

Projet: Réalisation d'un simulateur de CPU

Un processeur est l'unité centrale de traitement (*Central Processing Unit*, CPU) chargée d'exécuter les instructions d'un programme. Il coordonne l'accès à la mémoire, effectue des calculs et prend en charge le contrôle du flux d'exécution des programmes. 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 processeur est constitué de plusieurs éléments fondamentaux :

- **Les registres** : ce sont des emplacements mémoire internes au processeur permettant de stocker temporairement des valeurs nécessaires aux calculs et à l'exécution des instructions. Parmi eux, on trouve les registres généraux (AX, BX, CX, DX), les registres d'indexation et les registres de contrôle (IP, SP, FLAGS).
- **L'unité de contrôle** : elle orchestre l'exécution des instructions en gérant la récupération des instructions, leur décodage et leur exécution.
- **L'unité arithmétique et logique (ALU)** : elle est responsable des opérations mathématiques et logiques (addition, soustraction, comparaisons, etc.).
- **L'accès à la mémoire** : le processeur communique avec la mémoire vive (RAM) pour charger les instructions et stocker temporairement des données. Il utilise différentes stratégies d'adressage pour accéder aux données de manière efficace.

Dans le cadre de ce projet, nous allons implémenter une version simplifiée de ce fonctionnement, en nous concentrant sur un ensemble réduit de registres et de modes d'adressage, tout en conservant les principes fondamentaux du traitement des instructions et de la gestion de la mémoire.

Le projet sera à rendre sur moodle quelques jours avant les soutenances (date à préciser), et les soutenances auront lieu pendant la dernière séance de TD/TP. Sur moodle, vous trouverez un document récapitulatif de ce qui est attendu dans votre rendu. Pour les étudiants qui ne peuvent être présents le jour de la soutenance, vous devez impérativement contacter votre chargé de TD afin de lui fournir un justificatif et prévoir avec elle/lui une autre date de soutenance (au plus près de la semaine de rendu). En effet, les soutenances sont obligatoires, et toute absence injustifiée sera sanctionnée par la note de zéro. Enfin, nous vous rappelons que la note de ce projet fait partie de la note du module, mais ne fait pas partie du contrôle continu. Plus précisément, elle ne peut pas être compensée avec la règle du max et sera conservée en seconde session. **Attention** : Ce projet doit obligatoirement être réalisé en binôme. Si vous ne trouvez pas de binôme, vous devez envoyer un mail à votre chargé de TD pour qu'il vous en trouve un.

Exercice 1 – Implémentation d'une table de hachage générique

Dans cet exercice, nous allons implémenter une table de hachage générique en C. L'objectif est de créer une structure permettant d'associer des clés (chaînes de caractères) à des valeurs de type générique `void*`. Cette flexibilité nous permettra de concevoir un dictionnaire associant des clés à n'importe quel type de structure. La table de hachage utilisera une taille fixe de `TABLE_SIZE = 128` et appliquera un mécanisme de sondage/probing linéaire en cas de collision. Ainsi on utilisera les structures suivantes :

```
1 typedef struct hashentry{
2     char* key;
3     void* value;
4 } HashEntry;
```

```

5 typedef struct hashmap {
6     int size;
7     HashEntry* table;
8 } HashMap;

```

Remarque : quand on supprime un élément dans une table de hachage avec adressage ouvert, on peut utiliser une TOMBSTONE pour éviter d’interrompre les chaînes de probing. Une TOMBSTONE est un élément spécial, distinct de la valeur nulle, qui remplace tout élément supprimé. Contrairement à NULL, elle permet de continuer la recherche et d’assurer l’accessibilité des éléments. Bien sûr, une TOMBSTONE doit être supprimée si un nouvel élément souhaite être inséré à sa position. Ici, on utilisera par exemple un TOMBSTONE définie par :

```

1 #define TOMBSTONE ((void*)-1)

```

Q 1.1 Implémentez une fonction `unsigned long simple_hash(const char *str)` permettant de convertir une chaîne de caractères en un indice dans la table de hachage.

Q 1.2 Écrivez une fonction `HashMap *hashmap_create()` permettant d’allouer dynamiquement une table de hachage et d’initialiser ses cases à NULL.

Q 1.3 Implémentez la fonction `int hashmap_insert(HashMap *map, const char *key, void *value)` permettant d’insérer un élément dans la table de hachage.

Q 1.4 Réalisez une fonction `void *hashmap_get(HashMap *map, const char *key)` permettant de récupérer un élément à partir de sa clé.

Q 1.5 Ajoutez une fonction `int hashmap_remove(HashMap *map, const char *key)` permettant de supprimer un élément de la table de hachage tout en assurant la continuité du sondage linéaire.

