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 récapitulant 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 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 trouver 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;
```

```c
typedef struct hashmap {
    int size;
    HashEntry* table;
} 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 :

```c
#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.

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

```

```c
typedef struct memoryHandler {
    void **memory; // Tableau de pointeurs vers la memoire allouee
    int total_size; // Taille totale de la memoire geree
    Segment *free_list; // Liste chainee des segments de memoire libres
    HashMap *allocated; // Table de hachage (nom, segment)
} 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 :

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

```
.DATA
x DW 15
.CODE
loop: MOV AX,6
ADD BX,10
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 stucture `PaserResult` suivante :

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

```
    int code_count;                 // Nombre d'instructions .CODE
    HashMap *labels;                // labels -> indices dans code_instructions
    HashMap *memory_locations;      // noms de variables -> adresse memoire
} 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 :

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

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

$$\texttt{data\_instructions} : \begin{bmatrix} \text{"X"} & \text{"DW"} & \text{"3"} \\ \text{"y"} & \text{"DB"} & \text{"4"} \\ \text{"arr"} & \text{"DB"} & \text{"5,6,7,8"} \\ \text{"z"} & \text{"DB"} & \text{"9"} \end{bmatrix} \quad \texttt{memory\_locations} : \begin{Bmatrix} \text{"X"} \to 0 \\ \text{"y"} \to 1 \\ \text{"arr"} \to 2 \\ \text{"z"} \to 6 \end{Bmatrix}$$

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 :

```
.CODE
loop: MOV AX,6
ADD BX,10
end: JMP loop
```

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

$$\texttt{code\_instructions} : \begin{bmatrix} \text{"MOV"} & \text{"AX"} & \text{"6"} \\ \text{"ADD"} & \text{"BX"} & \text{"10"} \\ \text{"JMP"} & \text{"loop"} & \text{""} \end{bmatrix} \quad \texttt{labels} : \begin{Bmatrix} \text{"loop"} \to 0 \\ \text{"end"} \to 2 \end{Bmatrix}$$

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

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

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

$$\texttt{data\_instructions} : \begin{bmatrix} \text{"x"} & \text{"DW"} & \text{"42"} \\ \text{"y"} & \text{"DB"} & \text{"10"} \\ \text{"arr"} & \text{"DB"} & \text{"20, 21, 22, 23"} \end{bmatrix} \quad \texttt{code\_instructions} : \begin{bmatrix} \text{"MOV"} & \text{"AX"} & \text{"x"} \\ \text{"ADD"} & \text{"AX"} & \text{"y"} \\ \text{"JMP"} & \text{"loop"} & \text{""} \end{bmatrix}$$

$$\texttt{labels} : \begin{Bmatrix} \text{"start"} \to 0 \\ \text{"loop"} \to 1 \end{Bmatrix} \quad \texttt{memory\_locations} : \begin{Bmatrix} \text{"x"} \to 0 \\ \text{"arr"} \to 1 \\ \text{"y"} \to 5 \end{Bmatrix}$$

$\texttt{data\_count} = 3 \qquad \texttt{code\_count} = 3$

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`.

Exercice 4 – Allocation d'un segment de données

Dans cet exercice, nous allons implémenter les mécanismes permettant d'allouer un segment en fonction des déclarations de variables récupérées par le parser. Le nom de ce segment de données sera "DS" (pour "data segment"). Nous allons aussi nous intéresser au stockage de ces données, à leur récupération et à leur affichage. Toutes ses opérations seront réalisées par notre processeur simulé (CPU) composé des élements suivants :

