Rapport de projet
Structure de donnée

Nom 1 : Achmad Firza Fuadi
Nom 2 : Toyin Ayodele

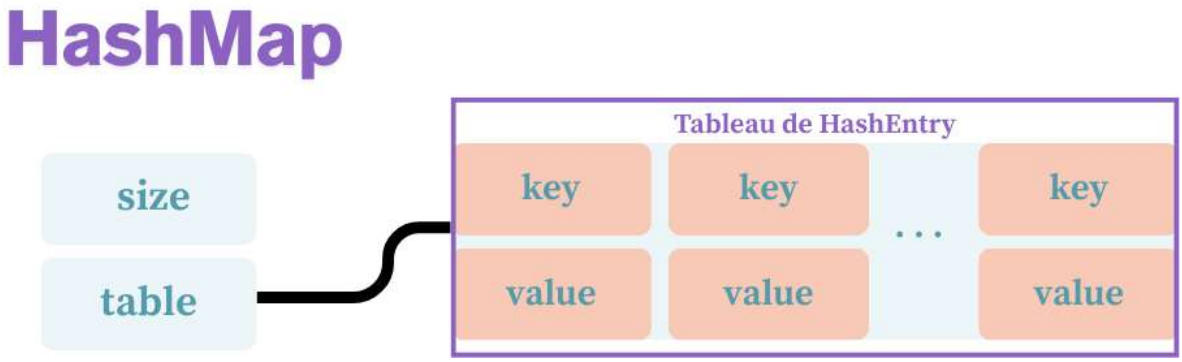
Le projet que l'on a eu a réaliser porte sur un CPU. Un CPU, ou processeur, est l'unité centrale de traitement qui exécute les instructions d'un programme. Son fonctionnement repose sur un cycle d'instruction, dans lequel il récupère une instruction en mémoire, la décode, puis l'exécute en manipulant les registres et la mémoire.
Le but de ce projet était donc d'implémenter une version simplifiée de ce fonctionnement.

Descriptions des structures manipulées

- **HashEntry :**

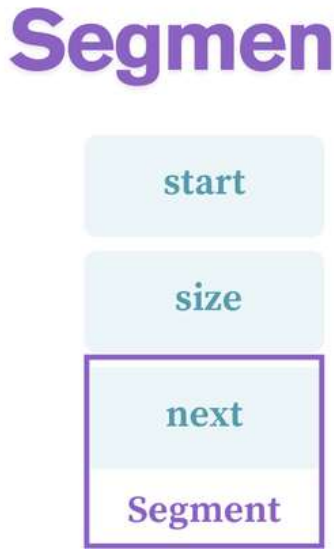
- Structure permettant d'associer des clés (chaînes de caractères) à des valeurs de type générique void*.

- **HashMap :**



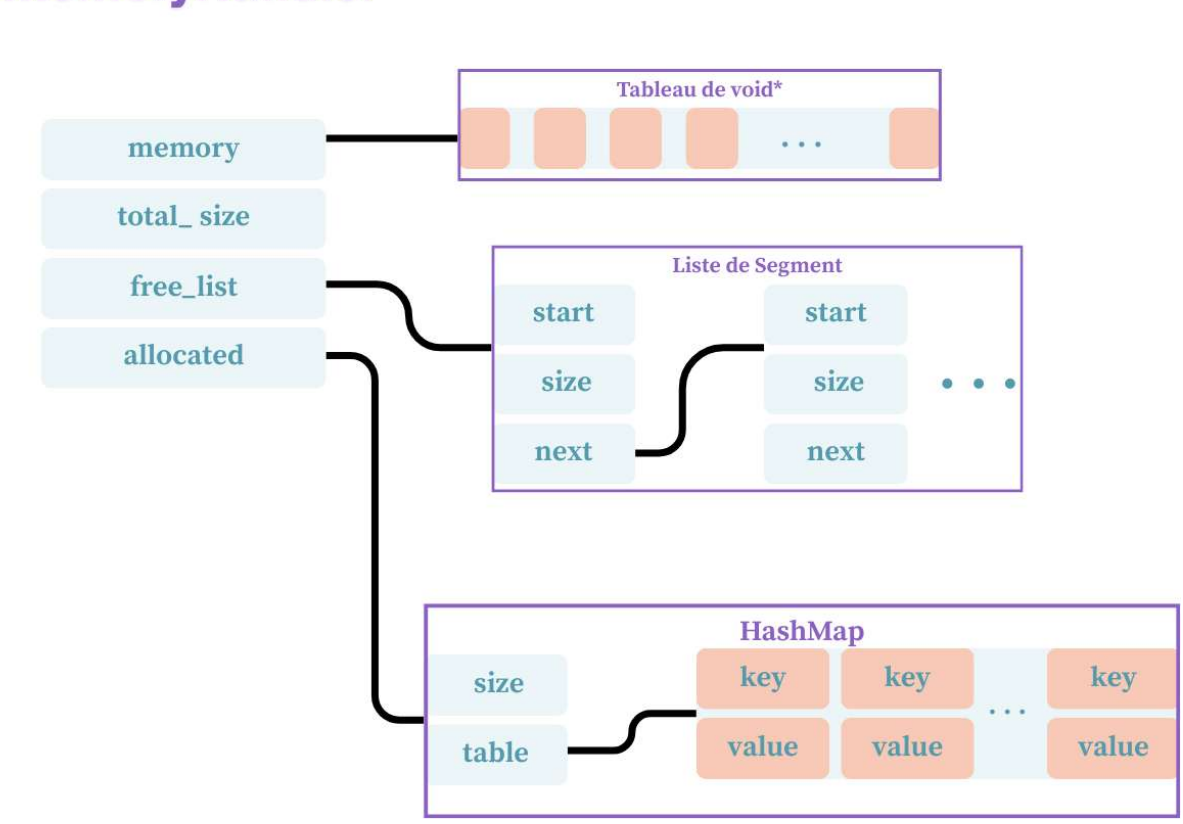
- Structure modélisant une table de hachage (variable *table*).
- Table de hachage de taille *size*.

