Projet: Réalisation d’un simulateur de CPU
LU2IN006 - 2024-2025

Andre Bertok | Olivier Huang

SOMMAIRE

Resumer du Projet

Reformulation du sujet :

Ce projet consiste à implémenter un simulateur de CPU en C:

- Table de hachage générique : Création d'une structure permettant de stocker des paires clé-valeur avec gestion des collisions par sondage linéaire et suppression via TOMBSTONE.

- Gestion de mémoire dynamique : Allocation/libération de segments mémoire, fusion des espaces libres adjacents, et utilisation d'une table de hachage pour suivre les segments alloués.

- Parser de pseudo-assembleur : Analyse de fichiers assembleur pour extraire les sections .DATA (variables) et .CODE (instructions), ainsi que les étiquettes (labels). Les adresses mémoire des variables sont calculées séquentiellement.

- Simulation du CPU :
    - Initialisation des registres (AX, BX, CX, DX, IP, etc.) et gestion de la mémoire.

    - Allocation d'un segment de données (DS) pour stocker les variables déclarées.

    - Résolution des modes d'adressage (immédiat, registre, direct, indirect) via des expressions régulières.
    - Exécution d'instructions (MOV, ADD, JMP, etc.) avec mise à jour des registres (ZF, SF).

- Segments supplémentaires :
    - Pile (SS) : Gestion via les registres SP/BP et opérations PUSH/POP.
    - Segment dynamique (ES) : Allocation/libération via des stratégies (First Fit, Best Fit, Worst Fit) et adressage explicite (ex: [ES:AX]).

Structure des fichiers :

- **hash.h / hash.c**
    - Table de hachage générique (clé = chaîne, valeur = pointeur)

- **memory.h / memory.c**
    - Gestion des segments mémoire (libres/alloués), store/load

- **parser.h / parser.c**
    - Analyseur pseudo-assembleur (.DATA et .CODE)

- **cpu.h / cpu.c**
    - Simulateur de CPU (registres, adressage, exécution) et segments SS/DS/CS/ES

- **test_exo[1-8].c**
    - Programmes de test pour chaque exercice

- **Makefile**
    - Option de compilation des .h/.c et tests

Structure des jeux de tests :

Nous avons choisi de faire les jeux de tests à base de choix que le correcteur devra choisir pour tester chaque fonctionnalité.

Après compilation et exécution d'un des test_exo[1-8], choisissez parmi les options affichées et suivez les indications.

Exercice 1 - Implémentation d'une table de hachage générique

Structure:

- HashEntry composé de:
  - Une chaîne de caractère qui représente la clé de hachage
  - Une valeur de type quelconque

- HashMap composé de:
  - Un entier pour la taille de la table de hachage
  - Un entier pour l'emplacement actuel dans la table de hachage
  - Une table de hachage

Fonctions:

- simple_hash(const char *str)
  - Une fonction de hachage qui convertit une chaîne de caractère en entier
- h(const char *str, int i)
  - Fonction de probing linéaire
- hashmap_create()
  - Elle alloue et initialise une structure Hashmap
- hashmap_insert(HashMap *map, const char *key, void *value)
  - Fonction pour insérer un élément dans la table de hachage, on boucle sur la table de hachage grâce à la fonction de probing h pour trouver une place.
- hashmap_get(HashMap *map, const char *key)
  - Fonction pour récupérer un élément à partir de sa clé, on boucle sur la table de hachage grâce à la fonction de probing h pour trouver la clé
- hashmap_remove(HashMap *map, const char *key)
  - Fonction pour supprimer un élément de la table de hachage à partir de sa clé, on utilise la fonction de probing h pour trouvé sa clé
- free_HashMap(HashMap *hm)
  - Fonction qui libère la mémoire d'une structure HashMap
- afficher_hashmap(HashMap* hp)
  - Fonction pour afficher une structure HashMap
  -

Jeux de tests

Après initialisation de la table de hachage, nous pouvons tester les différentes fonctions à notre disposition en affichant entre temps la table pour vérifier si tout se passe bien et si les valeurs sont correctes. Les valeurs(test1 100;clé="test1", valeur=100) à insérer et supprimer sont des valeurs choisies par l'utilisateur.

0.Initialisation de la table de hachage
-execute hashmap_create()

```
Option:
0 - Initialiser la table de hachage
1 - Insérer un élément dans la table de hachage
2 - Supprimer un élément dans la table de hachage
3 - récupérer la valeur associé à une clé
4 - Afficher la table de hachage
5 - Quitter
entrée une action:
0
====================================
Initialisation réussi
```

1. ajoute un élément à la table de hachage
-execute hashmap_insert(..)
2. Supprime un élément de la table de hachage
-execute hashmap_remove(..)
2. récupérer un élément de la table de hachage à partir de sa clé
-execute hashmap_get(..)
-affiche l'élément

```
Option:
0 - Initialiser la table de hachage
1 - Insérer un élément dans la table de hachage
2 - Supprimer un élément dans la table de hachage
3 - récupérer la valeur associé à une clé
4 - Afficher la table de hachage
5 - Quitter
entrée une action:
3
====================================
Veuillez choisir la clé à rechercher
test1
Valeur trouvée : 100

Option:
0 - Initialiser la table de hachage
1 - Insérer un élément dans la table de hachage
2 - Supprimer un élément dans la table de hachage
3 - récupérer la valeur associé à une clé
4 - Afficher la table de hachage
5 - Quitter
entrée une action:
2
====================================
Veuillez choisir la clé de l'élément a supprimer
test1
```

4.Afficher_hashmap(..)
-affichage complet de la structure HashMap

```
Option:
0 - Initialiser la table de hachage
1 - Insérer un élément dans la table de hachage
2 - Supprimer un élément dans la table de hachage
3 - récupérer la valeur associé à une clé
4 - Afficher la table de hachage
5 - Quitter
entrée une action:
1
====================================
Veuillez choisir la clé et la valeur associé (on choisira un int pour la valeur pour facilité)
test1 100

Option:
0 - Initialiser la table de hachage
1 - Insérer un élément dans la table de hachage
2 - Supprimer un élément dans la table de hachage
3 - récupérer la valeur associé à une clé
4 - Afficher la table de hachage
5 - Quitter
entrée une action:
4
====================================
size : 128
current_mem : 0
table[113] : key = test1, value = 100
```

