Université Hassan 1er
Faculté des Sciences et Techniques Settat

Simulateur de microprocesseur Motorola 6809

Encadré par: **Mr. BENALLA Hicham**

Réalisé par : **Ayoub Chahib**
Romisio Maria Farinha

ANNÉE UNIVERSITAIRE 2025 - 2026

SOMMAIRE :

Présentation du Motorola 6809 :

Le Motorola 6809 est un microprocesseur 8 bits avec certaines capacités 16 bits, conçu par Motorola et introduit en 1978. Il est compatible source avec le Motorola 6800, mais apporte de fortes améliorations (architecture plus avancée, nouvelles instructions, nouveaux registres, meilleurs modes d'adressage). À son époque, il était considéré comme l'un des microprocesseurs 8 bits les plus puissants.On peut identifier plusieurs caractéristiques fondamentales, telles que :

- ✓ Jeu d'instructions flexible et modes d'adressage variés:
  Le Motorola 6809 possède environ 59 instructions fondamentales. Cependant, l'introduction de différents modes d'adressage permet d'obtenir plus de 1460 codes opérationnels possibles. On distingue plusieurs types d'adressage, tels que : inhérent, immédiat, direct, étendu, indexé, relatif, indirect... Cette richesse rend la programmation en assembleur plus puissante et plus flexible, permettant une grande flexibilité dans l'accès aux données et la manipulation de la mémoire.

- ✓ Architecture supérieure :
  Le Motorola 6809 était plus avancé que les autres processeurs 8 bits de son époque car il avait plus de registres et des modes d'adressage plus flexibles, ce qui permettait une programmation plus facile et plus puissante.

- ✓ Support avancé des interruptions et multitâche :
  Le 6809 offrait une gestion des interruptions plus flexible et structurée, facilitant la gestion simultanée de plusieurs tâches ou événements. À l'inverse de nombreux processeurs 8 bits à cette époque, il était capable de gérer des routines d'interruption plus sophistiquées sans compromettre des informations cruciales.

Influence du Motorola 6809 :

Le microprocesseur Motorola 6809 fut l'un des plus puissants de sa catégorie à son époque et connut un succès important puisqu'il fut utilisé dans une multitude d'appareils dans les années 70 et 80. On le retrouvait dans des ordinateurs personnels, mais aussi dans des micro-ordinateurs utilisés dans les foyers, et même dans certaines consoles de jeux vidéo. Il servit également d'électronique principale pour des machines équipées du système d'exploitation multitâche. Par sa puissance et sa flexibilité, le 6809 contribua donc fortement à l'évolution des machines informatiques de la fin des années 70 et du début des années 80.

Introduction générale :

Pour approfondir la connaissance des architectures de microprocesseurs et comprendre les principes des langages bas niveau, comme celui de l'assembleur, l'utilisation d'un simulateur est indispensable. Il est notamment nécessaire pour :

- ✓ Analyse du fonctionnement interne : Le simulateur permet de visualiser comment les instructions sont exécutées ou encore comment les registres sont utilisés et, de manière plus générale, permet de découvrir les différentes parties du processeur 6809, ce qui est difficile à réaliser avec un processeur réel.
- ✓ Expérimentation sans risque : Il s'agit d'un outil idéal pour écrire des programmes, déboguer et tout simplement pour découvrir le fonctionnement du processeur sans avoir peur de le mettre en panne.
- ✓ Accessibilité et utilisation : Les simulateurs permettent de faire tourner le processeur 6809 sur n'importe quelle machine moderne et donc offrent un accès à l'architecture 6809 à tous les étudiants et développeurs sans aucune contrainte d'ordre matérielle ou logiciel.

Buts du simulateur 6809 :

Cette partie présente les objectifs du simulateur du microprocesseur Motorola 6809, en montrant ses objectifs d'utilisation en tant qu'outil pédagogique et comme outil fonctionnel :

- ✓ Faciliter la compréhension du fonctionnement du microprocesseur Motorola 6809 et identifier le rôle des registres, des instructions et des composants internes.
- ✓ Proposer un environnement permettant d'écrire, d'exécuter et de déboger des programmes de bas niveau sans utiliser une vraie machine.
- ✓ Proposer une fonctionnalité pas à pas pour aider l'utilisateur à suivre l'exécution d'un programme et à observer les modifications de l'état du processeur.
- ✓ Montrer comment fonctionne la mémoire dans un ordinateur en faisant la différence entre la mémoire vive (RAM) et la mémoire morte (ROM), et comment le processeur y accède.
- ✓ Proposer un outil pédagogique simple et interactif pour apprendre l'architecture des microprocesseurs et les notions de base de la programmation bas niveau.

Organisation fonctionnelle des fichiers :

Les fichiers ArchitecteInterne.java, Editeur.java, Main.java, RAM.java, ROM.java, Programme.java, représentent différentes composantes du simulateur Motorola 6809. L'ensemble de ces fichiers correspond à la base logicielle du simulateur, réalisant la simulation du comportement du 6809 ainsi que l'exécution de programmes.