Q 1.6 Implémentez une fonction `void hashmap_destroy(HashMap *map)` permettant de libérer toute la mémoire allouée à la table de hachage.

Exercice 2 – Gestion dynamique de la mémoire

Dans cet exercice, nous allons implémenter un gestionnaire de mémoire simple en C. Le gestionnaire de mémoire divise l’espace mémoire en **segments** de taille variable. Chaque segment peut être alloué à un programme ou libéré pour être réutilisé. L’allocation et la libération de segments suivent des étapes bien définies. Dans notre cas, lorsqu’un segment est alloué, il est retiré de la liste des segments libres, puis il est ajouté à une **table de hachage** (correspondant aux segments alloués) en lui associant un nom unique. Lorsqu’un segment est libéré, il est supprimé de la table de hachage et réinséré dans la liste des segments libres (en le fusionnant avec d’autres segments libres adjacents s’il en existe, afin de limiter la fragmentation de la mémoire).

L’implémentation du gestionnaire reposera ainsi sur deux structures essentielles : la structure **Segment** représentant un bloc de mémoire (qui peut être libre ou alloué) et la structure **MemoryHandler** représentant l’état global du gestionnaire de mémoire.

```

1 typedef struct segment {
2     int start; // Position de debut (adresse) du segment dans la memoire
3     int size; // Taille du segment en unites de memoire
4     struct Segment *next; // Pointeur vers le segment suivant dans la liste chainee
5 } Segment;
6

```

```
7 typedef struct memoryHandler {  
8     void **memory; // Tableau de pointeurs vers la memoire allouee  
9     int total_size; // Taille totale de la memoire gereee  
10    Segment *free_list; // Liste chainee des segments de memoire libres  
11    HashMap *allocated; // Table de hachage (nom, segment)  
12 } MemoryHandler;
```

Lors de l'initialisation du gestionnaire, la mémoire est constituée d'un **unique segment libre** couvrant la totalité de l'espace mémoire disponible. Par exemple, pour une mémoire initiale de taille 1024, la liste des segments libres contient un unique segment : `[start=0, size=1024]`.

Q 2.1 Implémentez une fonction `MemoryHandler *memory_init(int size)` permettant d'initialiser le gestionnaire de mémoire.

On souhaite à présent écrire une fonction qui permet de savoir s'il est possible d'allouer un segment commençant à une position donnée. Cette allocation est possible s'il existe un segment libre contenant le segment à allouer, c-à-d s'il existe un segment libre `seg` tel que `seg->start <= start` et `seg->end >= start+size`, où `start` est l'adresse du segment que l'on souhaite allouer et `size` est sa taille.

Q 2.2 Implémentez une fonction `find_free_segment(MemoryHandler* handler, int start, int size, Segment** prev)` qui retourne un tel segment libre s'il en existe, l'argument `prev` permettant alors de récupérer un pointeur sur le segment libre qui le précède dans la liste chaînée `free_list`. Cette fonction retourne NULL dans le cas contraire.

On s'intéresse maintenant à l'allocation de segments. L'allocation d'un segment nommé `X` de taille `size` à l'adresse `start` suit les étapes suivantes :

1. Vérifier qu'un espace mémoire libre suffisant est disponible à l'adresse indiquée (à l'aide de la fonction précédente).
2. Si c'est le cas, créer un nouveau segment `new_seg` de taille `size` à l'adresse `start`, puis l'ajouter à la table de hachage `allocated`, avec `X` comme clé.
3. Remplacer dans `free_list` le segment libre qui contient `new_seg` par deux segments libres : un avant `start` et un après `start+size`.

Exemple : supposons une mémoire initiale de taille **1024**. Si l'utilisateur commence par allouer un segment nommé `"data"` de **100 unités** à l'adresse **200**, l'état devient :

- La table des segments alloués contient le segment `[200, 100]`, avec `"data"` comme clé.
- La liste des segments libres comporte désormais deux segments : `[0, 200]` et `[300, 724]`.

Q 2.3 Implémentez une fonction `int create_segment(MemoryHandler *handler, const char *name, int start, int size)` permettant d'allouer dynamiquement un segment de mémoire de taille `size` à l'adresse mémoire `start`.

La libération d'un segment consiste essentiellement à le retirer de la table de hachage `allocated` pour l'ajouter dans la liste `free_list`. Cet ajout doit cependant se faire de manière à optimiser la réutilisation des espaces libres, c'est-à-dire en fusionnant ce nouveau segment libre avec les éventuels segments libres qui lui sont adjacents. Sur l'exemple précédent, la libération de `[200, 100]` doit ainsi conduire à l'unique segment libre `[0, 1024]` résultant de la fusion de `[200, 100]` avec les segments libres adjacents `[0, 200]` et `[300, 724]`.