Exercice 2 - Gestion dynamique de la mémoire

Structure:

- Segment composé de:
  - Un entier pour la position de début
  - Un entier pour la taille du segment
  - un autre Segment pour pointer vers le prochain Segment

- MemoryHandler composé de:
  - Un tableau de pointeur vers la mémoire allouée
  - Un entier pour la taille totale de la mémoire allouée
  - Une structure Segment qui représente la liste chaînée des segments de mémoire libres
  - Une structure HashMap la table de hachage

Fonctions

- memory_init(int size)
  - Fonction pour allouer et initialiser une structure MemoryHandler
- find_free_segment(MemoryHandler* handler, int start, int size, Segment** prev)
  - Fonction pour trouver s'il y a un segment libre qui commence à start de minimum de taille size, on retourne NULL sinon
- create_segment(MemoryHandler *handler, const char *name, int start, int size)
  - Elle alloue un segment de taille size à la position start s'il y a de la place dans le gestionnaire de mémoire
- remove_segment(MemoryHandler *handler, const char *name)
  - Libère la mémoire d'un segment de nom name et fusionne si besoin dans allocated les segment libre
- print_memory(MemoryHandler *m)
  - Fonction d'affichage du gestionnaire de mémoire
- print_segments(Segment* s)
  - Fonction d'affichage d'un segment
- free_segments(Segment* seg)
  - Fonction pour libérer la mémoire d'un segment
- free_memoryHandler(MemoryHandler *m)
  - Fonction pour libérer la mémoire d'un gestionnaire de mémoire

Jeux de tests

0. Initialisation de la mémoire(ici on choisi 1024)
1. Création du segment test1

```
Option:
0 - Initialiser la mémoire
1 - Créer un segment
2 - Supprimer un segment
3 - Afficher le gestionnaire de mémoire
4 - Quitter
entrée une action:
0
========================================
Choisissez la taille de votre gestionnaire de mémoire
1024

Option:
0 - Initialiser la mémoire
1 - Créer un segment
2 - Supprimer un segment
3 - Afficher le gestionnaire de mémoire
4 - Quitter
entrée une action:
1
========================================
Veuillez choisir le nom, start et taille de votre segment(séparée par un espace)
test1 0 100
```

2.On supprime le segment choisi(ici test1)

```
Option:
0 - Initialiser la mémoire
1 - Créer un segment
2 - Supprimer un segment
3 - Afficher le gestionnaire de mémoire
4 - Quitter
entrée une action:
2
========================================
Veuillez entrer le nom du segment
test1
```

3. On affiche le Gestionnaire
 - Affiche les valeurs du segment de données et l'état du CPU.

```
Option:
0 - Initialiser la mémoire
1 - Créer un segment
2 - Supprimer un segment
3 - Afficher le gestionnaire de mémoire
4 - Quitter
entrée une action:
3
========================================

Affichage MemoryHandler
taille total: 1024

Table de hachage "allocated"
size : 128
current_mem : 0
table[113] : key = test1, value = 0

Liste chainee des segments de memoire libres "free_list"

Segment :
start: 100
size: 924

Tableau de pointeurs vers la memoire allouee "*memory"

N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
```

Exercice 3 - Conception d'un parser pour un langage pseudo-assembleur

Structure:

- Instruction : composé de 3 chaine de caractere
    - 1: Représente le **mnemonic** de l'instruction (MOV, ADD, etc)
    - 2: Représente le premier opérande de l'instruction (DW,DB, ou le type de variable dans .DATA)
    - 3: Représente le second opérande ou valeur de la variable dans .DATA

- ParserResult composé de:
    - 2 tableaux d'instructions (.DATA et .CODE)
    - 2 entiers pour leur taille respective
    - 2 tables de hachage
        - labels -> indices de code_instructions
        - nom de variable -> adresse mémoire

Fonctions :

- parse_data_instruction(const char *line, HashMap *memorylocations)
    - Fonction qui permet d'analyser et stocker une ligne de la section .DATA d'un programme en pseudo-assembleur. Elle associe chaque variable à son adresse séquentielle.
- parse_code_instruction(const char *line, HashMap *labels, int code count)
    - Fonction qui permet d'analyser et stocker une ligne de la section .CODE d'un programme en pseudo-assembleur.
- parse(const char *filename)
    - Fonction qui analyse et parse un fichier passé en paramètre elle construit et renvoie la structure ParserResult du fichier
- free_parser_result(ParserResult *result)
  Elle libère la mémoire d'une structure ParserResult

Jeux de tests:

Ce programme permet de tester un parser de pseudo-assembleur, capable d'analyser les sections .DATA et .CODE d'un fichier source. Il propose les différentes opérations :

1. Parser une instruction .DATA (test de parse_data_instruction)
   - Affiche les instructions extraites et la table de hachage mémoire associée.

```
Votre choix : 1
L'instruction: arr DB 5,6,7,8
L'instruction: z DB 9


=== Contenu de la table de hachage ===
size : 128
current_mem : 5
table[69] : key = arr, value = 0
table[122] : key = z, value = 4

=== Instructions extraites ===
mnemonic: arr
operand1: DB
operand2: 5,6,7,8
mnemonic: z
operand1: DB
operand2: 9
```