```c
typedef struct {
    MemoryHandler* memory_handler; // Gestionnaire de memoire
    HashMap *context;              // Registres (AX, BX, CX, DX)
} CPU;
```

Un CPU est donc composé d'un **gestionnaire de mémoire** (`MemoryHandler`) permettant d'allouer et de gérer la mémoire du programme et d'une **table de hachage** (`HashMap`) permettant de stocker les valeurs des registres (emplacements mémoire internes au processeur). Pour simplifier, nous ne considérons ici que les registres généraux `AX, BX, CX, DX` et nous supposerons qu'ils correspondent à des entiers.

Q 4.1 Implémentez une fonction `CPU *cpu_init(int memory_size)` permettant d'initialiser le processeur simulé en :

- Allouant dynamiquement une structure `CPU`.
- Initialisant son gestionnaire de mémoire de taille `memory_size`.
- Créant une table de hachage contenant les registres généraux `AX`, `BX`, `CX`, `DX` en leur associant la valeur zéro.

Q 4.2 Implémentez une fonction `void cpu_destroy(CPU *cpu)` permettant de libérer toutes les ressources allouées par le processeur simulé, notamment les registres, la table de hachage et le gestionnaire de mémoire.

On s'intéresse maintenant au stockage et à la récupération des données.

Q 4.3 Implémentez une fonction `void* store(MemoryHandler *handler, const char *segment_name, int pos, void *data)` permettant de stocker une donnée `data` à la position `pos` de `segment_name`. Cette fonction devra d'abord vérifier que `segment_name` est bien un segment alloué, et que `pos` ne dépasse pas la taille du segment. Si c'est le cas, la donnée est insérée dans le champs `memory` à la bonne position (tenant compte de la position de début du segment). Par exemple, si le segment alloué est $[200, 100]$ et que l'on souhaite ajouter la donnée à la position 50 de ce segment, la donnée sera stockée dans `memory` à la position 250.

Q 4.4 Implémentez une fonction `void* load(MemoryHandler *handler, const char *segment_name, int pos)` permettant de récupérer la donnée stockée à la position `pos` de `segment_name`.

L'allocation des données par le CPU consiste en les étapes suivantes :

1. **Calcul de l'espace nécessaire au stockage des variables.** Dans l'exemple de la question 3.1, les variables `x`, `y` et `z` ont besoin de 1 emplacement chacune, alors que la variable `arr` a besoin de 4 emplacements. Au total, il faudra donc déclarer un segment de taille 7. Dans le cadre de ce projet, on considère que les données à stocker sont des entiers (pour simplifier).
2. **Déclaration d'un segment nommé `DS` dans le MemoryHandler.** Sur l'exemple, après déclaration du segment, le `MemoryHandler` contient un seul segment libre ([7,1017] si la taille initiale est 1024) et un seul segment alloué correspondant au segment de données ({`DS` : [0,7]}).
3. **Remplissage du segment de données avec les valeurs déclarées dans la section `.DATA` du code.** Sur notre exemple, le tableau **memory** évolue ainsi :
   [NULL, NULL, ..., NULL] $\longrightarrow$ $[p1, p2, p3, p4, p5, p6, p7, \texttt{NULL}, \ldots, \texttt{NULL}]$
   où `p1` est un pointeur sur un entier contenant 3 (la valeur de x), `p2` est un pointeur sur un entier contenant 4 (la valeur de y), `p3` est un pointeur sur un entier contenant 5 (la valeur du premier élément de `arr`), etc.

Q 4.5 Implémentez une fonction `void allocate_variables(CPU *cpu, Instruction** data_instructions, int data_count)` permettant d'allouer dynamiquement le segment de données en fonction des déclarations récupérées par le parser.

Q 4.6 Implémentez une fonction `void print_data_segment(CPU *cpu)` permettant d'afficher le contenu du segment de données (nom "DS").

Exercice 5 – Expressions régulières et résolution d'adressage

Dans cet exercice, nous allons utiliser des expressions régulières (*regex*) en C pour identifier les différents modes d'adressage en pseudo-assembleur. Chaque mode d'adressage détermine comment une opérande doit être interprétée et où se trouve la valeur manipulée. Dans le cadre de ce projet, nous nous intéressons aux modes d'adressage suivants :

- **Adressage immédiat** : l'opérande est une valeur numérique directement contenue dans l'instruction. Par exemple, la valeur 42 est immédiate dans l'instruction `"MOV AX,42"`.
- **Adressage par registre** : l'opérande est le nom d'un registre du processeur. Par exemple, `BX` est le registre source de l'instruction `"MOV AX,BX"`.
- **Adressage direct** : l'opérande spécifie directement l'adresse en mémoire où se trouve la valeur. Par exemple, on écrit `"MOV AX,[5]"` pour indiquer que la valeur à charger se trouver à la position 5 du segment de données.
- **Adressage indirect par registre** : l'opérande spécifie un registre qui contient l'adresse mémoire où se trouve la valeur. Par exemple, on écrit `"MOV BX,[AX]"` pour indiquer que la valeur à charger est celle située à l'adresse contenue dans le registre `AX`.

Les expressions régulières permettent de reconnaître et d'extraire des motifs spécifiques dans du texte. Elles sont couramment utilisées pour valider des entrées utilisateur, rechercher des motifs dans un fichier ou transformer des chaînes de caractères. Dans le cadre de ce projet, notre objectif est de formuler des expressions régulières permettant d'identifier le mode d'adressage. Par exemple, pour l'adressage immédiat, au lieu de tester si chaque caractère de l'opérande est bien un chiffre entre 0 et 9, il s'agit d'écrire une expression régulière (un motif) caractérisant la syntaxe d'un nombre de manière générale.

Avant de chercher à détecter les différents modes d'adressage, voyons quelques éléments fondamentaux des expressions régulières utiles à leur construction :

- `"[0-9]"` : Correspond à un chiffre.
- `"[A-Z]"` : Correspond à une lettre majuscule.
- `"[a-zA-Z]"` : Correspond à n'importe quelle lettre, majuscule ou minuscule.
- `"A"` : Correspond exactement au caractère `'A'`.
- `^` : Indique le début de la chaîne de caractères.
- `$` : Indique la fin de la chaîne de caractères.
- `+` : Quantificateur indiquant au moins une occurrence du motif qui le précède.
- `*` : Quantificateur indiquant zéro ou plusieurs occurrences du motif qui le précède.
- {`n`} : Indique exactement `n` occurences du motif qui le précède.
- `()` : Groupe des expressions pour les traiter comme une unité.
- `|` : Opérateur logique "OU" permettant d'associer plusieurs motifs alternatifs.

Si on souhaite que le motif contiennent un caractère spécial (comme `[`, `)`, `*`, `+`, etc.), il faut utiliser un échappement. Par exemple, l'expression régulière `^(\+33|0)[1-9][0-9]{8}$` permet de spécifier le

format d'un numéro de téléphone français, qui commence soit par +33 soit par 0, suivi d'un chiffre entre 1 et 9, puis de 8 autres chiffres. Autre exemple, l'expression régulière `^[A-Z][a-z]*( [A-Z][a-z]*)*$` permet de spécifier le format d'un nom de famille, qui commence par une majuscule, suivie de minuscules, et qui peut être suivie par d'autres mots au même format (précédés par un espace).

Nous devons maintenant utiliser ces constructions de base pour concevoir des expressions régulières qui correspondent aux différents types d'adressage en assembleur. En langage C, vous pouvez tester des expressions régulières à l'aide de la fonction suivante :