Q 2.4 Implémentez une fonction `int remove_segment(MemoryHandler *handler, const char *name)` permettant de libérer un segment de mémoire alloué et de l'ajouter à la liste des segments libres comme décrit ci-dessus.

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

Un **parser** est un programme ou une fonction qui analyse une entrée textuelle pour en extraire une structure exploitable par un ordinateur. Il est souvent utilisé pour interpréter du code source en vue de son exécution ou de sa traduction vers un autre langage. Dans notre cas, nous allons développer un parser capable de comprendre un **pseudo-assembleur**, c'est-à-dire un langage inspiré de l'assembleur mais simplifié pour des besoins pédagogiques ou spécifiques.

Un programme écrit en pseudo-assembleur est organisé en **sections**, qui sont des parties distinctes du code ayant chacune un rôle précis. Les deux sections fondamentales sont :

- **La section .DATA**, qui contient les déclarations de variables et de constantes. Par exemple, l'instruction "X DW 10" permet de déclarer une variable de type DW nommée X et initialisée à la valeur 10.
- **La section .CODE**, qui regroupe les instructions exécutables du programme. Par exemple, l'instruction "MOV AX,7" met la valeur 7 dans AX.

Le parser devra identifier ces sections et extraire les instructions qui y figurent. Pour cela, le parser reposera sur la structure **Instruction** suivante :

```
1 typedef struct {
2     char *mnemonic;    // Instruction mnemonic (ou nom de variable pour .DATA)
3     char *operand1;    // Premier operande (ou type pour .DATA)
4     char *operand2;    // Second operande (ou initialisation pour .DATA)
5 } Instruction;
```

Plus précisément :

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

Le parser devra aussi prendre en compte les **étiquettes/labels**, utilisés pour les sauts conditionnels et les boucles. Plus précisément, une étiquette est un identifiant symbolique permettant de marquer des positions dans le code (l'adresse de la ligne où se trouve le label). Elle est placée en tout début de ligne, suivie du caractère ":", et précède généralement le mnémonique de l'instruction. Par exemple, dans le code suivant, on a deux étiquettes : "loop" et "end".

```
1 .DATA
2 x DW 15
3 .CODE
4 loop: MOV AX,6
5 ADD BX,10
6 end: JMP loop
```

L'étiquette "loop" marque ici la position de la première ligne de code, ce qui permet à l'instruction "JMP loop" de revenir à cette ligne pour répéter les opérations "MOV AX,6" et "ADD BX,10".

Finalement, le résultat du parsing sera stockée dans la structure **PaserResult** suivante :

```
1 typedef struct {
2     Instruction **data_instructions; // Tableau d'instructions .DATA
3     int data_count;                // Nombre d'instructions .DATA
4     Instruction **code_instructions; // Tableau d'instructions .CODE
```

```

5 |     int code_count;           // Nombre d'instructions .CODE
6 |     HashMap *labels;         // labels -> indices dans code_instructions
7 |     HashMap *memory_locations; // noms de variables -> adresse memoire
8 | } ParserResult;

```

Cette structure contient tous les éléments extraits du fichier, c'est-à-dire :

- `data_instructions` : tableau de pointeurs vers des structures `Instruction` correspondant à la section `.DATA`.
- `data_count` : nombre d'éléments dans le tableau `data_instructions`.
- `code_instructions` : tableau de pointeurs vers des structures `Instruction` pour la section `.CODE`.
- `code_count` : nombre d'éléments dans le tableau `code_instructions`.
- `labels` : table de hachage associant les noms d'étiquettes (comme "loop") à leur position dans le tableau `code_instructions`.
- `memory_locations` : table de hachage associant les noms de variables à son adresse séquentielle.

Q 3.1 Implémentez une fonction `Instruction *parse_data_instruction(const char *line, HashMap *memory_locations)` qui permet d'analyser et stocker une ligne de la section `.DATA` d'un programme en pseudo-assembleur. Cette fonction décompose la ligne en trois éléments : le nom de la variable (`mnemonic`), son type (`operand1`, par exemple `DW` ou `DB`) et sa valeur (`operand2`). Ces trois éléments sont séparés par des espaces dans la ligne. Cette fonction associe également chaque variable à son adresse séquentielle dans le dictionnaire `memory_locations`, en tenant compte du nombre d'éléments déclarés. Par exemple, considérons les déclarations suivantes :

```

1 | .DATA
2 | X DW 3
3 | Y DB 4
4 | arr DB 5,6,7,8
5 | z DB 9

