Projet: Réalisation d’un simulateur de CPU

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Groupe** : AKKA Boutaina && AYED Yasmine

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## **Contexte et objectifs**

Ce projet vise à développer un simulateur de processeur (CPU) simplifié en langage C, intégrant :

* Un **gestionnaire de mémoire** dynamique (allocation/libération de segments
* Un **parser** pour un pseudo-assembleur (.DATA/.CODE)
* Des **modes d’adressage** (immédiat, registre, direct, indirect, override segment)
* La **gestion d’un segment de données** (“DS”), de code (“CS”), de pile (“SS”) et d’un **extra segment** (“ES”)

Le simulateur supporte également l’exécution pas à pas des instructions, la gestion des registres généraux et de drapeaux (ZF, SF, IP) ainsi que les opérations arithmétiques et logiques fondamentales.

## 

## **📂 Structure du projet**

.

├── include/ # Fichiers d’en-tête (.h)

│ ├── hashmap.h

│ ├── memory\_handler.h

│ ├── assembler\_parser.h

│ ├── parser\_result.h

│ ├── cpu.h

│ ├── code\_segment.h

│ ├── stack\_segment.h

│ └── extra\_segment.h

├── src/ # Code source (.c)

│ ├── hashmap.c

│ ├── memory\_handler.c

│ ├── assembler\_parser.c

│ ├── parser\_result.c

│ ├── cpu.c

│ ├── code\_segment.c

│ ├── stack\_segment.c

│ └── extract\_segment.c

├── tests/ # Tests unitaires

│ ├── test\_memory.c

│ ├── test\_parser.c

│ ├── test\_cpu.c

│ └── test\_assembler\_parser.c

├── obj/ # Objets compilés (.o)

├── bin/ # Binaries des tests

└── Makefile # Automatisation de la compilation

## **Exercice 1 – Table de hachage générique**

* **simple\_hash()** : fonction de hachage pour chaînes (
* **hashmap\_create() / destroy()** : allocation et nettoyage de la table
* **hashmap\_insert(), remove() ...** : insertion, recherche, suppression avec probing linéaire et TOMBSTONE
* **Tests** : couverture complète des collisions, suppressions et rebouclages.

## **Exercice 2 – Gestion dynamique de la mémoire**

* **MemoryHandler** : structure avec memory[], liste chaînée free\_list et allocated (HashMap)
* **memory\_init()** : init d’un unique segment libre couvrant toute la RAM
* **find\_free\_segment()** : recherche d’un segment libre adapté citeturn1file3
* **create\_segment() / remove\_segment()** : allocation et libération avec split/fusion des segments
* **Tests** : création/suppression multiple, fragmentation et compaction.

## **Exercice 3 – Parser pseudo-assembleur**

* **Instruction** : structure à 3 champs (mnemonic, op1, op2)
* **parse\_data\_instruction()** : découpage des lignes .DATA, calcul séquentiel des adresses
* **parse\_code\_instruction()** : gestion des labels et extraction des opérandes
* **parse() / free\_parser\_result()** : lecture séquentielle du fichier, redimensionnement dynamique des tableaux, remplissage des HashMaps labels et memory\_locations
* **Tests** : parsing des sections .DATA/.CODE, vérification des tables et des adresses.

## **Exercice 4 – Allocation du segment de données (“DS”)**

* **cpu\_init() / cpu\_destroy()** : initialisation du CPU, création de la HashMap context avec AX, BX, CX, DX à 0 citeturn1file7
* **store() / load()** : stocker et récupérer un pointeur dans “DS” en vérifiant bornes et existence de segment
* **allocate\_variables()** : calcul de la taille totale, création de “DS”, remplissage avec valeurs .DATA
* **print\_data\_segment()** : affichage formaté des données en mémoire
* **Tests** : store/load, allocation dynamique et affichage.

## **Exercice 5 – Modes d’adressage & MOV**

* **matches()** : wrapper regex ---------------- (fournie)
* **immediate\_addressing()** : pool de constantes dans constant\_pool (HashMap)
* **register\_addressing() / direct / indirect** : reconnaissance par regex et accès mémoire/register
* **segment\_override\_addressing()** : “[SEG:REG]” pour ES/CS/SS/DS
* **handle\_MOV()** : copie d’un int de src vers dest (void\* interface) citeturn1file1
* **resolve\_addressing()** : itère tous les modes et retourne le premier succès
* **Tests** : résolution de chaque mode, pool de constantes, MOV effectif.

## **Exercice 6 – Segment de code (“CS”) & exécution**

* **search\_and\_replace()** : substitution de variables/labels dans les instructions (pré-traitement)
* **allocate\_code\_segment()** : création de “CS” de taille code\_count, stockage des Instruction\* et init IP à 0
* **fetch\_next\_instruction()** : lecture à IP, incrémentation avec contrôle de limites
* **handle\_instruction()** : implémente MOV, ADD, CMP (mise à jour ZF/SF), JMP/JZ/JNZ/HALT
* **execute\_instruction()** : wrapper qui appelle resolve\_addressing() + handle\_instruction()
* **run\_program()** : exécution pas à pas, affichage états CPU avant/après, contrôle utilisateur

## **Exercice 7 – Segment de pile (“SS”)**

* **cpu\_init()** : allocation de SP et BP (int\* en contexte), création de “SS” taille 128 sous “DS” et init SP=BP=fin de SS
* **push\_value() / pop\_value()** : décrément/incrément SP, gestion underflow/overflow et accès via memory\_handler
* **handle\_instruction()** : support des mnémoniques PUSH [reg] et POP [reg] (par défaut AX)

## **Exercice 8 – Extra Segment (“ES”) & alloc dynamique**

* **segment\_override\_addressing()** : mode “[ES:REG]” intégré à resolve\_addressing()
* **find\_free\_address\_strategy()** : First/Best/Worst Fit dans free\_list, renvoi de l’adresse de début
* **alloc\_es\_segment()** : lit AX=size, BX=stratégie, crée ES (HashMap “allocated”), ZF=0/1 selon succès, init mémoire à 0, stocke base ES dans registre ES
* **free\_es\_segment()** : libère chaque donnée de ES, supprime le segment, ES=-1
* **handle\_instruction()** : support ALLOC et FREE

## **Bilan et perspectives**

L’ensemble des fonctionnalités spécifiées pour les exercices a été implémenté.

Le simulateur gère :

* Allocation/libération de tous types de segments (DS, CS, SS, ES)
* Stockage et récupération de données, toolbox d’adressage
* Parsing complet d’un pseudo-assembleur et exécution pas à pas
* Table de hachage générique et pool de constantes

**Perspectives d’amélioration :**

* Gestion de la mémoire virtuelle et caches
* Ajout d’instructions supplémentaires (MUL, DIV, I/O)
* Interface graphique pour visualisation de la RAM et des registres