instruction (RI) et des drapeaux

### **Gestion de la mémoire (RAM et ROM)**

Le simulateur distingue clairement deux espaces mémoire :

- **ROM** : contient le programme assembleur traduit en code machine hexadécimal. Les instructions sont sauvegardées et chargées séquentiellement avec gestion des adresses.
- **RAM** : utilisée pour stocker les données et résultats intermédiaires durant l'exécution.

Une interface graphique permet de visualiser et modifier le contenu de la RAM et de la ROM en temps réel.

Les accès mémoire sont contrôlés et synchronisés avec les instructions exécutées

### **Modes d'adressage supportés**

Le simulateur prend en charge plusieurs **modes d'adressage du 6809**, détectés automatiquement

- **Immédiat** (#valeur)
- **Direct**
- **Étendu**
- **Indexé** (avec registres X, Y, U, S et décalage juste le déplacement nulle)
- **Relatif** (branches avec labels)
- **Registre uniquement** (EXG, TFR)
- **Inhérent** (instructions sans opérande)

Chaque instruction est validée syntaxiquement avant exécution afin d'éviter les erreurs.

### **Instructions supportées**

Le simulateur implémente un ensemble représentatif d'instructions du Motorola 6809, notamment :

- **Chargement / stockage** : LD, ST

- Arithmétiques : ADD, SUB, CMP
- Logiques et binaires : COM, NEG, AND, OR, EOR
- Décalage et rotation : LSL, LSR, ROL, ROR, INC, DEC
- Contrôle de flot : JMP, RTS
- Branches conditionnelles : BEQ, BNE, BMI, BPL, BCC, BCS, BVC, BVS, BRA
- Transfert : TFR, EXG
- Divers : NOP, CLR, END, SWI

Chaque instruction met à jour correctement les registres et les flags concernés

## Débogage et visualisation

Le simulateur offre un environnement de **débogage visuel interactif** :

- Exécution **pas à pas** ou **continue**
- Affichage en temps réel des registres et flags
- Visualisation graphique de l'ALU et des valeurs binaires
- Suivi dynamique du contenu mémoire (RAM / ROM)

Ces fonctionnalités facilitent la compréhension du fonctionnement interne du processeur et l'analyse des programmes assembleur.

## Gestion des fichiers

Le simulateur permet :

- La création de nouveaux fichiers assembleur
- Le chargement et la sauvegarde de programmes (.asmb)
- La réinitialisation complète de l'environnement (CPU, RAM, ROM, registres)

## **Réalisation et organisation technique**

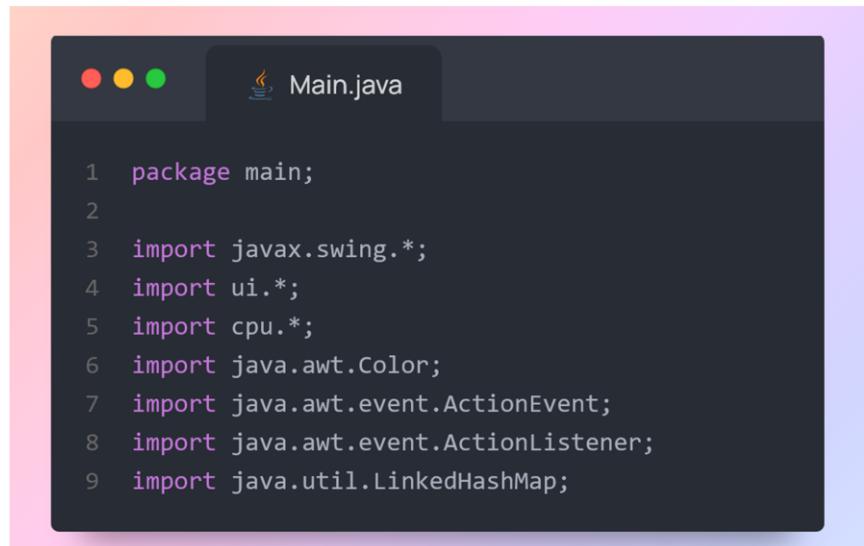
Dans la suite de ce document, nous procéderons à l'analyse et à l'explication des autres classes du projet. Nous détaillerons le rôle de chaque composant du simulateur, notamment les classes responsables de l'interface graphique (GUI, CustomPanel, BitDisplay, ControlPanel, CustomScroller), celles qui gèrent les éléments matériels simulés (CPU, Registers, RAM, ROM, ALU) ainsi que les interactions entre eux. Pour chaque classe, nous décrirons sa structure, ses méthodes principales et son fonctionnement, afin de comprendre comment elles contribuent à la simulation complète du microprocesseur Motorola 6809 et à l'expérience utilisateur finale. Cette analyse permettra de saisir la logique globale du programme et d'apprécier la manière dont les différents modules coopèrent pour reproduire le comportement d'un microprocesseur réel.

### Présentation du point d'entrée de l'application (*Main.java*)

La classe **Main.java** représente le point de démarrage de l'application de simulation du microprocesseur **Motorola 6809**, elle assure la mise en place de l'environnement général du programme et coordonne l'interaction entre les différents modules du simulateur.

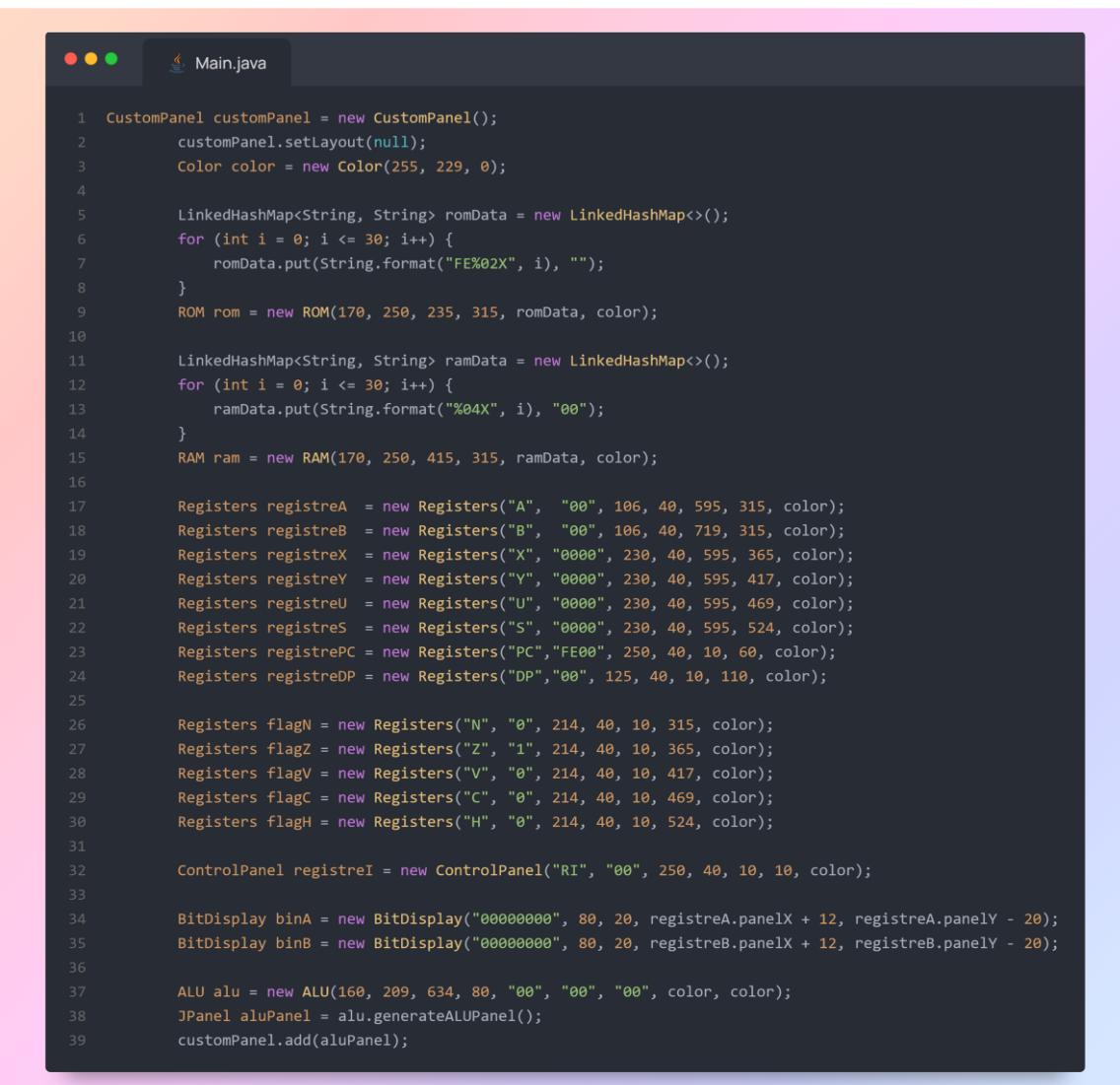
Cette classe initialise l'interface graphique, instancie les composants matériels simulés tels que le CPU, la RAM, la ROM, les registres et l'ALU, et établit le lien entre les éléments de l'interface et les actions associées, elle gère également le chargement, l'exécution et le contrôle des instructions assembleur,

garantissant ainsi le bon déroulement de la simulation, donc elle joue un rôle essentiel dans l'organisation globale de l'application en assurant la cohérence, la communication et le bon fonctionnement de l'ensemble du système simulé



```
1 package main;
2
3 import javax.swing.*;
4 import ui.*;
5 import cpu.*;
6 import java.awt.Color;
7 import java.awt.event.ActionEvent;
8 import java.awt.event.ActionListener;
9 import java.util.LinkedHashMap;
```

Ce code importe les bibliothèques nécessaires pour créer l'interface graphique et gérer les interactions utilisateur, utiliser des composants personnalisés (ui), manipuler les classes du processeur (cpu), gérer les couleurs, les événements comme les clics, et stocker des données dans des structures ordonnées comme LinkedHashMap, on va voir quelque chose comme ça dans tous les programmes ( packages/Library )



```
1 CustomPanel customPanel = new CustomPanel();
2     customPanel.setLayout(null);
3     Color color = new Color(255, 229, 0);
4
5     LinkedHashMap<String, String> romData = new LinkedHashMap<>();
6     for (int i = 0; i <= 30; i++) {
7         romData.put(String.format("FE%02X", i), "");
8     }
9     ROM rom = new ROM(170, 250, 235, 315, romData, color);
10
11    LinkedHashMap<String, String> ramData = new LinkedHashMap<>();
12    for (int i = 0; i <= 30; i++) {
13        ramData.put(String.format("%04X", i), "00");
14    }
15    RAM ram = new RAM(170, 250, 415, 315, ramData, color);
16
17    Registers registreA = new Registers("A", "00", 106, 40, 595, 315, color);
18    Registers registreB = new Registers("B", "00", 106, 40, 719, 315, color);
19    Registers registreX = new Registers("X", "0000", 230, 40, 595, 365, color);
20    Registers registreY = new Registers("Y", "0000", 230, 40, 595, 417, color);
21    Registers registreU = new Registers("U", "0000", 230, 40, 595, 469, color);
22    Registers registreS = new Registers("S", "0000", 230, 40, 595, 524, color);
23    Registers registrePC = new Registers("PC", "FE00", 250, 40, 10, 60, color);
24    Registers registreDP = new Registers("DP", "00", 125, 40, 10, 110, color);
25
26    Registers flagN = new Registers("N", "0", 214, 40, 10, 315, color);
27    Registers flagZ = new Registers("Z", "1", 214, 40, 10, 365, color);
28    Registers flagV = new Registers("V", "0", 214, 40, 10, 417, color);
29    Registers flagC = new Registers("C", "0", 214, 40, 10, 469, color);
30    Registers flagH = new Registers("H", "0", 214, 40, 10, 524, color);
31
32    ControlPanel registreI = new ControlPanel("RI", "00", 250, 40, 10, 10, color);
33
34    BitDisplay binA = new BitDisplay("00000000", 80, 20, registreA.panelX + 12, registreA.panelY - 20);
35    BitDisplay binB = new BitDisplay("00000000", 80, 20, registreB.panelX + 12, registreB.panelY - 20);
36
37    ALU alu = new ALU(160, 209, 634, 80, "00", "00", "00", color, color);
38    JPanel aluPanel = alu.generateALUPanel();
39    customPanel.add(aluPanel);
```

Le code crée un panneau personnalisé (CustomPanel) et initialise les composants d'un CPU simulé : la **ROM** et la **RAM** avec des données vides, plusieurs **registres** (A, B, X, Y, U, S, PC, DP) et les **flags** (N, Z, V, C, H), un panneau de contrôle (ControlPanel) et des affichages binaires pour A et B. Il configure également l'**ALU** avec son panneau graphique et l'ajoute au panneau principal, en utilisant une couleur uniforme pour tous les composants.



```
1     customPanel.add(ram.generateRAMPanel());
2     customPanel.add(rom.generateROMPanel());
3
4     customPanel.add(registreA.generateRegisterPanel());
5     customPanel.add(registreB.generateRegisterPanel());
6     customPanel.add(registreX.generateRegisterPanel());
7     customPanel.add(registreY.generateRegisterPanel());
8     customPanel.add(registreU.generateRegisterPanel());
9     customPanel.add(registreS.generateRegisterPanel());
10    customPanel.add(registrePC.generateRegisterPanel());
11    customPanel.add(registreDP.generateRegisterPanel());
12
13    customPanel.add(flagN.generateRegisterPanel());
14    customPanel.add(flagZ.generateRegisterPanel());
15    customPanel.add(flagV.generateRegisterPanel());
16    customPanel.add(flagC.generateRegisterPanel());
17    customPanel.add(flagH.generateRegisterPanel());
18
19    customPanel.add(registreI.generateControPanel());
20    customPanel.add(binA.generateBitPanel());
21    customPanel.add(binB.generateBitPanel());
22
23
24
25    CPU cpu = new CPU(
26        ram, rom,
27        registreA, registreB,
28        registreX, registreY,
29        registreU, registreS,
30        registrePC, registreDP,
31        registreI,
32        flagN, flagZ, flagV, flagC, flagH,
33        alu,
34        binA, binB
35    );
```

Le code ajoute tous les panneaux générés de la RAM, ROM, registres, flags, contrôles et affichages binaires au panneau principal (customPanel) puis crée un objet CPU en passant tous ces composants pour les relier et permettre leur interaction dans la simulation du processeur.



```
1  SwingUtilities.invokeLater(() -> new GUI(customPanel, new ActionListener() {
2      @Override
3      public void actionPerformed(ActionEvent e) {
4
5          GUI gui = (GUI) ((JButton) e.getSource()).getTopLevelAncestor();
6          String actionCommand = e.getActionCommand();
7
8          cpu.initializeLines(gui);
9
10         switch (actionCommand) {
11             case "RUN":  cpu.executeAll(gui);    break;
12             case "STEP": cpu.executeStep(gui);   break;
13             case "SAVE": cpu.saveProgram(gui);  break;
14             case "CLEAR":cpu.clearProgram(gui); break;
15             case "NEW":  cpu.createNewFile(gui); break;
16             case "OPEN": cpu.openFile(gui);    break;
17             case "EXIT": System.exit(0);       break;
18         }
19     });
20 });
21 }
```

Le code utilise `SwingUtilities.invokeLater` pour lancer l'interface graphique avec le panneau principal, puis définit un **écouteur d'actions** pour les boutons. Selon le bouton cliqué (RUN, STEP, SAVE, CLEAR, NEW, OPEN, EXIT), il appelle les méthodes correspondantes du CPU pour exécuter, sauvegarder, nettoyer, créer ou ouvrir un programme, ou quitter l'application.

## Présentation du l'interface graphique principale (GUI.java)



```
1  public JTextArea codeEditor;
2  public JTextArea notesArea;
3  public JButton btnRun, btnStep, btnSave;
4  public JButton btnNew, btnOpen, btnExit;
5  public int introDuration = 11;
6
7  public GUI(JPanel cpuPanel, ActionListener sharedListener) {
8
9      setSize(1250, 650);
10     setTitle("Moto6809 Emulator");
11     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12     setLayout(new BorderLayout());
13     setUndecorated(true);
14     setLocationRelativeTo(null);
15
16     ImageIcon appIcon = new ImageIcon(
17         getClass().getResource("/icons/logo.png")
18     );
19     setIconImage(appIcon.getImage());
20
21     Color background  = new Color(0, 0, 0);
22     Color foreground  = Color.WHITE;
23     Color runColor   = new Color(0, 153, 76);
24     Color stepColor  = new Color(255, 153, 51);
25     Color saveColor  = new Color(0, 102, 204);
26     Color newColor   = new Color(153, 51, 255);
27     Color exitColor  = new Color(255, 51, 51);
28     Color openColor  = new Color(102, 102, 102);
29     Color buttonHover = new Color(40, 40, 40);
30     Color accent     = new Color(72, 255, 21);
31
32     JPanel rootPanel = new JPanel(new BorderLayout());
33     rootPanel.setBackground(background);
34
35     JPanel editorPanel = new JPanel(new BorderLayout());
36     editorPanel.setBackground(background);
37     editorPanel.setPreferredSize(
38         new Dimension((int) (getWidth() * 0.3), getHeight())
39     );
39     editorPanel.setBorder(new EmptyBorder(20, 20, 20, 0));
40
41     codeEditor = new JTextArea();
42     codeEditor.setBackground(new Color(20, 20, 20));
43     codeEditor.setForeground(foreground);
44     codeEditor.setFont(new Font("Source Code Pro", Font.PLAIN, 20));
45     codeEditor.setCaretColor(Color.RED);
46     codeEditor.setBorder(
47         BorderFactory.createLineBorder(accent, 2, true)
48     );
49     codeEditor.setLineWrap(true);
50
51     codeEditor.addFocusListener(new java.awt.event.FocusAdapter() {
52         @Override
53         public void focusGained(java.awt.event.FocusEvent e) {
54             codeEditor.setBorder(
55                 BorderFactory.createLineBorder(accent, 2, true)
56             );
57         }
58     });
59 }
```

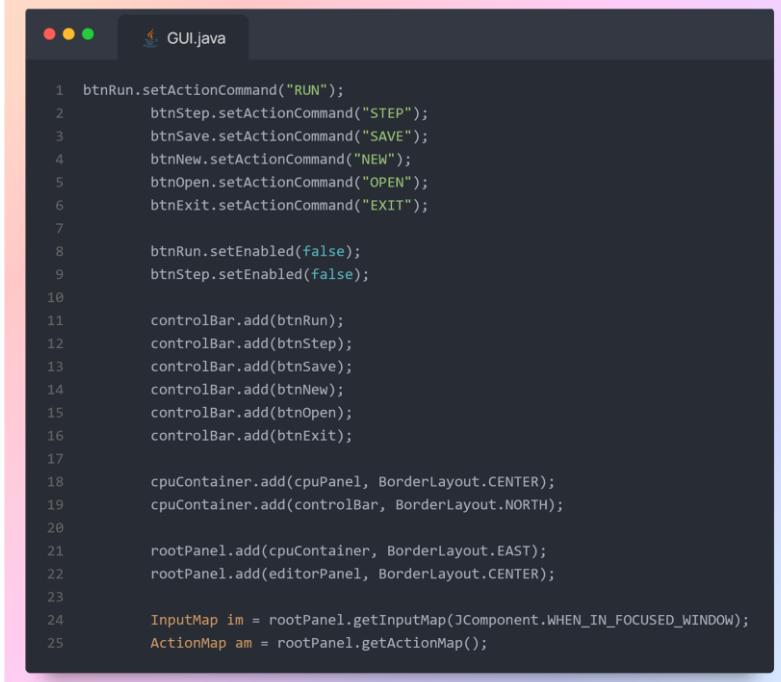
Le code crée une fenêtre JFrame pour l'émulateur Moto6809, définit sa taille, titre, icône et couleurs principales. Il initialise un panneau principal (rootPanel) et un panneau pour l'éditeur de code (editorPanel). Ensuite, il configure un **éditeur de code (JTextArea)** avec fond sombre, texte blanc, police spécifique, couleur du curseur, bordure colorée et gestion du focus pour que la bordure reste accentuée lorsqu'on clique dedans.

```

1  @Override
2      public void focusLost(java.awt.event.FocusEvent e) {
3          codeEditor.setBorder(
4              BorderFactory.createLineBorder(accent, 2, true)
5          );
6      }
7  });
8
9  JScrollPane editorScroll = new JScrollPane(codeEditor);
10 editorScroll.setBorder(BorderFactory.createEmptyBorder());
11 editorScroll.getVerticalScrollBar().setUI(new CustomScroller());
12
13 editorPanel.add(editorScroll, BorderLayout.CENTER);
14
15 notesArea = new JTextArea();
16 notesArea.setEditable(false);
17 notesArea.setBackground(new Color(10, 10, 10));
18 notesArea.setForeground(new Color(180, 180, 180));
19 notesArea.setFont(new Font("Segoe UI", Font.ITALIC, 13));
20 notesArea.setLineWrap(true);
21 notesArea.setWrapStyleWord(true);
22 notesArea.setMargin(new Insets(8, 10, 8, 10));
23
24 notesArea.setText(
25         "Ce simulateur a été développé par Youssef Fissal et Youness Irida "
26         + "dans le cadre du projet Module d'Architecture des Ordinateurs "
27         + "pour simuler le microprocesseur Motorola 6809.\n"
28         + "Faculté des Sciences et Techniques de Settat (FSTS), "
29         + "Université Hassan Ier.\n"
30         + "© Tous droits réservés 2025."
31 );
32
33 JScrollPane notesScroll = new JScrollPane(notesArea);
34 notesScroll.setPreferredSize(new Dimension(0, 120));
35 notesScroll.setBorder(
36     BorderFactory.createLineBorder(accent, 1, true)
37 );
38 notesScroll.getVerticalScrollBar().setUI(new CustomScroller());
39
40 editorPanel.add(notesScroll, BorderLayout.SOUTH);
41
42 JPanel cpuContainer = new JPanel(new BorderLayout());
43 cpuContainer.setBackground(background);
44 cpuContainer.setPreferredSize(
45     new Dimension((int) (getWidth() * 0.7), getHeight())
46 );
47 cpuContainer.setBorder(new EmptyBorder(20, 20, 20, 20));
48
49 JPanel controlBar = new JPanel(new GridLayout(1, 6, 18, 5));
50 controlBar.setBackground(background);
51 controlBar.setBorder(new EmptyBorder(0, 0, 10, 0));
52
53 btnRun = createControlButton("Exécuter ▶", runColor, foreground, buttonHover, accent, sharedListener);
54 btnStep = createControlButton("Pas à Pas ⏪", stepColor, foreground, buttonHover, accent, sharedListener);
55 btnSave = createControlButton("Enregistrer ✓", saveColor, foreground, buttonHover, accent, sharedListener);
56 btnNew = createControlButton("Exporter 📁", newColor, foreground, buttonHover, accent, sharedListener);
57 btnOpen = createControlButton("Importer 📁", openColor, foreground, buttonHover, accent, sharedListener);
58 btnExit = createControlButton("Quitter ✕", exitColor, foreground, buttonHover, accent, sharedListener);

```

Le code configure le comportement lorsque l'éditeur de code perd le focus pour garder la bordure accentuée, ajoute des **barres de défilement personnalisées** pour le code et les notes (CustomScroller), crée une zone de notes (notesArea) non modifiable avec des informations sur le projet et son auteur, et prépare le panneau du CPU ainsi qu'une **barre de contrôle** avec les boutons principaux (Exécuter, Pas à Pas, Enregistrer, Exporter, Importer, Quitter) avec couleurs, effets au survol et écouteur partagé pour gérer les actions.



```
1 btnRun.setActionCommand("RUN");
2     btnStep.setActionCommand("STEP");
3     btnSave.setActionCommand("SAVE");
4     btnNew.setActionCommand("NEW");
5     btnOpen.setActionCommand("OPEN");
6     btnExit.setActionCommand("EXIT");
7
8     btnRun.setEnabled(false);
9     btnStep.setEnabled(false);
10
11    controlBar.add(btnRun);
12    controlBar.add(btnStep);
13    controlBar.add(btnSave);
14    controlBar.add(btnNew);
15    controlBar.add(btnOpen);
16    controlBar.add(btnExit);
17
18    cpuContainer.add(cpuPanel, BorderLayout.CENTER);
19    cpuContainer.add(controlBar, BorderLayout.NORTH);
20
21    rootPanel.add(cpuContainer, BorderLayout.EAST);
22    rootPanel.add(editorPanel, BorderLayout.CENTER);
23
24    InputMap im = rootPanel.getInputMap(JComponent.WHEN_IN_FOCUSED_WINDOW);
25    ActionMap am = rootPanel.getActionMap();
```

Le code assigne des **commandes d'action** à chaque bouton (RUN, STEP, SAVE, NEW, OPEN, EXIT) et désactive temporairement RUN et STEP. Ensuite, il ajoute les boutons à la **barre de contrôle**, place le panneau CPU et la barre de contrôle dans le cpuContainer, puis organise les panneaux principaux dans rootPanel. Enfin, il prépare les **cartes d'entrée et d'action** (InputMap et ActionMap) pour permettre les raccourcis clavier.



```
1 im.put(KeyboardStroke.getKeyStroke("control S"), "SAVE");
2     am.put("SAVE", new AbstractAction() {
3         @Override
4             public void actionPerformed(ActionEvent e) {
5                 btnSave.doClick();
6             }
7         });
8
9     im.put(KeyboardStroke.getKeyStroke("control X"), "RUN");
10    am.put("RUN", new AbstractAction() {
11        @Override
12            public void actionPerformed(ActionEvent e) {
13                btnRun.doClick();
14            }
15        });
16
17    im.put(KeyboardStroke.getKeyStroke("ENTER"), "STEP");
18    am.put("STEP", new AbstractAction() {
19        @Override
20            public void actionPerformed(ActionEvent e) {
21                btnStep.doClick();
22            }
23        });
24
25    im.put(KeyboardStroke.getKeyStroke("control N"), "NEW");
26    am.put("NEW", new AbstractAction() {
27        @Override
28            public void actionPerformed(ActionEvent e) {
29                btnNew.doClick();
30            }
31        });
32
33    im.put(KeyboardStroke.getKeyStroke("control O"), "OPEN");
34    am.put("OPEN", new AbstractAction() {
35        @Override
36            public void actionPerformed(ActionEvent e) {
37                btnOpen.doClick();
38            }
39        });
40
41    im.put(KeyboardStroke.getKeyStroke("ESCAPE"), "EXIT");
42    am.put("EXIT", new AbstractAction() {
43        @Override
44            public void actionPerformed(ActionEvent e) {
45                btnExit.doClick();
46            }
47        });
48
49    add(rootPanel);
50    setVisible(true);
51 }
```

Le code associe des **raccourcis clavier** aux boutons de l'interface : Ctrl+S pour SAVE, Ctrl+X pour RUN, Entrée pour STEP, Ctrl+N pour NEW, Ctrl+O pour OPEN et Échap pour EXIT. Chaque raccourci déclenche virtuellement le clic du bouton correspondant avec doClick(). Enfin, le panneau principal (rootPanel) est ajouté à la fenêtre et rendu visible.



