Université Hassan 1er

Faculté des Sciences et Techniques de Settat

Département de Mathématiques et Informatique

RAPPORT DE PROJET

Simulateur de Microprocesseur Motorola 6809

Module : Architecture des ordinateurs et assembleur

Filière : LST Génie Informatique

Réalisé par :
CHAHOUB Nouhaila
ENNAJI Aymen
AMROUN Hiba

Encadré par :
Pr. BENALLA Hicham

Année Universitaire : 2025 - 2026

Remerciements

Nous tenons à exprimer notre profonde gratitude envers toutes les personnes qui ont contribué à la réalisation de ce projet.

Nos remerciements s'adressent en premier lieu au **Professeur BENALLA Hicham**, notre encadrant pédagogique, pour ses conseils avisés, son accompagnement constant, et pour nous avoir transmis les connaissances essentielles en architecture des ordinateurs et programmation assembleur.

Nous remercions également l'ensemble du corps enseignant de la **Faculté des Sciences et Techniques de Settat**, particulièrement le département de Mathématiques et Informatique, pour la qualité de la formation dispensée. Nous adressons nos sincères remerciements aux membres du jury qui ont accepté d'évaluer ce travail.

Résumé

Ce rapport présente la conception et la réalisation d'un **simulateur complet du microprocesseur Motorola 6809**, développé en Java dans le cadre du module d'Architecture des Ordinateurs. Le Motorola 6809 est reconnu comme l'un des microprocesseurs 8 bits les plus avancés grâce à son architecture sophistiquée.

Le simulateur implémente l'intégralité du jeu d'instructions du 6809, avec une **interface graphique ergonomique** permettant de visualiser en temps réel l'état des registres, le contenu de la mémoire, et l'exécution pas-à-pas des programmes.

Les fonctionnalités principales incluent :

— Éditeur de code assembleur avec validation en temps réel
— Moteur d'exécution supportant tous les modes d'adressage
— Débogueur avec navigation avant/arrière
— Visualiseurs interactifs pour RAM, ROM et registres
— Système de gestion d'état avec historique

Mots-clés : Microprocesseur, Motorola 6809, Simulateur, Assembleur, Java, Débogueur.

Abstract

This report presents the design and implementation of a **complete Motorola 6809 microprocessor simulator**, developed in Java for the Computer Architecture module. The simulator implements the full 6809 instruction set with an ergonomic GUI for real-time visualization of registers, memory, and step-by-step execution.

Keywords : Microprocessor, Motorola 6809, Simulator, Assembler, Java, Debugger.

Table des matières

Table des figures

Liste des tableaux

Liste des Abréviations

ALU	Arithmetic Logic Unit
CCR	Condition Code Register
CPU	Central Processing Unit
GUI	Graphical User Interface
PC	Program Counter
RAM	Random Access Memory
ROM	Read-Only Memory
UAL	Unité Arithmétique et Logique

Introduction Générale

Contexte du Projet

Le microprocesseur constitue le cœur de tout système informatique. Comprendre son fonctionnement interne est fondamental pour tout informaticien. Le **Motorola 6809**, introduit en 1978, représente une étape importante dans l'évolution des microprocesseurs 8 bits.

Ce projet s'inscrit dans le module *Architecture des Ordinateurs et Assembleur* de la LST Génie Informatique à la FST Settat.

Problématique

L'apprentissage de l'architecture des microprocesseurs présente plusieurs défis :

— Difficulté de visualiser le fonctionnement interne
— Absence d'environnement pratique d'expérimentation
— Besoin d'outils de débogage adaptés
— Complexité de la gestion des registres et mémoire

Objectifs

Objectifs Fonctionnels :

1. Implémenter un moteur d'exécution fidèle au 6809
2. Fournir une interface graphique intuitive
3. Développer un débogueur avec exécution pas-à-pas
4. Intégrer un système de validation syntaxique

Objectifs Pédagogiques :

1. Faciliter la compréhension du cycle d'instruction
2. Visualiser l'impact sur registres et mémoire
3. Démontrer les modes d'adressage

Méthodologie

Le développement a suivi :

1. **Phase d'Analyse :** Étude de l'architecture 6809
2. **Phase de Conception :** Modélisation UML

3. **Phase de Réalisation :** Implémentation itérative
4. **Phase de Tests :** Validation fonctionnelle
5. **Phase de Documentation :** Rédaction du rapport

Chapitre 1

Fondements Théoriques

1.1 Architecture Générale du Microprocesseur

1.1.1 Le Cycle d'Exécution

Le fonctionnement repose sur un cycle répétitif à trois phases :

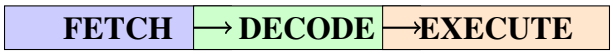

FIGURE 1.1 – Cycle d'Exécution des Instructions

1. FETCH : Le PC pointe vers l'instruction en mémoire, elle est chargée dans le RI, le PC est incrémenté.
2. DECODE : L'unité de contrôle analyse l'opcode et détermine le mode d'adressage.
3. EXECUTE : L'opération est effectuée, les flags CCR sont mis à jour.

1.2 Les Registres

TABLE 1.1 – Registres du Motorola 6809

Registre	Taille	Fonction
A	8 bits	Accumulateur
B	8 bits	Accumulateur
D	16 bits	Concaténation A :B
X, Y	16 bits	Registres d'index
S, U	16 bits	Pointeurs de pile
PC	16 bits	Program Counter
CCR	8 bits	Flags de condition

1.2.1 Le Registre CCR

TABLE 1.2 – Structure du CCR

Bit 7	6	5	4	3	2	1	0
E	F	H	I	N	Z	V	C

— **C (Carry)** : Retenue
— **V (Overflow)** : Débordement
— **Z (Zero)** : Résultat nul
— **N (Negative)** : Résultat négatif
— **I (IRQ Mask)** : Masque IRQ
— **H (Half-Carry)** : Demi-retenue (BCD)
— **F (FIRQ Mask)** : Masque FIRQ
— **E (Entire)** : Sauvegarde complète

1.3 Les Modes d'Adressage

TABLE 1.3 – Modes d'Adressage

Mode	Syntaxe	Exemple
Inhérent	-	`NOP`
Immédiat	`#$XX`	`LDA #$FF`
Direct	`<$XX`	`LDA <$80`
Étendu	`$XXXX`	`LDA $2000`
Indexé	`,X`	`LDA ,X`

1.3.1 Mode Indexé Avancé

Le mode indexé offre plusieurs variantes :
— Sans déplacement : `,X`
— Déplacement constant : `5,X`
— Accumulateur : `A,X` ou `D,X`
— Auto-incrémentation : `,X+` ou `,X++`
— Auto-décrémentation : `,-X` ou `,--X`
— Indirect : `[,X]`
Calcul de l'adresse effective :

$$EA = Base + Deplacement$$

1.4 Jeu d'Instructions

TABLE 1.4 – Classification des Instructions

Catégorie	Exemples
Transfert	LDA, LDB, STA, STB
Arithmétiques	ADDA, SUBA, MUL
Logiques	ANDA, ORA, EORA
Déplacements	ASLA, LSRA, ROLA
Comparaison	CMPA, CMPX
Branchements	BEQ, BNE, JMP
Pile	PSHS, PULS

1.5 Espace Mémoire

Le 6809 adresse 64 Ko (216 = 65536 octets) :
— **RAM** : 0x0000 - 0xEFFF
— **ROM** : 0xF000 - 0xFFF9
— **Vecteurs** : 0xFFFA - 0xFFFF

Chapitre 2

Analyse et Spécification

2.1 Analyse des Besoins

2.1.1 Besoins Fonctionnels

BF1 - Gestion du Code
— Saisie via éditeur intégré
— Validation syntaxique en temps réel
— Messages d'erreur explicites
— Sauvegarde/chargement fichiers
BF2 - Exécution
— Compilation assembleur → machine
— Support tous modes d'adressage
— Gestion correcte des flags
BF3 - Débogage
— Exécution pas-à-pas
— Navigation avant/arrière
— Affichage instruction courante
BF4 - Visualisation
— État registres en temps réel
— Contenu RAM/ROM avec pagination
— Édition manuelle registres/mémoire

2.1.2 Besoins Non-Fonctionnels

— **Performance** : < 100ms par instruction
— **Utilisabilité** : Interface intuitive
— **Fiabilité** : Pas de crash
— **Maintenabilité** : Code documenté et modulaire

