

# Rapport : Introduction au langage d'assemblage

SANNA Thomas, L3STI

24 novembre 2024



UNIVERSITÀ  
DI CORSICA

PASQUALE  
PAOLI

# Table des matières

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>   | <b>4</b>  |
| 1.1      | Qu'est-ce que l'assembleur ? . . . . .                            | 4         |
| 1.2      | Pourquoi utiliser l'assembleur ? . . . . .                        | 4         |
| 1.3      | Inconvénients de l'assembleur . . . . .                           | 5         |
| 1.4      | Comparaison entre un langage de haut niveau et l'assembleur       | 5         |
| 1.4.1    | Langages de haut niveau . . . . .                                 | 5         |
| 1.4.2    | Langage assembleur . . . . .                                      | 5         |
| 1.4.3    | Comparaison . . . . .   | 7         |
| 1.5      | Applications de l'assembleur . . . . .                            | 7         |
| <b>2</b> | <b>Historique</b>   | <b>9</b>  |
| 2.1      | Les débuts de l'assembleur . . . . .                              | 9         |
| 2.2      | L'évolution des langages assembleurs . . . . .                    | 9         |
| 2.3      | L'assembleur et les mini-ordinateurs . . . . .                    | 10        |
| 2.4      | L'ère des microprocesseurs . . . . .                              | 10        |
| 2.5      | L'assembleur aujourd'hui . . . . .                                | 10        |
| 2.6      | L'impact de l'assembleur sur l'informatique . . . . .             | 10        |
| <b>3</b> | <b>Architecture matérielle</b>                                    | <b>11</b> |
| 3.1      | Unité centrale de traitement (CPU) . . . . .                      | 11        |
| 3.2      | Mémoire (RAM) . . . . .   | 12        |
| 3.3      | Bus . . . . .   | 12        |
| 3.4      | Mémoire cache . . . . .   | 13        |
| 3.5      | Périphériques d'entrée/sortie (I/O) . . . . .                     | 13        |
| 3.6      | Architecture de Harvard vs. Architecture de von Neumann . . . . . | 13        |
| <b>4</b> | <b>Notions de base en assembleur</b>                              | <b>15</b> |
| 4.1      | Définition de variables . . . . .                                 | 15        |
| 4.1.1    | Déclaration de variables dans la section .data . . . . .          | 15        |
| 4.1.2    | Déclaration de variables dans la section .bss . . . . .           | 15        |
| 4.2      | Instructions . . . . .  | 16        |

---

|          |  |           |
|----------|--|-----------|
| 4.2.1    | MOV . . . . .  | 16        |
| 4.2.2    | ADD . . . . .  | 17        |
| 4.2.3    | CMP . . . . .  | 17        |
| 4.3      | Registres . . . . .  | 18        |
| 4.3.1    | Registres généraux . . . . .   | 18        |
| 4.3.2    | Registres de segment . . . . .   | 19        |
| 4.3.3    | Registres de pointeur et d'index . . . . .                                 | 19        |
| 4.3.4    | Registre d'instruction et de drapeaux . . . . .                            | 20        |
| 4.4      | Pointeurs . . . . .  | 20        |
| 4.4.1    | Concept de pointeur . . . . .  | 20        |
| 4.4.2    | Utilisation des pointeurs en assembleur . . . . .                          | 20        |
| <b>5</b> | <b>Installation et exécution de l'assembleur</b>                           | <b>21</b> |
| 5.1      | Installation sur Windows . . . . .   | 21        |
| 5.1.1    | Étapes d'installation . . . . .  | 21        |
| 5.1.2    | Compilation et exécution d'un programme en assembleur                      | 21        |
| 5.2      | Installation sur Linux . . . . .   | 22        |
| 5.2.1    | Étapes d'installation . . . . .  | 22        |
| 5.2.2    | Compilation et exécution d'un programme en assembleur                      | 22        |
| 5.3      | Installation sur macOS . . . . .   | 23        |
| 5.3.1    | Étapes d'installation . . . . .  | 23        |
| 5.3.2    | Compilation et exécution d'un programme en assembleur                      | 23        |
| <b>6</b> | <b>Exemples de programme en assembleur</b>                                 | <b>24</b> |
| 6.1      | Composition d'un code assembleur . . . . .                                 | 24        |
| 6.1.1    | Commentaires . . . . .   | 24        |
| 6.1.2    | Directives (ou Sections) . . . . .   | 25        |
| 6.2      | Exemple de programme en assembleur : Addition de deux<br>nombres . . . . . | 25        |
| 6.3      | Exemple de programme en assembleur : Lecture de données . .                | 26        |
| 6.3.1    | Explication des codes des appels système . . . . .                         | 27        |
| 6.3.2    | Résultat . . . . .   | 27        |
| <b>7</b> | <b>Conclusion</b>  | <b>28</b> |

# Chapitre 1

## Introduction

L'assembleur, ou langage d'assemblage, est un langage de programmation de bas niveau qui est étroitement lié à l'architecture matérielle d'un ordinateur. Contrairement aux langages de haut niveau comme le C ou le Python, l'assembleur permet de manipuler directement les registres et la mémoire de la machine.