- **Segment :**



- Liste chainée de segments.
- Un segment est caractérisé par sa position de début dans la mémoire (variable *start*), et sa taille (variable *size*).

- **MemoryHandler :**



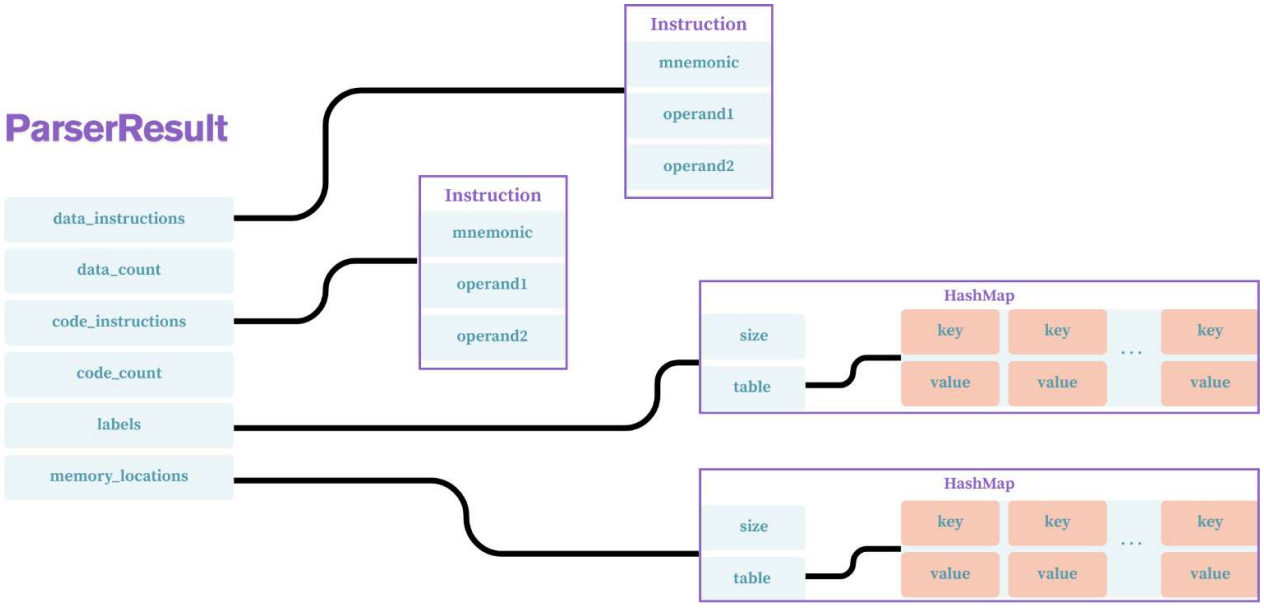
- Structure caractérisant le gestionnaire de mémoire.
- Le gestionnaire contient :

- Un tableau de pointeurs vers la mémoire allouée(variable *memory*);
- La taille totale de la mémoire (variable *total_size*);
- Liste chainée des segments de mémoire libres (*free_list*);
- Table de hachage des segments alloués, associant le nom du segment à la structure de ce segment.

- **Instruction :**

- Structure permettant de stocker une instruction.
- *mnemonic* représente le mnémonique de l'instruction (par exemple MOV, ADD, JMP. . .) ou le nom de la variable pour les instructions.
- *operand1* (optionnel) représente le premier opérande (registre, variable, valeur immédiate) ou le type de la variable
- *operand2* (optionnel) second opérande ou valeur de la variable.

