
CompareObjects.css
HACKING &
SÉCURITÉ
VOLUME 1 : L'ASSEMBLEUR
X86 SOUS WINDOWS AVEC
NASM
KHALED AHOUNOU

Hacking & sécurité
Volume 1 : l'assembleur x86 sous Windows avec Nasm

AVANT-PROPOS

Le livre que lisez actuellement à une longue histoire derrière lui. En effet, ce livre a mis cinq ans avant de sortir. On pourrait se demander comment on peut passer cinq ans à écrire un seul livre. Je vais donc vous décrire un peu le parcours de la création de ce livre.

J'ai toujours été (et je suis encore) un passionné de nouvelles technologies. Déjà petit, je passais des heures entières, les yeux rivés devant l'écran de mon téléviseur à voir des films de science-fiction. Je connaissais également, tous les super-héros de l'univers Marvel et je rêvais de pouvoir aussi « sauver le monde » un jour.

Beaucoup plus tard, j'ai eu accès à un ordinateur. Mon envie dévorante pour la science m'a fait très vite comprendre les principes de bases de l'informatique. Je rivalisais déjà en 6 mois avec les inscrits en 3ème année de licence informatique. La plupart de mes nouveaux amis croyaient, à tort, que j'avais un ordinateur depuis ma naissance.

Ensuite, comme tout nouveau programmeur qui se respecte, je voulais créer mon propre système d'exploitation. Mais, je me suis heurté à plusieurs problèmes. J'avais beau écumé tout Internet, il n'y avait pas moyen de trouver un livre décent pour m'apprendre comment faire. En me baladant de forums en forums, j'ai fini par comprendre que j'étais venu trop tôt. Mais, à cause de mon acharnement à comprendre les choses, je me suis lancé le défi de comprendre « l'assembleur » que tout le monde semblait citer comme point de départ.

Mais, à nouveau, j'avais beau essayé de comprendre les livres sur l'assembleur, ils ressemblaient tous à des hiéroglyphes. On aurait dit que ces livres assumaient qu'on savait déjà programmer en assembleur avant de les acheter. J'ai donc décidé de m'apprendre tout seul ce langage. C'était plus facile à dire qu'à faire. J'ai même été découragé tant il semblait difficile de comprendre sans aucune aide un langage de si bas niveau. Ensuite, pour des raisons personnelles, j'ai abandonné le projet durant 3 ans et demi.

J'ai fini par reprendre le projet et j'ai compris que l'assembleur était un programme et que le langage d'assemblage (que je nomme affectueusement ASM) était ce que je cherchais à apprendre. Ensuite, j'ai découvert qu'au-delà de la création de systèmes d'exploitation (où il était d'ailleurs très peu utilisé par la plupart des programmeurs), l'ASM permettait bien plus. On pouvait carrément devenir de vrais hackers (ou des protecteurs de systèmes - ce que j'ai choisi).

Je me suis vu investi de superpouvoirs comme dans les animations. Mais, Avec de grands pouvoirs viennent de grandes responsabilités. J'ai donc décidé de bien comprendre et ensuite de rendre de façon pédagogique cette connaissance à la communauté informatique qui m'a tant donné.

L'idée d'écrire un livre qui n'assumait plus de prérequis était né. Ce livre est celui que vous lisez maintenant. C'est un livre écrit par un passionné pour les passionnés. Il vous investira de grands pouvoirs sur la machine. Mais, également, il interpellera votre responsabilité vis-à-vis de la société. N'oubliez pas : avec de grands pouvoirs, viennent de grandes responsabilités.

REMERCIEMENTS

Aucun livre n'est le fruit d'une seule et même personne. En fait, ce livre n'aurait jamais vu le jour sans les nombreuses personnes qui m'ont apporté leur soutien ou leur aide. La première personne que je tiens à remercier est Grâce, qui partage ma vie depuis plusieurs années et qui ne tarit pas de mots d'encouragements à mon égard. Ensuite, il y la communauté des bêta-testeurs du site OpenClassrooms (anciennement siteduzéro) d'où ce projet est né. Il y a Guy GRAVE alias Mewtow qui a été l'un des premiers lecteurs de mes brouillons. Ses conseils avisés et son envie d'avoir une progression claire et pédagogique ont scellé le point de départ pour un livre de qualité (enfin j'espère). Il y a aussi en particulier Ariel KAMOYEDJI alias Cybernétique, Yacoub Aden Miguil alias YAM, Léo CHAUSSE et Vincent BALLY qui ont été, au-delà de simples lecteurs, de puissants moteurs qui m'ont permis de finir ce projet que je comptais abandonner.

INTRODUCTION

Vous rêvez de devenir un expert en cybersécurité[1], de craquer des logiciels ou comprendre comment les protéger contre les crackers. Vous désirez comprendre le fonctionnement des ordinateurs, écrire des programmes qui font des choses énormes mais ne pèsent que quelques Kilooctets. Vous voulez profiter des fonctionnalités les plus récentes de votre microprocesseur afin de rendre plus rapide l'exécution de vos programmes. Vous voulez être capable de créer et d'analyser le fonctionnement des virus, des rootkits, et des vers informatiques. Vous souhaitez créer vos propres **exploits informatiques** et **shellcodes** ou simplement comprendre leur fonctionnement. Vous mourrez d'envie de créer votre propre système d'exploitation ou en comprendre le fonctionnement. Nul doute. **Vous avez besoin du langage d'assemblage** (appelé à tort assembleur).

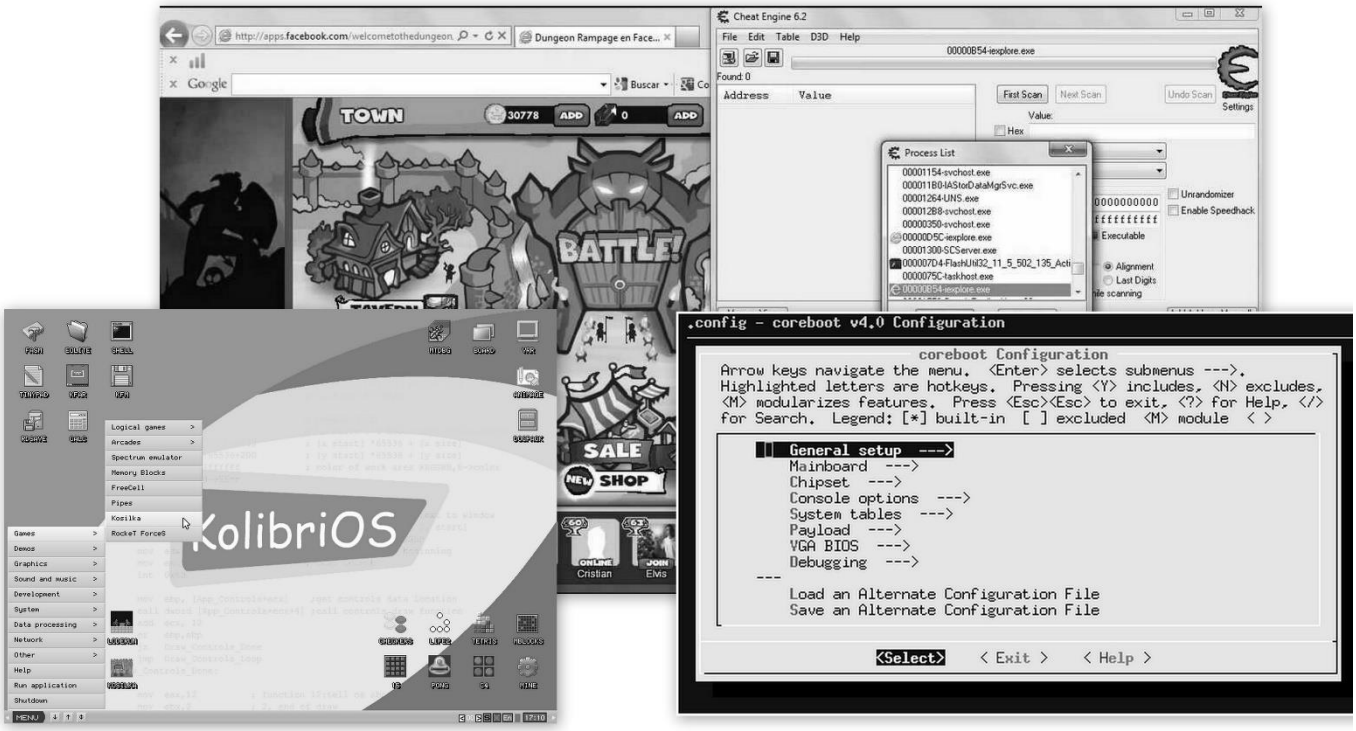
Ce que je vous propose ici, c'est de vous apprendre progressivement à utiliser ce langage de programmation[2] en partant d'un **niveau débutant**. Ce livre est écrit en plusieurs volumes. Ceci est le premier volume. Voici ce que **je vous garantis** que vous **allez comprendre** en lisant ce premier volume :

- le fonctionnement des ordinateurs ;
- programmer en langage d'assemblage ;
- consolider vos connaissances à travers des démos, plusieurs Travaux Dirigés, Travaux Pratiques, exercices et QCMs ;
- comprendre les bases du cracking et craquer votre premier programme ;
- créer des shellcodes et apprendre les principales vulnérabilités applicatives ;
- Exploiter les vulnérabilités applicatives sur les systèmes **32 et 64 bits modernes** en contournant les protections usuelles ;
- et plein d'autres choses !

Tout ceci, avec les seules connaissances que vous aurez acquises durant la lecture de ce premier volume. D'ailleurs, pour l'exemple, le système d'exploitation le plus rapide et léger du monde, **KolibriOs**, prends seulement 10 secondes du démarrage à l'affichage du bureau. Et, il est capable de fonctionner rien qu'avec 8 MB de RAM ! Car, il est presque entièrement fait en « assembleur ». Il y aussi d'autres outils faits en utilisant les connaissances en « assembleur » comme **Cheat Engine** qui permet de faire de la modification de valeurs en mémoire afin de tricher à des jeux vidéo (par exemple, bloquer le nombre de vies du personnage au maximum pour ne plus perdre de vies) ou simplement espionner un programme en exécution tout en changeant des valeurs. Il y a également **coreboot** qui vise à remplacer le BIOS des ordinateurs en le rendant plus sécurisé et rapide. Motivé ? Parfait. C'est parti !

1 Il est important de noter que la sécurité informatique est un domaine extrêmement vaste. Ici, je parle spécifiquement de sécurité applicative. La sécurité des réseaux ou la sécurité des applications web sont, par exemple, exclues.

2 Certains considèrent le langage d'assemblage comme un ensemble de mnémoniques qui n'a pas assez de couches d'abstraction pour être considéré comme un langage de programmation à part entière. Nous y reviendrons plus tard. Pour plus de simplicité, nous, nous le considèrerons comme un langage de programmation.



KolibriOs - Cheat Engine[3] - coreboot

3 Source de l'image https://www.youtube.com/watch?v=hsS5eqDZhIU

Première partie

Les bases du langage d'assemblage

CHAPITRE 1 : GÉNÉRALITÉS

Vous avez décidé de vous lancer dans la programmation. Mais, vous ne savez pas par quoi débuter. Le but de ce chapitre est, justement, de vous introduire au monde de la programmation. Nous allons aborder les questions courantes, mais importantes, que se posent les débutants en programmation. Nous allons donc, poser les bonnes bases pour comprendre la programmation en langage d'assemblage. Sans plus tarder, je vous ouvre les portes du monde de la programmation. Bonne lecture !!

Programmer, c'est quoi ?

Nous allons commencer par définir le terme « programmer ». Non, non, je ne me moque pas de vous. On va vraiment définir ce mot. N'oubliez pas qu'ici, on commence tout à partir d'un niveau débutant. Si vous avez l'impression de perdre du temps en lisant ce qui suit, détrompez-vous. Il est important, pour la suite, de comprendre tout cela. De plus, ça servira à mettre tout le monde au même niveau. Bon, on ne va pas tout de même y passer la nuit. Dans le domaine informatique, programmer, c'est le fait de créer des programmes informatiques. Mais, un programme c'est quoi ? Pour répondre à cette question, je vous propose de répondre à la question comment fonctionne un programme.

Un programme c'est quoi ?

Pour bien comprendre comment fonctionne un programme, nous allons nous inspirer de l'exemple d'un étudiant à la faculté. En effet, ce dernier suit un programme spécifique chaque jour. Il dira par exemple : « le matin entre 8h et 10h, j'ai un cours sur l'électronique. Juste après, c'est plutôt l'informatique et puis le soir… ». Bref, on se rend facilement compte **qu'il exécute une suite d'opérations claires dans un ordre spécifique**. On dira alors qu'il a programmé sa journée. De cet exemple banal, on peut déjà déduire trois choses :

- qui parle de programme, parle d'une série d'opérations ou de traitements ;
- les traitements sont définis sur une donnée (dans le cas de l'étudiant, la donnée est le temps) ;
- les opérations en question suivent un ordre bien définit (l'étudiant n'ira pas par exemple au cours de 10h avant celui de 8h).

Voilà ! Si vous avez compris cela, vous avez tout compris. En effet, un programme informatique ne s'écarte pas trop de cette logique. Créer un **programme informatique, c'est le fait définir un ensemble d'opérations ou de traitements sur des données.** La différence avec l'exemple de l'étudiant, est qu'un ordinateur ne se donne pas d'instructions. C'est plutôt un humain, en l'occurrence un programmeur, qui définira les instructions et l'ordre dans lequel elles doivent être exécutées grâce à un **langage de programmation**.

Ces instructions demanderont à l'ordinateur d'effectuer certaines opérations. Comme, mettre en gras une partie de texte, jouer de la musique ou encore lire une vidéo. Voici des exemples de programmes :

- Un éditeur de texte (Notepad++ par exemple)
- votre navigateur web (Firefox, Google chrome, Internet Explorer…)
- Un jeu vidéo (Fantastic 4 par exemple)
- Un lecteur multimédia (VLC par exemple)
- …

Afin de bien comprendre le concept de programme, le tableau suivant montre quelques exemples de programmes, et identifie par la même occasion, les types de traitements et les données traitées qui leur sont associés.