2. Parser un fichier assembleur complet ("assembler_exo3-4.txt")
   - Identifie les instructions .DATA et .CODE
   - Gère les labels et les localisations mémoire
   - Affiche la structure complète (ParserResult)

```
Votre choix : 2

=== Résultat de l'analyse ===
data_count: 3
0 :
mnemonic: x
operand1: DW
operand2: 42
1 :
mnemonic: arr
operand1: DB
operand2: 20,21,22,23
2 :
mnemonic: y
operand1: DB
operand2: 10

code_count: 3
0 :
mnemonic: MOV
operand1: AX
operand2: x
1 :
mnemonic: ADD
operand1: AX
operand2: y
2 :
mnemonic: JMP
operand1: loop
operand2:

size : 128
current_mem : 0
table[46] : key = start, value = 0
table[58] : key = loop, value = 1
size : 128
current_mem : 6
table[69] : key = arr, value = 1
table[120] : key = x, value = 0
table[121] : key = y, value = 5
```

```
1    .DATA
2    x DW 42
3    arr DB 20,21,22,23
4    y DB 10
5    .CODE
6    start: MOV AX,x
7    loop: ADD AX,y
8    JMP loop
```

3. Quitter le programme

```
Votre choix : 3
Exit !
==28916==
==28916== HEAP SUMMARY:
==28916==     in use at exit: 0 bytes in 0 blocks
==28916==   total heap usage: 64 allocs, 64 frees, 14,814 bytes allocated
==28916==
==28916== All heap blocks were freed -- no leaks are possible
==28916==
==28916== For lists of detected and suppressed errors, rerun with: -s
==28916== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Exercice 4 - Allocation d’un segment de données

Structure:

- CPU composé de:
  - Une structure MemoryHandler qui représente le gestionnaire de mémoire
  - Une structure HashMap qui représente un registre
  - Une structure HashMap pour stocker les valeurs immédiates

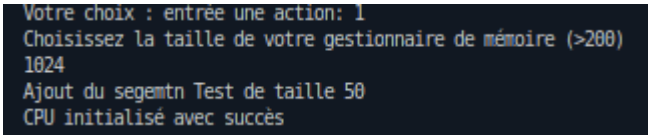
Fonctions

- cpu_init(int memory_size)
  - Elle alloue et initialise une structure de type CPU
- cpu_destroy(CPU *cpu)
  - Elle libère la mémoire d’une structure de type CPU
- store(MemoryHandler *handler, const char *segment_name, int pos, void *data)
  - Fonction qui permet de stocker une donnée data à la position pos de segment name. Récupère le segment via son nom depuis la HashMap allocated et calcul son adresse mémoire et insère la nouvelle donnée
- load(MemoryHandler *handler, const char *segment_name, int pos)
  - Charge une donnée stockée dans un segment mémoire à une position donnée. si segment_name existe retourne sa position
- allocate_variables(CPU *cpu, Instruction** data_instructions, int data_count)
  - Fonction qui alloue un segment de données (“DS”) et y stocke les variables extraites des instructions de données.
- print_data_segment(CPU *cpu)
  - Fonction annexe a print_data_segment
- print_segment_data(CPU *cpu, const char *segment_name)
  - Affiche les données d’un segment donné. Elle récupere le segment grâce à segment_name et affiche chaque valeur
- print_cpu(CPU* cpu)
  - Fonction d’affichage d’une structure de type CPU

Jeux de tests

Ce programme simule un CPU avec gestion de mémoire segmentée. Il propose les différentes opérations :

1. Initialisation du CPU :
   - L'utilisateur choisit une taille mémoire (>200).
   - Un segment "TEST" est créé (taille 50, à l’adresse 129).



2. Libération du CPU :
   - Libère toutes les ressources allouées (segments, mémoire, CPU).

```
Votre choix : entrée une action: 2
Liberation du CPU...
```

3. Stockage dans le segment TEST (fonction store()):
   - Permet de stocker une valeur à une position donnée (0 à 49) dans le segment "TEST".

```
Votre choix : entrée une action: 3
Entrer la valeur (int) que vous voulez stocker dans le segment "TEST" et la postion dans le segment (entre 0 et 50) (separer par un espace)
42 6
Valeur 42 stockée à la position 6 dans le segment TEST
```

4. Lecture dans le segment TEST (fonction load()) :
   - Affiche la valeur stockée à une position du segment "TEST".

```
Votre choix : entrée une action: 4
Entrer la position dans le segment TEST (entre 0 et 50)
6
Valeur chargée à la position 6 dans le segment TEST: 42
```

5. Allocation dynamique depuis assembler_exo3-4.txt.txt :
   - Lis le fichier “assembler_exo3-4.txt”, extrait les variables et alloue un segment mémoire pour les stocker.

```
.DATA
x DW 42
arr DB 20,21,22,23
y DB 10
.CODE
start: MOV AX,x
loop: ADD AX,y
JMP loop
```

```
Votre choix : entrée une action: 5
Allocation dynamique du segment de données réussie
```

6. Affichage du segment de données :
   - Affiche les valeurs du segment de données et l'état du CPU.

```
Votre choix : entrée une action: 6

Affichage MemoryHandler
taille total: 1024

Table de hachage "allocated"
size : 128
current_mem : 0
table[23] : key = DS, value = 179
table[38] : key = SS, value = 0
table[64] : key = TEST, value = 129

Liste chainee des segments de memoire libres "free_list"

Segment :
start: 128
size: 1

Segment :
start: 185
size: 839

Tableau de pointeurs vers la memoire allouee "*memory"

N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N O N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N O O O O O O N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N

Table de hachage "context"
size : 128
current_mem : 0
table[18] : key = BP, value = 128
table[24] : key = ES, value = -1
table[25] : key = AX, value = 0
table[26] : key = BX, value = 0
table[27] : key = CX, value = 0
table[28] : key = DX, value = 0
table[29] : key = IP, value = 0
table[30] : key = SF, value = 0
table[32] : key = ZF, value = 0
table[35] : key = SP, value = 128

Table de hachage "constant_pool"
size : 128
current_mem : 0

DS Segment

0: 42
1: 20
2: 21
3: 22
4: 23
5: 10

Segment ES not found.
```

7. Affichage du segment TEST :
   - Affiche le contenu du segment "TEST"

```
Votre choix : entrée une action: 7

TEST Segment

0: (nil)
1: (nil)
2: (nil)
3: (nil)
4: (nil)
5: (nil)
6: 42
7: (nil)
8: (nil)
```

8. Quitter :
   - Libère toutes les ressources et termine le programme.