### 1.1 Qu'est-ce que l'assembleur ?

L'assembleur est un langage de programmation qui traduit les instructions du programmeur en code machine compréhensible par le processeur. Chaque instruction en assembleur correspond à une instruction machine spécifique, ce qui permet un contrôle très précis du matériel.

### 1.2 Pourquoi utiliser l'assembleur ?

L'utilisation de l'assembleur présente plusieurs avantages :

- **Performance** : Les programmes en assembleur peuvent être extrêmement rapides et efficaces car ils sont optimisés pour l'architecture spécifique du processeur.
- **Contrôle** : L'assembleur permet un contrôle total sur le matériel, ce qui est essentiel pour les systèmes embarqués, les pilotes de périphériques et les applications en temps réel.
- **Compréhension approfondie** : Apprendre l'assembleur aide à comprendre le fonctionnement interne des ordinateurs, ce qui peut être bénéfique pour le débogage et l'optimisation des programmes en langages de haut niveau.

## 1.3 Inconvénients de l'assembleur

- Malgré ses avantages, l'assembleur présente également des inconvénients :
- **Complexité** : La programmation en assembleur est complexe et nécessite une connaissance approfondie de l'architecture matérielle.
  - **Portabilité** : Les programmes en assembleur sont spécifiques à une architecture de processeur particulière et ne peuvent pas être facilement portés sur d'autres architectures.
  - **Temps de développement** : Écrire des programmes en assembleur prend généralement plus de temps que d'utiliser des langages de haut niveau en raison de la nécessité de gérer les détails de bas niveau.

## 1.4 Comparaison entre un langage de haut niveau et l'assembleur

Les langages de haut niveau, comme le Python, le C ou le Java, sont conçus pour être faciles à lire et à écrire pour les humains. Ils permettent aux développeurs de se concentrer sur la logique du programme sans se soucier des détails de l'architecture matérielle. Cependant, ces langages doivent être traduits en code machine compréhensible par le processeur, ce qui implique plusieurs étapes de compilation et d'interprétation.

### 1.4.1 Langages de haut niveau

Lorsqu'un programme est écrit dans un langage de haut niveau, il passe par plusieurs étapes avant d'être exécuté par le processeur :

1. **Compilation** : Le code source est traduit en code intermédiaire (par exemple, bytecode pour Java) ou directement en code machine (comme pour le C).
2. **Interprétation** : Pour certains langages comme Python, le code source est interprété ligne par ligne par un interpréteur, qui traduit chaque ligne en instructions machine à la volée.
3. **Optimisation** : Les compilateurs modernes effectuent diverses optimisations pour améliorer les performances du code généré.

### 1.4.2 Langage assembleur

En revanche, l'assembleur est beaucoup plus proche du matériel. Chaque instruction en assembleur correspond directement à une instruction machine

spécifique. Voici les étapes typiques pour traduire un programme en assembleur en code machine :

1. **Assemblage** : Le code assembleur est traduit en code machine par un assembleur.
2. **Lien** : Les fichiers objets générés par l'assembleur sont liés pour créer un exécutable.

### 1.4.3 Comparaison

- **Abstraction** : Les langages de haut niveau offrent un niveau d'abstraction élevé, ce qui simplifie la programmation mais peut entraîner une perte de contrôle sur le matériel. L'assembleur, en revanche, offre un contrôle précis sur le matériel mais nécessite une connaissance approfondie de l'architecture du processeur.
- **Performance** : Les programmes en assembleur peuvent être extrêmement rapides et efficaces car ils sont optimisés pour l'architecture spécifique du processeur. Les langages de haut niveau dépendent des optimisations du compilateur pour atteindre des performances similaires.
- **Portabilité** : Les programmes en langages de haut niveau sont généralement plus portables entre différentes architectures matérielles. Les programmes en assembleur sont spécifiques à une architecture de processeur particulière.
- **Temps de développement** : Écrire des programmes en langages de haut niveau prend généralement moins de temps que d'utiliser l'assembleur en raison de la simplicité et de l'abstraction offertes par ces langages.

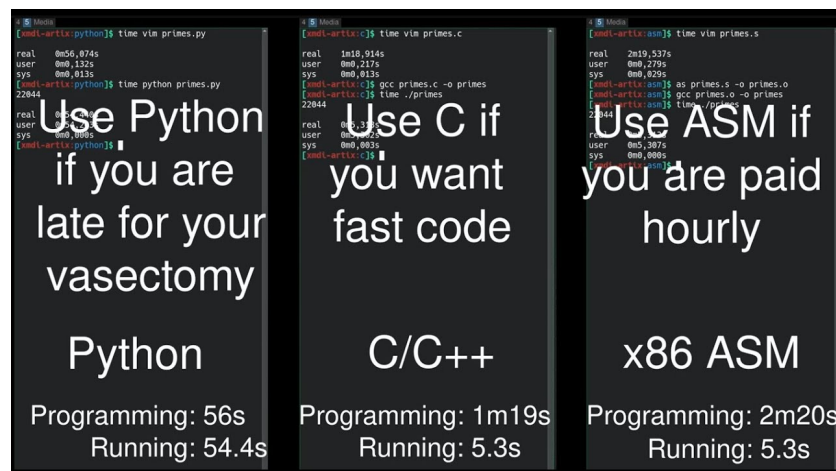


FIGURE 1.1 – Comparaison humoristique entre Python, C et ASM. Source : <https://www.youtube.com/watch?v=c-6SfKWmH0k> TUTORIALS s. d.

