MOF6809

Simulateur du Microprocesseur Motorola 6809

Application graphique de simulation permettant l'écriture, la compilation et l'exécution de programmes en assembleur 6809 avec visualisation en temps réel des registres et de la mémoire.

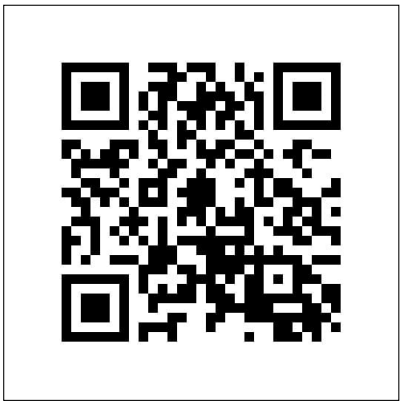
Réalisé par :

- Mohammed Abdelkhalek
- Mohamed Aoulichak
- Oussama Ait Mendil
- Fouad Ayyad

Projet Universitaire — Année Académique 2024-2025

Accès Rapide aux Ressources

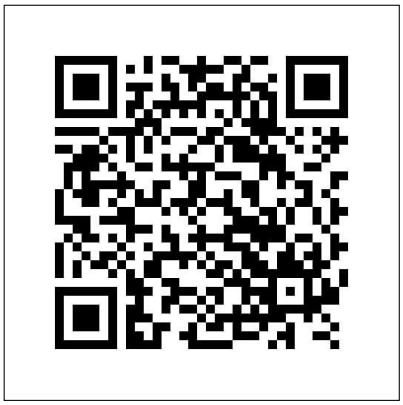
Scannez les QR Codes pour accéder directement aux ressources du projet



Repository GitHub

Code source complet du projet

`github.com/OsKing00/MOF6809`



Présentation Interactive

Slides de présentation du projet

`Présentation en ligne`

Instructions : Scannez le QR Code de gauche pour accéder au code source sur GitHub, ou celui de droite pour visualiser la présentation interactive du projet.

Table des matières

Chapitre 1

Introduction

1.1 Contexte du Projet

Le microprocesseur **Motorola 6809**, lancé en 1978, représente l'une des architectures 8 bits les plus avancées de son époque. Utilisé dans de nombreux systèmes emblématiques tels que le *TRS-80 Color Computer*, le *Dragon 32/64*, et diverses bornes d'arcade, ce processeur se distingue par son jeu d'instructions riche et ses modes d'adressage sophistiqués.

Note Importante

Le 6809 est considéré comme le processeur 8 bits le plus puissant de sa génération, offrant des caractéristiques habituellement réservées aux processeurs 16 bits : deux accumulateurs, deux registres d'index, deux pointeurs de pile, et un registre de page directe.

Dans un contexte pédagogique, la compréhension du fonctionnement interne d'un microprocesseur est fondamentale pour tout étudiant en informatique ou en électronique. Cependant, manipuler du matériel réel présente plusieurs contraintes : coût, disponibilité, risque de dommages, et difficulté de visualisation des états internes.

1.2 Problématique

La problématique centrale de ce projet peut être formulée ainsi :

Problématique

Comment permettre à un utilisateur d'apprendre et d'expérimenter le fonctionnement du microprocesseur Motorola 6809 sans disposer du matériel physique, tout en offrant une visualisation claire et pédagogique de chaque étape d'exécution ?

Cette question soulève plusieurs défis techniques :

- **Fidélité de l'émulation** : Reproduire avec exactitude le comportement du processeur original, notamment la gestion des registres, des drapeaux (flags) et des modes d'adressage.
- **Interface utilisateur intuitive** : Concevoir une interface graphique permettant de visualiser simultanément le code source, la mémoire (RAM/ROM), et l'état des registres.
- **Compilation et validation** : Implémenter un compilateur capable de traduire le code assembleur en code machine, tout en détectant et signalant les erreurs de syntaxe.
- **Modes d'exécution** : Offrir une exécution pas-à-pas pour le débogage et une exécution continue pour les tests rapides.

1.3 Objectifs du Projet

Le projet **MOF6809** a été développé avec les objectifs suivants :

1.3.1 Objectifs Pédagogiques

1. Permettre aux étudiants de comprendre le cycle d'exécution d'une instruction (fetch-decode-execute).
2. Visualiser en temps réel l'impact de chaque instruction sur les registres et la mémoire.
3. Faciliter l'apprentissage du langage assembleur 6809 par la pratique.
4. Démystifier le fonctionnement de l'Unité Arithmétique et Logique (UAL).

1.3.2 Objectifs Techniques

1. Implémenter un simulateur fonctionnel du 6809 en Java avec interface graphique Swing.
2. Supporter les principaux modes d'adressage : **Immédiat**, **Direct**, **Étendu**, et **Inhérent**.

3. Gérer un jeu d'instructions représentatif incluant : chargement (LD), stockage (ST), arithmétique (ADD, SUB), logique (AND, OR), comparaison (CMP), et transfert (TFR, EXG).
4. Afficher les drapeaux du registre de condition (CCR) : Carry, Zero, Negative, Overflow, Half-Carry.

1.4 Structure du Rapport

Ce rapport technique est organisé comme suit :

Chapitre	Contenu
2	**Analyse Fonctionnelle** : Description détaillée de chaque fonctionnalité du simulateur, avec les entrées, traitements et sorties.
3	**Analyse Technique** : Architecture du code, extraits commentés, explication des algorithmes critiques (UAL, gestion mémoire, décodage).
4	**Conclusion** : Bilan du projet, limitations identifiées et perspectives d'amélioration.

TABLE 1.1 – Organisation du rapport

1.5 Technologies Utilisées

Catégorie	Technologie	Justification
Langage	Java 8+	Portabilité, robustesse, gestion mémoire automatique
Interface graphique	Java Swing	Bibliothèque native, pas de dépendances externes
Architecture	Procédurale / Façade	Simplicité de mise en œuvre, code maintenable
IDE recommandé	Eclipse / IntelliJ	Support complet Java, débogage intégré

TABLE 1.2 – Stack technologique du projet

Chapitre 2

Analyse Fonctionnelle

Ce chapitre présente une description exhaustive de chaque fonctionnalité du simulateur MOF6809. Pour chaque fonction, nous détaillons les **entrées** attendues, les **traitements** effectués, et les **sorties** produites.

2.1 Vue d'Ensemble de l'Interface

L'interface graphique du simulateur est divisée en plusieurs zones fonctionnelles distinctes, chacune ayant un rôle précis dans le processus de simulation.

Éditeur de Code	$\rightarrow$	**Compilateur**
		$\downarrow$
Registres	$\leftrightarrow$ **UAL** $\rightarrow$	**Mémoire ROM/RAM**

TABLE 2.1 – Architecture fonctionnelle du simulateur MOF6809

2.2 Fonctionnalité 1 : Éditeur de Code Assembleur

Éditeur de Code

Zone de texte intégrée permettant la saisie de programmes en langage assembleur 6809. L'éditeur constitue le point d'entrée principal pour l'utilisateur.