2.2 Cas d'Utilisation

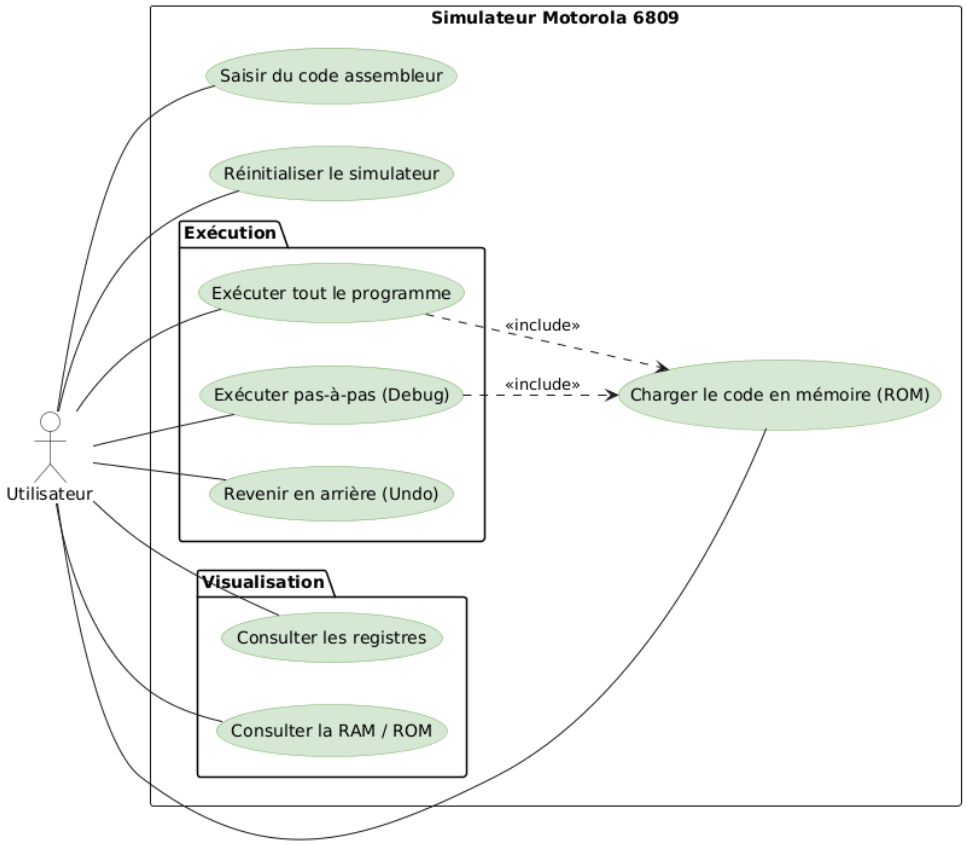

FIGURE 2.1 – Diagramme de Cas d'Utilisation

2.2.1 UC1 : Saisir du Code

Scénario nominal :

1. Utilisateur clique "New"
2. Système ouvre éditeur
3. Utilisateur saisit instructions
4. Système valide syntaxe
5. Utilisateur clique "Load"
6. Système compile et charge

2.2.2 UC2 : Exécuter Pas-à-Pas

Scénario nominal :

1. Utilisateur clique "Step"
2. Système ouvre débogueur
3. Utilisateur clique "Next"
4. Système exécute instruction

5. Système met à jour affichage
6. Répéter jusqu'à END

2.3 Matrice de Priorités

TABLE 2.1 – Prioritisation des Besoins

Besoin	**Priorité**	**Statut**
Instructions de base	Haute	✓
Modes d'adressage simples	Haute	✓
Visualisation registres	Haute	✓
Validation syntaxique	Haute	✓
Exécution pas-à-pas	Haute	✓
Modes indexés avancés	Haute	✓
Navigation arrière	Moyenne	✓

