LE LANGAGE D'ASSEMBLAGE (ASSEMBLEUR)

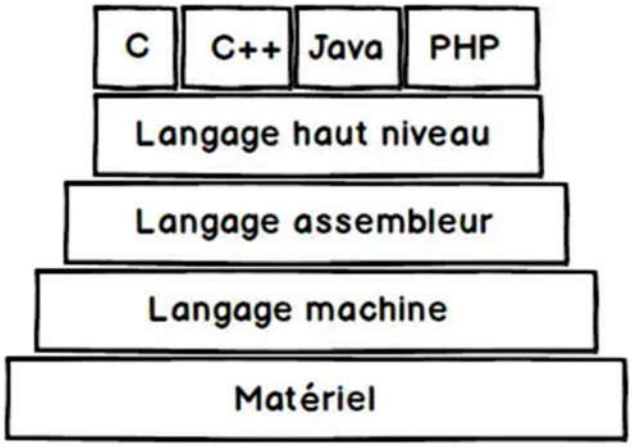
PLAN DE COURS

I. Introduction

I.1. Définition

Le langage assembleur est un langage de programmation de **bas niveau** qui permet d'écrire des programmes en utilisant des instructions compréhensibles par la machine, telles que des codes opérationnels, des adresses mémoire et des registres.

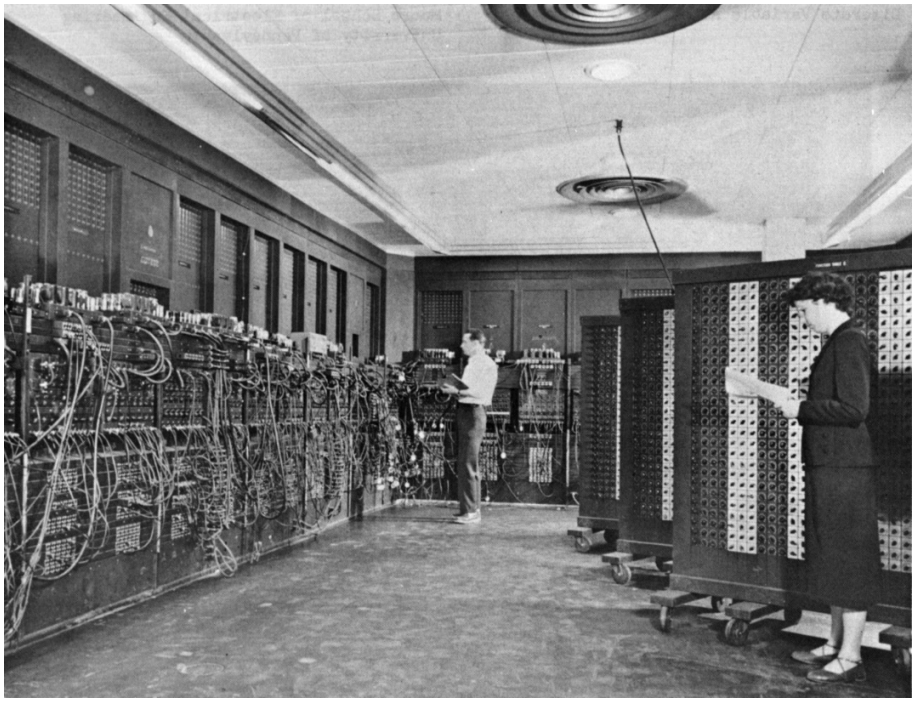
Contrairement aux langages de haut niveau, qui sont généralement plus abstraits et indépendants de la plateforme, le langage assembleur est spécifique à une architecture matérielle particulière, et il est souvent utilisé pour écrire des logiciels qui nécessitent une performance maximale et un contrôle fin sur les ressources de la machine.



Les programmes écrits en langage assembleur sont généralement difficiles à lire et à maintenir, mais ils peuvent être très efficaces en termes de vitesse d'exécution et d'utilisation de la mémoire.

I.2. Contexte historique de son utilisation

Le langage assembleur est né dans les **années 1940**, avec **les premiers ordinateurs électroniques tels que l'ENIAC et le Mark I**. À l'époque, les ordinateurs étaient programmés en binaire, ce qui était fastidieux et propice aux erreurs.



Le langage assembleur a été créé pour faciliter la programmation des ordinateurs en permettant aux programmeurs de travailler avec des mnémoniques pour représenter les instructions de bas niveau plutôt que de les écrire en binaire.

Dans les années 1950 et 1960, les ordinateurs sont devenus plus sophistiqués et le langage assembleur est devenu un outil important pour les programmeurs qui travaillaient sur des projets de grande envergure. Les langages de programmation de haut niveau tels que **COBOL, FORTRAN et BASIC ont également été développés pendant cette période,** mais le langage assembleur est resté important pour **la programmation système** et la **programmation de bas niveau**.

Dans les années 1970 et 1980, l'utilisation du langage assembleur a commencé à décliner avec l'arrivée de langages de programmation plus modernes.

Cependant, **le langage assembleur est toujours utilisé dans certains domaines**, tels que la **programmation système**, **les pilotes de périphériques**, **les logiciels embarqués** et **les jeux vidéo**. Il reste un outil important pour les programmeurs qui ont besoin d'un contrôle fin sur le matériel de l'ordinateur.

II. Les concepts de base du langage assembleur

II.1.Architecture d'un microprocesseur

Les processeurs modernes sont des circuits complexes qui comprennent des milliards de transistors, mais leur architecture est basée sur les mêmes principes que les processeurs des années 1970.

Les composants clés d'un processeur sont les suivants :

L'unité de contrôle (CU) : c'est le composant qui **contrôle le flux des données** à l'intérieur du processeur. Elle lit les instructions du programme, les décrypte et coordonne le transfert de données entre les différents composants.

L'unité arithmétique et logique (ALU) : c'est le composant qui effectue les opérations arithmétiques et logiques. Elle est capable de réaliser des opérations de base telles que l'addition, la soustraction, la multiplication, la division, les opérations logiques et les opérations de comparaison.

Les registres : ce sont des **mémoires internes du processeur** qui permettent de stocker des données temporaires **pour les opérations en cours.** Les registres sont très rapides mais limités en taille.

La mémoire cache : c'est une **mémoire interne du processeur** qui **contient des copies des données récemment utilisées** par le processeur. La mémoire cache est plus rapide que la mémoire vive (RAM) mais moins rapide que les registres.

Les bus : ce sont les lignes de communication qui permettent aux différents composants du processeur de communiquer entre eux et avec la mémoire vive et les périphériques externes.