2.2.1 Entrées

Type	Format	Exemple
Mnémonique	Instruction assembleur	`LDA`, `ADDA`, `STB`
Mode d'adressage	Préfixe spécial	`#` (immédiat), `<` (direct), `>` (étendu)
Opérande	Valeur hexadécimale	`$99`, `$1234`, `$FF`
Terminaison	Mot-clé obligatoire	`END` (fin de programme)

TABLE 2.2 – Types d'entrées acceptées par l'éditeur

2.2.2 Traitements

L'éditeur effectue les traitements suivants lors de la saisie :

1. **Capture du texte** : Récupération du contenu via `TextCodeArea.getText()`
2. **Découpage en lignes** : Séparation par retour à la ligne (`split("textbackslash r?textbackslash n")`)
3. **Normalisation** : Conversion en majuscules pour uniformiser le traitement
4. **Stockage** : Insertion dans une `ArrayList<String>` pour traitement ultérieur

2.2.3 Sorties

→ Liste ordonnée des lignes de code prêtes pour la compilation

→ Affichage persistant dans la zone de texte pour modification

2.3 Fonctionnalité 2 : Compilateur (Analyse et Traduction)

Compilateur

Module responsable de l'analyse lexicale, syntaxique et de la génération du code machine à partir du code assembleur source.

2.3.1 Entrées

- **T1** : ArrayList contenant les lignes de code assembleur
- **InstructionName**[] : Tableau des mnémoniques extraits (partie P1)
- **AdressageMode**[] : Tableau des caractères de mode (`#`, `<`, `>`, `\0`)
- **Valeur**[] : Tableau des opérandes hexadécimales

2.3.2 Traitements

Le processus de compilation se déroule en plusieurs phases :

Algorithme : Processus de Compilation

1. **Phase 1 - Extraction** (Classe `Instructions`)
   - `GetP1()` : Extrait le mnémonique (avant l'espace)
   - `GetP2()` : Extrait l'opérande complet (après l'espace)
   - `GetModeAdressage()` : Identifie le préfixe de mode
   - `GetValue()` : Isole la valeur hexadécimale
2. **Phase 2 - Validation** (Classe `Erreurs`)
   - Vérification de la syntaxe de chaque instruction
   - Validation du format hexadécimal (1 ou 2 octets)
   - Contrôle de la présence du `END`
   - Comptage des erreurs détectées
3. **Phase 3 - Génération** (Classe `Decodage`)
   - Conversion mnémonique → opcode hexadécimal
   - Calcul de la taille de chaque instruction
   - Écriture séquentielle en ROM

2.3.3 Sorties

Sortie	Description
`ValROM[]`	Tableau d'entiers contenant le code machine (opcodes + opérandes)
`ERRORArea`	Zone de texte affichant "Code Correct" ou les erreurs ligne par ligne
`PC_val[]`	Tableau des adresses du Program Counter pour chaque instruction

TABLE 2.3 – Sorties produites par le compilateur

2.4 Fonctionnalité 3 : Gestion de la Mémoire

2.4.1 Mémoire ROM (Read-Only Memory)

ROM - Mémoire Programme

Zone mémoire en lecture seule stockant le programme compilé. Adressée de `$FC00` à `$FFFF` (1024 octets).

Entrées

- Code machine généré par le compilateur
- Adresse de base : `$FC00`

Traitements

1. Initialisation à `$FF` (valeur par défaut)
2. Écriture séquentielle des opcodes et opérandes
3. Ajout du marqueur de fin `$3F` (instruction SWI)

Sorties

→ Tableau visuel avec colonnes Adresse/Valeur

→ Mise à jour dynamique du `JTable` ROM

2.4.2 Mémoire RAM (Random Access Memory)

RAM - Mémoire Données

Zone mémoire en lecture/écriture pour le stockage des données. Adressée de `$0000` à `$04FF` (1280 octets).

Entrées

- Instructions de stockage (STA, STB, STD, etc.)
- Adresse cible et valeur à écrire

Traitements

1. Initialisation à `$00`
2. Écriture lors de l'exécution des instructions ST*
3. Lecture lors des modes d'adressage direct/étendu

Sorties

→ Affichage dans le `JTable` RAM

→ Valeurs accessibles pour les calculs UAL

2.5 Fonctionnalité 4 : Tableau de Bord des Registres

Registres du 6809

Ensemble des registres internes du processeur, affichés en temps réel et mis à jour après chaque instruction.

2.5.1 Registres Implémentés

Registre	Taille	Init	Fonction
`A`	8 bits	`00`	Accumulateur principal
`B`	8 bits	`00`	Accumulateur secondaire
`D`	16 bits	`0000`	Double accumulateur (A :B concaténés)
`X`	16 bits	`0000`	Registre d'index X
`Y`	16 bits	`0000`	Registre d'index Y
`S`	16 bits	`0000`	Pointeur de pile système
`U`	16 bits	`0000`	Pointeur de pile utilisateur
`PC`	16 bits	`FC00`	Compteur ordinal (Program Counter)
`DP`	8 bits	`00`	Registre de page directe

TABLE 2.4 – Registres du microprocesseur 6809 simulés

2.5.2 Entrées

- Résultats des calculs de l'UAL
- Valeurs chargées depuis la mémoire
- Transferts entre registres (TFR, EXG)

2.5.3 Traitements

1. Stockage dans des `ArrayList<Integer>` pour historique
2. Formatage hexadécimal (2 digits pour 8 bits, 4 pour 16 bits)
3. Masquage avec `& 0xFF` ou `& 0xFFFF` pour éviter les débordements

2.5.4 Sorties

→ Affichage dans les `JTextField` correspondants

→ Format : majuscules, zéros préfixés (`0A`, `00FF`)

2.6 Fonctionnalité 5 : Registre de Condition (CCR)

CCR - Condition Code Register

Registre 8 bits contenant les drapeaux (flags) résultant des opérations arithmétiques et logiques. Chaque bit représente une condition spécifique.