Le simulateur du microprocesseur Motorola 6809 est un projet modulaire, dans lequel chaque grande fonctionnalité est gérée dans un fichier Java à part. Cette conception, qui repose sur la programmation orientée objet en Java, offre une meilleure lisibilité du projet grâce à des classes qui ont chacune leur propre rôle. La distinction entre le contrôle du processeur, de la mémoire et de l'interface utilisateur empêche également les mauvaises interactions entre ces entités :

Main.java : permet une navigation globale dans les différentes fonctionnalités du simulateur.

MotoController(.java+.fxml) : .java contrôlle l'interface graphique et **.fxml** permet de générer l'interface graphique.

Logique.java : comprendre la simulation de l'architecte d'un microprocesseur ainsi que l'exécution des instructions.

Memoire.java : représente la mémoire vive utilisée par l'ordinateur pour stocker temporairement des données ou des instructions et représente la mémoire morte contenant des données ou des instructions pouvant être uniquement lues.

OpFlags : Il permet de faire les opérations arithmétiques et gérer les drapeaux.

Architecture du Simulateur Motorola 6809 :

L'architecture globale du simulateur Motorola 6809 repose sur un ensemble de fichiers java, chacun représentant une composante essentielle du système simulé :

Logique.java :

Le fichier Logique.java est au cœur de notre simulation du microprocesseur Motorola 6809. C'est le module qui simule l'architecture interne du processeur, gère les registres, l'état du processeur et qui réalise l'exécution des instructions.C'est celui qui reçoit les instructions, qui les interprète et qui modifie l'état du processeur selon les besoins.

Il doit aussi interagir avec les modules Memoire.java avec lequelle il lit ou écrit les données nécessaires à l'exécution des programmes. Il doit communiquer avec MotoController(.java+.fxml) pour récupérer les instructions à exécuter, et enfin avec nos interfaces graphiques pour pouvoir afficher l'état du processeur tout au long de l'exécution.

Memoire.java :

Le fichier Memoire.java représente la mémoire vive et la mémoire morte du système simulé. Il s'agit d'une mémoire temporaire dans laquelle sont placés à l'exécution les données et les codes des programmes et au même temps d'une mémoire dans laquelle on place des données ou des instructions permanentes qui ne doivent pas être modifiées pendant l'exécution

Cette classe est principalement utilisée par Logique.java, en effet, l'architecte réalise des opérations de lecture et d'écriture dans la mémoire et de récupérer des instructions ou des données nécessaires à l'exécution du programme. Par ailleurs, MotoController(.java+.fxml) et l'interface utilisateur sont également concernés, dans la mesure où l'on peut voir le contenu de la mémoire lors du déroulement d'un programme.

MotoController(.java+.fxml) :

Le fichier MotoController(.java+.fxml) correspond à l'interface de l'utilisateur du simulateur. Il permet à l'utilisateur de saisir, de modifier et d'exécuter des programmes destinés au microprocesseur Motorola 6809.

Quand un programme est lancé, MotoController(.java+.fxml) envoie les instructions à Logique.java pour qu'elles soient interprétées et exécutées. Il communique aussi avec les composants mémoire pour visualiser l'utilisation de la mémoire, et avec les interfaces de visualisation pour suivre l'exécution.

Main.java :

Le fichier Main.java représente le menu principal du simulateur. Il donne à l'utilisateur la possibilité de naviguer à travers les différents modules du simulateur, comme la motocontroller ,la logique ou la mémoire. En effet, on peut considérer que c'est grâce à lui que le simulateur est utilisable dans sa globalité.

Le main est en relation avec tous les autres modules du simulateur puisqu'il commande l'ouverture ou la fermeture de leur interface, mais il n'agit pas au cœur du traitement de l'exécution des instructions.Il permet de séparer la logique avec l'interface.

OpFlags.java

Le fichier OpFlags.java a pour role principal de gérer les drapeaux (flags) du processeur lors de l'exécution des instructions.

Elle est utilisée par le module Logique.java lors de l'exécution des instructions arithmétiques afin de détecter les retenues , débordements, négatif et zero et de mettre à jour les flags .Les informations calculées par OpFlags sont intégrées à l'état interne du CPU et transmises à l'interface utilisateur via MotoController.

Environnement technologiques du simulateur :

JAVA :

- ✓ Programmation orientée objet : La nature orientée objet de Java permet de modéliser le simulateur à travers des classes représentant chacun des composants (processeur, mémoire, interface), et ainsi d'organiser son code de manière intuitive.
- ✓ Portabilité : Java est une technologie multiplateforme, c'est à dire que les applications écrites en Java peuvent être exécutées sur n'importe quel système d'exploitation sans nécessiter une quelconque modification du code. Ainsi, le simulateur sera utilisable sur plusieurs plateformes.

- ✓ Lisibilité et maintenance : Enfin, parmi les nombreux avantages de Java, on peut citer sa syntaxe relativement simple et ses nombreux outils qui facilitent la lecture, la maintenance et l'évolution du code , particulièrement adaptés à un projet pédagogique.