```
8. Quitter
Votre choix : entrée une action: 8
Liberation du CPU...
==19195==
==19195== HEAP SUMMARY:
==19195==     in use at exit: 0 bytes in 0 blocks
==19195==   total heap usage: 98 allocs, 98 frees, 27,321 bytes allocated
==19195==
==19195== All heap blocks were freed -- no leaks are possible
==19195==
==19195== For lists of detected and suppressed errors, rerun with: -s
==19195== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Exercice 5 - Expressions régulières et résolution d’adressage

Fonctions

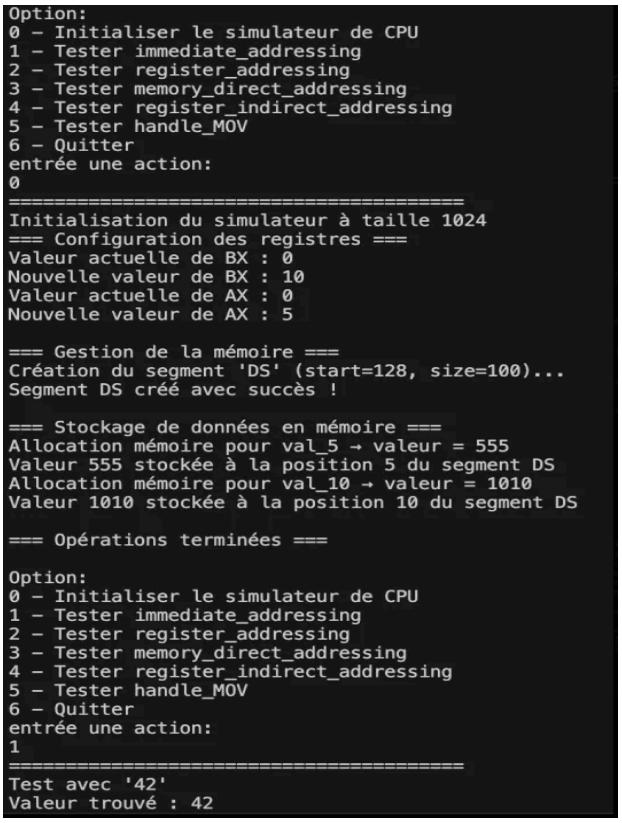
- immediate_addressing(CPU *cpu, const char *operand)
    - Elle vérifie si l'opérande correspond à son type d'adressage, stock la valeur dans constant_pool si elle n’y est pas déjà et retourne un pointeur vers cette valeur
- register_addressing(CPU *cpu, const char *operand)
    - Elle vérifie si l'opérande correspond à son type d'adressage, retourne le pointeur vers la valeur du registre dans constant_pool si elle existe
- memory_direct_addressing(CPU *cpu, const char *operand)
    - Elle vérifie si l'opérande correspond à son type d'adressage, extrait son adresse numérique et retourne le pointeur vers la valeur stocké à cette adresse
- register_indirect_addressing(CPU *cpu, const char *operand)
    - Elle vérifie si l'opérande correspond à son type d'adressage, extrait le nom du registre, récupère sa valeur qui servira d’adresse mémoire, puis accède à la valeur stockée à cette adresse dans le segment de données. Enfin retourne un pointeur vers cette valeur,
- resolve_addressing(CPU *cpu, const char *operand)
    - Fonction qui permet d’identifier le type d'adressage
- handle_MOV(CPU* cpu, void* src, void* dest)
    - Fonction qui simule l’instruction MOV

Jeux de tests

Ici nous avons décidé de faire des tests directement au lieu de vous demander de fournir les expressions régulières (“42”, “BX”, “[5]”, “[AX]”). On ajoute des données au cpu pour facilité le reste. On peut voir le résultat après utilisation de handle_MOV sur [BX] et CX.

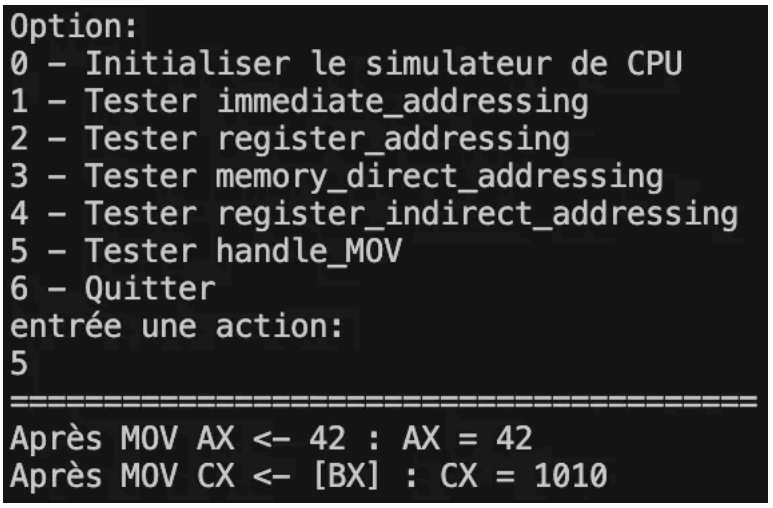
1.2.3.4.Vérification des différents types d’adressage