L'unité de gestion de mémoire (MMU) : c'est le composant qui permet au processeur d'accéder à la mémoire vive.

Le fonctionnement précis d'un processeur dépend de son architecture spécifique, mais les principes de base sont les mêmes pour tous les processeurs.

II.2.Les registres du processeur

Les registres sont des emplacements de stockage de données très rapides qui se trouvent à l'intérieur du processeur.

Le langage assembleur utilise des registres pour stocker des données temporaires ou des résultats intermédiaires lors de l'exécution d'instructions.

Les registres sont souvent nommés en fonction de leur fonction, par exemple **"AX"** pour le **registre accumulateur**, **"BX"** pour le **registre de base**, **"CX"** pour le **registre de compteur** et **"DX"** pour le **registre de données**.

En langage assembleur, chaque registre est identifié par un nom, souvent une abréviation de sa fonction, et une taille spécifique exprimée en bits, par exemple 8 bits pour le registre AL (octet inférieur de AX) ou 16 bits pour le registre AX.

Les registres sont souvent **utilisés pour stocker des valeurs immédiates ou des adresses mémoire**, et sont utilisés dans des opérations telles que l'addition, la soustraction, la multiplication et la division.

Les registres sont généralement **divisés en deux catégories** : les registres généraux et les registres spécifiques.

Les registres généraux peuvent être utilisés pour stocker toutes sortes de données, tandis que les registres spécifiques ont des fonctions dédiées telles que le pointeur de pile, le compteur de programme, le registre de segment ou le registre de base d'adresse.

En langage assembleur, les instructions qui manipulent les registres sont appelées instructions de mouvement de données (MOV), qui copient les données d'un emplacement à un autre. Par exemple, la commande "MOV AX, BX" copie la valeur du registre BX dans le registre AX. Les instructions arithmétiques telles que ADD, SUB, MUL et DIV sont également utilisées pour effectuer des opérations mathématiques sur les registres.

Il est important de noter que les registres ont une taille limitée, ce qui signifie qu'ils ne peuvent stocker qu'une certaine quantité de données. Les données restantes doivent être stockées dans la mémoire vive. Il est également important de noter que les registres sont volatils, ce qui signifie

que leur contenu est perdu lorsqu'un système est éteint ou redémarré. Pour cette raison, les registres ne peuvent être utilisés que pour stocker des données temporaires ou des résultats intermédiaires.

II.3.La mémoire vive (RAM) et la mémoire morte (ROM)

En langage assembleur, la mémoire vive (RAM) et la mémoire morte (ROM) sont deux types de mémoire importants qui peuvent être utilisés pour stocker des programmes et des données.

La mémoire vive (RAM) est une mémoire volatile qui permet aux programmes de stocker des données temporairement pendant l'exécution. La RAM est généralement plus rapide que la mémoire morte et peut être écrite et lue plusieurs fois. En langage assembleur, l'accès à la RAM se fait généralement à l'aide d'adresses mémoire absolues ou relatives, en utilisant des instructions telles que MOV (déplacement) pour transférer des données entre des registres et des emplacements mémoire.

La mémoire morte (ROM) est une mémoire non volatile qui stocke des données en permanence, même après la coupure de l'alimentation. Les données stockées dans la ROM sont généralement des programmes de démarrage, des pilotes de périphériques ou des données qui doivent être conservées en permanence, telles que des informations de configuration. En langage assembleur, l'accès à la mémoire morte peut être effectué à l'aide d'adresses mémoire absolues ou relatives, tout comme pour la RAM.

Les deux types de mémoire ont des avantages et des inconvénients. La RAM offre une grande flexibilité et peut être réécrite plusieurs fois, mais est volatile et nécessite une alimentation constante pour conserver les données. La ROM est non volatile et conserve les données en permanence, mais ne peut pas être réécrite ou modifiée facilement.

Ensemble, la RAM et la ROM sont souvent utilisées pour stocker des programmes et des données dans les ordinateurs et autres systèmes électroniques. En langage assembleur, les programmeurs doivent comprendre comment accéder à ces types de mémoire et comment les utiliser efficacement pour développer des programmes et des systèmes fiables.

II.4.Les opérations élémentaires

En langage assembleur, les opérations élémentaires sont des instructions qui permettent de manipuler des données et d'effectuer des calculs. Voici quelques exemples d'opérations élémentaires courantes en assembleur :

a. MOV

Syntaxe :

```
MOV destination, source
```

Exemple :

```
MOV AX, 5 ; Copie la valeur 5 dans le registre AX
MOV BX, AX ; Copie la valeur de AX dans le registre BX
```

Explication : L'instruction MOV copie les données d'un emplacement mémoire à un autre. Dans cet exemple, la première instruction copie la valeur 5 dans le registre AX, tandis que la seconde instruction copie la valeur de AX dans le registre BX.

b. ADD/SUB :

Syntaxe :

```
ADD destination, source
SUB destination, source
```

Exemple :

```
MOV AX, 5
ADD AX, 3 ; Ajoute 3 à la valeur de AX
SUB AX, 2 ; Soustrait 2 à la valeur de AX
```

Explication : Les instructions ADD et SUB effectuent des opérations d'addition et de soustraction sur des données. Dans cet exemple, la première instruction copie la valeur 5 dans le registre AX, la seconde instruction ajoute 3 à la valeur de AX, et la troisième instruction soustrait 2 à la valeur de AX.

c. CMP :

Syntaxe :

```
CMP destination, source
```

Exemple :

```
MOV AX, 5
CMP AX, 5 ; Compare la valeur de AX à 5
```

Explication : L'instruction CMP compare deux valeurs et définit des indicateurs de drapeau en fonction du résultat. Dans cet exemple, la première instruction copie la valeur 5 dans le registre AX, tandis que la seconde instruction compare la valeur de AX à 5 et définit un indicateur de drapeau en fonction du résultat.

JMP :

Syntaxe :

```
JMP label
```

Exemple :

```
JMP label1 ; Sauter à la position de l'étiquette label1
label1:
```

Explication : L'instruction JMP saute à une adresse spécifiée dans le programme. Dans cet exemple, l'instruction JMP saute à la position de l'étiquette label1. L'étiquette est une référence à une adresse mémoire spécifique dans le programme.

III. La syntaxe et les éléments du langage assembleur

III.1. Les mnémoniques

En langage assembleur, les mnémoniques sont des symboles qui représentent des instructions ou des opérations spécifiques du processeur. Les mnémoniques sont utilisés pour faciliter la lecture et l'écriture des instructions en langage assembleur.