JavaFX:

- ✓ Destiné au développement d'interfaces graphiques en java . Il permet de créer des applications riches, interactives et bien structurées, allant de simples fenêtres à des interfaces complexes.
- ✓ JavaFX favorise une bonne organisation du code en séparant la logique applicative du design graphique. Cette séparation est souvent réalisée grâce à l'utilisation de FXML et des classes contrôleurs, ce qui rend le projet plus lisible et plus facile à maintenir.
- ✓ JavaFX propose une large bibliothèque de composants graphiques tels que boutons, champs de texte, tableaux, menus et zones d'affichage.Ces composants permettent de construire des interfaces complètes et adaptées aux besoins de l'utilisateur.

Problématique du projet :

La compréhension du fonctionnement interne d'un microprocesseur, tel que le Motorola 6809, constitue un défi de taille pour les étudiants en raison notamment de la complexité de son architecture et de la programmation en langage machine. En effet, l'utilisation d'un processeur réel ne permet pas de toujours voir comment les instructions sont exécutées, comment les registres changent et comment le processeur interagit avec la mémoire. De ce fait, la réalisation d'un simulateur représente une aide à la compréhension du microprocesseur et à l'analyse de son comportement.

Voici les principaux défis de conception que nous avons identifiés pour ce projet :

- ✓ Permettre une visualisation claire du déroulement des instructions et de l'évolution des états internes du CPU.
- ✓ Comprendre comment les registres sont utilisés et pourquoi ils sont essentiels dans l'exécution d'un programme bas niveau.
- ✓ Modéliser le fonctionnement d'une mémoire et comment le processeur s'y connecte.
- ✓ Offrir un environnement simple de créer, exécuter et déboguer des programmes en langage assembleur.
- ✓ Réaliser un outil éducatif facile d'accès, sans besoin d'avoir un exemplaire physique de chaque étape.

Composants du simulateur Motorola 6809 :

Main.java :

Interface de navigation du simulateur :

Main.java est un composant fondamental de l'usage du simulateur puisque c'est à partir de lui que l'utilisateur va pouvoir visualiser la première interface. C'est un peu le centre de contrôle de son utilisation où l'on va pouvoir choisir différentes fonctionnalités du simulateur puis revoir étranger par d'autres modules.

Principales fonctionnalités :

- ✓ Rassembler dans une interface, les différentes commandes du simulateur. Permettre une navigation générale dans le simulateur.
- ✓ Donner accès aux fonctionnalités telles que l'aide, autres types de support,...

Interactions avec les autres composants :

- ✓ Permet d'ouvrir le MotoController afin de fournir un environnement de saisie des programmes.
- ✓ Accéder aux interfaces de la mémoire vive (Memoire.java) et de la mémoire morte (ROM.java) afin de visualiser leur contenu.
- ✓ Permet d'afficher l'état du microprocesseur en visualisant son architecture interne (Logique.java).
- ✓ Il permet de séparer la logique avec l'interface.

MotoController(.java+.fxml) :

Interface de programmation et d'exécution :

MotoController(.java+.fxml) est l'interface principale qui permet à l'utilisateur de communiquer avec le simulateur. Il permet à l'utilisateur de créer, modifier et exécuter des programmes pour le microprocesseur Motorola 6809. Cela fait de ce MotoController un outil très utile dans l'apprentissage de la programmation bas niveau et dans l'expérimentation du processeur simulé.

Principales fonctionnalités :

- ✓ Fournir un environnement de saisie de programmes assembleur du Motorola 6809.
- ✓ Permettre l'exécution directe des programmes depuis le MotoController.
- ✓ Offrir des fonctionnalités de débogage telles que l'exécution pas à pas.
- ✓ Mettre en évidence certaines parties du code pour améliorer la lisibilité.

Interactions avec les autres composants :

- ✓ Envoie les instructions saisies par l'utilisateur au module Logique.java pour leur interprétation et exécution.
- ✓ Interagit avec la mémoire (Memoire.java) afin de visualiser l'impact des instructions sur la mémoire.

Enfin, Main.java et MotoController(.java+.fxml) font partie intégrante de l'interface utilisateur du simulateur Motorola 6809. Le menu donne à l'utilisateur l'accès à l'ensemble des fonctionnalités du simulateur ainsi qu'aux modules principaux. Le MotoController(.java+.fxml) est l'espace où l'utilisateur saisit et exécute les programmes, tout en observant les effets de ces actions sur le processeur et la mémoire. L'interaction étroite des composants crée une interface qui permet à l'utilisateur de naviguer sans difficulté dans le simulateur. Cela aboutit à une interface conviviale tout en offrant une expérience d'émulation complète et éducative.

Logique.java :

Cœur logique du simulateur

Logique.java est le cœur logique du simulateur Motorola 6809. Il est responsable de la simulation de l'architecture interne du microprocesseur, incluant la gestion des registres comme les accumulateurs et les pointeurs, l'état du processeur et l'exécution des instructions.