## 1.5 Applications de l'assembleur

L'assembleur est utilisé dans plusieurs domaines spécifiques :

- **Systèmes embarqués** : Les microcontrôleurs et autres systèmes em-



barqués utilisent souvent l'assembleur pour des performances optimales.

- **Développement de systèmes d'exploitation** : Les noyaux de systèmes d'exploitation et les pilotes de périphériques sont souvent écrits en assembleur pour un contrôle précis du matériel.
- **Optimisation de code** : Les sections critiques de certains programmes peuvent être écrites en assembleur pour maximiser les performances.

# Chapitre 2

## Historique

Le langage assembleur a été développé dans les années 1950 pour simplifier la programmation des premiers ordinateurs. Avant l'assembleur, les programmeurs devaient écrire des programmes en code machine, ce qui était extrêmement fastidieux et sujet aux erreurs.

### 2.1 Les débuts de l'assembleur

Les premiers ordinateurs, comme l'ENIAC et l'UNIVAC, utilisaient des commutateurs et des câbles pour programmer des instructions en code machine. Cela rendait la programmation très laborieuse et sujette aux erreurs. Pour simplifier ce processus, les langages assembleurs ont été créés. Le premier assembleur connu est celui développé pour l'EDSAC (Electronic Delay Storage Automatic Calculator) en 1949 par Maurice Wilkes et son équipe à l'Université de Cambridge.

### 2.2 L'évolution des langages assembleurs

Au fil des années, les langages assembleurs ont évolué pour s'adapter aux nouvelles architectures de processeurs. Chaque nouvelle génération de processeurs a introduit de nouvelles instructions et de nouvelles fonctionnalités, ce qui a conduit à la création de nouveaux assembleurs. Par exemple, l'assembleur pour le processeur IBM 704, développé dans les années 1950, a introduit des instructions pour les opérations en virgule flottante, ce qui a permis de réaliser des calculs plus complexes.

## 2.3 L'assembleur et les mini-ordinateurs

Dans les années 1960 et 1970, l'assembleur a joué un rôle crucial dans le développement des mini-ordinateurs. Les mini-ordinateurs, comme le PDP-8 de Digital Equipment Corporation (DEC), étaient plus petits et moins coûteux que les ordinateurs centraux, ce qui les rendait accessibles à un plus grand nombre d'utilisateurs. Les langages assembleurs pour ces machines ont permis aux programmeurs de tirer pleinement parti de leurs capacités limitées en optimisant les performances et l'utilisation de la mémoire.

## 2.4 L'ère des microprocesseurs

Avec l'avènement des microprocesseurs dans les années 1970, l'assembleur est devenu encore plus important. Les premiers microprocesseurs, comme l'Intel 4004 et le Motorola 6800, avaient des capacités limitées et nécessitaient une programmation en assembleur pour maximiser leur efficacité. L'assembleur a également été utilisé pour développer les premiers systèmes d'exploitation et logiciels pour micro-ordinateurs, comme le CP/M et le MS-DOS.

## 2.5 L'assembleur aujourd'hui

Aujourd'hui, l'assembleur est toujours utilisé dans des domaines spécifiques où la performance et le contrôle du matériel sont essentiels. Les systèmes embarqués, les pilotes de périphériques, les noyaux de systèmes d'exploitation et les applications en temps réel sont souvent développés en assembleur. De plus, l'assembleur est utilisé pour optimiser les sections critiques de certains programmes écrits en langages de haut niveau.

## 2.6 L'impact de l'assembleur sur l'informatique

L'assembleur a eu un impact profond sur le développement de l'informatique. Il a permis de simplifier la programmation des premiers ordinateurs, de tirer pleinement parti des capacités des mini-ordinateurs et des microprocesseurs, et de développer des logiciels performants et efficaces. Apprendre l'assembleur aide également à comprendre le fonctionnement interne des ordinateurs, ce qui est bénéfique pour le débogage et l'optimisation des programmes en langages de haut niveau.

# Chapitre 3

## Architecture matérielle

Pour comprendre l'assembleur, il est essentiel de connaître les composants de base d'un ordinateur :

### 3.1 Unité centrale de traitement (CPU)

L'unité centrale de traitement, ou CPU (Central Processing Unit), est le cerveau de l'ordinateur. Elle exécute les instructions des programmes en effectuant des opérations arithmétiques, logiques, de contrôle et d'entrée/sortie. Le CPU est composé de plusieurs sous-composants clés :

- **Unité arithmétique et logique (ALU)** : Effectue les opérations arithmétiques et logiques.
- **Unité de contrôle** : Dirige les opérations du CPU en récupérant les instructions de la mémoire, en les décodant et en les exécutant.
- **Registres** : Petites unités de stockage à accès rapide utilisées pour stocker temporairement des données et des instructions.
- **Cache** : Mémoire rapide située à l'intérieur du CPU pour stocker les données fréquemment utilisées et réduire les temps d'accès.

