Projet LU2IN006: Réalisation d'un simulateur de CPU

PROKHOROV Vadim

TERMOS Haya

Tables des matières

1. Description du projet

Ce projet consiste à simuler une version simplifiée d'un CPU (Central Processing Unit = Unité Centrale de Traitement) en langage C.
Ce processeur simplifié est constitué de plusieurs élements :

Les registres: mémoire interne au CPU, utilisée pour stocker temporairement des opérandes, résultats intermédiaires et adresses.
Ces registres vont êtres représentés par des clés (nom du registre) associés à des variables contenant un pointeur vers la valeur stockée par le registre.
Ils vont être stockés dans une table de hachage générique.

L'unité arithmétique et logique (ALU): effectue les opérations mathématiques et logiques (addition, comparaison, etc.)

L'accès à la mémoire: le CPU communique avec la mémoire vive (RAM) qu'on va représenter par une structure englobant des tables de hachage et des listes chaînées.
Pour cela, il utilise différentes stratégies d'adressage pour accéder aux données.
Cet adressage va être décodé à l'aide de la correspondance avec des expressions régulières.

L'unité de contrôle: dirige l'éxecution des instructions en gérant la récupération des instructions, leur décodage et leur éxecution qui vont être effectués à travers plusieurs fonctions qui s'appellent entre elles pour gérer les differentes étapes.

Le fonctionnement du CPU repose alors sur une suite d'instructions, constituant un programme écrit en pseudo-assembleur, qu'on va extraire de fichiers `.txt` à l'aide des fonctions `scan` de la bibliothèque `stdio.h`.
Ce genre de programme comporte 2 sections fondamentales:
- La section **.DATA** qui contient la déclaration de variables et de constantes.
- La section **.CODE** qui contient les instructions du programme.

2. Méthodes d'utilisation du projet, tests disponibles

Le code du projet peut être utilisé aussi bien à des fins de démonstration, en utilisant les tests préparés par les créateurs, que dans des conditions réelles de programmation dans le langage pseudo-assembleur, dont la syntaxe sera

discutée dans la section « Conception de programme en Pseudo-Assembleur » de ce document.

2.1. Programmes des testes intermédiaires

Le projet offre à l'utilisateur un choix de 6 programmes de test intermédiaires qui montrent le fonctionnement des différents modules utilisés dans le projet.

Liste des programmes d'essai, méthodes de compilation, but de l'exécution :

- **main1** : Méthodes et fonctions des tables de hachage HashMap

  ```
  >> make main1
  >> ./main1
  ```

- **main2** : Travail avec les segments de mémoire

  ```
  >> make main2
  >> ./main2
  ```

- **main3** : Travail avec pseudo-assembleur parser

  ```
  >> make main3
  >> ./main3
  ```

- **main4** : Module CPU, fonctions et méthodes de travail

  ```
  >> make main4
  >> ./main4
  ```

- **main5** : Ce programme permet à l'utilisateur de se familiariser avec la méthode **MOV** dans un environnement spécialement préparé.

  ```
  >> make main5
  >> ./main5
  ```

- **main6** : Test d'exécution du code d'un projet de programme en pseudo-assembleur avec indication des étapes intermédiaires

  ```
  >> make main6
  >> ./main6
  ```

2.2. Utiliser le code du projet pour des tâches réelles

Pour utiliser le projet dans un environnement réel, l'utilisateur est encouragé à utiliser l'une des méthodes suivantes d'exécution du code en pseudo-assembleur :

- **Interface graphique d'exécution** L'interface graphique du programme, située dans le fichier maincpu, permet à l'utilisateur d'exécuter un code pseudo-assembleur à partir de fichiers .txt dans un format étape par étape.

  Il est intéressant de l'utiliser pour une sélection flexible des paramètres, ainsi que pour voir le contenu de chaque objet à chaque étape de l'exécution du programme.

  Cette interface permet à l'utilisateur de sélectionner indépendamment chacune des étapes de l'exécution du programme, depuis l'analyse jusqu'au lancement du programme.