Chaque instruction est représentée par un mnémonique suivi de ses opérandes, qui peuvent être des registres, des valeurs immédiates ou des adresses mémoire. Les mnémoniques sont généralement des abréviations des noms des instructions ou des opérations qu'ils représentent.

Voici quelques exemples de mnémoniques courants en langage assembleur pour l'architecture x86

MNEMONIQUE	SIGNIFICATION
MOV	déplace une valeur d'un emplacement à un autre
ADD	ajoute une valeur à une autre
SUB	soustrait une valeur d'une autre
CMP	compare deux valeurs

AND	effectue un ET logique entre deux valeurs
OR	effectue un OU logique entre deux valeurs
XOR	effectue un OU exclusif logique entre deux valeurs
JMP	saute à une adresse mémoire spécifique
CALL	appelle une fonction ou une sous-routine
RET	retourne d'une fonction ou d'une sous-routine

III.2. Les opérandes

Les opérandes sont les données sur lesquelles les instructions agissent. Ils peuvent être des registres, des valeurs immédiates ou des adresses mémoire.

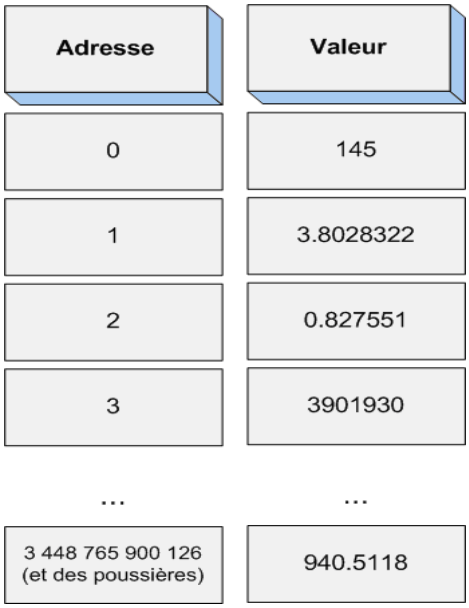
Les opérandes sont spécifiés après le mnémonique de l'instruction dans un ordre spécifique, qui peut varier selon l'architecture du processeur ou le langage assembleur utilisé.

Voici quelques exemples d'opérandes courants en langage assembleur pour l'architecture x86 :

Registres : Les registres peuvent contenir des valeurs entières, des pointeurs de mémoire et d'autres types de données.

Valeurs immédiates : les valeurs immédiates sont des constantes spécifiées directement dans l'instruction. Les valeurs immédiates peuvent être utilisées comme opérandes pour les opérations arithmétiques, les déplacements de données et d'autres types d'opérations.

Adresses mémoire : les adresses mémoire sont des emplacements de stockage de données dans la mémoire vive. Les adresses mémoire sont spécifiées sous forme de valeurs hexadécimales ou de symboles, et peuvent être utilisées comme opérandes pour les opérations de chargement et de stockage de données.



III.3. Les directives

Les directives (ou pseudo-instructions) sont **des instructions qui ne sont pas exécutées par le processeur**, mais qui sont utilisées par l'assembleur pour générer du code machine ou effectuer d'autres tâches lors de l'assemblage du programme.

Les directives sont généralement spécifiques à chaque assembleur, mais il existe certaines directives communes que l'on retrouve dans de nombreux langages assembleur.

Voici quelques exemples de directives courantes en langage assembleur :

.data : cette directive indique au compilateur que les lignes de code suivantes contiennent des données à stocker en mémoire. Par exemple :

```
.data
variable1:  dw 10  ; stocke un entier de 2 octets (word)
variable2:  db  'hello', 0  ; stocke une chaîne de caractères suivie d'un octet nul
```

.text : cette directive indique au compilateur que les lignes de code suivantes contiennent du code à exécuter. Par exemple :

```
.text
main:
   mov eax, 1 ; met la valeur 1 dans le registre eax
   int 0x80  ; appelle la fonction système correspondant au code d'interruption 0x80
   ret       ; retourne au système d'exploitation
```

.globl : cette directive indique au compilateur qu'un symbole est défini dans un autre fichier source et doit être rendu disponible pour d'autres fichiers qui sont liés ensemble pour créer l'exécutable final. Par exemple :

```
 .globl  fonction1
fonction1:
   ; code de la fonction
```

.equ : cette directive définit une constante avec une valeur donnée. Par exemple :

```
.equ   TAILLE_TABLE, 100
```

Ces directives ne sont que quelques exemples parmi les nombreuses possibilités offertes par les langages assembleur.

Les directives sont un outil puissant pour les programmeurs en langage assembleur, car elles permettent de gérer de manière précise les données et le code machine générés par l'assembleur.

III.4. Les macros

Les macros en langage assembleur sont des sections de code qui sont écrites une fois et peuvent être réutilisées à plusieurs reprises dans un programme. Elles sont **similaires aux fonctions dans les langages de programmation de haut niveau**, mais sont spécifiques au langage assembleur.

Les macros sont définies à l'aide de la directive **MACRO**, suivie du nom de la macro et des paramètres. Le code de la macro est ensuite écrit sous forme de bloc de code normal, en utilisant les paramètres définis pour personnaliser le code. La directive **ENDM** est utilisée pour indiquer la fin de la macro.

Voici un exemple simple de macro en langage assembleur pour effectuer l'addition de deux nombres :

```
MACRO ADD_TWO_NUMBERS num1, num2
   mov ax, num1
   add ax, num2
ENDM
```

Cette macro prend deux paramètres, num1 et num2, et effectue l'addition de ces deux nombres en utilisant les instructions mov et add. Elle peut être appelée dans le code principal du programme en utilisant le nom de la macro suivi des deux nombres à ajouter :

```
ADD_TWO_NUMBERS 10, 20
```

Cette instruction sera remplacée par le code de la macro, avec num1 évalué à 10 et num2 évalué à 20.

Les macros sont très utiles pour éviter la duplication de code et pour rendre le code plus lisible et plus facile à maintenir.

IV. Les instructions du langage assembleur

IV.1. Les instructions de transfert de données

Les instructions de transfert de données sont utilisées pour déplacer des données d'une position mémoire à une autre position mémoire, ou pour transférer des données entre des registres.