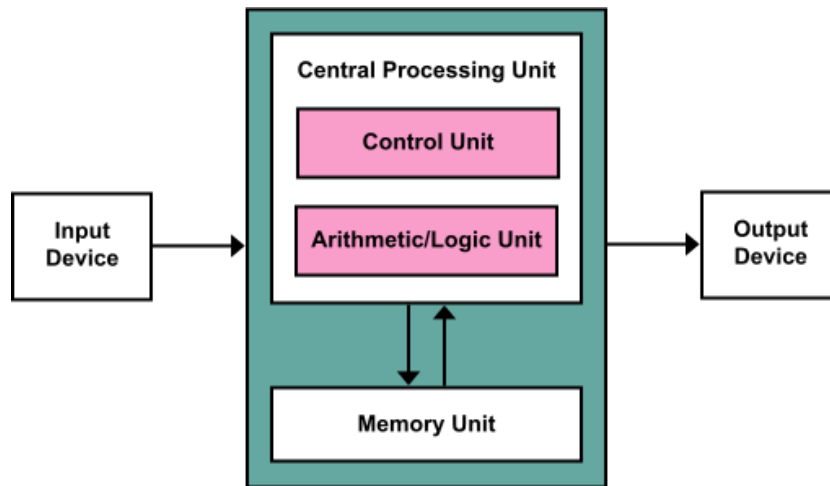


FIGURE 3.1 – Architecture de base d'un CPU. Source : [https://computersciencewiki.org/index.php?title=File:Cpu\\_diagram.png](https://computersciencewiki.org/index.php?title=File:Cpu_diagram.png) (MACKENTY s. d.)

## 3.2 Mémoire (RAM)

La mémoire vive, ou RAM (Random Access Memory), est utilisée pour stocker les données et les instructions pendant leur utilisation. La RAM est volatile, ce qui signifie qu'elle perd son contenu lorsque l'ordinateur est éteint. Elle est organisée en cellules de mémoire, chacune ayant une adresse unique.

## 3.3 Bus

Les bus sont des systèmes de communication qui transfèrent les données entre les composants de l'ordinateur. Il existe plusieurs types de bus :

- **Bus de données** : Transporte les données entre le CPU, la mémoire et les périphériques.
- **Bus d'adresse** : Transporte les adresses de mémoire des données que le CPU doit lire ou écrire.
- **Bus de contrôle** : Transporte les signaux de contrôle et de commande entre le CPU et les autres composants.

### 3.4 Mémoire cache

La mémoire cache est une mémoire rapide située à l'intérieur du CPU ou à proximité. Elle stocke les données fréquemment utilisées pour réduire les temps d'accès et améliorer les performances. Il existe généralement plusieurs niveaux de cache (L1, L2, L3), chaque niveau étant plus rapide mais plus petit que le précédent.

### 3.5 Périphériques d'entrée/sortie (I/O)

Les périphériques d'entrée/sortie permettent à l'ordinateur de communiquer avec le monde extérieur. Ils incluent des dispositifs tels que les claviers, les souris, les écrans, les imprimantes et les disques durs. Les opérations d'entrée/sortie sont souvent gérées par des contrôleurs spécifiques et peuvent être programmées en assembleur pour un contrôle précis.

### 3.6 Architecture de Harvard vs. Architecture de von Neumann

Il existe deux principales architectures de systèmes informatiques :

- **Architecture de von Neumann** : Utilise un seul bus pour les instructions et les données, ce qui peut entraîner des goulots d'étranglement.
- **Architecture de Harvard** : Utilise des bus séparés pour les instructions et les données, permettant des accès simultanés et améliorant les performances.

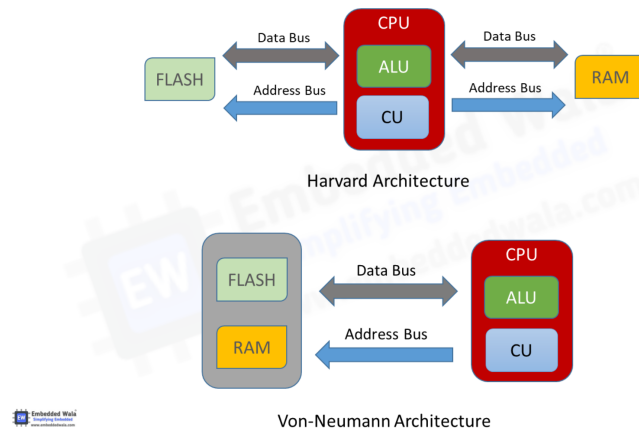


FIGURE 3.2 – Architecture de von Neumann vs. Architecture de Harvard. Source : <https://embeddedwala.com/Blogs/embeddedsystem/harvard-vs-von-neumann-architecture> (EMBEDDEDWALA s. d.)