Principales fonctionnalités :

- ✓ Simuler les registres internes du microprocesseur (PC, A, B, X, Y, U, S, etc.).
- ✓ Interpréter et exécuter les instructions de l'ensemble d'instructions Motorola 6809.
- ✓ Gérer les drapeaux de condition liés aux opérations arithmétiques et logiques.
- ✓ Mettre à jour dynamiquement l'état du processeur lors de l'exécution pas à pas.

Interactions avec les autres composants :

- ✓ Reçoit des instructions envoyées par MotoController(.java+.fxml) pour les interpréter et les exécuter.

- ✓ Effectue des opérations de lecture et d'écriture dans la mémoire et la ROM.
- ✓ Met à jour en temps réem l'affichage de l'état interne du processeur pour l'interface utilisateur.

Memoire.java :

Simulation de stockage des données :

Memoire.java représente la mémoire du système simulé. Elle permet de simuler le stockage temporaire et permanante les données et les instructions utttilisées lors de l'exécution des programmes.

Principales fonctionnalités :

- ✓ Simulation de stockage permanante des programmes (stocker les opcodes, post-octect et les opérandes).
- ✓ Simulation de stockage temporaire lors e l'exécution des prrogrammes .

Interactions avec les autres éléments :

- ✓ Mise en œuvre dans Logique.java pour la lecture et l'écriture de données lors de l'exécution.
- ✓ Disponible via MotoController.java pour afficher ou modifier le contenu en mémoire.
- ✓ Présentée à l'utilisateur par une interface graphique spécifique.

OpFlags :

Gestion des drapeaux du processeur :

OpFlags est une classe utilitaire du simulateur du microprocesseur Motorola 6809 dont le role est de modéliser le comportement du registre d'état du processeur et de faire les opérations arithmétiques .Elle regroupe l'ensemble des méthodes nécessaires à l'analyse des résultats produits par les opérations arithmétiques et logiques exécutées par le CPU, afin de déterminer l'état correct des drapeaux de condition.

Principales fonctionnalités :

- ✓ Réalisation des opérations arithmétiques en hexadécimal.
- ✓ Support direct de la logique du processeur simulé .
- ✓ Simulation du comportement des flags du Motorola 6809.
- ✓ Détectction des conditions liées aux flags(débordement, retenue,négatif et zéro).

Interactions avec les autres éléments :

- ✓ OpFlags.java est utilisé par Logique.java lors de l'exécution des instructions arithmétiques afin d'effectuer les calculs et de déterminer l'état des flags. Ses résultats sont intégrés à l'état interne du processeur simulé au niveau de Logique.java.
- ✓ Les flags calculés par OpFlags sont transmis directement à MOTOController.java, qui se charge de les afficher à l'utilisateur à travers l'interface graphique du simulateur .

Émulation du jeu d'instructions du Motorola 6809 :

Principe d'émulation :

L'émulation du jeu d'instructions du microprocesseur Motorola 6809 vise à simuler, via un logiciel, le comportement du processeur réel pendant l'exécution des instructions en langage assembleur.

Dans le simulateur, chaque instruction d'un programme est lue, décodée et exécutée séquentiellement, exactement comme le ferait un processeur Motorola 6809. Le fichier Logique.java(CPU) gère principalement ce processus.

Cycle d'exécution du processeur :

Le simulateur implémente le cycle classique ***fetch-decode-execute*** , permettant de lire l'instruction, de l'analyser et de l'exécuter.

L'exécution des instructions du 6809 assure que les résultats sont proches que possibles de ceux obtenus dans un vrai matériel.

Type d'instructions prises en charge :

Le simulateur prend en charge les principales catégories d'instructions du Motorola 6809 :

- ✓ Instructions de transfert de données,
- ✓ Instructions arithmétiques et logiques,
- ✓ Instructions de test.

Exécution Pas à Pas :

Cette fonctionnalité permet à l'utilisateur d'exécuter le programme instruction par instruction, offrant un contrôle détaillé et une observation du comportement du simulateur à chaque étape.

Visualisation de la Mémoire :

Affichage graphique :

Affichage graphique de la mémoire : Le simulateur propose une visualisation graphique de la mémoire, permettant d'afficher le contenu de manière claire et facilement interprétable par l'utilisateur.

Synchronisation en temps réel : L'état de la mémoire est actualisé dynamiquement au cours de l'exécution des programmes, reflétant les changements dès qu'ils se produisent.

Interaction Utilisateur :

Accès et manipulation de la mémoire : Les utilisateurs peuvent visualiser, modifier (à travers les instructions) ou examiner les données dans la mémoire. Cela permet une interaction directe avec la mémoire via l'interface graphique , essentielle pour tester et déboguer des programmes.

Cette fonctionnalité facilite à comprendre les interactions entre le mémoire et les programmes, une partie cruciale de la programmation au niveau système.

En résumé, le simulateur reproduit de manière précise le comportement du processeur d'origine en émulation du jeu d'instructions Motorola 6809. Des outils de débogage, comme le pas à pas, offrent une analyse détaillée et simplifient la compréhension du fonctionnement des programmes. De plus, la représentation de la mémoire permet à l'usager d'interagir directement avec la mémoire volatile et non-volatile via l'interface graphique, favorisant ainsi une meilleure appréhension des processus de gestion de la mémoire dans les systèmes intégrés.