2.6.1 Structure du CCR

Bit	Index	Nom	Condition de mise à 1
E	[0]	Entire	Sauvegarde complète lors d'interruption
F	[1]	FIRQ Mask	Masque d'interruption rapide
H	[2]	Half Carry	Retenue du bit 3 vers le bit 4
I	[3]	IRQ Mask	Masque d'interruption standard
N	[4]	Negative	Résultat négatif (bit 7 = 1)
Z	[5]	Zero	Résultat égal à zéro
V	[6]	Overflow	Dépassement de capacité signé
C	[7]	Carry	Retenue sortante (bit 7 pour 8 bits)

TABLE 2.5 – Drapeaux du registre de condition CCR

2.6.2 Entrées

- Opérandes d'entrée (avant opération)
- Résultat de l'opération
- Type d'opération (arithmétique, logique, chargement)

2.6.3 Traitements (Méthodes de l'UAL)

- `CheckCarry8()` / `CheckCarry16()` : Détection de retenue
- `CheckHalfCarry8()` : Détection de demi-retenue
- `CheckZero()` : Test si résultat = 0
- `CheckSigne()` : Test du bit de signe
- `CheckOverFlow()` : Détection de débordement signé

2.6.4 Sorties

→ 8 `JTextField` individuels (CC1 à CC8)

→ Valeur 0 ou 1 pour chaque flag

2.7 Fonctionnalité 6 : Modes d'Exécution

2.7.1 Mode "Execute All" (Exécution Complète)

Execute All

Exécute l'intégralité du programme compilé en une seule action, affichant l'état final des registres et de la mémoire.

Entrées

- Programme compilé en ROM
- État initial des registres

Traitements

1. Récupération du dernier index des ArrayList de valeurs
2. Affichage des valeurs finales de chaque registre
3. Mise à jour de la RAM avec l'état final

Sorties

→ État final de tous les registres

→ Contenu final de la RAM

→ Dernière instruction exécutée dans `INSTRTextField`

2.7.2 Mode "Step By Step" (Pas à Pas)

Step By Step

Exécute une instruction à chaque clic, permettant d'observer l'évolution progressive de l'état du processeur.

Entrées

- Variable `step` : index de l'instruction courante
- ArrayList des valeurs calculées pour chaque registre

Traitements

1. Incrémentation de `step` et des index par registre
2. Vérification des limites (ne pas dépasser la taille des ArrayList)
3. Affichage de la valeur à l'index courant pour chaque registre
4. Mise à jour de `PC` et de l'instruction courante

Sorties

→ État des registres après l'instruction N
→ Instruction courante affichée
→ Valeur du PC pointant vers l'instruction suivante

2.8 Fonctionnalité 7 : Gestion des Erreurs

Module d'Erreurs

Système de validation syntaxique qui vérifie la conformité de chaque ligne de code avant compilation.

2.8.1 Types d'Erreurs Détectées

Type d'Erreur	Exemple Invalide	Message
Mnémonique inconnu	`LOAD #$99`	Error At line X
Mode d'adressage invalide	`LDA @$99`	Error At line X
Valeur non hexadécimale	`LDA #$GG`	Error At line X
Taille incorrecte	`LDA #$1234`	Error At line X
END manquant	(absence de END)	Missing END at last line

TABLE 2.6 – Types d'erreurs détectées par le compilateur

2.8.2 Entrées

- Ligne de code complète
- Composants extraits (P1, P21, P22)

2.8.3 Traitements

La méthode `CheckSyntaxErrors()` effectue une série de validations :

1. Vérification du mnémonique dans les listes autorisées
2. Validation du caractère de mode d'adressage
3. Contrôle du format hexadécimal de l'opérande
4. Vérification de la cohérence taille instruction/opérande

2.8.4 Sorties

→ `"Code Correct"` si aucune erreur

→ Liste des erreurs avec numéro de ligne

→ Compteur d'erreurs (`ErrorNumbers`)

2.9 Fonctionnalité 8 : Actions Utilitaires

2.9.1 Bouton Reset

Fonction : Réinitialise tous les registres à leur valeur par défaut

Entrée : Clic utilisateur

Traitement : Remise à zéro des TextField et des ArrayList

Sortie : Interface dans l'état initial, ROM/RAM réinitialisées

2.9.2 Bouton Clear All

Fonction : Efface le contenu de l'éditeur de code

Entrée : Clic utilisateur

Traitement : `TextCodeArea.setText(null)`

Sortie : Zone de code vide, prête pour nouvelle saisie

2.10 Récapitulatif des Modes d'Adressage

Mode	Symbole	Syntaxe	Description
Immédiat	`#`	`LDA #$99`	Valeur directe dans l'instruction
Direct	`<`	`LDA <$20`	Adresse sur 1 octet (page 0)
Étendu	`>` ou rien	`LDA $1234`	Adresse complète sur 2 octets
Inhérent	-	`CLRA`, `NOP`	Pas d'opérande, action implicite

TABLE 2.7 – Modes d'adressage supportés par le simulateur

Chapitre 3

Analyse Technique et Code Source

Ce chapitre constitue le cœur technique du rapport. Nous y présentons l'architecture du code, les extraits les plus significatifs, et une analyse algorithmique détaillée de chaque composant.

3.1 Arborescence du Projet

Structure des Fichiers

Le projet suit une organisation en packages Java standard, séparant clairement l'interface graphique de la logique métier.

```
MOF6809/
|-- src/
|   |-- module-info.java
|   |-- Main/
|   |   +-- MainClass.java           # Point d'entree (15 lignes)
|   |-- graphicUserInterface/
|   |   +-- GUIClass.java            # Interface Swing (1121 lignes)
|   +-- programMethodes/
|       |-- Decodage.java            # Conversion opcode (281 lignes)
|       |-- Erreurs.java             # Validation syntaxe (376
    lignes)
|       |-- Instructions.java        # Parsing assembleur (130
    lignes)
|       |-- Memoire.java             # Gestion RAM/ROM (100 lignes)
|       |-- Registres.java           # Calcul PC (25 lignes)
|       +-- UAL.java                 # Unite Arithmetique (2907
    lignes)
```

```
|-- resources/
|   +-- images/
|       |-- Icon.png
|       +-- Logo.png
+-- rapport/
    +-- rapport.tex
```

Listing 3.1 – Arborescence du projet MOF6809

Note Importante

La classe `UAL` représente à elle seule près de 50% du code source total, témoignant de la complexité de l'émulation des opérations du processeur.

3.2 Point d'Entrée : MainClass.java

```java
package Main;

import javax.swing.SwingUtilities;
import graphicUserInterface.GUIClass;

public class MainClass {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(GUIClass::new);
    }
}
```

Listing 3.2 – Classe principale - Point d'entrée de l'application

Analyse technique :

- **Ligne 7** : Utilisation de `SwingUtilities.invokeLater()` pour garantir que l'interface graphique est créée dans l'**Event Dispatch Thread** (EDT), conformément aux bonnes pratiques Swing.
- **Ligne 7** : La syntaxe `GUIClass::new` est une **référence de méthode** Java 8, équivalente à `() -> new GUIClass()`. Elle invoque le constructeur par défaut de `GUIClass`.
- **Pattern Façade** : La classe `MainClass` ne contient aucune logique métier. Elle délègue entièrement à `GUIClass` qui orchestre tous les composants.