Exemples de programme	Exemple de traitement	Donnée traitée
VLC	Lire	Vidéo
Microsoft Word	Mettre en gras	Texte
Lecteur musique	Jouer	musique

J'espère qu'avec ce tableau, le concept de programme informatique (la définition d'un traitement sur des données) est assez clair maintenant. Il faut avouer que les exemples de traitements que j'ai mis dans le tableau précédent sont assez « généralistes ». Dans un ordinateur, les traitements préprogrammés sont plus basiques. Par exemple le traitement « jouer » n'existe pas réellement. C'est plutôt un ensemble de traitements basiques qui est effectué par l'ordinateur. Des traitements comme la localisation du son à jouer, son décodage grâce à un décodeur, et la vibration des hauts parleurs pour sa lecture. Ce sont les différentes combinaisons de traitements basiques qui permettent la création de programmes plus complexes.
Par exemple, un jeu vidéo joue du son, reçoit des frappes du clavier et affiche à très grande vitesse des images sur le moniteur. C'est un programme complexe crée à partir de traitements basiques sur des données comme du son, et des images. La figure suivante montre les images de quelques programmes exécutables.

Quelques programmes : Google chrome - VLC - Notepad++ - Fantastic 4

Google chrome traite du texte, des images et de la vidéo et les affiche dans une fenêtre, VLC traite des fichiers vidéo et audio, Notepad++ permet de traiter du texte brute et enfin Fantastic 4 est un jeu vidéo. Maintenant que vous savez ce que sont les programmes, passons à ce qui permet de les « créer » : les langages de programmation.

Qu'est-ce qu'un langage de programmation ?

Un ordinateur n'est rien d'autre qu'un agencement intelligent de circuits électroniques. Or, les systèmes électriques, sont caractérisés par deux états :

- état ouvert ;
- état fermé.

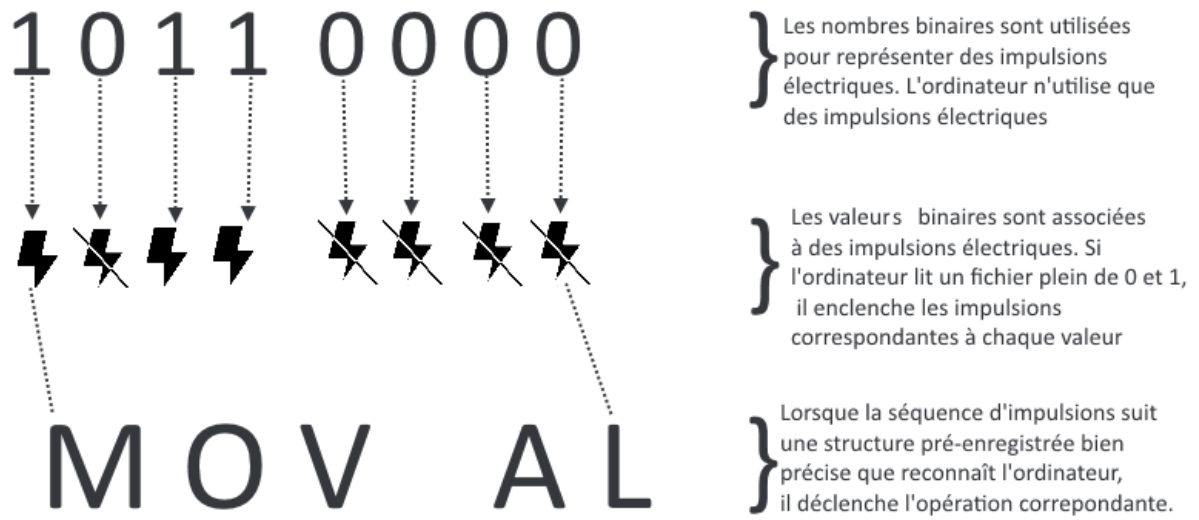
De là, est née l'idée d'utiliser deux symboles distincts pour représenter les différents états d'un circuit électrique. Ces symboles sont 0 et 1. Le « 0 » signifie l'absence de courant dans le circuit et le « 1 » la présence de courant. Ces deux symboles sont appelés des **nombres binaires**. C'est-à-dire des nombres qui ne peuvent prendre que deux valeurs possibles (0 ou 1). Nos interrupteurs sont d'ailleurs, la preuve que ces nombres binaires sont bien représentatifs de l'état dans lequel se trouve un circuit électrique. L'image suivante montre le lien entre les nombres binaires et les différents états d'un circuit électrique. Lorsque l'interrupteur est sur le 1, le courant passe. S'il est sur le 0, le courant ne passe pas.

Un interrupteur (source Wikipédia)

Vous commencez sûrement par vous demander le lien entre électricité et langages de programmation. Pour comprendre cela, imaginez que je vous demande d'allumer et d'éteindre trois fois la lampe de votre chambre. En supposant qu'elle est éteinte, je pense que sans trop de difficultés, vous finirez par avoir comme résultat la séquence suivante :

```
allumer-éteindre-allumer-éteindre-allumer-éteindre.
```

Exprimons cela sous forme d'une suite de nombres binaires. Cela revient à `1-0-1-0-1-0`. Donc chaque fois que je vous dirai `1-0-1-0-1-0`, vous exécuterez l'instruction `allume_3_fois`. Vous venez d'avoir un aperçu de ce qu'on nomme le langage machine ou code binaire. C'est-à-dire les séquences binaires spécifiques reconnues par un ordinateur pour déclencher des traitements précis. Les concepteurs de microprocesseurs (la puce électronique responsable du traitement des données dans votre ordinateur) utilisent exactement le même principe pour préenregistrer des traitements spécifiques que votre ordinateur pourra invoquer en utilisant la bonne séquence d'impulsions électriques. Il y a donc quelque part dans votre ordinateur, des séquences d'impulsions (codées sous forme binaire) qui, une fois invoquées, permettent d'effectuer un traitement précis. Ainsi, pour qu'un ordinateur traite directement une instruction donnée par un humain, cette dernière devrait donc ressembler à quelque chose comme… Ça : `1011 0000 0110 0000` (d'ailleurs, cette séquence binaire signifie exactement, copie 96 dans le registre AL sur les microprocesseurs x86). Le schéma suivant montre la correspondance entre l'électricité, le langage machine, et le langage d'assemblage.



L'ordinateur ne comprend que les séries d'impulsions électriques préenregistrées

Le problème avec ces séquences de code machine, c'est qu'elles sont difficiles, voire impossible, à retenir pour un humain. Les programmeurs ont donc eu la géniale idée, d'inventer de nouveaux langages compréhensibles et facilement mémorisables par les humains. Ils ont aussi décidé, par la même occasion, de créer des « traducteurs » qui se chargeraient de traduire ces nouveaux langages en langage machine. Ainsi sont nés, les langages de programmation.

Le programmeur n'a donc plus qu'à connaitre le langage et ne plus se soucier des séquences binaires. Un langage de programmation est donc un **ensemble de** mots réservés **obéissants à une certaine** syntaxe**, permettant de créer des programmes informatiques**. C'est-à-dire, c'est un langage qui permet de définir des traitements sur des données. Que les expressions comme « syntaxe » et « mots

réservés » ne vous fassent pas peur. Quand je parle de mots réservés je veux juste dire que seuls certains mots sont utilisables par le programmeur. Ceux définis par le créateur du langage et pas d'autres. Il faudra donc les connaître. Quant au mot syntaxe, il fait allusion au fait que l'ordre d'agencement de ces mots réservés est très important sinon le « traducteur » signale une erreur. Pour vous donner une idée, prenons l'exemple de la langue française. Lorsque vous utilisez mal un article en mettant un « la » à la place d'un « le », bonjour la catastrophe. Eh bien, c'est pareil en programmation. Après la traduction en langage machine, le résultat obtenu est un programme exécutable. C'est-à-dire un programme que l'ordinateur peut directement exécuter. Un programme exécutable est donc le résultat obtenu **après** la traduction en langage machine d'une suite d'instructions écrites par un programmeur dans le langage de programmation de son choix.

Le langage d'assemblage c'est quoi ?

Le langage d'assemblage (ou ASM[4]) est le langage de programmation le plus proche de la machine. En fait, il en est tellement proche, que chaque instruction ASM **correspond mot-à-mot** avec une séquence binaire. **L'ASM n'est donc qu'une forme humainement lisible du code machine**. Par exemple, la séquence de code machine suivante: `1011 0000 0110 0000` correspond exactement à la ligne de code `mov AL, 96`. C'est d'ailleurs pour cela que l'opération inverse à l'assemblage, **le désassemblage**, est possible. En effet, Tout programme exécutable étant représenté au niveau binaire dans l'ordinateur après assemblage, on peut retrouver sa forme[5] ASM en partant du programme exécutable. Car, chaque code machine correspond à une instruction ASM.

Le langage d'assemblage, un monstre à plusieurs têtes

Avant toute chose, nous allons brièvement reparler de ce que l'on appelle un microprocesseur.

Le processeur (CPU, pour Central Processing Unit, soit Unité Centrale de Traitement) est le cerveau de l'ordinateur. Il permet de manipuler des informations numériques, c'est-à-dire des informations codées sous forme binaire, et d'exécuter les instructions stockées en mémoire.

- www.commentcamarche.net

Pour beaucoup, je parie que cette définition n'est pas tout à fait claire. Je vais donc mieux l'expliquer. Un microprocesseur ou simplement processeur est une

4 ASM vient d'ASseMbleur. C'est plus court.

5 Le désassemblage d'un exécutable n'est pas toujours possible. En effet, l'usage des techniques d'offuscation, peut rendre un programme tellement complexe que le désassemblage peut s'en retrouver très difficile, voire impossible.

puce électronique présente dans un ordinateur et qui traite les différentes informations que nous envoyons à ce dernier et exécute les instructions pour notre plus grand bonheur. Voilà à quoi peut ressembler un microprocesseur :

Un microprocesseur Intel 80386DX

J'imagine que vous ne voyez pas encore le lien entre microprocesseur et ASM. Patience… J'y viens. En fait, comme nous l'avons vu dans la définition précédente, un microprocesseur constitue le cerveau d'un ordinateur. Donc, comme tout cerveau qui se respecte, il dirige toutes les actions au sein de l'ordinateur. Comprenez par « actions », les instructions exécutées par ce dernier. S'il contrôle les instructions exécutées, il coule alors de source qu'il comprend le langage qui définit ces instructions.

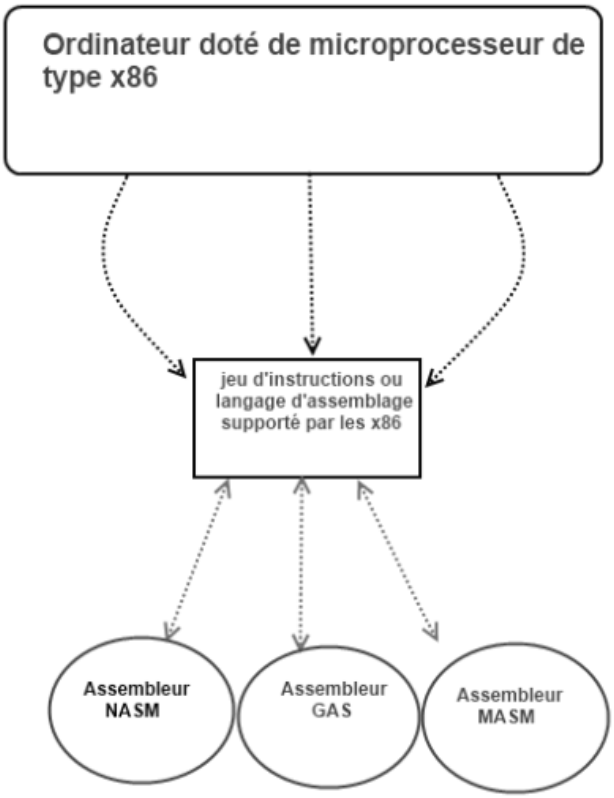
Vous voyez enfin où je veux en venir ? Vraiment pas ?? Eh bien, des affirmations précédentes, il apparaît clairement que **c'est le microprocesseur qui définit le langage machine à utiliser sur un ordinateur.** Car, c'est lui qui reconnaît et exécute les différentes séquences binaires préenregistrées. On ne pourra donc s'adresser directement à un ordinateur que dans le langage machine que comprend son microprocesseur. **Pas autrement !**[6] Le problème, c'est qu'il existe une multitude de famille de microprocesseurs. La conséquence directe, est qu'il existe autant de langages machine que de familles de microprocesseurs différents. Et, comme l'ASM a une correspondance directe avec le langage machine, il varie donc autant que varie le langage machine. Et vous n'êtes même pas encore au bout de vos peines. Figurez-vous que, non content d'être aussi diversifié, le langage d'assemblage ne spécifie aucune règle quant à sa syntaxe. Ce qui fait que plusieurs assembleurs existent par microprocesseur. Chacun respectant une syntaxe arbitraire !

On peut donc, avoir plusieurs assembleurs différents pour un même microprocesseur µp I et chaque assembleur respectera une syntaxe différente.