Voici quelques exemples d'instructions de transfert de données courantes :

MOV (Move) : Cette instruction copie des données d'une source à une destination. La syntaxe générale de l'instruction MOV est la suivante :

```
MOV destination, source
```

Par exemple, pour copier le contenu du registre AX dans le registre BX, on peut utiliser l'instruction suivante :

```
MOV BX, AX
```

XCHG (Exchange) : Cette instruction échange le contenu de deux registres ou d'une position mémoire et d'un registre. La syntaxe générale de l'instruction XCHG est la suivante :

```
XCHG destination, source
```

Par exemple, pour échanger le contenu des registres AX et BX, on peut utiliser l'instruction suivante :

```
XCHG AX, BX
```

LEA (Load Effective Address) : Cette instruction charge l'adresse effective d'une position mémoire dans un registre. La syntaxe générale de l'instruction LEA est la suivante :

```
LEA destination, source
```

Par exemple, pour charger l'adresse de la variable var1 dans le registre AX, on peut utiliser l'instruction suivante :

```
LEA AX, var1
```

PUSH (Push onto Stack) : Cette instruction pousse des données sur la pile, en décrémentant le pointeur de pile et en copiant les données de la source sur la position mémoire pointée par le pointeur de pile. La syntaxe générale de l'instruction PUSH est la suivante :

```
PUSH source
```

Par exemple, pour pousser le contenu du registre AX sur la pile, on peut utiliser l'instruction suivante :

```
PUSH AX
```

POP (Pop from Stack) : Cette instruction retire des données de la pile, en copiant les données de la position mémoire pointée par le pointeur de pile dans la destination, puis en incrémentant le pointeur de pile. La syntaxe générale de l'instruction POP est la suivante :

```
POP destination
```

Par exemple, pour retirer une donnée de la pile et la stocker dans le registre BX, on peut utiliser l'instruction suivante :

```
POP BX
```

Ces instructions de transfert de données sont essentielles en langage assembleur pour manipuler les données et les stocker en mémoire.

Cependant, il est important de comprendre comment elles fonctionnent en détail, car certaines d'entre elles peuvent affecter les drapeaux de l'unité arithmétique et logique (ALU) ou avoir d'autres effets secondaires sur le fonctionnement du programme.

IV.2. Les instructions arithmétiques

Les instructions arithmétiques en langage assembleur permettent d'effectuer des opérations mathématiques sur des données stockées dans des registres ou des emplacements mémoire.

Voici quelques exemples d'instructions arithmétiques courantes :

ADD : Cette instruction permet d'additionner deux opérandes et de stocker le résultat dans le premier opérande. Par exemple, pour ajouter la valeur de EBX à la valeur de EAX et stocker le résultat dans EAX, on peut utiliser l'instruction suivante :

```
ADD EAX, EBX
```

SUB : Cette instruction permet de soustraire un opérande d'un autre et de stocker le résultat dans le premier opérande. Par exemple, pour soustraire la valeur de EDX de la valeur de EAX et stocker le résultat dans EAX, on peut utiliser l'instruction suivante :

```
SUB EAX, EDX
```

INC : Cette instruction permet d'incrémenter la valeur d'un opérande de 1. Par exemple, pour incrémenter la valeur de ECX de 1, on peut utiliser l'instruction suivante :

```
INC ECX
```

DEC : Cette instruction permet de décrémenter la valeur d'un opérande de 1. Par exemple, pour décrémenter la valeur de EDI de 1, on peut utiliser l'instruction suivante :

```
DEC EDI
```

MUL : Cette instruction permet de multiplier deux opérandes non signés et de stocker le résultat dans un registre spécial appelé EAX. Par exemple, pour multiplier la valeur de EBX par la valeur de ESI et stocker le résultat dans EAX, on peut utiliser l'instruction suivante :

```
MUL ESI, EBX
```

DIV : Cette instruction permet de diviser la valeur contenue dans EAX par une valeur non signée et de stocker le quotient dans EAX et le reste dans un registre spécial appelé EDX. Par exemple, pour diviser la valeur de EAX par la valeur 5 et stocker le quotient dans EAX et le reste dans EDX, on peut utiliser l'instruction suivante :

```
MOV EDX, 0
MOV EAX, 100
DIV DWORD PTR [5]
```

Ces instructions ne sont pas exhaustives, mais elles donnent une idée des opérations arithmétiques de base que l'on peut effectuer en langage assembleur.

IV.3. Les instructions de contrôle de flux

Les instructions de contrôle en langage assembleur permettent de gérer le flux d'exécution du programme. Elles incluent les instructions de saut, les instructions de comparaison, et les instructions de boucle. Voici quelques exemples d'instructions de contrôle courantes :

JMP : Cette instruction permet de sauter à une autre partie du programme en spécifiant une adresse de destination. Par exemple, pour sauter à l'adresse 0x12345678, on peut utiliser l'instruction suivante :

```
JMP 0x12345678
```

CMP : Cette instruction permet de comparer deux valeurs en soustrayant l'une de l'autre et en modifiant les indicateurs de drapeau en conséquence. Par exemple, pour comparer la valeur de EAX avec celle de EBX, on peut utiliser l'instruction suivante :

```
CMP EAX, EBX
```

JZ/JNZ : Ces instructions permettent de sauter à une adresse de destination si le drapeau ZF (Zero Flag) est défini ou non-défini respectivement. Par exemple, pour sauter à l'adresse 0x12345678 si EAX est égal à EBX, on peut utiliser l'instruction suivante :

```
CMP EAX, EBX
JZ 0x12345678
```

LOOP : Cette instruction permet de répéter une boucle un certain nombre de fois en décrémentant le compteur de boucle ECX à chaque itération. Par exemple, pour répéter une boucle jusqu'à ce que ECX atteigne zéro, on peut utiliser l'instruction suivante :

```
LOOP loop_label
```

CALL/RET : Ces instructions permettent d'appeler une procédure ou une fonction et de revenir à l'instruction suivante après l'exécution de la procédure. Par exemple, pour appeler la fonction my_func, on peut utiliser l'instruction suivante :

```
CALL my_func
```

Et pour revenir à l'instruction suivante après l'exécution de my_func, on peut utiliser l'instruction suivante à la fin de la fonction :

```
RET
```

V. Les outils et les environnements de développement pour programmer en assembleur

V.1. Les éditeurs de texte

Il n'existe pas d'éditeur de texte spécifique au langage assembleur, car les fichiers de code assembleur sont en réalité des fichiers texte standard. Toutefois, certains éditeurs de texte sont mieux adaptés pour écrire du code assembleur que d'autres en raison de leurs fonctionnalités et de leur prise en charge des langages de programmation.

Voici quelques éditeurs de texte populaires pour programmer en langage assembleur :

Visual Studio Code : Cet éditeur de texte open source est très populaire parmi les développeurs de tous les horizons, y compris ceux qui programment en assembleur. Il dispose de fonctionnalités telles que la coloration syntaxique, l'auto-complétion et la prise en charge de nombreux plugins et extensions qui peuvent aider à simplifier le processus de développement.