3.3 Interface Graphique : GUIClass.java

3.3.1 Initialisation et Déclaration des Composants

```
1  public class GUIClass {
2      public GUIClass() {
3          initComponents();
4      }
5
6      // Instances des classes metier
7      Memoire MEM = new Memoire();
8      Erreurs ERR = new Erreurs();
9      Instructions INST = new Instructions();
10     UAL UAL = new UAL();
11     Registres REG = new Registres();
12
13     // Tableaux memoire
14     private int AdROM[] = MEM.AdressROM();
15     private int AdRAM[] = MEM.AdressRAM();
16     private int ValROM[] = MEM.InitValeurROM();
17     private int ValRAM[] = MEM.InitValeurRAM();
18
19     // Variables d'etat pour l'execution pas-a-pas
20     private int step = 0;
21     private int stepA = 0, stepB = 0, stepS = 0;
22     private int stepX = 0, stepY = 0, stepDP = 0;
23
24     // Stockage des valeurs calculees
25     ArrayList<Integer> A_val = new ArrayList<>();
26     ArrayList<Integer> B_val = new ArrayList<>();
27     ArrayList<Integer> S_val = new ArrayList<>();
28     ArrayList<int[]> CCR = new ArrayList<>();
```

Listing 3.3 – Déclaration des instances métier dans GUIClass

Analyse technique :

- **Lignes 7-11** : Instanciation des 5 classes métier au niveau de la classe (et non dans une méthode). Cela garantit leur disponibilité durant toute la vie de l'objet `GUIClass`.
- **Lignes 14-17** : Les tableaux mémoire sont initialisés dès la construction via les méthodes de `Memoire`. La ROM démarre à `$FC00`, la RAM à `$0000`.
- **Lignes 20-22** : Les variables `step*` servent de curseurs pour le mode pas-à-pas. Chaque registre possède son propre index pour gérer les cas où une instruction ne modifie pas tous les registres.
- **Lignes 25-28** : L'utilisation d'`ArrayList<Integer>` permet de stocker l'historique complet des valeurs de chaque registre, essentiel pour le mode Step By

Step.

3.3.2 Gestionnaire d'Événement : Bouton Compile

```
Compile.addActionListener(e -> {
    // Recuperation et decoupage du code source
    String CodeAssembly = TextCodeArea.getText();
    String[] lines = CodeAssembly.split("\\r?\\n", -1);
    T1.clear();
    T1.addAll(Arrays.asList(lines));
    T1.replaceAll(String::toUpperCase);
    int tai_T1 = T1.size();

    // Extraction des composants de chaque instruction
    InstructionName = INST.GetP1(T1);       // Mnemonique
    P2 = INST.GetP2(T1);                    // Operande complet
    AdressageMode = INST.GetModeAdressage(P2);  // Mode (#, <, >)
    Valeur = INST.GetValue(P2);             // Valeur hex

    // Calcul des tailles pour le PC
    int P1_NombreOctets[] = ERR.P1_NumberOctets(T1, InstructionName,
                                                AdressageMode,
                                                   taille);
    int P22_NombreOctets[] = ERR.P22_NumberOctets(Valeur, taille);

    // Verification des erreurs
    ERRORArea.setText(ERR.AfficheCompilation(T1, InstructionName,
                                             AdressageMode, Valeur));
    int Err = ERR.ErrorNumbers(T1, InstructionName, AdressageMode,
       Valeur);

    if (Err == 0) {
        // Generation du code machine en ROM
        ValROM = MEM.ValeurROM(T1, InstructionName, AdressageMode,
                               Valeur, P1_NombreOctets,
                                  P22_NombreOctets, taille);

        // Calcul des adresses PC
        PC_val = REG.PC(P1_NombreOctets, P22_NombreOctets, AdROM,
           taille);

        // Execution via l'UAL - Calcul de toutes les valeurs
        UAL.CalculMembers Calc = UAL.Calcule(T1, PC_val,
           InstructionName,
                                AdressageMode, ValROM, AdRAM,
                                P1_NombreOctets, P22_NombreOctets);

```

```
39          // Recuperation des resultats
40          A_val = Calc.A;
41          B_val = Calc.B;
42          CCR = Calc.CCR;
43          // ... autres registres
44      }
45  });
```

Listing 3.4 – ActionListener du bouton Compile - Cœur de la compilation

Analyse technique détaillée :

Algorithme : Flux de Compilation

1. **Acquisition** (lignes 3-8) : Le texte brut est récupéré, découpé en lignes, et normalisé en majuscules.
2. **Parsing** (lignes 11-14) : La classe `Instructions` décompose chaque ligne en ses constituants : mnémonique, mode, valeur.
3. **Validation** (lignes 21-24) : La classe `Erreurs` vérifie la syntaxe et retourne le nombre d'erreurs.
4. **Génération** (lignes 28-29) : Si aucune erreur, le code machine est généré et stocké dans `ValROM[]`.
5. **Exécution** (lignes 35-37) : L'UAL simule l'exécution complète et retourne un objet `CalculMembers` contenant tous les états.

Attention

La méthode `Calcule()` de l'UAL est appelée une seule fois lors de la compilation. Elle pré-calcule **tous** les états intermédiaires, qui sont ensuite utilisés par les modes Execute All et Step By Step.

3.4 Parsing des Instructions : Instructions.java

3.4.1 Extraction du Mnémonique (P1)

```
1  public String[] GetP1(ArrayList<String> T) {
2      int taille = T.size();
3      P1 = new String[taille];
4      int i = 0;
5
6      while (i < taille) {
7          String line = T.get(i);
```

```
8           if (line == null) {
9               P1[i] = "";
10              i++;
11              continue;
12          }
13          line = line.trim();
14          if (line.equalsIgnoreCase("END"))
15              break;
16
17          char[] SS = line.toCharArray();
18          P1[i] = "";
19
20          // Extraction caractere par caractere jusqu'a l'espace
21          for (int j = 0; j < SS.length && SS[j] != ' '; j++)
22              P1[i] += SS[j];
23
24          i++;
25      }
26      return P1;
27  }
```

Listing 3.5 – Méthode GetP1 - Extraction du mnémonique

Analyse technique :

- **Lignes 8-12** : Gestion robuste des lignes nulles ou vides pour éviter les `NullPointerException`.
- **Lignes 14-15** : Détection du mot-clé `END` qui marque la fin du programme. La boucle s'arrête immédiatement.
- **Lignes 21-22** : Algorithme simple mais efficace : parcours caractère par caractère jusqu'au premier espace. Pour `"LDA #$99"`, retourne `"LDA"`.