-Affiche le résultat trouvé selon le type d’adressage



5. Simulation de l’instruction MOV.

-Affiche le résultat avant-après



Exercice 6 - Allocation d'un segment de codes et exécution

Fonctions

- replace_label(Instruction *instr, HashMap *labels)
    - Fonction annexe pour remplacer les étiquettes par leur adresse dans le code
- replace_memvar(Instruction *instr, HashMap *memloc)
    - Fonction annexe pour remplacer les variables par leur adresse dans le segment de données
- resolve_constants(ParserResult *result)
    - Utilise les fonction replace_label et replace_memvar pour remplacer les variables et les étiquettes par leur adresse respectivement dans le segment de données et dans le code
- allocate_code_segment(CPU *cpu, Instruction **code_instructions, int code_count)
    - Fonction qui alloue et initialise un segment "CS" pour y stocker les instructions
- handle_instruction(CPU *cpu, Instruction *instr, void* src, void *dest)
    - Fonction qui identifie l'instruction puis l'exécute
- execute_instruction(CPU *cpu, Instruction *instr)
    - Fonction qui résout les adresse des opérande en fonction du type d'adressage puis utilise handle_instruction
- fetch_next_instruction(CPU *cpu)
    - Fonction qui récupère la prochaine instruction depuis CS[IP] puis incrémente IP
- run_program(CPU *cpu)
    - utilise toutes les fonctions précédentes pour afficher l'état final de cpu

Jeux de tests

Le programme ici permet de parser un fichier assembleur (exo6_assembler.txt) et de charger ses instructions dans le segment code pour les exécuter. On peut voir l'exécution complète en utilisant l'option 4 run program

1. Parser un fichier assembleur complet ("assembler_exo3-4.txt")
   - Identifie les instructions .DATA et .CODE
   - Gère les labels et les localisations mémoire
   - Affiche la structure complète (ParserResult)

```
Option:
0 - Initialiser le simulateur de CPU
1 - Parsing du fichier exo6_assembler.txt
2 - Initialiser le segment 'CS'
3 - Handle instruction
4 - Run Program
5 - Quitter
entrée une action:
1
=======================================
=== Instructions AVANT resolve_constants ===
mnemonic: MOV
operand1: AX
operand2: [X]
=======================================
mnemonic: ADD
operand1: AX
operand2: [Y]
=======================================
mnemonic: JMP
operand1: loop
operand2:
=======================================

=== Instructions APRÈS resolve_constants ===
mnemonic: MOV
operand1: AX
operand2: [0]
=======================================
mnemonic: ADD
operand1: AX
operand2: [5]
=======================================
mnemonic: JMP
operand1: 1
operand2:
=======================================
```

2.Initialisation du segment “CS

-Affiche le segment et le nombre d’instruction contenu

```
Option:
0 - Initialiser le simulateur de CPU
1 - Parsing du fichier exo6_assembler.txt
2 - Initialiser le segment 'CS'
3 - Handle instruction
4 - Run Program
5 - Quitter
entrée une action:
2
=======================================
Segment DS (données) initialisé avec 3 cellules.
Segment CS (code) initialisé avec 3 instructions.
Segment CS initialisé avec succès.

Option:
0 - Initialiser le simulateur de CPU
1 - Parsing du fichier exo6_assembler.txt
2 - Initialiser le segment 'CS'
3 - Handle instruction
4 - Run Program
5 - Quitter
entrée une action:
3
=======================================
Instruction à exécuter : MOV AX [0]
Après exécution, AX = 42
```

4. exécute le tout à l'aide de run program pour vérifier que toutes les autre marches

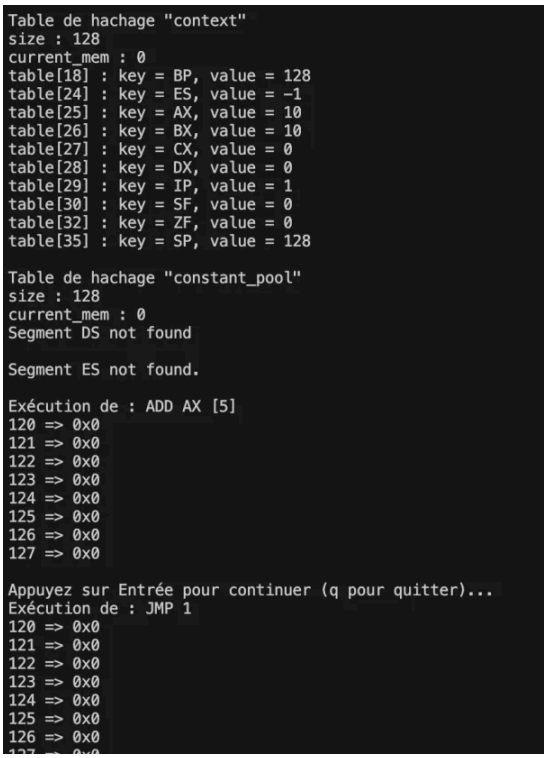
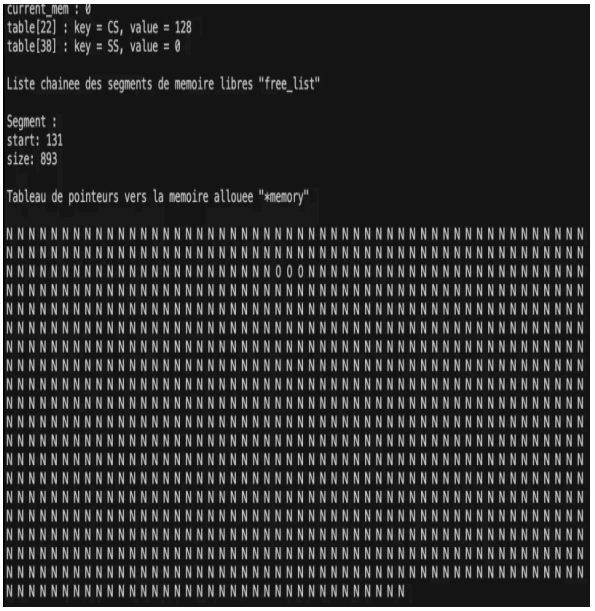