The screenshot shows a Java code editor with the file name "GUI.java" at the top. The code is a Java class for creating a JButton with custom painting logic. It overrides the paintComponent method to draw a rounded rectangle with a black border and white fill, with the text centered inside. It also handles mouse events by repainting the button. The code uses Java's AWT and Swing components like JButton, Graphics2D, and FontMetrics.

```
1 private JButton createControlButton(
2         String label,
3         Color base,
4         Color text,
5         Color hover,
6         Color accent,
7         ActionListener listener
8     ) {
9
10    JButton btn = new JButton(label) {
11        @Override
12        protected void paintComponent(Graphics g) {
13
14            Graphics2D g2 = (Graphics2D) g.create();
15            g2.setRenderingHint(
16                RenderingHints.KEY_ANTIALIASING,
17                RenderingHints.VALUE_ANTIALIAS_ON
18            );
19
20            ButtonModel model = getModel();
21
22            if (!model.isEnabled()) g2.setColor(base.darker());
23            else if (model.isPressed()) g2.setColor(accent.darker());
24            else if (model.isRollover()) g2.setColor(hover);
25            else g2.setColor(base);
26
27            g2.fillRoundRect(
28                0, 0, getWidth(), getHeight(),
29                getHeight(), getHeight()
30            );
31
32            g2.setColor(
33                model.isPressed() ? Color.BLACK :
34                model.isRollover() ? accent : text
35            );
36
37            FontMetrics fm = g2.getFontMetrics();
38            int stringWidth = fm.stringWidth(getText());
39            int stringHeight = fm.getAscent();
40
41            g2.drawString(
42                getText(),
43                (getWidth() - stringWidth) / 2,
44                (getHeight() + stringHeight) / 2 - 3
45            );
46
47            g2.dispose();
48        }
49    };
50
51    btn.setRolloverEnabled(true);
52    btn.setFocusPainted(false);
53    btn.setContentAreaFilled(false);
54    btn.setBorderPainted(false);
55    btn.setCursor(new Cursor(Cursor.HAND_CURSOR));
56    btn.addActionListener(listener);
57
58    btn.addMouseListener(new java.awt.event.MouseAdapter() {
59        @Override
60        public void mouseEntered(java.awt.event.MouseEvent e) {
61            btn.repaint();
62        }
63
64        @Override
65        public void mouseExited(java.awt.event.MouseEvent e) {
66            btn.repaint();
67        }
68    });
69
70    return btn;
71 }
```

La méthode `createControlButton` crée un **JButton personnalisé** avec coins arrondis, couleurs dynamiques selon l'état du bouton (désactivé, pressé, survolé) et texte centré. Elle désactive le remplissage et les bordures par défaut, change le curseur en main, ajoute un **écouteur d'action** pour les clics et un **MouseListener** pour redessiner le bouton au survol. Cela permet d'avoir des boutons stylisés et interactifs dans l'interface.

## Présentation des composants de l'interface utilisateur (Package ui)

### Le composant du display des bits (BitDisplay.java)



```
1 package ui;
2
3 import javax.swing.*;
4 import java.awt.*;
5
6 public class BitDisplay {
7
8     public String binaryValue;
9     public int panelWidth;
10    public int panelHeight;
11    public int posX;
12    public int posY;
13    public JLabel valueLabel;
14
15    public BitDisplay(String binaryValue, int width, int height, int x, int y) {
16        this.binaryValue = binaryValue;
17        this.panelWidth = width;
18        this.panelHeight = height;
19        this.posX = x;
20        this.posY = y;
21    }
22
23    public JPanel generateBitPanel() {
24
25        JPanel panel = new JPanel();
26        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
27        panel.setPreferredSize(new Dimension(panelWidth, panelHeight));
28        panel.setBounds(posX, posY, panelWidth, panelHeight);
29        panel.setBackground(new Color(6, 26, 83));
30        panel.setAlignmentX(Component.CENTER_ALIGNMENT);
31        panel.setAlignmentY(Component.CENTER_ALIGNMENT);
32
33        valueLabel = new JLabel(binaryValue, SwingConstants.CENTER);
34        valueLabel.setForeground(Color.WHITE);
35        valueLabel.setFont(new Font("Roboto", Font.BOLD, 12));
36        valueLabel.setOpaque(false);
37        valueLabel.setPreferredSize(new Dimension(panelWidth, 30));
38        valueLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
39
40        panel.add(valueLabel);
41
42        return panel;
43    }
44}
```

La classe BitDisplay sert à créer un **élément graphique pour afficher une valeur binaire** dans l’interface du CPU. Elle stocke la valeur binaire (binaryValue), les dimensions du panneau (panelWidth, panelHeight) et sa position (posX, posY). La méthode generateBitPanel crée un **JPanel** avec une couleur de fond bleu foncé et un BoxLayout vertical, puis ajoute un **JLabel centré** qui affiche la valeur binaire en blanc avec une police en gras. Le panneau et le label sont alignés au centre et adaptés à la taille spécifiée, ce qui permet de représenter visuellement des registres ou bits d’une manière claire et lisible dans l’interface graphique.

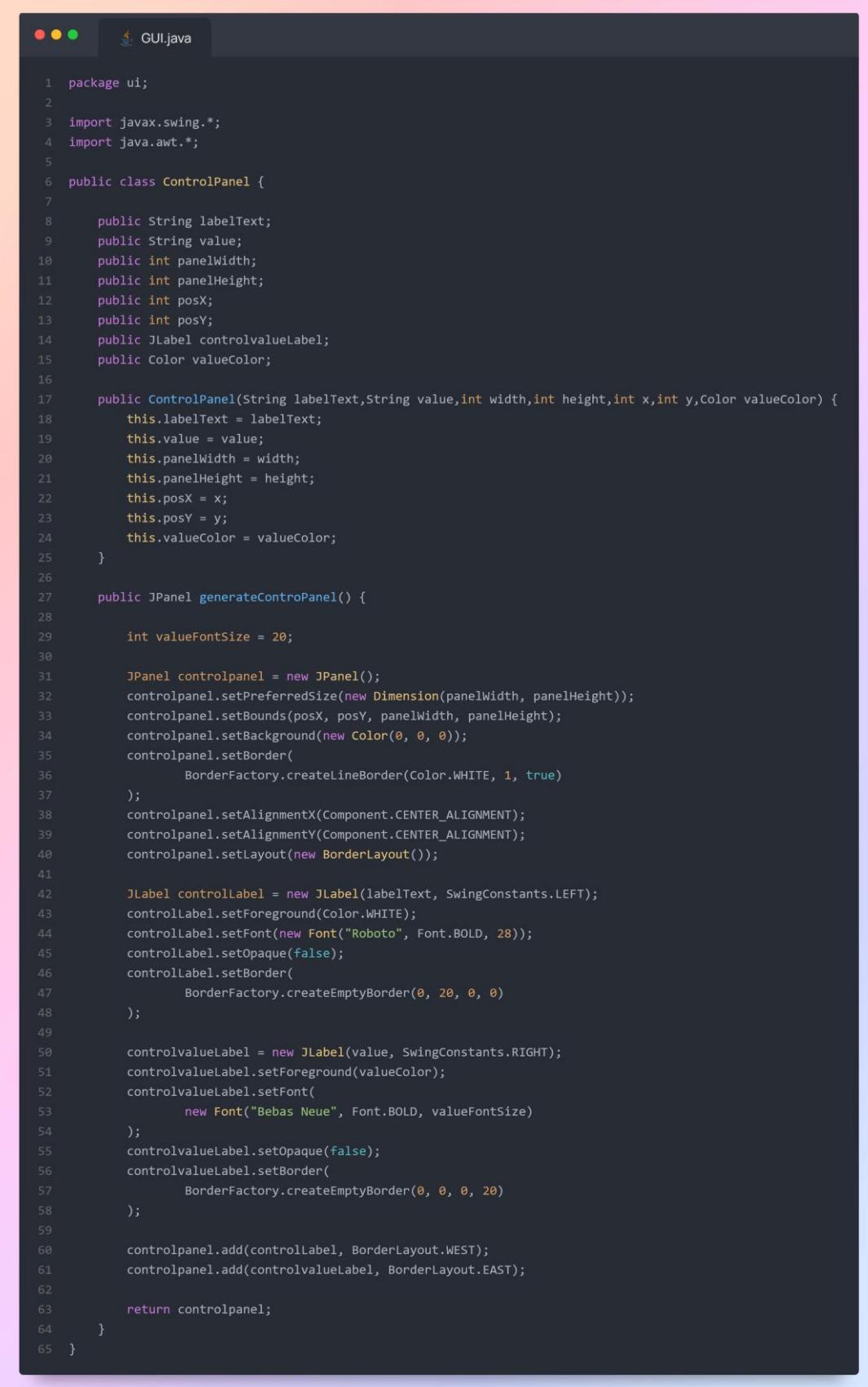
### [Le composant effet d’ombre \(ComponentShadow.java\)](#)



```
1 package ui;
2
3 import javax.swing.border.Border;
4 import java.awt.*;
5
6 public class ComponentShadow implements Border {
7
8     @Override
9     public Insets getBorderInsets(Component composant) {
10         return new Insets(10, 10, 10, 10);
11     }
12
13     @Override
14     public boolean isBorderOpaque() {
15         return false;
16     }
17
18     @Override
19     public void paintBorder(
20             Component composant,
21             Graphics g,
22             int x,
23             int y,
24             int largeur,
25             int hauteur
26     ) {
27
28         Graphics2D g2d = (Graphics2D) g;
29         g2d.setRenderingHint(
30             RenderingHints.KEY_ANTIALIASING,
31             RenderingHints.VALUE_ANTIALIAS_ON
32         );
33
34         g2d.setColor(new Color(0, 0, 0, 80));
35
36         g2d.fillRoundRect(
37             x + 5,
38             y + 5,
39             largeur - 10,
40             hauteur - 10,
41             10,
42             10
43         );
44     }
45 }
```

La classe ComponentShadow implémente l’interface Border pour créer un **effet d’ombre autour d’un composant Swing**. Elle définit des **marges internes** de 10 pixels (getBorderInsets) et précise que la bordure n’est pas opaque (isBorderOpaque). La méthode paintBorder utilise Graphics2D avec **antialiasing** pour dessiner un **rectangle arrondi semi-transparent** derrière le composant, créant un effet de profondeur et de relief visuel qui rend l’interface plus esthétique.

### Le composant panneau graphique du control (ControlPanel.java)



```
1 package ui;
2
3 import javax.swing.*;
4 import java.awt.*;
5
6 public class ControlPanel {
7
8     public String labelText;
9     public String value;
10    public int panelWidth;
11    public int panelHeight;
12    public int posX;
13    public int posY;
14    public JLabel controlvalueLabel;
15    public Color valueColor;
16
17    public ControlPanel(String labelText, String value, int width, int height, int x, int y, Color valueColor) {
18        this.labelText = labelText;
19        this.value = value;
20        this.panelWidth = width;
21        this.panelHeight = height;
22        this.posX = x;
23        this.posY = y;
24        this.valueColor = valueColor;
25    }
26
27    public JPanel generateControPanel() {
28
29        int valueFontSize = 20;
30
31        JPanel controlpanel = new JPanel();
32        controlpanel.setPreferredSize(new Dimension(panelWidth, panelHeight));
33        controlpanel.setBounds(posX, posY, panelWidth, panelHeight);
34        controlpanel.setBackground(new Color(0, 0, 0));
35        controlpanel.setBorder(
36            BorderFactory.createLineBorder(Color.WHITE, 1, true)
37        );
38        controlpanel.setAlignmentX(Component.CENTER_ALIGNMENT);
39        controlpanel.setAlignmentY(Component.CENTER_ALIGNMENT);
40        controlpanel.setLayout(new BorderLayout());
41
42        JLabel controlLabel = new JLabel(labelText, SwingConstants.LEFT);
43        controlLabel.setForeground(Color.WHITE);
44        controlLabel.setFont(new Font("Roboto", Font.BOLD, 28));
45        controlLabel.setOpaque(false);
46        controlLabel.setBorder(
47            BorderFactory.createEmptyBorder(0, 20, 0, 0)
48        );
49
50        controlvalueLabel = new JLabel(value, SwingConstants.RIGHT);
51        controlvalueLabel.setForeground(valueColor);
52        controlvalueLabel.setFont(
53            new Font("Bebas Neue", Font.BOLD, valueFontSize)
54        );
55        controlvalueLabel.setOpaque(false);
56        controlvalueLabel.setBorder(
57            BorderFactory.createEmptyBorder(0, 0, 0, 20)
58        );
59
60        controlpanel.add(controlLabel, BorderLayout.WEST);
61        controlpanel.add(controlvalueLabel, BorderLayout.EAST);
62
63        return controlpanel;
64    }
65}
```

La classe ControlPanel permet de créer un **panneau graphique** affichant un label et sa valeur associée. Elle stocke le texte du label (labelText), la valeur (value), les dimensions et la position du panneau, ainsi qu'une couleur pour la valeur (valueColor). La méthode generateControPanel crée un JPanel avec **bordure blanche**, fond noir et disposition BorderLayout. Elle ajoute un **JLabel à gauche** pour le texte du label et un **JLabel à droite** pour la valeur, avec des polices et couleurs personnalisées. Ce panneau sert à afficher visuellement des informations comme les registres ou contrôles du CPU dans l'interface graphique de manière claire et esthétique.

### Le composant panneau graphique principale (CustomPanel.java)



```
1 package ui;
2
3 import javax.swing.*;
4 import java.awt.*;
5
6 public class CustomPanel extends JPanel {
7
8     private Image backgroundImage;
9
10    public CustomPanel() {
11        setOpaque(false);
12       setLayout(null);
13        ImageIcon icon = new ImageIcon(
14            getClass().getResource("/icons/Background.png")
15        );
16        backgroundImage = icon.getImage();
17    }
18
19    @Override
20    protected void paintComponent(Graphics g) {
21        super.paintComponent(g);
22
23        if (backgroundImage != null) {
24            g.drawImage(
25                backgroundImage,
26                0,
27                0,
28                getWidth(),
29                getHeight(),
30                this
31            );
32        } else {
33            g.setColor(Color.LIGHT_GRAY);
34            g.fillRect(0, 0, getWidth(), getHeight());
35        }
36    }
37 }
```

La classe CustomPanel étend JPanel pour créer un **panneau avec une image de fond**. Dans le constructeur, elle charge une image (Background.png) depuis les ressources et rend le panneau non opaque avec un layout nul (null) pour un positionnement libre des composants. La méthode paintComponent redessine le panneau : si l'image de fond est disponible, elle est étirée pour remplir tout le panneau, sinon le panneau est rempli avec une couleur grise claire. Cela permet d'avoir une interface visuelle plus attractive pour l'émulateur CPU.

## Le composant scrollbar stylisé (CustomScroller.java)

```
1  public int panelWidth;
2  public int panelHeight;
3  public int posX;
4  public int posY;
5  public LinkedHashMap<String, String> RamMemoryData;
6  public ArrayList<JPanel> cellPanels;
7  public JPanel containerPanel;
8  public Color highlightColor;
9
10 public RAM(int width, int height, int x, int y, LinkedHashMap<String, String> data, Color color) {
11     this.panelWidth = width;
12     this.panelHeight = height;
13     this.posX = x;
14     this.posY = y;
15     this.RamMemoryData = data;
16     this.cellPanels = new ArrayList<>();
17     this.highlightColor = color;
18 }
19
20 public JPanel generateRAMPPanel() {
21     JPanel memoryPanel = new JPanel();
22     memoryPanel.setLayout(new BorderLayout());
23     memoryPanel.setPreferredSize(new Dimension(panelWidth, panelHeight));
24     memoryPanel.setBounds(posX, posY, panelWidth, panelHeight);
25     memoryPanel.setBackground(new Color(6, 26, 83));
26     memoryPanel.setBorder(new ComponentShadow());
27
28     JLabel headerLabel = new JLabel("RAM", SwingConstants.CENTER);
29     headerLabel.setForeground(Color.WHITE);
30     headerLabel.setFont(new Font("Roboto Black", Font.BOLD, 22));
31     memoryPanel.add(headerLabel, BorderLayout.NORTH);
32
33     containerPanel = new JPanel();
34     containerPanel.setLayout(new BoxLayout(containerPanel, BoxLayout.Y_AXIS));
35     containerPanel.setBackground(new Color(6, 26, 83));
36     containerPanel.setBorder(BorderFactory.createEmptyBorder(10, 0, 0, 0));
37
38     for (Map.Entry<String, String> entry : RamMemoryData.entrySet()) {
39         JPanel memoryCell = new JPanel();
40         memoryCell.setLayout(new BorderLayout());
41         memoryCell.setBackground(new Color(6, 26, 83));
42         memoryCell.setBorder(BorderFactory.createEmptyBorder(2, 10, 2, 10));
43
44         JLabel addressLabel = new JLabel(entry.getKey(), SwingConstants.LEFT);
45         addressLabel.setForeground(Color.WHITE);
46         addressLabel.setFont(new Font("Bebas Neue", Font.PLAIN, 14));
47         memoryCell.add(addressLabel, BorderLayout.WEST);
48
49         JLabel valueLabel = new JLabel(entry.getValue(), SwingConstants.RIGHT);
50         valueLabel.setForeground(Color.WHITE);
51         valueLabel.setFont(new Font("Bebas Neue", Font.PLAIN, 14));
52         memoryCell.add(valueLabel, BorderLayout.EAST);
53
54         addHoverEffect(addressLabel, valueLabel);
55
56         containerPanel.add(memoryCell);
57
58         JPanel separator = new JPanel();
59         separator.setPreferredSize(new Dimension(0, 1));
60         separator.setMaximumSize(new Dimension(Integer.MAX_VALUE, 1));
61         separator.setBackground(Color.WHITE);
62         containerPanel.add(separator);
63
64         cellPanels.add(memoryCell);
65     }
66
67     JScrollPane scrollPane = new JScrollPane(containerPanel);
68     scrollPane.setBorder(BorderFactory.createEmptyBorder());
69     scrollPane.setBackground(new Color(40, 40, 40));
70     scrollPane.getVerticalScrollBar().setUI(new CustomScroller());
71     scrollPane.getVerticalScrollBar().setUnitIncrement(6);
72     scrollPane.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
73     memoryPanel.add(scrollPane, BorderLayout.CENTER);
74
75     return memoryPanel;
76 }
```

La classe CustomScroller étend BasicScrollBarUI pour créer un **scrollbar stylisé**. Elle définit la couleur du **curseur (thumb)** en jaune et la piste (track) en noir, supprime les boutons de défilement visibles en les rendant invisibles, et fixe la taille minimale et maximale du curseur pour qu'il soit étroit et uniforme. De plus, la méthode getPreferredSize ajuste l'épaisseur du scrollbar selon son orientation (verticale ou horizontale), ce qui donne un aspect plus moderne et épuré à l'interface graphique.

### Le composant panneau graphique du RAM (RAM.java)

```
CustomScroller.java

1  public int panelWidth;
2  public int panelHeight;
3  public int posX;
4  public int posY;
5  public LinkedHashMap<String, String> RamMemoryData;
6  public ArrayList<Panel> cellPanels;
7  public JPanel containerPanel;
8  public Color highlightColor;
9
10 public RAM(int width, int height, int x, int y, LinkedHashMap<String, String> data, Color color) {
11     this.panelWidth = width;
12     this.panelHeight = height;
13     this.posX = x;
14     this.posY = y;
15     this.RamMemoryData = data;
16     this.cellPanels = new ArrayList<>();
17     this.highlightColor = color;
18 }
19
20 public JPanel generateRAMPanel() {
21     JPanel memoryPanel = new JPanel();
22     memoryPanel.setLayout(new BorderLayout());
23     memoryPanel.setPreferredSize(new Dimension(panelWidth, panelHeight));
24     memoryPanel.setBounds(posX, posY, panelWidth, panelHeight);
25     memoryPanel.setBackground(new Color(6, 26, 83));
26     memoryPanel.setBorder(new ComponentShadow());
27
28     JLabel headerLabel = new JLabel("RAM", SwingConstants.CENTER);
29     headerLabel.setForeground(Color.WHITE);
30     headerLabel.setFont(new Font("Roboto Black", Font.BOLD, 22));
31     memoryPanel.add(headerLabel, BorderLayout.NORTH);
32
33     containerPanel = new JPanel();
34     containerPanel.setLayout(new BoxLayout(containerPanel, BoxLayout.Y_AXIS));
35     containerPanel.setBackground(new Color(6, 26, 83));
36     containerPanel.setBorder(BorderFactory.createEmptyBorder(10, 0, 0, 0));
37
38     for (Map.Entry<String, String> entry : RamMemoryData.entrySet()) {
39         JPanel memoryCell = new JPanel();
40         memoryCell.setLayout(new BorderLayout());
41         memoryCell.setBackground(new Color(6, 26, 83));
42         memoryCell.setBorder(BorderFactory.createEmptyBorder(2, 10, 2, 10));
43
44         JLabel addressLabel = new JLabel(entry.getKey(), SwingConstants.LEFT);
45         addressLabel.setForeground(Color.WHITE);
46         addressLabel.setFont(new Font("Bebas Neue", Font.PLAIN, 14));
47         memoryCell.add(addressLabel, BorderLayout.WEST);
48
49         JLabel valueLabel = new JLabel(entry.getValue(), SwingConstants.RIGHT);
50         valueLabel.setForeground(Color.WHITE);
51         valueLabel.setFont(new Font("Bebas Neue", Font.PLAIN, 14));
52         memoryCell.add(valueLabel, BorderLayout.EAST);
53
54         addHoverEffect(addressLabel, valueLabel);
55
56         containerPanel.add(memoryCell);
57
58         JPanel separator = new JPanel();
59         separator.setPreferredSize(new Dimension(0, 1));
60         separator.setMaximumSize(new Dimension(Integer.MAX_VALUE, 1));
61         separator.setBackground(Color.WHITE);
62         containerPanel.add(separator);
63
64         cellPanels.add(memoryCell);
65     }
66
67     JScrollPane scrollPane = new JScrollPane(containerPanel);
68     scrollPane.setBorder(BorderFactory.createEmptyBorder());
69     scrollPane.setBackground(new Color(40, 40, 40));
70     scrollPane.getVerticalScrollBar().setUI(new CustomScroller());
71     scrollPane.getVerticalScrollBar().setUnitIncrement(6);
72     scrollPane.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
73     memoryPanel.add(scrollPane, BorderLayout.CENTER);
74
75     return memoryPanel;
76 }
```

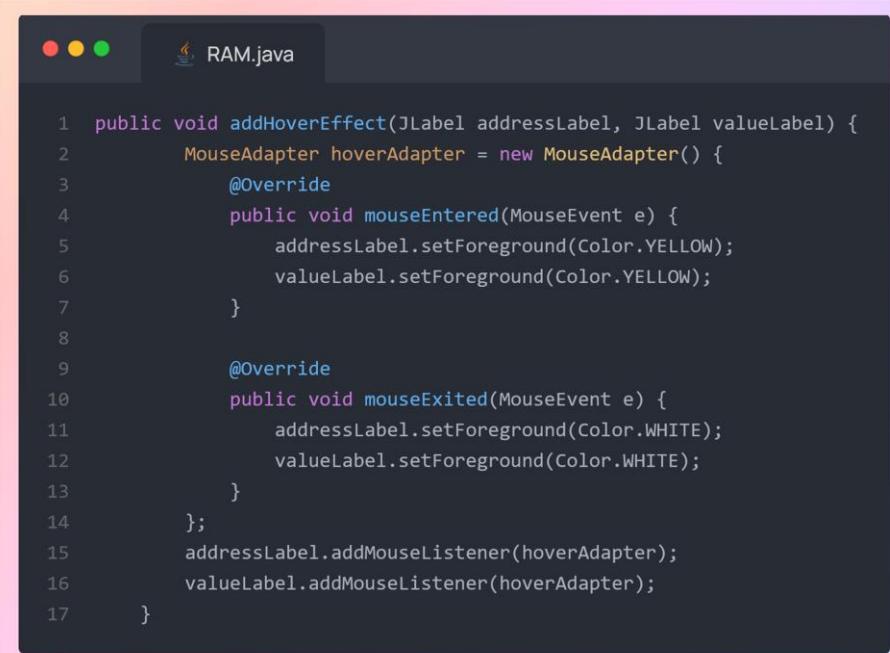
La classe RAM crée un **panneau graphique représentant la RAM**. Elle stocke la largeur, hauteur, position et les données mémoire (RamMemoryData), ainsi qu'une couleur pour mettre en surbrillance les cellules. La méthode generateRAMPanel construit un JPanel principal avec un **titre “RAM”**, puis un panneau interne (containerPanel) qui contient une **liste verticale de cellules mémoire**. Chaque cellule montre l'adresse à gauche et la valeur à droite avec des polices blanches et un fond bleu foncé. Un **séparateur blanc** est ajouté entre les cellules pour plus de clarté, et un **CustomScroller** est utilisé pour permettre le défilement vertical. Chaque cellule est aussi enregistrée dans cellPanels pour permettre une manipulation ou surbrillance future.

```
 1  public void updateRAM(LinkedHashMap<String, String> newMemoryData) {
 2      this.RamMemoryData = newMemoryData;
 3      cellPanels.clear();
 4      containerPanel.removeAll();
 5
 6      for (Map.Entry<String, String> entry : newMemoryData.entrySet()) {
 7          JPanel memoryCell = new JPanel(new BorderLayout());
 8          memoryCell.setBackground(new Color(6,26,83));
 9          memoryCell.setBorder(BorderFactory.createEmptyBorder(1, 6, 1, 6));
10
11          JLabel addressLabel = new JLabel(entry.getKey(), SwingConstants.LEFT);
12          addressLabel.setForeground(Color.WHITE);
13          addressLabel.setFont(new Font("Segoe UI", Font.PLAIN, 14));
14          memoryCell.add(addressLabel, BorderLayout.WEST);
15
16          JLabel valueLabel = new JLabel(entry.getValue(), SwingConstants.RIGHT);
17          valueLabel.setForeground(Color.WHITE);
18          valueLabel.setFont(new Font("Segoe UI", Font.PLAIN, 14));
19          memoryCell.add(valueLabel, BorderLayout.EAST);
20
21          addHoverEffect(addressLabel, valueLabel);
22
23          containerPanel.add(memoryCell);
24
25          JPanel separator = new JPanel();
26          separator.setPreferredSize(new Dimension(0, 1));
27          separator.setMaximumSize(new Dimension(Integer.MAX_VALUE, 1));
28          separator.setBackground(Color.WHITE);
29          containerPanel.add(separator);
30
31          cellPanels.add(memoryCell);
32      }
33
34      containerPanel.revalidate();
35      containerPanel.repaint();
36  }
37
38  public void resetRAM() {
39      LinkedHashMap<String, String> ramData = new LinkedHashMap<>();
40
41      this.RamMemoryData = ramData;
42      cellPanels.clear();
43      containerPanel.removeAll();
44
45      for (Map.Entry<String, String> entry : RamMemoryData.entrySet()) {
46          JPanel cellPanel = new JPanel();
47          cellPanel.setLayout(new BorderLayout());
48          cellPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
49
50          JLabel keyLabel = new JLabel(entry.getKey(), SwingConstants.LEFT);
51          keyLabel.setForeground(new Color(180, 180, 180));
52          keyLabel.setFont(new Font("Segoe UI", Font.PLAIN, 14));
53          cellPanel.add(keyLabel, BorderLayout.WEST);
54
55          JLabel valueLabel = new JLabel(entry.getValue(), SwingConstants.RIGHT);
56          valueLabel.setForeground(new Color(180, 180, 180));
57          valueLabel.setFont(new Font("Segoe UI", Font.PLAIN, 14));
58          cellPanel.add(valueLabel, BorderLayout.EAST);
59
60          addHoverEffect(keyLabel, valueLabel);
61
62          containerPanel.add(cellPanel);
63          containerPanel.add(Box.createVerticalStrut(10));
64          containerPanel.add(cellPanel);
65      }
66
67      containerPanel.revalidate();
68      containerPanel.repaint();
69  }
```

a classe RAM inclut aussi deux autres méthodes principales :

1. **updateRAM** : met à jour dynamiquement les valeurs affichées de la RAM. Elle remplace les données actuelles par newMemoryData, vide la liste des panneaux de cellules (cellPanels) et le panneau interne (containerPanel), puis recrée chaque cellule avec son adresse et sa valeur, ajoute un **séparateur blanc** entre elles et applique un effet au survol avec addHoverEffect. Enfin, le panneau est **revalidé et repassé au rendu** pour afficher les changements immédiatement.
2. **resetRAM** : réinitialise complètement la RAM à un état vide. Elle crée un nouveau LinkedHashMap vide, supprime toutes les cellules actuelles et reconstruit le panneau avec des **labels gris clair** pour les clés et valeurs. Des espaces verticaux sont ajoutés pour séparer les cellules et addHoverEffect est appliqué pour maintenir la cohérence visuelle.