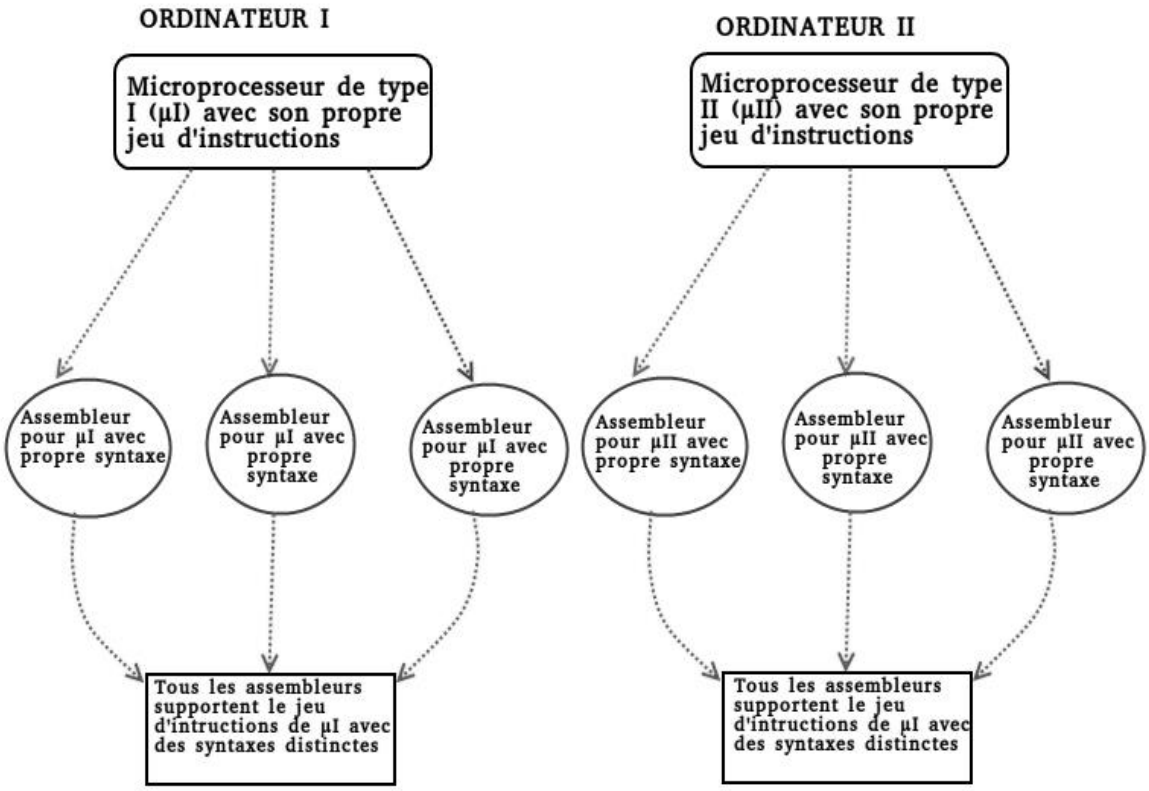
6 Il est important de savoir qu'il est néanmoins possible d'exécuter, à l'aide d'un émulateur (un programme qui simule les capacités d'un microprocesseur), les instructions propres à un microprocesseur sur un autre microprocesseur.

Mais, au final, ils respectent tous le même jeu d'instructions ou langage machine sous-jacent du microprocesseur cible (l'ensemble des différentes séquences binaires préenregistrées par les concepteurs du microprocesseur). Par exemple, pour le jeu d'instructions des microprocesseurs de la famille x86, il existe des assembleurs comme Nasm, Masm, fasm, Gas... Ces derniers diffèrent par leurs syntaxes et/ou le fait qu'ils ne génèrent pas leur code machine de la même façon. Néanmoins, ils utilisent tous, le même jeu d'instructions.

Pour résumer, on dira donc, que le microprocesseur d'un ordinateur définit le jeu d'instructions et d'assembleurs à employer pour lui donner des instructions. Mais, comme il existe plusieurs types de microprocesseurs alors, il existe également plusieurs langages d'assemblage. **En plus, il existe plusieurs assembleurs par microprocesseur**. Les figures suivantes récapitulent tout cela.



Plusieurs assembleurs qui supportent le même jeu d'instructions pour un microprocesseur x86



Différents microprocesseurs, différents jeux d'instructions – plusieurs assembleurs par microprocesseur avec différentes syntaxes mais supportent même jeu d'instructions

Ce qu'il faut retenir des schémas précédents, c'est que µp I utilise un langage d'assemblage (le jeu d'instructions) différent de µp II car µI est d'une famille différente de µII. Donc, en règle générale, lorsqu'on a deux ordinateurs, dotés chacun d'un microprocesseur, et que l'ordinateur I possède un microprocesseur de différente famille de celui de l'ordinateur II, alors ces deux ordinateurs ne pourront pas utiliser le même langage d'assemblage[7]. Il faut alors, choisir un assembleur après avoir choisi le type de microprocesseur sur lequel on veut programmer. La famille de microprocesseurs dont nous allons exploiter le jeu d'instructions tout au long de ce livre est celle des x86[8] car c'est celui qui équipe la quasi-totalité des ordinateurs en circulation. Quant aux assembleurs qu'on peut utiliser avec cette famille, voici une liste non exhaustive des plus courants :

- fasm (flat assembler) ;
- Masm (Microsoft Macro assembler) ;

7 À moins de s'armer d'un émulateur.

8 Si vous ne comprenez pas le terme x86, ne vous inquiétez pas. Un chapitre entier sera consacré à l'étude des microprocesseurs. Il faut aussi dire que seuls les sous-familles x86-32 et x86-64 seront étudiés dans ce volume. La sous-famille x86-16 est exclue.

- Gas (GNU assembler)[9] ;
- Yasm ;
- Nasm (Netwide assembler) ;
- ...Et bien d'autres !

Dans la documentation de Nasm, vous verrez dès les premières pages, une liste d'assembleurs et en quoi Nasm procure plus d'avantages.

En haut, ou en bas...

Les langages de programmation peuvent être **grossièrement** classés en deux catégories :

- langages de haut niveau ;
- langages de bas niveau.

Un langage de haut niveau est un langage de programmation qui propose un ensemble d'instructions généralistes et se charge de les traduire en instructions compréhensibles par l'ordinateur. Vous n'aurez pas, par exemple, à gérer directement la mémoire ou à savoir comment les données y seront enregistrées. Le langage de haut niveau s'en charge en proposant des moyens plus ou moins faciles de le faire. La syntaxe aussi est simplifiée : elle est plus proche du langage humain. C'est fait exprès pour faciliter la lisibilité du code. La conséquence directe est que le débogage (détection et correction des erreurs dans un code-source) devient beaucoup plus facile.

Un langage de haut niveau possède les caractéristiques suivantes :

- Il ne dépend pas du matériel sous-jacent ;
- Il est plus orienté vers la création de programmes utilisateurs ;
- Il est d'utilité générale ;
- Il expose une puissante couche d'abstraction ;
- Chacune de ses instructions correspond à plusieurs instructions machines.

Je vais expliquer chacun de ces points en détails.

On parle **d'indépendance par rapport au matériel sous-jacent**, lorsqu'aucun effort supplémentaire n'est nécessaire au programmeur afin de créer un programme qui fonctionnera sur plusieurs ordinateurs dont les composants internes sont différents. Ainsi, quel que soit le microprocesseur, le type de mémoire, ou encore les périphériques (écran, clavier, imprimante...) dont est doté l'ordinateur sur lequel on programme, le langage de haut niveau permet de programmer sans être au courant de ces détails. Il donne également la possibilité de porter ces mêmes programmes vers des ordinateurs dotés de différents

9 Gas est un assembleur particulier. En effet, il respecte la syntaxe **AT&T**. C'est une syntaxe où la différence majeure est l'inversion de l'ordre des opérandes d'une instruction. La deuxième syntaxe qui existe est la **syntaxe Intel**. On la qualifie ainsi, car elle respecte la façon dont les manuels Intel présentent du code ASM. C'est la syntaxe Intel que ce livre utilise.

composants internes sans, ou avec très peu, de modification du code-source de départ.

On dit qu'un langage est **orienté vers la création de programmes utilisateurs** lorsqu'il met à la disposition du programmeur des mécanismes pour créer des programmes utilisateurs (par opposition aux programmes systèmes). Ainsi, on retrouvera par exemple des morceaux de codes tout prêts que le programmeur peut invoquer pour effectuer des tâches que les utilisateurs font de façon récurrente. Par exemple, l'affichage de données à l'écran. La saisie de donnée à partir du clavier…Il y a aussi plus de facilité pour la déclaration de données complexes (tableaux, structures…). Si vous ne comprenez pas cela, ne vous inquiétez pas car ce n'est pas important pour le moment.

On dit qu'un langage est d'**utilité générale** lorsqu'il n'est pas cloisonné à une tâche spécifique. Autrement dit, on peut créer une large palette de programmes différents avec (par exemple, des éditeurs de texte, jeux vidéo, traitements de texte, programmes systèmes…).

On dit qu'un langage expose une **couche d'abstraction** lorsqu'il cache les détails liés à son fonctionnement interne au programmeur qui s'en sert. C'est-à-dire que le langage met à la disposition du programmeur des fonctionnalités assez faciles d'utilisation qui permettent d'accomplir des tâches assez complexes sans que ce dernier sache comment. Par exemple, le programmeur peut faire jouer du son à la machine sans comprendre comment fonctionne une carte son. Ou encore, il peut créer un navigateur sans comprendre comment fonctionne une carte réseau.

Pour terminer, la dernière caractéristique d'un langage de haut niveau est que **chacune de ses expressions correspond à une série d'instructions machine**. Par exemple, dans un langage de haut niveau, afficher une phrase à l'écran peut se faire avec une seule ligne de code. Par contre lorsque c'est traduit en langage machine, cela peut être équivalent à plusieurs lignes de codes.

Tout langage qui est caractérisé par ces cinq points est donc un langage de haut niveau. Nous avons vu ce que c'est qu'un langage de haut niveau. Parlons maintenant des langages de bas niveau. Les langages de bas niveau permettent d'utiliser presque directement les ressources (microprocesseur, mémoires…) de votre ordinateur. Il est un peu plus difficile de programmer avec ces types de langages qu'avec les langages de haut niveau. Mais, la difficulté est compensée par la rapidité d'exécution du code, l'accessibilité à certaines fonctionnalités inaccessibles à partir de langages de haut niveau et un contrôle plus précis du programmeur sur la machine.

Voici les caractéristiques d'un langage de bas niveau :

- Il dépend fortement du matériel sous-jacent ;
- Il est plus orienté vers la création de programmes systèmes ;
- Il est plus utilisé dans des cas spécifiques ;
- Chacune de ses instructions correspond à une seule instruction machine ;
- Il n'expose presque pas de couche d'abstraction.

Encore une fois, nous allons détailler chacun de ces points.

Un **langage de bas niveau est dit dépendant du matériel** car avec un tel langage, les détails liés au fonctionnement interne de la machine doivent être connu par le programmeur afin de créer des programmes. Il y a aussi le fait qu'un tel langage est fortement lié à un seul type de microprocesseur. Il ne peut donc fonctionner avec un autre ordinateur doté d'un microprocesseur différent.

La deuxième caractéristique d'un **langage de bas niveau** est qu'il dispose d'attributs qui **facilitent plus la création de programmes systèmes que la création de programmes utilisateurs** (bien que ceci soit aussi possible). Ainsi les tâches courantes (affichage à l'écran, interaction à travers le clavier…) dont un programme utilisateur a souvent besoin ne sont pas directement greffées au langage. Le programmeur se voit donc souvent réinventer la roue en implémentant toutes ces fonctionnalités ou bien il est obligé à se lier avec des bibliothèques ou APIs tierces (des sortes de codes tout prêts à l'emploi).

Un langage bas niveau est très souvent employé pour des tâches spécifiques comme rendre plus précis le contrôle du programmeur sur la machine, lui donner accès à des composants de l'ordinateur (coprocesseurs, mémoires caches..) inaccessible depuis un langage de haut niveau ou encore lui permettre l'utilisation d'instructions spécifiquement optimisées pour la vitesse d'exécution. Autrement dit, il est beaucoup plus difficile, bien que ce soit possible, d'utiliser dans un cadre général ce genre de langage. Par exemple, il est bien plus difficile, bien que ce soit possible, de programmer un jeu en ASM que de le faire dans un langage de plus haut niveau que lui.

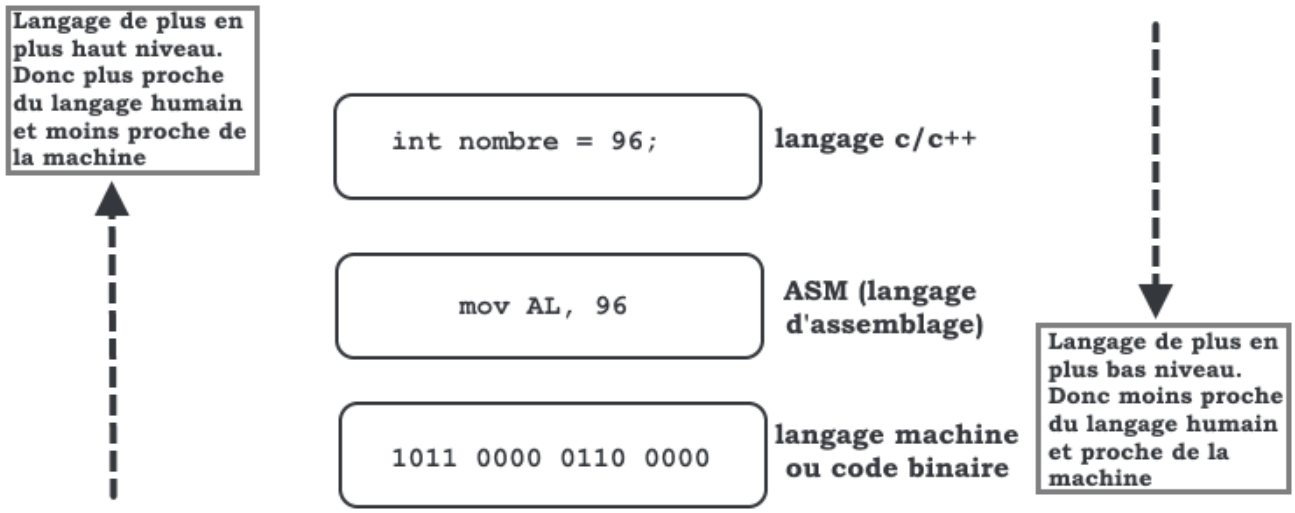
Avec un langage de bas niveau chaque instruction correspond à une séquence binaire exacte. Ni plus, ni moins. Autrement dit, c'est ce que vous écrivez dans votre code qui est exécuté, mot pour mot, par la machine. Car, contrairement aux langages de haut niveau, chacune des instructions employées correspond effectivement à une suite de nombres binaires qui eux-mêmes correspondent à une opération précise enregistrée dans le microprocesseur par ses concepteurs.