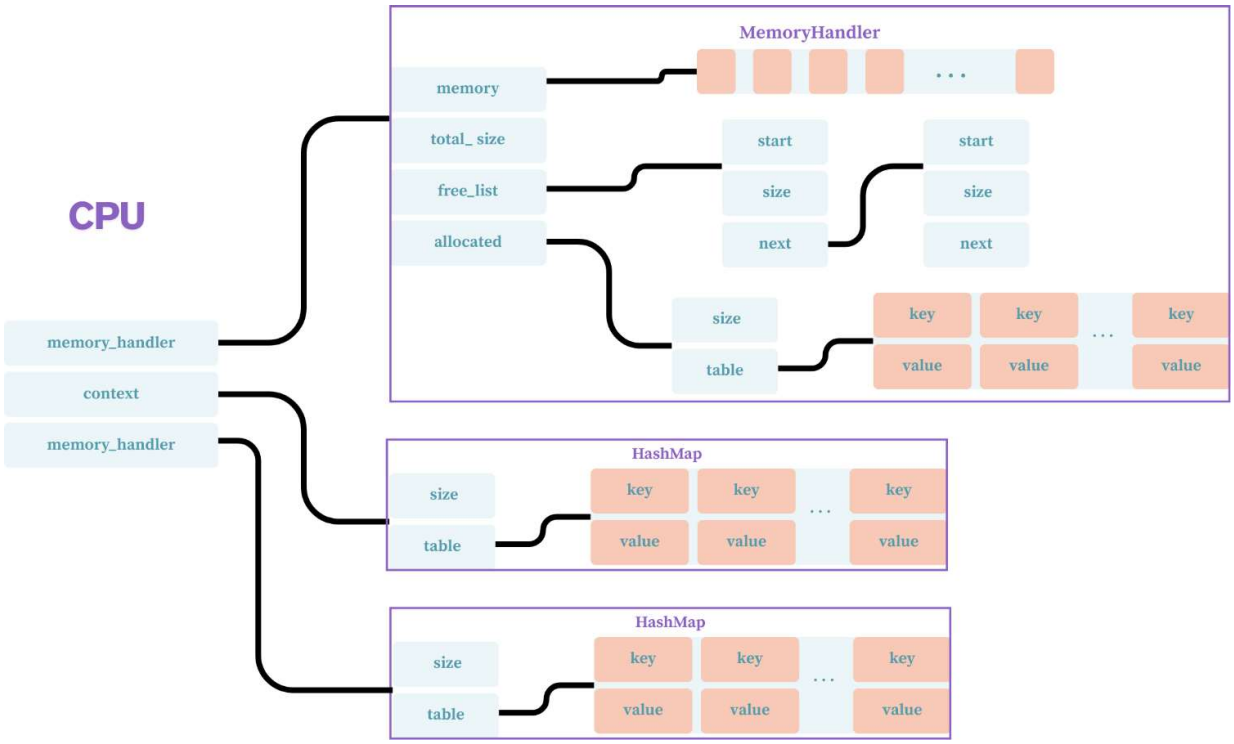
- **ParserResult :**



- Structure stockant le résultat du parsing.

- Cette structure contient :
    - Tableau de pointeurs *data_instructions* vers des structures Instruction correspondant à la section .DATA;
    - Tableau de pointeurs *code_instructions* vers des structures Instructioncorrespondant à la section .CODE;
    - Le nombre d'éléments dans le tableau data_instructions (variable *data_count);*
    - Le nombre d'éléments dans le tableau *code_instructions* (variable *code_count);*
    - Une table de hachage associant les noms d'étiquettes (comme "loop") à leur position dans le tableau code instructions;
    - Une table de hachage associant les noms de variables à son adresse séquentielle (variable *memory_locations).*

- **CPU :**



- Structure modélisant un CPU.
- Un CPU est composé d'un gestionnaire de mémoire *memory_handler,* d'une table de hachage *context* permettant de stocker les valeurs des registres (emplacements mémoire internes au processeur) et d'une table de hachage *constant_pool* pour stocker les valeurs immédiates.

Description globale du code et fonctions principales:

Le code est organisé de la façon suivante :

- Les fichiers .c contenant le code des différentes fonctions : plus precisement, exercice1.c contient les fonctions de *HashMap*; segment.c contient les fonctions

de *Segment;* parser.c contient les fonctions de *ParserResult*; cpu.c contient les fonctions de *CPU*; et cpu_core.c contient les fonctions principales nécessaires à l'exécution du programme;
- Les fichiers .h contenant les structures et les entêtes des fonctions;
- Le Makefile compilant tout les fichiers.
- Le main.c pour l'exécution du programme

L'objectif de ce code est de simuler un processeur.

Les fonctions principales du programme incluent :

- cpu_init : initialise le processeur.