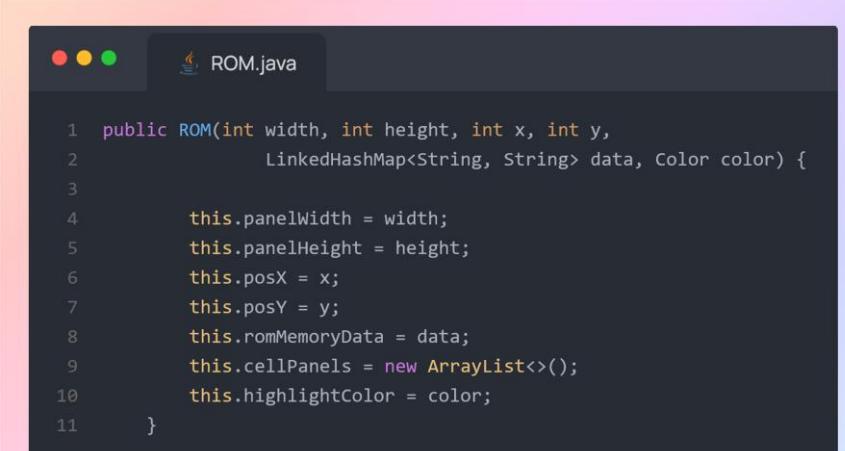
Ces méthodes permettent au simulateur de **mettre à jour et réinitialiser la mémoire en temps réel**, ce qui est essentiel pour simuler les opérations d'un microprocesseur.



```
1 public void addHoverEffect(JLabel addressLabel, JLabel valueLabel) {  
2     MouseAdapter hoverAdapter = new MouseAdapter() {  
3         @Override  
4         public void mouseEntered(MouseEvent e) {  
5             addressLabel.setForeground(Color.YELLOW);  
6             valueLabel.setForeground(Color.YELLOW);  
7         }  
8         @Override  
9         public void mouseExited(MouseEvent e) {  
10            addressLabel.setForeground(Color.WHITE);  
11            valueLabel.setForeground(Color.WHITE);  
12        }  
13    };  
14    addressLabel.addMouseListener(hoverAdapter);  
15    valueLabel.addMouseListener(hoverAdapter);  
16}  
17}
```

La méthode addHoverEffect prend deux JLabel (l'adresse et la valeur d'une cellule mémoire) et leur ajoute un **MouseAdapter**. Lorsque le curseur de la souris entre sur le label (mouseEntered), la couleur du texte devient **jaune**, et lorsqu'il sort (mouseExited), elle revient à **blanc**. Cela améliore la lisibilité et l'interaction avec les cellules de RAM dans l'interface graphique du simulateur

### [Le composant panneau graphique du ROM \(ROM.java\)](#)



```
1 public ROM(int width, int height, int x, int y,  
2           LinkedHashMap<String, String> data, Color color) {  
3  
4     this.panelWidth = width;  
5     this.panelHeight = height;  
6     this.posX = x;  
7     this.posY = y;  
8     this.romMemoryData = data;  
9     this.cellPanels = new ArrayList<>();  
10    this.highlightColor = color;  
11}
```

```

1  public JPanel generateROMPanel() {
2      JPanel romPanel = new JPanel(new BorderLayout());
3      romPanel.setPreferredSize(new Dimension(panelWidth, panelHeight));
4      romPanel.setBounds(posX, posY, panelWidth, panelHeight);
5      romPanel.setBackground(new Color(6, 26, 83));
6      romPanel.setBorder(new ComponentShadow());
7
8      JLabel headerLabel = new JLabel("ROM", SwingConstants.CENTER);
9      headerLabel.setForeground(Color.WHITE);
10     headerLabel.setFont(new Font("Roboto Black", Font.BOLD, 22));
11     romPanel.add(headerLabel, BorderLayout.NORTH);
12
13     JPanel containerPanel = new JPanel();
14     containerPanel.setLayout(new BoxLayout(containerPanel, BoxLayout.Y_AXIS));
15     containerPanel.setBackground(new Color(6, 26, 83));
16     containerPanel.setBorder(BorderFactory.createEmptyBorder(10, 0, 0, 0));
17
18     for (Map.Entry<String, String> entry : romMemoryData.entrySet()) {
19         JPanel romCellPanel = new JPanel(new BorderLayout());
20         romCellPanel.setBackground(new Color(6, 26, 83));
21         romCellPanel.setBorder(BorderFactory.createEmptyBorder(1, 6, 1, 6));
22
23         JLabel addressLabel = new JLabel(entry.getKey(), SwingConstants.LEFT);
24         addressLabel.setForeground(Color.WHITE);
25         addressLabel.setFont(new Font("Bebas Neue", Font.PLAIN, 14));
26         romCellPanel.add(addressLabel, BorderLayout.WEST);
27
28         JLabel valueLabel = new JLabel(entry.getValue(), SwingConstants.RIGHT);
29         valueLabel.setForeground(Color.WHITE);
30         valueLabel.setFont(new Font("Bebas Neue", Font.PLAIN, 14));
31         romCellPanel.add(valueLabel, BorderLayout.EAST);
32
33         addHoverEffect(addressLabel, valueLabel);
34
35         containerPanel.add(romCellPanel);
36
37         JPanel separator = new JPanel();
38         separator.setPreferredSize(new Dimension(0, 1));
39         separator.setMaximumSize(new Dimension(Integer.MAX_VALUE, 1));
40         separator.setBackground(Color.WHITE);
41         containerPanel.add(separator);
42
43         cellPanels.add(romCellPanel);
44     }
45
46     JScrollPane scrollPane = new JScrollPane(containerPanel);
47     scrollPane.setBorder(BorderFactory.createEmptyBorder());
48     scrollPane.setBackground(new Color(40, 40, 40));
49     scrollPane.getVerticalScrollBar().setUI(new CustomScroller());
50     scrollPane.getVerticalScrollBar().setUnitIncrement(6);
51     scrollPane.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
52
53     romPanel.add(scrollPane, BorderLayout.CENTER);
54
55     return romPanel;
56 }

```

La classe ROM crée un **panneau graphique pour afficher la mémoire ROM** dans l’émulateur CPU. Elle stocke la largeur, la hauteur, la position, les données mémoire (romMemoryData) et une couleur de surbrillance. La méthode generateROMPanel construit un panneau principal avec un **titre “ROM”**, puis un panneau interne contenant une **liste verticale de cellules**, chacune affichant l’adresse à gauche et la valeur à droite avec un **effet de survol**. Des séparateurs blancs sont ajoutés entre les cellules pour plus de clarté, et un **CustomScroller** permet le défilement vertical, rendant l’ensemble lisible et interactif dans l’interface graphique.

```

1  public void updateROM(LinkedHashMap<String, String> romMemoryData) {
2      this.romMemoryData = romMemoryData;
3      this.cellPanels.clear();
4      containerPanel.removeAll();
5
6      for (Map.Entry<String, String> entry : romMemoryData.entrySet()) {
7          JPanel romCellPanel = new JPanel(new BorderLayout());
8          romCellPanel.setBackground(new Color(6, 26, 83));
9          romCellPanel.setBorder(BorderFactory.createEmptyBorder(1, 6, 1, 6));
10
11         JLabel addressLabel = new JLabel(entry.getKey(), SwingConstants.LEFT);
12         addressLabel.setForeground(Color.WHITE);
13         addressLabel.setFont(new Font("Segoe UI", Font.PLAIN, 14));
14         romCellPanel.add(addressLabel, BorderLayout.WEST);
15
16         JLabel valueLabel = new JLabel(entry.getValue(), SwingConstants.RIGHT);
17         valueLabel.setForeground(Color.WHITE);
18         valueLabel.setFont(new Font("Segoe UI", Font.PLAIN, 14));
19         romCellPanel.add(valueLabel, BorderLayout.EAST);
20
21         addHoverEffect(addressLabel, valueLabel);
22
23         containerPanel.add(romCellPanel);
24
25         JPanel separator = new JPanel();
26         separator.setPreferredSize(new Dimension(0, 1));
27         separator.setMaximumSize(new Dimension(Integer.MAX_VALUE, 1));
28         separator.setBackground(Color.WHITE);
29         containerPanel.add(separator);
30
31         cellPanels.add(romCellPanel);
32     }
33
34     containerPanel.revalidate();
35     containerPanel.repaint();
36 }
37
38 public void setCurrent(String addressKey) {
39     for (JPanel cellPanel : cellPanels) {
40         for (Component component : cellPanel.getComponents()) {
41             if (component instanceof JLabel) {
42                 label.setForeground(Color.WHITE);
43                 label.setFont(new Font("Segoe UI", Font.PLAIN, 14));
44             }
45         }
46     }
47
48     for (JPanel cellPanel : cellPanels) {
49         JLabel addressLabel = (JLabel) cellPanel.getComponent(0);
50         if (addressLabel.getText().equals(addressKey)) {
51             this.currentAddress = addressKey;
52             addressLabel.setForeground(highlightColor);
53             addressLabel.setFont(new Font("Segoe UI", Font.BOLD, 14));
54
55             JLabel valueLabel = (JLabel) cellPanel.getComponent(1);
56             valueLabel.setForeground(highlightColor);
57             valueLabel.setFont(new Font("Segoe UI", Font.BOLD, 14));
58             break;
59         }
60     }
61
62     containerPanel.revalidate();
63     containerPanel.repaint();
64 }

```

La classe ROM inclut aussi deux méthodes pour **mettre à jour et mettre en évidence les cellules de mémoire**. La méthode updateROM remplace les données ROM existantes par romMemoryData, vide les cellules actuelles et reconstruit chaque cellule avec son adresse et sa valeur, en appliquant un effet au survol et des séparateurs blancs, puis redessine le panneau. La méthode setCurrent permet de **mettre en surbrillance une cellule spécifique** correspondant à l'adresse donnée (addressKey) en changeant la couleur et la police de l'adresse et de sa valeur, tandis que toutes les autres cellules retrouvent leur apparence normale. Cela permet de visualiser facilement l'état actuel de la ROM dans l'interface du simulateur,

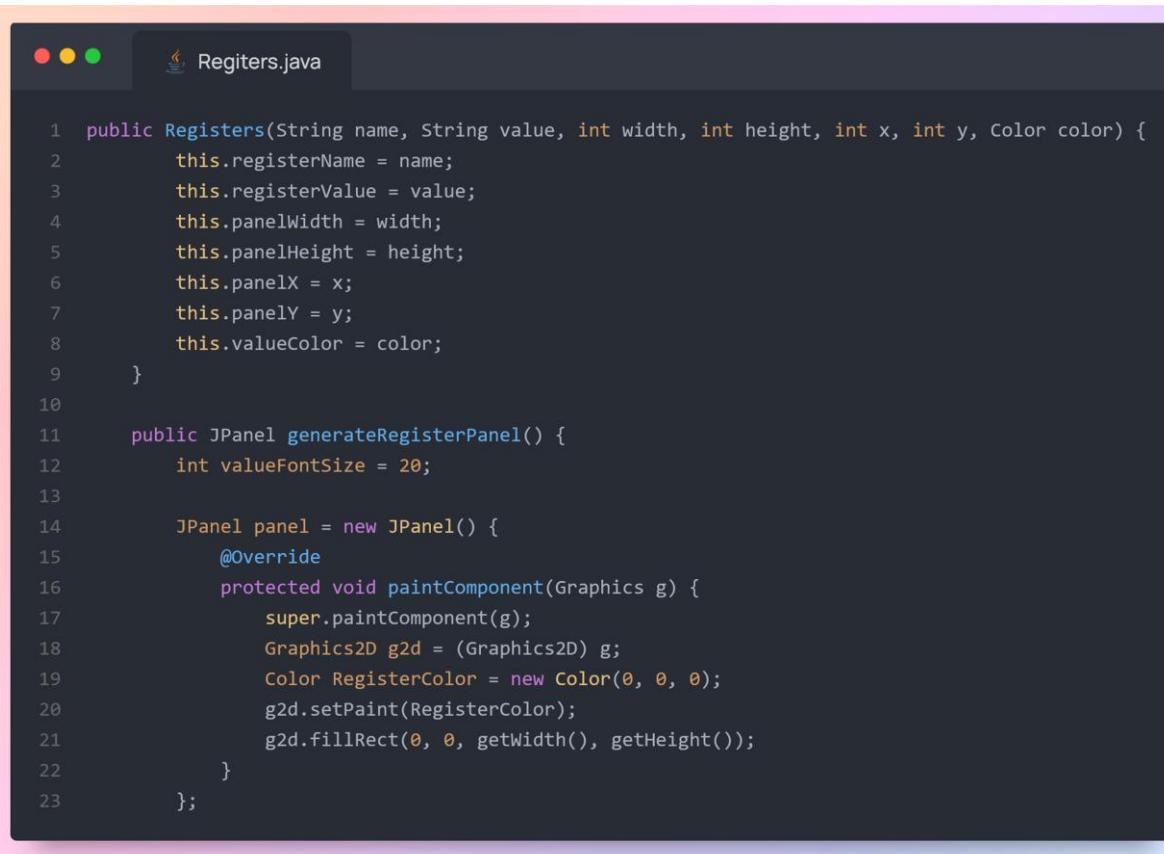


```
ROM.java

1 public void addHoverEffect(JLabel addressLabel, JLabel valueLabel) {
2     MouseAdapter hoverAdapter = new MouseAdapter() {
3         @Override
4         public void mouseEntered(MouseEvent e) {
5             addressLabel.setForeground(Color.YELLOW);
6             valueLabel.setForeground(Color.YELLOW);
7         }
8         @Override
9         public void mouseExited(MouseEvent e) {
10            addressLabel.setForeground(Color.WHITE);
11            valueLabel.setForeground(Color.WHITE);
12        }
13    };
14    addressLabel.addMouseListener(hoverAdapter);
15    valueLabel.addMouseListener(hoverAdapter);
16 }
```

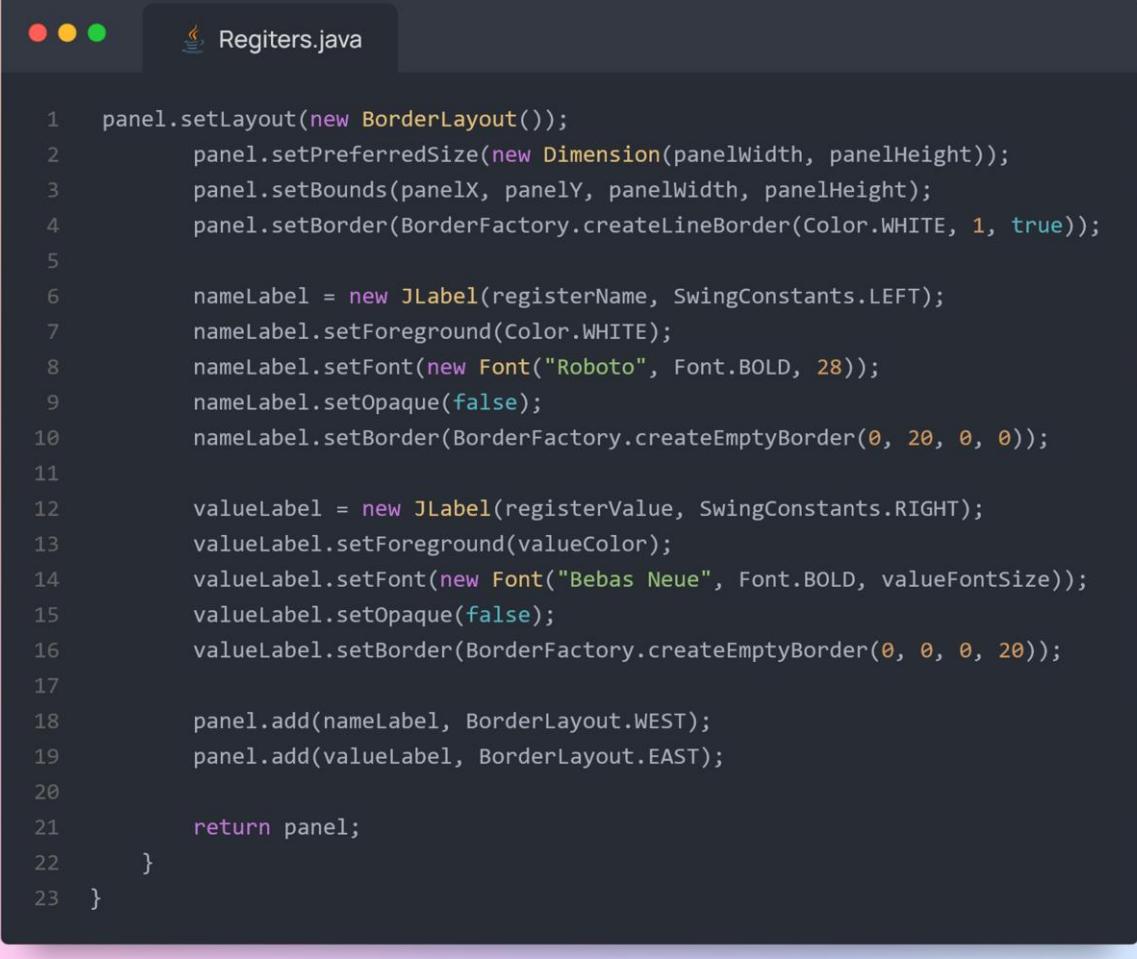
La méthode addHoverEffect prend deux JLabel (l'adresse et la valeur d'une cellule) et leur ajoute un **MouseAdapter**. Lorsque la souris entre sur l'un des labels (mouseEntered), la couleur du texte devient **jaune**, et lorsqu'elle sort (mouseExited), elle redevient **blanche**. Cet effet visuel améliore l'interaction et la lisibilité des cellules de ROM dans l'interface graphique du simulateur.

### Le composant panneau graphique des registres (Registers.java)



```
Registers.java

1 public Registers(String name, String value, int width, int height, int x, int y, Color color) {
2     this.registerName = name;
3     this.registerValue = value;
4     this.panelWidth = width;
5     this.panelHeight = height;
6     this.panelX = x;
7     this.panelY = y;
8     this.valueColor = color;
9 }
10
11 public JPanel generateRegisterPanel() {
12     int valueFontSize = 20;
13
14     JPanel panel = new JPanel() {
15         @Override
16         protected void paintComponent(Graphics g) {
17             super.paintComponent(g);
18             Graphics2D g2d = (Graphics2D) g;
19             Color RegisterColor = new Color(0, 0, 0);
20             g2d.setPaint(RegisterColor);
21             g2d.fillRect(0, 0, getWidth(), getHeight());
22         }
23     };
24 }
```

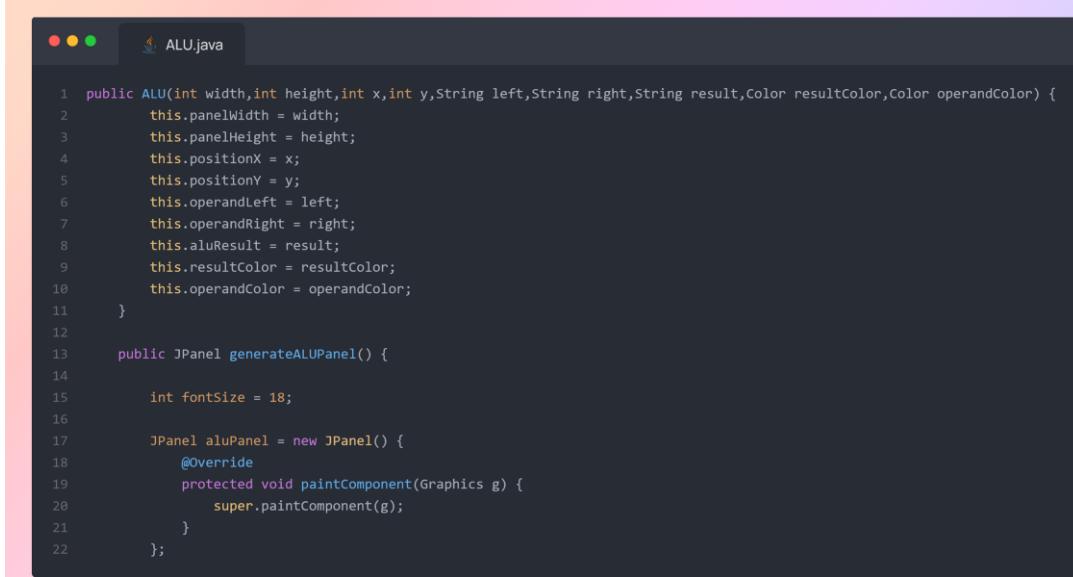


```

1  panel.setLayout(new BorderLayout());
2      panel.setPreferredSize(new Dimension(panelWidth, panelHeight));
3      panel.setBounds(panelX, panelY, panelWidth, panelHeight);
4      panel.setBorder(BorderFactory.createLineBorder(Color.WHITE, 1, true));
5
6      nameLabel = new JLabel(registerName, SwingConstants.LEFT);
7      nameLabel.setForeground(Color.WHITE);
8      nameLabel.setFont(new Font("Roboto", Font.BOLD, 28));
9      nameLabel.setOpaque(false);
10     nameLabel.setBorder(BorderFactory.createEmptyBorder(0, 20, 0, 0));
11
12     valueLabel = new JLabel(registerValue, SwingConstants.RIGHT);
13     valueLabel.setForeground(valueColor);
14     valueLabel.setFont(new Font("Bebas Neue", Font.BOLD, valueFontSize));
15     valueLabel.setOpaque(false);
16     valueLabel.setBorder(BorderFactory.createEmptyBorder(0, 0, 0, 20));
17
18     panel.add(nameLabel, BorderLayout.WEST);
19     panel.add(valueLabel, BorderLayout.EAST);
20
21     return panel;
22 }
23 }
```

La classe Registers crée un **panneau graphique pour représenter un registre du CPU**. Elle stocke le nom et la valeur du registre, ainsi que sa taille, sa position et la couleur de la valeur. La méthode generateRegisterPanel construit un JPanel avec un **fond noir et bordure blanche**, puis ajoute un **JLabel à gauche** pour le nom du registre et un **JLabel à droite** pour sa valeur, avec des polices et couleurs personnalisées. Ce panneau permet d'afficher visuellement les registres dans l'interface du simulateur de manière claire et lisible.

### Le composant d'Unité Arithmétique et Logique (ALU.java)



```

1  public ALU(int width,int height,int x,int y,String left,String right,String result,Color resultColor,Color operandColor) {
2      this.panelWidth = width;
3      this.panelHeight = height;
4      this.positionX = x;
5      this.positionY = y;
6      this.operandLeft = left;
7      this.operandRight = right;
8      this.aluResult = result;
9      this.resultColor = resultColor;
10     this.operandColor = operandColor;
11 }
12
13     public JPanel generateALUPanel() {
14
15         int fontSize = 18;
16
17         JPanel aluPanel = new JPanel() {
18             @Override
19             protected void paintComponent(Graphics g) {
20                 super.paintComponent(g);
21             }
22         };
23 }
```

```

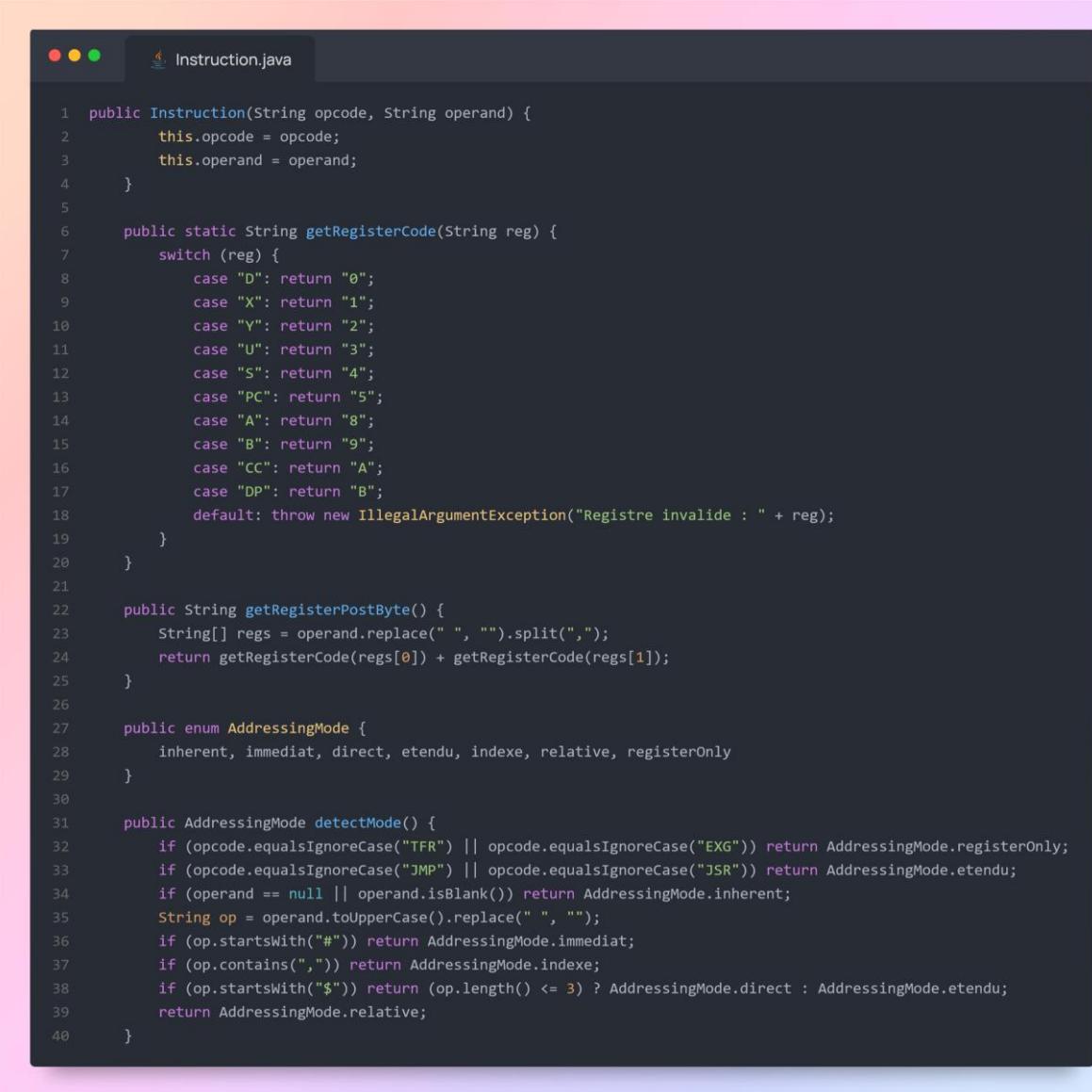
1  aluPanel.setLayout(null);
2  aluPanel.setOpaque(false);
3  aluPanel.setPreferredSize(new Dimension(panelWidth, panelHeight));
4  aluPanel.setBounds(positionX, positionY, panelWidth, panelHeight);
5
6  leftOperandLabel = new JLabel(operandLeft);
7  leftOperandLabel.setBounds(10, 10, panelWidth / 2 - 20, 30);
8  leftOperandLabel.setHorizontalTextPosition(SwingConstants.LEFT);
9  leftOperandLabel.setFont(new Font("Arial", Font.BOLD, fontSize));
10 leftOperandLabel.setForeground(operandColor);
11
12 rightOperandLabel = new JLabel(operandRight);
13 rightOperandLabel.setBounds(
14     panelWidth / 2 + 10,
15     10,
16     panelWidth / 2 - 20,
17     30
18 );
19 rightOperandLabel.setHorizontalTextPosition(SwingConstants.RIGHT);
20 rightOperandLabel.setFont(new Font("Arial", Font.BOLD, fontSize));
21 rightOperandLabel.setForeground(operandColor);
22
23 int resultSize = 40;
24
25 resultLabel = new JPanel();
26 resultLabel.setBounds(
27     (panelWidth - resultSize) / 2,
28     panelHeight - resultSize - 10,
29     resultSize,
30     resultSize
31 );
32 resultLabel.setBackground(new Color(6, 26, 83));
33 resultLabel.setLayout(new BorderLayout());
34 resultLabel.setBorder(
35     BorderFactory.createLineBorder(Color.WHITE, 1, true)
36 );
37
38 resultValue = new JLabel(aluResult);
39 resultValue.setHorizontalAlignment(SwingConstants.CENTER);
40 resultValue.setFont(new Font("Arial", Font.BOLD, fontSize + 6));
41 resultValue.setForeground(resultColor);
42
43 resultLabel.add(resultValue, BorderLayout.CENTER);
44
45 aluPanel.add(leftOperandLabel);
46 aluPanel.add(rightOperandLabel);
47 aluPanel.add(resultLabel);
48
49 return aluPanel;
50 }
51
52 public void updateALU(String left, String right, String result) {
53     this.operandLeft = left;
54     this.operandRight = right;
55     this.aluResult = result;
56
57     if (leftOperandLabel != null)
58         leftOperandLabel.setText(left);
59
60     if (rightOperandLabel != null)
61         rightOperandLabel.setText(right);
62
63     if (resultValue != null)
64         resultValue.setText(result);
65 }

```

La classe ALU crée un **panneau graphique pour simuler visuellement l'Unité Arithmétique et Logique** du CPU. Elle stocke les opérandes gauche et droite, le résultat de l'opération, les dimensions et la position du panneau, ainsi que les couleurs pour distinguer les opérandes et le résultat. La méthode generateALUPanel construit un JPanel transparent et positionné, avec **deux labels en haut pour afficher les opérandes** et un **panneau central avec bordure blanche pour le résultat**, dont le texte est centré et mis en valeur par une couleur spécifique. La méthode updateALU permet de **modifier dynamiquement les valeurs affichées** pour les opérandes et le résultat, ce qui rend possible le suivi en temps réel des calculs effectués par le CPU dans le simulateur. Cette approche rend l'interface plus intuitive et visuellement claire pour l'utilisateur.