Implémentation Technique :

Main.java :

Définition de classe :

public class Main extends Application {

Rôle :la classe Main étend Application, indiquant qu'elle sert de lancer l'application (l'interface graphique), la création de la fenêtre, le cycle de vie (démarrage,arrêt), et de créer un lien entre la logique.java et le MotoContoller.java.

Initialisation de l'interface graphique (le lien entre la logique et le MotoController) :

```
@Override
	public void start(Stage primaryStage) {
		try {
			FXMLLoader fxmlLoader = new
FXMLLoader(getClass().getResource("MOTOController.fxml"));
			Parent root = fxmlLoader.load();
			MOTOController Controller = fxmlLoader.getController();
			Logique logic = new Logique(Controller);
			Controller.setLogic(logic);
			Logique.Inii(logic);
			//AnchorPane root =
(AnchorPane)FXMLLoader.load(getClass().getResource("MOTOController.fxml"));
			Scene scene = new Scene(root);

	//scene.getStylesheets().add(getClass().getResource("application.css").toExternalForm());
			primaryStage.setScene(scene);
			primaryStage.show();
		} catch(Exception e) {
			e.printStackTrace();
		}
	}

	public static void main(String[] args) {
		launch(args);
	}
```

Méthode clé :

```
public void start(Stage primaryStage) {
```

Logique.java:

Définition de classe :

```
public class Logique
{
```

Rôle : La classe logique (CPU) a le role de mettre en oeuvre le cycle classique ***fetch-decode-execute,*** et il contient aussi les instructions et de modifier le contenu des registres, flags, accumulateurs, les mémoires via le controller.

Le cycle classique:

Fetch : Permet de chercher et vérifier la syntaxe du programme.

```
public String fetch(String programme)
      {
            String[] lignes = programme.split("\\R");
            if(lignes.length == 0)
            {
                  return "Pas d'instruction.";
            }
            else
            {
                  int numLigne = 1;

                for (String ligne : lignes)
                {
                    if (ligne.isBlank())
                    {
                        numLigne++;
                        continue;
                    }

                    if (ligne.equals("END"))
                    {
                        return null;
                    }
                    String[] mot = ligne.split("\\s+");
                    String inst = mot[0].toUpperCase();
                    String mod  = (mot.length > 1) ? mot[1] : "";

                    String mode = decode(mod);
```

```
                    if ("FALSE".equals(mode)) {
                        return "Mode invalide à la ligne " + numLigne + ".";
                    }

                    String key = inst + mode;
                    if (!instructions.containsKey(inst) || !opcode.containsKey(key)) {
                        return "Instruction invalide à la ligne " + numLigne + " : " + inst +
".";
                    }

                    numLigne++;
                }

                return "END manquant à la fin du programme.";
            }
        }
```

