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

Compilation et exécution

La compilation et l’exécution des tests sont entièrement automatisées grâce au fichier `Makefile`, qui contient l’ensemble des instructions nécessaires.

Pour compiler l’ensemble du projet ainsi que les tests, il suffit d’exécuter la commande suivante dans le terminal :

make

Une fois la compilation terminée, chaque test est disponible dans le dossier `bin/`. Pour exécuter un test spécifique, il suffit de lancer le binaire correspondant. Par exemple :

./bin/test_memory

- Vous pouvez également cibler directement un test avec `make` :

make bin/test_memory

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