```c
int matches(const char *pattern, const char *string) {
    regex_t regex;
    int result = regcomp(&regex, pattern, REG_EXTENDED);
    if (result) {
        fprintf(stderr, "Regex compilation failed for pattern: %s\n", pattern);
        return 0;
    }
    result = regexec(&regex, string, 0, NULL, 0);
    regfree(&regex);
    return result == 0;
}
```

Cette fonction compile le motif, vérifie sa correspondance avec une chaîne, libère les ressources, puis retourne un booléen indiquant si une correspondance a été trouvée.

Adressage immédiat

L'adressage immédiat traite des constantes littérales (comme "42") contrairement aux autres modes d'adressage qui manipulent des adresses mémoire. Pour que toutes les fonctions d'adressage retournent uniformément des pointeurs void*, nous devons allouer de la mémoire pour ces valeurs. Pour cela, on va utiliser un pool de constantes, qui stocke chaque valeur une seule fois dans une table de hachage : la clé est la chaîne de caractères (ex : "42") et la valeur est un pointeur vers la mémoire allouée. Quand une valeur immédiate est rencontrée, on vérifie d'abord si elle existe déjà dans le pool avant de créer une nouvelle allocation. Cela évite les duplications tout en maintenant l'uniformité de l'interface.

Pour implémenter ce pool de constantes, on modifie la structure CPU de la manière suivante :