Sublime Text : Cet éditeur de texte est également très populaire parmi les développeurs et peut être utilisé pour programmer en assembleur. Il est rapide, léger et dispose de nombreuses fonctionnalités telles que la coloration syntaxique, la recherche et le remplacement de texte, ainsi que la prise en charge de plugins tiers.

Notepad++ : Ce petit éditeur de texte est populaire parmi les programmeurs de tous les niveaux et est particulièrement utile pour les développeurs qui travaillent sur des ordinateurs moins puissants. Il dispose de fonctionnalités telles que la coloration syntaxique, la prise en charge des macros et la comparaison de fichiers.

Emacs : Cet éditeur de texte est un choix populaire pour les programmeurs qui aiment personnaliser leur environnement de développement. Il dispose de nombreuses fonctionnalités utiles pour la programmation en assembleur, telles que la coloration syntaxique et la prise en charge des macros.

Vim : Cet éditeur de texte est un choix populaire pour les programmeurs expérimentés qui aiment travailler en ligne de commande. Il dispose de nombreuses fonctionnalités telles que la coloration syntaxique et la prise en charge des plugins qui peuvent aider à simplifier le processus de développement.

En général, tous les éditeurs de texte avec une coloration syntaxique pour l'assembleur peuvent être utilisés pour programmer en assembleur, mais certains éditeurs de texte offrent des fonctionnalités supplémentaires pour faciliter le développement.

V.2. Les assembleurs

Un assembleur est un programme qui permet de traduire le code source écrit en langage assembleur en langage machine. Le code source en langage assembleur est généralement plus facile à comprendre pour les programmeurs que le langage machine, qui est composé de 0 et de 1.

Il existe de nombreux assembleurs différents pour différentes architectures de processeurs, notamment :

NASM (Netwide Assembler) : C'est un assembleur open source et multi-plateforme pour les processeurs Intel x86 et x86-64. Il est très populaire auprès des développeurs qui travaillent sur des systèmes d'exploitation bas niveau, tels que les noyaux de système d'exploitation.

MASM (Microsoft Macro Assembler) : C'est un assembleur propriétaire pour les processeurs Intel x86. Il est souvent utilisé avec les outils de développement Microsoft, tels que Visual Studio.

TASM (Turbo Assembler) : C'est un assembleur propriétaire pour les processeurs Intel x86. Il est souvent utilisé pour développer des applications en mode réel MS-DOS.

FASM (Flat Assembler) : C'est un assembleur open source pour les processeurs Intel x86 et x86-64. Il est connu pour être très rapide et efficace.

GAS (GNU Assembler) : C'est un assembleur open source pour de nombreuses architectures de processeurs, y compris ARM, MIPS et PowerPC. Il est souvent utilisé avec les outils de développement GNU, tels que GCC.

HLA (High Level Assembler) : C'est un assembleur propriétaire qui prend en charge un langage assembleur de haut niveau pour les processeurs Intel x86. Il est conçu pour faciliter la programmation en assembleur pour les programmeurs qui préfèrent les langages de programmation de haut niveau.

Chaque assembleur a ses propres caractéristiques et sa propre syntaxe, donc il est important pour un programmeur de choisir l'assembleur qui convient le mieux à son projet et à son expérience en programmation en assembleur.

V.3. Les éditeurs de lien (Linker)

Les éditeurs de liens sont des programmes qui sont utilisés pour lier les différents modules d'un programme en un seul exécutable ou bibliothèque. En langage assembleur, les éditeurs de liens sont souvent utilisés pour combiner plusieurs fichiers objets en un seul exécutable.

Les éditeurs de liens peuvent effectuer différentes tâches, telles que la résolution des symboles, la vérification des dépendances des bibliothèques et la création de tables de réadressage. Ils permettent également de spécifier des options de lien, telles que l'ordre des segments, les adresses de début et de fin, etc.

Voici quelques exemples d'éditeurs de liens populaires en langage assembleur :

ld : ld est l'éditeur de liens par défaut sur la plupart des systèmes Unix. Il peut être utilisé pour lier des fichiers objets en un exécutable ou une bibliothèque.

Microsoft Link : Microsoft Link est l'éditeur de liens fourni avec la suite de développement Microsoft Visual Studio. Il est utilisé pour lier des fichiers objets en un exécutable ou une bibliothèque sur les systèmes Windows.

GNU Binutils : GNU Binutils est une collection d'outils de développement, y compris un éditeur de liens, qui est souvent utilisé avec le compilateur GNU GCC pour développer des programmes en langage assembleur.

Tlink : Tlink est un éditeur de liens fourni avec le compilateur Turbo Assembler de Borland pour les systèmes DOS.

Ces éditeurs de liens peuvent être utilisés pour lier les différents modules d'un programme en un seul exécutable ou bibliothèque. Les programmeurs en langage assembleur doivent souvent utiliser des éditeurs de liens pour lier les différents fichiers objets générés par leur programme en un seul fichier exécutable ou bibliothèque.

V.4. Les débogueurs

Un débogueur est un outil logiciel qui permet de détecter et de corriger les erreurs dans le code source d'un programme. Les débogueurs offrent des fonctionnalités telles que la possibilité de suivre l'exécution du programme, de mettre en pause l'exécution à un point donné, de vérifier les valeurs des variables, d'inspecter la mémoire, de modifier les valeurs des variables et d'exécuter le code pas à pas.

Voici quelques exemples de débogueurs populaires :

GDB (GNU Debugger) : C'est un débogueur open source pour de nombreuses langues de programmation, y compris C, C++, Ada et Fortran. Il est souvent utilisé avec les outils de développement GNU, tels que GCC.

WinDbg : C'est un débogueur propriétaire pour les systèmes d'exploitation Windows. Il est souvent utilisé pour déboguer des pilotes de périphériques et d'autres composants système bas niveau.

LLDB : C'est un débogueur open source pour les langues de programmation C, C++ et Objective-C. Il est souvent utilisé avec l'IDE Xcode sur macOS et iOS.

OllyDbg : C'est un débogueur propriétaire pour les systèmes d'exploitation Windows. Il est souvent utilisé pour déboguer des applications compilées en langage assembleur.

IDA Pro : C'est un débogueur et désassembleur propriétaire pour de nombreuses architectures de processeurs. Il est souvent utilisé pour l'analyse de logiciels malveillants et la rétro-ingénierie de programmes.

Les débogueurs sont des outils précieux pour les programmeurs lorsqu'ils cherchent à identifier et à corriger les erreurs dans leur code. Les fonctionnalités avancées des débogueurs permettent de localiser les erreurs plus rapidement et plus efficacement, ce qui peut considérablement réduire le temps nécessaire pour résoudre les problèmes dans le code.