  Il faut également noter que lors de l'utilisation de l'interface graphique, l'exécution du code en pseudo-assembleur se fait progressivement, ligne par ligne, sur ordre de l'utilisateur. Cela permet de voir clairement l'impact des instructions exécutées sur le contenu de la mémoire de l'unité centrale virtuelle.

  Démarrer l'interface graphique :

  ```
  >> make maincpu
  >> ./maincpu
  ```

- **L'exécuteur du code : executor.c** Ce module permet d'exécuter directement un programme en langage pseudo-assembleur et d'afficher immédiatement le résultat final de son fonctionnement.

  L'exécuteur combine un compilateur (analyse le code en instructions, interpole les constantes) et un exécuteur du programme (stocke les constantes et les instructions en mémoire, exécute le programme).

  Démarrer l'exécuteur :

  ```
  >> make executor
  >> ./executor filename.txt
  ```

  où **filename.txt** est un fichier contenant des instructions en pseudo-assembleur et se conformant clairement à la syntaxe requise décrite à la section 4.

- **Code de test en pseudo-assembleur** L'utilisateur a le choix entre 4 programmes de test qui peuvent être utilisés comme exemples de problèmes réels.

  ▸ **test.txt:**

Effectue une multiplication du contenu du registre AX par BX en créant une boucle qui exécute AX fois l'ajout de BX à l'accumulateur. Le résultat est écrit dans le registre CX

- **test2.txt:**

  Démontre le fonctionnement de la pile, le déplacement du pointeur d'exécution, ainsi que la méthode d'allocation de la mémoire dynamique et son utilisation (mise en place de valeurs). Il applique ensuite l'effacement de la mémoire dynamique et la terminaison du programme.

- **setup_env_test.txt:**

  Utilisé dans main5 pour démontrer les principes de la fonction MOV, qui déplace des valeurs en mémoire.

- **test_pseudo_asm.txt:**

  Création d'objets en mémoire, déplacement de valeurs

3. Description des structures utilisées

`HashEntry`:élément dans la table de hachage constitué d'une clé et d'une valeur associée à cette clé

```
typedef struct hashentry {
    char *key; //clé
    void *value; //valeur associée
} HashEntry;
```

`HashMap`: table de hachage avec résolution de collisions par probing linéaire. C'est une table de `HashEntry` directement et non pas de pointeurs vers `HashEntry`.

```
typedef struct hashmap {
    int size; //taille de la table
    HashEntry *table; //table de HashEntry
} HashMap;
```

`Segment`: segment, qui est définie par sa position de début et sa taille, dans la mémoire (à laquelle le CPU va accéder pour stocker et extraire des données) Chaque segment a un pointeur vers un éventuel prochain, on a donc une liste chaînée de segments.

```
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;
```

MemoryHandler: gestionnaire de mémoire, représente l'état global de la mémoire. Contient la taille totale de la mémoire, une liste chaînée des segments de mémoire libre, et une table de hachage des segments alloués.
La mémoire elle-même prend la forme d'un tableau de pointeurs génériques `void*` (`memory`) pour stocker nos données génériques.

```
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;
```

Instruction: stocke les composantes (`char*`) d'une instruction, extraites de la chaîne de caractères contenant cette instruction.
Ces composantes diffèrent selon si c'est une instruction .DATA ou .CODE .

```
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;
```

ParserResult: Résultat du Parser qui est une série de fonctions qui vont lire les instructions et en extraire ce qui est exploitable par le CPU
Ce résultat va contenir le nombre d'instructions et un tableau d'instructions pour les 2 sections du programme.
En ce qui concerne uniquement .CODE, il contient une table de hachage `HashMap` pour les labels (qui associe le label à l'indice de son instruction dans le tableau d'instructions).
Et pour .DATA il contient une table de hachage qui associe le nom d'une variable à l'adresse du premier élément de cette variable.

```
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;
```

`CPU`: le CPU qui est constitué de ses registres stockés dans la `HashMap` nommée `context`, d'une autre `HashMap` pour stocker les constantes vers lesquelles on a déjà créé des pointeurs pour éviter d'allouer plusieurs espaces mémoires pour une même valeur (associe la constante à son adresse dans la mémoire), et finalement d'un `MemoryHandler` auquel le cpu a accès.

```
typedef struct {
    MemoryHandler *memory_handler; // Gestionnaire de memoire
    HashMap *context; // Registres (AX, BX, CX, DX)
    HashMap *constant_pool; // Table de hachage pour stocker les valeurs
immediates
} CPU;
```