Chapitre 3

Conception Architecturale

3.1 Architecture Globale

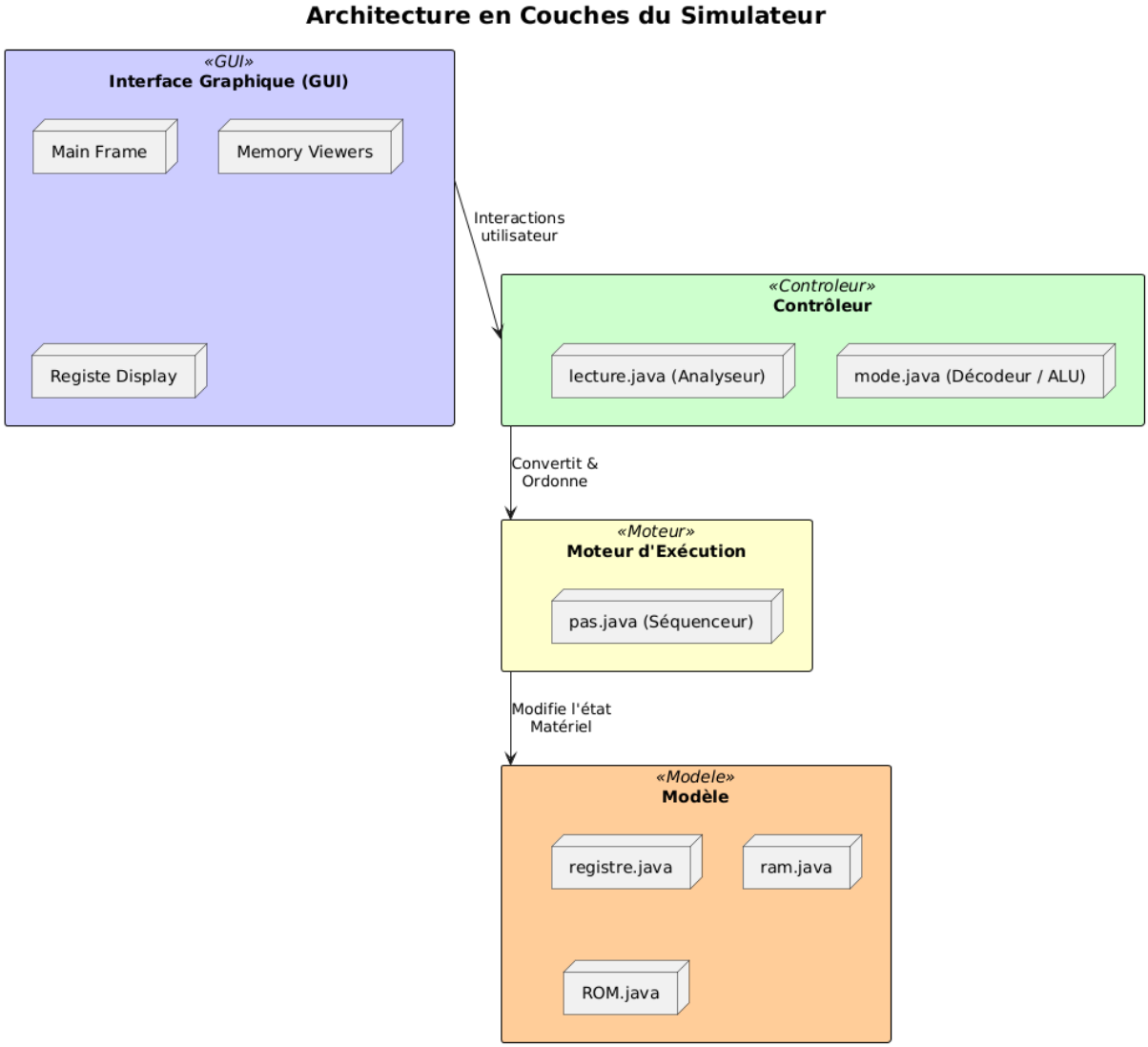

FIGURE 3.1 – Architecture en Couches

3.2 Diagramme de Classes Principal

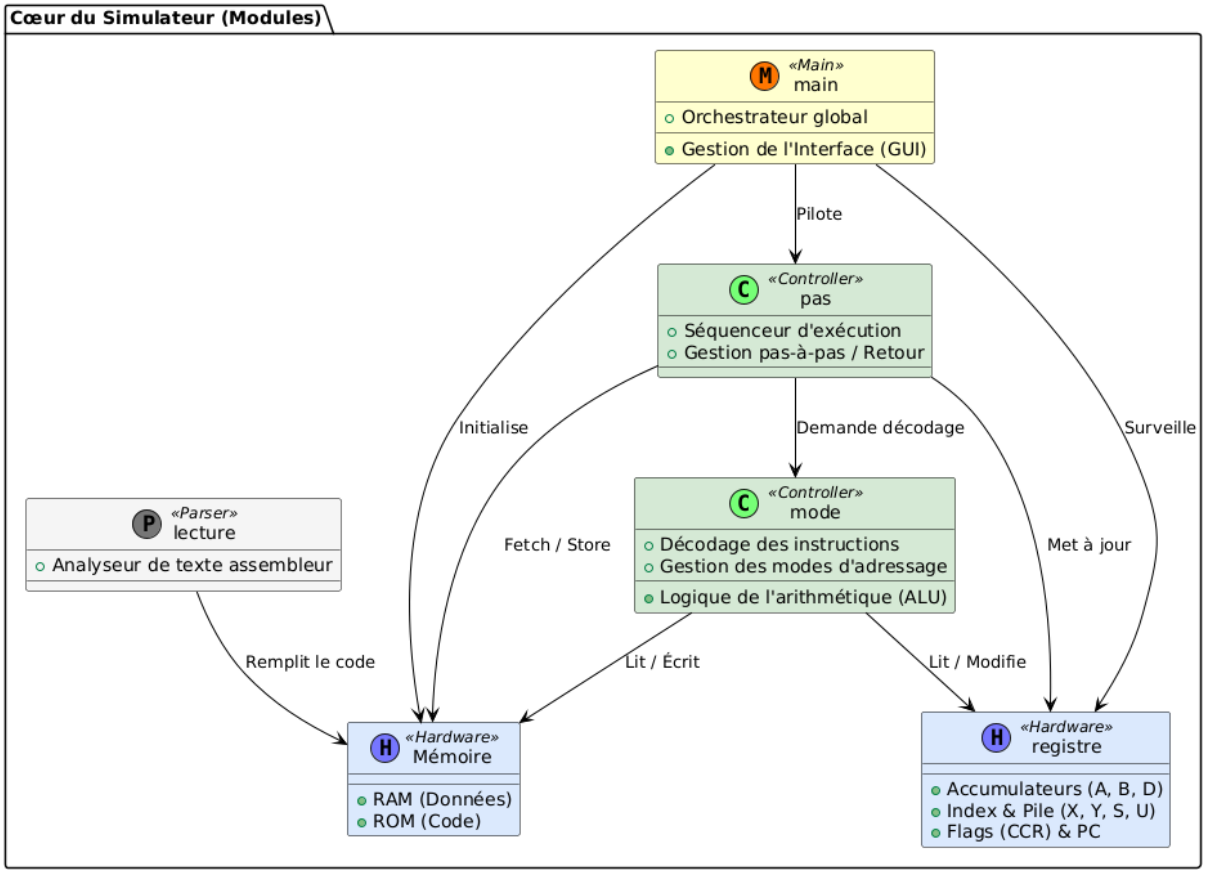

FIGURE 3.2 – Diagramme de classe

3.3 Design Patterns

3.3.1 Singleton - BackendManager

Garantit une instance unique du gestionnaire :

```java
public class BackendManager {
    private static BackendManager instance;
    private ROM rom;
    private ram ram;
    private registre reg;
    private mode modeDetector;
    private ArrayList<ArrayList<String>> assemblyLines;

    private BackendManager() {}

    public static BackendManager getInstance() {
        if (instance == null) {
            instance = new BackendManager();
        }
        return instance;
    }

    public void initialize(ROM rom, ram ram, registre reg, mode modeDetector) {
        this.rom = rom;
        this.ram = ram;
        this.reg = reg;
        this.modeDetector = modeDetector;
    }

    public void executeStep() {
        // Call your pas.java to execute one step
        // Then notify GUI components to update
    }

    // Gette
```

FIGURE 3.3 – Pattern Singleton – Implémentation en Java

3.3.2 Observer - Mise à Jour GUI

Utilisation de `Timer` Swing pour rafraîchissement :

```
updateTimer = new Timer(500, e -> updateDisplay());
updateTimer.start();
```

Listing 3.1 – Pattern Observer avec Timer

3.3.3 State - Gestion Historique

La classe `StateManager` sauvegarde les états :

```java
public class StateManager {
    private ArrayList<ExecutionState> stateHistory = new ArrayList<>();
    private int currentStateIndex = -1;

    // Public inner class so pas.java can access it
    public static class ExecutionState {
        public Map<String, String> ramState;
        public Map<String, String> romState;
        public Map<String, String> registerState;
        public int romWritePointer;
        public int currentStep;
        public String description;

        public ExecutionState(Map<String, String> ram, Map<String, String> rom,
                              Map<String, String> reg, int romPtr, int step, String desc) {
            // Deep copy of RAM
            this.ramState = new LinkedHashMap<>();
            for (Map.Entry<String, String> entry : ram.entrySet()) {
                this.ramState.put(entry.getKey(), entry.getValue());
            }

            // Deep copy of ROM
            this.romState = new LinkedHashMap<>();
            for (Map.Entry<String, String> entry : rom.entrySet()) {
                this.romState.put(entry.getKey(), entry.getValue());
            }

            // Deep copy of registers
            this.registerState = new LinkedHashMap<>(reg);
            this.romWritePointer = romPtr;
            this.currentStep = step;
            this.description = desc;
        }
    }

    // Save current state
```

FIGURE 3.4 – Gestion d'états

```java
// Save current state
    public void saveState(Map<String, String> ram, Map<String, String> rom,
                          Map<String, String> registers, int romWritePointer,
                          int currentStep, String description) {

        // Remove future states if we're going back and then executing new steps
        if (currentStateIndex < stateHistory.size() - 1) {
            stateHistory = new ArrayList<>(stateHistory.subList(0, currentStateIndex + 1));
        }

        ExecutionState state = new ExecutionState(ram, rom, registers, romWritePointer, currentStep, description);
        stateHistory.add(state);
        currentStateIndex++;

        System.out.println("[StateManager] State saved: " + description + " (Index: " + currentStateIndex + ")");
    }

    // Check if we can go back
    public boolean canGoBack() {
        return currentStateIndex > 0;
    }

    // Check if we can go forward
    public boolean canGoForward() {
        return currentStateIndex < stateHistory.size() - 1;
    }

    // Go back one state
    public ExecutionState goBack() {
        if (!canGoBack()) {
            return null;
        }
        currentStateIndex--;
        return stateHistory.get(currentStateIndex);
    }

    // Go forward one state
    public ExecutionState goForward() {
        if (!canGoForward()) {
            return null;
        }
        currentStateIndex++;
        return stateHistory.get(currentStateIndex);
    }

    // Get current state index
    public int getCurrentStateIndex() {
        return currentStateIndex;
    }

    // Get total states
    public int getTotalStates() {
        return stateHistory.size();
    }

    // Get state description
    public String getCurrentStateDescription() {
        if (currentStateIndex >= 0 && currentStateIndex < stateHistory.size()) {
            return stateHistory.get(currentStateIndex).description;
        }
        return "Initial State";
    }

    // Clear all states
    public void clear() {
        stateHistory.clear();
        currentStateIndex = -1;
    }
}
```

FIGURE 3.5 – Gestion d'états(suite)