3.4.2 Extraction du Mode d'Adressage

```
1   public char[] GetModeAdressage(String P2[]) {
2       P21 = new char[P2.length];
3       int k = 0;
4
5       while (k < P2.length) {
6           P21[k] = '\0';  // Valeur par defaut (mode etendu/inherent)
7
8           if (P2[k] == null) {
9               k++;
10              continue;
11          }
12
```

```
13          String s = P2[k].trim();
14          if (s.length() == 0) {
15              k++;
16              continue;
17          }
18
19          char SS[] = s.toCharArray();
20
21          // Le premier caractere determine le mode si != '$'
22          if (SS[0] != '$')
23              P21[k] = SS[0];  // '#' pour immediat, '<' pour direct
24
25          k++;
26      }
27      return P21;
28  }
```

Listing 3.6 – Méthode GetModeAdressage - Identification du mode

Analyse technique :

- **Ligne 6** : Le caractère nul `'\0'` représente le mode étendu ou inhérent (pas de préfixe).
- **Lignes 22-23** : Logique de détection :
  - Si `SS[0] == '#'` ⇒ Mode **Immédiat**
  - Si `SS[0] == '<'` ⇒ Mode **Direct**
  - Si `SS[0] == '>'` ⇒ Mode **Étendu** (explicite)
  - Si `SS[0] == '$'` ⇒ Mode **Étendu** (implicite, valeur seule)

3.5 Décodage des Instructions : Decodage.java

3.5.1 Conversion Mnémonique vers OpCode

```
1  public String InstructionIoOpCode(String P1, char P21, String T) {
2      String X = "";
3
4      if (P21 == '#') {  // Mode Immediat
5          // Instructions de chargement
6          if (P1.equalsIgnoreCase("LDA"))
7              X = "86";
8          else if (P1.equalsIgnoreCase("LDB"))
9              X = "C6";
10         else if (P1.equalsIgnoreCase("LDD"))
```

```
11              X = "CC";
12          else if (P1.equalsIgnoreCase("LDS"))
13              X = "10CE";  // Opcode sur 2 octets (prefixe 10)
14          else if (P1.equalsIgnoreCase("LDY"))
15              X = "108E";  // Opcode sur 2 octets (prefixe 10)
16
17          // Instructions arithmetiques
18          else if (P1.equalsIgnoreCase("ADDA"))
19              X = "8B";
20          else if (P1.equalsIgnoreCase("ADDB"))
21              X = "CB";
22          else if (P1.equalsIgnoreCase("SUBA"))
23              X = "80";
24          else if (P1.equalsIgnoreCase("SUBB"))
25              X = "C0";
26
27          // Instructions logiques
28          else if (P1.equalsIgnoreCase("ANDA"))
29              X = "84";
30          else if (P1.equalsIgnoreCase("ORA"))
31              X = "8A";
32      }
33      return X;
34  }
```

Listing 3.7 – Méthode InstructionIoOpCode - Table de correspondance

Analyse technique :

- **Structure** : Table de correspondance implémentée via chaîne `if-else`. Bien que verbeux, cet approche est claire et facilement extensible.
- **Lignes 12-15** : Les instructions `LDS` et `LDY` utilisent un **préfixe $10**, caractéristique du 6809 pour étendre le jeu d'instructions.
- **Opcodes retournés** : Format String hexadécimal pour faciliter le traitement ultérieur. Exemples :

Instruction	OpCode	Taille	Total (avec opérande)
`LDA #$99`	`$86`	1 octet	2 octets
`LDD #$1234`	`$CC`	1 octet	3 octets
`LDS #$FFFF`	`$10CE`	2 octets	4 octets
`CLRA`	`$4F`	1 octet	1 octet (inhérent)

TABLE 3.1 – Exemples d'opcodes et tailles d'instructions

3.6 Gestion de la Mémoire : Memoire.java

3.6.1 Initialisation de la ROM

```
public int[] AdressROM() {
    int[] AdressROM = new int[1024];
    for (int i = 0; i < 1024; i++) {
        AdressROM[i] = 0xFC00 + i;  // Adresses de FC00 a FFFF
    }
    return AdressROM;
}

public int[] InitValeurROM() {
    int[] ValeurROM = new int[1024];
    for (int k = 0; k < 1024; k++) {
        ValeurROM[k] = 0xFF;  // Valeur par defaut (octet non
            programme)
    }
    return ValeurROM;
}

public int[] ValeurROM(ArrayList<String> AA, String[] P1, char[] P21,
                       String P22[], int P1_NumberOctets[],
                       int P22_NumberOctets[], int taille) {
    int ValueROM[] = new int[1024];
    int j = 0;

    for (int k = 0; k < taille; k++) {
        String Y = "";

        if (P1_NumberOctets[k] == 1) {
            // Opcode sur 1 octet (ex: LDA -> 86)
            Y = MEM_DEC.InstructionIoOpCode(P1[k], P21[k],
                AA.get(k));
            ValueROM[j] = MEM_DEC.hexToSignedInt(Y.substring(0, 2))
                & 0xFF;
            j++;
        }

        if (P1_NumberOctets[k] == 2) {
            // Opcode sur 2 octets (ex: LDS -> 10CE)
            Y = MEM_DEC.InstructionIoOpCode(P1[k], P21[k],
                AA.get(k));
            // Ecriture octet par octet
            ValueROM[j] = MEM_DEC.HexaStringToInt(Y.substring(0, 2))
                & 0xFF;
```

```
38              j++;
39              ValueROM[j] = MEM_DEC.HexaStringToInt(Y.substring(2, 4))
                    & 0xFF;
40              j++;
41          }
42
43          // Ecriture de l'operande
44          if (P22_NumberOctets[k] == 1) {
45              ValueROM[j] = MEM_DEC.hexToSignedInt(P22[k]) & 0xFF;
46              j++;
47          }
48
49          ValueROM[j] = 0x3F;  // Marqueur SWI (fin)
50      }
51      return ValueROM;
52  }
```

Listing 3.8 – Initialisation et écriture en ROM

Analyse technique :

- **Lignes 3-4** : La ROM du 6809 est mappée en haut de l'espace d'adressage (`$FC00`-`$FFFF`), zone traditionnellement réservée au code de démarrage.
- **Ligne 12** : Initialisation à `$FF` (tous bits à 1), convention pour les mémoires EPROM vierges.
- **Lignes 26-30** : Traitement des opcodes sur 1 octet. Le masquage `& 0xFF` garantit une valeur sur 8 bits.
- **Lignes 33-41** : Pour les opcodes sur 2 octets, décomposition en poids fort puis poids faible (Big Endian, convention Motorola).
- **Ligne 49** : Le marqueur `$3F` (instruction `SWI`) signale la fin du programme au simulateur.