Decode :Permet de décoder l'opérande.

```
public String decode(String m)
      {
            if(m == null || m.isEmpty())
                  return "INHERENT";
            if(m.length() == 4 && m.startsWith("#$") && (((m.charAt(2) >= '0' && m.charAt(2) <=
'9') || (m.charAt(2) >= 'A' && m.charAt(2) <= 'F')) && ((m.charAt(3) >= '0' && m.charAt(3) <= '9')
|| (m.charAt(3) >= 'A' && m.charAt(3) <= 'F'))))
                  return "IMMEDIAT";
            else if(m.length() == 6 && m.startsWith("#$") && (((m.charAt(2) >= '0' && m.charAt(2)
<= '9') || (m.charAt(2) >= 'A' && m.charAt(2) <= 'F')) && ((m.charAt(3) >= '0' && m.charAt(3) <=
'9') || (m.charAt(3) >= 'A' && m.charAt(3) <= 'F')) && ((m.charAt(4) >= '0' && m.charAt(4) <= '9')
|| (m.charAt(4) >= 'A' && m.charAt(4) <= 'F')) && ((m.charAt(5) >= '0' && m.charAt(5) <= '9') ||
(m.charAt(5) >= 'A' && m.charAt(5) <= 'F'))))
                  return "IMMEDIAT3+";
            else if((m.length() == 5 && m.startsWith("$") && (((m.charAt(1) >= '0' && m.charAt(1)
<= '9') || (m.charAt(1) >= 'A' && m.charAt(1) <= 'F')) && ((m.charAt(2) >= '0' && m.charAt(2) <=
'9') || (m.charAt(2) >= 'A' && m.charAt(2) <= 'F')) && ((m.charAt(3) >= '0' && m.charAt(3) <= '9')
|| (m.charAt(3) >= 'A' && m.charAt(3) <= 'F')) && ((m.charAt(4) >= '0' && m.charAt(4) <= '9') ||
(m.charAt(4) >= 'A' && m.charAt(4) <= 'F'))))
                        || (m.length() == 6 && m.startsWith(">$") && (((m.charAt(2) >= '0' &&
m.charAt(2) <= '9') || (m.charAt(2) >= 'A' && m.charAt(2) <= 'F')) && ((m.charAt(3) >= '0' &&
m.charAt(3) <= '9') || (m.charAt(3) >= 'A' && m.charAt(3) <= 'F')) && ((m.charAt(4) >= '0' &&
m.charAt(4) <= '9') || (m.charAt(4) >= 'A' && m.charAt(4) <= 'F')) && ((m.charAt(5) >= '0' &&
m.charAt(5) <= '9') || (m.charAt(5) >= 'A' && m.charAt(5) <= 'F')))))
                  return "ETENDU";
            else if((m.length() == 7 && m.startsWith("[$") && (((m.charAt(2) >= '0' &&
m.charAt(2) <= '9') || (m.charAt(2) >= 'A' && m.charAt(2) <= 'F')) && ((m.charAt(3) >= '0' &&
m.charAt(3) <= '9') || (m.charAt(3) >= 'A' && m.charAt(3) <= 'F')) && ((m.charAt(4) >= '0' &&
m.charAt(4) <= '9') || (m.charAt(4) >= 'A' && m.charAt(4) <= 'F')) && ((m.charAt(5) >= '0' &&
m.charAt(5) <= '9') || (m.charAt(5) >= 'A' && m.charAt(5) <= 'F'))) && m.endsWith("]"))
                        || (m.length() == 8 && m.startsWith("[>$") && (((m.charAt(3) >= '0' &&
m.charAt(3) <= '9') || (m.charAt(3) >= 'A' && m.charAt(3) <= 'F')) && ((m.charAt(4) >= '0' &&
m.charAt(4) <= '9') || (m.charAt(4) >= 'A' && m.charAt(4) <= 'F')) && ((m.charAt(5) >= '0' &&
m.charAt(5) <= '9') || (m.charAt(5) >= 'A' && m.charAt(5) <= 'F')) && ((m.charAt(6) >= '0' &&
m.charAt(6) <= '9') || (m.charAt(6) >= 'A' && m.charAt(6) <= 'F'))) && m.endsWith("]")))
                  return "ETENDUINDIRECT";
```

```
            else if(m.length() == 4 && m.startsWith("<$") && (((m.charAt(2) >= '0' && m.charAt(2)
<= '9') || (m.charAt(2) >= 'A' && m.charAt(2) <= 'F')) && ((m.charAt(3) >= '0' && m.charAt(3) <=
'9') || (m.charAt(3) >= 'A' && m.charAt(3) <= 'F'))))
                  return "DIRECT";
            else if((m.length() == 2 && m.startsWith(",") && (m.charAt(1) == 'X' || m.charAt(1)
== 'Y' || m.charAt(1) == 'U' || m.charAt(1) == 'S'))
                        || (m.length() == 3 && m.startsWith(",") && m.charAt(1) == 'P' &&
m.charAt(2) == 'C'))
                  return "INDEXEDEPNULL";
            else if(m.length() == 3 && m.startsWith(",") && m.charAt(1) == '-' && (m.charAt(2) ==
'X' || m.charAt(2) == 'Y' || m.charAt(2) == 'U' || m.charAt(2) == 'S'))
                  return "INDEXEAUTODEC1";
            else if(m.length() == 4 && m.startsWith(",") && m.charAt(1) == '-' && m.charAt(2) ==
'-' && (m.charAt(3) == 'X' || m.charAt(3) == 'Y' || m.charAt(3) == 'U' || m.charAt(3) == 'S'))
                  return "INDEXEAUTODEC2";
            else if(m.length() == 3 && m.startsWith(",") && (m.charAt(1) == 'X' || m.charAt(1) ==
'Y' || m.charAt(1) == 'U' || m.charAt(1) == 'S') && m.charAt(2) == '+')
                  return "INDEXEAUTOINC1";
            else if(m.length() == 4 && m.startsWith(",") && (m.charAt(1) == 'X' || m.charAt(1) ==
'Y' || m.charAt(1) == 'U' || m.charAt(1) == 'S') && m.charAt(2) == '+' && m.charAt(3) == '+')
                  return "INDEXEAUTOINC2";
            else if(m.length() == 3 && m.startsWith("A,") && (m.charAt(2) == 'X' || m.charAt(2)
== 'Y' || m.charAt(2) == 'U' || m.charAt(2) == 'S'))
                  return "INDEXEDEPA";
            else if(m.length() == 3 && m.startsWith("B,") && (m.charAt(2) == 'X' || m.charAt(2)
== 'Y' || m.charAt(2) == 'U' || m.charAt(2) == 'S'))
                  return "INDEXEDEPB";
            else if(m.length() == 3 && m.startsWith("D,") && (m.charAt(2) == 'X' || m.charAt(2)
== 'Y' || m.charAt(2) == 'U' || m.charAt(2) == 'S'))
                  return "INDEXEDEPD";
            else if(m.length() == 5 && m.startsWith("$") && (((m.charAt(1) >= '0' && m.charAt(1)
<= '9') || (m.charAt(1) >= 'A' && m.charAt(1) <= 'F')) && ((m.charAt(2) >= '0' && m.charAt(2) <=
'9') || (m.charAt(2) >= 'A' && m.charAt(2) <= 'F'))) && m.charAt(3) == ',' && (m.charAt(4) == 'X'
|| m.charAt(4) == 'Y' || m.charAt(4) == 'U' || m.charAt(4) == 'S'))
                  return "INDEXEDEPCONS1OCT";
            else if(m.length() == 7 && m.startsWith("$") && (((m.charAt(1) >= '0' && m.charAt(1)
<= '9') || (m.charAt(1) >= 'A' && m.charAt(1) <= 'F')) && ((m.charAt(2) >= '0' && m.charAt(2) <=
'9') || (m.charAt(2) >= 'A' && m.charAt(2) <= 'F')) && ((m.charAt(3) >= '0' && m.charAt(3) <= '9')
|| (m.charAt(3) >= 'A' && m.charAt(3) <= 'F')) && ((m.charAt(4) >= '0' && m.charAt(4) <= '9') ||
(m.charAt(4) >= 'A' && m.charAt(4) <= 'F'))) && m.charAt(5) == ',' && (m.charAt(6) == 'X' ||
m.charAt(6) == 'Y' || m.charAt(6) == 'U' || m.charAt(6) == 'S'))
                  return "INDEXEDEPCONS2OCT";
            else
                  return "FALSE";
      }
```