Table de hachage "context"
size : 128
current_mem : 0
table[18] : key = BP, value = 128
table[24] : key = ES, value = -1
table[25] : key = AX, value = 10
table[26] : key = BX, value = 10
table[27] : key = CX, value = 0
table[28] : key = DX, value = 0
table[29] : key = IP, value = 1
table[30] : key = SF, value = 0
table[32] : key = ZF, value = 0
table[35] : key = SP, value = 128
Table de hachage "constant_pool"
size : 128
current_mem : 0
Segment DS not found
Segment ES not found.
Exécution de : ADD AX [5]
120 => 0x0
121 => 0x0
122 => 0x0
123 => 0x0
124 => 0x0
125 => 0x0
126 => 0x0
127 => 0x0
Appuyez sur Entrée pour continuer (q pour quitter)...
Exécution de : JMP 1
120 => 0x0
121 => 0x0
122 => 0x0
123 => 0x0
124 => 0x0
125 => 0x0
126 => 0x0


current_mem : 0
table[22] : key = CS, value = 128
table[38] : key = SS, value = 0
Liste chainee des segments de memoire libres "free_list"
Segment :
start: 131
size: 893
Tableau de pointeurs vers la memoire allouee "*memory"

Exercice 7 - Implémentation d’un segment de pile (Stack Segment)

Fonctions

- push_value(CPU *cpu, int value)
    - Récupère le segment “SS” et le registre “SP” pour décrémenter “SP” pour allouer et stocker value dedans et place le pointeur dans memory[SP]
- pop_value(CPU *cpu, int *dest)
    - Lis la valeur pointée par memory[SP] , la copie dans dest puis libère la mémoire et incrémente SP

Jeux de tests

Ce programme permet de tester la gestion d’un segment de pile (SS) dans un CPU simulé. Il repose sur deux registres : SP (pointeur de pile) et BP (base de pile). Il propose les différentes opérations :

1. Initialisation du CPU :
   - Initialise la mémoire, crée le segment de pile "SS" et positionne SP et BP.

```
Votre choix : entrée une action: 1
Choisissez la taille de votre gestionnaire de mémoire (>200)
1024
CPU initialisé avec succès
```

2. PUSH sur la pile :
   - Ajoute une valeur au sommet de la pile. Renvoie une erreur si la pile est pleine.

```
Votre choix : entrée une action: 2
Valeur (int) a PUSH sur la pile : 45
la fonction push_value a ete executer avec succès avec 45
```

```
Votre choix : entrée une action: 2
Valeur (int) a PUSH sur la pile : 79
la fonction push_value a ete executer avec succès avec 79
```

```
Votre choix : entrée une action: 2
Valeur (int) a PUSH sur la pile : 63
la fonction push_value a ete executer avec succès avec 63
```

3. POP depuis la pile :
   - Retire la valeur au sommet de la pile. Renvoie une erreur si la pile est vide.

```
Votre choix : entrée une action: 3
la fonction pop_value a ete executer avec succès et a retourner 63
```

4. Lecture du fichier assembler_exo7.txt :
   - Parse le fichier, alloue les segments de données et de code, puis exécute les instructions.

```
Votre choix : entrée une action: 4

=== État initial du CPU ===

Affichage MemoryHandler
taille total: 1024

Table de hachage "allocated"
size : 128
current_mem : 0
table[22] : key = CS, value = 131
table[23] : key = DS, value = 128
table[38] : key = SS, value = 0

Liste chainee des segments de memoire libres "free_list"

Segment :
start: 135
size: 889

Tableau de pointeurs vers la memoire allouee "*memory"

N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N O O O O O O O O O N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N

Table de hachage "context"
size : 128
current_mem : 0
table[18] : key = BP, value = 128
table[24] : key = ES, value = -1
table[25] : key = AX, value = 0
table[26] : key = BX, value = 0
table[27] : key = CX, value = 0
table[28] : key = DX, value = 0
table[29] : key = IP, value = 0
table[30] : key = SF, value = 0
table[32] : key = ZF, value = 0
table[35] : key = SP, value = 126

Table de hachage "constant_pool"
size : 128
current_mem : 0

DS Segment

0: 10
1: 25
2: 1

Segment ES not found.
```

```
Exécution de : MOV BX [0]
120 => (nil)
121 => (nil)
122 => (nil)
123 => (nil)
124 => (nil)
125 => (nil)
126 => 79
127 => 45

Appuyez sur Entrée pour continuer (q pour quitter)...
Exécution de : PUSH BX
120 => (nil)
121 => (nil)
122 => (nil)
123 => (nil)
124 => (nil)
125 => 10
126 => 79
127 => 45

Appuyez sur Entrée pour continuer (q pour quitter)...

Exécution de : PUSH [1]
120 => (nil)
121 => (nil)
122 => (nil)
123 => (nil)
124 => 25
125 => 10
126 => 79
127 => 45

Appuyez sur Entrée pour continuer (q pour quitter)...

Exécution de : POP DX
120 => (nil)
121 => (nil)
122 => (nil)
123 => (nil)
124 => (nil)
125 => 10
126 => 79
127 => 45

Appuyez sur Entrée pour continuer (q pour quitter)...

Fin du programme (IP hors limites)

=== État final du CPU ===

Affichage MemoryHandler
taille total: 1024

Table de hachage "allocated"
size : 128
current_mem : 0
table[22] : key = CS, value = 131
table[23] : key = DS, value = 128
table[38] : key = SS, value = 0

Liste chainee des segments de memoire libres "free_list"

Segment :
start: 135
size: 889
```

```
Tableau de pointeurs vers la memoire allouee "*memory"

N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N O O O O O O O O O O N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N
N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N N

Table de hachage "context"
size : 128
current_mem : 0
table[18] : key = BP, value = 128
table[24] : key = ES, value = -1
table[25] : key = AX, value = 0
table[26] : key = BX, value = 10
table[27] : key = CX, value = 0
table[28] : key = DX, value = 25
table[29] : key = IP, value = 4
table[30] : key = SF, value = 0
table[32] : key = ZF, value = 0
table[35] : key = SP, value = 125

Table de hachage "constant_pool"
size : 128
current_mem : 0

DS Segment

0: 10
1: 25
2: 1

Segment ES not found.
```

```
5.DATA
X DW 10
Y DB 25
Z DB 1
.CODE
MOV BX,[X]
PUSH BX
PUSH [Y]
POP DX
```