3.7 Unité Arithmétique et Logique : UAL.java

3.7.1 Structure de Données de Retour

```
1  public static class CalculMembers {
2      public ArrayList<Integer> A;    // Historique accumulateur A
3      public ArrayList<Integer> B;    // Historique accumulateur B
4      public ArrayList<Integer> S;    // Historique pointeur pile S
5      public ArrayList<Integer> U;    // Historique pointeur pile U
6      public ArrayList<Integer> X;    // Historique registre index X
7      public ArrayList<Integer> Y;    // Historique registre index Y
```

```
8     public ArrayList<Integer> DP;    // Historique page directe
9     public ArrayList<int[]> CCR;     // Historique flags (8 bits)
10    public ArrayList<int[]> Valram; // Historique etat RAM
11
12    public CalculMembers(ArrayList<Integer> A, ArrayList<Integer> B,
13                         ArrayList<Integer> S, ArrayList<Integer> Y,
14                         ArrayList<Integer> X, ArrayList<Integer> U,
15                         ArrayList<Integer> DP, ArrayList<int[]> CCR,
16                         ArrayList<int[]> Valram) {
17        this.A = A; this.B = B; this.S = S; this.Y = Y;
18        this.X = X; this.U = U; this.DP = DP;
19        this.CCR = CCR; this.Valram = Valram;
20    }
21 }
```

Listing 3.9 – Classe interne CalculMembers - DTO pour les résultats

Analyse technique :

- **Pattern DTO** : `CalculMembers` est un **Data Transfer Object** permettant de retourner multiple valeurs depuis `Calcule()`.
- **ArrayList vs Array** : L'utilisation d'ArrayList permet de stocker un nombre variable d'états (1 par instruction exécutée).
- **Ligne 9** : `CCR` est un `ArrayList<int[]>` où chaque `int[8]` représente l'état des 8 flags après une instruction.

3.7.2 Vérification du Drapeau Carry (Retenue)

```
1  public int CheckCarry8(int A[], int ADDA[]) {
2      int Carry = 0;
3      int i = 7;  // Bit de poids fort
4
5      // Cas trivial : somme des bits de poids fort
6      if (A[i] + ADDA[i] == 2)
7          Carry = 1;  // 1 + 1 = 10 binaire -> retenue
8
9      if (A[i] + ADDA[i] == 0)
10         Carry = 0;  // 0 + 0 = 0 -> pas de retenue
11
12     // Cas 1+0 ou 0+1 : propagation possible
13     while (A[i] + ADDA[i] == 1 && i >= 0) {
14         i--;
15         if (A[i] + ADDA[i] == 2) {
16             Carry = 1;  // Retenue propagee
17             break;
```

```
18          } else if (A[i] + ADDA[i] == 0) {
19              Carry = 0;  // Propagation arretee
20              break;
21          }
22      }
23      return Carry;
24  }
```

Listing 3.10 – Méthode CheckCarry8 - Détection de retenue sur 8 bits

Analyse technique :

- **Entrées** : Deux tableaux de 8 entiers représentant les bits des opérandes (LSB en index 0).
- **Lignes 6-10** : Détection directe au bit 7 (MSB) : si les deux bits valent 1, il y a forcément retenue.
- **Lignes 13-22** : Algorithme de propagation de retenue. Si la somme vaut 1 (0+1 ou 1+0), il faut vérifier si une retenue arrive des bits inférieurs.

Note Importante

Cet algorithme implémente la logique d'un **additionneur à propagation de retenue** (Ripple Carry Adder), fidèle au comportement matériel du 6809.

3.7.3 Vérification du Drapeau Overflow (Débordement)

```
1  public int CheckOverFlow(int entr1, int entr2, int Result) {
2      int V = 0;
3
4      // Overflow possible seulement si les deux entrees ont le meme
           signe
5      if (entr1 * entr2 >= 0) {
6          // Verifier si le resultat a change de signe
7          if (entr1 * Result >= 0) {
8              V = 0;  // Meme signe -> pas d'overflow
9          } else {
10             V = 1;  // Signe different -> OVERFLOW
11         }
12     } else {
13         V = 0;  // Signes opposes -> jamais d'overflow
14     }
15     return V;
16 }
```

Listing 3.11 – Méthode CheckOverFlow - Détection de débordement signé

Analyse technique :

- **Principe mathématique** : Un débordement signé survient lorsque deux nombres de même signe produisent un résultat de signe opposé.
- **Ligne 5** : `entr1 * entr2 >= 0` vérifie si les deux entrées ont le même signe (produit positif) ou sont de signes opposés (produit négatif).
- **Lignes 7-10** : Si les entrées ont le même signe, on vérifie si le résultat a conservé ce signe. Si non, c'est un overflow.
- **Exemple** : $127 + 1 = 128$ en non-signé, mais -128 en signé 8 bits $\Rightarrow$ Overflow $= 1$.

3.7.4 Conversion Décimal vers Complément à 2

```
1  public int[] DecimalToTwosComplement8(int valeur) {
2      int[] resultat = new int[8];
3
4      for (int i = 0; i < 8; i++) {
5          resultat[i] = (valeur >> i) & 1;
6      }
7
8      return resultat;
9  }
10
11 public int[] DecimalToTwosComplement16(int valeur) {
12     int[] resultat = new int[16];
13
14     for (int i = 0; i < 16; i++) {
15         resultat[i] = (valeur >> i) & 1;
16     }
17
18     return resultat;
19 }
```

Listing 3.12 – Conversion en représentation binaire complément à 2

Analyse technique :

- **Ligne 5** : L'opérateur » effectue un décalage à droite de `i` positions, puis `& 1` isole le bit de poids faible.
- **Résultat** : Tableau où `resultat[0]` contient le LSB et `resultat[7]` (ou [15]) le MSB.
- **Complément à 2** : Java utilise nativement le complément à 2 pour les entiers signés, donc aucune conversion supplémentaire n'est nécessaire.

3.7.5 Exécution d'une Instruction : Exemple LDA

```
1  if (P1[k].equalsIgnoreCase("LDA")) {
2      entry1 = 0; entry2 = 0; res = 0;
3      entry1 = A;  // Ancienne valeur de A
4
5      // Chargement de la nouvelle valeur depuis la ROM
6      A = ValueROM[Temp - 1] & 0xFF;
7      entry2 = ValueROM[Temp - 1];
8
9      // Conversion en binaire pour calcul des flags
10     entry1Bin8 = DecimalToTwosComplement8(entry1);
11     entry2Bin8 = DecimalToTwosComplement8(entry2);
12     res = toSignedByte(A, 1);
13
14     // Mise a jour des flags CCR
15     flags[0] = CheckParity(res);          // P - Parite
16     flags[4] = CheckSigne(res);           // N - Negatif
17     flags[5] = CheckZero(res);            // Z - Zero
18     flags[7] = CheckOverFlow(entry1, entry2, res);  // V - Overflow
19
20     CCR.add(flags.clone());               // Sauvegarde etat flags
21     Valram.add(valram.clone());           // Sauvegarde etat RAM
22
23     // Enregistrement des valeurs de tous les registres
24     valeursA.add(toSignedByte(A, 1));
25     valeursB.add(toSignedByte(B, 1));
26     valeursDP.add(toSignedByte(DP, 1));
27     valeursS.add(toSignedByte(S, 2));
28     valeursY.add(toSignedByte(Y, 2));
29     valeursX.add(toSignedByte(X, 2));
30     valeursU.add(toSignedByte(U, 2));
31 }
```

Listing 3.13 – Traitement de l'instruction LDA dans la méthode Calcule

Analyse technique :

- **Ligne 6** : Lecture de l'opérande depuis la ROM. `Temp - 1` pointe sur l'octet suivant l'opcode. Le masquage `& 0xFF` garantit une valeur non signée sur 8 bits.
- **Lignes 15-18** : Calcul des flags affectés par `LDA` :
  - **N** (Negative) : mis à 1 si le bit 7 du résultat est 1
  - **Z** (Zero) : mis à 1 si le résultat est 0
  - **V** (Overflow) : toujours remis à 0 pour LDA

- **Ligne 20** : `flags.clone()` crée une copie du tableau pour éviter que les modifications ultérieures n'affectent les valeurs déjà enregistrées.
- **Lignes 24-30** : Même si seul A est modifié, tous les registres sont enregistrés pour maintenir la synchronisation dans le mode pas-à-pas.