# Chapitre 4

## Notions de base en assembleur

### 4.1 Définition de variables

En assembleur, les variables sont des emplacements de mémoire utilisés pour stocker des données. Les variables peuvent être déclarées dans différentes sections du programme, telles que la section `.data` ou la section `.bss`. Voici comment déclarer des variables en assembleur :

#### 4.1.1 Déclaration de variables dans la section `.data`

Les variables déclarées dans la section `.data` sont initialisées avec des valeurs spécifiques. Voici un exemple de déclaration de variables dans la section `.data` :

```
1  section .data
2      message db "Hello, World!", 10 ; Declare une chaine de
           caracteres avec un saut de ligne a la fin (10 en ASCII
           )
3      number db 42 ; Declare un octet (db = define byte) (8
           bits)
4      mot dw 1234 ; Declare un mot (dw = define word) (16 bits
           )
5      dmot dd 12345678 ; Declare un double mot (dd = define
           double) (32 bits)
6      qmot dq 1234567890123456789 ; Declare un quadruple mot (
           dq = define quadruple) (64 bits)
```

#### 4.1.2 Déclaration de variables dans la section `.bss`

Les variables déclarées dans la section `.bss` ne sont pas initialisées avec des valeurs spécifiques.



## 4.2 Instructions

Les instructions en assembleur sont des commandes simples qui disent au CPU quoi faire. Chaque instruction correspond généralement à une opération de bas niveau, comme déplacer des données ou effectuer une opération arithmétique. Voici quelques-unes des instructions de base les plus couramment utilisées en assembleur MIKA56 2013 :

### 4.2.1 MOV

L’instruction ‘mov’ est utilisée pour copier des données d’un emplacement à un autre. Elle peut copier des données entre des registres, de la mémoire vers un registre, d’un registre vers la mémoire, ou charger une valeur immédiate dans un registre.

```
1 mov eax, ebx      ; Copier la valeur de ebx dans eax
2 mov eax, [var]    ; Charger la valeur de la variable var dans
   eax
3 mov [var], eax    ; Stocker la valeur de eax dans la variable
   var
4 mov eax, 10       ; Charger la valeur immzdiante 10 dans eax
```

### 4.2.2 ADD

L’instruction ‘add’ est utilisée pour additionner deux opérandes. Le résultat est stocké dans le premier opérande.

```
1 add eax, ebx    ; Ajouter la valeur de ebx a eax
2 add eax, [var]   ; Ajouter la valeur de la variable var a eax
3 add [var], eax   ; Ajouter la valeur de eax a la variable var
4 add eax, 10      ; Ajouter la valeur immediate 10 a eax
```

### 4.2.3 CMP

L’instruction ‘cmp’ est utilisée pour comparer deux opérandes. Elle soustrait le second opérande du premier et met à jour les drapeaux du registre FLAGS sans stocker le résultat.

```
1 cmp eax, ebx    ; Comparer eax et ebx
2 cmp eax, [var]   ; Comparer eax et la valeur de la variable
   var
3 cmp eax, 10      ; Comparer eax et la valeur immediate 10
```

Il existe encore beaucoup d’autres instructions en assembleur, mais celles-ci sont les plus couramment utilisées.

## 4.3 Registres

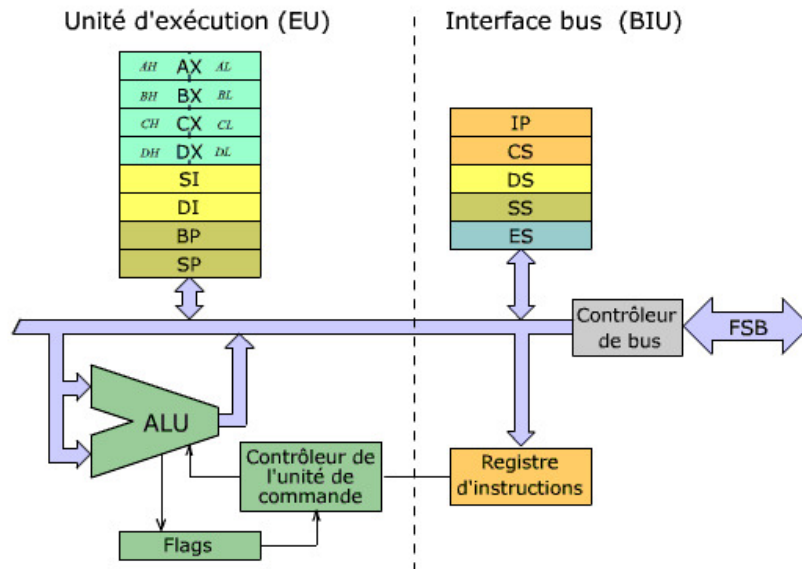


FIGURE 4.1 – Registres du processeur x86. Source : (COURSTECHINFO s. d.)