La dernière caractéristique des **langages de bas niveau est qu'ils ne disposent presque pas de couche d'abstraction**. On ne retrouve pas de mécanismes de déclaration de données complexes (structures, tableaux…), de structures de contrôles (conditions, boucles, fonctions…), ou encore de facilités qui permettent de communiquer avec les périphériques (écran, clavier, …) sans connaître leurs fonctionnements internes.

Comme langage de haut niveau nous avons, par exemple, le langage JAVA, le langage C++, le langage C… Et comme langage de bas niveau nous avons le code binaire (ce n'est pas un langage mais la représentation qu'utilise l'ordinateur de façon interne), et le langage d'assemblage (qui est le langage le plus bas niveau possible lisible par un humain).

En fait tout cela est relatif. L'ASM est plus haut niveau que le binaire (en terme d'abstraction) mais il est plus bas niveau que le langage C (bien que le langage C soit relativement bas niveau car dans ce langage on se préoccupe encore de certains détails liés aux spécificités des machines sur lesquelles on programme).

Tout dépend donc du point de vue selon lequel on se place. Bon je crois qu'un petit schéma ne serait pas de refus pour mettre tout le monde d'accord.



Langages de programmation - bas niveau - haut niveau

Sur le schéma, on voit que, plus le langage est bas niveau, plus il est proche du matériel. Autrement dit, plus il faudra connaître les détails liés au fonctionnement interne de l'ordinateur qu'on programme. Et, moins d'abstractions sont disponibles. Par contre, plus le langage est haut niveau, plus il dispose de couches d'abstraction et est proche du langage humain.

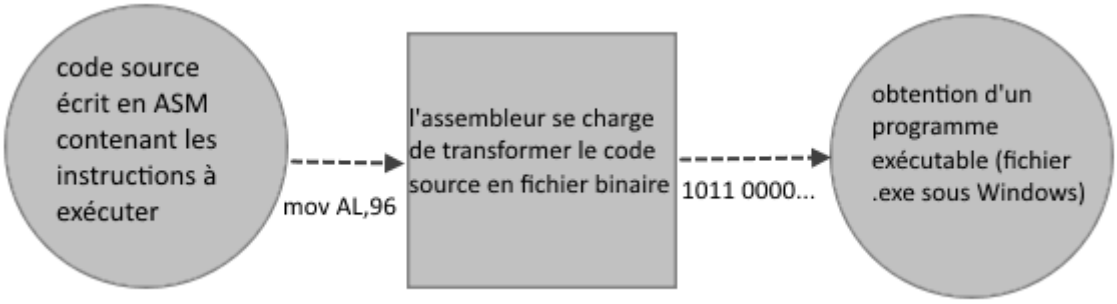
Assemblage et compilation

Lorsque vous utilisez un langage de plus haut niveau que l'ASM, « le traducteur » qui transforme votre code-source en code machine est appelé **compilateur**. Le compilateur effectue une suite de tâches. Tout d'abord, il se charge de traduire chaque ligne de votre code-source en son équivalent en ASM[10]. L'action de « traduire » est alors appelée, dans ce cas, la **compilation**.

Après la compilation, on obtient un code intermédiaire écrit en langage d'assemblage. Le code en langage d'assemblage ainsi obtenu, est alors, à son tour, assemblé (transcrit en langage machine) grâce à un programme spécial appelé **assembleur**. L'action de « transcrire » est alors appelée, dans ce cas, **l'assemblage**. C'est donc un abus de parler de « langage assembleur » car l'assembleur désigne un programme et non un langage. Comme on l'a déjà vu, le terme correct est plutôt « langage d'assemblage ».

10 Ce n'est pas toujours le cas. Par exemple le compilateur C de Microsoft, contrairement à GCC, transforme directement un code C en code machine sans forcément générer du code ASM.

Une image étant plus parlante, voici le récapitulatif sous forme de schéma. Les étapes où le compilateur transforme le code source écrit dans un langage de haut niveau en ASM ont été ignorées exprès. Car le code est directement écrit en ASM (n'oublions pas qu'ici on désire programmer en ASM).



Étapes simplifiées de l'assemblage d'un programme

Comme le suggère le titre de cette figure, ce sont les étapes simplifiées de l'assemblage d'un programme car en réalité, il existe quelques étapes intermédiaires avant la génération du programme exécutable.

Pourquoi apprendre l'ASM ?

Aujourd'hui, il est très peu courant d'écrire entièrement un programme autonome en ASM. On utilise plus le langage d'assemblage en l'associant à un langage de haut niveau. Nous allons donc présenter les avantages de la programmation en ASM.

Quels sont les avantages de la programmation en ASM ?

- L'ASM est utile lorsque la performance, surtout la rapidité, est un facteur critique. Dans ce cas, si les limites algorithmiques[11] sont atteintes, le dernier recours est alors d'utiliser les instructions spécifiques de la machine cible. Ces instructions sont souvent directement encodées dans le microprocesseur. L'ASM est alors LE[12] langage pour les utiliser ;
- L'ASM est aussi utile, lorsque la machine sur laquelle on programme, dispose de peu de mémoire (par exemple les microcontrôleurs). L'emploi de langages de haut niveau, a tendance[13], à pénaliser le programme en termes de performances et de contrôle sur la machine dans ces cas ;

---

11 Un algorithme défini un ensemble fini d'étapes logiques à suivre pour résoudre un problème. Dans notre cas, cela peut être un algorithme qui décrit comment résoudre un problème informatique.

12 Lorsque les fonctions intrinsèques (qui ne sont que l'ASM déguisé) ne sont pas disponibles dans le compilateur utilisé pour le microprocesseur ciblé.

13 Il faut néanmoins préciser, qu'avec les mémoires qui deviennent de moins en moins chères et les microprocesseurs de plus en plus puissants, ces cas, bien qu'existants, sont de plus en plus rares.

- L'ASM est très utile dans le domaine de la sécurité informatique. Les attaquants ou les chercheurs de vulnérabilités l'utilisent pour la découverte et l'exploitation des vulnérabilités applicatives, la création de shellcodes, d'exploits, de virus, de rootkits, et de patchs. On peut alors mieux contrer ces techniques une fois qu'on en comprend le fonctionnement ;
- L'ASM permet le reverse-engineering ou rétro-ingénierie. Le reverse-engineering, en informatique, est l'art de partir d'un programme déjà compilé/assemblé et d'en extraire le code[14] afin d'en comprendre le fonctionnement ou de le modifié à loisir pour ajouter ou retrancher des fonctionnalités. Cette technique est, par exemple, utilisé en virologie pour produire des protections contre les virus informatiques ou pour faciliter l'interopérabilité entre programmes non documentés ;
- Connaitre l'ASM permet de craquer (faire sauter les protections) des programmes, ou des formats propriétaires[15]. On peut par exemple copier un algorithme ultra secret d'une entreprise par rétro-ingénierie afin d'en produire un meilleur ou encore rendre gratuit un programme payant. Comprendre cela permet de développer des techniques d'offuscation (l'art de rendre son code impénétrable) plus solides ;
- L'ASM permet également de créer des drivers ou pilotes informatiques et d'écrire des programmes systèmes en accédant à des ressources matérielles (périphériques) à travers des instructions qui sont difficiles voire impossible à utiliser dans d'autres langages.
- Lors de la création de système d'exploitation, au niveau du boot loader[16] (si on en recrée un), il est impossible d'échapper à l'utilisation de L'ASM. Au niveau de la gestion de tâches systèmes (instructions `LGTD` et `LLDT`), l'ASM est également indispensable.
- L'ASM permet aussi de mieux comprendre et corriger les bugs difficiles à traiter qui apparaissent lors de l'exécution d'un programme compilé ou assemblé[17].
- Programmer en ASM permet une compréhension plus profonde du fonctionnement d'un microprocesseur. Ainsi, même si vous n'utilisez jamais l'ASM, cette compréhension de la machine vous permettra de créer des programmes plus performants dans d'autres langages[18].

---

14 Le code est en ASM. Mais, les commentaires disparaissent et les labels sont tous remplacés par des adresses mémoires en dur (ou dans certains cas, comme avec IDA PRO, des labels auto-générés) ce qui rend un peu complexe la compréhension du code. Mais, avec de l'expérience, c'est relativement aisé de s'en sortir.

15 Il est illégal, de craquer des programmes ou des formats propriétaires sans l'autorisation des créateurs de ces programmes ou formats.

16 Il faut dire que ce cas est valide lorsque l'ordinateur est doté d'un BIOS. Dans le cas où c'est un UEFI qui est présent, L'ASM n'est pas indispensable. On va d'ailleurs créer notre propre boot loader dans un autre volume de cette série de livres.

17 Par exemple, les bugs de corruption mémoire (pile ou tas). La compréhension des dumps mémoires…etc. sont bien plus faciles à gérer quand on comprend l'ASM.

18 Surtout avec les langages qui sont relativement bas niveau comme le C. En effet, la compréhension des conventions d'appel, de piles, d'alignement mémoires permettent par exemple, au programmeur averti de mieux choisir l'agencement des données dans ses structures, l'utilisation de fonctions ou d'écriture directe de blocs de codes ou de macros, une meilleure sécurité du code…etc.

Comme vous pouvez le constater, connaître l'ASM peut être utile même si vous ne programmez jamais en ASM. Mais, l'ASM n'a pas que de bons côtés. Il possède aussi quelques inconvénients. Explorons le revers de la médaille.

Quels sont les inconvénients de la programmation en ASM ?

- Un code écrit en ASM n'est pas portable[19] ;
- Programmer en ASM est long et fastidieux. Un simple code d'affichage de texte par exemple peut prendre plusieurs lignes. Ne parlons pas d'un programme plus complexe ;
- L'ASM est très vulnérable aux erreurs de codages. En effet, dans certains langages de haut niveau, ce qu'on écrit en mémoire, est souvent vérifié par le compilateur. En ASM, tout cela est laissé à la charge du programmeur. Ce dernier se doit donc d'être rigoureux[20] et chevronné pour utiliser l'ASM ;
- Les compilateurs modernes sont devenus plus performants et optimisent donc mieux le code que le programmeur ASM moyen. L'ASM est donc inutile dans les cas courants d'optimisation et il faut avoir un niveau conséquent pour pouvoir l'utiliser afin d'optimiser un programme ;
- L'existence des **fonctions intrinsèques** dans certains compilateurs modernes a réduit le besoin de l'ASM pour certaines instructions où il était indispensable.

À qui est destiné ce livre ?

Ce livre ne vise aucun public en particulier. Il s'adresse à toute personne désireuse de comprendre le vrai fonctionnement des microprocesseurs et de savoir programmer en langage d'assemblage. Il est néanmoins vrai, que ceux qui veulent devenir des analystes en vulnérabilités applicatives, des chercheurs de vulnérabilités (pentesters), des shell-codeurs, des créateurs d'exploits, des retro-ingénieurs, des programmeurs de virus ou d'antivirus, des développeurs systèmes, seront les plus à même d'apprécier ce genre de contenu.

La lecture de ce livre ne nécessite aucun prérequis. Il faut juste savoir lire et écrire. Et, vous n'avez pas besoin d'un niveau hors normes en mathématiques[21]. Il est vrai que comprendre les opérations de base comme l'addition, la soustraction, la multiplication et la division s'avère obligatoire. Mais c'est tout…Ou presque car, ne pas être un dur à cuire des maths, ne vous dispense pas d'avoir un bon esprit d'analyse et de disposer d'une certaine logique.

---

19 La portabilité est le fait qu'un code puisse être compilé ou assemblé sur différentes machines avec différents microprocesseurs sans modification. L'ASM n'est pas portable. Il est vrai qu'avec les macros on peut faire de l'assemblage conditionnel (si tel microprocesseur il faudra utiliser telle instruction) mais cela reste assez limité.

20 Entre nous, je trouve cela absolument bien. Un programmeur rigoureux en vaut plusieurs.

21 Il est quand même important de souligner que, si l'on n'a pas forcément besoin d'un niveau hors normes en mathématiques pour les logiciels courants, il est tout à fait impossible de réaliser des programmes « orientés mathématiques » comme les logiciels de chiffrement, d'intelligence artificielle ou de modélisation 3D, sans avoir un niveau conséquent en maths.

En résumé

L'ordinateur utilise L'électricité pour effectuer des traitements. Les états des circuits électriques de l'ordinateur sont représentés par des nombres binaires (des 0 et des 1). De longues séquences de nombres binaires sont préenregistrées dans l'ordinateur afin de réaliser des traitements précis. L'ensemble de ces séquences est appelé le langage machine. Pour programmer un ordinateur, il faut savoir utiliser un langage de programmation. Un code écrit dans un langage de programmation doit être « traduit » en langage machine avant d'être utilisable par l'ordinateur. L'ASM est le langage de programmation que nous allons étudier dans ce livre. L'assembleur est le programme qui transforme le code que nous écrivons en ASM en code machine. Dès le prochain chapitre, nous allons aborder les différents composants de l'ordinateur et leurs fonctionnements.

QCM

Dans les sections « QCM » je vais souvent poser des questions qui ont rapport avec les chapitres précédents. Cela dit, voici vos premières questions.

1) Qu'est-ce qu'un programme informatique ?

a) C'est tout fichier qui se termine par « .exe ».
b) C'est un fichier dans lequel on écrit des codes.
c) C'est la définition d'un traitement sur des données.