```

L'analyse de ces lignes par `parse_data_instruction` devrait produire :

$$\text{data_instructions : } \begin{bmatrix} \text{"X"} & \text{"DW"} & \text{"3"} \\ \text{"Y"} & \text{"DB"} & \text{"4"} \\ \text{"arr"} & \text{"DB"} & \text{"5,6,7,8"} \\ \text{"z"} & \text{"DB"} & \text{"9"} \end{bmatrix} \quad \text{memory_locations : } \left\{ \begin{array}{l} \text{"X"} \rightarrow 0 \\ \text{"Y"} \rightarrow 1 \\ \text{"arr"} \rightarrow 2 \\ \text{"z"} \rightarrow 6 \end{array} \right\}$$

L'adresse associée à chaque variable dans `memory_locations` représente l'emplacement mémoire du premier élément de cette variable. En particulier, `z` est ici associée à l'adresse 6 car `X` occupe l'adresse 0 (un seul élément), `y` occupe l'adresse 1 (un seul élément) et `arr` occupe les adresses 2 à 5 (4 éléments).

Remarque : Dans cet assembleur simplifié, pour le calcul des adresses mémoire, on ne distinguera pas les types `DW` et `DB`. Chaque élément occupe une seule unité de mémoire, indépendamment de son type.

Q 3.2 Implémentez une fonction `Instruction *parse_code_instruction(const char *line, HashMap *labels, int code_count)` qui permet d'analyser et stocker une ligne de la section `.CODE`. Une ligne instruction commence éventuellement par une étiquette, suivie du mnémonique, un espace, puis les deux opérandes séparées par une virgule (le deuxième opérande étant facultatif). Si une étiquette est présente, il faut l'ajouter à la table de hachage `labels`, avec pour valeur `code_count` qui correspond au numéro de la ligne dans `.CODE` (les indices commençant à zéro). Par exemple, considérons la section `.CODE` suivante :

```

1 .CODE
2 loop: MOV AX,6
3 ADD BX,10
4 end: JMP loop

```

L'analyse de ces lignes par `parse_code_instruction` devrait produire ces structures :

$$\text{code_instructions : } \begin{bmatrix} \text{"MOV"} & \text{"AX"} & \text{"6"} \\ \text{"ADD"} & \text{"BX"} & \text{"10"} \\ \text{"JMP"} & \text{"loop"} & \text{""} \end{bmatrix} \quad \text{labels : } \left\{ \begin{array}{l} \text{"loop"} \rightarrow 0 \\ \text{"end"} \rightarrow 2 \end{array} \right\}$$

Q 3.3 Implémentez une fonction `ParserResult *parse(const char *filename)` qui analyse un fichier assembleur complet en identifiant les sections `.DATA` et `.CODE` et en traitant chaque ligne de la manière appropriée. Elle doit construire une structure complète `ParserResult` qui contient tous les éléments extraits du fichier. La fonction doit gérer l'allocation dynamique des tableaux d'instructions (`data_instructions` et `code_instructions`) et redimensionner ces tableaux selon les besoins. Par exemple, pour un fichier contenant

```

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

```

la fonction `parse` doit créer une structure `ParserResult` complète :

$$\begin{aligned} \text{data_instructions : } & \begin{bmatrix} \text{"x"} & \text{"DW"} & \text{"42"} \\ \text{"y"} & \text{"DB"} & \text{"10"} \\ \text{"arr"} & \text{"DB"} & \text{"20,21,22,23"} \end{bmatrix} & \text{code_instructions : } & \begin{bmatrix} \text{"MOV"} & \text{"AX"} & \text{"x"} \\ \text{"ADD"} & \text{"AX"} & \text{"y"} \\ \text{"JMP"} & \text{"loop"} & \text{""} \end{bmatrix} \\ \text{labels : } & \left\{ \begin{array}{l} \text{"start"} \rightarrow 0 \\ \text{"loop"} \rightarrow 1 \end{array} \right\} & \text{memory_locations : } & \left\{ \begin{array}{l} \text{"x"} \rightarrow 0 \\ \text{"arr"} \rightarrow 1 \\ \text{"y"} \rightarrow 5 \end{array} \right\} \\ \text{data_count} = 3 & & \text{code_count} = 3 & \end{aligned}$$

Q 3.4 Tester votre fonction `parse` sur l'exemple de la question précédente.

Q 3.5 Écrivez une fonction `void free_parser_result(ParserResult *result)` permettant de supprimer un élément de type `ParserResult`.

La suite du projet sera disponible la semaine du 17 mars.