3.8 Validation des Erreurs : Erreurs.java

```
1  public boolean CheckSyntaxErrors(String T, String P1, char P21,
       String P22) {
2      if (
3          // Mode Immediat 8 bits (LDA, ADDA, etc.)
4          ((CheckInstructionP1_2_octet(P1)) &&
               (CheckModeImmediat(P21)) &&
5           (isHexadecimal(P22)) && (CheckHexa1octet(P22)) &&
               (CheckSpaces(T)))
6          ||
7          // Mode Immediat 16 bits (LDD, LDX, etc.)
8          ((CheckInstructionP1_3_octet(P1)) &&
               (CheckModeImmediat(P21)) &&
9           (isHexadecimal(P22)) && (CheckHexa2octet(P22)) &&
               (CheckSpaces(T)))
10         ||
11         // Mode Immediat 16 bits avec prefixe (LDS, LDY)
12         ((CheckInstructionP1_4_octet(P1)) &&
               (CheckModeImmediat(P21)) &&
13          (isHexadecimal(P22)) && (CheckHexa2octet(P22)) &&
               (CheckSpaces(T)))
14         ||
15         // Mode Direct
16         ((CheckDirect(P1)) && (CheckModeDirect(P21)) &&
17          (isHexadecimal(P22)) && (CheckHexa1octet(P22)) &&
               (CheckSpaces(T)))
18         ||
19         // Mode Etendu
20         ((CheckEtendu(P1)) && (CheckModeEtendu(P21)) &&
21          (isHexadecimal(P22)) && (CheckHexa2octet(P22)) &&
               (CheckSpaces(T)))
22         ||
23         // Transfert/Echange (TFR, EXG)
24         (CheckTFR(T))
25         ||
26         // Mode Inherent (CLRA, NOP, etc.)
27         (CheckInherant(T))
28     ) {
29         return true;  // Syntaxe correcte
```

```
30      }
31      return false;  // Erreur de syntaxe
32  }
```

Listing 3.14 – Méthode CheckSyntaxErrors - Validation complète d'une ligne

Analyse technique :

- **Structure** : Expression booléenne composée de 7 clauses OR, chacune représentant un cas de syntaxe valide.
- **Lignes 4-5** : Validation du mode immédiat 8 bits. Les 5 conditions doivent être vraies simultanément :
  1. Mnémonique dans la liste des instructions 8 bits
  2. Caractère de mode = '#'
  3. Opérande au format hexadécimal valide
  4. Opérande sur exactement 2 caractères (1 octet)
  5. Exactement un espace dans la ligne
- **Lignes 24-27** : Les instructions `TFR`, `EXG`, et inhérentes sont traitées comme des cas spéciaux car leur syntaxe diffère (pas de '#$').

3.9 Calcul du Program Counter : Registres.java

```
1  public int[] PC(int[] P1_NumberOctets, int P22_NumberOctets[],
2                  int AddROM[], int taille) {
3      int[] PC = new int[taille + 1];
4
5      int k = 0;
6      int temp = 0;
7
8      while (k < taille) {
9          PC[k] = AddROM[temp];  // Adresse de l'instruction k
10
11         // Avancer de (taille opcode + taille operande) octets
12         temp += P1_NumberOctets[k] + P22_NumberOctets[k];
13         k++;
14     }
15
16     PC[k] = AddROM[temp];  // Adresse finale (apres derniere
           instruction)
17     return PC;
18 }
```

Listing 3.15 – Méthode PC - Calcul des adresses d'instructions

Analyse technique :

- **Ligne 3** : Le tableau PC a une taille de `taille + 1` pour stocker l'adresse de fin.
- **Ligne 9** : `AddROM[temp]` donne l'adresse absolue en ROM (à partir de `$FC00`).
- **Ligne 12** : L'incrémentation de `temp` simule l'avancement du PC réel. Par exemple :
  - `LDA #$99` : 1 (opcode) + 1 (opérande) = 2 octets
  - `LDS #$1234` : 2 (opcode 10CE) + 2 (opérande) = 4 octets

k	Instruction	PC[k]	Taille	temp après
0	`LDA #$99`	`$FC00`	2	2
1	`ADDA #$01`	`$FC02`	2	4
2	`LDS #$FFFF`	`$FC04`	4	8
3	`END`	`$FC08`	-	-

TABLE 3.2 – Exemple de calcul du Program Counter

Chapitre 4

Conclusion

4.1 Bilan du Projet

Le projet **MOF6809** a permis de développer un simulateur fonctionnel et pédagogique du microprocesseur Motorola 6809. Ce travail a atteint les objectifs fixés initialement, tout en révélant des axes d'amélioration pour de futures versions.

4.1.1 Objectifs Atteints

Objectif	Statut	Commentaire
Interface graphique intuitive	**Atteint**	Swing avec visualisation RAM/ROM/Registres
Compilation assembleur → code machine	**Atteint**	Analyse lexicale et syntaxique fonctionnelle
Mode d'adressage Immédiat	**Atteint**	Support complet avec validation
Mode d'adressage Direct	**Atteint**	Adressage page zéro implémenté
Mode d'adressage Étendu	**Atteint**	Adressage 16 bits complet
Mode d'adressage Inhérent	**Atteint**	Instructions sans opérande
Exécution pas-à-pas	**Atteint**	Débogage instruction par instruction
Gestion des flags CCR	**Atteint**	C, Z, N, V, H calculés correctement
Détection d'erreurs syntaxiques	**Atteint**	Messages d'erreur avec numéro de ligne

TABLE 4.1 – Récapitulatif des objectifs atteints

4.1.2 Compétences Développées

Ce projet a permis à l'équipe de développer et consolider plusieurs compétences :

- **Architecture des processeurs** : Compréhension approfondie du cycle fetch-decode-execute, de la gestion des registres et des modes d'adressage.
- **Programmation Java** : Maîtrise de Swing pour les interfaces graphiques, manipulation de structures de données (ArrayList, tableaux), et gestion d'événements.
- **Compilation** : Implémentation d'un compilateur simplifié avec analyse lexicale (tokenisation) et génération de code machine.
- **Arithmétique binaire** : Manipulation du complément à 2, détection de retenue et de débordement, opérations bit à bit.
- **Travail en équipe** : Répartition des tâches, intégration de modules développés séparément, documentation technique.