Les registres sont des emplacements de stockage internes au CPU. Ils sont utilisés pour stocker temporairement des données pendant l'exécution des instructions. Les registres couramment utilisés incluent :

### 4.3.1 Registres généraux

Les registres généraux sont utilisés pour diverses opérations arithmétiques, logiques et de manipulation de données. Les registres généraux incluent AX, BX, CX et DX, chacun ayant des sous-parties spécifiques.

- **AX (Accumulator Register)** : Utilisé principalement pour les opérations arithmétiques et logiques. Il est souvent impliqué dans les opérations de multiplication et de division. AX peut être divisé en deux registres de 8 bits : AH (High byte) et AL (Low byte).

- **AH** : Partie haute du registre AX.

- **AL** : Partie basse du registre AX.

NB : Il y a une partie haute et une partie basse pour chaque registre général.

- **BX (Base Register)** : Utilisé comme pointeur de base pour accéder aux données en mémoire. Il est souvent utilisé dans les opérations de

calcul d'adresse. BX peut également être divisé en BH (High byte) et BL (Low byte).

- **CX (Count Register)** : Utilisé principalement comme compteur dans les boucles et les opérations de chaîne. Par exemple, il est utilisé par les instructions de répétition comme 'LOOP', 'REP', 'REPE', et 'REPNE'. CX peut être divisé en CH (High byte) et CL (Low byte).
- **DX (Data Register)** : Utilisé pour les opérations d'entrée/sortie et certaines opérations arithmétiques. Par exemple, il est utilisé en combinaison avec AX pour les opérations de multiplication et de division de grande taille. DX peut être divisé en DH (High byte) et DL (Low byte).

### 4.3.2 Registres de segment

Les registres de segment sont utilisés pour accéder à différentes sections de la mémoire. Ils incluent CS, DS, SS et ES.

- **CS (Code Segment)** : Contient l'adresse de base du segment de code, où les instructions du programme sont stockées.
- **DS (Data Segment)** : Contient l'adresse de base du segment de données, où les variables sont stockées.
- **SS (Stack Segment)** : Contient l'adresse de base du segment de pile, utilisé pour la gestion de la pile d'exécution.
- **ES (Extra Segment)** : Utilisé comme segment supplémentaire pour les opérations de chaîne et autres opérations de données.

### 4.3.3 Registres de pointeur et d'index

Les registres de pointeur et d'index sont utilisés pour le calcul d'adresse et la manipulation de données.

- **SP (Stack Pointer)** : Pointeur de pile, utilisé pour gérer la pile d'exécution. Il pointe vers le sommet de la pile.
- **BP (Base Pointer)** : Pointeur de base, utilisé pour accéder aux variables locales dans la pile. Il est souvent utilisé pour les appels de fonctions.
- **SI (Source Index)** : Utilisé pour les opérations de chaîne et les transferts de données. Il pointe vers la source des données.
- **DI (Destination Index)** : Utilisé pour les opérations de chaîne et les transferts de données. Il pointe vers la destination des données.

### 4.3.4 Registre d'instruction et de drapeaux

- **IP (Instruction Pointer)** : Pointeur d'instruction, contient l'adresse de la prochaine instruction à exécuter.
- **FLAGS (Registre de drapeaux)** : Contient des indicateurs de statut qui reflètent les résultats des opérations. Les drapeaux incluent le drapeau de zéro (ZF), le drapeau de transport (CF), le drapeau de signe (SF), et d'autres.

## 4.4 Pointeurs

Les pointeurs sont des variables qui contiennent des adresses mémoire. En assembleur, les pointeurs sont souvent utilisés pour accéder directement à la mémoire. Comprendre les pointeurs est crucial pour manipuler efficacement les données en assembleur.

### 4.4.1 Concept de pointeur

Un pointeur est une variable qui stocke l'adresse mémoire d'une autre variable. Par exemple, si une variable A est stockée à l'adresse mémoire 0x1000, un pointeur P peut contenir la valeur 0x1000. Cela permet au programme d'accéder directement à la valeur de A en utilisant P.

### 4.4.2 Utilisation des pointeurs en assembleur

En assembleur, les pointeurs sont utilisés pour manipuler les données en mémoire de manière efficace. Voici quelques exemples d'utilisation des pointeurs en assembleur :

- **Accès direct à la mémoire** : Les pointeurs permettent d'accéder directement aux adresses mémoire, ce qui est essentiel pour les opérations de bas niveau.
- **Manipulation de tableaux** : Les pointeurs sont utilisés pour parcourir et manipuler les éléments des tableaux.
- **Gestion de la pile** : Les pointeurs sont utilisés pour gérer la pile d'exécution, en particulier pour les appels de fonctions et la gestion des variables locales.

# Chapitre 5

## Installation et exécution de l'assembleur

Maintenant que l'on a appris tout cela, il est nécessaire d'installer un assembleur pour pouvoir écrire et exécuter des programmes en assembleur. Voici comment installer un assembleur sur Windows, Linux et macOS.

### 5.1 Installation sur Windows

Pour installer un assembleur sur Windows, nous utiliserons NASM (Net-wide Assembler), un assembleur populaire pour le langage assembleur x86.