4. Conception de programme en Pseudo-Assembleur

Un point important de la description de ce projet est qu'il est adapté à un langage spécifique, le pseudo-assembleur, dont les opérations peuvent être traitées par le module CPU. Vous trouverez ci-dessous les conditions requises pour la syntaxe du fichier d'entrée au format .txt, ainsi que l'explication de ses méthodes et la démonstration de leur fonctionnement.

4.1. Partie .DATA

La partie .DATA est destinée à l'initialisation et à l'attribution de valeurs aux variables.

Voici un exemple de partie .DATA contenant l'initialisation des variables **x**,**y** au format valeur entière, qui est abrégé **DW**, **DB** dans le projet. Pour simplifier le travail avec les données, ces deux types représentent des nombres entiers. De plus, deux tableaux d'entiers sont initialisés : **list**, **arr**.

Chaque variable est spécifiée à partir d'un nouveau côté dans le format suivant:

`>> nom_de_la_variable  type_de_données  valeur(s)`

REMARQUE IMPORTANTE : Il convient de noter que les valeurs que le tableau contiendra après l'initialisation sont écrites séquentiellement, séparées par des virgules, sans espace.

Exemple de code .DATA en Pseudo-Assembleur:

```
.DATA
x DW 42
arr DB 20,21,22,23
```

```
list DB 10,15,2345654,78438,20
y DB 10
```

4.2. Partie .CODE

La partie .CODE est destinée à toutes les opérations de code : attribution de valeurs, addition, opérations logiques, allocation de mémoire et effacement etc. Plus de détails sur chacune des opérations disponibles dans la partie « Les méthodes et leur fonctionnement ».

Le pointeur .CODE suit sans saut de ligne la dernière des variables désignées ou le pointeur .DATA s'il n'y a pas de variables

Exemple de code en Pseudo-Assembleur:

```
.DATA
a DW 1
b DB 2
arr DB 3,4,5,6
.CODE
MOV AX d
MOV BX [b]
loop: ADD CX BX
ADD DX 1
CMP AX DX
JNZ loop
```

REMARQUE IMPORTANTE : Chaque opération suit la précédente sur une nouvelle ligne. Toutes les opérations sont effectuées de manière séquentielle, sauf si le programme prévoit un autre comportement. Les opérateurs s'écrivent avec des virgules et sans espace: `"BX,2"`

4.3. Les méthodes et leur fonctionnement

- **`MOV dest,src`** : Transfert d'une valeur de **src** à **dest**

  dest ← src

- **`ADD dest,src`** : Addition des valeurs **src** et **dest**, enregistrement du résultat dans **dest**

  dest ← (src+dest)

- **`CMP dest,src`** : L'opération effectue une soustraction logique de **dest - src** (sans modifier les valeurs) et met à jour les drapeaux:
  - ZF = 1 si et seulement si dst = src
  - SF = 1 si et seulement si dst < src

- **`JMP address`** : Opération de transition vers la ligne spécifiée en modifiant la valeur de l'indicateur IP qui est responsable du flux d'exécution.

  IP ← adresse

- **`JZ address`** : Opération de transition vers la ligne spécifiée si l'opération précédente s'est terminée par un résultat nul

  Si **ZF = 1** alors **IP ← adresse**

- **`JNZ address`** : Opération de transition vers la ligne spécifiée si l'opération précédente s'est terminée par un résultat non-nul

  Si **ZF = 0** alors **IP ← adresse**

- **`HALT`** : Termine l'exécution du programme en déplaçant le pointeur de l'exécuteur IP à la fin du segment de code.