V.5. Un simulateur de processeur

Un simulateur de processeur est un outil logiciel qui permet de simuler le fonctionnement d'un processeur. Les simulateurs de processeur sont souvent utilisés pour le développement de logiciels bas niveau, tels que les systèmes d'exploitation et les pilotes de périphériques.

Voici quelques exemples de simulateurs de processeur populaires :

QEMU : C'est un émulateur open source qui prend en charge de nombreuses architectures de processeurs, y compris ARM, MIPS, PowerPC et x86. QEMU peut être utilisé pour émuler des systèmes d'exploitation invités, tels que Linux, Windows et macOS.

Bochs : C'est un émulateur open source qui prend en charge de nombreuses architectures de processeurs, y compris x86, x86-64 et PowerPC. Bochs peut être utilisé pour émuler des systèmes d'exploitation invités, tels que Linux, Windows et FreeBSD.

VirtualBox : C'est un hyperviseur propriétaire qui prend en charge de nombreuses architectures de processeurs, y compris x86, x86-64 et ARM. VirtualBox peut être utilisé pour exécuter des systèmes d'exploitation invités, tels que Linux, Windows et macOS.

gem5 : C'est un simulateur open source qui prend en charge de nombreuses architectures de processeurs, y compris ARM, MIPS et x86. gem5 est souvent utilisé pour la recherche et le développement de nouveaux processeurs et systèmes d'exploitation.

Simics : C'est un simulateur propriétaire qui prend en charge de nombreuses architectures de processeurs, y compris ARM, MIPS et x86. Simics est souvent utilisé pour le développement de systèmes embarqués et de logiciels bas niveau.

Les simulateurs de processeur sont des outils puissants pour les développeurs de logiciels bas niveau qui cherchent à tester et à valider leur code sur différentes architectures de processeurs sans avoir à utiliser du matériel physique. Ils permettent également de simuler des scénarios de test difficiles à reproduire sur du matériel réel, ce qui peut faciliter la détection et la correction des bugs.

VI. Mise au point d'un programme en assembleur

VI.1. Affichier « Bonjour a tous »

Voici un exemple de programme en assembleur x86 qui affiche "Bonjour à tous" en utilisant la fonctionnalité d'affichage de la console du système d'exploitation :

```
section .data
    message db 'Bonjour a tous',0Ah ; message à afficher avec un saut de ligne
section .text
    global _start
_start:
    ; écrire le message à la console
    mov eax, 4          ; numéro du service système pour écrire sur la console
    mov ebx, 1          ; descripteur de fichier pour la console (1 = stdout)
    mov ecx, message    ; adresse du message à afficher
    mov edx, 14         ; longueur du message (y compris le saut de ligne)
    int 0x80            ; appeler le système d'exploitation pour écrire le
message

    ; quitter le programme
    mov eax, 1          ; numéro du service système pour quitter
    xor ebx, ebx        ; code de sortie du programme (0)
    int 0x80            ; appeler le système d'exploitation pour quitter
```

Pour assembler et exécuter ce programme en NASM sur un système d'exploitation Linux, vous pouvez suivre ces étapes :

[1]. Copiez le code ci-dessus dans un fichier nommé **bonjour.asm**

[2]. Ouvrez un terminal et placez-vous dans le même répertoire que le fichier bonjour.asm.

[3]. Installez le compilateur NASM si ce n'est pas déjà fait : **sudo apt-get install nasm (pour Ubuntu/Debian)**.

[4]. Assemblez le programme en tapant **nasm -f elf bonjour.asm**

[5]. Liez le programme en tapant **ld -m elf_i386 -s -o bonjour bonjour.o**

[6]. Exécutez le programme en tapant **./bonjour**. Vous devriez voir le message "Bonjour a tous" s'afficher sur la console.

Notez que la syntaxe en assembleur peut varier légèrement en fonction de l'architecture et du système d'exploitation utilisé.