Execute :Permet d'exécuter une instruction .

```
public static void execute(String inst, String mod)
      {
            Consumer<String> ins = instructions.get(inst);
          if (ins != null) ins.accept(mod);
      }
```

Autres méthodes importantes :

Ecrire mémoire :

Cette méthode permet d'écrire dans la mémoire en utilisant la logique , écrire/stocker premièrement dans la classe mémoire et après appeler le controller pour modifier la ligne correspondant à l'adresse donné dans la tableview qui simule la mémoire.

```
public void ecrireMemoire(int address, int val) {
          Memoire.ecrire(address, val);
          Controller.atualiserCaseMemoire(address);
      }
```

ProgMémoire :

Il permet après l'assemblage de stocker les données(les OpCodes, postoctects et opérandes).

Une remarque importante : pour obtenir le post-octets de chaque mode d'adressage indexé (par exp : déplacement nulle, auto inc/dec, déplacement avec les accumulateurs…), on remplace où il y a R par les bits qu'il correspond à cette base(x,y,u,s).Avec pour x les bits sont (00), pour y (01), pour u (10), et pour s (11), et on obtient une post-octet en hexadécimal.

```
public void progMemoire(String prog)
      {
```

NbOct:

Cette méthode permet de renvoyer nombre d'octet sur lequelle une telle instruction est codée en utilisant la logique de prendre toujours le dernier caractère (parce que on a utilisé la logique pour mettre le nombre d'octets sur le dernier caractère de notre chaine de caractères dans le HashMap opcode).

```
public int nbOct(String ins, String op)
      {
            String cle = ins + op;
            String res = opcode.get(cle);
            if(res == null) throw new IllegalArgumentException();
            return Integer.parseInt(res.substring(res.length()-1));

      }
```

Inii:

Il permet d'initialiser les instructions dans les hashmap (le tableau qui contient les instructions et les modes d'adressages comme clé et qui nous donne les OpCodes, post-octets et les Nombre d'octets pour chaque mode d'adressage).

Une remarque importante : les R seront remplacés dans ProgMémoire pour obtenir le post-octets pour une base précise.

```
static void Inii(Logique logic)
      {
```

OpCode:

- ✓ Il permet de renvoyer l'opcode de chaque instruction avec une respéctive de mode d'adressage.

```
public String opcode(String ins, String mode)
      {
```

Atualflags :

- ✓ Il permet d'actualiser le champs des flags via MOTOController.java.

```
public void atualflags(String f)
      {
```

MotoController.java :

```
public class MOTOController
{
```

Rôle : Il permet de controller l'interface graphique, par exemple en cliquant sur un boutton et de la communication entre le simulateur et l'utilisateur, c'est celui qui a le droit de modifier les componants de l'interface graphique(.fxml).

Controlle de l'interface graphique :

```
public void setLogic(Logique logic)
     {
```

- ✓ Il permet de connecter la logique à un instance de controller.

```
public void initialize()
     {
```

- ✓ Il permet d'initialiser les tableview(tabRam,tabRom et tabProg) et les contenus des accumulateurs, registres et les flags.

```
private void compile()
     {
```

- ✓ Il permet de vérifier la syntaxe de programme.

```
private void run()
     {
```

- ✓ Il permet d'exécuter le programme complet une fois la vérification de syntaxe est réussie.

```
private void stepbystep()
     {
```

- ✓ Il permet d'exécuter le programme pas à pas une fois la vérification de syntaxe est réussie.

```
private void reset()
     {
```

- ✓ Il permet de réinitialiser les registres et les accumulateurs, les flags, les tablaux…

```
private void insertTabProg()
    {
```

✓ Il permet d'insérer dans la tableview tabProg le programme déjà compiler.

```
public void atualiserRAMTab() {
```

✓ Il permet d'actualiser la tableview TabRAM.

```
public void atualiserROMTab() {
```

✓ Il permet d'actualiser la tableview TabROM.

```
public void atualiserCaseMemoire(int address) {
```

✓ Il permet d'actualiser une case mémoire correspondante à l'adresse passé en argument.

```
public void initFlags()
    {
```

✓ Il permet d'initialiser les flags.

```
public void init()
    {
```

✓ Il permet d'initialiser les registres, accumulateurs et les flags.