3.4 Diagramme de Séquence - Exécution Instruction

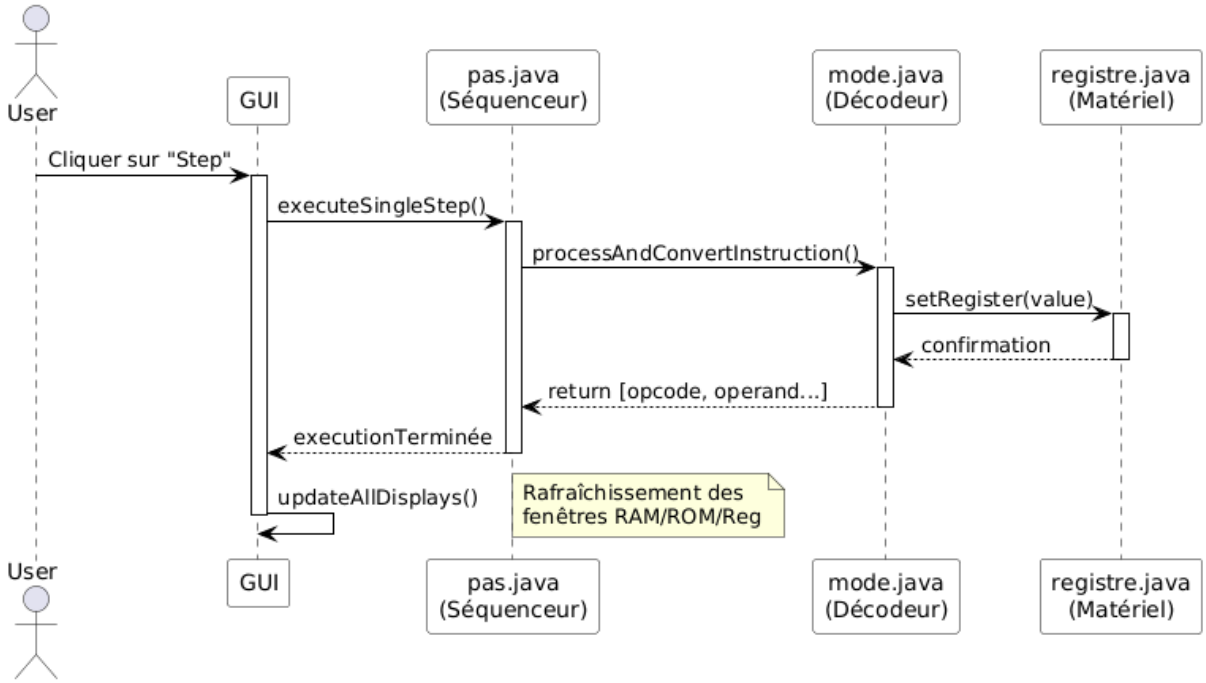

FIGURE 3.6 – Séquence d'Exécution d'une Instruction

Chapitre 4

Réalisation et Implémentation

4.1 Algorithme du step by step

```java
private void executeSingleStep(int stepIndex) {
        if (stepIndex >= assemblyLines.size())
            return;

        ArrayList<String> currentLine = assemblyLines.get(stepIndex);

        if (currentLine.size() >= 1 && !currentLine.get(0).equals("END")) {
            String firstWord = currentLine.get(0);
            String secondWord = (currentLine.size() > 1) ? currentLine.get(1) : "";

            // Execute this instruction
            String[] converted = modeDetector.processAndConvertInstruction(firstWord, secondWord, reg);
            String opcode = converted[0];
            String cleanedOperand = converted[1];

            // Store instruction in ROM
            storeInstructionInROM(firstWord, secondWord, opcode, cleanedOperand);

            // Update RAM based on mode
            updateRAMWithInstruction(firstWord, opcode, cleanedOperand);

            // Update PC
            boolean shouldSkip = modeDetector.shouldSkipPCUpdate();
            if (!shouldSkip) {
                int currentRomAddress = rom.getCurrentAddressInt();
                reg.setPC(currentRomAddress);
                modeDetector.resetSkipPCIncrementFlag();
            }

        } else if (currentLine.get(0).equals("END")) {
            // Store END instruction in ROM
            storeInstructionInROM("END", "", "3F", "");

            // Update RAM to show END opcode
            updateRAMWithInstruction("END", "3F", "");

            if (!modeDetector.shouldSkipPCUpdate()) {
                int currentRomAddress = rom.getCurrentAddressInt();
                reg.setPC(currentRomAddress - 1);
            }
            modeDetector.resetSkipPCIncrementFlag();
        }
    }

```

FIGURE 4.1 – Algorithme du step by step

4.2 Compilation Assembleur

4.2.1 Détermination du Mode d'Adressage

```java
public String determineMode(String secondWord) {

        // MINIMAL FIX: Check if string is empty before charAt(0)
        if (secondWord == null || secondWord.isEmpty()) {
            return inherent;
        }

        // PHASE 1: Détecter le mode indexé AVANT les autres modes
        if (secondWord.contains(",")) {
            return indexe; // Retourne "Indexed"
        }

        if (secondWord.charAt(0) == '#') {

            // immediat
            return immediat;

        }

        if (secondWord.charAt(0) == '>') {

            // etendu
            return etendu;

        }

        if (secondWord.charAt(0) == '<') {

            // direct
            return direct;
        }

        // PHASE 2: Détecter $XXXX (4 hex digits) comme mode Extended
        if (secondWord.startsWith("$") && secondWord.replace("$", "").length() == 4) {
            return etendu;
        }

        // Check for $XX (2 hex digits) = direct
        if (secondWord.startsWith("$") && secondWord.replace("$", "").length() == 2) {
            return direct;
        }

        return "Unknown mode";
    }
```

FIGURE 4.2 – Détermination du Mode d'Adressage

4.2.2 Traduction Mnémonique → Opcode

```java
public String[] processAndConvertInstruction(
    String instruction, String operand, registre reg) {

    String mode = determineMode(operand);
    String opcode = "00";
    String cleanedOperand = operand;

    if (instruction.equals("LDA")) {
        if (mode.equals("immediat")) {
            opcode = "86";  // LDA immediate
            cleanedOperand = operand.replace("#$", "");
            reg.setA(cleanedOperand);
            updateFlagsFor8BitLoad(
                Integer.parseInt(cleanedOperand, 16), reg);
        }
        else if (mode.equals("etendu")) {
            opcode = "B6";  // LDA extended
            String addr = operand.replace("$", "");
            int address = Integer.parseInt(addr, 16);
            String val = readFromRAM(address);
            reg.setA(val);
            updateFlagsFor8BitLoad(
                Integer.parseInt(val, 16), reg);
        }
        // ... autres modes
    }

    return new String[] {opcode, cleanedOperand};
}
```

FIGURE 4.3 – Méthode processAndConvertInstruction

4.3 Calcul d'Adresse

```
switch (effectiveType) {
            case "ZERO_OFFSET":
                effectiveAddr = baseAddr;
                break;

            case "OFFSET_5_BIT":
                effectiveAddr = baseAddr + value;
                break;

            case "AUTO_INC":
                effectiveAddr = baseAddr;
                newRegValue = baseAddr + value;
                break;

```

```
15              case "AUTO_DEC":
16                  newRegValue = baseAddr - value;
17                  effectiveAddr = newRegValue;
18                  break;
19
20              case "OFFSET_8_BIT":
21                  effectiveAddr = baseAddr + (byte) value;
22                  System.out.println("[INDEXED PHASE 4] Type=
                        OFFSET_8_BIT Reg=" + register +
23                          " Offset=$" + String.format("%02X", value)
                            +
24                          " (sign =" + (byte) value + ") => EffAddr=
                            " +
25                          String.format("%04X", effectiveAddr & 0
                            xFFFF));
26                  break;
27
28              case "OFFSET_16_BIT":
29                  effectiveAddr = baseAddr + (short) value;
30                  System.out.println("[INDEXED PHASE 4] Type=
                        OFFSET_16_BIT Reg=" + register +
31                          " Offset=$" + String.format("%04X", value)
                            +
32                          " (sign =" + (short) value + ") => EffAddr
                            =" +
33                          String.format("%04X", effectiveAddr & 0
                            xFFFF));
34                  break;
35
36              case "PC_REL_8_BIT":
37                  int currentPC8 = Integer.parseInt(reg.getPC(), 16);
38                  effectiveAddr = (currentPC8 + 3) + (byte) value;
39                  effectiveAddr = effectiveAddr & 0xFFFF;
40                  System.out.println("[INDEXED PHASE 5] Type=
                        PC_REL_8_BIT PC=" + String.format("%04X",
                        currentPC8) +
41                          " Offset=$" + String.format("%02X", value)
                            +
42                          " (sign =" + (byte) value + ") => EffAddr=
                            " +
43                          String.format("%04X", effectiveAddr));
44                  break;
45
46              case "PC_REL_16_BIT":
47                  int currentPC16 = Integer.parseInt(reg.getPC(), 16)
                        ;
48                  effectiveAddr = (currentPC16 + 4) + (short) value;
49                  effectiveAddr = effectiveAddr & 0xFFFF;
50                  System.out.println("[INDEXED PHASE 5] Type=
                        PC_REL_16_BIT PC=" + String.format("%04X",
                        currentPC16) +
51                          " Offset=$" + String.format("%04X", value)
                            +
52                          " (sign =" + (short) value + ") => EffAddr
                            =" +
53                          String.format("%04X", effectiveAddr));
54                  break;
55
```

```
56              default:
57                  return baseAddr;
58          }
```

Listing 4.1 – Calcul d'Adresse de base

4.4 Interface Graphique

4.4.1 Architecture GUI

L'interface utilise Swing avec :

— **Main Frame** : Fenêtre principale avec toolbar
— **RAMDisplayFrame** : Vue de la RAM avec JTable éditable
— **ROMDisplayFrame** : Vue de la ROM
— **RegisterDisplayFrame** : Vue des registres
— **StepExecutionFrame** : Débogueur pas-à-pas

4.4.2 Éditeur de Cellules Hexadécimales

```java
class HexCellEditor extends DefaultCellEditor {
        private JTextField textField;

        public HexCellEditor() {
            super(new JTextField());
            textField = (JTextField) getComponent();
            textField.setHorizontalAlignment(JTextField.CENTER);
        }

        @Override
        public boolean stopCellEditing() {
            String value = textField.getText();
            if (isValidHex(value)) {
                return super.stopCellEditing();
            } else {
                textField.setBorder(BorderFactory.createLineBorder(Color.RED));
                return false;
            }
        }

        @Override
        public Component getTableCellEditorComponent(JTable table, Object value,
                                                     boolean isSelected, int row, int column) {
            textField.setBorder(BorderFactory.createLineBorder(Color.BLUE, 2));
            return super.getTableCellEditorComponent(table, value, isSelected, row, column);
        }
    }
```

FIGURE 4.4 – Classe HexCellEditor

4.4.3 Rafraîchissement Automatique

```
updateTimer = new Timer(500, e -> {
    SwingUtilities.invokeLater(() -> {
        aField.setText(registers.getA());
        bField.setText(registers.getB());
        // ... autres registres
        updateFlagDisplay();
    });
});
updateTimer.start();
```

Listing 4.2 – Timer pour Mise à Jour Continue

4.5 Gestion de l'Historique

4.5.1 Sauvegarde d'État

```
private void saveCurrentState(String description) {
        Map<String, String> registers = new LinkedHashMap<>();
        registers.put("A", reg.getA());
        registers.put("B", reg.getB());
        registers.put("D", reg.getD());
        registers.put("X", reg.getX());
        registers.put("Y", reg.getY());
        registers.put("S", reg.getS());
        registers.put("U", reg.getU());
        registers.put("CCR", reg.getCCR());
        registers.put("PC", reg.getPC());

        int romPtr = rom.getCurrentAddressInt();
        stateManager.saveState(ram.getram(), rom.getMemory(), registers, romPtr, currentStep, description);
    }
```