## Présentation des composants d'analyse et d'exécution (Package cpu)

### Le composant du control des instructions (Instruction.java)



```
 1 public Instruction(String opcode, String operand) {
 2     this.opcode = opcode;
 3     this.operand = operand;
 4 }
 5
 6     public static String getRegisterCode(String reg) {
 7         switch (reg) {
 8             case "D": return "0";
 9             case "X": return "1";
10             case "Y": return "2";
11             case "U": return "3";
12             case "S": return "4";
13             case "PC": return "5";
14             case "A": return "8";
15             case "B": return "9";
16             case "CC": return "A";
17             case "DP": return "B";
18             default: throw new IllegalArgumentException("Registre invalide : " + reg);
19         }
20     }
21
22     public String getRegisterPostByte() {
23         String[] regs = operand.replace(" ", "").split(",");
24         return getRegisterCode(regs[0]) + getRegisterCode(regs[1]);
25     }
26
27     public enum AddressingMode {
28         inherent, immediat, direct, etendu, indexe, relative, registerOnly
29     }
30
31     public AddressingMode detectMode() {
32         if (opcode.equalsIgnoreCase("TFR") || opcode.equalsIgnoreCase("EXG")) return AddressingMode.registerOnly;
33         if (opcode.equalsIgnoreCase("JMP") || opcode.equalsIgnoreCase("JSR")) return AddressingMode.etendu;
34         if (operand == null || operand.isBlank()) return AddressingMode.inherent;
35         String op = operand.toUpperCase().replace(" ", "");
36         if (op.startsWith("#")) return AddressingMode.immediat;
37         if (op.contains(",")) return AddressingMode.indexe;
38         if (op.startsWith("$")) return (op.length() <= 3) ? AddressingMode.direct : AddressingMode.etendu;
39         return AddressingMode.relative;
40     }
}
```

La classe Instruction joue un rôle crucial dans le simulateur du CPU 6809, car elle représente chaque instruction du processeur avec son opcode (le code de l'opération) et son operand (la donnée ou le registre associé). Elle inclut la méthode getRegisterCode qui convertit les noms des registres (comme A, B, X, Y, U, S, PC, CC, DP) en codes hexadécimaux utilisés par le CPU, ce qui est indispensable pour simuler correctement les instructions qui manipulent plusieurs registres. La méthode getRegisterPostByte combine les codes de deux registres pour créer le post-byte des instructions spéciales comme TFR ou EXG, permettant de gérer les transferts entre registres.

L'énumération AddressingMode définit tous les modes d'adressage possibles du 6809 : inherent, immediat, direct, etendu, indexe, relative et registerOnly. La méthode detectMode analyse automatiquement l'instruction et son opérande pour déterminer le mode d'adressage utilisé. Par exemple, si l'opérande commence par #, le mode est immédiat ; si elle contient une virgule, le mode est indexé, etc.

Cette classe est donc fondamentale pour le fonctionnement du simulateur, car elle permet à la fois de traduire les instructions en codes machine compréhensibles par le CPU simulé, de gérer les registres correctement, et de déterminer le comportement exact de chaque instruction selon son mode d'adressage, ce qui rend la simulation fidèle et précise.

```

1  private static final Map<String, Map<AddressingMode, String>> OPCODES = new HashMap<>();
2
3  static {
4      OPCODES.put("LDA", Map.of(AddressingMode.immediat, "86", AddressingMode.direct, "96", AddressingMode.etendu, "B6", AddressingMode.indexe, "A6"));
5      OPCODES.put("LDB", Map.of(AddressingMode.immediat, "C6", AddressingMode.direct, "D6", AddressingMode.etendu, "F6", AddressingMode.indexe, "E6"));
6      OPCODES.put("LDX", Map.of(AddressingMode.immediat, "8E", AddressingMode.direct, "9E", AddressingMode.etendu, "BE", AddressingMode.indexe, "AE"));
7      OPCODES.put("LDY", Map.of(AddressingMode.immediat, "108E", AddressingMode.direct, "109E", AddressingMode.etendu, "10BE", AddressingMode.indexe, "10AE"));
8      OPCODES.put("LDU", Map.of(AddressingMode.immediat, "CE", AddressingMode.direct, "DE", AddressingMode.etendu, "FE", AddressingMode.indexe, "EE"));
9      OPCODES.put("LDS", Map.of(AddressingMode.immediat, "10CE", AddressingMode.direct, "10DE", AddressingMode.etendu, "10FE", AddressingMode.indexe, "10EE"));
10     OPCODES.put("STA", Map.of(AddressingMode.direct, "97", AddressingMode.etendu, "B7", AddressingMode.indexe, "A7"));
11     OPCODES.put("STB", Map.of(AddressingMode.direct, "D7", AddressingMode.etendu, "F7", AddressingMode.indexe, "E7"));
12     OPCODES.put("STX", Map.of(AddressingMode.direct, "9F", AddressingMode.etendu, "BF", AddressingMode.indexe, "AF"));
13     OPCODES.put("STY", Map.of(AddressingMode.direct, "109F", AddressingMode.etendu, "10BF", AddressingMode.indexe, "10AF"));
14     OPCODES.put("STU", Map.of(AddressingMode.direct, "??", AddressingMode.etendu, "??", AddressingMode.indexe, "??""));
15     OPCODES.put("STS", Map.of(AddressingMode.direct, "??", AddressingMode.etendu, "??", AddressingMode.indexe, "??""));
16     OPCODES.put("ADD", Map.of(AddressingMode.immediat, "88", AddressingMode.direct, "9B", AddressingMode.etendu, "BB", AddressingMode.indexe, "AB"));
17     OPCODES.put("ADD", Map.of(AddressingMode.immediat, "CB", AddressingMode.direct, "DB", AddressingMode.etendu, "FB", AddressingMode.indexe, "EB"));
18     OPCODES.put("SUBA", Map.of(AddressingMode.immediat, "88", AddressingMode.direct, "90", AddressingMode.etendu, "B8", AddressingMode.indexe, "A0"));
19     OPCODES.put("SUBB", Map.of(AddressingMode.immediat, "C0", AddressingMode.direct, "D0", AddressingMode.etendu, "F0", AddressingMode.indexe, "E0"));
20     OPCODES.put("CMPA", Map.of(AddressingMode.immediat, "81", AddressingMode.direct, "91", AddressingMode.etendu, "B1", AddressingMode.indexe, "A1"));
21     OPCODES.put("CMPB", Map.of(AddressingMode.immediat, "C1", AddressingMode.direct, "D1", AddressingMode.etendu, "F1", AddressingMode.indexe, "E1"));
22     OPCODES.put("ANDA", Map.of(AddressingMode.immediat, "84", AddressingMode.direct, "94", AddressingMode.indexe, "A4", AddressingMode.etendu, "B4"));
23     OPCODES.put("ANDB", Map.of(AddressingMode.immediat, "C4", AddressingMode.direct, "D4", AddressingMode.indexe, "E4", AddressingMode.etendu, "F4"));
24     OPCODES.put("ORA", Map.of(AddressingMode.immediat, "8A", AddressingMode.direct, "9A", AddressingMode.indexe, "AA", AddressingMode.etendu, "BA"));
25     OPCODES.put("ORB", Map.of(AddressingMode.immediat, "CA", AddressingMode.direct, "DA", AddressingMode.indexe, "EA", AddressingMode.etendu, "FA"));
26     OPCODES.put("EORA", Map.of(AddressingMode.immediat, "88", AddressingMode.direct, "98", AddressingMode.indexe, "A8", AddressingMode.etendu, "B8"));
27     OPCODES.put("EORB", Map.of(AddressingMode.immediat, "C8", AddressingMode.direct, "D8", AddressingMode.indexe, "E8", AddressingMode.etendu, "F8"));
28     OPCODES.put("LSLA", Map.of(AddressingMode.inherent, "48"));
29     OPCODES.put("LSLB", Map.of(AddressingMode.inherent, "58"));
30     OPCODES.put("LSRA", Map.of(AddressingMode.inherent, "44"));
31     OPCODES.put("LSRB", Map.of(AddressingMode.inherent, "54"));
32     OPCODES.put("ROLA", Map.of(AddressingMode.inherent, "49"));
33     OPCODES.put("ROLB", Map.of(AddressingMode.inherent, "59"));
34     OPCODES.put("RORA", Map.of(AddressingMode.inherent, "46"));
35     OPCODES.put("RORB", Map.of(AddressingMode.inherent, "56"));
36     OPCODES.put("CLRA", Map.of(AddressingMode.inherent, "4F"));
37     OPCODES.put("CLR", Map.of(AddressingMode.inherent, "5F"));
38     OPCODES.put("INCA", Map.of(AddressingMode.inherent, "4C"));
39     OPCODES.put("INC", Map.of(AddressingMode.inherent, "5C"));
40     OPCODES.put("DECA", Map.of(AddressingMode.inherent, "4A"));
41     OPCODES.put("DEC", Map.of(AddressingMode.inherent, "5A"));
42     OPCODES.put("COMA", Map.of(AddressingMode.inherent, "43"));
43     OPCODES.put("COM", Map.of(AddressingMode.inherent, "53"));
44     OPCODES.put("NEGA", Map.of(AddressingMode.inherent, "40"));
45     OPCODES.put("NEG", Map.of(AddressingMode.inherent, "50"));
46     OPCODES.put("NOP", Map.of(AddressingMode.inherent, "12"));
47     OPCODES.put("RTS", Map.of(AddressingMode.inherent, "39"));
48     OPCODES.put("JMP", Map.of(AddressingMode.etendu, "E"));
49     OPCODES.put("JSR", Map.of(AddressingMode.etendu, "BD"));
50     OPCODES.put("BRA", Map.of(AddressingMode.relative, "28"));
51     OPCODES.put("BED", Map.of(AddressingMode.relative, "27"));
52     OPCODES.put("BNE", Map.of(AddressingMode.relative, "26"));
53     OPCODES.put("BCC", Map.of(AddressingMode.relative, "24"));
54     OPCODES.put("BCS", Map.of(AddressingMode.relative, "25"));
55     OPCODES.put("BMI", Map.of(AddressingMode.relative, "28"));
56     OPCODES.put("BPL", Map.of(AddressingMode.relative, "2A"));
57     OPCODES.put("BVC", Map.of(AddressingMode.relative, "49"));
58     OPCODES.put("BVS", Map.of(AddressingMode.relative, "59"));
59     OPCODES.put("TFR", Map.of(AddressingMode.registerOnly, "1F"));
60     OPCODES.put("EXG", Map.of(AddressingMode.registerOnly, "1E"));
61     OPCODES.put("END", Map.of(AddressingMode.inherent, "3F"));
62     OPCODES.put("SMI", Map.of(AddressingMode.inherent, "3F"));
63     OPCODES.put("BRA", Map.of(AddressingMode.relative, "28"));
64     OPCODES.put("BEQ", Map.of(AddressingMode.relative, "27"));
65     OPCODES.put("BNE", Map.of(AddressingMode.relative, "26"));
66     OPCODES.put("BCC", Map.of(AddressingMode.relative, "24"));
67     OPCODES.put("BCS", Map.of(AddressingMode.relative, "25"));
68     OPCODES.put("BMI", Map.of(AddressingMode.relative, "28"));
69     OPCODES.put("BPL", Map.of(AddressingMode.relative, "2A"));
70     OPCODES.put("BVC", Map.of(AddressingMode.relative, "49"));
71     OPCODES.put("BVS", Map.of(AddressingMode.relative, "59"));
72     OPCODES.put("JMP", Map.of(AddressingMode.etendu, "7E", AddressingMode.indexe, "6E", AddressingMode.relative, "20"));
73   });
74 }

```

Ce bloc de code définit une **table centrale des opcodes** du processeur sous la forme d'une structure Map qui associe chaque instruction assembleur (comme LDA, ADD, JMP, etc.) à ses **codes machine hexadécimaux** selon le **mode d'adressage** utilisé (immédiat, direct, étendu, indexé, relatif, inhérent ou registre). Chaque entrée permet ainsi de traduire automatiquement une instruction lisible par l'humain en une valeur binaire compréhensible par le CPU. Cette table est **fondamentale pour l'assembleur ou le simulateur**, car elle garantit une correspondance exacte entre la syntaxe assembleur et l'architecture matérielle du processeur. Elle facilite aussi la détection d'erreurs (instruction ou mode non supporté), rend le système extensible (ajout facile de nouvelles instructions) et assure une exécution correcte et cohérente du programme, ce qui en fait un élément clé pour la fiabilité et la précision de la simulation du processeur.

```

1 public String getOpcodeHex() {
2     AddressingMode mode = detectMode();
3     Map<AddressingMode, String> modes = OPCODES.get(opcode);
4     if (modes == null || !modes.containsKey(mode)) throw new IllegalArgumentException("Opcode non supporté : " + opcode + " / " + mode);
5     return modes.get(mode);
6 }
7
8 public static String getAddress(int k) {
9     return Integer.toHexString(0xFFFF + k).toUpperCase();
10}
11
12 public static String filterOperand(String operand) {
13     return operand.replaceAll("[\$\\{\\}]", "");
14}
15
16 public static boolean isSyntaxCorrect(Instruction instr, Map<String, Integer> labelsMap) {
17     if (instr == null) { erreurMessag = "Erreur : instruction nulle"; return false; }
18     if (instr.opcode == null || instr.opcode.isBlank()) { erreurMessag = "Erreur : opcode manquant"; return false; }
19     instr.opcode = instr.opcode.toUpperCase();
20     if (!OPCODES.containsKey(instr.opcode)) { erreurMessag = "Erreur : opcode non supporté : " + instr.opcode; return false; }
21     AddressingMode mode;
22     try { mode = instr.detectMode(); } catch (Exception e) { erreurMessag = "Erreur : mode d'adressage invalide"; return false; }
23     Map<AddressingMode, String> modes = OPCODES.get(opcode);
24     if (!modes.containsKey(mode)) { erreurMessag = "Erreur : mode " + mode + " non valide pour " + instr.opcode; return false; }
25     if (mode != AddressingMode.inherent) {
26         if (instr.operand == null || instr.operand.isBlank()) { erreurMessag = "Erreur : opérande manquant pour " + instr.opcode; return false; }
27         if (mode == AddressingMode.immedia && !instr.operand.startsWith("#")) { erreurMessag = "Erreur : opérande immédiat attendu (#)"; return false; }
28         if (mode == AddressingMode.index && !instr.operand.contains(",")) { erreurMessag = "Erreur : opérande index invalide"; return false; }
29         if (mode == AddressingMode.registerOnly && !instr.operand.matches("[A-Z],[A-Z]")) { erreurMessag = "Erreur : syntaxe EXG/TFR invalide"; return false; }
30         if (mode == AddressingMode.relative && labelsMap == null && !labelsMap.containsKey(instr.operand)) { erreurMessag = "Erreur : label introuvable pour " + instr.opcode; return false; }
31     }
32     try {
33         if (instr.operand != null &&
34             mode != AddressingMode.relative &&
35             mode != AddressingMode.index &&
36             mode != AddressingMode.registerOnly &&
37             mode != AddressingMode.inherent &&
38             mode != AddressingMode.etendu) {
39             Integer.parseInt(filterOperand(instr.operand), 16);
40         }
41     } catch (NumberFormatException e) { erreurMessag = "Erreur : opérande non hexadécimal : " + instr.operand; return false; }
42     return true;
43 }
44 }

```

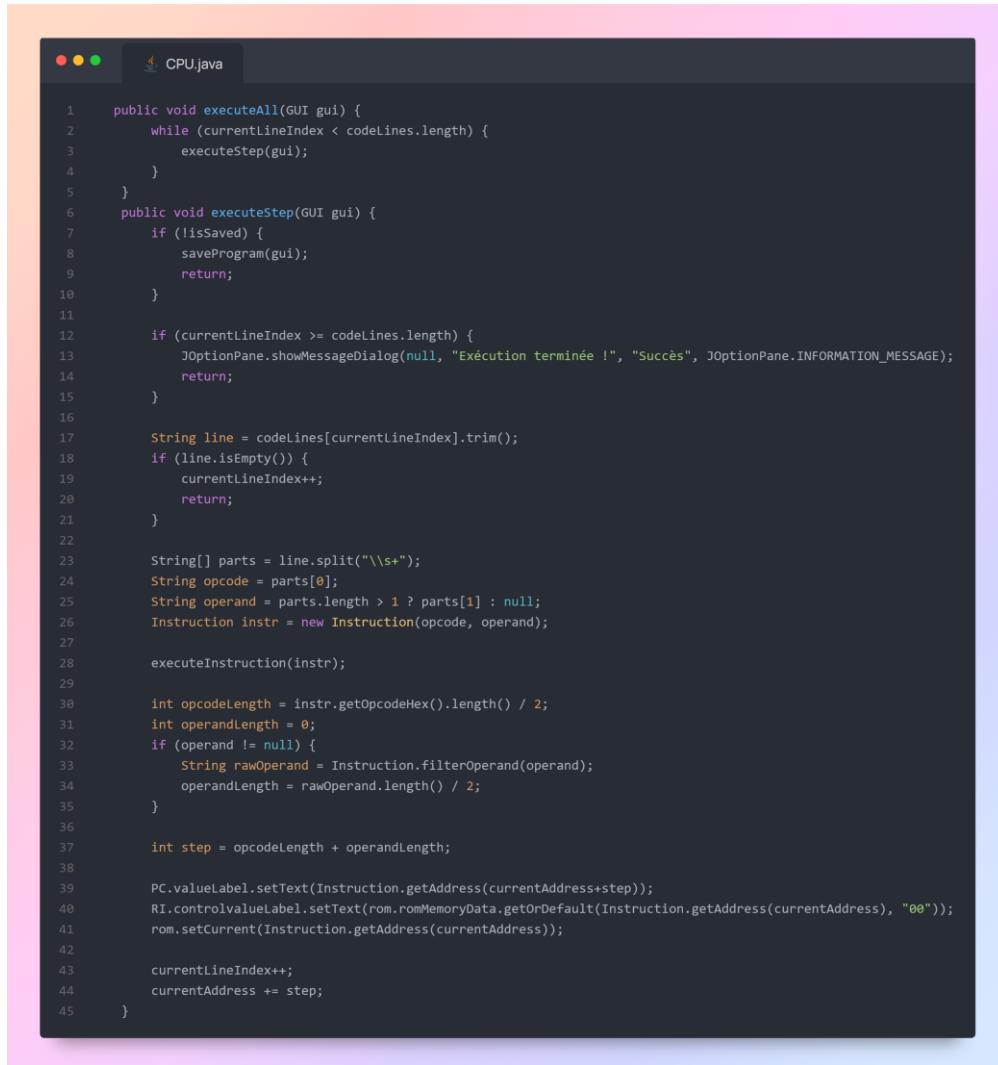
Ce code regroupe des méthodes essentielles pour le **fondement correct d'un assebleur ou d'un simulateur de processeur**, car il assure à la fois la traduction et la validation des instructions. La méthode `getOpcodeHex` permet d'obtenir le code machine hexadécimal exact d'une instruction en fonction de son mode d'adressage, garantissant ainsi une exécution fidèle au processeur cible. La méthode `getAddress` calcule des adresses mémoire de manière cohérente à partir d'une base fixe, tandis que `filterOperand` nettoie les opérandes pour faciliter leur analyse numérique. La méthode la plus importante, `isSyntaxCorrect`, joue un rôle clé de **sécurité et de fiabilité**, car elle vérifie minutieusement la validité de l'instruction (opcode existant, mode d'adressage compatible, opérande correct, labels définis, format hexadécimal valide) et empêche toute erreur avant l'assemblage ou l'exécution. Ensemble, ces fonctions sont indispensables pour garantir que le code assebleur est correct, compréhensible par la machine et exempt d'erreurs logiques ou syntaxiques, ce qui est fondamental pour la stabilité et la précision du système.