- **`PUSH src`** : L'opération place la valeur **src** au sommet de la pile (Stack segment)

- **`POP dest`** : Supprime la valeur supérieure de la pile (Stack segment), en l'affectant à **dest**

- **`ALLOC`** : Cette opération permet d'effectuer une allocation dynamique de extra segment (ES) selon les règles suivantes:

  - **size(ES) = AX**
  - **stratégie d'allocation (0,1,2) = BX**

    PRÉREQUIS :
    – AX < MEMORY_SIZE
    – BX = 0,1,2

    (Plus d'informations sur la stratégie d'allocation dans la description de la fonction find_free_address_strategy())

    Exemple d'utilisation:

```
MOV AX, 10        <- Choix de la taille de extra segment (ES)
MOV BX, 0         <- Choix d'une stratégie d'allocation
ALLOC             <- Allocation dynamique de extra segment (ES)
MOV AX,5          <- Affectation de la valeur 5 au AX
MOV [ES:AX],10    <- Affectation 10 au 6ème élément de ES
```

- **`FREE`** : Libère le segment ES alloué dynamiquement

Pour plus d'informations sur les types d'opérandes possibles pour les instructions, voir la section décrivant les différentes fonctions d'adressage.

- **Labels :**

  Les étiquettes (labels), les commandes d'apparence «label:», sont un outil de contrôle du pointeur d'exécution, et permettent de donner des commandes pour passer d'une ligne à l'autre.
  - **Syntaxe des étiquettes** : une étiquette est un mot sans espace suivi de deux points. **Tout autre format entraînera une erreur de compilation**
  - **Déclaration et application** : L'étiquette est placée au début de la ligne et peut précéder l'instruction.

    ```
    line 6 >> loop: MOV AX,6
    line 7 >> ADD DX,AX
    line 8 >> JMP loop
    ```

    À son tour, l'appel du nom de l'étiquette en tant qu'opérande 1 des commandes **JMP, JZ, JNZ** déplacera le pointeur d'exécution **IP** à la ligne où l'étiquette a été spécifiée.

5. Description des fonctions principales

5.1. Fonctions hashmap

- **`unsigned long simple_hash(const char *str)`**

Fonction de hachage de la clé (`char*`): prend la somme des valeurs ASCII de chaque caractère de la chaîne, et la multiplie par le nombre d'or inverse. Elle prend ensuite la partie décimale du résultat et le multiplie par la taille de la table de hachage: après un cast en `int` on obtient un entier entre 0 et `taille -1` qui représente l'indice haché de cette clé.

- **`unsigned long simple_hash(const char *str)`**

Fonction de hachage de la clé (`char*`): prend la somme des valeurs ASCII de chaque caractère de la chaîne, et la multiplie par le nombre d'or inverse. Elle prend ensuite la partie décimale du résultat et le multiplie par la taille de la table de hachage: après un cast en `int` on obtient un entier entre 0 et `taille -1` qui représente l'indice haché de cette clé.

- **`int hashmap_insert(HashMap *map, const char* key, void* value)`**

Fonction d'insertion d'un élément (clé et son value) dans la table de hachage à l'aide de la méthode de probing linéaire pour la résolution de collisions
Elle commence à partir de l'indice haché calculé par la fonction `simple_hash` et si la case est déjà occupée, elle essaie la prochaine jusqu'à trouver une case vide. Cela se fait de façon modulaire: quand on arrive à la dernière case du tableau, on le reparcours depuis le début jusqu'à retomber sur la case de l'indice haché de départ ce qui signifierait que la table est remplie.
L'indice de la case de l'élément sera alors égal à h(c)+i avec h la fonction de hachage, c la clé, et i le nombre de probes.

- **`void * hashmap_get(HashMap *map, const char * key)`**

Fonction qui retourne le value de la clé en argument. Fait les mêmes étapes que `hashmap_insert` en vérifiant (au lieu de la disponibilité de la case) si la clé dans la case est égale à celle de l'argument.
À noter qu'il faut mettre une condition sur les TOMBSTONE (marqueur qui permet la distinction entre une case vide et une case qui a été occupée puis vidée) pour bien gérer le cas d'un élément qui a été crée mais supprimé.