FIGURE 4.5 – Méthode de sauvegarde d'État

4.5.2 Navigation Arrière

```java
public boolean goBackOneStep() {
    StateManager.ExecutionState previousState = stateManager.goBack();
    if (previousState == null) {
        return false;
    }

    // Restore RAM
    ram.getram().clear();
    ram.getram().putAll(previousState.ramState);

    // Restore ROM
    Map<String, String> romMemory = rom.getMemory();
    romMemory.clear();
    romMemory.putAll(previousState.romState);

    // Restore ROM write pointer
    rom.setWritePointer(previousState.romWritePointer);

    // Restore registers - INCLUDING PC!
    reg.setA(previousState.registerState.get("A"));
    reg.setB(previousState.registerState.get("B"));
    reg.setD(previousState.registerState.get("D"));
    reg.setX(previousState.registerState.get("X"));
    reg.setY(previousState.registerState.get("Y"));
    reg.setS(previousState.registerState.get("S"));
    reg.setU(previousState.registerState.get("U"));
    reg.setCCR(previousState.registerState.get("CCR"));

    // CRITICAL FIX: Restore PC properly
    String pcValue = previousState.registerState.get("PC");
    try {
        int pcInt = Integer.parseInt(pcValue, 16);
        reg.setPC(pcInt);
    } catch (Exception e) {
        reg.setPC(0);
    }

    // Restore current step
    currentStep = previousState.currentStep;

    System.out.println("[GUI] Went back to step " + currentStep + " (PC=" + reg.getPC() + ")");
    return true;
}
```

FIGURE 4.6 – Méthode de navigation Arrière

Chapitre 5

Tests et Validation

5.1 Stratégie de Tests

5.1.1 Types de Tests

TABLE 5.1 – Stratégie de Tests

Type	Objectif	Couverture
Unitaires	Test méthodes individuelles	100%
Intégration	Test interaction composants	100%
Fonctionnels	Test cas d'utilisation	100%
Système	Test application complète	90%

5.2 Programmes de Test

5.2.1 Test 1 : Instructions de Base

```
LDA #$FF     ; Charger FF dans A
LDB #$01     ; Charger 01 dans B
ADDA #$01    ; A = A + 1 = 00 (overflow)
SUBA #$10    ; A = A - 10 = F0
STA $2000    ; Stocker A en memoire
END
```

Listing 5.1 – Programme Test Basique

Résultats attendus :

— A = F0
— RAM[2000] = F0
— Flags Z=0, N=1, V=0, C=1

5.2.2 Test 2 : Mode Indexé

```
LDX #$3000    ; X = 3000
LDA #$AA      ; A = AA
STA ,X        ; RAM[3000] = AA
```

```
STA 1,X       ; RAM[3001] = AA
LDB ,X+       ; B = AA, X = 3001
LDB ,X++      ; B = AA, X = 3003
END
```

Listing 5.2 – Test Adressage Indexé

Résultats attendus :
— X = 3003
— RAM[3000] = AA
— RAM[3001] = AA
— B = AA

5.3 Validation des Flags

TABLE 5.2 – Tests de Validation des Flags CCR

Opération	Résultat	Z	N	V	C
LDA #$00	A=00	1	0	0	0
LDA #$FF	A=FF	0	1	0	0
ADDA #$01 (A=FF)	A=00	1	0	0	1
SUBA #$01 (A=00)	A=FF	0	1	0	1

5.4 Tests d'Interface

5.4.1 Tests Manuels GUI

1. **Chargement code** : Vérifier validation syntaxique
2. **Exécution** : Tester Run et Step
3. **Visualisation** : Vérifier mise à jour temps réel
4. **Édition** : Modifier registres/mémoire manuellement
5. **Navigation** : Tester Back/Forward
6. **Sauvegarde** : Exporter fichier .asm

5.4.2 Tests de Robustesse

TABLE 5.3 – Tests de Robustesse

Scénario	Comportement	Résultat
Code invalide	Message d'erreur clair	✓
Opérande manquant	Erreur de validation	✓
Édition hex invalide	Bordure rouge, refus	✓
Back sans historique	Bouton désactivé	✓
Mémoire pleine	Gestion gracieuse	✓

5.5 Performance

TABLE 5.4 – Mesures de Performance

Opération	**Temps moyen**
Exécution 1 instruction	< 5 ms
Compilation 100 lignes	< 50 ms
Mise à jour GUI	16 ms (60 fps)
Sauvegarde état	< 10 ms
Chargement fichier	< 100 ms

Chapitre 6

Manuel Utilisateur

6.1 Installation

6.1.1 Prérequis

— Java JDK 8 ou supérieur
— Système : Windows, Linux ou macOS

6.1.2 Lancement

```
javac processeur/*.java
java processeur.main
```

6.2 Guide d'Utilisation

6.2.1 Étape 1 : Écrire du Code

1. Cliquez sur **"New"**
2. L'éditeur de code s'ouvre
3. Saisissez vos instructions assembleur
4. Terminez par `END`
5. Cliquez **"Load Code"**

Exemple de code valide :

```
LDAA #$FF
STAA $2000
LDX #$3000
DECA
END
```

6.2.2 Étape 2 : Exécuter

Deux modes d'exécution :
Mode Run (Exécution Complète) :

— Cliquez **"Run"**
— Toutes les instructions s'exécutent
— Résultat final affiché

Mode Step (Pas-à-Pas) :
— Cliquez **"Step"**
— Fenêtre de débogage s'ouvre
— Utilisez **"Next"** pour avancer
— Utilisez **"Back"** pour reculer
— **"Run Rest"** pour finir rapidement

6.2.3 Étape 3 : Visualiser

Visualiser les Registres :
— Cliquez **"Registers"**
— Double-cliquez une valeur pour modifier
— Flags CCR affichés en binaire

Visualiser la RAM :
— Cliquez **"RAM"**
— Sélectionnez une plage d'adresses (0000-0FFF, 1000-1FFF, etc.)
— Double-cliquez une valeur pour la modifier
— Format : 2 chiffres hexadécimaux (00-FF)

Visualiser la ROM :
— Cliquez **"ROM"**
— Consultez les instructions compilées
— Double-cliquez pour modifier si nécessaire
— Cliquez **"Refresh"** pour actualiser

6.2.4 Étape 4 : Déboguer

Navigation dans l'Exécution :

1. Ouvrez le débogueur avec **"Step"**
2. La ligne courante est marquée »>
3. Les lignes exécutées sont marquées []
4. Utilisez les boutons :
   — **Next** → : Exécuter ligne suivante
   — ← **Back** : Revenir en arrière
   — **Run Rest** : Finir l'exécution

6.2.5 Étape 5 : Sauvegarder

— Cliquez **" Save"**
— Choisissez l'emplacement
— Le fichier .asm contient votre code avec métadonnées

6.3 Messages d'Erreur

TABLE 6.1 – Messages d'Erreur Courants

Message	**Solution**
"Instruction inconnue 'XXX'"	Vérifiez l'orthographe, utilisez MAJUSCULES
"Mode d'adressage invalide"	Vérifiez la syntaxe de l'opérande (#$, $, ,X)
"L'instruction ne supporte pas le mode IMMÉDIAT"	Certaines instructions (STA, STB) n'acceptent pas #$
"Valeur hexadécimale invalide"	Utilisez uniquement 0-9, A-F
"Code vide"	Entrez au moins une instruction avant END

6.4 Exemples de Programmes

6.4.1 Programme 1 : Manipulation de Registres et Transferts

```
; Manipulation de registres et m moire
LDA #$15 ; Charger valeur dans A
LDB #$20 ; Charger valeur dans B
STB $1000 ; Stocker B en m moire
ADDA $1000 ; A = A + valeur    l'adresse $1000
STA $1001 ; Stocker r sultat
TFR A,B ; Transf rer A vers B
CLRA ; Effacer A
INCB ; Incr menter B
END
```

Listing 6.1 – Manipulation de registres

6.4.2 Programme 2 : Opérations Arithmétiques

```
; Diverses operations arithmetiques
LDA #$30 ; A = 48
LDB #$10 ; B = 16
STD $3000 ; Stocker D (A:B) en $3000-$3001
SUBB #$05 ; B = B - 5
ADDA $3001 ; A = A + ancienne valeur de B
MUL ; D = A    B
STD $3002 ; Stocker resultat multiplication
LDA #$FF ; Charger valeur maximale
INCA ; A = 0 (avec retenue)
CLRB ; B = 0
ADDD $3002 ; D = D + valeur en $3002-$3003
STD $3004 ; Stocker resultat final
END
```

Listing 6.2 – Opérations arithmétiques

6.5 Raccourcis et Astuces

TABLE 6.2 – Conseils d'Utilisation

Conseil	**Description**
Commentaires	Utilisez ; pour documenter votre code
Majuscules	Les instructions doivent être en MAJUSCULES
END obligatoire	Terminez toujours par `END`
Mode immédiat	Préfixe `#$` pour valeurs constantes
Adresses	`$XX` = direct, `$XXXX` = étendu
Édition directe	Double-cliquez pour modifier registres/mémoire
Sauvegarde fréquente	Utilisez Save régulièrement