## Le composant du control des d'exécution (CPU.java)

```

1 public class CPU {
2     private MMU ram;
3     private ControlPanel cp;
4     private Registers A, B, X, Y, U, S, PC, DP;
5     private Registers N, Z, V, C, H;
6     private ControlPanel K1;
7     private StackMemory stack;
8     private java.util.Stack<Integer> callStack = new java.util.Stack<>();
9     private ALU alu;
10
11    String[] codelines;
12    Boolean isRun = false;
13    int currentLineIndex = 0;
14    int currentRegister = 0;
15    LinkedHashMap<String, Integer> labelsMap = new LinkedHashMap<>();
16
17    public CPU(ROM rom, ROM rom, Registers A, Registers B, Registers X, Registers Y, Registers U, Registers S, Registers PC, Registers DP, ControlPanel K1, Registers N, Registers Z, Registers V, Registers C, Registers H, ALU alu, BitDisplay bina, BitDisplay binB) {
18        this.ram = rom;
19        this.ram = rom;
20        this.currentLineIndex = 0;
21
22        this.A = A;
23        this.B = B;
24        this.X = X;
25        this.Y = Y;
26        this.U = U;
27        this.S = S;
28        this.PC = PC;
29        this.DP = DP;
30        this.N = N;
31        this.Z = Z;
32        this.V = V;
33        this.C = C;
34        this.H = H;
35
36        this.alu = alu;
37        this.bina = bina;
38        this.binB = binB;
39        this.stack = callStack;
40    }
41
42    public void initialize(ControlPanel cp) {
43        codelines = cp.codelines.getTxt().replaceAll("\\r\\n", "\\n").toUpperCase().split("\\n");
44
45        for (int i = 0; i < codelines.length; i++) {
46            if (codelines[i].contains(":")) {
47                labelsMap.put(codelines[i].split(":")[0], i);
48            }
49        }
50
51        for (int i = 0; i < codelines.length; i++) {
52            codelines[i] = codelines[i].contains(":") ? "" : codelines[i];
53        }
54    }
55
56    public void run() {
57        while (isRun) {
58            if (currentLineIndex < codelines.length) {
59                String line = codelines[currentLineIndex];
60
61                if (line.startsWith("LD")) {
62                    String[] args = line.split(" ");
63                    String regName = args[1];
64                    String memAddress = args[2];
65
66                    int regIndex = Registers.getRegIndex(regName);
67                    int memIndex = Integer.parseInt(memAddress);
68
69                    A.setRegValue(regIndex, ram.read(memIndex));
70
71                    System.out.println("LD " + regName + " " + memAddress);
72                }
73
74                else if (line.startsWith("ST")) {
75                    String[] args = line.split(" ");
76                    String memAddress = args[1];
77                    String regName = args[2];
78
79                    int memIndex = Integer.parseInt(memAddress);
80                    int regIndex = Registers.getRegIndex(regName);
81
82                    ram.write(memIndex, A.getRegValue(regIndex));
83
84                    System.out.println("ST " + memAddress + " " + regName);
85                }
86
87                else if (line.startsWith("ADD")) {
88                    String[] args = line.split(" ");
89                    String regName = args[1];
90                    String regName2 = args[2];
91
92                    int regIndex = Registers.getRegIndex(regName);
93                    int regIndex2 = Registers.getRegIndex(regName2);
94
95                    int result = alu.add(A.getRegValue(regIndex), A.getRegValue(regIndex2));
96
97                    A.setRegValue(regIndex, result);
98
99                    System.out.println("ADD " + regName + " " + regName2);
100                }
101
102                else if (line.startsWith("SUB")) {
103                    String[] args = line.split(" ");
104                    String regName = args[1];
105                    String regName2 = args[2];
106
107                    int regIndex = Registers.getRegIndex(regName);
108                    int regIndex2 = Registers.getRegIndex(regName2);
109
110                    int result = alu.subtract(A.getRegValue(regIndex), A.getRegValue(regIndex2));
111
112                    A.setRegValue(regIndex, result);
113
114                    System.out.println("SUB " + regName + " " + regName2);
115                }
116
117                else if (line.startsWith("MUL")) {
118                    String[] args = line.split(" ");
119                    String regName = args[1];
120                    String regName2 = args[2];
121
122                    int regIndex = Registers.getRegIndex(regName);
123                    int regIndex2 = Registers.getRegIndex(regName2);
124
125                    int result = alu.multiply(A.getRegValue(regIndex), A.getRegValue(regIndex2));
126
127                    A.setRegValue(regIndex, result);
128
129                    System.out.println("MUL " + regName + " " + regName2);
130                }
131
132                else if (line.startsWith("DIV")) {
133                    String[] args = line.split(" ");
134                    String regName = args[1];
135                    String regName2 = args[2];
136
137                    int regIndex = Registers.getRegIndex(regName);
138                    int regIndex2 = Registers.getRegIndex(regName2);
139
140                    int result = alu.divide(A.getRegValue(regIndex), A.getRegValue(regIndex2));
141
142                    A.setRegValue(regIndex, result);
143
144                    System.out.println("DIV " + regName + " " + regName2);
145                }
146
147                else if (line.startsWith("NOT")) {
148                    String[] args = line.split(" ");
149                    String regName = args[1];
150
151                    int regIndex = Registers.getRegIndex(regName);
152
153                    int result = alu.not(A.getRegValue(regIndex));
154
155                    A.setRegValue(regIndex, result);
156
157                    System.out.println("NOT " + regName);
158                }
159
160                else if (line.startsWith("AND")) {
161                    String[] args = line.split(" ");
162                    String regName = args[1];
163                    String regName2 = args[2];
164
165                    int regIndex = Registers.getRegIndex(regName);
166                    int regIndex2 = Registers.getRegIndex(regName2);
167
168                    int result = alu.and(A.getRegValue(regIndex), A.getRegValue(regIndex2));
169
170                    A.setRegValue(regIndex, result);
171
172                    System.out.println("AND " + regName + " " + regName2);
173                }
174
175                else if (line.startsWith("OR")) {
176                    String[] args = line.split(" ");
177                    String regName = args[1];
178                    String regName2 = args[2];
179
180                    int regIndex = Registers.getRegIndex(regName);
181                    int regIndex2 = Registers.getRegIndex(regName2);
182
183                    int result = alu.or(A.getRegValue(regIndex), A.getRegValue(regIndex2));
184
185                    A.setRegValue(regIndex, result);
186
187                    System.out.println("OR " + regName + " " + regName2);
188                }
189
190                else if (line.startsWith("XOR")) {
191                    String[] args = line.split(" ");
192                    String regName = args[1];
193                    String regName2 = args[2];
194
195                    int regIndex = Registers.getRegIndex(regName);
196                    int regIndex2 = Registers.getRegIndex(regName2);
197
198                    int result = alu.xor(A.getRegValue(regIndex), A.getRegValue(regIndex2));
199
200                    A.setRegValue(regIndex, result);
201
202                    System.out.println("XOR " + regName + " " + regName2);
203                }
204
205                else if (line.startsWith("SHL")) {
206                    String[] args = line.split(" ");
207                    String regName = args[1];
208                    String regName2 = args[2];
209
210                    int regIndex = Registers.getRegIndex(regName);
211                    int regIndex2 = Registers.getRegIndex(regName2);
212
213                    int result = alu.shl(A.getRegValue(regIndex), A.getRegValue(regIndex2));
214
215                    A.setRegValue(regIndex, result);
216
217                    System.out.println("SHL " + regName + " " + regName2);
218                }
219
220                else if (line.startsWith("SHR")) {
221                    String[] args = line.split(" ");
222                    String regName = args[1];
223                    String regName2 = args[2];
224
225                    int regIndex = Registers.getRegIndex(regName);
226                    int regIndex2 = Registers.getRegIndex(regName2);
227
228                    int result = alu.shr(A.getRegValue(regIndex), A.getRegValue(regIndex2));
229
230                    A.setRegValue(regIndex, result);
231
232                    System.out.println("SHR " + regName + " " + regName2);
233                }
234
235                else if (line.startsWith("JMP")) {
236                    String[] args = line.split(" ");
237                    String labelName = args[1];
238
239                    int labelIndex = labelsMap.get(labelName);
240
241                    currentLineIndex = labelIndex;
242
243                    System.out.println("JMP " + labelName);
244                }
245
246                else if (line.startsWith("HLT")) {
247                    isRun = false;
248
249                    System.out.println("HALT");
250                }
251
252                else {
253                    System.out.println("Unknown instruction: " + line);
254                }
255            }
256        }
257    }
258
259    public void printRegisters() {
260        System.out.println("Registers: " + A + ", " + B + ", " + X + ", " + Y + ", " + U + ", " + S + ", " + PC + ", " + DP + ", " + N + ", " + Z + ", " + V + ", " + C + ", " + H);
261    }
262
263    public void printMemory() {
264        System.out.println("Memory: " + ram);
265    }
266
267    public void printControlPanel() {
268        System.out.println("Control Panel: " + cp);
269    }
270
271    public void printBitDisplays() {
272        System.out.println("Bit Displays: " + bina + ", " + binB);
273    }
274
275    public void printStack() {
276        System.out.println("Call Stack: " + callStack);
277    }
278
279    public void printALU() {
280        System.out.println("ALU: " + alu);
281    }
282
283    public void printLabelsMap() {
284        System.out.println("Labels Map: " + labelsMap);
285    }
286
287    public void printRegistersMap() {
288        System.out.println("Registers Map: " + Registers.getRegistersMap());
289    }
290
291    public void printControlPanelMap() {
292        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
293    }
294
295    public void printBitDisplayMaps() {
296        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
297    }
298
299    public void printCallStack() {
300        System.out.println("Call Stack: " + callStack);
301    }
302
303    public void printStackMap() {
304        System.out.println("Stack Map: " + StackMemory.getStackMemoryMap());
305    }
306
307    public void printMMU() {
308        System.out.println("MMU: " + ram);
309    }
310
311    public void printRegistersMap() {
312        System.out.println("Registers Map: " + Registers.getRegistersMap());
313    }
314
315    public void printControlPanelMap() {
316        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
317    }
318
319    public void printBitDisplayMaps() {
320        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
321    }
322
323    public void printCallStack() {
324        System.out.println("Call Stack: " + callStack);
325    }
326
327    public void printStackMap() {
328        System.out.println("Stack Map: " + callStack);
329    }
330
331    public void printMMU() {
332        System.out.println("MMU: " + ram);
333    }
334
335    public void printRegistersMap() {
336        System.out.println("Registers Map: " + Registers.getRegistersMap());
337    }
338
339    public void printControlPanelMap() {
340        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
341    }
342
343    public void printBitDisplayMaps() {
344        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
345    }
346
347    public void printCallStack() {
348        System.out.println("Call Stack: " + callStack);
349    }
350
351    public void printStackMap() {
352        System.out.println("Stack Map: " + callStack);
353    }
354
355    public void printMMU() {
356        System.out.println("MMU: " + ram);
357    }
358
359    public void printRegistersMap() {
360        System.out.println("Registers Map: " + Registers.getRegistersMap());
361    }
362
363    public void printControlPanelMap() {
364        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
365    }
366
367    public void printBitDisplayMaps() {
368        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
369    }
370
371    public void printCallStack() {
372        System.out.println("Call Stack: " + callStack);
373    }
374
375    public void printStackMap() {
376        System.out.println("Stack Map: " + callStack);
377    }
378
379    public void printMMU() {
380        System.out.println("MMU: " + ram);
381    }
382
383    public void printRegistersMap() {
384        System.out.println("Registers Map: " + Registers.getRegistersMap());
385    }
386
387    public void printControlPanelMap() {
388        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
389    }
390
391    public void printBitDisplayMaps() {
392        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
393    }
394
395    public void printCallStack() {
396        System.out.println("Call Stack: " + callStack);
397    }
398
399    public void printStackMap() {
400        System.out.println("Stack Map: " + callStack);
401    }
402
403    public void printMMU() {
404        System.out.println("MMU: " + ram);
405    }
406
407    public void printRegistersMap() {
408        System.out.println("Registers Map: " + Registers.getRegistersMap());
409    }
410
411    public void printControlPanelMap() {
412        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
413    }
414
415    public void printBitDisplayMaps() {
416        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
417    }
418
419    public void printCallStack() {
420        System.out.println("Call Stack: " + callStack);
421    }
422
423    public void printStackMap() {
424        System.out.println("Stack Map: " + callStack);
425    }
426
427    public void printMMU() {
428        System.out.println("MMU: " + ram);
429    }
430
431    public void printRegistersMap() {
432        System.out.println("Registers Map: " + Registers.getRegistersMap());
433    }
434
435    public void printControlPanelMap() {
436        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
437    }
438
439    public void printBitDisplayMaps() {
440        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
441    }
442
443    public void printCallStack() {
444        System.out.println("Call Stack: " + callStack);
445    }
446
447    public void printStackMap() {
448        System.out.println("Stack Map: " + callStack);
449    }
450
451    public void printMMU() {
452        System.out.println("MMU: " + ram);
453    }
454
455    public void printRegistersMap() {
456        System.out.println("Registers Map: " + Registers.getRegistersMap());
457    }
458
459    public void printControlPanelMap() {
460        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
461    }
462
463    public void printBitDisplayMaps() {
464        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
465    }
466
467    public void printCallStack() {
468        System.out.println("Call Stack: " + callStack);
469    }
470
471    public void printStackMap() {
472        System.out.println("Stack Map: " + callStack);
473    }
474
475    public void printMMU() {
476        System.out.println("MMU: " + ram);
477    }
478
479    public void printRegistersMap() {
480        System.out.println("Registers Map: " + Registers.getRegistersMap());
481    }
482
483    public void printControlPanelMap() {
484        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
485    }
486
487    public void printBitDisplayMaps() {
488        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
489    }
490
491    public void printCallStack() {
492        System.out.println("Call Stack: " + callStack);
493    }
494
495    public void printStackMap() {
496        System.out.println("Stack Map: " + callStack);
497    }
498
499    public void printMMU() {
500        System.out.println("MMU: " + ram);
501    }
502
503    public void printRegistersMap() {
504        System.out.println("Registers Map: " + Registers.getRegistersMap());
505    }
506
507    public void printControlPanelMap() {
508        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
509    }
510
511    public void printBitDisplayMaps() {
512        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
513    }
514
515    public void printCallStack() {
516        System.out.println("Call Stack: " + callStack);
517    }
518
519    public void printStackMap() {
520        System.out.println("Stack Map: " + callStack);
521    }
522
523    public void printMMU() {
524        System.out.println("MMU: " + ram);
525    }
526
527    public void printRegistersMap() {
528        System.out.println("Registers Map: " + Registers.getRegistersMap());
529    }
530
531    public void printControlPanelMap() {
532        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
533    }
534
535    public void printBitDisplayMaps() {
536        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
537    }
538
539    public void printCallStack() {
540        System.out.println("Call Stack: " + callStack);
541    }
542
543    public void printStackMap() {
544        System.out.println("Stack Map: " + callStack);
545    }
546
547    public void printMMU() {
548        System.out.println("MMU: " + ram);
549    }
550
551    public void printRegistersMap() {
552        System.out.println("Registers Map: " + Registers.getRegistersMap());
553    }
554
555    public void printControlPanelMap() {
556        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
557    }
558
559    public void printBitDisplayMaps() {
560        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
561    }
562
563    public void printCallStack() {
564        System.out.println("Call Stack: " + callStack);
565    }
566
567    public void printStackMap() {
568        System.out.println("Stack Map: " + callStack);
569    }
570
571    public void printMMU() {
572        System.out.println("MMU: " + ram);
573    }
574
575    public void printRegistersMap() {
576        System.out.println("Registers Map: " + Registers.getRegistersMap());
577    }
578
579    public void printControlPanelMap() {
580        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
581    }
582
583    public void printBitDisplayMaps() {
584        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
585    }
586
587    public void printCallStack() {
588        System.out.println("Call Stack: " + callStack);
589    }
590
591    public void printStackMap() {
592        System.out.println("Stack Map: " + callStack);
593    }
594
595    public void printMMU() {
596        System.out.println("MMU: " + ram);
597    }
598
599    public void printRegistersMap() {
600        System.out.println("Registers Map: " + Registers.getRegistersMap());
601    }
602
603    public void printControlPanelMap() {
604        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
605    }
606
607    public void printBitDisplayMaps() {
608        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
609    }
610
611    public void printCallStack() {
612        System.out.println("Call Stack: " + callStack);
613    }
614
615    public void printStackMap() {
616        System.out.println("Stack Map: " + callStack);
617    }
618
619    public void printMMU() {
620        System.out.println("MMU: " + ram);
621    }
622
623    public void printRegistersMap() {
624        System.out.println("Registers Map: " + Registers.getRegistersMap());
625    }
626
627    public void printControlPanelMap() {
628        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
629    }
630
631    public void printBitDisplayMaps() {
632        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
633    }
634
635    public void printCallStack() {
636        System.out.println("Call Stack: " + callStack);
637    }
638
639    public void printStackMap() {
640        System.out.println("Stack Map: " + callStack);
641    }
642
643    public void printMMU() {
644        System.out.println("MMU: " + ram);
645    }
646
647    public void printRegistersMap() {
648        System.out.println("Registers Map: " + Registers.getRegistersMap());
649    }
650
651    public void printControlPanelMap() {
652        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
653    }
654
655    public void printBitDisplayMaps() {
656        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
657    }
658
659    public void printCallStack() {
660        System.out.println("Call Stack: " + callStack);
661    }
662
663    public void printStackMap() {
664        System.out.println("Stack Map: " + callStack);
665    }
666
667    public void printMMU() {
668        System.out.println("MMU: " + ram);
669    }
670
671    public void printRegistersMap() {
672        System.out.println("Registers Map: " + Registers.getRegistersMap());
673    }
674
675    public void printControlPanelMap() {
676        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
677    }
678
679    public void printBitDisplayMaps() {
680        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
681    }
682
683    public void printCallStack() {
684        System.out.println("Call Stack: " + callStack);
685    }
686
687    public void printStackMap() {
688        System.out.println("Stack Map: " + callStack);
689    }
690
691    public void printMMU() {
692        System.out.println("MMU: " + ram);
693    }
694
695    public void printRegistersMap() {
696        System.out.println("Registers Map: " + Registers.getRegistersMap());
697    }
698
699    public void printControlPanelMap() {
700        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
701    }
702
703    public void printBitDisplayMaps() {
704        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
705    }
706
707    public void printCallStack() {
708        System.out.println("Call Stack: " + callStack);
709    }
710
711    public void printStackMap() {
712        System.out.println("Stack Map: " + callStack);
713    }
714
715    public void printMMU() {
716        System.out.println("MMU: " + ram);
717    }
718
719    public void printRegistersMap() {
720        System.out.println("Registers Map: " + Registers.getRegistersMap());
721    }
722
723    public void printControlPanelMap() {
724        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
725    }
726
727    public void printBitDisplayMaps() {
728        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
729    }
730
731    public void printCallStack() {
732        System.out.println("Call Stack: " + callStack);
733    }
734
735    public void printStackMap() {
736        System.out.println("Stack Map: " + callStack);
737    }
738
739    public void printMMU() {
740        System.out.println("MMU: " + ram);
741    }
742
743    public void printRegistersMap() {
744        System.out.println("Registers Map: " + Registers.getRegistersMap());
745    }
746
747    public void printControlPanelMap() {
748        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
749    }
750
751    public void printBitDisplayMaps() {
752        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
753    }
754
755    public void printCallStack() {
756        System.out.println("Call Stack: " + callStack);
757    }
758
759    public void printStackMap() {
760        System.out.println("Stack Map: " + callStack);
761    }
762
763    public void printMMU() {
764        System.out.println("MMU: " + ram);
765    }
766
767    public void printRegistersMap() {
768        System.out.println("Registers Map: " + Registers.getRegistersMap());
769    }
770
771    public void printControlPanelMap() {
772        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
773    }
774
775    public void printBitDisplayMaps() {
776        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
777    }
778
779    public void printCallStack() {
780        System.out.println("Call Stack: " + callStack);
781    }
782
783    public void printStackMap() {
784        System.out.println("Stack Map: " + callStack);
785    }
786
787    public void printMMU() {
788        System.out.println("MMU: " + ram);
789    }
790
791    public void printRegistersMap() {
792        System.out.println("Registers Map: " + Registers.getRegistersMap());
793    }
794
795    public void printControlPanelMap() {
796        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
797    }
798
799    public void printBitDisplayMaps() {
800        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
801    }
802
803    public void printCallStack() {
804        System.out.println("Call Stack: " + callStack);
805    }
806
807    public void printStackMap() {
808        System.out.println("Stack Map: " + callStack);
809    }
810
811    public void printMMU() {
812        System.out.println("MMU: " + ram);
813    }
814
815    public void printRegistersMap() {
816        System.out.println("Registers Map: " + Registers.getRegistersMap());
817    }
818
819    public void printControlPanelMap() {
820        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
821    }
822
823    public void printBitDisplayMaps() {
824        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
825    }
826
827    public void printCallStack() {
828        System.out.println("Call Stack: " + callStack);
829    }
830
831    public void printStackMap() {
832        System.out.println("Stack Map: " + callStack);
833    }
834
835    public void printMMU() {
836        System.out.println("MMU: " + ram);
837    }
838
839    public void printRegistersMap() {
840        System.out.println("Registers Map: " + Registers.getRegistersMap());
841    }
842
843    public void printControlPanelMap() {
844        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
845    }
846
847    public void printBitDisplayMaps() {
848        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
849    }
850
851    public void printCallStack() {
852        System.out.println("Call Stack: " + callStack);
853    }
854
855    public void printStackMap() {
856        System.out.println("Stack Map: " + callStack);
857    }
858
859    public void printMMU() {
860        System.out.println("MMU: " + ram);
861    }
862
863    public void printRegistersMap() {
864        System.out.println("Registers Map: " + Registers.getRegistersMap());
865    }
866
867    public void printControlPanelMap() {
868        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
869    }
870
871    public void printBitDisplayMaps() {
872        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
873    }
874
875    public void printCallStack() {
876        System.out.println("Call Stack: " + callStack);
877    }
878
879    public void printStackMap() {
880        System.out.println("Stack Map: " + callStack);
881    }
882
883    public void printMMU() {
884        System.out.println("MMU: " + ram);
885    }
886
887    public void printRegistersMap() {
888        System.out.println("Registers Map: " + Registers.getRegistersMap());
889    }
890
891    public void printControlPanelMap() {
892        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
893    }
894
895    public void printBitDisplayMaps() {
896        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
897    }
898
899    public void printCallStack() {
900        System.out.println("Call Stack: " + callStack);
901    }
902
903    public void printStackMap() {
904        System.out.println("Stack Map: " + callStack);
905    }
906
907    public void printMMU() {
908        System.out.println("MMU: " + ram);
909    }
910
911    public void printRegistersMap() {
912        System.out.println("Registers Map: " + Registers.getRegistersMap());
913    }
914
915    public void printControlPanelMap() {
916        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
917    }
918
919    public void printBitDisplayMaps() {
920        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
921    }
922
923    public void printCallStack() {
924        System.out.println("Call Stack: " + callStack);
925    }
926
927    public void printStackMap() {
928        System.out.println("Stack Map: " + callStack);
929    }
930
931    public void printMMU() {
932        System.out.println("MMU: " + ram);
933    }
934
935    public void printRegistersMap() {
936        System.out.println("Registers Map: " + Registers.getRegistersMap());
937    }
938
939    public void printControlPanelMap() {
940        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
941    }
942
943    public void printBitDisplayMaps() {
944        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
945    }
946
947    public void printCallStack() {
948        System.out.println("Call Stack: " + callStack);
949    }
950
951    public void printStackMap() {
952        System.out.println("Stack Map: " + callStack);
953    }
954
955    public void printMMU() {
956        System.out.println("MMU: " + ram);
957    }
958
959    public void printRegistersMap() {
960        System.out.println("Registers Map: " + Registers.getRegistersMap());
961    }
962
963    public void printControlPanelMap() {
964        System.out.println("Control Panel Map: " + ControlPanel.getControlPanelMap());
965    }
966
967    public void printBitDisplayMaps() {
968        System.out.println("Bit Display Maps: " + BitDisplay.getBinaMap() + ", " + BitDisplay.getBinBMap());
969    }
970
971    public void printCallStack() {
972        System.out.println("Call Stack: " + callStack);
973    }
97
```

Cette classe **CPU** représente le cœur du simulateur de processeur, car elle centralise et coordonne tous les composants matériels et logiques nécessaires à l'exécution d'un programme assembleur. Elle regroupe la mémoire **RAM** et **ROM**, les registres principaux (A, B, X, Y, U, S, PC, DP), les registres d'état (N, Z, V, C, H), l'**ALU**, l'instruction courante (RI), ainsi que des éléments visuels comme l'affichage binaire et les panneaux de contrôle, ce qui permet une simulation fidèle et pédagogique du fonctionnement interne du CPU. La pile `callStack` gère les appels de sous-programmes, tandis que les variables comme `currentLineIndex`, `currentAddress` et `labelsMap` assurent le suivi précis de l'exécution du code et des étiquettes. La méthode `initializeLines` est particulièrement importante car elle prépare le code source en supprimant les commentaires, en normalisant l'écriture, en détectant les labels et en nettoyant les lignes, ce qui garantit une exécution correcte et ordonnée des instructions. Cette classe est donc essentielle, car elle fait le lien entre le code assembleur, l'exécution logique et l'interface graphique, assurant la cohérence, la fiabilité et la compréhension du fonctionnement du processeur simulé.



```

1  public void executeAll(GUI gui) {
2      while (currentLineIndex < codeLines.length) {
3          executeStep(gui);
4      }
5  }
6  public void executeStep(GUI gui) {
7      if (!isSaved) {
8          saveProgram(gui);
9          return;
10     }
11
12     if (currentLineIndex >= codeLines.length) {
13         JOptionPane.showMessageDialog(null, "Exécution terminée !", "Succès", JOptionPane.INFORMATION_MESSAGE);
14         return;
15     }
16
17     String line = codeLines[currentLineIndex].trim();
18     if (line.isEmpty()) {
19         currentLineIndex++;
20         return;
21     }
22
23     String[] parts = line.split("\\s+");
24     String opcode = parts[0];
25     String operand = parts.length > 1 ? parts[1] : null;
26     Instruction instr = new Instruction(opcode, operand);
27
28     executeInstruction(instr);
29
30     int opcodeLength = instr.getOpcodeHex().length() / 2;
31     int operandLength = 0;
32     if (operand != null) {
33         String rawOperand = Instruction.filterOperand(operand);
34         operandLength = rawOperand.length() / 2;
35     }
36
37     int step = opcodeLength + operandLength;
38
39     PC.valueLabel.setText(Instruction.getAddress(currentAddress+step));
40     RI.controlvalueLabel.setText(rom.romMemoryData.getOrDefault(Instruction.getAddress(currentAddress), "00"));
41     rom.setCurrent(Instruction.getAddress(currentAddress));
42
43     currentLineIndex++;
44     currentAddress += step;
45 }

```

Les méthodes **executeAll** et **executeStep** jouent un rôle central dans le processus d'exécution du programme au sein du processeur. La méthode **executeAll** permet d'exécuter automatiquement l'ensemble des instructions, l'une après l'autre, jusqu'à la fin du programme. En revanche, **executeStep** assure une exécution pas à pas : elle vérifie d'abord que le programme est bien sauvegardé, ignore les lignes vides, analyse chaque ligne pour en extraire l'opcode et l'opérande, puis crée une instruction qui sera exécutée. Elle calcule ensuite la taille réelle de l'instruction en mémoire afin de mettre à jour correctement le compteur ordinal (**PC**), le registre d'instruction (**RI**) et l'adresse courante dans la ROM. Cette approche est très importante car elle garantit une exécution précise et cohérente des instructions, tout en permettant de visualiser et de comprendre le fonctionnement interne du processeur, étape par étape, ce qui est essentiel pour l'apprentissage et le débogage.

```

1  public void saveProgram(GUI gui) {
2      for (int i = 0; i < codeLines.length; i++) {
3          String line = codeLines[i].trim();
4          if (line.isEmpty()) continue;
5          String[] parts = line.split("\\s+");
6          String opcode = parts[0];
7          String operand = parts.length > 1 ? parts[1] : null;
8          Instruction instr = new Instruction(opcode, operand);
9          if (operand != null && !Instruction.isSyntaxCorrect(instr, labelsMap)) {
10              JOptionPane.showMessageDialog(null,
11                  "Erreur de syntaxe à la ligne " + (i + 1) + "\n" + Instruction.errorMessage,
12                  "Erreur", JOptionPane.ERROR_MESSAGE);
13          }
14      }
15  }
16  // END
17  int lastInstrIndex = -1;
18
19  for (int i = codeLines.length - 1; i >= 0; i--) {
20      if (!codeLines[i].trim().isEmpty()) {
21          lastInstrIndex = i;
22          break;
23      }
24  }
25
26  if (lastInstrIndex == -1 || 
27      !codeLines[lastInstrIndex].trim().equalsIgnoreCase("END")) {
28
29      JOptionPane.showMessageDialog(
30          null,
31          "Erreur : le programme doit se terminer par l'instruction END.",
32          "Erreur d'exécution",
33          JOptionPane.ERROR_MESSAGE
34      );
35      return; //arrêt de saveProgram
36  }
37
38  isSaved = true;
39  gui.btnSave.setText("Supprimer");
40  gui.btnSave.setActionCommand("CLEAR");
41  gui.btnStep.setEnabled(true);
42  gui.btnStep.setForeground(Color.WHITE);
43  gui.btnRun.setEnabled(true);
44  gui.btnRun.setForeground(Color.WHITE);
45
46  LinkedHashMap<String, String> romMap = new LinkedHashMap<>();
47  int k = 0;
48
49  for (int i = 0; i < codeLines.length; i++) {
50      if (codeLines[i].equals("")) continue;
51      String line = codeLines[i].trim();
52      String[] parts = line.split("\\s+");
53      String opcode = parts[0];
54      String operand = parts.length > 1 ? parts[1] : null;
55      Instruction ins = new Instruction(opcode, operand);
56      String opcodeHex = ins.getOpcodeHex().toUpperCase();
57      for (int j = 0; j < opcodeHex.length(); j += 2) {
58          romMap.put(Instruction.getAddress(k++), opcodeHex.substring(j, j + 2));
59      }
60
61      // TFR / EXG
62      if (ins.detectMode() == Instruction.AddressingMode.registerOnly) {
63          String postByte = ins.getRegisterPostByte();
64          romMap.put(Instruction.getAddress(k++), postByte);
65          continue;
66      }
67
68      if (operand != null) {
69          String rawOperand = Instruction.filterOperand(ins.operand);
70          if (!rawOperand.matches("\\$[0-9A-Fa-f]+") && !rawOperand.startsWith("#")) continue;
71
72          rawOperand = Instruction.filterOperand(rawOperand);
73          for (int p = 0; p < rawOperand.length(); p += 2) {
74              String byteStr = (p + 2 < rawOperand.length()) ? rawOperand.substring(p, p + 2) : rawOperand.substring(p);
75              romMap.put(Instruction.getAddress(k++), byteStr);
76          }
77
78          if (rawOperand.contains(",")) {
79              romMap.put(Instruction.getAddress(k++), getIndexPostByte(rawOperand));
80              String[] part = rawOperand.split(",");
81              if (!part[0].isEmpty()) {
82                  int val = part[0].startsWith("$") ? Integer.parseInt(part[0].substring(1), 16)
83                      : Integer.parseInt(part[0]);
84                  romMap.put(Instruction.getAddress(k++), String.format("%02X", val & 0xFF));
85              }
86          } else if (rawOperand.startsWith("#")) {
87              String imm = rawOperand.replace("#", "").replace("$", "");
88              if (imm.length() > 2) {
89                  romMap.put(Instruction.getAddress(k++), imm.substring(0, 2));
90                  romMap.put(Instruction.getAddress(k++), imm.substring(2, 4));
91              } else {
92                  romMap.put(Instruction.getAddress(k++), imm);
93              }
94          } else if (rawOperand.startsWith("$")) {
95              String addr = rawOperand.replace("$", "");
96              if (addr.length() > 2) {
97                  romMap.put(Instruction.getAddress(k++), addr.substring(0, 2));
98                  romMap.put(Instruction.getAddress(k++), addr.substring(2, 4));
99              } else {
100                  romMap.put(Instruction.getAddress(k++), addr);
101              }
102          }
103      }
104  }
105
106  rom.updateROM(romMap);
107  PC.valueLabel.setText(Instruction.getAddress(currentLineIndex));
108
109 }
110 }
```

La méthode **saveProgram** est essentielle car elle assure la validation, l’assemblage et le chargement correct du programme avant son exécution. Elle commence par analyser chaque ligne du code afin de vérifier la syntaxe des instructions et des opérandes, en affichant un message d’erreur précis en cas de problème. Ensuite, elle impose une règle fondamentale : le programme doit obligatoirement se terminer par l’instruction **END**, garantissant ainsi une fin d’exécution claire et contrôlée. Une fois ces vérifications réussies, la méthode prépare l’environnement d’exécution en activant les boutons nécessaires de l’interface graphique, puis traduit chaque instruction en code machine hexadécimal qu’elle stocke dans la mémoire ROM, octet par octet, en tenant compte des différents modes d’adressage (immédiat, indexé, registre, etc.). Cette étape est très importante car elle fait le lien entre le code assembleur écrit par l’utilisateur et sa représentation binaire compréhensible par le processeur, assurant ainsi une exécution fiable, structurée et pédagogique du programme.



```

1  public void clearProgram(GUI gui) {
2      isSaved = false;
3      currentLineIndex = currentAddress = 0;
4      gui.btnExit.setText("Enregistrer ✓");
5      gui.btnExit.setActionCommand("SAVE");
6      gui.btnStep.setEnabled(false);
7      gui.btnRun.setEnabled(false);
8
9      LinkedHashMap<String, String> emptyROM = new LinkedHashMap<>();
10     for (int i = 0; i <= 30; i++) {
11         emptyROM.put(String.format("FE%02X", i), "");
12     }
13     rom.updateROM(emptyROM);
14
15     LinkedHashMap<String, String> emptyRAM = new LinkedHashMap<>();
16     for (int i = 0; i <= 30; i++) {
17         emptyRAM.put(String.format("%04X", i), "00");
18     }
19     ram.updateRAM(emptyRAM);
20     alu.updateALU("00", "00", "00");
21
22     // Reset registers
23     A.valueLabel.setText("00"); B.valueLabel.setText("00");
24     U.valueLabel.setText("0000"); S.valueLabel.setText("0000");
25     X.valueLabel.setText("0000"); Y.valueLabel.setText("0000");
26     DP.valueLabel.setText("00"); RI.controlvalueLabel.setText("00");
27     PC.valueLabel.setText("FE00");
28     C.valueLabel.setText("0"); N.valueLabel.setText("0");
29     Z.valueLabel.setText("1"); H.valueLabel.setText("0");
30     V.valueLabel.setText("0"); binA.valueLabel.setText("00000000");
31     binB.valueLabel.setText("00000000");
32 }