- **`int hashmap_remove(HashMap *map,const char *key)`**

Fonction qui supprime un élément de la table de hachage. Fait les mêmes étapes que `hashmap_get` et libère la clé avant de la remplacer par un TOMBSTONE. À noter que cette fonction met le champ value de l'élément à NULL sans le libérer, au cas où on l'utilisait autre part. Pour bien libérer le value et éviter les fuites mémoires, en appelant cette fonction, il faudra donc récupérer le value avec la fonction `hashmap_get` et le libérer, ou bien appeler une fonction qui le fait.

5.2. Fonctions memory handler et ses segments

- **`Segment* find_free_segment(MemoryHandler* handler, int start, int size, Segment ** prev)`**

Fonction qui renvoie le segment libre et de bonne taille et position dans la liste des segments libres, où on pourra allouer le segment de taille et de position de début précisés en argument. Elle retourne `NULL` si la section demandé n'est pas disponible.
Elle met également le segment précédent au segment retourné pour pouvoir le pointer ultérieurement vers l'éventuel espace libre après le segment alloué dans `free_list`.
Pour faire tout cela, elle parcourt la liste (en sauvegardant le précedent de

chaque segment courant dans la boucle) jusqu'à retrouver le segment ou se trouve la position `start` demandée, et le retourne s'il est assez large, en stockant sans précédant dans le pointeur dédié à cet effet dans les paramètres.

- **`int create_segment(MemoryHandler * handler,const char *name,int start, int size)`**

Fonction qui crée le segment à la position `start` et de taille `size` demandée dans le `memoryhandler`.
Elle prend le segment libre initial dans lequel on a alloué et si il y a de l'espace libre avant, elle merge cet espace avec le precedent dans la `free_list`, elle regarde aussi s'il y a de l'espace libre après et le merge avec le suivant dans la liste, puis elle finit par allouer le segment en le créant et en l'ajoutant à la hashmap `allocated`

- **`int remove_segment(MemoryHandler * handler, const char *name)`**

Fonction qui supprime un segment. Elle le supprime de la hashmap `allocated` et le libère.
Elle merge cet espace libéré avec le précédent (éventuel) dans `free_list` et fait de même avec le prochain (éventuel), sauf que si les 2 existent elle merge le tout.

- **`int find_free_address_strategy(MemoryHandler *handler, int size, int strategy)`**

Renvoie l'adresse de début d'un segment libre (qui est assez large oour la taille `size` demandée, selon la stratégie choisie : First Fit (0) trouve le premier assez grand, Best Fit (1) celui avec le moins de perte d'espace, Worst Fit (2) le plus grand.

5.3. Fonctions parser

- **`Instruction *parse_data_instruction(const char * line, HashMap *memory_locations)`**

Fonction qui parse une instruction `data`. Elle extrait le mnemonic (variable), l'operand 1 (type) et l'operand 2 (valeur), et rempli les champs d'une variable de type `Instruction` qu'elle retourne.
S'il ya plusieurs valeurs (des virgules dans operand 2) il les met chacune dans une case différentes du tableau `memory` qui représente la mémoire. Il ajoute à la variable statique `nb_occ` le nombre de ces valeurs. Cette variable contient donc au dernier appel le nombre total de valeurs qu'on va utiliser comme taille pour l'allocation du segment de données.

-**`Instruction *parse_code_instruction(const char *line, HashMap *labels, int code_count)`** Fonction qui parse une instruction de code. Elle fait de même que la fonction `parse_data_instruction` (même si l'extraction est plus compliquée parceque le format est moins uniforme, et il y a possibilité de présence d'un label detecté par les 2 points `:`, sauf dans le cas ou c'est un opérande de la forme `[SEGMENT:REGISTRE]`) mais la présence d'un paramètre `code_count` permet de donner à la fonction directement l'indice de la case ou il va se trouver dans le segment de code, pour pouvoir associer cette « adresse » à chaque label dans la hashmap `labels`.

- **`ParserResult *parse(const char *filename)`**

Fonction qui récupère un programme en pseudo-assembleur d'un fichier .txt, et qui le parse et récupère le résultat dans un `ParserResult` qu'il retourne. Pour faire ça elle initialise un `ParserResult` et lis le fichier ligne par ligne à l'aide des fonctions `fgets` et `sscanf` prédefinies, et parse chaque ligne en une instruction à l'aide des fonctions `parse_data_instruction` puis `parse_code_instruction`.