Mémoire :

```
public class Memoire
{
```

Role : Simuler la mémoire réelle et de stocker les données .

Simulation de la mémoire :

```
package simul.moto;

public class Memoire
{

    public static final int RAM_SIZE = 0xFC00;
    public static final int ROM_START = 0xFC00;
    public static final int MEM_SIZE = 0x10000;


    private static final byte[] mem = new byte[MEM_SIZE];


    public static void ecrire(int address, int value) {
        mem[address] = (byte)(value & 0xFF);
    }


    public static int lire(int address) {
        return mem[address] & 0xFF;
    }


    public static void reset() {
        for (int i = 0; i < MEM_SIZE; i++) {
            mem[i] = 0;
        }
    }
}
```

Fonctionnement :

La méthode ecrire permet d'écire dans une instance mémoire, et la méthode lire permet de lire dans une instance mémoire et la méthode reset permet de mettre tout le contenu à la valeur 0(initialiser).

OpFlags :

```
public class OpFlags
{
```

Role: Il contient des méthodes qui permettent de faire des opérations arithmétiques, vérification s'il existe débordement ,retenue,négatif,zéro.

Exécution des opérations arithmétiques et vérification des flags :

```
public static String somme(String a, String b)
	{
		BigInteger hex1 = new BigInteger(a, 16);
		BigInteger hex2 = new BigInteger(b, 16);
		BigInteger sum = hex1.add(hex2);
		String res = sum.toString(16);
		return res;
	}
	public static String substraction(String a, String b)
	{
		BigInteger hex1 = new BigInteger(a, 16);
		BigInteger hex2 = new BigInteger(b, 16);
		BigInteger sub = hex1.subtract(hex2);
		String res = sub.toString(16);
		return res;
	}
	public static String multiplication(String a, String b)
	{
		BigInteger hex1 = new BigInteger(a, 16);
		BigInteger hex2 = new BigInteger(b, 16);
		BigInteger mul = hex1.multiply(hex2);
		String res = mul.toString(16);
		return res;
	}
```

✓ Chaque méthode fait une opération arithmétique qu'il le correspond.

```
public static String flags(int a, int b, int res)
	{
		int b7a = (a >> 7) & 1;
		int b7b = (b >> 7) & 1;
		int b7res = (res >> 7) & 1;

		boolean vsum = (b7a == b7b) && (b7a != b7res);
		boolean csum = (res & 0x100) != 0;
		boolean vsub = (b7a != b7b) && (b7a != b7res);
		boolean csub = (a & 0xFF) < (b & 0xFF);
		boolean cmul = (b7b & 1) != 0;
		boolean z = (res & 0xFF) == 0;
		boolean n = (b7res & 1) != 0
				;
		String flags = "";
		if(vsum) return flags += "vsum";
		else if(csum) return flags += "csum";
		else if(vsub) return flags += "vsub";
		else if(csub) return flags += "csub";
		else if(cmul) return flags += "cmul";
```

```
            else if(n) return flags += "n";
            else if(z) return flags += "z";

            else return "PASDE";
        }
```

✓ Il permet de vérifier : retenues, débordements, négatif,zéro.

```
int b7a = (a >> 7) & 1;
            int b7b = (b >> 7) & 1;
            int b7res = (res >> 7) & 1;
```

✓ Cette partie permet de déplacer tous les bits et mettre le bit 7 à la position 0, donc en considérant que bit 7.

```
boolean vsum = (b7a == b7b) && (b7a != b7res);
            boolean csum = (res & 0x100) != 0;
            boolean vsub = (b7a != b7b) && (b7a != b7res);
            boolean csub = (a & 0xFF) < (b & 0xFF);
            boolean cmul = (b7b & 1) != 0;
            boolean z = (res & 0xFF) == 0;
            boolean n = (b7res & 1) != 0
```

✓ Ces variables booléens définissent si il y a ou pas des débordements, retenues,négatif,zero.

- **Csum :** il définit si il y a retenu dans une opération du somme.
- **Vsum :** il définit si il y a débordement dans une opération du somme.
- **Vsub :** il définit si il y a débordement dans une opération du soustraction.
- **Csub :** il définit si il y a retenu dans une opération du soustraction.
- **Cmul :** il définit si il ya retenu dans une opération de multiplication.
- **Z :**il définit si le résulat est zéro.
- **N :** il définit si le résulat est négatif.