```

La méthode **clearProgram** permet de réinitialiser complètement l’environnement de simulation afin de repartir d’un état propre. Elle commence par annuler l’état de sauvegarde du programme et remettre à zéro les compteurs internes, puis met à jour l’interface graphique en réactivant le bouton *Enregistrer* et en désactivant les commandes d’exécution. Ensuite, elle vide le contenu de la mémoire ROM et de la mémoire RAM en y plaçant des valeurs neutres, réinitialise l’ALU, puis remet tous les registres du processeur (A, B, X, Y, U, S, PC, DP, ainsi que les indicateurs C, N, Z, H et V) à leurs valeurs initiales. Cette méthode garantit ainsi que le processeur, la mémoire et l’interface utilisateur reviennent dans un état cohérent et stable, prêt à accueillir un nouveau programme sans aucune influence de l’exécution précédente.

```

1  public static void saveTextToFile(String textContent) {
2      JFileChooser folderPicker = new JFileChooser();
3      folderPicker.setDialogTitle("Sélectionnez un dossier");
4      folderPicker.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
5
6      int userChoice = folderPicker.showOpenDialog(null);
7      if (userChoice == JFileChooser.APPROVE_OPTION) {
8          File chosenFolder = folderPicker.getSelectedFile();
9          String newFileName = JOptionPane.showInputDialog("Entrez le nom du fichier (sans extension):");
10
11         if (newFileName != null && !newFileName.trim().isEmpty()) {
12             Path newFilePath = Paths.get(chosenFolder.getAbsolutePath(), newFileName + ".asmb");
13             try {
14                 Files.createDirectories(newFilePath.getParent());
15                 Files.createFile(newFilePath);
16                 try (BufferedWriter writer = new BufferedWriter(new FileWriter(newFilePath.toFile()))) {
17                     writer.write(textContent);
18                 }
19                 JOptionPane.showMessageDialog(null, "Fichier créé avec succès : " + newFilePath);
20             } catch (IOException ex) {
21                 JOptionPane.showMessageDialog(null,
22                     "Erreur lors de la création du fichier : " + ex.getMessage(),
23                     "Erreur", JOptionPane.ERROR_MESSAGE);
24             }
25         } else {
26             JOptionPane.showMessageDialog(null, "Le nom du fichier ne peut pas être vide.",
27                     "Erreur", JOptionPane.ERROR_MESSAGE);
28         }
29     } else {
30         JOptionPane.showMessageDialog(null, "Aucun dossier sélectionné.",
31                     "Erreur", JOptionPane.ERROR_MESSAGE);
32     }
33 }
34 public static String loadTextFile() {
35     JFileChooser filePicker = new JFileChooser();
36     filePicker.setDialogTitle("Sélectionnez un fichier à lire");
37
38     int userChoice = filePicker.showOpenDialog(null);
39     if (userChoice == JFileChooser.APPROVE_OPTION) {
40         File chosenFile = filePicker.getSelectedFile();
41         try (BufferedReader fileReader = new BufferedReader(new FileReader(chosenFile))) {
42             StringBuilder fileContent = new StringBuilder();
43             String currentLine;
44             while ((currentLine = fileReader.readLine()) != null) {
45                 fileContent.append(currentLine).append("\n");
46             }
47             return fileContent.toString();
48         } catch (IOException ex) {
49             JOptionPane.showMessageDialog(null,
50                     "Erreur lors de la lecture du fichier : " + ex.getMessage(),
51                     "Erreur", JOptionPane.ERROR_MESSAGE);
52         }
53     } else {
54         JOptionPane.showMessageDialog(null,
55                     "Aucun fichier sélectionné.",
56                     "Erreur", JOptionPane.ERROR_MESSAGE);
57     }
58     return null;
59 }
60

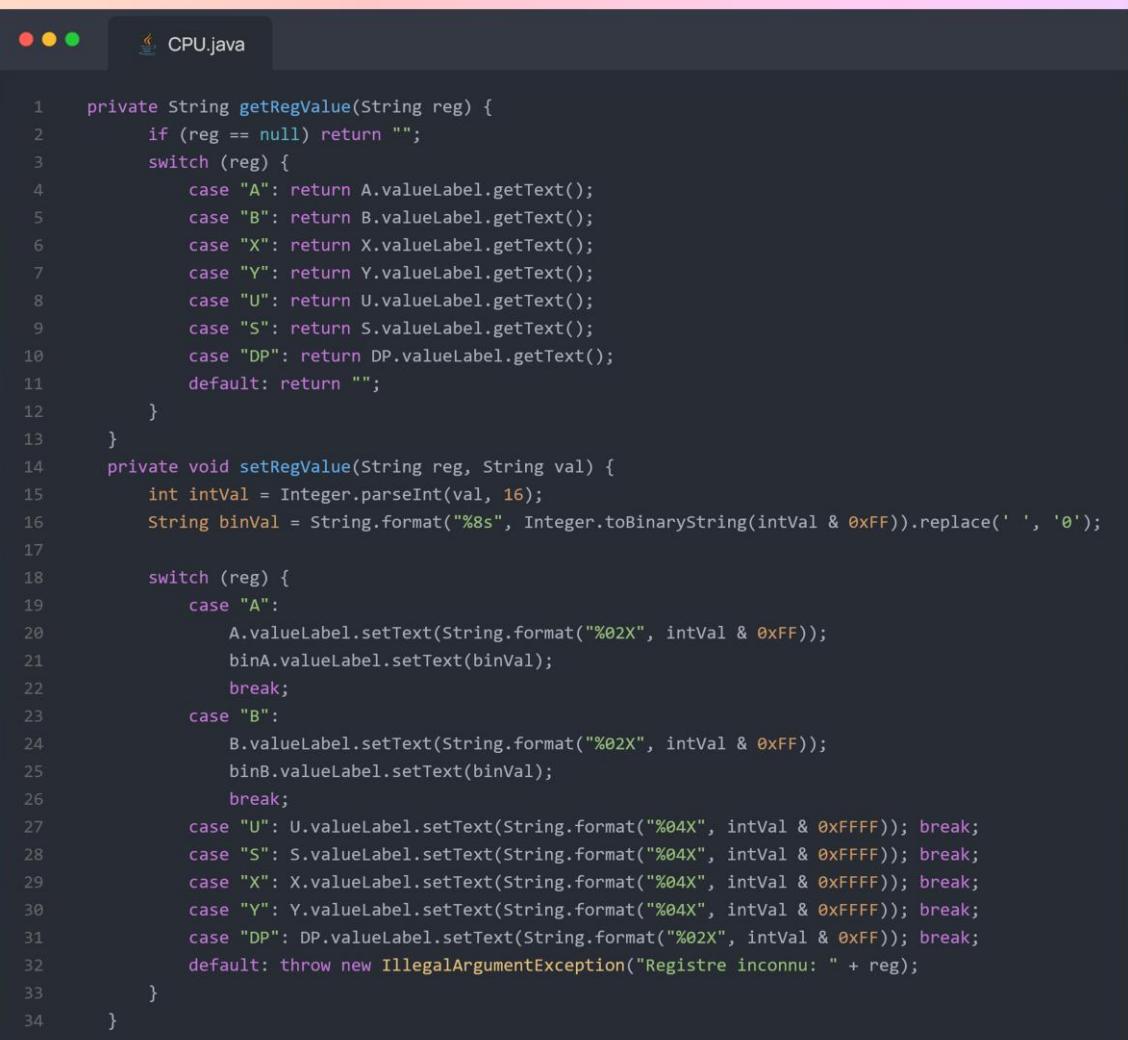
```

Ces deux méthodes permettent de gérer simplement la sauvegarde et le chargement de fichiers texte dans l’application. La méthode **saveTextToFile** ouvre d’abord une fenêtre pour choisir un dossier, puis demande à l’utilisateur le nom du fichier, crée un fichier avec l’extension *.asmb* et y écrit le contenu fourni, tout en affichant des messages clairs en cas de succès ou d’erreur. De son côté, **loadTextFile** permet à l’utilisateur de sélectionner un fichier existant, lit son contenu ligne par ligne et le retourne sous forme de chaîne de caractères, avec une gestion des erreurs si la lecture échoue ou si aucun fichier n’est choisi. Ces méthodes sont importantes car elles offrent à l’utilisateur la possibilité de conserver son travail, de le recharger ultérieurement et d’assurer une interaction fiable entre l’éditeur de code et le système de fichiers.



```
1 public void createNewFile(GUI gui) {
2     saveTextToFile(gui.codeEditor.getText());
3 }
4 public void openFile(GUI gui) {
5     String content = loadTextFile();
6     if (content != null) gui.codeEditor.setText(content);
7 }
```

Les méthodes `createNewFile(GUI gui)` et `openFile(GUI gui)` servent à intégrer la gestion des fichiers dans l'interface graphique de l'application. La première récupère le texte actuel de l'éditeur et utilise `saveTextToFile` pour permettre à l'utilisateur de sauvegarder ce contenu dans un fichier, facilitant ainsi l'enregistrement du programme en cours. La seconde méthode ouvre un fichier existant grâce à `loadTextFile` et insère son contenu directement dans l'éditeur, permettant à l'utilisateur de continuer à modifier un programme déjà sauvegardé. Ensemble, elles simplifient l'interaction entre l'éditeur de code et le système de fichiers, rendant l'application pratique et fonctionnelle pour la création, la sauvegarde et l'ouverture de programmes ASM.

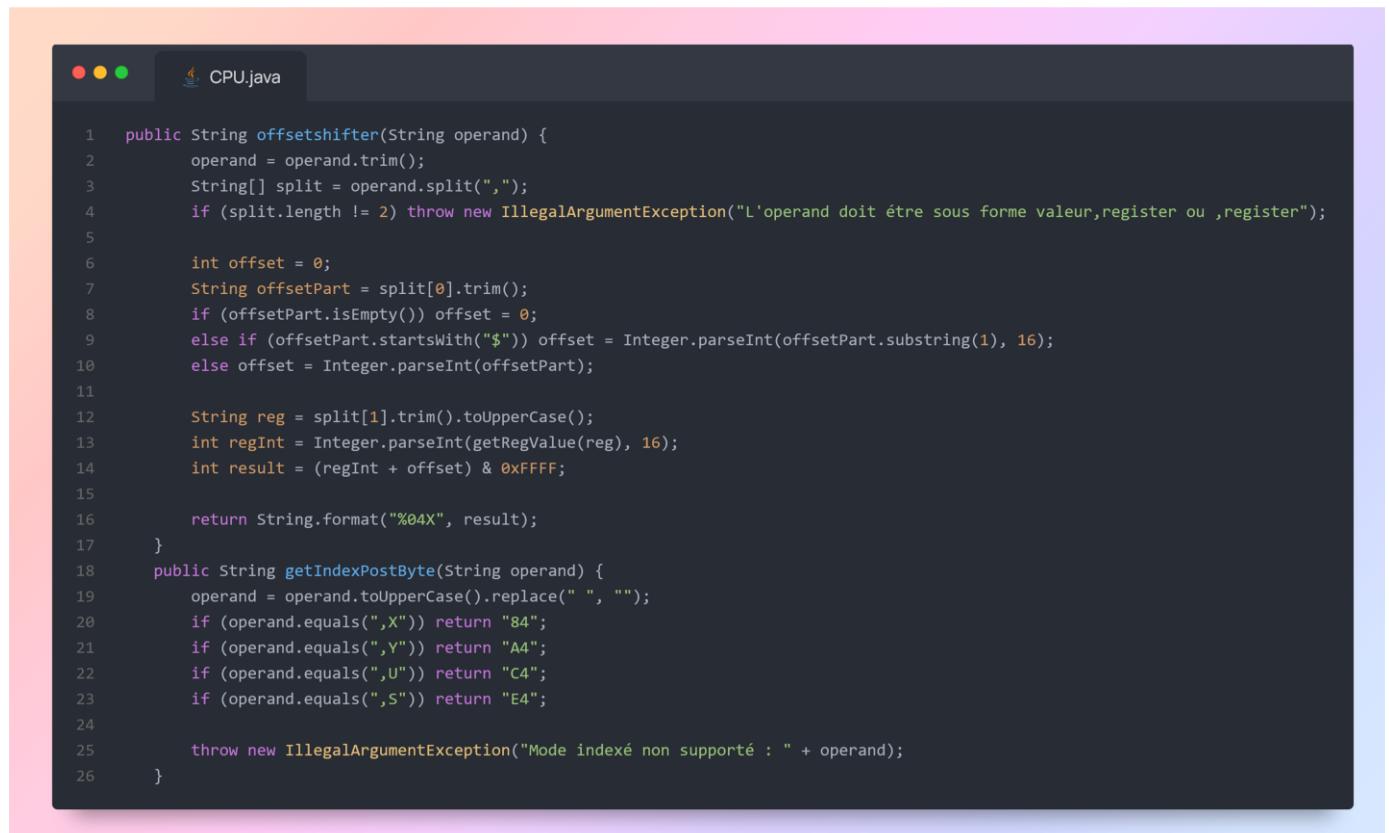


```
1 private String getRegValue(String reg) {
2     if (reg == null) return "";
3     switch (reg) {
4         case "A": return A.valueLabel.getText();
5         case "B": return B.valueLabel.getText();
6         case "X": return X.valueLabel.getText();
7         case "Y": return Y.valueLabel.getText();
8         case "U": return U.valueLabel.getText();
9         case "S": return S.valueLabel.getText();
10        case "DP": return DP.valueLabel.getText();
11        default: return "";
12    }
13 }
14 private void setRegValue(String reg, String val) {
15     int intval = Integer.parseInt(val, 16);
16     String binval = String.format("%8s", Integer.toBinaryString(intval & 0xFF)).replace(' ', '0');
17
18     switch (reg) {
19         case "A":
20             A.valueLabel.setText(String.format("%02X", intval & 0xFF));
21             binA.valueLabel.setText(binval);
22             break;
23         case "B":
24             B.valueLabel.setText(String.format("%02X", intval & 0xFF));
25             binB.valueLabel.setText(binval);
26             break;
27         case "U": U.valueLabel.setText(String.format("%04X", intval & 0xFFFF)); break;
28         case "S": S.valueLabel.setText(String.format("%04X", intval & 0xFFFF)); break;
29         case "X": X.valueLabel.setText(String.format("%04X", intval & 0xFFFF)); break;
30         case "Y": Y.valueLabel.setText(String.format("%04X", intval & 0xFFFF)); break;
31         case "DP": DP.valueLabel.setText(String.format("%02X", intval & 0xFF)); break;
32         default: throw new IllegalArgumentException("Registre inconnu: " + reg);
33     }
34 }
```

Ces deux méthodes gèrent la lecture et l'écriture des valeurs des registres du CPU de manière pratique et uniforme. La méthode getRegValue(String reg) retourne la valeur actuelle d'un registre donné sous forme de chaîne hexadécimale, en vérifiant le nom du registre et en accédant à son valueLabel.

Elle simplifie l'accès aux valeurs des registres sans manipuler directement les composants graphiques.

La méthode setRegValue(String reg, String val) prend une valeur hexadécimale et met à jour le registre correspondant. Pour les registres 8 bits comme A et B, elle met aussi à jour l'affichage binaire associé (binA et binB), ce qui permet de visualiser la valeur en binaire. Pour les registres 16 bits (X, Y, U, S) ou le registre à 8 bits DP, elle formate correctement la valeur avant de l'afficher. Cette abstraction est essentielle pour que les autres parties du CPU puissent manipuler les registres de façon sécurisée et cohérente, tout en maintenant l'interface graphique synchronisée.



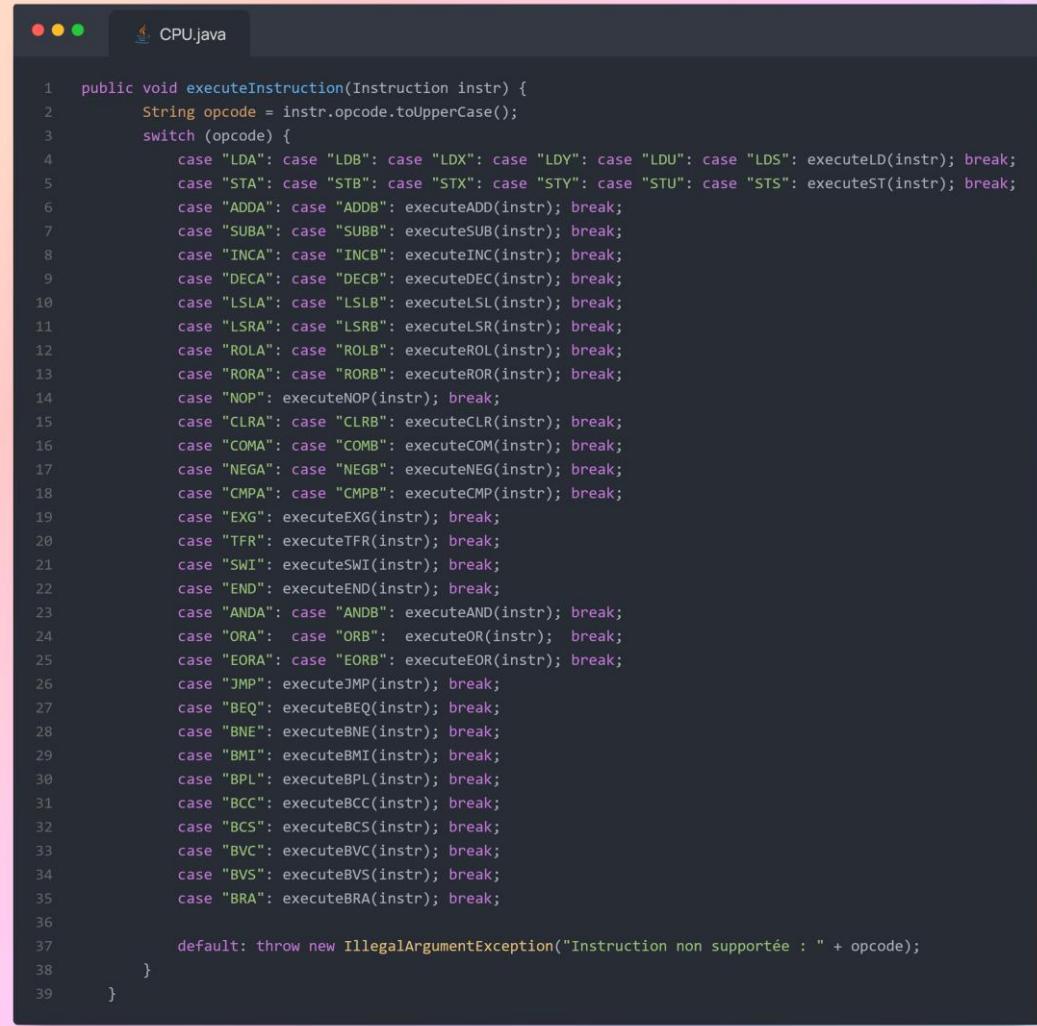
```
1 public String offsetshifter(String operand) {
2     operand = operand.trim();
3     String[] split = operand.split(",");
4     if (split.length != 2) throw new IllegalArgumentException("L'operand doit être sous forme valeur,register ou ,register");
5
6     int offset = 0;
7     String offsetPart = split[0].trim();
8     if (offsetPart.isEmpty()) offset = 0;
9     else if (offsetPart.startsWith("$")) offset = Integer.parseInt(offsetPart.substring(1), 16);
10    else offset = Integer.parseInt(offsetPart);
11
12    String reg = split[1].trim().toUpperCase();
13    int regInt = Integer.parseInt(getRegValue(reg), 16);
14    int result = (regInt + offset) & 0xFFFF;
15
16    return String.format("%04X", result);
17 }
18 public String getIndexPostByte(String operand) {
19     operand = operand.toUpperCase().replace(" ", "");
20     if (operand.equals(",X")) return "84";
21     if (operand.equals(",Y")) return "A4";
22     if (operand.equals(",U")) return "C4";
23     if (operand.equals(",S")) return "E4";
24
25     throw new IllegalArgumentException("Mode indexé non supporté : " + operand);
26 }
```

Ces deux méthodes sont liées au traitement des instructions en mode indexé dans le CPU, qui est un mode d'adressage où un registre sert de base et un offset peut y être ajouté pour accéder à la mémoire.

La méthode offsetshifter(String operand) calcule l'adresse effective lorsque l'instruction utilise un offset avec un registre. Elle sépare l'opérande en deux parties : l'offset (valeur immédiate ou hexadécimale) et le registre (X, Y, U, S). Ensuite, elle additionne l'offset à la valeur actuelle du registre, applique un masque pour rester dans 16 bits, et retourne le résultat sous forme hexadécimale sur 4 caractères. Cela permet de générer automatiquement l'adresse finale à utiliser pour l'instruction.

La méthode getIndexPostByte(String operand) détermine le “post-octet” associé à une instruction indexée simple (c'est-à-dire sans offset ou avec offset implicite), en fonction du registre utilisé. Chaque registre a un code spécifique (X → 84, Y → A4, U → C4, S → E4) qui sera inséré dans la ROM. Si l'opérande ne correspond pas à ces registres, elle lève une exception, garantissant que seules les instructions indexées valides sont utilisées.

En résumé, ces méthodes sont essentielles pour gérer correctement les instructions indexées et calculer les adresses mémoire dans le simulateur CPU.



```

1  public void executeInstruction(Instruction instr) {
2      String opcode = instr.opcode.toUpperCase();
3      switch (opcode) {
4          case "LDA": case "LDB": case "LDX": case "LDY": case "LDU": case "LDS": executeLD(instr); break;
5          case "STA": case "STB": case "STX": case "STY": case "STU": case "STS": executeST(instr); break;
6          case "ADDA": case "ADBB": executeADD(instr); break;
7          case "SUBA": case "SUBB": executeSUB(instr); break;
8          case "INCA": case "INCB": executeINC(instr); break;
9          case "DECA": case "DECBB": executeDEC(instr); break;
10         case "LSLA": case "LSLB": executeLSL(instr); break;
11         case "LSRA": case "LSRB": executeLSR(instr); break;
12         case "ROLA": case "ROLB": executeROL(instr); break;
13         case "RORA": case "RORB": executeROR(instr); break;
14         case "NOP": executeNOP(instr); break;
15         case "CLRA": case "CLRBB": executeCLR(instr); break;
16         case "COMA": case "COMB": executeCOM(instr); break;
17         case "NEGA": case "NEGB": executeNEG(instr); break;
18         case "CMPA": case "CMPP": executeCMP(instr); break;
19         case "EXG": executeEXG(instr); break;
20         case "TFR": executeTFR(instr); break;
21         case "SWI": executeSWI(instr); break;
22         case "END": executeEND(instr); break;
23         case "ANDA": case "ANDB": executeAND(instr); break;
24         case "ORA": case "ORB": executeOR(instr); break;
25         case "EORA": case "EORB": executeEOR(instr); break;
26         case "JMP": executeJMP(instr); break;
27         case "BEQ": executeBEQ(instr); break;
28         case "BNE": executeBNE(instr); break;
29         case "BMI": executeBMI(instr); break;
30         case "BPL": executeBPL(instr); break;
31         case "BCC": executeBCC(instr); break;
32         case "BCS": executeBCS(instr); break;
33         case "BVC": executeBVC(instr); break;
34         case "BVS": executeBVS(instr); break;
35         case "BRA": executeBRA(instr); break;
36
37         default: throw new IllegalArgumentException("Instruction non supportée : " + opcode);
38     }
39 }

```

Cette méthode `executeInstruction(Instruction instr)` joue un rôle central dans le simulateur CPU : elle sert de **dispatcher** pour toutes les instructions possibles.

Pour chaque instruction passée en paramètre, elle identifie l'opcode (en majuscules) et, selon celui-ci, appelle la méthode spécifique qui exécute l'opération correspondante. Par exemple :

- ▣ Les instructions de chargement (LDA, LDB, LDX, etc.) sont redirigées vers `executeLD(instr)`.
- ▣ Les instructions de stockage (STA, STB, etc.) utilisent `executeST(instr)`.
- ▣ Les opérations arithmétiques (ADDA, SUBB, INCA, DECB, etc.) sont dirigées vers leurs fonctions respectives (`executeADD`, `executeSUB`, `executeINC`, `executeDEC`).
- ▣ Les instructions logiques et de manipulation de bits (ANDA, ORA, EORA, LSLA, ROLA, etc.) ont aussi leur propre méthode.
- ▣ Les instructions de contrôle (JMP, BEQ, BNE, BRA, etc.) sont gérées par leurs méthodes dédiées.
- ▣ Des instructions particulières comme TFR, EXG, SWI ou END ont également des traitements spécifiques.

Si l'instruction n'est pas reconnue, la méthode lève une exception avec un message clair : "Instruction non supportée".

En résumé, cette fonction est **essentielle** car elle centralise la logique d'exécution : elle reçoit une instruction et la dirige vers la méthode appropriée qui va réellement modifier les registres, la mémoire ou les drapeaux du CPU. C'est le **cœur du moteur d'exécution** du simulateur.



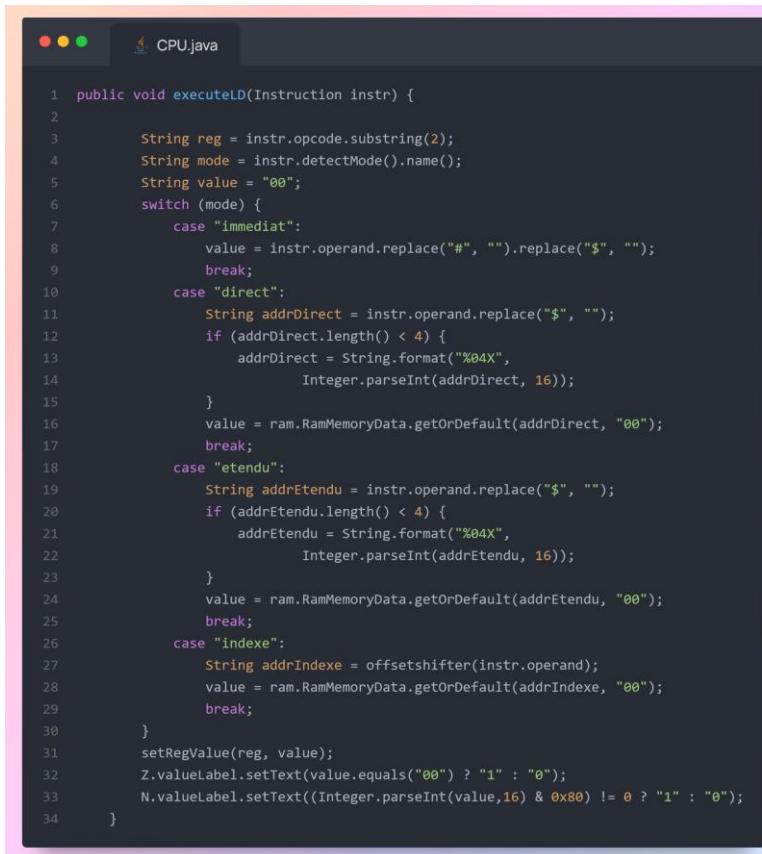
```

1 private void jumpToLabel(String label) {
2     if (!labelsMap.containsKey(label)) throw new IllegalArgumentException("Label non trouvé: " + label);
3
4     currentIndex = labelsMap.get(label);
5     currentAddress = 0;
6     for (int i = 0; i < currentIndex; i++) {
7         String line = codeLines[i].trim();
8         if (line.isEmpty()) continue;
9         String[] parts = line.split("\\s+");
10        Instruction instrTmp = new Instruction(parts[0], parts.length > 1 ? parts[1] : null);
11        currentAddress += instrTmp.getOpcodeHex().length() / 2;
12        if (parts.length > 1) currentAddress += Instruction.filterOperand(parts[1]).length() / 2;
13    }
14
15    RI.controlValueLabel.setText(rom.romMemoryData.getAddress(Instruction.getAddress(currentAddress)));
16    PC.valueLabel.setText(Instruction.getAddress(currentAddress));
17    rom.setCurrent(Instruction.getAddress(currentAddress));
18 }
19
20

```

La méthode `jumpToLabel(String label)` permet au CPU simulé de se déplacer directement vers une ligne de code identifiée par un label spécifique. Elle commence par vérifier si le label existe dans `labelsMap` et lance une exception si ce n'est pas le cas. Ensuite, elle met à jour `currentIndex` pour pointer sur la ligne du label et calcule `currentAddress` en additionnant les tailles des instructions et de leurs opérandes situées avant cette ligne, assurant ainsi que le compteur de programme (PC) reflète la position exacte dans la mémoire. Enfin, elle met à jour l'interface graphique en affichant l'instruction courante dans le registre RI, en positionnant le PC et en indiquant la position dans la ROM. Cette méthode est essentielle pour gérer correctement les instructions de saut et les branchements conditionnels dans l'émulation.