- **`void free_parser_result(ParserResult *parser)`**

Fonction qui libère le parser. Elle est notable parcequ'elle ne libère que les tableaux et pas les instructions qu'ils contiennent, puisque cela est fait plus tard par la fonction qui libère le CPU. (on évite donc un double `free`).

5.4. Fonctions CPU

- **`CPU *cpu_init(int memory_size)`**

Fonction qui initialise le CPU et ses composants et alloue le segment de pile `Stack Segment` de taille 128. L'argument `memory_size` doit donc être au moins égal à 128 (préférablement supérieur pour avoir assez d'espace pour les autres segments comme celui du code ou des données).

- **Description des registres du processeur :**
  - **AX, BX, CX, DX** – registres de valeurs avec lesquels l'utilisateur peut interagir directement en manipulant leur valeur à l'aide d'instructions standard.
  - **IP (pour "instruction pointer")** – sert à indiquer le numéro de la prochaine instruction à exécuter
  - **ZF (pour "zero flag")** – sert à indiquer si la dernière opération a abouti à un résultat nul (valeur 1) ou non (valeur 0).
  - **SF (pour "sign flag")** – sert à indiquer si la dernière opération a abouti à un résultat strictement négatif (valeur 1) ou non (valeur 0).

- **SP (pour "stack pointer")** – pointe toujours vers le sommet de la pile, indiquant la dernière adresse utilisée. Chaque opération "PUSH" décrémente SP, tandis que chaque opération "POP" l'incrémente.
- **BP (pour "base pointer")** – fournit un point de référence stable pour accéder aux paramètres et variables locales d'une fonction, servant de base pour le référencement des données sur la pile. Dans notre cas, il s'agit d'un pointeur sur le bas de la pile, c'est-à-dire l'adresse mémoire la plus grande (car la pile s'étend vers les adresses mémoire inférieures).
- **ES (pour "extra segment")** – contient l'adresse du premier emplacement de mémoire dans l'extra segment, permet d'accéder à un segment alloué dynamiquement

- **`void allocate_variables(CPU *cpu, Instruction** data_instructions,int data_count)`**

Fonction qui alloue le segment de données `DATA SEGMENT` et qui y stocke les valeurs des variables contenues dans le tableau `data_instructions` du parser

- **Fonctions d'adressage à l'aide d'expressions régulières**

Ce sont des fonctions qui prennent en paramètre un CPU et un operand, et qui renvoient le value/espace mémoire sur lequel pointe cet opérande

`-void * immediate_addressing(CPU * cpu, const char * operand)` :
Traite l'adressage immédiat qui est une valeur numérique directement, qui est reconnu par un match à l'expression régulière `^[0-9]+$`, retourne un pointeur vers cette fonction dans une table de hachage `constant_pool` (où sa présence est vérifiée auparavant pour éviter les doublons). ex: MOV AX,**42**

`-void *register_addressing(CPU * cpu, const char*operand)` :
Traite l'adressage par registre, qui est reconnu par un match à l'expression régulière `^[A-D]X$`, retourne un pointeur vers le value associé à ce registre dans la table de hachage `context` du CPU. ex: MOV AX,**BX**

`-void *memory_direct_addressing(CPU * cpu, const char*operand)`:
Traite l'adressage direct, qui est reconnu par un match à l'expression régulière `^\[[0-9]+\]$`, retourne un pointeur vers l'adresse qui est entre crochets dans le segment de données `DS` du tableau `memory` du `memoryhandler` du CPU. ex: MOV AX,[**5**]

`-void * register_indirect_addressing(CPU * cpu, const char*operand)`:
Traite l'adressage direct, qui est reconnu par un match à l'expression régulière `^\[[A-D]X\]$`, récupère la valeur contenue dans le registre qui va être l'adresse et retourne un pointeur vers cette adresse dans le segment de données `DS` du tableau `memory` du `memoryhandler` du CPU. ex: MOV AX,[**BX**]

`-void *segment_override_addressing(CPU *cpu, const char *operand):` Traite l'adressage avec redéfinition (override) de segment , qui est reconnu par un match à l'expression régulière `^\[(D|C|S|E)S:[A-D]X\]$`, récupère la valeur contenue dans le registre (après les 2 points) qui va être l'adresse et retourne un pointeur vers cette adresse dans le segment de choix (avant les 2 points) du tableau `memory` du `memoryhandler` du CPU. ex: MOV **[ES,AX]**, 10

`-void *resolve_addressing(CPU *cpu, const char *operand)` Teste toutes les fonctions d'addressage sur un opérande pour le résoudre.