```c
typedef struct CPU {
    // Champs existants...
    HashMap *constant_pool;  // Table de hachage pour stocker les valeurs immédiates
} CPU;
```

Q 5.1 Mettez a jour les fonctions `CPU *cpu_init(int memory_size)` et `void cpu_destroy(CPU *cpu)` pour prendre en compte le constant pool.

Q 5.2 Implémentez une fonction `void *immediate_addressing(CPU *cpu, const char *operand)` permettant de traiter l'adressage immédiat. La fonction doit vérifier si l'opérande correspond à ce mode d'adressage (en utilisant une regex), stocker cette valeur dans le constant pool si elle n'y est pas déjà, et retourner un pointeur vers cette valeur stockée.

Adressage par registre

L'adressage par registre est un mode d'adressage où l'opérande est le nom d'un registre du processeur. Dans notre cas, les registres existants sont `AX`, `BX`, `CX` et `DX`.

Q 5.3 Implémentez une fonction `void *register_addressing(CPU *cpu, const char *operand)` permettant de traiter l'adressage par registre. Cette fonction doit vérifier si l'opérande fourni correspond au format d'un nom de registre (à l'aide d'une regex), puis vérifier si ce registre existe dans le contexte du CPU. Si le registre existe, la fonction retourne un pointeur vers la valeur du registre, ce qui permettra de lire ou modifier son contenu.

Adressage direct

L'adressage direct par mémoire est un mode d'adressage où l'opérande spécifie directement l'adresse en mémoire à laquelle accéder. L'opérande est alors de la forme `"[i]"` où `i` est sa position dans le segment de données.

Q 5.4 Implémentez une fonction `void *memory_direct_addressing(CPU *cpu, const char *operand)` permettant de traiter l'adressage direct par mémoire. Cette fonction doit vérifier si l'opérande fourni correspond au format d'une adresse mémoire directe (à l'aide d'une regex), extraire l'adresse numérique, puis accéder à la valeur stockée à cette adresse dans le segment de données. Elle retourne un pointeur vers cette valeur, ce qui permettra de la lire ou de la modifier dans le cadre d'une instruction.

Addressage indirect par registre

L'adressage indirect par registre est un mode d'adressage où l'opérande spécifie un registre qui contient l'adresse mémoire à laquelle accéder. L'opérande est alors de la forme `"[XX]"` où `XX` est le nom d'un registre du processeur.

Q 5.5 Implémentez une fonction `void *register_indirect_addressing(CPU *cpu, const char *operand)` permettant de traiter l'adressage indirect par registre. Cette fonction doit vérifier si l'opérande correspond au format d'un registre entre crochets (à l'aide d'une regex), extraire le nom du registre, récupérer sa valeur qui servira d'adresse mémoire, puis accéder à la valeur stockée à cette adresse dans le segment de données. Elle retourne un pointeur vers cette valeur, ce qui permettra de la lire ou de la modifier dans le cadre d'une instruction.

Les fonctions que nous venons d'écrire permettent de résoudre différents types d'adressage, en retournant un pointeur sur la valeur indiquée par l'opérande. Ceci permet notamment de simuler des instructions en pseudo-assembleur avec des fonctions C qui prennent en argument ces pointeurs.

Q 5.6 Implémentez une fonction `void handle_MOV(CPU* cpu, void* src, void* dest)` permettant de simuler le comportement de l'instruction MOV en pseudo-assembleur. Cette fonction prend un pointeur vers l'emplacement de la valeur source et un pointeur vers l'emplacement de destination, puis copie la valeur depuis la source vers la destination.

Q 5.7 En utilisant la fonction suivante (permettant de créer une structure CPU et d'initialiser les valeurs des registres), écrivez un main permettant de tester les 4 types d'addressage en utilisant la fonction `handle_MOV`.

```
1 CPU* setup_test_environment() {
2     // Initialiser le CPU
3     CPU *cpu = cpu_init(1024);
4     if (!cpu) {
5         printf("Error: CPU initialization failed\n");
6         return NULL;
7     }
8
```

```
    // Initialiser les registres avec des valeurs specifiques
    int *ax = (int *)hashmap_get(cpu->context, "AX");
    int *bx = (int *)hashmap_get(cpu->context, "BX");
    int *cx = (int *)hashmap_get(cpu->context, "CX");
    int *dx = (int *)hashmap_get(cpu->context, "DX");

    *ax = 3;
    *bx = 6;
    *cx = 100;
    *dx = 0;

    // Creer et initialiser le segment de donnees
    if (!hashmap_get(cpu->memory_handler->allocated, "DS")) {
        create_segment(cpu->memory_handler, "DS", 0, 20);

        // Initialiser le segment de donn es avec des valeurs de test
        for (int i = 0; i < 10; i++) {
            int *value = (int *)malloc(sizeof(int));
            *value = i * 10 + 5; // Valeurs 5, 15, 25, 35...
            store(cpu->memory_handler, "DS", i, value);
        }
    }
    printf("Test environment initialized.\n");
    return cpu;
}
```

Q 5.8 Implémentez une fonction `void *resolve_addressing(CPU *cpu, const char *operand)` permettant d'identifier automatiquement le mode d'adressage d'un opérande et de résoudre sa valeur. Cette fonction doit essayer successivement chaque mode d'adressage implémenté précédemment et retourner le résultat du premier mode applicable.

La suite du projet sera disponible la semaine du 31 mars.