#### 5.1.1 Étapes d'installation

1. Téléchargez NASM depuis le site officiel : <https://www.nasm.us/>
2. Exécutez le fichier d'installation et suivez les instructions pour installer NASM.
3. Ajoutez le répertoire d'installation de NASM à la variable d'environnement PATH pour pouvoir l'exécuter depuis n'importe quel répertoire dans l'invite de commandes.

#### 5.1.2 Compilation et exécution d'un programme en assembleur

1. Ouvrez un éditeur de texte (comme Notepad++) et écrivez votre programme en assembleur. Enregistrez le fichier avec l'extension '.asm'.
2. Ouvrez l'invite de commandes et naviguez jusqu'au répertoire contenant votre fichier '.asm'.

3. Compilez le programme en utilisant la commande suivante :

```
1 nasm -f win32 programme.asm -o programme.obj
```

4. Liez le fichier objet pour créer un exécutable :

```
1 gcc programme.obj -o programme.exe
```

5. Exécutez le programme :

```
1 programme.exe
```

## 5.2 Installation sur Linux

Sur Linux, NASM est également disponible et peut être installé via le gestionnaire de paquets.

### 5.2.1 Étapes d'installation

1. Ouvrez un terminal.
2. Installez NASM en utilisant le gestionnaire de paquets de votre distribution. Par exemple, sur Debian/Ubuntu, utilisez la commande suivante :

```
1 sudo apt-get install nasm
```

### 5.2.2 Compilation et exécution d'un programme en assembleur

1. Ouvrez un éditeur de texte (comme gedit ou vim) et écrivez votre programme en assembleur. Enregistrez le fichier avec l'extension '.asm'.
2. Ouvrez un terminal et naviguez jusqu'au répertoire contenant votre fichier '.asm'.
3. Compilez le programme en utilisant la commande suivante :

```
1 nasm -f elf32 programme.asm -o programme.o
```

4. Liez le fichier objet pour créer un exécutable :

```
1 ld -m elf_i386 -s -o programme programme.o
```

5. Exécutez le programme :

```
1 ./programme
```

## 5.3 Installation sur macOS

Sur macOS, vous pouvez également utiliser NASM.

### 5.3.1 Étapes d'installation

1. Installez Homebrew si ce n'est pas déjà fait en suivant les instructions sur <https://brew.sh/>.
2. Installez NASM en utilisant Homebrew :

```
1 brew install nasm
```

### 5.3.2 Compilation et exécution d'un programme en assembleur

1. Ouvrez un éditeur de texte (comme TextEdit ou vim) et écrivez votre programme en assembleur. Enregistrez le fichier avec l'extension '.asm'.
2. Ouvrez un terminal et naviguez jusqu'au répertoire contenant votre fichier '.asm'.
3. Compilez le programme en utilisant la commande suivante :

```
1 nasm -f macho32 programme.asm -o programme.o
```

4. Liez le fichier objet pour créer un exécutable :

```
1 ld -macosx_version_min 10.7.0 -o programme programme.o  
-lSystem
```

5. Exécutez le programme :

```
1 ./programme
```



# Chapitre 6

## Exemples de programme en assembleur

Avant de montrer un exemple de programme en assembleur, il est important de comprendre comment est composé un code assembleur. Un programme en assembleur est généralement divisé en plusieurs sections, chacune ayant un rôle spécifique.

### 6.1 Composition d'un code assembleur

Un programme en assembleur est typiquement composé des sections suivantes :

- **Section `.data`** : Cette section est utilisée pour déclarer les données statiques ou globales. Les variables définies ici sont accessibles tout au long de l'exécution du programme.
- **Section `.bss`** : Cette section est utilisée pour déclarer les variables non initialisées. Elle est similaire à la section `.data`, mais les variables ne sont pas initialisées avec des valeurs spécifiques.
- **Section `.text`** : Cette section contient le code exécutable du programme. C'est ici que les instructions du programme sont écrites.

#### 6.1.1 Commentaires

Les commentaires en assembleur sont utilisés pour expliquer le code et sont ignorés par l'assembleur. Ils commencent par un point-virgule (`;`).

### 6.1.2 Directives (ou Sections)

Les directives sont des instructions spéciales pour l'assembleur. Elles commencent généralement par un point (‘.’) et ne sont pas traduites en instructions machine. Par exemple, ‘.data’, ‘.bss’, et ‘.text’ sont des directives.

