Initiation à l'assembleur 8086 sur système GNU/Linux

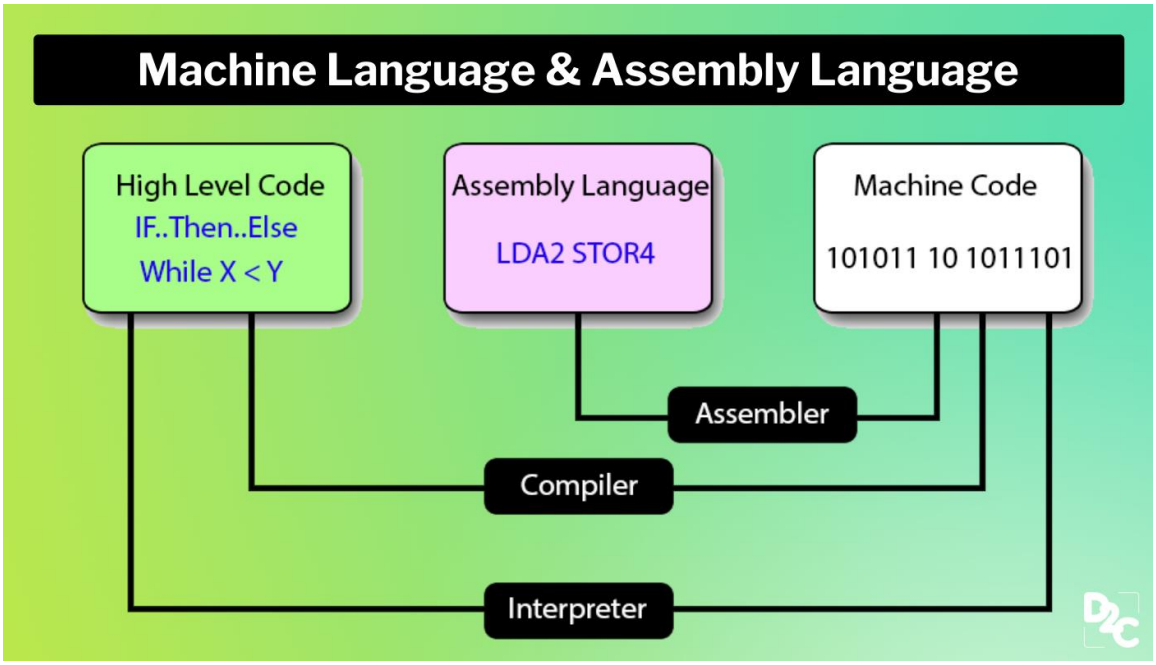
D'après Rémi HOARAU

Partie I : Introduction et prérequis d'installation

I. Pourquoi apprendre un langage d'assembleur ?

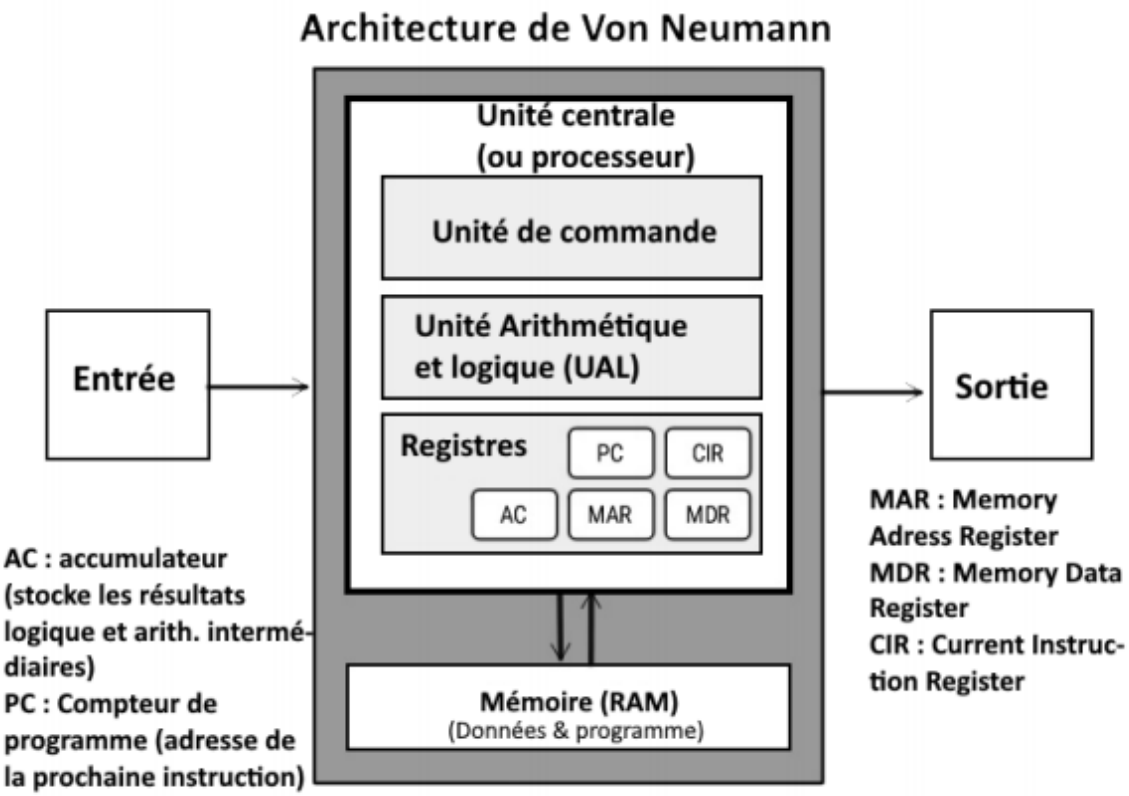
Il existe des langages de haut niveau tels que PHP, Python, Perl, Javascript, etc... Ils sont dis de haut niveau puisque la syntaxe de leurs instructions sont proches du langage humain. Cela implique une meilleure compréhension du code pour l'humain mais une difficulté de compréhension pour la machine : c'est pourquoi des logiciels tels que des compilateurs ou des interpréteurs permettent de transformer le langage "humain" en langage machine afin que cette dernière puisque exécuter les instructions écrites.