- **`void handle_MOV(CPU * cpu, void * src, void *dest)`**

Fonction d'affectation, transfère ce qui est pointé par `src` vers ce que pointe `dest`

- **`int push_value(CPU *cpu, int value)`**

Fonction qui empile une valeur sur la pile du CPU en la stockant dans le segment SS (Stack Segment = Segment de pile) à l'adresse SP (Stack Pointer), puis décrémente SP.

- **`int pop_value(CPU *cpu, int* dest)`**

Dépile la valeur du sommet de la pile dans dest, depuis le segment SS à l'adresse SP + 1, puis incrémente SP. Retourne 1 si succès, −1 sinon.

- **`int alloc_es_segment(CPU* cpu)`**

Fonction d'allocation de l'`Extra Segment` de la taille de la valeur qui se retrouve dans le registre AX en utilisant la fonction `find_free_address_strategy` pour trouver le segment convenable selon la méthode choisie dans le registre BX.

5.5. Fonctions d'exécution

- **`int resolve_constants(ParserResult * result)`**

Fonction qui résout les constantes : elle parcourt toutes les instructions du segment .CODE et remplace les variables par leur adresse dans la mémoire `hashmap memory_locations` dans le parser), et les labels par leur adresse dans le code (`hashmap labels` dans le parser).
Elle utilise la fonction search_and_replace pour faire ces remplacements.

- **`void allocate_code_segment(CPU *cpu, Instruction **code_instructions, int code_count)`**

Fonction qui alloue le segment de code qui contient les instructions. Elle crée le segment et récopie les instructions de code à partir du tableau `code_instructions` du parser.

- **`int handle_instruction(CPU *cpu, Instruction *instr, void *src, void *dest)`**

laisses ca -> ok mais met la partie ou tu les a detaille en bas Traite une instruction en fonction de son mnémonique. Utilise les pointeurs src (valeur source) et dest (emplacement de destination) selon le type d'opération (MOV, ADD, CMP, JMP, JZ, JNZ, HALT, PUSH, POP, ALLOC, FREE). Met à jour les registres, le contexte ou la mémoire en conséquence. Le travail de chaque mnémonique est détaillé dans la partie « Conception de programme en Pseudo-Assembleur »

- **`int execute_instruction(CPU *cpu, Instruction *instr)`**

Exécute une instruction en fonction de son mnémonique. Résout les opérandes via resolve_addressing puis appelle handle_instruction. Les sauts (JMP, JZ, JNZ) et PUSH utilisent operand1 comme source, POP utilise operand1 comme destination, les autres instructions utilisent operand2 comme source et operand1 comme destination.

- **`Instruction *fetch_next_instruction(CPU *cpu)`**

Charge l'instruction suivante à exécuter depuis le segment CS (code segment), en utilisant la valeur actuelle de IP (Instruction Pointer) comme index, puis incrémente IP.

- **`int run_program_preview(CPU *cpu)`**

Lance l'exécution pas-à-pas du programme chargé dans le CPU. Affiche l'état initial, puis attend un appui sur « Entrée » pour exécuter chaque instruction une à une (fetch + execute). Permet à l'utilisateur de quitter à tout moment avec « Q ». Affiche l'état du CPU après chaque exécution.

- **`void print_entire_cpu(CPU *cpu)`**

Affiche l'état complet du CPU, incluant les segments SS, DS, CS, ES (pile, données, code, extra), en parcourant chaque segment et en imprimant soit la valeur entière correspondante soit un pour les cases vides. Affiche également le contexte (hashmap des registres). Utile pour le débogage et l'observation de l'état de la machine après exécution d'instructions.