### Les méthodes d'exécution des instructions (CPU.java)



```

1 public void executeID(Instruction instr) {
2
3     String reg = instr.opcode.substring(2);
4     String mode = instr.detectMode().name();
5     String value = "00";
6     switch (mode) {
7         case "immédiat":
8             value = instr.operand.replace("#", "").replace("$", "");
9             break;
10        case "direct":
11            String addrDirect = instr.operand.replace("$", "");
12            if (addrDirect.length() < 4) {
13                addrDirect = String.format("%04X",
14                    Integer.parseInt(addrDirect, 16));
15            }
16            value = ram.RamMemoryData.getOrDefault(addrDirect, "00");
17            break;
18        case "étendu":
19            String addrEtendu = instr.operand.replace("$", "");
20            if (addrEtendu.length() < 4) {
21                addrEtendu = String.format("%04X",
22                    Integer.parseInt(addrEtendu, 16));
23            }
24            value = ram.RamMemoryData.getOrDefault(addrEtendu, "00");
25            break;
26        case "indexé":
27            String addrIndexe = offsetshifter(instr.operand);
28            value = ram.RamMemoryData.getOrDefault(addrIndexe, "00");
29            break;
30    }
31    setRegValue(reg, value);
32    Z.valueLabel.setText(value.equals("00") ? "1" : "0");
33    N.valueLabel.setText((Integer.parseInt(value, 16) & 0x80) != 0 ? "1" : "0");
34 }

```

La méthode executeLD(Instruction instr) simule l'exécution des instructions de chargement (LD) dans le CPU. Elle commence par déterminer le registre cible à partir de l'opcode (par exemple, LDA cible le registre A) et détecte le mode d'adressage de l'instruction (immédiat, direct, étendu ou indexé). Ensuite, elle récupère la valeur à charger selon ce mode :

- En mode **immédiat**, la valeur est directement extraite de l'opérande, en supprimant # ou \$.
- En mode **direct** ou **étendu**, l'adresse est normalisée sur 4 caractères hexadécimaux et la valeur est lue dans la mémoire RAM à cette adresse.
- En mode **indexé**, l'adresse effective est calculée via offsetshifter et la valeur correspondante est récupérée de la RAM.

Enfin, la méthode met à jour le registre cible avec la valeur récupérée et ajuste les **flags** de condition : le flag Z (Zero) est mis à 1 si la valeur est 00 et le flag N (Negative) est défini selon le bit de poids fort de la valeur chargée. Cette fonction est cruciale car elle permet de transférer correctement les données depuis la mémoire vers les registres, ce qui est une opération fondamentale dans tout processeur.



```

1  public void executeLD(Instruction instr) {
2      String reg = instr.opcode.substring(2);
3      String operand = instr.operand;
4      String mode = instr.detectMode().name();
5      String value = getRegValue(reg);
6
7      String highByte = value.length() > 2 ? value.substring(0, value.length() - 2) : "00";
8      String lowByte = value.length() > 2 ? value.substring(value.length() - 2) : value;
9
10     switch (mode) {
11         case "direct":
12             String directAddr = DP.valueLabel.getText() + operand.substring(1);
13             if (value.length() > 2) {
14                 ram.RamMemoryData.put(directAddr, highByte);
15                 int nextAddr = Integer.parseInt(directAddr, 16) + 1;
16                 ram.RamMemoryData.put(String.format("%04X", nextAddr), lowByte);
17             } else {
18                 ram.RamMemoryData.put(directAddr, lowByte);
19             }
20             break;
21         case "etendu":
22             String extAddr = operand.substring(1);
23             if (value.length() > 2) {
24                 ram.RamMemoryData.put(extAddr, highByte);
25                 int nextAddr = Integer.parseInt(extAddr, 16) + 1;
26                 ram.RamMemoryData.put(String.format("%04X", nextAddr), lowByte);
27             } else {
28                 ram.RamMemoryData.put(extAddr, lowByte);
29             }
30             break;
31         case "indexe":
32             String indexAddr = offsetshifter(operand.replace("$", ""));
33             if (value.length() > 2) {
34                 ram.RamMemoryData.put(indexAddr, highByte);
35                 int nextAddr = Integer.parseInt(indexAddr, 16) + 1;
36                 ram.RamMemoryData.put(String.format("%04X", nextAddr), lowByte);
37             } else {
38                 ram.RamMemoryData.put(indexAddr, lowByte);
39             }
40             break;
41     }
42
43     ram.updateRAM(ram.RamMemoryData);
44 }

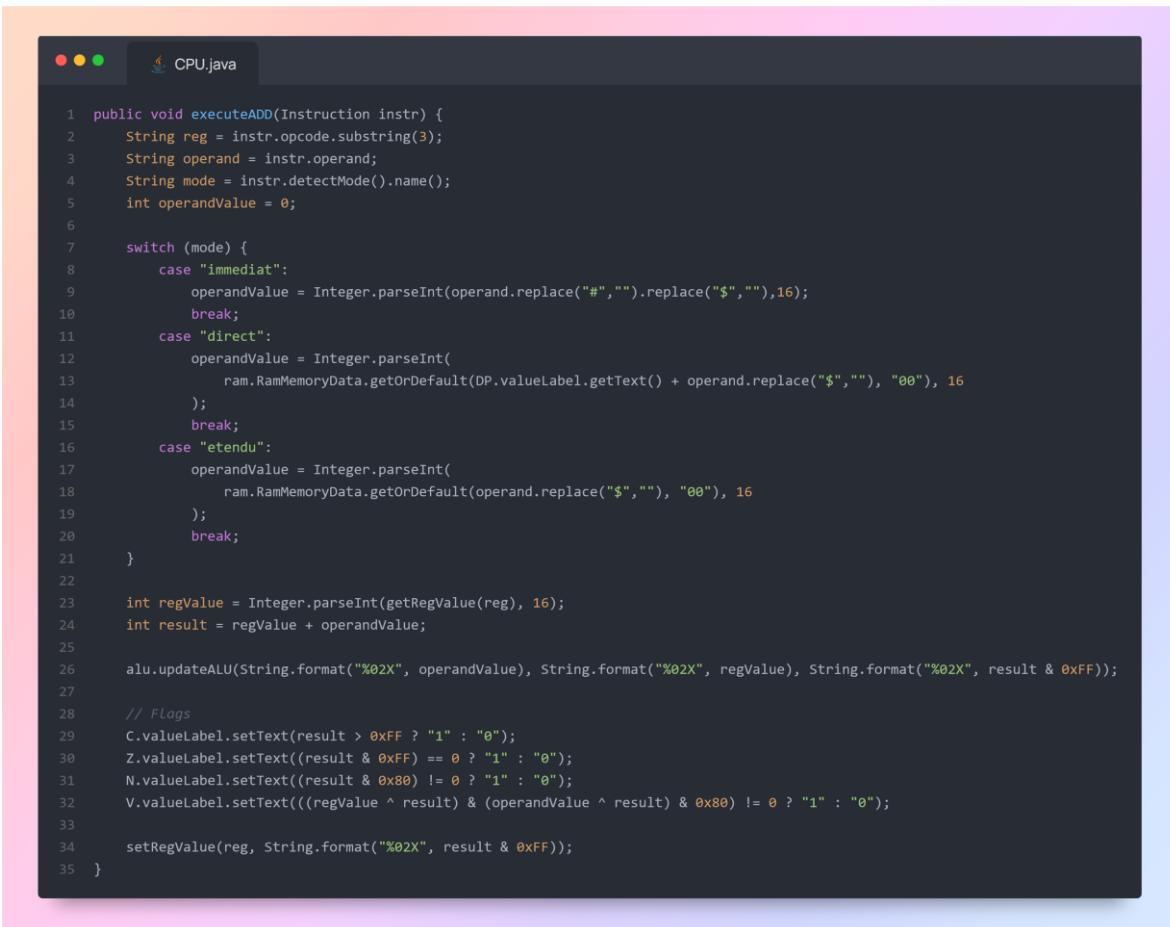
```

La méthode executeST(Instruction instr) gère les instructions de stockage (ST) dans la mémoire. Elle commence par identifier le registre source à partir de l'opcode (STA → A, STX → X, etc.) et récupère sa valeur actuelle. Cette valeur est ensuite divisée en **octets haut et bas** si elle est sur 16 bits, sinon seul l'octet bas est utilisé.

Selon le **mode d'adressage** de l'instruction :

- En mode **direct**, l'adresse finale est calculée en combinant le registre DP et l'opérande, puis la valeur du registre est stockée dans la RAM à cette adresse (sur un ou deux octets selon la taille du registre).
- En mode **étendu**, l'adresse est directement extraite de l'opérande et la valeur du registre est écrite sur un ou deux octets.
- En mode **indexé**, l'adresse effective est calculée via la fonction offsetshifter et la valeur est également stockée sur un ou deux octets.

Enfin, la méthode met à jour la mémoire RAM graphique via ram.updateRAM(ram.RamMemoryData). Cette fonction est essentielle pour transférer les données des registres vers la mémoire, permettant ainsi la sauvegarde de l'état du processeur.



```
1 public void executeADD(Instruction instr) {
2     String reg = instr.opcode.substring(3);
3     String operand = instr.operand;
4     String mode = instr.detectMode().name();
5     int operandValue = 0;
6
7     switch (mode) {
8         case "immediat":
9             operandValue = Integer.parseInt(operand.replace("#", "").replace("$", ""), 16);
10            break;
11        case "direct":
12            operandValue = Integer.parseInt(
13                ram.RamMemoryData.getOrDefault(DP.valueLabel.getText() + operand.replace("$", ""), "00"), 16
14            );
15            break;
16        case "etendu":
17            operandValue = Integer.parseInt(
18                ram.RamMemoryData.getOrDefault(operand.replace("$", ""), "00"), 16
19            );
20            break;
21    }
22
23    int regValue = Integer.parseInt(getRegValue(reg), 16);
24    int result = regValue + operandValue;
25
26    alu.updateALU(String.format("%02X", operandValue), String.format("%02X", regValue), String.format("%02X", result & 0xFF));
27
28    // Flags
29    C.valueLabel.setText(result > 0xFF ? "1" : "0");
30    Z.valueLabel.setText((result & 0xFF) == 0 ? "1" : "0");
31    N.valueLabel.setText((result & 0x80) != 0 ? "1" : "0");
32    V.valueLabel.setText(((regValue ^ result) & (operandValue ^ result)) & 0x80) != 0 ? "1" : "0");
33
34    setRegValue(reg, String.format("%02X", result & 0xFF));
35 }
```

La méthode executeADD(Instruction instr) réalise l'instruction **ADD** pour les registres A ou B. Elle commence par déterminer le **registre cible** et l'**opérande** associée, puis identifie le **mode d'adressage** utilisé (immédiat, direct ou étendu).

Selon le mode : pour **immédiat**, l'opérande est directement convertie depuis l'hexadécimal ; pour **direct**, l'opérande est récupérée depuis la RAM en combinant le registre DP et l'adresse ; pour **étendu**, l'opérande est lue directement à partir de l'adresse spécifiée.

Ensuite, la méthode additionne la valeur du registre avec celle de l'opérande. Les **drapeaux du processeur** sont mis à jour : **C** pour le dépassement, **Z** si le résultat est nul, **N** si le bit de signe est 1, et **V** si un dépassement signé se produit.

Enfin, le registre cible est mis à jour avec le résultat, et l'ALU est actualisée pour refléter les valeurs utilisées et le résultat final.

```

1  public void executeSUB(Instruction instr) {
2      String reg = instr.opcode.substring(3);
3      String operand = instr.operand;
4      String mode = instr.detectMode().name();
5      int operandValue = 0;
6
7      switch (mode) {
8          case "immédiat":
9              operandValue = Integer.parseInt(operand.replace("#", "").replace("$", ""), 16);
10             break;
11         case "direct":
12             operandValue = Integer.parseInt(
13                 ram.RamMemoryData.getOrDefault(DP.valueLabel.getText() + operand.replace("$", ""), "00"), 16
14             );
15             break;
16         case "étendu":
17             operandValue = Integer.parseInt(
18                 ram.RamMemoryData.getOrDefault(operand.replace("$", ""), "00"), 16
19             );
20             break;
21     }
22
23     int regValue = Integer.parseInt(getRegValue(reg), 16);
24     int result = regValue - operandValue;
25
26     alu.updateALU(String.format("%02X", operandValue), String.format("%02X", regValue), String.format("%02X", result & 0xFF));
27
28     // Flags
29     C.valueLabel.setText(result < 0 ? "1" : "0"); // Carry / Borrow
30     Z.valueLabel.setText((result & 0xFF) == 0 ? "1" : "0"); // Zero
31     N.valueLabel.setText((result & 0x80) != 0 ? "1" : "0"); // Negative
32     V.valueLabel.setText(((regValue ^ operandValue) & (regValue ^ result) & 0x80) != 0 ? "1" : "0"); // Overflow
33
34     setRegValue(reg, String.format("%02X", result & 0xFF));
35 }

```

La méthode `executeSUB(Instruction instr)` gère l'instruction **SUB** pour un registre donné (A ou B). Elle commence par identifier le registre cible et l'opérande, puis détermine le **mode d'adressage** (immédiat, direct ou étendu).

En fonction du mode, elle récupère la valeur de l'opérande : pour le mode **immédiat**, elle convertit directement la valeur hexadécimale ; pour le mode **direct**, elle lit la valeur en RAM en utilisant le registre DP et l'adresse ; pour le mode **étendu**, elle lit la valeur directement depuis l'adresse fournie.

Ensuite, la méthode effectue la **soustraction** entre la valeur du registre et l'opérande, met à jour l'**ALU** avec les valeurs du registre, de l'opérande et le résultat. Les **drapeaux** sont mis à jour : **C** pour l'emprunt, **Z** si le résultat est nul, **N** si le bit de signe est actif, et **V** si un dépassement signé se produit. Enfin, le registre cible est actualisé avec le résultat de la soustraction.

```

1  public void executeCMP(Instruction instr) {
2      String reg = instr.opcode.substring(3);
3      int regVal = Integer.parseInt(getRegValue(reg), 16);
4      String operand = instr.operand;
5      String mode = instr.detectMode().name();
6      int memVal = 0;
7
8      switch (mode) {
9          case "immédiat":
10             memVal = Integer.parseInt(operand.replace("#", "").replace("$", ""), 16);
11             break;
12         case "direct":
13             memVal = Integer.parseInt(rom.romMemoryData.getOrDefault(DP.valueLabel.getText() + operand.replace("$", ""), "00"), 16);
14             break;
15         case "étendu":
16             memVal = Integer.parseInt(rom.romMemoryData.getOrDefault(operand.replace("$", ""), "00"), 16);
17             break;
18     }
19
20     int result = regVal - memVal;
21     Z.valueLabel.setText((result & 0xFF) == 0 ? "1" : "0");
22     N.valueLabel.setText((result & 0x80) != 0 ? "1" : "0");
23     C.valueLabel.setText(result < 0 ? "1" : "0");
24 }

```

La méthode executeCMP(Instruction instr) effectue une **comparaison** entre la valeur d'un registre (A ou B) et une opérande selon le **mode d'adressage**. Elle commence par extraire la valeur du registre et de l'opérande, qui peut être immédiate, directe ou étendue. Ensuite, elle effectue une **soustraction logique** du registre moins l'opérande, mais sans stocker le résultat dans le registre. Seuls les **drapeaux** sont mis à jour : **Z** si le résultat est nul, **N** si le bit de signe est actif, et **C** si un emprunt est nécessaire (résultat négatif). Cette opération permet de vérifier la relation entre le registre et l'opérande sans modifier le registre.



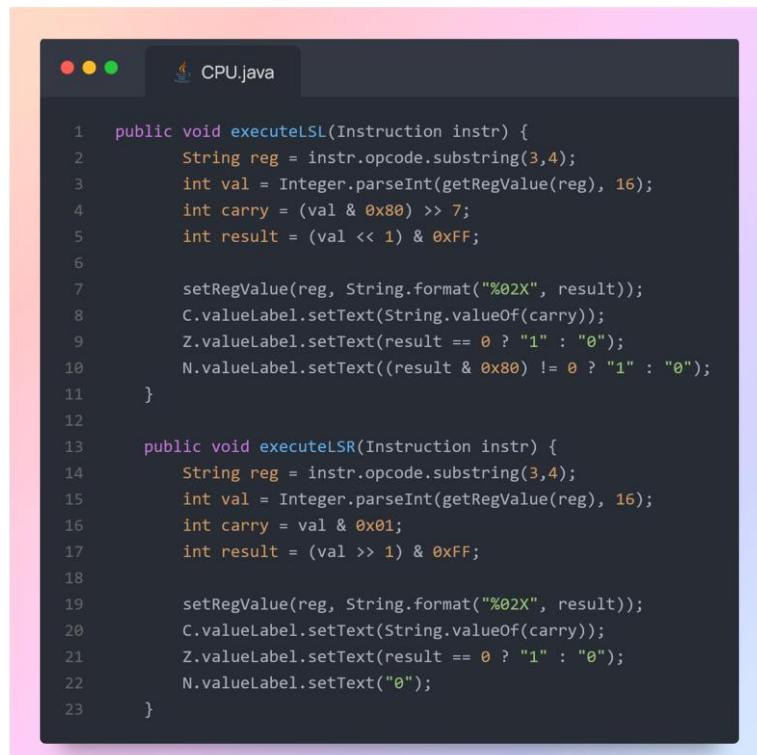
```

1  public void executeINC(Instruction instr) {
2      String reg = instr.opcode.substring(3);
3      int result = Integer.parseInt(getRegValue(reg), 16) + 1;
4
5      Z.valueLabel.setText((result & 0xFF) == 0 ? "1" : "0");
6      N.valueLabel.setText((result & 0x80) != 0 ? "1" : "0");
7      V.valueLabel.setText((Integer.parseInt(getRegValue(reg), 16) == 0x7F) ? "1" : "0");
8
9      setRegValue(reg, String.format("%02X", result & 0xFF));
10 }
11
12 public void executeDEC(Instruction instr) {
13     String reg = instr.opcode.substring(3);
14     int currentValue = Integer.parseInt(getRegValue(reg), 16);
15     int result = currentValue - 1;
16
17     Z.valueLabel.setText((result & 0xFF) == 0 ? "1" : "0");
18     N.valueLabel.setText((result & 0x80) != 0 ? "1" : "0");
19     V.valueLabel.setText((currentValue == 0x80) ? "1" : "0");
20
21     setRegValue(reg, String.format("%02X", result & 0xFF));
22 }

```

Ces deux méthodes gèrent **l'incrémentation et la décrémentation** d'un registre (A ou B).

La méthode executeINC(Instruction instr) augmente la valeur du registre de 1, puis met à jour les **drapeaux** : **Z** si le résultat est nul, **N** si le bit de signe est actif, et **V** si un dépassement de capacité (overflow) se produit, c'est-à-dire si la valeur initiale était 0x7F et la méthode executeDEC(Instruction instr) diminue la valeur du registre de 1, et met à jour les mêmes drapeaux : **Z** si le résultat est nul, **N** si le bit de signe est actif, et **V** si un dépassement se produit lorsque la valeur initiale était 0x80, ces opérations permettent de manipuler le registre tout en tenant compte de l'état logique pour le processeur.



```

1  public void executeSL(Instruction instr) {
2      String reg = instr.opcode.substring(3,4);
3      int val = Integer.parseInt(getRegValue(reg), 16);
4      int carry = (val & 0x80) >> 7;
5      int result = (val << 1) & 0xFF;
6
7      setRegValue(reg, String.format("%02X", result));
8      C.valueLabel.setText(String.valueOf(carry));
9      Z.valueLabel.setText(result == 0 ? "1" : "0");
10     N.valueLabel.setText((result & 0x80) != 0 ? "1" : "0");
11 }
12
13 public void executeLSR(Instruction instr) {
14     String reg = instr.opcode.substring(3,4);
15     int val = Integer.parseInt(getRegValue(reg), 16);
16     int carry = val & 0x01;
17     int result = (val >> 1) & 0xFF;
18
19     setRegValue(reg, String.format("%02X", result));
20     C.valueLabel.setText(String.valueOf(carry));
21     Z.valueLabel.setText(result == 0 ? "1" : "0");
22     N.valueLabel.setText("0");
23 }

```

Ces deux méthodes effectuent des **décalages logiques** sur un registre (A ou B) et mettent à jour les drapeaux du processeur.

La méthode executeLSL(Instruction instr) réalise un **décalage logique à gauche (LSL)** : le bit le plus significatif (bit 7) est placé dans le drapeau **C** (carry), la valeur du registre est multipliée par 2 (décalée d'un bit à gauche), et les drapeaux **Z** et **N** sont mis à jour selon que le résultat est nul ou négatif.

La méthode executeLSR(Instruction instr) réalise un **décalage logique à droite (LSR)** : le bit le moins significatif (bit 0) est placé dans le drapeau **C**, le registre est divisé par 2 (décalé d'un bit à droite), **Z** est mis à jour si le résultat est nul, et **N** est toujours remis à 0 car le décalage à droite logique ne produit jamais de bit de signe.



```
1 public void executeROL(Instruction instr) {
2     String reg = instr.opcode.substring(3,4);
3     int val = Integer.parseInt(getRegValue(reg), 16);
4     int carryIn = Integer.parseInt(C.valueLabel.getText());
5     int carryOut = (val & 0x80) >> 7;
6     int result = ((val << 1) & 0xFF) | carryIn;
7
8     setRegValue(reg, String.format("%02X", result));
9     C.valueLabel.setText(String.valueOf(carryOut));
10    Z.valueLabel.setText(result == 0 ? "1" : "0");
11    N.valueLabel.setText((result & 0x80) != 0 ? "1" : "0");
12 }
13
14 public void executeROR(Instruction instr) {
15     String reg = instr.opcode.substring(3,4);
16     int val = Integer.parseInt(getRegValue(reg), 16);
17     int carryIn = Integer.parseInt(C.valueLabel.getText());
18     int carryOut = val & 0x01;
19     int result = ((carryIn << 7) | (val >> 1)) & 0xFF;
20
21     setRegValue(reg, String.format("%02X", result));
22     C.valueLabel.setText(String.valueOf(carryOut));
23     Z.valueLabel.setText(result == 0 ? "1" : "0");
24     N.valueLabel.setText((result & 0x80) != 0 ? "1" : "0");
25 }
26 }
```

Ces méthodes gèrent les **rotations à travers le drapeau de retenue (carry)** pour les registres A ou B :

### 1. executeROL (Rotate Left through Carry) :

- ▣ Décale tous les bits du registre vers la gauche.
- ▣ Le bit 7 (le plus significatif) devient le **carry**.
- ▣ Le bit 0 du registre reçoit l'ancienne valeur du **carry**.
- ▣ Les drapeaux **Z** et **N** sont mis à jour selon le résultat.

### 2. executeROR (Rotate Right through Carry) :

- ▣ Décale tous les bits du registre vers la droite.
- ▣ Le bit 0 devient le **carry**.
- ▣ Le bit 7 du registre reçoit l'ancienne valeur du **carry**.
- ▣ Les drapeaux **Z** et **N** sont également mis à jour.

Ces opérations sont utiles pour manipuler des bits sur plusieurs octets ou pour les calculs arithmétiques avec retenue.



```
1 public void executeCLR(Instruction instr) {
2     setRegValue(instr.opcode.substring(3), "00");
3     Z.valueLabel.setText("1");
4     N.valueLabel.setText("0");
5     V.valueLabel.setText("0");
6     C.valueLabel.setText("0");
7 }
8
9 public void executeSWI(Instruction instr) {
10     currentLineIndex = codeLines.length; JOptionPane.showMessageDialog(null, "Execution step by step done !");
11 }
12
13 public void executeEND(Instruction instr) {
14     currentLineIndex = codeLines.length; JOptionPane.showMessageDialog(null, "Execution step by step done !");
15 }
16
17 public void executeCOM(Instruction instr) {
18     String reg = instr.opcode.substring(3);
19     int val = Integer.parseInt(getRegValue(reg), 16);
20     int complementValue = (~val) & 0xFF;
21     setRegValue(reg, String.format("%02X", complementValue));
22     Z.valueLabel.setText(complementValue == 0 ? "1" : "0");
23     N.valueLabel.setText((complementValue & 0x80) != 0 ? "1" : "0");
24     C.valueLabel.setText(complementValue > 0xFF ? "1" : "0");
25 }
```

Ces méthodes servent à gérer certaines opérations sur les registres et le contrôle du programme : executeCLR efface le contenu d'un registre en le mettant à zéro et ajuste les drapeaux du processeur (Z pour zéro, N pour négatif, V pour overflow et C pour carry), executeSWI et executeEND arrêtent l'exécution du programme étape par étape en affichant un message de fin, tandis que executeCOM effectue le complément à un d'un registre en inversant tous ses bits, puis met à jour les drapeaux Z (si le résultat est zéro), N (si le bit de poids fort est à 1) et C (selon le dépassement de 8 bits).



```
1 public void executeNEG(Instruction instr) {
2     String reg = instr.opcode.substring(3);
3     int val = Integer.parseInt(getRegValue(reg), 16);
4     int negVal = (~val + 1) & 0xFF;
5     setRegValue(reg, String.format("%02X", negVal));
6     C.valueLabel.setText(negVal == 0 ? "0" : "1");
7     Z.valueLabel.setText(negVal == 0 ? "1" : "0");
8 }
9
10 public void executeTFR(Instruction instr) {
11     String[] regs = instr.operand.split(",");
12     setRegValue(regs[1].trim(), getRegValue(regs[0].trim()));
13 }
14
15 public void executeEXG(Instruction instr) {
16     String[] regs = instr.operand.split(",");
17     String temp = getRegValue(regs[1].trim());
18     setRegValue(regs[1].trim(), getRegValue(regs[0].trim()));
19     setRegValue(regs[0].trim(), temp);
```

```

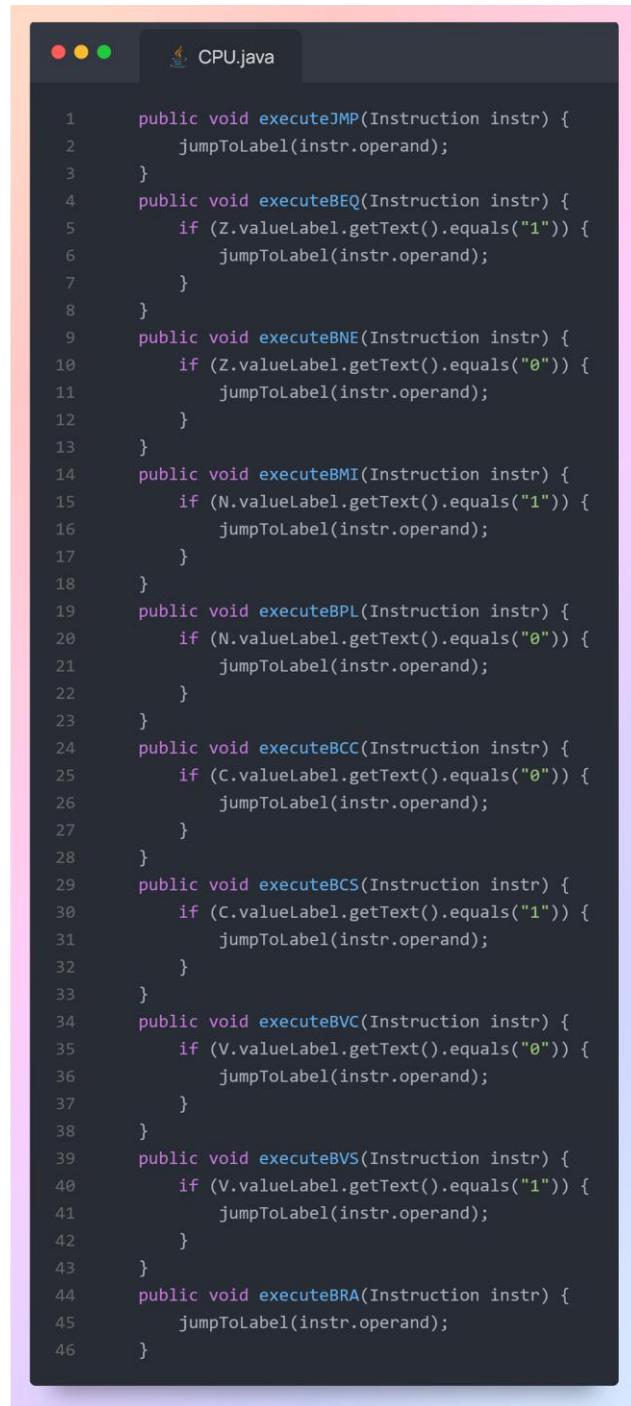
1  public void executeRTS(Instruction instr) {
2      if (!callStack.isEmpty()) {
3          currentAddress = callStack.pop();
4          int index = 0;
5          int addr = 0xFE00;
6
7          for (; index < codeLines.length; index++) {
8              if (codeLines[index].isEmpty()) continue;
9
10             int step = 1;
11             String line = codeLines[index].trim();
12             String[] parts = line.split("\\s+");
13
14             if (parts.length > 1) {
15                 String op = Instruction.FilterOperand(parts[1]);
16                 step += op.length() / 2;
17             }
18
19             if ((addr + step) > currentAddress) break;
20             addr += step;
21         }
22
23         currentLineIndex = index;
24
25         RI.controlValueLabel.setText(rom.ramMemoryData.get(Instruction.getAddress(currentAddress)));
26         PC.valueLabel.setText(Instruction.getAddress(currentAddress));
27         rom.setCurrent(Instruction.getAddress(currentAddress));
28     } else {
29         currentLineIndex = codeLines.length;
30     }
31 }
32 public void executeAND(Instruction instr) {
33
34     String reg = instr.opcode.endsWith("A") ? "A" : "B";
35     String mode = instr.detectMode().name();
36     int operandValue = 0;
37
38     switch (mode) {
39         case "immédiat":
40             operandValue = Integer.parseInt(
41                     instr.operand.replace("#", "").replace("$", ""), 16);
42             break;
43
44         case "direct":
45             operandValue = Integer.parseInt(
46                     ram.RamMemoryData.getOrDefault(
47                         DP.valueLabel.getText() + instr.operand.replace("$", ""),
48                         "00"),
49                     16);
50             break;
51
52         case "étendue":
53             operandValue = Integer.parseInt(
54                     ram.RamMemoryData.getOrDefault(
55                         instr.operand.replace("$", ""),
56                         "00"),
57                     16);
58             break;
59
60         case "indexée":
61             operandValue = Integer.parseInt(
62                     ram.RamMemoryData.getOrDefault(
63                         offsetshifter(instr.operand),
64                         "00"),
65                     16);
66             break;
67     }
68
69     int regValue = Integer.parseInt(getRegValue(reg), 16);
70     int result = regValue & operandValue;
71
72     setRegValue(reg, String.format("%02X", result));
73
74     Z.valueLabel.setText(result == 0 ? "1" : "0");
75     N.valueLabel.setText((result & 0x80) != 0 ? "1" : "0");
76     V.valueLabel.setText("0");
77 }