Chapitre 7

Difficultés Rencontrées et Solutions

7.1 Difficultés Techniques

7.1.1 Défi 1 : Gestion du Program Counter

Problème : Le PC devait être incrémenté différemment selon la taille de l'instruction :
— 1 octet : opcode seul (END)
— 2 octets : opcode + opérande 8 bits
— 3 octets : opcode + opérande 16 bits
— 4 octets : préfixe + opcode + opérande 16 bits
Solution Adoptée :

```
int instructionSize = 1; // Au minimum l'opcode

if (!opcode.equals("00")) {
    if (opcode.startsWith("10")) {
        // Prefixe page 2 (LDY, STY, etc.)
        instructionSize = 2;
        if (cleanedOperand.length() == 4) {
            instructionSize += 2; // Total: 4 bytes
        }
    } else {
        // Instructions normales
        int operandLength = cleanedOperand.length() / 2;
        instructionSize += operandLength;
    }

    int currentPC = Integer.parseInt(reg.getPC(), 16);
    int newPC = (currentPC + instructionSize) & 0xFFFF;
    reg.setPC(newPC);
}
```

Listing 7.1 – Calcul Dynamique de la Taille d'Instruction

7.1.2 Défi 2 : Mode Indexé avec Auto-Inc/Dec

Problème : Différencier post-incrémentation (`,X+`) et pré-décrémentation (`,-X`) :
— Post-inc : utiliser l'adresse **avant** modification
— Pré-dec : utiliser l'adresse **après** modification
Solution :

```
switch (type) {
    case "AUTO_INC":
        effectiveAddr = baseAddr;       // AVANT
        newRegValue = baseAddr + value; // Modifier a la fin
        break;

    case "AUTO_DEC":
        newRegValue = baseAddr - value; // Modifier avant
        effectiveAddr = newRegValue;    // a la fin
        break;
}

// Mise    jour du registre
reg.setX(String.format("%04X", newRegValue & 0xFFFF));
```

Listing 7.2 – Gestion Correcte des Effets de Bord

7.1.3 Défi 3 : Navigation Arrière avec Restauration Complète

Problème Initial : La navigation arrière ne restaurait pas correctement le PC, causant des incohérences.

Cause : Le PC était stocké en String mais pas correctement restauré lors du `goBack()`.

Solution Finale :

```
public boolean goBackOneStep() {
    ExecutionState prevState = stateManager.goBack();
    if (prevState == null) return false;

    // Restore ALL registers INCLUDING PC
    String pcValue = prevState.registerState.get("PC");
    try {
        int pcInt = Integer.parseInt(pcValue, 16);
        reg.setPC(pcInt); // CRITIQUE: Restaurer PC!
    } catch (Exception e) {
        reg.setPC(0);
    }

    // Restore RAM, ROM, autres registres...
    return true;
}
```

Listing 7.3 – Restauration Complète de l'État

7.1.4 Défi 4 : Validation Syntaxique en Temps Réel

Problème : Fournir des messages d'erreur clairs et constructifs.

Solution : Classe `InstructionValidator` avec suggestions :

```
public static String validateInstruction(
    String instruction, String operand) {

    // Verifier majuscules
    if (!instruction.equals(instruction.toUpperCase())) {
        return "ERREUR:L'instruction'" + instruction +
               "'doit etre en MAJUSCULES.\n" +
```

```
                "Correction suggeree:'" +
                instruction.toUpperCase() + "'";
    }

    // V rifier existence
    if (!VALID_INSTRUCTIONS.contains(instruction)) {
        String suggestion = findSimilarInstruction(instruction);
        if (suggestion != null) {
            return "ERREUR:Instruction inconnue'" +
                   instruction + "'.\n" +
                   "Vouliez-vous dire'" + suggestion + "'?";
        }
    }

    // V rifier mode d'adressage support ...
}
```

Listing 7.4 – Validation avec Suggestions

7.2 Difficultés de Conception

7.2.1 Choix Architectural

Dilemme : Centraliser la logique dans une classe unique vs. distribuer dans plusieurs classes ?
Décision : Architecture modulaire avec :
— `mode.java` : Logique d'exécution et traduction
— `pas.java` : Orchestration et gestion d'historique
— `registre.java, ram.java, ROM.java` : Modèles de données
— `*DisplayFrame.java` : Vues GUI indépendantes
Avantages :
— Séparation des responsabilités
— Testabilité accrue
— Maintenance facilitée

7.2.2 Gestion de l'État

Problématique : Comment sauvegarder efficacement 65536 octets de RAM à chaque étape ?
Solution Retenue : Deep copy complet malgré le coût mémoire :

```
public ExecutionState(Map<String, String> ram, ...) {
    // Deep copy de RAM
    this.ramState = new LinkedHashMap<>();
    for (Map.Entry<String, String> entry : ram.entrySet()) {
        this.ramState.put(entry.getKey(), entry.getValue());
    }
    // Idem pour ROM, registres...
}
```

Listing 7.5 – Deep Copy pour Historique

Alternative Envisagée (Non Retenue) : Sauvegarder uniquement les différences (delta), mais complexité élevée.

7.3 Difficultés d'Interface

7.3.1 Synchronisation GUI Backend

Problème : Éviter les conflits entre édition manuelle et mise à jour automatique.
Solution : Vérifier le focus avant mise à jour :

```
public void updateRegisterDisplay() {
    SwingUtilities.invokeLater(() -> {
        // N' craser que si le champ n'a pas le focus
        if (!aField.hasFocus()) {
            aField.setText(registers.getA());
        }
        if (!bField.hasFocus()) {
            bField.setText(registers.getB());
        }
        // ... autres registres
    });
}
```

Listing 7.6 – Mise à Jour Conditionnelle

7.3.2 Performance de la JTable (RAM/ROM)

Problème Initial : Afficher 65536 lignes gelait l'interface.
Solution : Pagination avec `JComboBox` (plages de 4096 octets) :

```
String[] ranges = new String[16];
for (int i = 0; i < 16; i++) {
    int start = i * 4096;
    int end = start + 4095;
    ranges[i] = String.format("%04X-%04X", start, end);
}
rangeSelector = new JComboBox<>(ranges);
rangeSelector.addActionListener(e -> updateRAMDisplay());
```

Listing 7.7 – Pagination de la Mémoire

7.4 Leçons Apprises

1. **Documentation Continue :** Commenter le code au fur et à mesure évite la dette technique
2. **Tests Unitaires Précoces :** Tester chaque instruction individuellement accélère le débogage
3. **Gestion d'État Robuste :** Anticiper les besoins de navigation (undo/redo) dès la conception
4. **Validation Stricte :** Fournir des messages d'erreur clairs améliore l'expérience utilisateur
5. **Modularité :** Séparer GUI et logique métier facilite les évolutions futures

7.5 Améliorations Futures

— **Points d'Arrêt (Breakpoints)** : Pause automatique à certaines lignes
— **Watch Variables** : Surveillance de registres/adresses spécifiques
— **Export Trace** : Sauvegarde de l'historique d'exécution en fichier
— **Mode Graphique Avancé** : Visualisation du chemin d'exécution (flowchart)
— **Profileur** : Analyse de performance (cycles, instructions les plus utilisées)
— **Interruptions Matérielles** : Simulation IRQ/FIRQ/NMI
— **Périphériques Virtuels** : Clavier, écran, ports série

Conclusion Générale

Synthèse du Projet

Ce projet de développement d'un simulateur pour le microprocesseur Motorola 6809 a permis d'atteindre les objectifs fixés au départ. Nous avons réussi à implémenter un environnement complet permettant de :

- Écrire et compiler du code assembleur 6809
- Exécuter des programmes avec fidélité au comportement matériel
- Visualiser en temps réel l'état du système (registres, mémoire, flags)
- Déboguer efficacement avec navigation bidirectionnelle
- Éditer interactivement les registres et la mémoire

Le simulateur supporte **l'intégralité du jeu d'instructions** du 6809, incluant tous les modes d'adressage (inhérent, immédiat, direct, étendu, indexé avec ses 16 variantes). La validation syntaxique en temps réel et les messages d'erreur explicites facilitent l'apprentissage.