2) Qu'est-ce qu'un programme exécutable ?

a) C'est tout fichier qui se termine par « .exe ».
b) C'est le résultat obtenu après traduction en code machine d'un programme informatique écrit dans un langage de programmation donné.
c) C'est la définition d'un traitement sur des données.

3) Qu'est-ce qu'un langage de programmation ?

a) C'est le langage de l'ordinateur.
b) C'est un ensemble de mots réservés avec une syntaxe définie qui est utilisé par un programmeur afin de créer des programmes informatiques.
c) C'est le langage dans lequel on discute avec l'ordinateur.

4) Qu'est-ce qu'un langage de haut niveau ?

a) C'est un langage de programmation seulement fait pour les experts.
b) C'est un langage de programmation très difficile.
c) C'est un langage de programmation qui est proche du langage humain et qui propose des couches d'abstraction pour gérer les détails liés à la machine.

5) Qu'est-ce qu'un langage de bas niveau ?

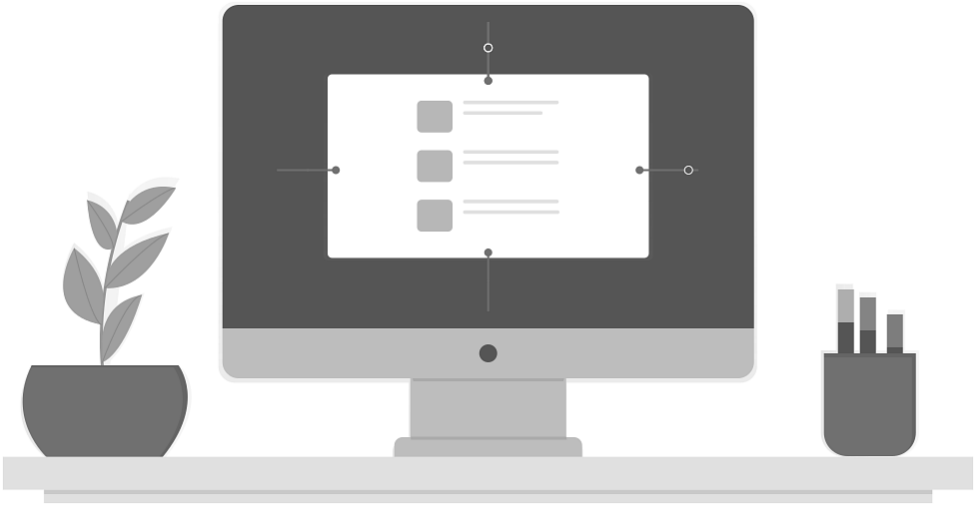
a) C'est le langage de programmation des débutants.
b) C'est un langage de programmation facile à apprendre.
c) C'est un langage de programmation qui ne propose pas trop de couches d'abstraction. Avec ce genre de langage on se soucie encore des détails liées à la machine (mémoire, CPU, périphériques) au lieu d'utiliser des instructions qui cachent et gèrent elles-mêmes ces détails.

6) Qu'est-ce qu'un assembleur ?

a) C'est un langage de programmation
b) C'est un programme qui transcrit des codes écrits en ASM en programmes exécutables.
c) C'est un robot qui assemble des composants informatiques.

CHAPITRE 2 : ORGANISATION DE L'ORDINATEUR

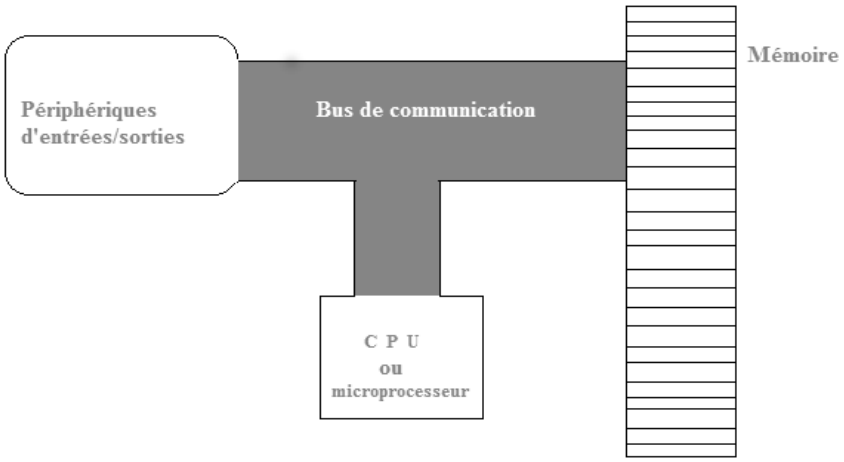
L'ASM se distingue particulièrement de la grande majorité des langages de programmation de plus haut niveau que lui par le fait qu'il est très étroitement lié au fonctionnement interne de l'ordinateur. Écrire un programme, même médiocre, en ASM, demande donc, tout au moins, des connaissances sommaires du fonctionnement interne d'un ordinateur. Mais, pour écrire un code performant, il faudrait avoir de solides bases concernant le fonctionnement de la machine.
Ce chapitre propose d'apporter les connaissances nécessaires pour bien comprendre le fonctionnement d'une machine. Du moins, les aspects de ce fonctionnement qui nous intéresse. Il n'est pas question d'écrire un chapitre de 300 pages. Nous allons juste, poser les bases nécessaires à votre évolution en ASM.



Architecture des ordinateurs

Tout le monde sait ce qu'on appelle un architecte… Bon, je rappelle pour ceux qui l'auraient oublié. Un architecte est juste une personne qui construit des plans pour une chose. Si cet architecte construit, par exemple, les plans d'une maison, il définira la façon dont les différents objets de la maison seront placés. Il dira donc si la porte ou la fenêtre doit être ici, la salle de bain ou une chambre doit être là …

Bref, dites-vous qu'avec un ordinateur c'est pareil. En effet, tout comme il existe un architecte pour une maison, il en existe aussi pour un ordinateur. La conséquence immédiate est qu'il existe aussi des « plans de construction » pour un ordinateur. Ces « plans de construction » sont appelés les architectures. Dans notre cas, on parlera d'architecture matérielle. En informatique, les architectures matérielles sont **simplement des normes qui spécifient la façon dont les composants majeurs du système sont disposés et interagissent pour former un système intelligent**. Les architectures matérielles les plus courantes sont : l'architecture Von Neumann et l'architecture Harvard. Il en existe aussi d'autres, comme les architectures Dataflow. Un programme écrit sur une architecture, ne fonctionnera pas sur une autre architecture nativement. À moins de s'armer d'un émulateur. L'architecture qui nous intéresse est celle de Von Neumann car elle est la plus utilisée lorsqu'il s'agit des ordinateurs. Dans cette architecture, on utilise une seule mémoire pour stocker à la fois les instructions et les données des programmes. La figure suivante présente l'architecture Von Neumann[22].



Architecture Von Neumann

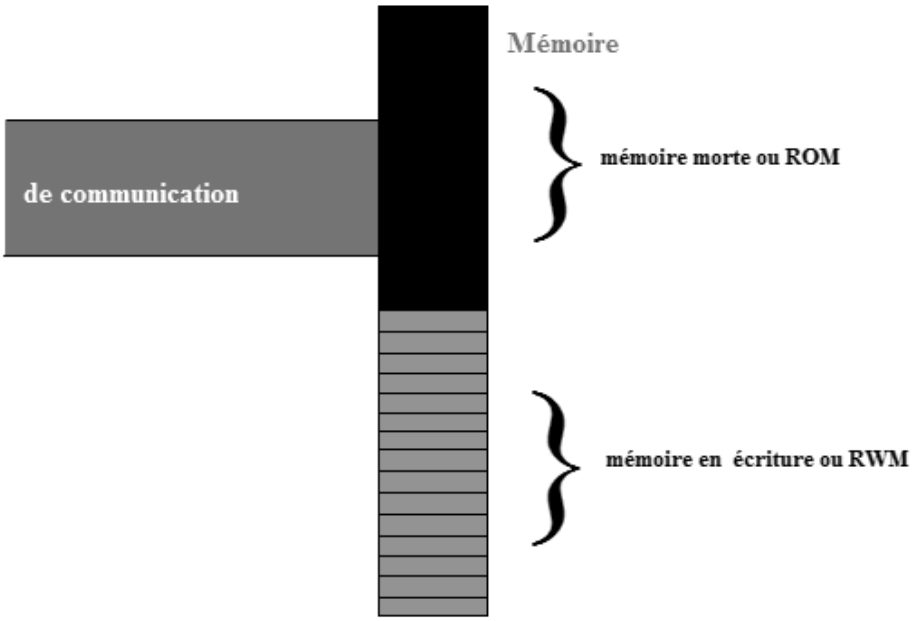
Sur le schéma on voit quatre composants :

- le CPU, processeur, ou encore microprocesseur ;
- ensuite, il y a la mémoire ;

22 La partie CPU a évolué au cours du temps, pour englober l'UAL (Unité Arithmétique et Logique) et l'UC (Unité de contrôle), des registres, des bus internes, et une interface externe. Voir l'article Wikipédia pour plus de détails.

- puis, les périphériques d'entrées/sorties ;
- et enfin le tout est relié par un bus de communication.

Nous n'allons pas nous mettre à expliquer ce que représente chaque composant du système tout de suite. On va plutôt, en faire de brèves descriptions. Commençons par la mémoire. En effet, après un zoom sur ce dernier, on se rend compte qu'il est composé de deux types de mémoires : la mémoire ROM et la mémoire RWM.



Zoom sur la partie mémoire de l'architecture Von Neumann

La mémoire ROM

Encore appelée mémoire morte, la mémoire ROM (Read Only memory) est comme son nom l'indique, une mémoire en lecture seule. Bon, vu la tête que vous faites, je vais reformuler ce que je viens de dire. En fait dans un ordinateur, on peut grossièrement classer les types de mémoires en deux catégories :

- les mémoires en lecture seule (ROM) ;
- les mémoires en lecture/écriture (RWM).

La mémoire ROM est une mémoire spéciale qui n'autorise pas qu'on puisse y inscrire, modifier ou supprimer des données. La seule opération[23] qui peut être

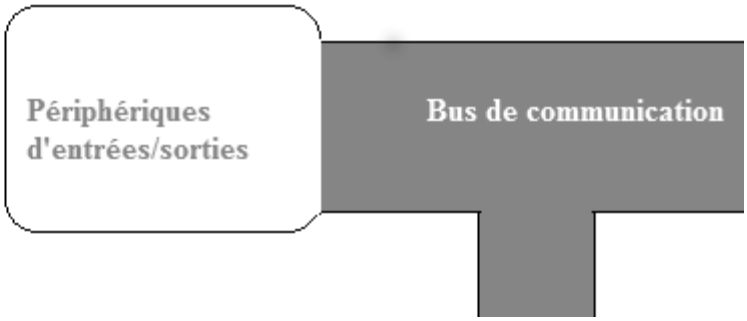
23 On peut parfois écrire dans ce type de mémoire. On parle alors de ROM programmable. Dans ce cas, un utilisateur expérimenté, avec un matériel spécial modifie cette mémoire. Les types de mémoires ROM modifiables sont les UVPROM, les PROM, les EPROM et les EEPROM.

effectuée sur ce type de mémoire est la lecture des données qui y sont déjà présentes (inscrites lors de la fabrication de la mémoire). On utilise ce type de mémoire pour les programmes qui doivent être rarement mises à jour. Une des utilisations classique de la ROM est le BIOS des ordinateurs. En effet, vu qu'un ordinateur n'est qu'un bout de métal qui ne sait quoi faire, les programmeurs ont créé un programme qui réside de façon permanente dans sa mémoire et qui démarre en même temps que lui. Ce programme est le BIOS. Il se charge d'initialiser les composants de la carte mère, de dire quels accessoires (clavier, disques durs, RAM…) sont présents et bien configurés et ensuite il passe la main au système d'exploitation[24]. Vu le caractère crucial du travail du BIOS, il a été placé en ROM. En effet, les données inscrites dans la ROM y demeurent en permanence même lorsqu'on éteint l'ordinateur[25].

La mémoire RWM

Encore appelée mémoire de travail, la mémoire de type RWM (Read Write Memory) est une mémoire en lecture et en écriture. Ainsi, on peut lire, modifier, supprimer, remplacer, ou ajouter des données sur ce type de mémoire sans restriction. En fait, une partie de cette mémoire est utilisée pour stocker les données utiles à nos programmes. Une autre partie est utilisée pour stocker le programme lui-même. L'avantage, est qu'on peut modifier le fonctionnement des programmes stockés sur ce type de mémoire.

On peut ainsi, facilement installer ou désinstaller des programmes de notre ordinateur. Ou encore, modifier leurs données durant leur exécution. Comme types de mémoire RWM on peut citer, par exemple, les disques durs et la mémoire RAM (Random Access Memory). Lorsque les programmes sont créés, ils sont stockés sur le disque dur. Lors de leur exécution par contre, ils sont chargés en mémoire RAM par le système d'exploitation. Passons maintenant aux périphériques.

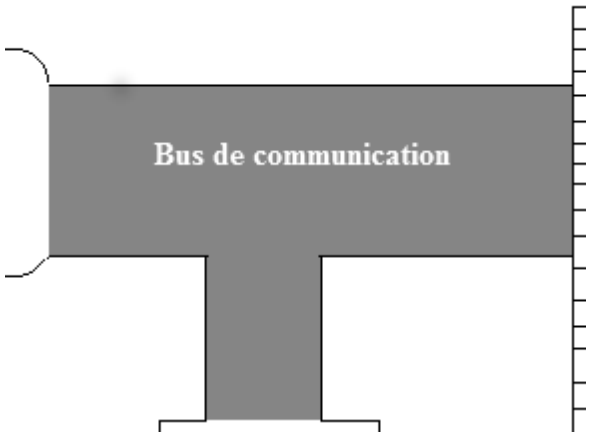


Zoom sur les périphériques de l'architecture Von Neumann

24 En fait, il passe la main au chargeur d'amorçage ou boot loader qui, lui, démarre le noyau du système d'exploitation.

25 Cet article Wikipédia, détails assez bien, le travail du BIOS. Nous allons revenir sur le BIOS lorsqu'on va aborder la création de notre propre boot loader.