VI.2. Calcul la somme de deux nombre

Voici un exemple de programme en assembleur x86 qui calcule la somme de deux nombres et l'affiche à l'écran en utilisant la fonctionnalité d'affichage de la console du système d'exploitation:

```
; ===========================
; quelques definitions utiles
; ===========================
%define __NR_exit       1
%define __NR_read       3
%define __NR_write      4

%define STDIN_FILENO    0
%define STDOUT_FILENO   1

; ======================
; quelques macros utiles
; ======================

; ------------------------------------------------
; execute un appel systeme avec 1 parametre (ebx)
;
%macro syscall2 2
    mov eax, %1
    mov ebx, %2
    int 0x80
%endmacro

; ------------------------------------------------------------
; execute un appel systeme avec 3 parametres (ebx, ecx, edx)
```

```
;
%macro syscall4 4
    mov eax, %1
    mov ebx, %2
    mov ecx, %3
    mov edx, %4
    int 0x80
%endmacro

; -------------------------------------------------------
; effectue la conversion alphabetique -> entier numerique
;
; prend 2 parametres :
; - l'adresse de la chaine qui represente le nombre a convertir (ex. nb1_str)
;   la chaine doit obligatoirement terminer par un \n (0Ah ou 10 en decimal)
; - l'adresse d'un emplacement libre pour stocker le nombre
;
; modifie les registres eax, edx et esi
;
%macro atoi 2
        mov esi, %1
        xor eax, eax
        xor edx, edx
    %%loop:
        mov dl, byte [esi]          ; on recupere le caractere du chiffre
courrant dans edx
        cmp dl, 10                  ; si dl contient un retour chariot on a fini
de traiter la chaine
        je %%finish
        lea eax, [eax * 4 + eax]    ; eax = eax * 5
        add eax, eax                ; eax = eax + eax, donc a l'arrivee on a
multiplie eax par 10 avec ces deux lignes
        add esi, 1                  ; on incremente le pointeur sur la chaine
pour passer au chiffre suivant
        and dl, 0x0F                ; c'est une astuce pour recuperer la valeur
decimale du chiffre, ca s'explique par la representation binaire, on aurait pu
faire un sub 48 a la place
        add eax, edx                ; on additionne la valeur du chiffre a eax
(qui a ete multiplie par 10 avant remember ?)
        jmp %%loop                  ; et on boucle
    %%finish:
        mov [%2], eax
%endmacro

; -------------------------------------------------------
```

```
; effectue la conversion entier numerique -> chaine ascii
;
; prend 3 parametres :
; - l'adresse du dword contenant le nombre a convertir
; - l'adresse d'un buffer libre pour stocker la chaine convertie
; - l'adresse d'un emplacement pour stocker la longueur de la chaine convertie
;
; modifie les registres eax, ebx, ecx et edi
;
%macro itoa 3
        mov eax, [%1]               ; on recupere le nombre la ou il est stocke
        mov edi, %2                 ; on recupere l'adresse du buffer de
destination
        mov ebx, 10                 ; diviseur = 10
        xor ecx, ecx
    %%first_loop:
        xor edx, edx
        div ebx                     ; on divise eax par 10, le quotient va dans
eax, le reste va dans edx
        push dx                     ; on stocke le reste, sachant qu'en divisant
par 10 a chaque fois on stocke les chiffres dans l'ordre inverse -> 3,2,1
        inc cl                      ; longueur (de la chaine) = longueur + 1
        test eax, eax               ; tant que eax != 0, c'est a dire tant que
qu'il y a un quotient qu'on peut diviser et nous donner un reste
        jnz %%first_loop
        mov [%3], ecx               ; ecx contient la longueur de la chaine, on
stocke la valeur pour pas la perdre
    %%second_loop:                  ; la seconde boucle permet de convertir
decimal -> ascii en ajoutant 48, et de remettre les chiffre dans l'ordre attendu
(grace aux pop)
        pop ax                      ; donc on recupere le dernier chiffre traite
        or al, 00110000b            ; on lui ajoute 48 (la encore c'est un peu
astucieux)
        mov byte [edi], al          ; et on le stocke dans le buffer de
destination, de gauche a droite donc dans l'ordre -> "123"
        inc edi
        loop %%second_loop          ; jusqu'a ce que ecx = 0, c'est a dire qu'on
ait traite toute la chaine
        mov byte [edi], 0
%endmacro

; ============
; les sections
; ============
section .data
```

```
    msg1:       db "1er nombre: "
    len1        equ $ - msg1

    msg2:       db "2eme nombre: "
    len2        equ $ - msg2

    msg3:       db "resultat: "
    len3        equ $ - msg3

    msg4        db 10
    len4        equ $ - msg4

section .bss
    ; ici on reserve de quoi stocker les chaines correspondantes aux differents
nombres qu'on manipule
    nb1_str:    resb 4
    nb2_str:    resb 4
    sum_str:    resb 5

    ; tandis qu'ici on reserve de quoi stocker leur valeur, qui tiendra dans un
registre, donc au maximum un dword
    nb1_dec:    resd 1
    nb2_dec:    resd 1
    sum_dec:    resd 1

    ; et ici on stockera la longueur de la chaine sum_str, c'est un nombre, on
reserve un dword
    sum_len:    resd 1

; =================
; Point d'entrée du programme
; =================
section .text
    global _start

    _start:
        syscall4 __NR_write, STDOUT_FILENO, msg1, len1  ; on affiche le premier
message
        syscall4 __NR_read, STDIN_FILENO, nb1_str, 4    ; on lit le premier
nombre
        atoi nb1_str, nb1_dec                           ; on convertit l'entree
utilisateur

        syscall4 __NR_write, STDOUT_FILENO, msg2, len2  ; on affiche le deuxieme
message
```

```
        syscall4 __NR_read, STDIN_FILENO, nb2_str, 4    ; on lit le deuxieme
nombre
        atoi nb2_str, nb2_dec                           ; on passe a la
moulinette pareil que pour le premier

        mov eax, [nb1_dec]                              ; on recupere le 1er
nombre
        mov ebx, [nb2_dec]                              ; on recupere le 2eme
        add eax, ebx                                    ; on additionne
        mov [sum_dec], eax                              ; on stocke le resultat
de l'addition
        itoa sum_dec, sum_str, sum_len                  ; et on convertit, mais
dans l'autre sens cette fois-ci

        syscall4 __NR_write, STDOUT_FILENO, msg3, len3          ; enfin on
affiche, le message d'abord
        syscall4 __NR_write, STDOUT_FILENO, sum_str, [sum_len]  ; suivi du
resultat
        syscall4 __NR_write, STDOUT_FILENO, msg4, len4          ; et du retour a
la ligne

        syscall2 __NR_exit, 0                           ; et on quitte
```

Pour exécuter sur NASM

```
#nasm -f elf somme.asm
#ld -m elf_i386 -o somme somme.o
#./somme
```

VI.3. Calcul d'un factoriel

Voici un exemple de programme en assembleur x64 qui calcule le factoriel d'un nombre :

```
Bits 64
section .data
    prompt db 'Entrez un entier positif: '
    error db 'Erreur: le nombre doit être positif.', 10
    resultmsg db 'La factorielle de %d est: %d', 10
    newline db 10
    num dq 0
```

```
    fact dq 1

section .text
    global _start

    extern printf

_start:
    ; Afficher le prompt et lire le nombre saisi
    mov rax, 4
    mov rdi, 1
    mov rsi, prompt
    mov rdx, 28
    syscall
    mov rax, 0
    mov rdi, 0
    mov rsi, num
    mov rdx, 8
    syscall

    ; Vérifier que le nombre est positif
    cmp qword [num], 0
    jl error

    ; Calculer la factorielle
    mov rax, [num]
    mov rbx, 1
loop:
    cmp rax, 0
    je end
    imul rbx, rax
    dec rax
    jmp loop

    ; Afficher le résultat
end:
    mov rdi, resultmsg
    mov rsi, num
    mov rdx, rbx
    xor rcx, rcx
    call printf

    ; Terminer le programme
    mov rax, 60
    xor rdi, rdi
```

```
    syscall

error:
    ; Afficher un message d'erreur et terminer le programme
    mov rax, 4
    mov rdi, 1
    mov rsi, error
    mov rdx, 32
    syscall
    mov rax, 60
    xor rdi, rdi
    syscall
```

Pour executer ce code en utilisat NASM sur linux en supposant que le nom du fichier est « factoriel.asm » :

```
#nasm -f elf64 -o factoriel.o factoriel.asm
#gcc -o factoriel factoriel.o -lc
#./factoriel
```

VII. Les avantages et les limites de l'utilisation du langage assembleur

VI.1. Les avantages

Le langage assembleur est un **langage de programmation de bas niveau qui permet de communiquer directement avec la machine en utilisant des instructions spécifiques au processeur**. Voici quelques avantages de l'utilisation du langage assembleur :

Performances : Le code assembleur peut être optimisé pour une performance maximale, car il permet un contrôle précis des instructions exécutées par la machine. Le code assembleur peut donc être particulièrement utile pour les programmes exigeants en termes de performances, tels que les jeux vidéo, les applications de traitement vidéo ou les calculs scientifiques complexes.