Apports Pédagogiques

Sur le plan pédagogique, ce projet a considérablement enrichi nos connaissances :

Architecture des Ordinateurs :

- Compréhension approfondie du cycle d'exécution fetch-decode-execute
- Maîtrise des modes d'adressage et de leur impact sur le code machine
- Gestion des flags de condition (CCR) et leur rôle dans les branchements
- Fonctionnement des piles matérielles (S et U)

Programmation Assembleur :

- Pratique intensive du langage assembleur 6809
- Compréhension des instructions arithmétiques, logiques, de transfert
- Utilisation des branchements conditionnels et boucles
- Optimisation au niveau instruction

Génie Logiciel :

- Application des principes de conception orientée objet
- Utilisation de design patterns (Singleton, Observer, State)
- Développement d'une architecture modulaire et maintenable
- Pratique des tests et validation

Compétences Techniques Acquises

Java et Swing :
— Développement d'interfaces graphiques complexes
— Gestion d'événements et mise à jour asynchrone
— Utilisation de JTable avec éditeurs personnalisés
— Gestion de la concurrence (SwingUtilities, Timer)
Algorithmique :
— Parsing et analyse syntaxique
— Calcul d'adresses effectives en mode indexé
— Gestion d'historique avec navigation bidirectionnelle
— Optimisation des structures de données (LinkedHashMap)
Long Terme :
— Simulation de périphériques (clavier, écran)
— Mode réseau pour communication inter-processeurs
— Intégration d'un assembleur macro

Impact et Utilité

Ce simulateur peut servir d'outil pédagogique pour :
— Les étudiants découvrant l'architecture des ordinateurs
— Les enseignants illustrant les concepts de microprocesseurs
— Les passionnés d'informatique rétro étudiant les systèmes 8 bits
Il démontre qu'un simulateur complet et ergonomique peut être développé avec des outils modernes (Java/Swing) tout en restant fidèle aux spécifications matérielles historiques.

Mot de Fin

En conclusion, ce projet nous a permis de mettre en pratique les connaissances théoriques acquises en cours d'Architecture des Ordinateurs. Au-delà de l'aspect technique, il nous a appris l'importance de :
— La rigueur dans la conception et l'implémentation
— La documentation continue du code
— Les tests systématiques
— L'attention portée à l'expérience utilisateur
— Le travail d'équipe et la communication
Nous espérons que ce rapport et le simulateur développé contribueront à faciliter l'apprentissage de l'architecture des microprocesseurs pour les futures promotions.

Bibliographie

[1] **BENALLA Hicham**, *Support de Cours - Architecture des Ordinateurs - Partie 1, 2, 3*, FST Settat.

[2] **Motorola Inc.**, *MC6809 8-Bit Microprocessor Datasheet*, 1981. http://www.6809.org.uk/dragon/mc6809.pdf

[3] **Leventhal, Lance A.**, *6809 Assembly Language Programming*, Osborne/McGraw-Hill, 1981.

[4] **Zaks, Rodnay**, *Programming the 6809*, Sybex, 1982.

[5] **Williams, Gregg**, *The Motorola 6809 : A Programmer's View*, BYTE Magazine, 1979.

[6] **Oracle Corporation**, *Java SE Documentation*, 2024. https://docs.oracle.com/en/java/

[7] **Oracle Corporation**, *Swing Tutorial*, 2024. https://docs.oracle.com/javase/tutorial/uiswing/

[8] **Gamma, Erich et al.**, *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[9] **The 6809 Resource**, *6809.org.uk - Documentation et Ressources*, 2024. http://www.6809.org.uk/

[10] **The LaTeX Project**, *LaTeX Documentation*, 2024. https://www.latex-project.org/

[11] **DeepSeek**, *Pour les recherches*. https://chat.deepseek.com/

[12] **Google ai studio**, *Pour les recherches*. https://aistudio.google.com/

Annexe A

Tables de Référence

A.1 Opcodes du Motorola 6809

TABLE A.1 – Table des Opcodes (Sélection Étendue)

Mnémonique	Mode	Opcode	Cycles	Flags
CHARGEMENT 8-BITS				
LDA	Immédiat	86	2	Z,N,V
LDA	Direct	96	3	Z,N,V
LDA	Étendu	B6	5	Z,N,V
LDA	Indexé	A6	4-7	Z,N,V
LDB	Immédiat	C6	2	Z,N,V
LDB	Direct	D6	3	Z,N,V
LDB	Étendu	F6	5	Z,N,V
LDB	Indexé	E6	4-7	Z,N,V
CHARGEMENT 16-BITS				
LDD	Immédiat	CC	3	Z,N,V
LDD	Direct	DC	4	Z,N,V
LDD	Étendu	FC	6	Z,N,V
LDD	Indexé	EC	6-9	Z,N,V
LDX	Immédiat	8E	3	Z,N,V
LDX	Direct	9E	4	Z,N,V
LDX	Étendu	FE	6	Z,N,V
LDX	Indexé	AE	6-9	Z,N,V
LDY	Immédiat	10 8E	4	Z,N,V
LDY	Direct	10 9E	5	Z,N,V
LDY	Étendu	10 FE	7	Z,N,V
LDY	Indexé	10 AE	7-10	Z,N,V
LDU	Immédiat	CE	3	Z,N,V
LDU	Direct	DE	4	Z,N,V
LDU	Étendu	EE	6	Z,N,V
LDU	Indexé		6-9	Z,N,V
STOCKAGE 8-BITS				
STA	Direct	97	4	Z,N,V

Mnémonique	Mode	Opcode	Cycles	Flags
STA	Étendu	B7	5	Z,N,V
STA	Indexé	A7	5-8	Z,N,V
STB	Direct	D7	4	Z,N,V
STB	Étendu	F7	5	Z,N,V
STB	Indexé	E7	5-8	Z,N,V
STOCKAGE 16-BITS				
STD	Direct	DD	5	Z,N,V
STD	Étendu	FD	6	Z,N,V
STD	Indexé	ED	7-10	Z,N,V
STX	Direct	9F	5	Z,N,V
STX	Étendu	BF	6	Z,N,V
STX	Indexé	AF	6-9	Z,N,V
ARITHMÉTIQUE				
ADDA	Immédiat	8B	2	Z,N,V,C
ADDA	Direct	9B	3	Z,N,V,C
ADDA	Étendu	BB	4	Z,N,V,C
ADDA	Indexé	AB	4-7	Z,N,V,C
SUBA	Immédiat	80	2	Z,N,V,C
SUBA	Direct	90	3	Z,N,V,C
SUBA	Étendu	B0	4	Z,N,V,C
SUBA	Indexé	A0	4-7	Z,N,V,C
CMPA	Immédiat	81	2	Z,N,V,C
CMPA	Direct	91	3	Z,N,V,C
CMPA	Étendu	B1	4	Z,N,V,C
CMPA	Indexé	A1	4-7	Z,N,V,C
LOGIQUE				
ANDA	Immédiat	84	2	Z,N,V
ANDA	Direct	94	3	Z,N,V
ANDA	Étendu	B4	4	Z,N,V
ANDA	Indexé	A4	4-7	Z,N,V
ORA	Immédiat	8A	2	Z,N,V
ORA	Direct	9A	3	Z,N,V
ORA	Étendu	BA	4	Z,N,V
ORA	Indexé	AA	4-7	Z,N,V
SAUT ET SOUS-ROUTINE				
JMP	Direct	0E	3	-
JMP	Étendu	7E	3	-
JMP	Indexé	6E	3	-
JSR	Direct	9D	7	-
JSR	Étendu	BD	8	-
JSR	Indexé	AD	7	-
RTS	Inhérent	39	5	-
BRANCHEMENT				
BEQ	Relatif	27	3	-
BNE	Relatif	26	3	-
BRA	Relatif	20	3	-

Mnémonique	Mode	Opcode	Cycles	Flags
BCC	Relatif	24	3	-
BCS	Relatif	25	3	-
INHÉRENT				
NOP	Inhérent	12	2	-
ABX	Inhérent	3A	3	-
CLRA	Inhérent	4F	2	Z,N,V
INCA	Inhérent	4C	2	Z,N,V
DECA	Inhérent	4A	2	Z,N,V

A.2 Post-Bytes du Mode Indexé

TABLE A.2 – Post-Bytes du Mode Indexé

Syntaxe	Description	Post-Byte
,R	Zéro offset	1RR00100
5,R	Offset 5-bits	0RRnnnnn
,R+	Post-inc +1	1RR00000
,R++	Post-inc +2	1RR00001
,-R	Pré-dec -1	1RR00010
,–R	Pré-dec -2	1RR00011
A,R	Acc A offset	1RR00110
B,R	Acc B offset	1RR00101
D,R	Acc D offset	1RR01011
n8,R	Offset 8-bits	1RR01000
n16,R	Offset 16-bits	1RR01001
[,R]	Indirect	1RR10100

Note : RR = 00 (X), 01 (Y), 10 (U), 11 (S)