Les périphériques d'entrées/sorties
Les périphériques sont des composants externes au microprocesseur et qui nous permettent d'échanger des informations avec lui. Par exemple, en lui fournissant des données lors de l'exécution d'un programme, on parlera d'entrée. Et, lorsque lui, il nous fournit des résultats suite à une opération, on parlera de sortie car la donnée vient de cette fois du microprocesseur (à travers un périphérique) et non de nous. Comme périphériques d'entrée, on peut citer le clavier et la souris. Et comme périphériques de sorties on peut citer l'écran[26] et l'imprimante.



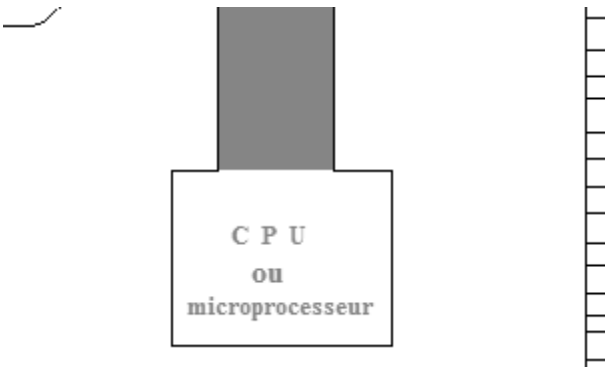
Zoom sur le bus de communication

Un bus pour transporter les informations
Comme cela a été dit précédemment, un microprocesseur, communique souvent avec la mémoire afin d'y enregistrer ou d'y extraire des données. Ces opérations ne sauraient se faire sans la présence du bus de communication. Le bus permet au microprocesseur de communiquer avec le monde extérieur. Ainsi, le bus de communication permet les échanges de données entre le microprocesseur et les autres composants d'un ordinateur.

Une analogie pertinente, serait de comparer les bus de communication aux transports en commun qui relient deux villes distantes. La seule différence ici est qu'au lieu de transporter des passagers, il transporte plutôt des informations. Nous reviendrons beaucoup plus tard, en détails sur le bus de communication[27]. Cela dit, il est temps de parler du dernier composant de l'architecture Von Neumann : le microprocesseur.

26 Dans certains cas, l'écran peut être, à la fois, une sortie et une entrée. C'est le cas des écrans tactiles. En effet, ils permettent de saisir des données (entrée) et de les afficher (sortie).

27 Ce bus est aussi appelé bus externe car le microprocesseur possède en son sein même, un bus (bus interne) qui permet le transport des données entre ses organes internes.



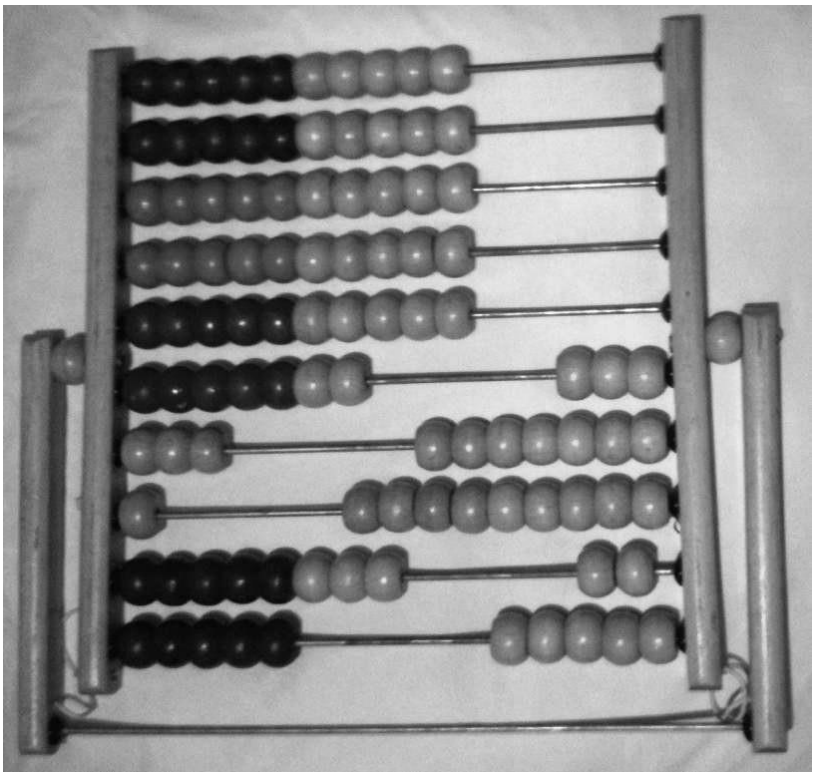
Zoom sur la partie microprocesseur de l'architecture Von Neumann

Le microprocesseur

Il était une fois...

Avant de parler des microprocesseurs, nous allons tout d'abord nous intéresser à l'histoire de l'ordinateur. En effet, si l'ordinateur apparaît comme une invention relativement récente, sa mise au point à nécessité plusieurs millénaires d'évolution et de découvertes successives. Ainsi, depuis la préhistoire (non, non, je n'exagère pas), pour faciliter les calculs, l'homme a inventé différents moyens pour arriver à calculer le plus rapidement possible tout en ayant des résultats fiables.

Tout d'abord ça a commencé avec les bouliers et abaques (environ 500 ans avant JC) puis vers 1632 la règle à calculs a été inventée. Il s'en est suivi plusieurs inventions comme l'invention de la Pascaline, de la machine de Babbage…



Un boulier (source Wikipédia)

Bref, toutes ces inventions ont progressivement, contribué à l'évolution de la notion de calcul et à l'élaboration d'une machine capable de calculer. Mais, les choses ont réellement commencé à se dessiner, grâce à trois faits majeurs :

- publication d'un ouvrage sur la logique par le scientifique Boole (1854);
- invention du tube à vide (1904) ;
- et l'article d'Allan Turing sur la calculabilité : machine de Turing (1937).

Quelques années après ces trois faits, des machines à calculer mécaniques sont nées. Plus tard, l'ENIAC, première machine électronique vit également le jour : c'était l'ancêtre de l'ordinateur. Mais, à cette époque, ces « ordinateurs » avaient au moins la taille d'un homme. Vous ne me croyez pas ? Voyez l'image suivante. En réalité, cette taille était due aux processeurs qui avaient une très grande taille et pesaient des tonnes.

ENIAC

La découverte des **transistors** et des **circuits intégrés**, favorisa un fait marquant : en 1971 une société connue sous le nom d'Intel, avait fabriqué une puce électronique composée de 2300 transistors (l'Intel 4004) pour un fabricant de calculatrices. Plus tard, deux officiers de l'armée américaine utilisèrent alors l'une de ces puces (une version évoluée du Intel 4400 : le 8080A) pour fabriquer l'ALTAIR 8800 (le vrai ancêtre des ordinateurs tel qu'on les connait aujourd'hui).

Le logo de la société Intel

En effet, cette puce électronique, qui n'est rien d'autre qu'un microprocesseur possédait la puissance et les composants nécessaires pour créer ce qu'on appelle un micro-ordinateur (plus connu sous le nom d'ordinateur ou de PC de nos jours) mais personne n'y avait pensé sauf ces officiers.

L'Altair 8800 (source Wikipédia)

Un microprocesseur (de micro qui veut dire petit, minuscule), est en fait un petit processeur. Il résulte donc d'une miniaturisation (comprenez, une mise à une plus petite échelle) des divers composants d'un processeur. De nos jours, c'est bien plus qu'une miniaturisation : plusieurs composants des processeurs originels sont remplacés par un seul composant dans les microprocesseurs modernes[28] rendant plus simple l'architecture interne des microprocesseurs. Mais complexifiant le travail de chaque composant qui se retrouve à faire le travail de plusieurs composants.

Un problème d'architecture

Mais, revenons à Intel. Convaincu par son succès avec le 8080, il sort beaucoup d'autres microprocesseurs jusqu'en 1978 où il sort le 8086 qui connut moins de succès que ce que la société attendait. En effet, le 8086 est un microprocesseur 16-bit (comprenez par cela qu'il manipule des paquets de 0 et de 1 par groupe de 16 – on reviendra dessus). Or, la plupart des périphériques de l'époque, était des périphériques dotés de bus de communication de 8-bit et étaient, par conséquent,

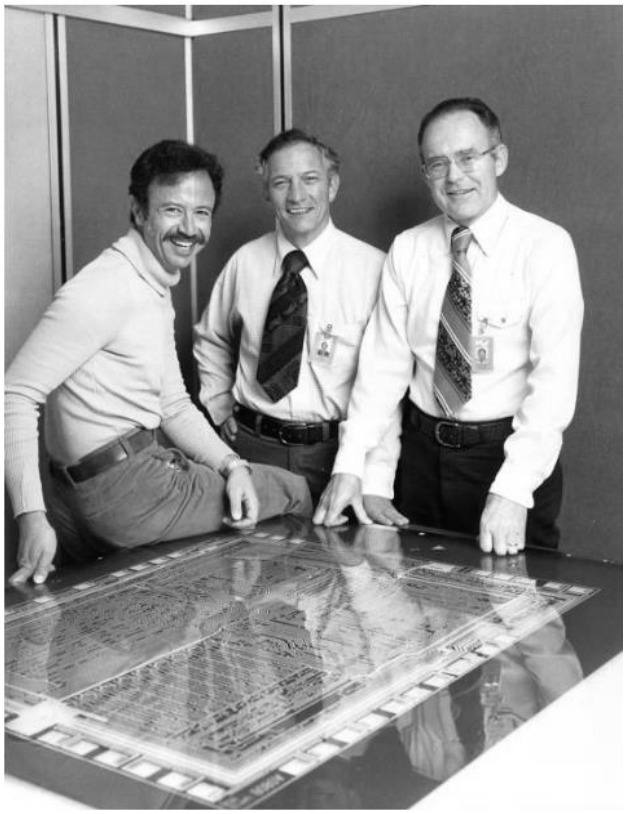
28 Si vous êtes intéressé par le processus de fabrication des microprocesseurs, voici une page qui résume un peu cela.

incompatibles avec le 8086. En plus de tout cela, le 8086 coûtait un peu cher à cause des capacités dont il était doté à l'époque.

Pour pallier à cet échec, la société Intel inventa le 8088 qui était quasiment identique (comprenez possède la même architecture) au 8086. La seule vraie différence, était que le 8088 disposait d'un bus de 8 bits et coûtait moins cher. La compatibilité du 8088 avec les périphériques de l'époque, son prix abordable et ses capacités firent de lui le microprocesseur dont IBM (une firme qui contrôlait la quasi-totalité des ventes d'ordinateurs à travers le monde) voulu doter sa gamme d'ordinateurs personnels (PC).

Le quasi-monopole d'IBM dans le domaine des ordinateurs fit la fortune d'Intel. En effet, vu que beaucoup de personnes achetaient les PC d'IBM, beaucoup de logiciels ont été créés pour cette plate-forme. Vous allez rire mais sachez qu'un microprocesseur possède également une architecture[29] (non, pas de panique je ne vais pas me mettre à présenter aussi les architectures des microprocesseurs). Le problème, c'est qu'à l'instar du fait qu'un programme écrit pour une architecture d'ordinateur ne fonctionnera pas sur une autre architecture d'ordinateur, un programme écrit pour une architecture de microprocesseur, ne fonctionnera pas non plus sur une autre architecture de microprocesseur. Or, les années qui ont suivis après le 8088, IBM continua à choisir les microprocesseurs Intel qui ne cessaient d'évoluer à chaque nouvelle version. Afin de maintenir une compatibilité entre les différentes versions des PC IBM, et que les anciens logiciels continuent à s'exécuter sur les PC de nouvelle génération dotés de nouveaux microprocesseurs, les ingénieurs de la société Intel devaient inventer une technique pour maintenir l'architecture du 8088 (qui en fait est quasiment la même architecture que le 8086) tout en créant de nouveaux microprocesseurs.

[29] Il en y en a principalement 2 : l'architecture RISC et l'architecture CISC. Les x86 ont une architecture CISC.



Les trois fondateurs d'Intel (Andy Grove, Robert Noyce, et Gordon Moore) devant le schéma du 8080

La solution : rétrocompatibilité

Pour finalement résoudre ce problème, Intel eu une géniale idée : garder l'architecture du 8086/8088 tout en ajoutant de nouvelles fonctionnalités. Ainsi, chaque fois qu'ils voudraient améliorer leur microprocesseur, ils devront le faire, sans supprimer les anciennes fonctionnalités. C'est ce qu'on appelle la compatibilité descendante ou rétrocompatibilité. Cela semble facile dit comme cela, mais, Intel a dû mettre en place des techniques très complexe pour arriver à ce résultat.

Ainsi, dans les années qui suivirent, lorsqu'Intel créa le 80186, il inclut les fonctionnalités du 8086 avant d'y ajouter des améliorations. Il en fit de même pour le 80286, qui cette fois, incluait non seulement les fonctionnalités du 8086, mais aussi, celles du 80186 (souvenez-vous que désormais, un microprocesseur doit être compatible avec tous ses prédécesseurs) avant d'inclure les propres fonctionnalités du 80286 (les améliorations).

x86 : une affaire de famille

Les microprocesseurs dont le « nom » était terminé par 86 commençaient à sérieusement s'agrandir. Ainsi, est née la grande famille des microprocesseurs x86 ou 80x86. Le « x » est mis pour la partie variable du « nom ». Par exemple, dans 8086, le « x » est 0 (il n'y a rien entre 80 et 86). Dans 80286, le x est plutôt le 2. En fait, le x représente tout simplement la sous-famille ou génération du

microprocesseur. Je m'explique : quand on prend le 80386 par exemple, il a d'autres frères qui lui ressemblent : 80386DX et 80386SX. Alors, pour désigner tous les microprocesseurs 80386 et ses clones, on dira les 386 c'est-à-dire les microprocesseurs x86 de la troisième génération avec leurs clones (frères).