4.2 Limitations Identifiées

Malgré les fonctionnalités implémentées, certaines limitations subsistent :

Attention

Les limitations suivantes sont documentées pour orienter les futures évolutions du projet.

4.2.1 Limitations Fonctionnelles

1. **Mode d'adressage Indexé non supporté** : Les instructions utilisant les registres X et Y comme base d'adressage (ex : `LDA ,X` ou `LDA 5,Y`) ne sont pas implémentées.
2. **Mode d'adressage Relatif absent** : Les instructions de branchement conditionnel (BEQ, BNE, BRA, etc.) ne sont pas supportées, limitant la création de boucles et conditions.
3. **Pas de gestion des interruptions** : Les vecteurs d'interruption (IRQ, FIRQ, NMI, SWI) ne sont pas simulés.
4. **Jeu d'instructions partiel** : Environ 40% des 59 instructions du 6809 sont implémentées.

4.2.2 Limitations Techniques

1. **Architecture monolithique** : La classe `UAL` (2907 lignes) gagnerait à être refactorisée en plusieurs classes spécialisées.
2. **Absence de tests unitaires** : Le projet ne dispose pas de suite de tests automatisés pour valider les modifications.
3. **Pas de sauvegarde/chargement** : Impossible de sauvegarder un programme ou l'état du simulateur.
4. **Performance** : Le pré-calcul de tous les états lors de la compilation peut être lent pour de longs programmes.

4.3 Perspectives d'Amélioration

4.3.1 Améliorations à Court Terme

Amélioration	Priorité	Description
Mode Indexé	Haute	Implémenter `LDA ,X`, `LDA n,X`, etc.
Branchements	Haute	Ajouter BEQ, BNE, BRA, BCC, BCS
Sauvegarde programme	Moyenne	Export/Import de fichiers .asm
Coloration syntaxique	Moyenne	Éditeur avec highlighting
Historique d'exécution	Basse	Liste des instructions exécutées

TABLE 4.2 – Améliorations prioritaires

4.3.2 Améliorations à Long Terme

- **Refactoring complet** : Appliquer les design patterns (Strategy pour les instructions, Observer pour l'UI) et diviser l'UAL en modules.
- **Tests automatisés** : Créer une suite JUnit couvrant toutes les instructions et modes d'adressage.
- **Émulation cycle-accurate** : Simuler les cycles d'horloge pour une fidélité temporelle au matériel.
- **Périphériques virtuels** : Ajouter des entrées/sorties simulées (clavier, afficheur 7 segments).
- **Interface moderne** : Migration vers JavaFX pour une UI plus réactive et esthétique.
- **Version Web** : Portage en JavaScript/WebAssembly pour un accès sans installation.

4.4 Statistiques du Projet

Métrique	Valeur
Nombre total de lignes de code	**∼5 200**
Nombre de classes Java	**7**
Nombre d'instructions supportées	**∼45**
Taille mémoire ROM simulée	**1 024 octets**
Taille mémoire RAM simulée	**1 280 octets**
Nombre de registres simulés	**9**
Modes d'adressage supportés	**4**
Membres de l'équipe	**4**

TABLE 4.3 – Statistiques du projet MOF6809

4.5 Remerciements

L'équipe tient à remercier :

* L'encadrement pédagogique pour les conseils et le suivi tout au long du projet.
* La communauté des développeurs d'émulateurs pour les ressources documentaires sur le 6809.
* Motorola (maintenant NXP/Freescale) pour avoir conçu ce processeur remarquable.

4.6 Mot de la Fin

«Le microprocesseur 6809, bien que conçu il y a plus de 45 ans, reste un excellent support pédagogique pour comprendre les fondements de l'architecture des ordinateurs. Ce projet nous a permis de toucher du doigt la complexité et l'élégance de ces systèmes qui ont posé les bases de l'informatique moderne.»

— L'équipe MOF6809

Annexe A

Annexe : Jeu d'Instructions Supporté

A.1 Instructions de Chargement (LD)

Mnémo.	OpCode	Octets	Mode	Action
LDA	$86	2	Immédiat	A ← valeur
LDB	$C6	2	Immédiat	B ← valeur
LDD	$CC	3	Immédiat	D ← valeur 16 bits
LDS	$10CE	4	Immédiat	S ← valeur 16 bits
LDU	$CE	3	Immédiat	U ← valeur 16 bits
LDX	$8E	3	Immédiat	X ← valeur 16 bits
LDY	$108E	4	Immédiat	Y ← valeur 16 bits

Table A.1 – Instructions de chargement

A.2 Instructions Arithmétiques

Mnémo.	OpCode	Octets	Mode	Action
ADDA	$8B	2	Immédiat	A ← A + valeur
ADDB	$CB	2	Immédiat	B ← B + valeur
ADDD	$C3	3	Immédiat	D ← D + valeur
SUBA	$80	2	Immédiat	A ← A - valeur
SUBB	$C0	2	Immédiat	B ← B - valeur
SUBD	$83	3	Immédiat	D ← D - valeur
MUL	$3D	1	Inhérent	D ← A × B

TABLE A.2 – Instructions arithmétiques

A.3 Instructions Logiques

Mnémo.	OpCode	Octets	Mode	Action
ANDA	$84	2	Immédiat	A ← A AND valeur
ANDB	$C4	2	Immédiat	B ← B AND valeur
ORA	$8A	2	Immédiat	A ← A OR valeur
ORB	$CA	2	Immédiat	B ← B OR valeur

TABLE A.3 – Instructions logiques

A.4 Instructions Inhérentes

Mnémonique	OpCode	Octets	Action
ABX	$3A	1	X ← X + B (non signé)
CLRA	$4F	1	A ← 0
CLRB	$5F	1	B ← 0
DECA	$4A	1	A ← A - 1
DECB	$5A	1	B ← B - 1
INCA	$4C	1	A ← A + 1
INCB	$5C	1	B ← B + 1
NOP	$12	1	Aucune opération

TABLE A.4 – Instructions inhérentes (sans opérande)

A.5 Instructions de Transfert

Mnémonique	OpCode	Octets	Action
TFR A,B	$1F89	2	B ← A
TFR B,A	$1F98	2	A ← B
EXG A,B	$1E89	2	A ↔ B
TFR X,Y	$1F12	2	Y ← X
TFR D,X	$1F04	2	X ← D

TABLE A.5 – Instructions de transfert et d'échange