Contrôle : Le programmeur a un contrôle total sur le comportement du processeur, ce qui est particulièrement important pour les programmes qui nécessitent un accès direct à la mémoire, tels que les pilotes de périphériques.

Flexibilité : Le langage assembleur permet de manipuler directement les bits et les octets de données, offrant une flexibilité totale pour la gestion des données.

Éducation : L'apprentissage du langage assembleur peut aider les programmeurs à mieux comprendre le fonctionnement interne de la machine, ce qui peut être bénéfique pour les programmes de niveau supérieur.

VI.2. Les inconvénients

Bien que le langage assembleur présente des avantages significatifs en termes de performances, de contrôle et de flexibilité, il présente également certains inconvénients, notamment :

Complexité : Le langage assembleur est un langage de bas niveau, ce qui signifie qu'il nécessite une compréhension détaillée du fonctionnement interne de la machine, de ses registres, de ses instructions, etc. **Il est souvent considéré comme l'un des langages de programmation les plus complexes à apprendre et à maîtriser.**

Maintenance : Le code assembleur est souvent difficile à maintenir, car il est généralement écrit pour un processeur spécifique et peut nécessiter des modifications importantes pour fonctionner sur une autre architecture de processeur. Le code assembleur peut également être difficile à lire et à comprendre, ce qui peut compliquer le processus de débogage.

Portabilité : Le code assembleur n'est pas portable entre les différents types de processeurs, ce qui signifie qu'il doit être réécrit pour chaque processeur cible. **Cela peut rendre le développement de logiciels multiplateformes plus difficile et plus coûteux.**

Temps de développement : Le code assembleur peut prendre plus de temps à développer que les langages de programmation de plus haut niveau, car il nécessite un plus grand effort de programmation pour écrire chaque instruction.

Sécurité : Les **erreurs de programmation en assembleur peuvent être graves et potentiellement dangereuses pour la sécurité**, car une seule erreur peut avoir un impact important sur le comportement du processeur.

Dans l'ensemble, le choix du langage de programmation dépend des besoins spécifiques du projet et des compétences des développeurs.

Le langage **assembleur peut être utile dans des situations où la vitesse et le contrôle précis sont essentiels**, mais peut-être moins pratique dans d'autres situations où la portabilité et la facilité de développement sont plus importantes.

VIII. Domaines d'applications du langage assembleur

Le langage assembleur est un langage de programmation de bas niveau qui peut être utilisé dans une variété d'applications où la performance et la maîtrise du processeur sont essentielles. Voici quelques exemples d'applications du langage assembleur :

Systèmes d'exploitation : Les systèmes d'exploitation, tels que Windows et Linux, contiennent souvent du code assembleur pour les parties les plus critiques du système, telles que le noyau du système d'exploitation, les pilotes de périphériques, etc.

Jeux vidéo : Les jeux vidéo nécessitent souvent des performances maximales pour offrir une expérience de jeu fluide. Le code assembleur peut être utilisé pour optimiser les parties critiques du code du jeu, telles que les boucles de rendu, les calculs de physique, etc.

Applications de traitement d'images et de vidéos : Les applications de traitement d'images et de vidéos nécessitent souvent des performances élevées pour effectuer des calculs complexes sur les images et les vidéos. Le code assembleur peut être utilisé pour optimiser les boucles de traitement d'images, les calculs de convolution, les transformations de Fourier, etc.

Pilotes de périphériques : Les pilotes de périphériques permettent aux systèmes d'exploitation de communiquer avec les périphériques matériels, tels que les cartes réseau, les cartes graphiques, les imprimantes, etc. Le code assembleur peut être utilisé pour accéder directement au matériel et optimiser les performances.

Microcontrôleurs : Les microcontrôleurs sont des ordinateurs intégrés qui sont utilisés dans une variété d'applications, telles que les systèmes embarqués, les contrôleurs de moteurs, les contrôleurs de température, etc. Le langage assembleur peut être utilisé pour programmer des microcontrôleurs en fournissant un contrôle fin sur les entrées et les sorties du microcontrôleur.

En effet, le langage assembleur est utilisé dans une variété d'applications où la performance et la maîtrise du processeur sont essentielles.

Il est particulièrement utile dans les systèmes critiques et les applications exigeantes en termes de performances, où chaque cycle CPU compte.

IX. Conclusion

VIII.1. Résumé des points clé

Voici quelques points clés à retenir concernant le langage assembleur :

- Le langage assembleur est un langage de programmation de bas niveau qui permet de contrôler directement le processeur d'un ordinateur.
- Il est généralement utilisé pour écrire des programmes système, des pilotes de périphériques et des programmes nécessitant une haute performance.
- Les instructions en langage assembleur sont écrites sous forme de code mnémonique, qui correspond à des instructions machine spécifiques.
- Le langage assembleur utilise des registres pour stocker les données et les résultats des opérations, ainsi que des modes d'adressage pour accéder aux données.
- Les instructions en langage assembleur comprennent des instructions arithmétiques, des instructions de déplacement et des instructions de contrôle.
- Les avantages du langage assembleur sont la vitesse d'exécution, le contrôle précis du processeur et la compacité du code.
- Les inconvénients du langage assembleur sont la complexité, la limitation de la portabilité et le risque d'erreurs dans le code.

VIII.2. Perspective d'avenirs

Le langage assembleur est un langage de programmation de bas niveau qui a été utilisé depuis des décennies pour écrire des programmes système et des pilotes de périphériques, ainsi que pour des tâches nécessitant une haute performance.

Bien que le langage assembleur ne soit pas aussi populaire qu'auparavant en raison des avancées technologiques et de la disponibilité de langages de programmation plus élevés de niveau, il reste toujours pertinent dans certaines industries.

Les perspectives d'avenir pour le langage assembleur dépendent des besoins de l'industrie informatique.

Il est possible que certaines industries continuent à utiliser le langage assembleur pour des tâches nécessitant une haute performance, telles que les programmes embarqués, les systèmes d'exploitation et les logiciels de bas niveau.

Cependant, avec l'essor de l'informatique en nuage et de l'Internet des objets, des langages de programmation plus modernes et plus accessibles sont de plus en plus utilisés.

En résumé, le langage assembleur continuera probablement à être utilisé pour des tâches spécifiques nécessitant une haute performance et un contrôle précis, mais il ne sera probablement pas utilisé aussi largement qu'auparavant.

Les développeurs peuvent également utiliser des langages de programmation de plus haut niveau pour la plupart des tâches de programmation, car ils offrent plus de flexibilité et de portabilité.