Mais, pour désigner plutôt la famille toute entière, on ne précisera pas la valeur du x. On dira tout simplement les x86, qui regroupent toute la famille des 8086 et clones, des 80186 et clones et ainsi de suite jusqu'à la toute dernière génération. La famille des microprocesseurs x86 désigne donc, tous les microprocesseurs dont le jeu d'instruction est compatible entre eux et compatible au 8086 originel. Ainsi, un programme qu'on peut exécuter sur notre bon vieux 8086 devrait sans problème s'exécuter sur le dernier né des microprocesseurs Intel[30] appartenant à la famille x86. Mais, pas dans le sens contraire. En effet, vous ne pourrez pas exécuter un programme qui, lors de sa création, a été écrit seulement avec des instructions du 80386 sur un 8086. Car, ce dernier, n'incluait pas ces instructions lors de sa fabrication.

La ruée vers les x86

Intel ayant de plus en plus du succès, d'autres sociétés se lancèrent aussi dans la fabrication de microprocesseurs. Les sociétés comme AMD et Motorola se firent vite un nom. Mais, la famille des x86 n'arrêtant pas d'avoir du succès, AMD et d'autres sociétés (Cyrix, Rise technologie et VIA) se sont lancés dans la fabrication de leurs propres microprocesseurs x86. Ainsi, aujourd'hui, bien qu'Intel domine en grande partie le marché des microprocesseurs pour PC grâce aux x86, AMD est également non-négligeable et fabrique aussi en quantité (et en qualité) des microprocesseurs appartenant à la famille x86.

Changement de nom mais pas de famille

Aux États-Unis, les noms de produits composés seulement de nombres ne peuvent être protégé (c'est d'ailleurs pour cela que les sociétés comme AMD ont pu créer des x86 bien que ce soit Intel qui ait sorti ce concept). Ainsi, pour éviter que d'autres sociétés se fassent la publicité sur son dos, Intel décida d'appeler ses microprocesseurs, à partir de la 5ème génération, par de « vrai » noms. Par exemple, le 586 est appelé Pentium.

Plus tard, après le Pentium, d'autres microprocesseurs ont été créé. Aujourd'hui, ce sont les microprocesseurs x86-64 qui sont les plus répandus sur le marché. Le « 64 » sert à préciser que, bien que ce sont des x86, ils manipulent des paquets de données par groupe de 64bits.

Je crois qu'avec toutes ces explications, vous comprenez enfin pourquoi mon choix s'est porté vers la famille des x86. Ils sont les microprocesseurs les plus répandus lorsqu'il s'agit des ordinateurs. Si vous n'avez donc pas un x86, vous ne pourrez pas utiliser ce livre. Il est quand même important de noter une chose, pour ceux qui auraient des doutes, la famille x86 d'AMD de Rise technologie, VIA,

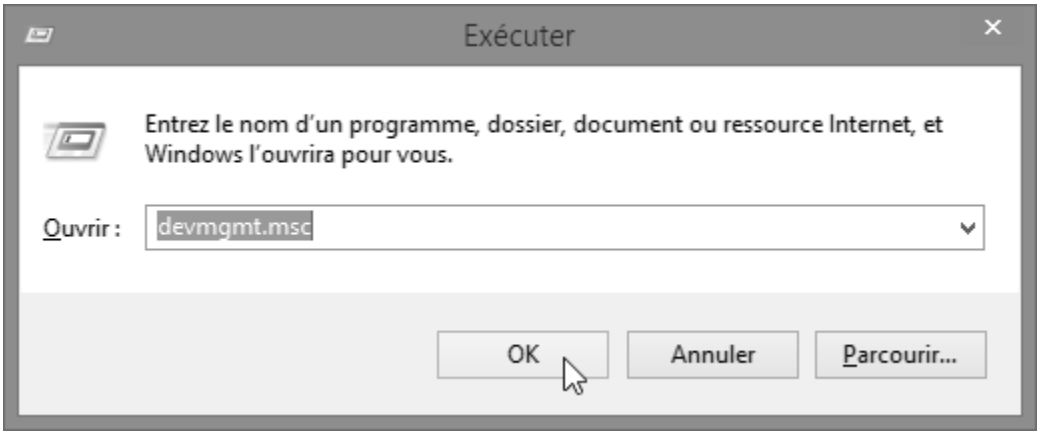
[30] Si le système d'exploitation permet l'activation du mode de compatibilité approprié à l'exécution du programme. Ce n'est pas toujours le cas. Ainsi, bien que les derniers microprocesseurs x86-64 permettent d'exécuter du bon vieux code 16 bits. Les versions 64 bits de Windows ne permettent pas d'activer le mode de compatible adéquat.

ou encore de Cyrix ont un jeu d'instructions qui est compatible entre eux et à Intel[31]. À très peu de différences près, un programme écrit sur l'un, fonctionnera donc sur les autres.

Pour ceux qui ne savent pas s'ils disposent d'un x86, pas de panique. Même si vous n'avez pas un x86, vous pourrez toujours faire de l'ASM mais…pas avec ce livre. En fait, strictement parlant, même si l'ASM varie d'une architecture de microprocesseur à un autre, certaines bases sont souvent communes à tous les ASM, comme par exemple l'utilisation du binaire, l'existence des instructions, les registres, la pile…Ce sont plutôt les mnémoniques, le nom des registres et l'architecture du microprocesseur qui varient souvent.

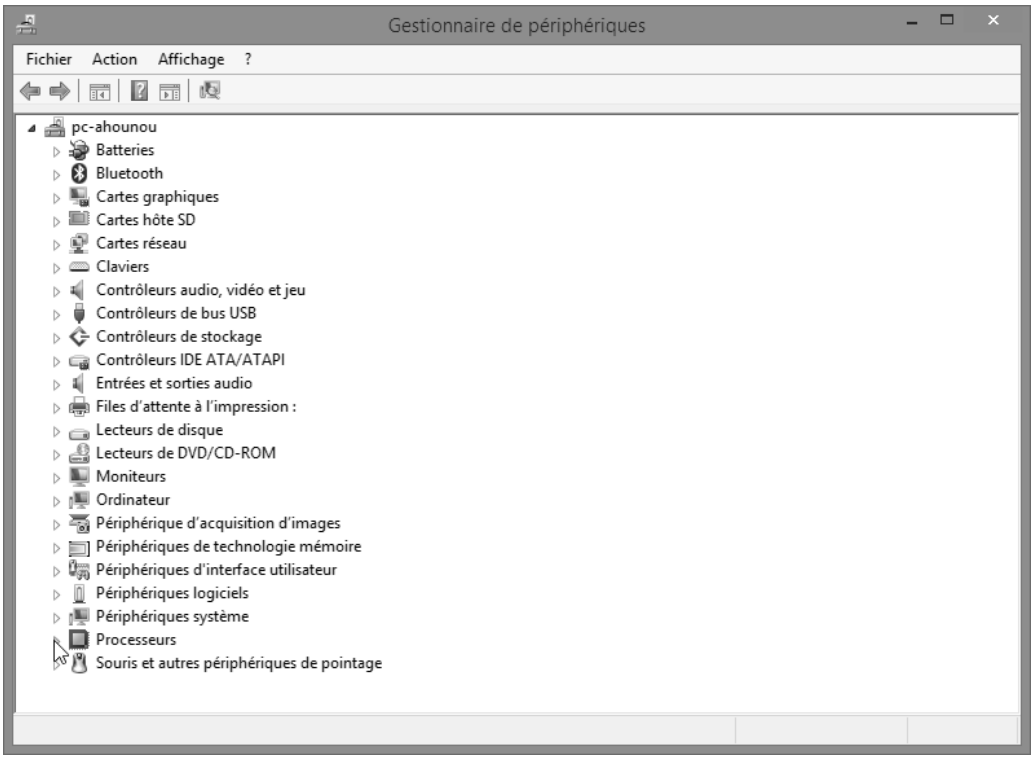
Vous pouvez donc toujours utiliser ce livre car j'ai essayé au maximum de généraliser ce qui pouvait l'être. Mais, lors des exercices, quand il faudra essayer les codes sources de ce livre, vous ne pourrez le faire nativement. Il faudra donc vous équiper d'un émulateur qui simule l'architecture x86 et qui est disponible pour votre ordinateur.

Pour savoir si votre microprocesseur est un x86, j'ai une petite astuce. Maintenez la touche Windows (celle avec le logo de Windows) et enfoncez la touche R. Une boite de dialogue devrait s'afficher :

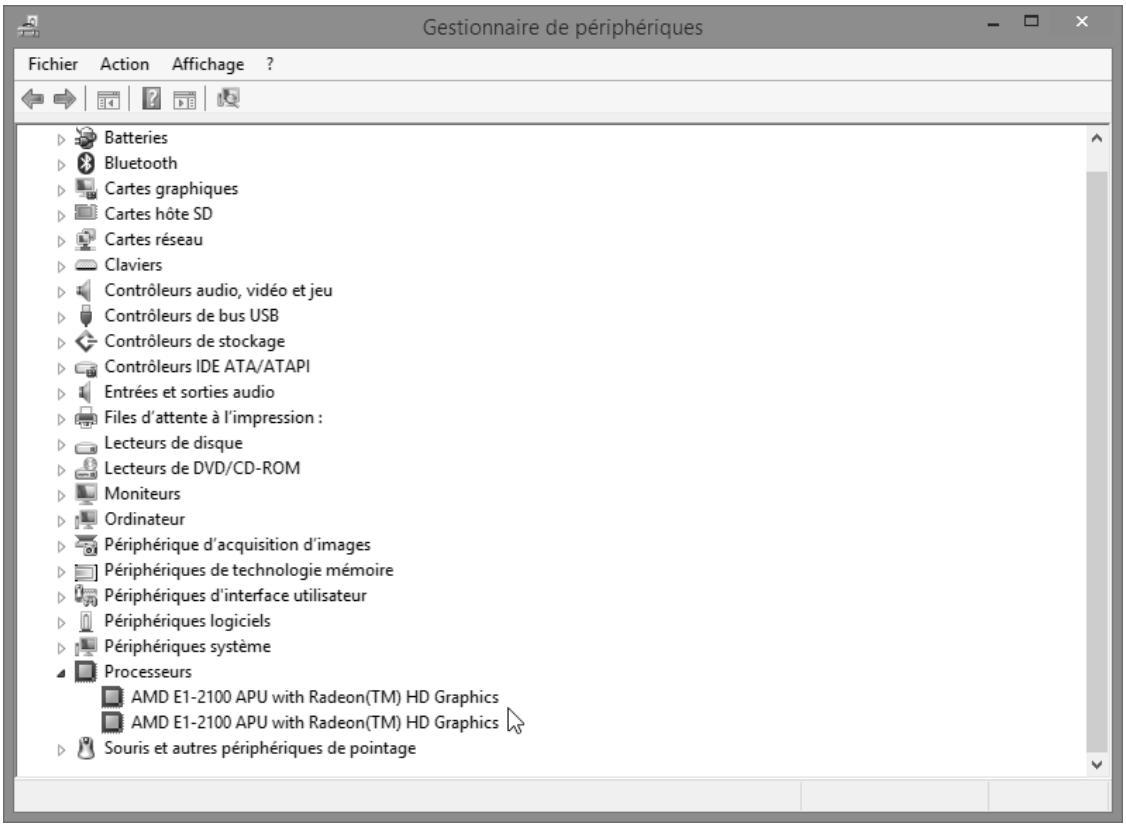


Tapez `devmgmt.msc` dans la boite de dialogue comme sur la capture d'écran précédente et validez ensuite en cliquant sur le butons « ok ». Là devrait apparaître une fenêtre qui ressemble à ceci :

31 Lorsqu'il s'agit des x86, Intel et AMD sont tout à fait compatibles entre eux exception faite des jeux d'instruction spécifiques 3DNow! et SSE2.

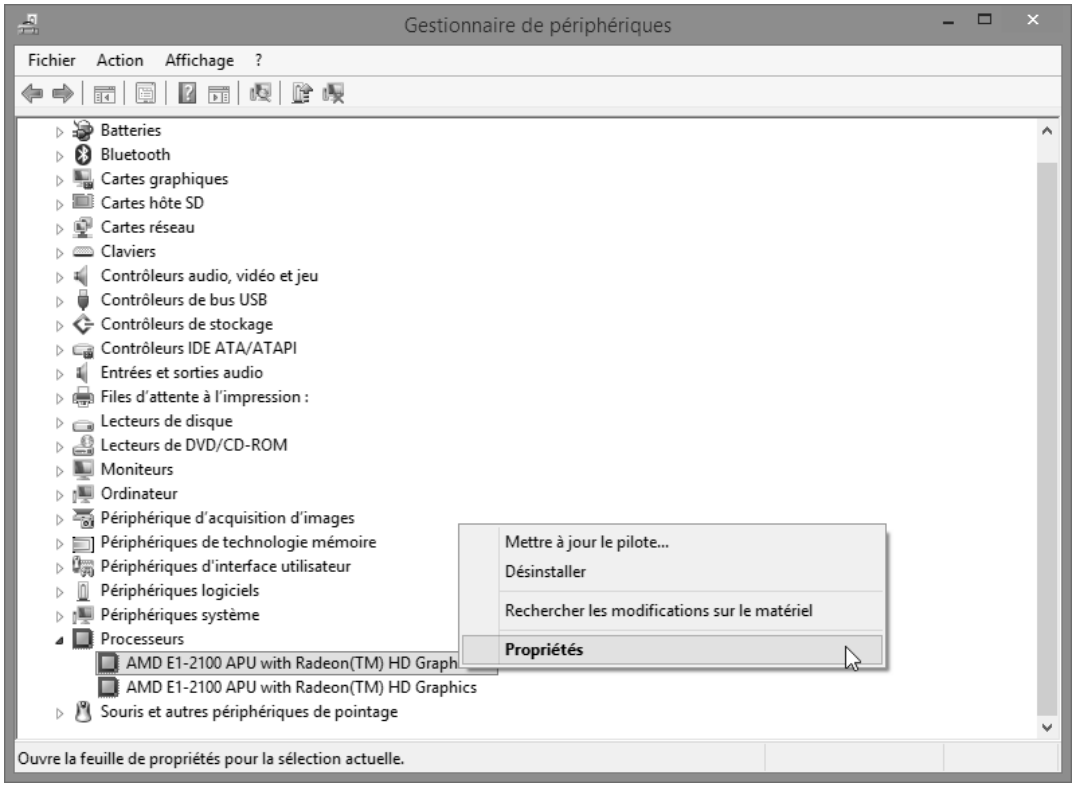


Regardez en bas et double-cliquez sur « processeurs ». Vous obtiendrez le nom de votre microprocesseur[32].