5. Affichage de la pile :
   - Affiche le contenu actuel du segment de pile (SS).

```
Votre choix : entrée une action: 5
120 => (nil)
121 => (nil)
122 => (nil)
123 => (nil)
124 => (nil)
125 => 10
126 => 79
127 => 45
```

6. Quitter :
   - Libère le CPU et termine le programme.

```
Votre choix : entrée une action: 6
Liberation du CPU...
==20301==
==20301== HEAP SUMMARY:
==20301==     in use at exit: 0 bytes in 0 blocks
==20301==   total heap usage: 1,327 allocs, 1,327 frees, 120,290 bytes allocated
==20301==
==20301== All heap blocks were freed -- no leaks are possible
==20301==
==20301== For lists of detected and suppressed errors, rerun with: -s
==20301== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Exercice 8 - Gestion de l'allocation dynamique (Extra Segment)

Fonctions

- segment_override_addressing(CPU* cpu, const char* operand)
    - Elle vérifie si l'opérande correspond à son type d'adressage, extrait le segment et le registre correspondants pour récupérer et retourner la donnée stockée dans le segment à la position spécifiée
- find_free_address_strategy(MemoryHandler *handler, int size, int strategy)
    - Trouve un segment libre selon la stratégie donnée(0 pour la première place disponible,1 pour le bloc avec la taille suffisante la plus rapprochée de size et 2 pour le bloc avec le plus d'espace)
- alloc_es_segment(CPU *cpu)
    - Alloue un segment "ES" qui utilise la taille dans le registre "AX" et la stratégie "BX" pour chercher une adresse avec la fonction find_free_adress_strategy et initialise la mémoire du segment à 0
- free_es_segment(CPU *cpu)
    - charge l'adresse "ES" pour libérer la mémoire de son segment

Jeux de tests

Ce programme teste la gestion du segment mémoire dynamique "ES" (Extra Segment), utilisé pour l'allocation et l'accès explicite via des adresses de type [ES:REGISTRE]. Il propose les différentes opérations :

1. Initialisation du CPU :
   - Initialise la mémoire et les registres, dont le registre ES (valeur initiale : -1).

```
Votre choix : entrée une action: 1
Choisissez la taille de votre gestionnaire de mémoire (>200)
1024
CPU initialisé avec succès
```

2. Création du segment ES :
   - Alloue dynamiquement un segment en utilisant une taille (AX) et une stratégie (BX) :
     0 = First Fit, 1 = Best Fit, 2 = Worst Fit.

```
Votre choix : entrée une action: 2
Entrer la valeur (int) pour la taille du segment ES (>5 minimum) et la strategie (separer par un espace)
50 0
AX = 50, BX (stratégie) = 0
Après ALLOC, registre ES = 128
```

3. Test de segment_override_addressing :
   - Lis une valeur dans le segment ES via un adressage explicite [ES:AX].

```
Votre choix : entrée une action: 3
Stocké 42 en ES[2]
Registre AX défini à 2
Appel a segment_override_addressing avec "[ES:AX]"
Lecture segment_override_addressing: 42
```

4. Test de resolve_addressing :
   - Valide la résolution d'un adressage [ES:BX] dans le segment.