Annexe B

Captures d'Écran

B.1 Fenêtre Principale

FIGURE B.1 – Interface Principale du Simulateur

B.2 Éditeur de Code

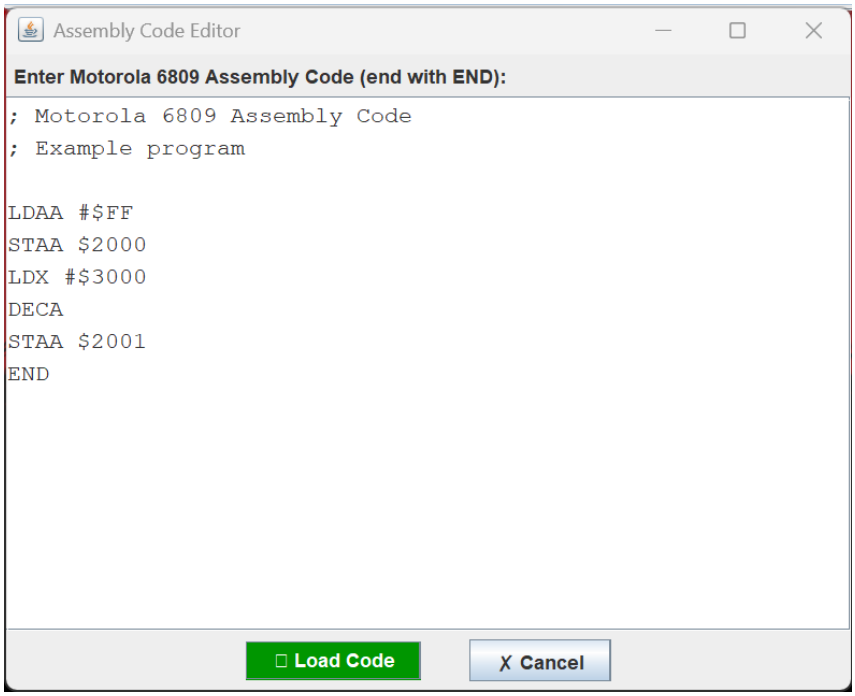

FIGURE B.2 – Éditeur de Code Assembleur

B.3 Débogueur Pas-à-Pas

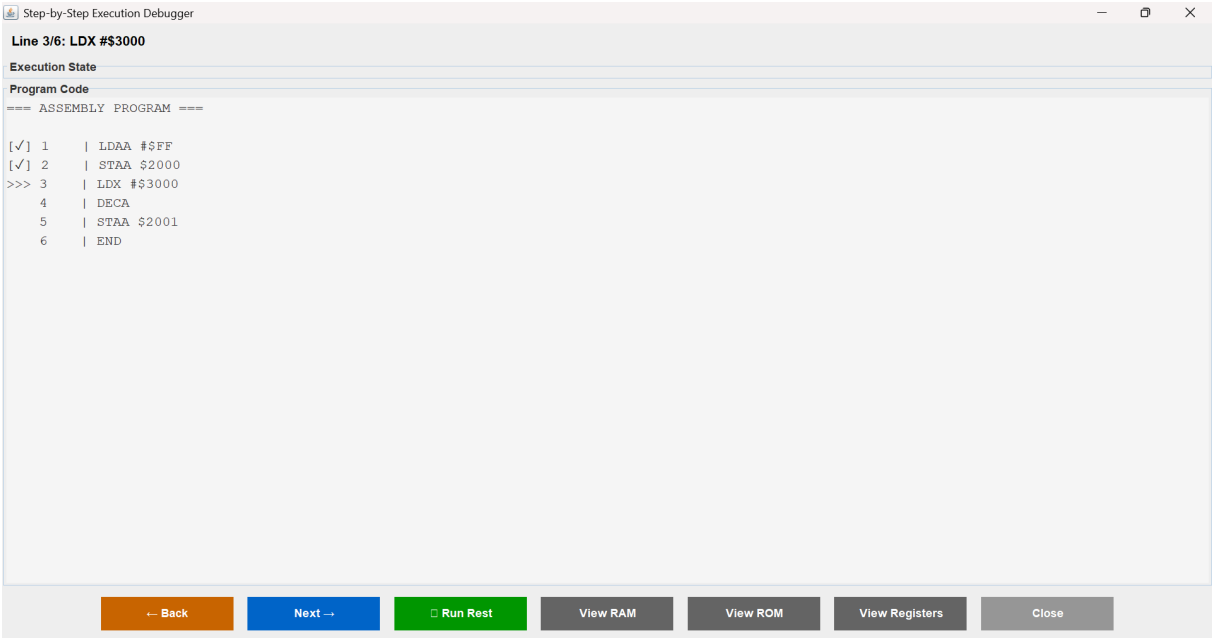

FIGURE B.3 – Mode Débogage Step-by-Step

B.4 Visualiseur de Registres

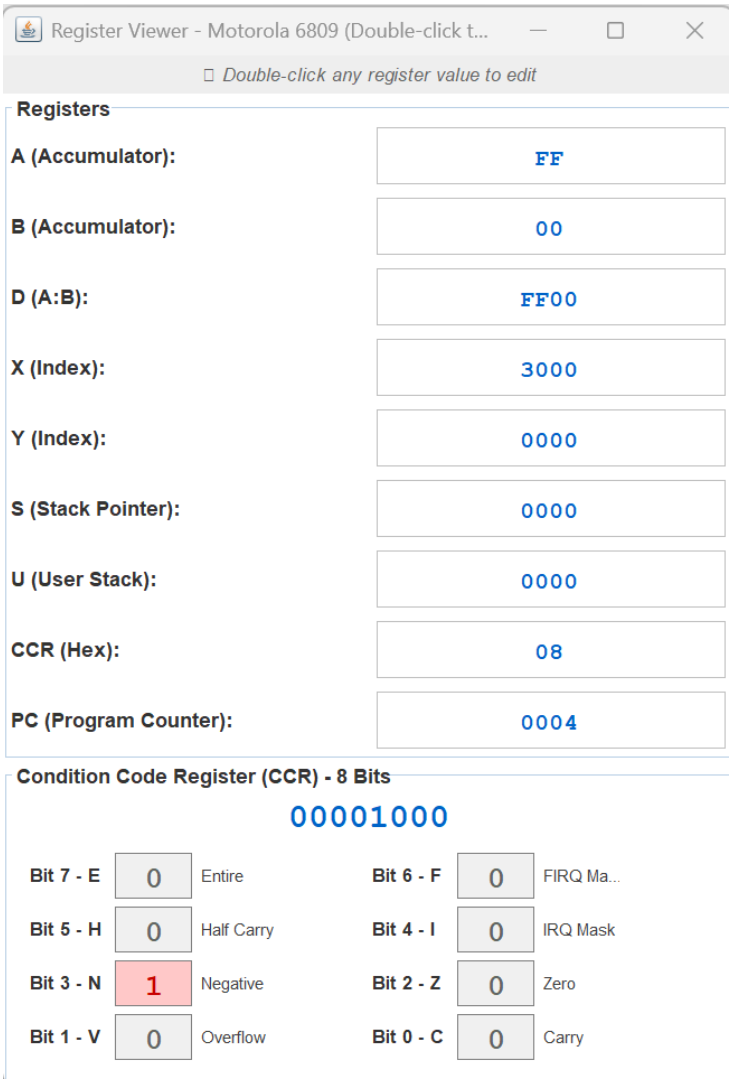

FIGURE B.4 – Visualiseur de Registres

Annexe C

Instructions d'Installation Détaillées

C.1 Installation sur Windows

1. Télécharger Java JDK 8+ depuis https://www.oracle.com/java/
2. Installer le JDK en suivant l'assistant
3. Ajouter Java au PATH :
   — Panneau de configuration → Système → Variables d'environnement
   — Ajouter `C:\Program Files\Java\jdk-XX\bin` au PATH
4. Ouvrir l'invite de commandes
5. Naviguer vers le dossier du projet
6. Compiler : `javac processeur/*.java`
7. Exécuter : `java processeur.main`

C.2 Installation sur Linux/macOS

1. Installer Java via le gestionnaire de paquets :
   — Ubuntu/Debian : `sudo apt install default-jdk`
   — macOS : `brew install openjdk`
2. Ouvrir un terminal
3. Naviguer vers le dossier du projet : `cd /chemin/vers/projet`
4. Compiler : `javac processeur/*.java`
5. Exécuter : `java processeur.main`

C.3 Dépendances

Le projet utilise uniquement :
— Java Standard Edition (SE) 8+
— Bibliothèque Swing (incluse dans Java SE)
Aucune dépendance externe n'est requise.

Annexe D

FAQ - Questions Fréquentes

D.1 Questions Techniques

Q : Puis-je utiliser des lettres minuscules pour les instructions ?
R : Non, les instructions doivent être en MAJUSCULES. Le validateur rejettera `lda` et suggérera `LDA`.

Q : Comment entrer une valeur hexadécimale ?
R : Utilisez le préfixe `$`. Exemple : `$FF`, `$2000`.

Q : Puis-je modifier les registres pendant l'exécution ?
R : Oui ! Double-cliquez sur une valeur de registre dans le visualiseur pour la modifier.

Q : Comment revenir en arrière dans l'exécution ?
R : Ouvrez le débogueur (bouton **Step**), puis utilisez le bouton ← **Back**.

Q : Y a-t-il une limite au nombre de lignes de code ?
R : Théoriquement non, mais le simulateur est optimisé pour des programmes de 100-200 lignes.

D.2 Questions Pédagogiques

Q : Quels sont les meilleurs programmes pour débuter ?
R : Commencez par des programmes simples :
— Charger une valeur : `LDA #$FF`
— Stocker en mémoire : `STA $2000`
— Arithmétique basique : `ADDA #$01`

Q : Comment comprendre le CCR ?
R : Observez les flags lors de l'exécution :
— Flag Z = 1 si résultat = 0
— Flag N = 1 si résultat négatif (bit 7 = 1)
— Flag C = 1 si retenue
— Flag V = 1 si débordement arithmétique