## 6.2 Exemple de programme en assembleur : Addition de deux nombres

Voici un exemple simple de programme en assembleur qui additionne deux nombres :

```
1 section .data
2   num1 db 5          ; Declaration de la variable num1 avec la
   valeur 5. (db = define byte)
3   num2 db 10         ; Declaration de la variable num2 avec la
   valeur 10
4   result db 0        ; Declaration de la variable result
   initialisee a 0
5
6 section .text
7   global _start      ; Declaration de l'etiquette globale _start
8
9   _start:
10  mov al, [num1]      ; Charger la valeur de num1 dans le
   registre AL
11  add al, [num2]      ; Ajouter la valeur de num2 a AL
12  mov [result], al    ; Stocker le resultat dans result
13
14  ; Exit
15  mov rax, 60         ; Code de sortie pour l'appel systeme
16  xor rdi, rdi        ; Code de retour (0 signifie succes)
17  syscall             ; Appel systeme pour terminer le programme
```

### 6.3 Exemple de programme en assembleur : Lecture de données

Voici un exemple de programme en assembleur qui lit une chaîne de caractères depuis l'entrée standard et l'affiche :

```

1  section .bss ; "bss" sert a declarer des variables non
   initialisees
2  buffer resb 128 ; Reserver 128 octets pour le buffer.
3
4  section .data
5  msg db "Entrez une chaine de caracteres: ", 0
6
7  section .text
8  global _start
9
10 _start:
11     ; Afficher le message
12     mov rax, 1          ; Code de l'appel systeme pour 'write'
13     ; (1)
14     mov rdi, 1          ; Descripteur de fichier pour stdout
15     ; (1)
16     mov rsi, msg        ; Adresse du message a afficher
17     mov rdx, 35         ; Longueur du message
18     syscall             ; Effectuer l'appel systeme 'write(1,
19     msg, 35)'
20
21     ; Lire l'entree du clavier
22     mov rax, 0          ; Code de l'appel systeme pour 'read'
23     ; (0)
24     mov rdi, 0          ; Descripteur de fichier pour stdin
25     ; (0)
26     mov rsi, buffer     ; Adresse du buffer ou stocker l'
27     entree
28     mov rdx, 128        ; Taille maximale de l'entree
29     syscall             ; Effectuer l'appel systeme 'read(0,
30     buffer, 128)'

```

```

31
32 ; Terminer le programme
33 mov rax, 60          ; Code de l'appel systeme pour 'exit'
34                       (60)
35 xor rdi, rdi         ; Code de retour (0 signifie succes)
36 syscall             ; Effectuer l'appel systeme 'exit(0)'

```

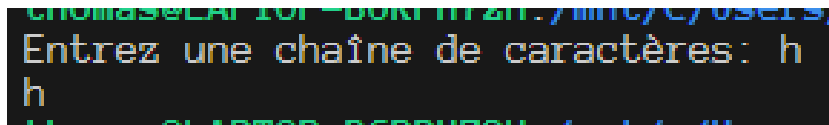
### 6.3.1 Explication des codes des appels système

Les appels système permettent aux programmes d'interagir avec le système d'exploitation. Voici quelques codes d'appels système couramment utilisés :

- **0** : 'read' - Lire des données depuis un fichier ou l'entrée standard.
- **1** : 'write' - Écrire des données dans un fichier ou la sortie standard.
- **60** : 'exit' - Terminer le programme.

NB - Pour une liste complète des appels système, voir la table des appels système Linux ici : VALSORDA 2013.

### 6.3.2 Résultat



```

Thomas@L3STI:~/BOUTIN21/miniC/05E15/
Entrez une chaîne de caractères: h

```

FIGURE 6.1 – Résultat de l'exécution du programme avec lecture de données.

On voit bien que le programme nous a demandé dans un premier temps d'entrer une chaîne de caractères, puis il nous l'a affichée suite à notre entrée "h". Pour enfin terminer le programme.

# Chapitre 7

## Conclusion

Bien que connaître l'Assembleur ne soit pas nécessaire pour la plupart des développeurs, il est important de comprendre les concepts de bas niveau pour apprécier pleinement le fonctionnement des ordinateurs. L'Assembleur est un langage puissant qui offre un contrôle précis sur le matériel, ce qui peut être utile dans des domaines spécifiques comme les systèmes embarqués, les pilotes de périphériques et l'optimisation de code. En apprenant l'Assembleur, on acquiert une compréhension plus profonde du fonctionnement interne des ordinateurs, ce qui peut être bénéfique pour le débogage et l'optimisation des programmes en langages de haut niveau.

# Bibliographie

- [Cou] COURSTECHINFO. *Introduction au Langage Assembleur*. URL : <https://courstechinfo.be/Programmation/IntroASM.html>.
- [Emb] EMBEDDEDWALA. *Harvard vs Von Neumann Architecture*. URL : <https://embeddedwala.com/Blogs/embeddedsystem/harvard-vs-von-neumann-architecture>.
- [Mac] Mr. MACKENTY. *Architecture of the central processing unit*. URL : [https://computersciencewiki.org/index.php/Architecture\\_of\\_the\\_central\\_processing\\_unit\\_%28CPU%29](https://computersciencewiki.org/index.php/Architecture_of_the_central_processing_unit_%28CPU%29).
- [Mik13] MIKA56. *CheatSheet ASM 8086*. 13 fév. 2013. URL : <https://cheatography.com/mika56/cheat-sheets/asm-8086/>.
- [Tut] IT TUTORIALS. *Python vs C/C++ vs Assembly side-by-side comparison*. URL : <https://www.youtube.com/watch?v=c-6SfKWmH0k>.
- [Val13] Filippo VALSORDA. *Searchable Linux Syscall Table*. 2013. URL : <https://filippo.io/linux-syscall-table/>.