```
Votre choix : entrée une action: 4
Registre BX mis à jour à 5
Stocké 42 en position 5 dans ES
Lecture mémoire [ES:BX] => 12
```

5. Test des stratégies d'allocation :
   - Affiche les adresses retournées pour les stratégies First Fit, Best Fit et Worst Fit.

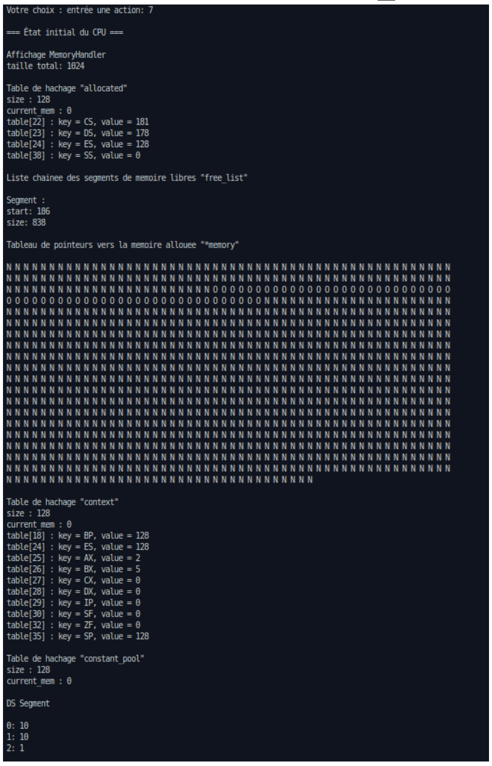
```
Votre choix : entrée une action: 5
Création de segments de test :
T1 @200, taille 50
T2 @300, taille 100
T3 @600, taille 30
First-fit (stratégie 0) pour 60 octets → adresse retournée : 400
Best-fit (stratégie 1) pour 60 octets → adresse retournée : 400
Worst-fit (stratégie 2) pour 60 octets → adresse retournée : 630
```

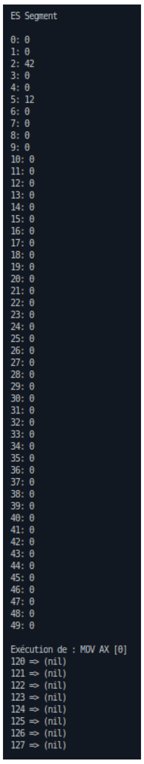
6. Libération du segment ES :
   - Libère les cases mémoire allouées dans ES et réinitialise le registre ES.

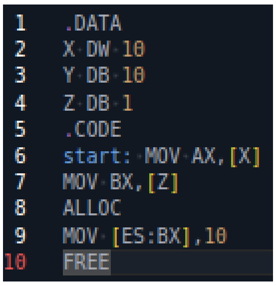
```
Votre choix : entrée une action: 6
Segment ES libéré avec succès
```

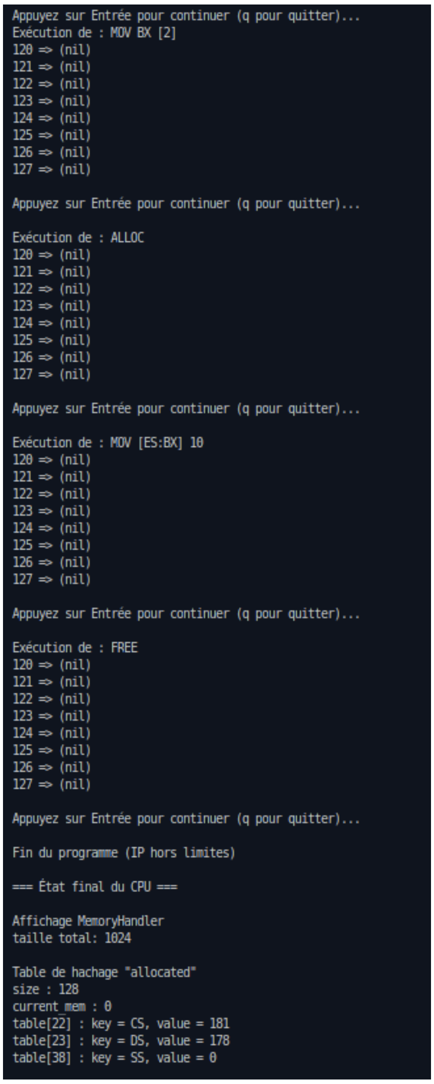
7. Test de ALLOC / FREE depuis un fichier :

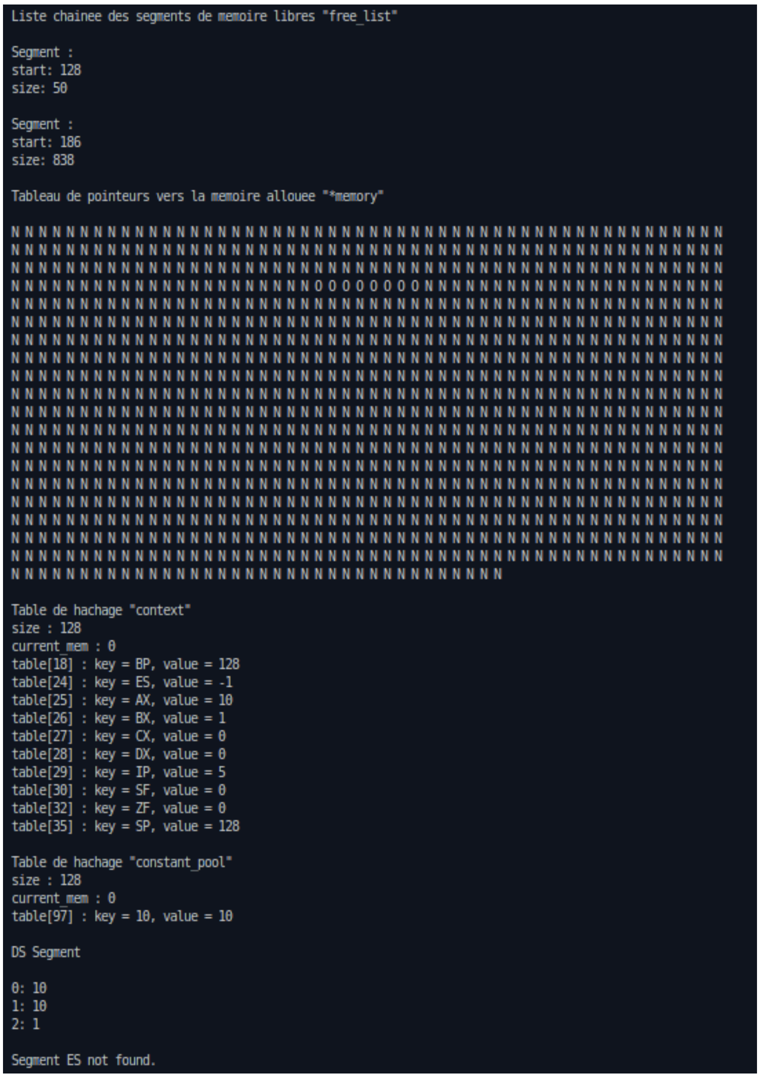
- Parse le fichier assembler_exo8.txt et exécute les instructions ALLOC/FREE.



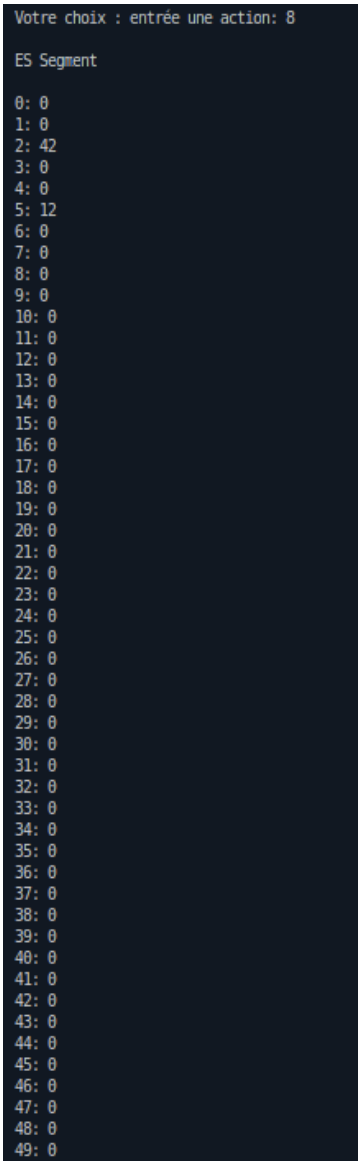








8. Affichage du segment ES :
   - Affiche toutes les valeurs contenues dans le segment ES.



9. Quitter :
   - Libère la mémoire et termine le programme.