Parallèlement, il existe des langages de bas niveau tels qu'un langage d'assembleur. L'assembleur est plus difficile à comprendre pour l'humain mais reste très proche de la machine : il n'y a ni d'étape d'interprétation ni de compilation. Les instructions d'un langage d'assembleur écrites en langage "humain" sont directement assemblées en binaire.



Alors, à quoi peut bien servir un langage d'assembleur ?

Les machines utilisant l'architecture de Von Neumann présentent des entrées, des sorties, une mémoire RAM et un processeur contenant l'UAL, des registres et une unité de commande :



Puisque le langage d'assembleur est proche de la machine, il est possible d'avoir un contrôle total sur la mémoire RAM, les registres du processeur et bien plus encore... et il permet de TOUT faire : gestion des périphériques d'entrée/sortie, réécriture d'un système d'exploitation, debug de softwares, etc... Et ce n'est pas le cas de la plupart des langages de haut niveau.

Cependant, il faut prendre en compte que l'apprentissage d'un tel langage doit se faire avec un public averti... On peut tout faire avec, mais on peut aussi mettre sa machine en danger ! Comme on dit, un grand pouvoir implique de grandes responsabilités !

Notez tout de même que la réécriture de certains logiciels en *flat assembly* (uniquement en assembleur) est totalement faisable mais très fastidieuse. C'est pourquoi il est souvent utilisé partiellement. Il faut également savoir que la présence de plusieurs langages d'assembleurs s'explique par la diversité de processeurs (Intel, AMD, ...) et d'architectures (ARM, MIPS, x86, ...). Durant tout au long de ce cours, nous nous intéresserons uniquement aux processeurs Intel d'architecture x86 sur sytème d'exploitation GNU/Linux (la programmation sur Windows est sensiblement la même mais quelques subtilités existent).

Dès à présent, concentrons-nous sur les logiciels requis pour pouvoir coder et assembler nos fichiers.

II. Logiciels d'assemblage et de liage des fichiers

Par défaut, le code assembleur a pour extension **.s** ou **.asm**. Vous pouvez utiliser un simple éditeur de fichier. Pour ma part, j'utiliserai Sublime Text.

Nous allons utiliser NASM (Netwide assembler), qui est un assembleur pour processeurs 8086 utilisant la syntaxe Intel. En fait, il existe la syntaxe AT&T et Intel. L'AT&T reste plus difficile à lire et à écrire mais propose une approche très complète sur la lecture et l'écriture en mémoire. Nous chercherons la simplicité en utilisant NASM.

Pour installer NASM, tapons cette commande :

```
sudo apt install nasm
```

Imaginons que nous voulions assembler le fichier test.s en utiliant nasm codé sur **32 bits** (j'en reparlerai plus tard) sur notre processeur **Intel 8086** sur système d'exploitation **GNU/Linux**. Voici la syntaxe de base à utiliser :

```
nasm –f elf32 test.s -o test.o
```

-f (format) : détermine le format du fichier objet de sortie à générer (le format **elf** *(executable and linkable format)* est un format de fichier binaire utilisé sur système Unix)

-o (output) : détermine le nom du fichier objet de sortie à générer (cet argument est optionnel car nasm se charge de donner le meme nom de fichier de sortie .o)

Lorsque l'on assemble un fichier **.s**, il en résulte un fichier objet **.o**. Ça ne vous rappelle rien ? Si si, le compilateur GCC permet aussi de façonner des fichiers objets .o à partir de code en langage C ou C++. GCC va ensuite lier tous les fichiers objets générés pour former un fichier exécutable : c'est exactement ce que l'on va faire manuellement, grâce à la commande **ld**.

```
ld –m elf_i386 test.o -o programme
```

-m (emulation) : Détermine l'émulation du lieur (ici, c'est du format elf et de l'Intel 80386, qui représente une évolution du 8086)

-o (output) : Détermine le nom du fichier de sortie, c'est à dire le fichier binaire exécutable (cet argument est optionnel car nasm se chargera de créer le fichier **a.out** si aucun nom n'est spécifié)

Voilà comment générer un fichier exécutable à partir d'un code.

Mais une question pourrait vous brûler les lèvres : Serait-il possible de lier des fichiers objets .o générés par GCC et liés avec des fichiers objets .o générés par nasm ? Bien sur que l'on peut ! Pourquoi se casser la tête à réinventer la roue en recodant la fonction *printf* en assembleur quand elle est disponible en langage C ?

Nous verrons comment faire un peu plus tard.

Commençons à coder.