```

La méthode executeRTS gère le retour d'une sous-routine : elle récupère l'adresse de retour depuis la pile (callStack) et met à jour l'index de la ligne et l'adresse du compteur de programme pour continuer l'exécution après l'appel. Si la pile est vide, elle termine l'exécution du programme. La méthode executeAND réalise une opération logique ET entre le contenu d'un registre (A ou B) et une valeur provenant de l'opérande, qui peut être immédiate, directe, étendue ou indexée, puis stocke le résultat dans le registre et met à jour les drapeaux Z (zéro) et N (négatif), en réinitialisant le drapeau V (overflow).

```
1  public void executeOR(Instruction instr) {
2
3      String reg = instr.opcode.endsWith("A") ? "A" : "B";
4      String mode = instr.detectMode().name();
5      int operandValue = 0;
6
7      switch (mode) {
8          case "immediat":
9              operandValue = Integer.parseInt(
10                  instr.operand.replace("#", "").replace("$", ""), 16);
11              break;
12
13          case "direct":
14              operandValue = Integer.parseInt(
15                  ram.RamMemoryData.getOrDefault(
16                      DP.valueLabel.getText() + instr.operand.replace("$", ""),
17                      "00"),
18                      16);
19              break;
20
21          case "etendu":
22              operandValue = Integer.parseInt(
23                  ram.RamMemoryData.getOrDefault(
24                      instr.operand.replace("$", ""),
25                      "00"),
26                      16);
27              break;
28
29          case "indexe":
30              operandValue = Integer.parseInt(
31                  ram.RamMemoryData.getOrDefault(
32                      offsetshifter(instr.operand),
33                      "00"),
34                      16);
35              break;
36      }
37
38      int regValue = Integer.parseInt(getRegValue(reg), 16);
39      int result = regValue | operandValue;
40
41      setRegValue(reg, String.format("%02X", result));
42
43      Z.valueLabel.setText(result == 0 ? "1" : "0");
44      N.valueLabel.setText((result & 0x80) != 0 ? "1" : "0");
45      V.valueLabel.setText("0");
46  }
47
48  private void executeFOR(Instruction instr) {
49
50      String reg = instr.opcode.endsWith("A") ? "A" : "B";
51      String mode = instr.detectMode().name();
52      int operandValue = 0;
53
54      switch (mode) {
55          case "immediat":
56              operandValue = Integer.parseInt(
57                  instr.operand.replace("#", "").replace("$", ""), 16);
58              break;
59
60          case "direct":
61              operandValue = Integer.parseInt(
62                  ram.RamMemoryData.getOrDefault(
63                      DP.valueLabel.getText() + instr.operand.replace("$", ""),
64                      "00"),
65                      16);
66              break;
67
68          case "etendu":
69              operandValue = Integer.parseInt(
70                  ram.RamMemoryData.getOrDefault(
71                      instr.operand.replace("$", ""),
72                      "00"),
73                      16);
74              break;
75
76          case "indexe":
77              operandValue = Integer.parseInt(
78                  ram.RamMemoryData.getOrDefault(
79                      offsetshifter(instr.operand),
80                      "00"),
81                      16);
82              break;
83
84      }
85
86      int regValue = Integer.parseInt(getRegValue(reg), 16);
87      int result = regValue ^ operandValue;
88
89      setRegValue(reg, String.format("%02X", result));
90
91      Z.valueLabel.setText(result == 0 ? "1" : "0");
92      N.valueLabel.setText((result & 0x80) != 0 ? "1" : "0");
93      V.valueLabel.setText("0");
94  }
```

Les méthodes executeOR et executeEOR effectuent des opérations logiques entre le contenu d'un registre (A ou B) et une valeur provenant de l'opérande. Dans executeOR, l'opération est un OU logique (`|`), tandis que dans executeEOR, c'est un OU exclusif (`^`). Selon le mode d'adressage de l'opérande (immédiat, direct, étendu ou indexé), la valeur est récupérée depuis le code ou la mémoire RAM. Le résultat est stocké dans le registre cible et les drapeaux Z (zéro) et N (négatif) sont mis à jour, tandis que le drapeau V (overflow) est réinitialisé à zéro.



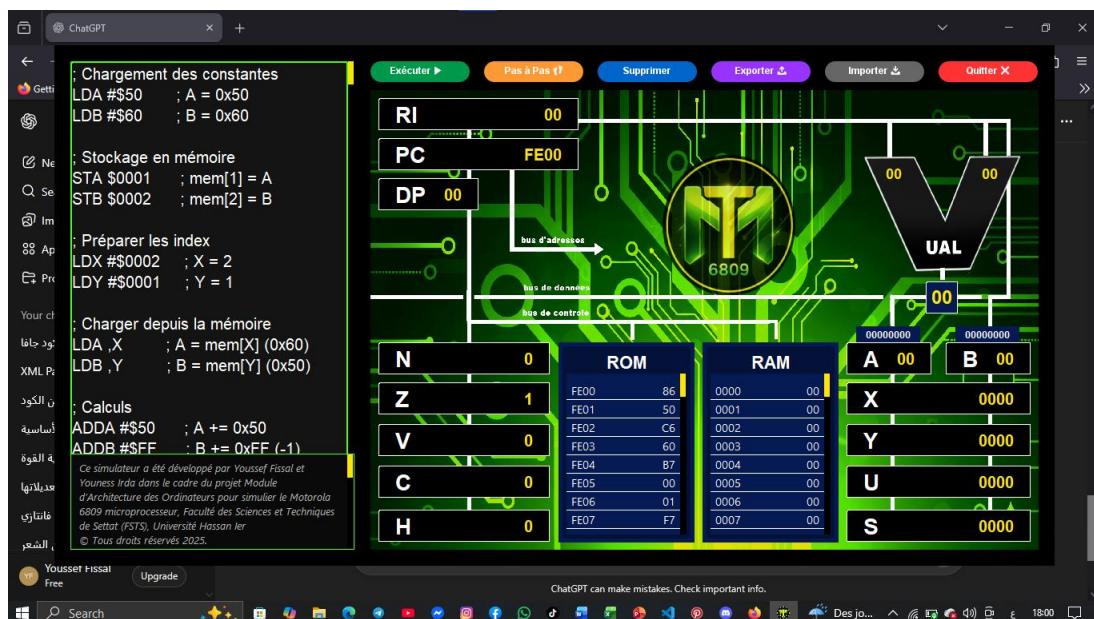
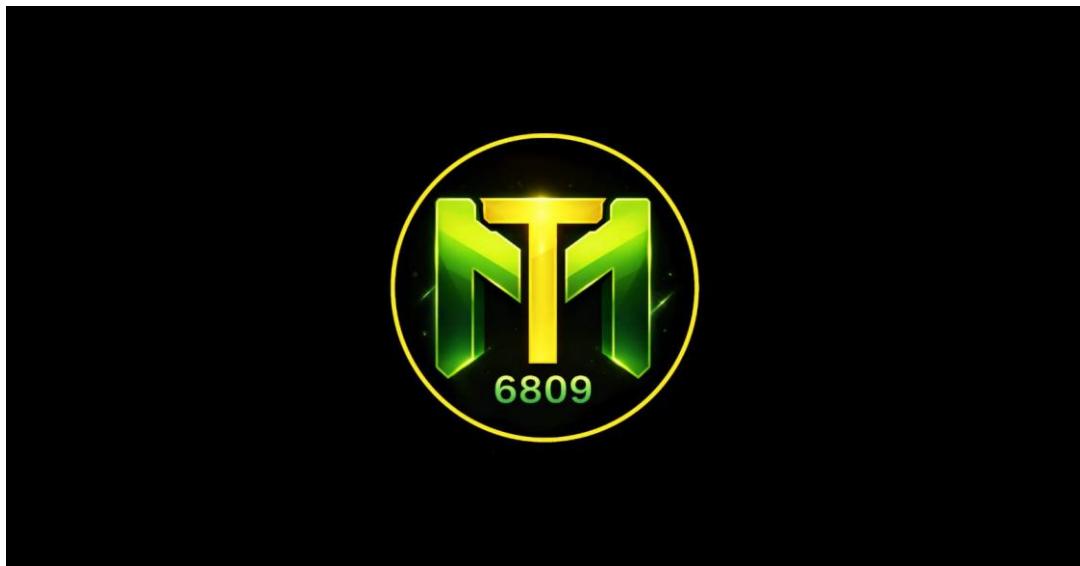
```

1  public void executeJMP(Instruction instr) {
2      jumpToLabel(instr.operand);
3  }
4  public void executeBEQ(Instruction instr) {
5      if (Z.valueLabel.getText().equals("1")) {
6          jumpToLabel(instr.operand);
7      }
8  }
9  public void executeBNE(Instruction instr) {
10     if (Z.valueLabel.getText().equals("0")) {
11         jumpToLabel(instr.operand);
12     }
13 }
14 public void executeBMI(Instruction instr) {
15     if (N.valueLabel.getText().equals("1")) {
16         jumpToLabel(instr.operand);
17     }
18 }
19 public void executeBPL(Instruction instr) {
20     if (N.valueLabel.getText().equals("0")) {
21         jumpToLabel(instr.operand);
22     }
23 }
24 public void executeBCC(Instruction instr) {
25     if (C.valueLabel.getText().equals("0")) {
26         jumpToLabel(instr.operand);
27     }
28 }
29 public void executeBCS(Instruction instr) {
30     if (C.valueLabel.getText().equals("1")) {
31         jumpToLabel(instr.operand);
32     }
33 }
34 public void executeBVC(Instruction instr) {
35     if (V.valueLabel.getText().equals("0")) {
36         jumpToLabel(instr.operand);
37     }
38 }
39 public void executeBVS(Instruction instr) {
40     if (V.valueLabel.getText().equals("1")) {
41         jumpToLabel(instr.operand);
42     }
43 }
44 public void executeBRA(Instruction instr) {
45     jumpToLabel(instr.operand);
46 }

```

Ces méthodes gèrent les instructions de saut et de branchement du processeur. executeJMP et executeBRA effectuent un saut inconditionnel vers un label donné. Les autres méthodes sont des sauts conditionnels : executeBEQ saute si le drapeau Z (zéro) est à 1, executeBNE si Z est à 0, executeBMI si N (négatif) est à 1, executeBPL si N est à 0, executeBCC si C (retenue/carry) est à 0, executeBCS si C est à 1, executeBVC si V (overflow) est à 0, et executeBVS si V est à 1. Chaque méthode utilise `jumpToLabel` pour mettre à jour l'index de la ligne courante et l'adresse du programme afin de poursuivre l'exécution depuis le label ciblé

## Des tests pour le simulateur



: Chargement des constantes  
 LDA #\$50 ; A = 0x50  
 LDB #\$60 ; B = 0x60

; Stockage en mémoire  
 STA \$0001 ; mem[1] = A  
 STB \$0002 ; mem[2] = B

; Préparer les index  
 LDX #\$0002 ; X = 2  
 LDY #\$0001 ; Y = 1

; Charger depuis la mémoire  
 LDA X ; A = mem[X] (0x60)  
 LDB ,Y ; B = mem[Y] (0x50)

; Calculs  
 ADDA #\$50 ; A += 0x50  
 ADDB #\$FF ; B += 0xFF (-1)

Ce simulateur a été développé par Youssif Fissal et Youness Idris dans le cadre du projet Module d'Architecture des Ordinateurs pour simuler le Motorola 6809 microprocesseur, Faculté des Sciences et Techniques de Settat (FSTS), Université Hassan Ier  
 © Tous droits réservés 2025.

: Calculs  
 ADDA #\$50 ; A += 0x50  
 ADDB #\$FF ; B += 0xFF (-1)  
 SUBA #\$D0 ; A = 0x60 + 0x70 combiné

; Échanges et transferts  
 EXG A,B ; swap A et B  
 TFR X,S ; X → S

; Rotations et décalages  
 ROLA ; A << 1  
 RORA ; A >> 1  
 ROLB ; B << 1  
 RORB ; B >> 1  
 LSLA ; A << 1 logique  
 LSLB ; B << 1 logique  
 LSRA ; A >> 1 logique  
 LSRR ; B >> 1 logique

Ce simulateur a été développé par Youssif Fissal et Youness Idris dans le cadre du projet Module d'Architecture des Ordinateurs pour simuler le Motorola 6809 microprocesseur, Faculté des Sciences et Techniques de Settat (FSTS), Université Hassan Ier  
 © Tous droits réservés 2025.

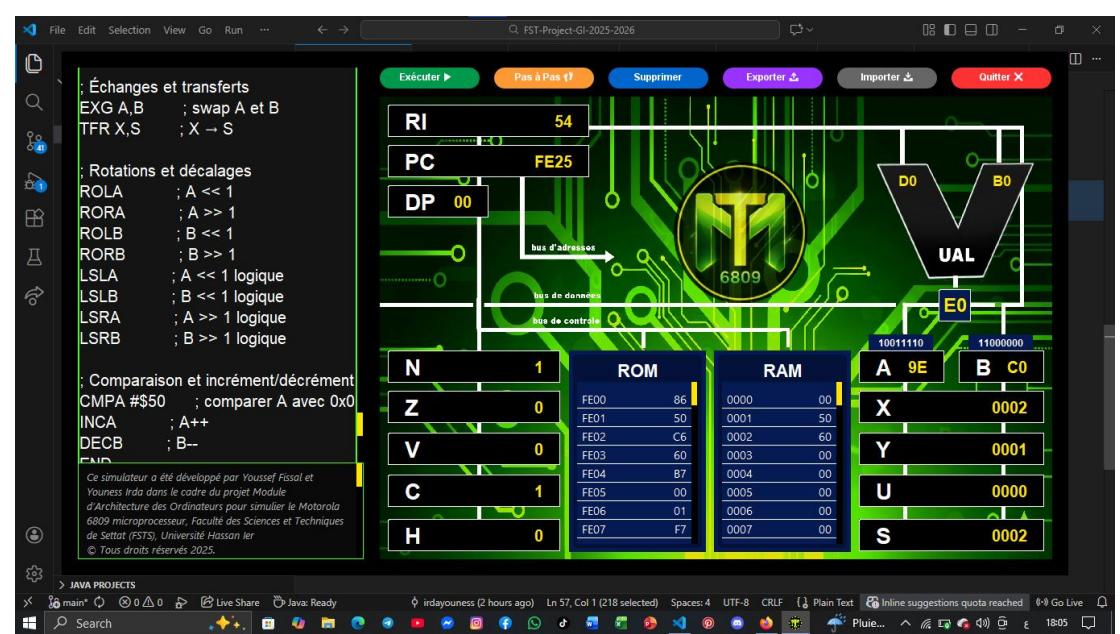
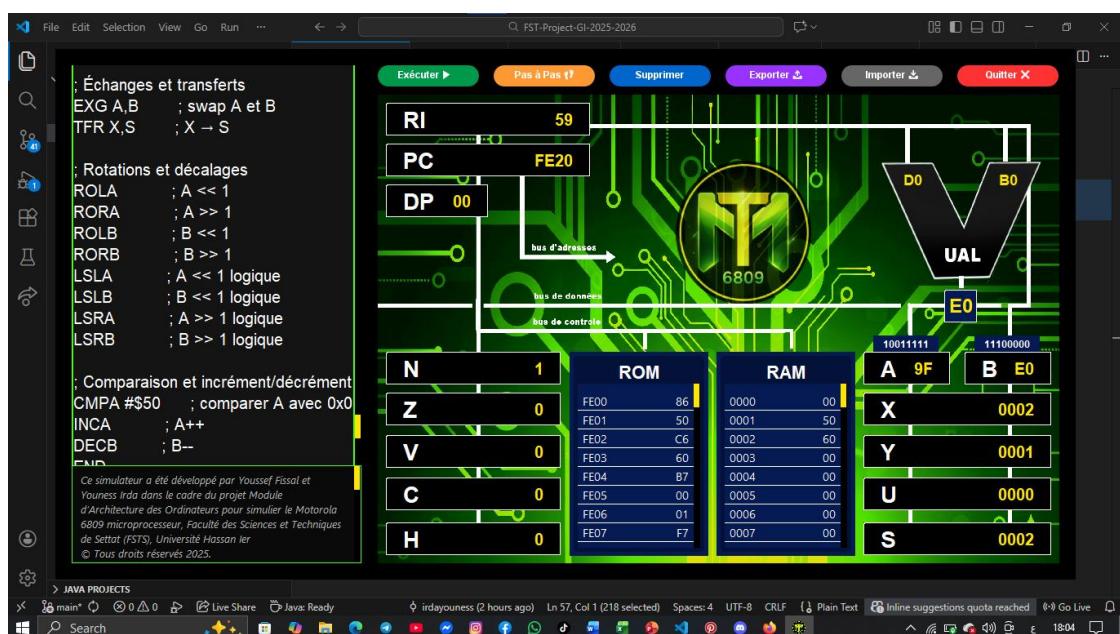
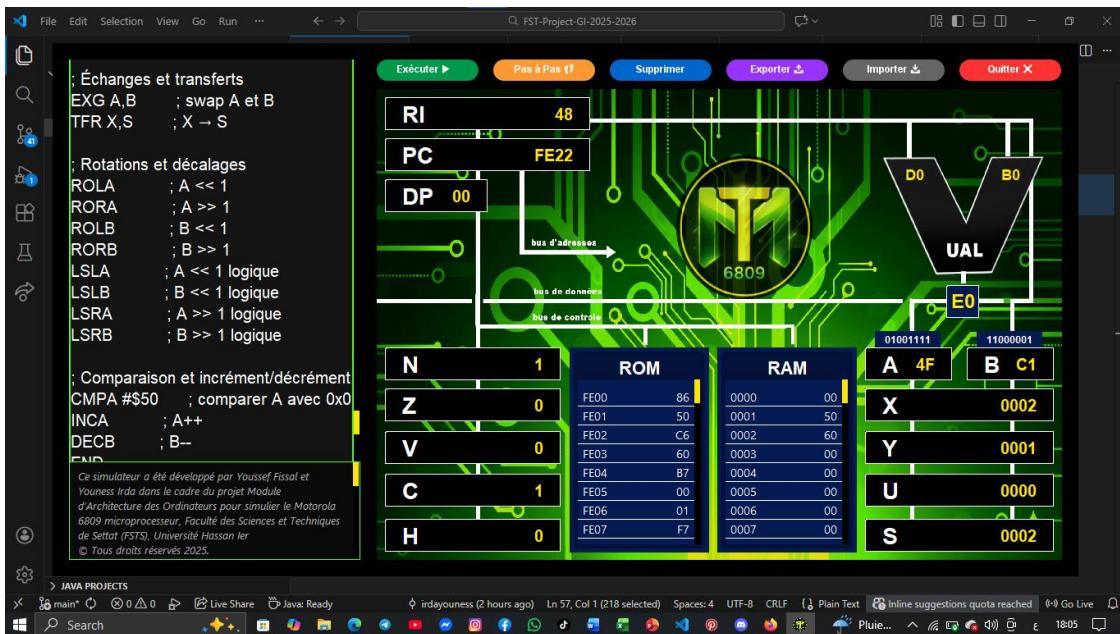
: Charger depuis la mémoire  
 LDA X ; A = mem[X] (0x60)  
 LDB ,Y ; B = mem[Y] (0x50)

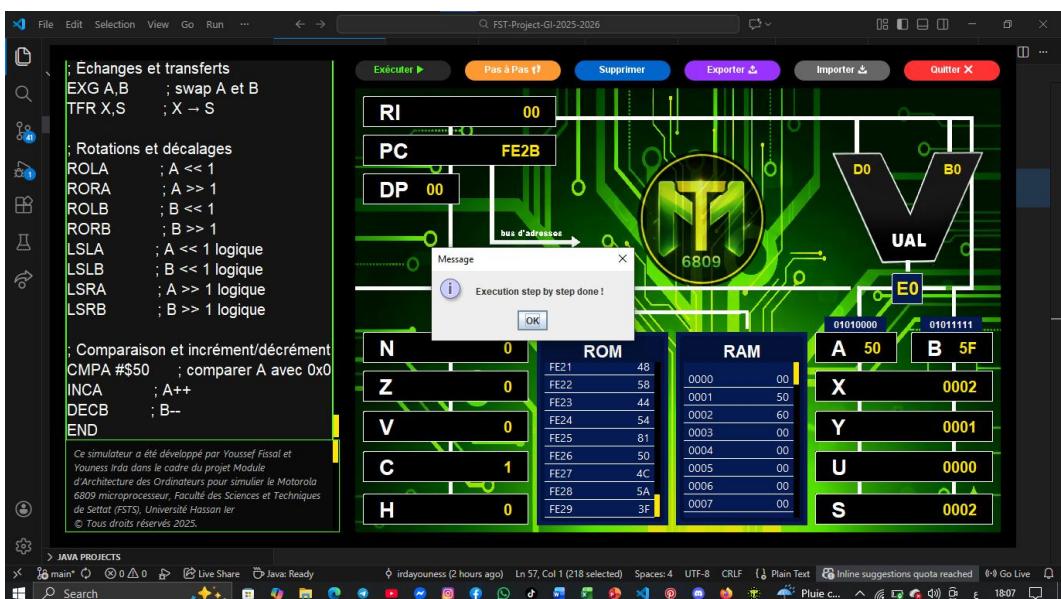
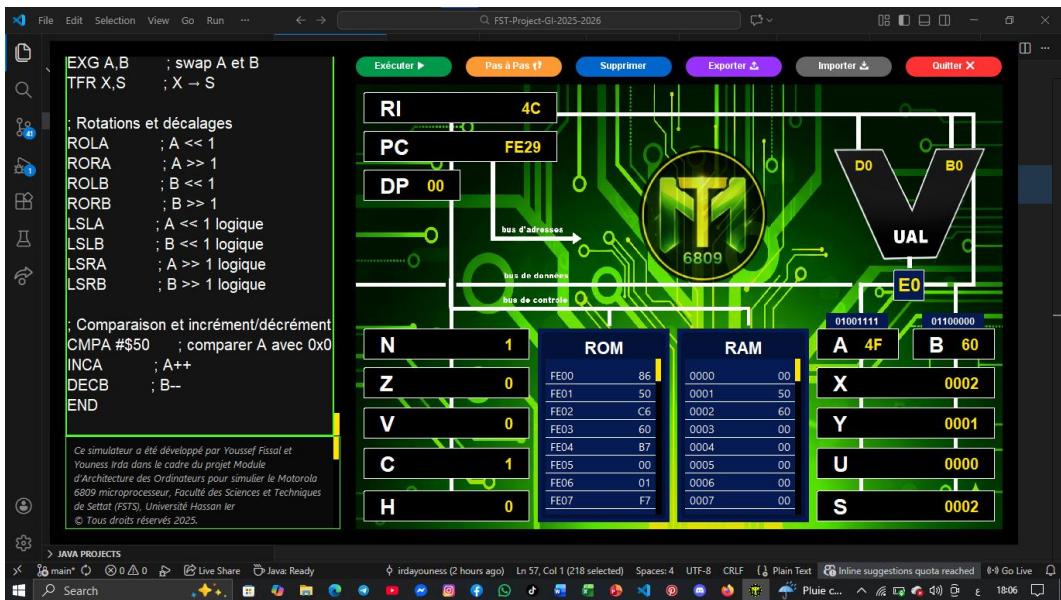
; Calculs  
 ADDA #\$50 ; A += 0x50  
 ADDB #\$FF ; B += 0xFF (-1)  
 SUBA #\$D0 ; A = 0x60 + 0x70 combiné

; Échanges et transferts  
 EXG A,B ; swap A et B  
 TFR X,S ; X → S

; Rotations et décalages  
 ROLA ; A << 1  
 RORA ; A >> 1  
 ROLB ; B << 1  
 RORB ; B >> 1

Ce simulateur a été développé par Youssif Fissal et Youness Idris dans le cadre du projet Module d'Architecture des Ordinateurs pour simuler le Motorola 6809 microprocesseur, Faculté des Sciences et Techniques de Settat (FSTS), Université Hassan Ier  
 © Tous droits réservés 2025.





## Défis et solutions

En raison du temps limité, nous avons dû renoncer à plusieurs fonctionnalités du Motorola 6809. Tout d'abord, nous nous sommes concentrés uniquement sur le déplacement nul dans le mode indexé. Malgré nos efforts pour inclure autant d'instructions que possible avec tous les modes, cela a généré plusieurs bugs dans le code, certains difficiles à corriger. Chaque correction nécessitait souvent de réécrire une partie du code, ce qui détériorait parfois les performances et faisait apparaître de nouveaux bugs.

Nos tentatives pour rendre le code parfait nous ont fait perdre un temps précieux, ce qui nous a empêchés d'implémenter toutes les fonctionnalités prévues. Nous avons finalement intégré seulement 33 instructions. De plus, la traduction exacte du fonctionnement du Motorola 6809 en code a posé plusieurs problèmes, nous obligeant à développer notre propre logique. Même si elle n'était pas strictement identique à l'original, elle fournit néanmoins des résultats cohérents et corrects dans l'interface.

En conclusion, nos efforts pour perfectionner le code ont limité la réalisation complète des capacités du projet, tant en termes de conception de l'interface que de fonctionnalités. Pour optimiser notre temps et améliorer la qualité finale, nous avons également sollicité les conseils d'anciens étudiants et apporté nos touches personnelles, donnant ainsi au projet sa propre identité et sa signature unique.

## **Conclusion et perspectives**

Le projet a permis de simuler le microprocesseur Motorola 6809 avec précision, incluant la gestion de la mémoire, l'émulation des instructions et une interface graphique. Il a facilité la compréhension de l'architecture des microprocesseurs et des défis liés à la simulation.

Le simulateur est fonctionnel et offre une base solide pour l'amélioration future, l'ajout de fonctionnalités et l'enseignement de l'informatique

## **Les Source et Références**

- ✓ ChatGPT : <https://chat.openai.com/>.
- ✓ W3Schools : <https://www.w3schools.com/>.
- ✓ Code to Image Converter : <https://10015.io/tools/code-to-image-converter>.
- ✓ Cours d'architecture des ordinateurs chapitre 3.
- ✓ 6809.uk : <https://6809.uk/>.