  On commence par initialiser un processeur avec ses registres, son gestionnaire de mémoires,…

- parse : crée une structure ParserResult complète à partir d'un fichier pseudo-assembleur.

  Un programme écrit en pseudo-assembleur est organisé en sections, qui sont des parties distinctes du code ayant chacune un rôle précis. Dans notre cas nous avons deux sections : La section .DATA, qui contient les déclarations de variables et de constantes, et la section .CODE, qui regroupe les instructions exécutables du programme. Donc la fonction parse identifie ces sections et extrait les instructions qui y figurent pour stocker le resultats du parsing dans une structure ParserResult.

- allocate_variables : permet d'allouer dynamiquement le segment de données (DS) en fonction des déclarations récupérées par le parser.
- allocate_code_segment : alloue un segment de code (CS), et y stocke les instructions de codes.
  On stocke dans la mémoire du CPU les instructions de données et de codes.
- run_program : exécute un programme préalablement chargé dans le CPU en mode pas à pas.

On peux maintenant exécuter les instructions se trouvant dans la mémoire du CPU. Cette fonction permet d'exécuter les instructions séquentiellement (en permettant à l'utilisateur de contrôler le deroulement) , et affiche l'état du CPU avant et après l'exécution.

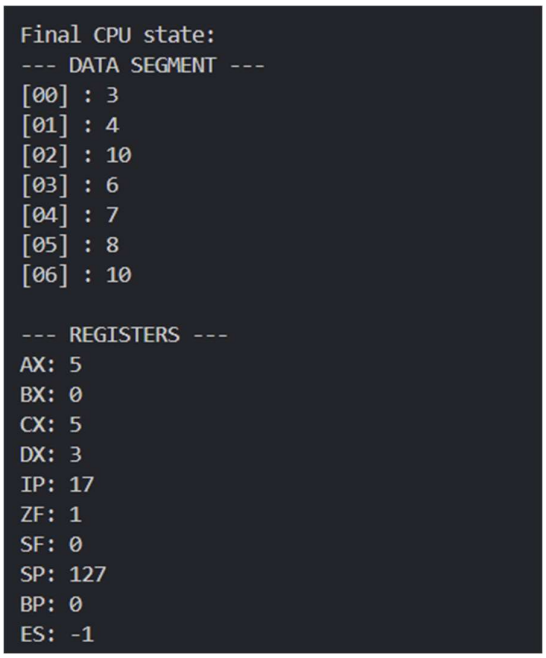
Tests réalisés :

```
.DATA
x DW 3
y DB 4
arr DB 5,6,7,8
z DB 10
.CODE
start: MOV AX,[x]
MOV BX,[AX]
ADD DX,[y]
MOV BX,0
ALLOC
loop: MOV CX,[arr]
PUSH
ADD AX,2
MOV [ES:0],99
ADD [arr],5
POP DX
CMP CX,5
JZ end
JMP loop
end: FREE
HALT
MOV AX,BX
```

Voici le fichier test.txt que nous utilisons pour tester le bon fonctionnement de notre programme.

```
running program...
Initial CPU state:
--- DATA SEGMENT ---
[00] : 3
[01] : 4
[02] : 5
[03] : 6
[04] : 7
[05] : 8
[06] : 10

--- REGISTERS ---
AX: 3
BX: 6
CX: 1
DX: 0
IP: 0
ZF: 0
SF: 0
SP: 127
BP: 0
ES: -1

--- EXECUTION CONTEXT ---
Previous: No previous instruction
Current: Beginning of program
Next: MOV AX [0]

--- INTERACTIVE PROMPT ---
Press ENTER to execute next instruction
Press 'q' to quit execution
```

```
--- DATA SEGMENT ---
[00] : 3
[01] : 4
[02] : 10
[03] : 6
[04] : 7
[05] : 8
[06] : 10

--- REGISTERS ---
AX: 5
BX: 0
CX: 5
DX: 3
IP: 15
ZF: 1
SF: 0
SP: 127
BP: 0
ES: -1

--- EXECUTION CONTEXT ---
Previous: JZ 14
Current: FREE
Next: HALT

--- INTERACTIVE PROMPT ---
Press ENTER to execute next instruction
Press 'q' to quit execution
```



Voici le résultat de l'exécution du program avec test.txt.

Analyse du code:

Le code est fonctionnel et bien structuré. Par contre, on a trouvé une erreur dans la fonction search_and_replace qui avait étée fournie dans l'enoncé à la ligne : int value = (int) (long)values->table[i].value; qu'on a changé à int value = *(int *)values->table[i].value; puisque le type de value est (void *). De plus, on n'a détecté aucune fuite mémoire (via valgrind).

Conclusion :

Le code remplit toutes les spécifications, avec des tests couvrant tous les cas. Les performances sont aussi satisfaisantes grâce à la structure de HashMap qui permet l'accès direct à une case.