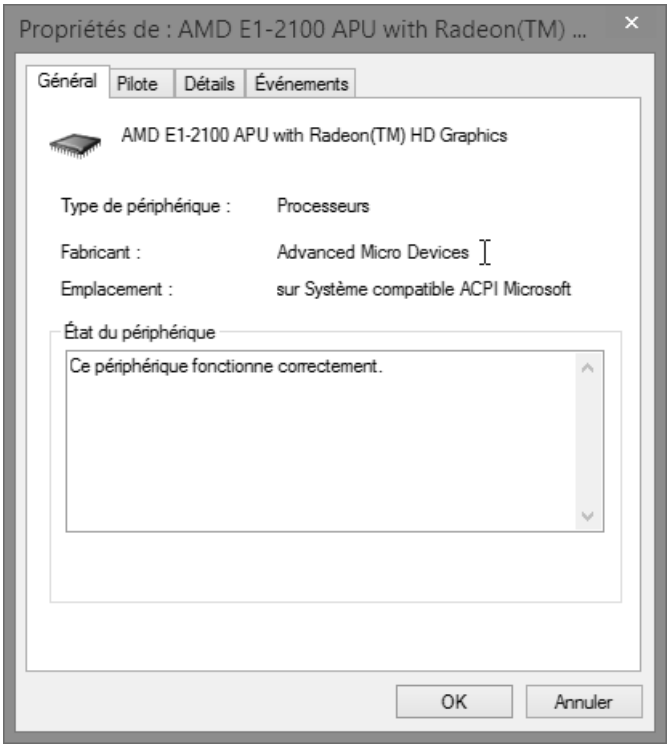


32 Si vous vous demandez pourquoi j'ai deux fois le même nom, c'est parce que j'ai un microprocesseur à deux cœurs.

Alors maintenant cliquez droit sur le nom de votre microprocesseur et choisissez sur « propriétés ».

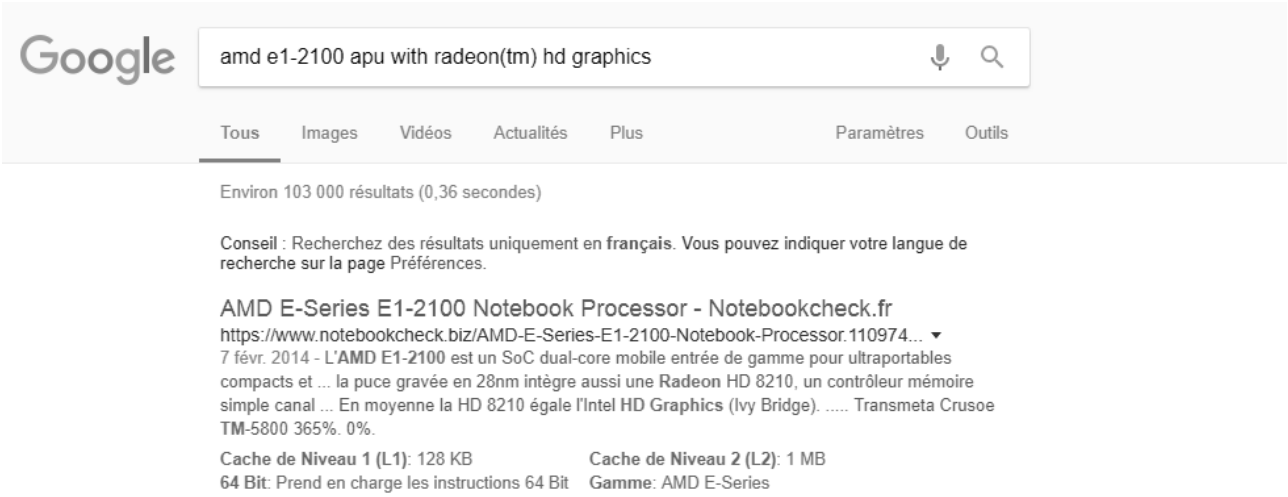


Là vous aurez une fenêtre semblable à la capture d'écran suivante.



Si comme moi votre fabricant est AMD (ou encore Intel, Cyrix, Rise technologie ou tout autre fabricant de x86) alors il y a de grandes chances que vous ayez un

x86-32 ou un x86-64. Pour en être sûr, tapez le nom de votre microprocesseur dans le moteur de recherche Google. Dans mon cas, voici ce que cela donne :



En cliquant, par exemple, sur le premier résultat de recherche, voilà ce que j'obtiens :

Gamme	AMD E-Series
Nom de code	Kabini
Gamme: E-Series Kabini	**AMD E2-3000** 1650 MHz 2 / 2 **AMD E1-2500** 1400 MHz 2 / 2 **AMD E1-2200** 1050 MHz 2 / 2 » **AMD E1-2100** 1000 MHz 2 / 2
Fréquence	1000 MHz
Cache de Niveau 1 (L1)	128 KB
Cache de Niveau 2 (L2)	1 MB
Nombres de cœurs/threads simultanés supportés	2 / 2
Consommation énergétique maximale (TDP = enveloppe thermique)	9 Watt(s)
Lithographie (procédé de fabrication)	28 nm
Fonctionnalités	SSE (1, 2, 3, 3S, 4.1, 4.2, 4A), x86-64, AES, AVX, DDR3L-1333
64 Bit	Prend en charge les instructions 64 Bit
Date de présentation	05/23/2013 = 1857 days old

En regardant la ligne « fonctionnalités » on voit que c'est bien un x86 (plus précisément un x86-64). Vous pouvez également recherchez le nom de votre microprocesseur sur internet et trouver son type avec cette méthode[33].

33 Il y a d'autres utilitaires qui peuvent être utilisé pour identifier les fonctionnalités dont est doté votre microprocesseur. Par exemple, l'utilitaire CPUID CPU-Z est très utile dans ce cadre.

Remontons le temps

Ici, on va voir une brève liste qui montre l'évolution des microprocesseurs x86. De plus, on va établir quelques conventions qui nous permettront, plus tard, de savoir quand est apparue une instruction. On évitera ainsi, d'utiliser des instructions qui visent un microprocesseur évolué sur des microprocesseurs antérieurs. Voici un tableau qui trace un peu l'évolution des x86.

Génération	Date de parution	Microprocesseurs
1 (x86-16)	1978	Intel 8086, Intel 8088
2	1982	Intel 80186, Intel 80188, NEC V20/V30
		Intel 80286
3 (IA-32) (x86-32)	1985	Intel 80386, AMD Am386
4	1989	Intel 80486, AMD Am486
5	1993	Pentium, Pentium MMX
5/6	1996	Cyrix 6x86, Cyrix MII, Cyrix III(2000) / VIA C3 (2001)
6	1995	Pentium Pro, AMD K5 Nx586 (1994), Rise mP6
	1997	AMD K6/-2/3,Pentium II/Pentium III, IDT/Centaur-C6
7	1999	Athlon, Athlon XP
	2000	Pentium 4
8 (x86-64)		Athlon 64, Opteron
	2004	Pentium 4 Prescott
9	2006	Intel Core 2
10	2007	AMD Phenom
11	2008	Intel Atom
		Intel Core i7
		VIA Nano
12	2010	Intel Sandy Bridge, AMD Bulldozer
13	2013	Intel Haswell
14	2015	Intel Skylake
15	2016-2017	Intel Kabylake, AMD Zen

Cette liste n'est pas exhaustive. Elle montre simplement les microprocesseurs x86 les plus répandues depuis l'originel 8086 jusqu'aux récents x86-64.

Quelques notations

Il est aussi bien de connaître, la signification de certaines notations. Ainsi, au lieu de x86, vous pourrez parfois voir i86 (intel86) qui précise que ce sont les x86 d'Intel. Mais, les deux notations sont équivalentes. Il y a aussi d'autres notations comme IA-32 (Intel Architecture 32 bits) et Intel 64. IA-32, x86-32, ou i386 fait

référence aux microprocesseurs x86 à partir du 386 qui sont 32 bits. Avant eux, on retrouve plutôt les x86-16 (ils manipulent des bits par paquets de 16). Quant aux Intel 64 (ou encore x86-64, x64, x86_64, EM64T, AMD64, IA-32e)[34], ce sont tous de synonymes. Il s'agit des microprocesseurs 64 bits qui disposent de la capacité d'activer la compatibilité avec les précédents x86. Quand on ne précise rien après le x86 on fait référence à toute la grande famille.

Quelques conventions

Pour faciliter la lecture du livre, dès le prochain chapitre, lorsque je présenterai une instruction, je préciserai à partir de quel microprocesseur elle est valide. En effet, certaines instructions sont apparues plus tard sur les microprocesseurs x86. Bien que ces microprocesseurs puissent exécuter les instructions de leurs prédécesseurs, le contraire n'est pas possible à cause de la compatibilité descendante. Il serait alors problématique d'aborder des instructions sans savoir à partir de quels microprocesseurs on peut les exécuter. Ainsi, lorsque je ne précise rien alors tous les microprocesseurs depuis l'original 8086 peuvent exécuter l'instruction présentée. Sinon, je mettrai entre parenthèses le microprocesseur à partir duquel l'instruction est exécutable.

Par exemple, si je présente une instruction comme `MOV`, je ne dirai rien mais, par contre, si je mets une autre instruction comme `MOVZX` je mettrai aussi (386+). Cela signifie qu'elle est apparu depuis le 80386 et qu'il faut au moins un 386 ou supérieur pour espérer l'exécuter. Vous avez maintenant les bases nécessaires en architecture d'ordinateur pour évoluer dans le monde de l'ASM.

34 Il ne faut surtout pas confondre, les processeurs **x86-64** et **IA-64** dont les instructions sont incompatibles entre elles. Pour plus d'informations visitez la page Wikipédia.

En résumé

Écrire un bon code en ASM demande une bonne compréhension du matériel. Une simple connaissance des instructions du microprocesseur est donc insuffisante. Nous avons abordé les principaux composants de l'ordinateur. On retrouve donc dans un ordinateur, le microprocesseur, qui est le cerveau de l'ordinateur. Car c'est lui qui exécute les instructions et détermine le langage machine à utiliser. On retrouve aussi, la mémoire qui stocke les instructions et les données nécessaires au fonctionnement du programme en cours d'exécution. Il y a également, les périphériques qui sont des composants externes au microprocesseur qui interprètent les données (affichage à l'écran, lecture de son, impression…). Le bus de communication, quant à lui, il permet la circulation des données entre le microprocesseur, la mémoire et les périphériques.

QCM

1) Qu'est-ce qu'une architecture matérielle ?

a) C'est l'ensemble des matériaux qu'on utilise pour construire un ordinateur.
b) C'est une norme qui régit la façon dont les composants majeurs d'un système informatique, sont disposés et interagissent pour former un système intelligent.
c) C'est un plan de construction, dont se sert chaque constructeur d'ordinateur dans le monde.

2) Quels sont le composants de l'architecture Von Neumann ?

a) La carte mère, le BIOS et le système d'exploitation.
b) La mémoire (ROM et RWM), le bus de communication, les périphériques, et le microprocesseur.
c) La mémoire et le microprocesseur.

3) C'est quoi une mémoire ROM ?

a) C'est une mémoire rarement mise à jour, où seule, la lecture des données est permise.
b) C'est une mémoire spéciale que seules les élites ont.
c) C'est une mémoire greffée sur le microprocesseur.

4) C'est quoi une mémoire RWM ?

a) C'est une mémoire difficile à fabriquer.
b) C'est une mémoire où l'écriture et la lecture de données sont permises.
c) C'est une mémoire que seuls les programmeurs les plus doués peuvent manipuler.

5) Où sont stockés les programmes lorsqu'ils ne sont pas en exécution ?

a) Ils sont stockés dans le CPU.
b) Ils sont stockés dans la mémoire RAM.
c) Ils sont stockés dans le disque dur.

6) Dans quel composant de l'ordinateur sont chargés les programmes lors de leur exécution ?

a) Ils sont chargés dans le bus de communication.
b) Ils sont chargés dans mémoire RAM.
c) Ils sont chargés dans le disque dur.

7) C'est quoi le microprocesseur ?

a) C'est une puce électronique qui effectue des calculs et exécute des instructions à l'intérieur d'un ordinateur.
b) C'est un composant permet à l'ordinateur de traiter du son.
c) C'est le petit frère d'un ordinateur.

8) C'est quoi un x86 ?

a) C'est un nombre que seul l'ordinateur comprend.
b) C'est un code informatique.
c) C'est tout microprocesseur compatible de façon descendante à l'original 8086 d'Intel.

9) C'est quoi un x86-64 ?

a) C'est une expression que seul un ordinateur peut comprendre.
b) C'est un code écrit en ASM.
c) C'est un microprocesseur 64 bit qui est compatible de façon descendante à la famille des x86.

10) C’est quoi un IA-64 ?

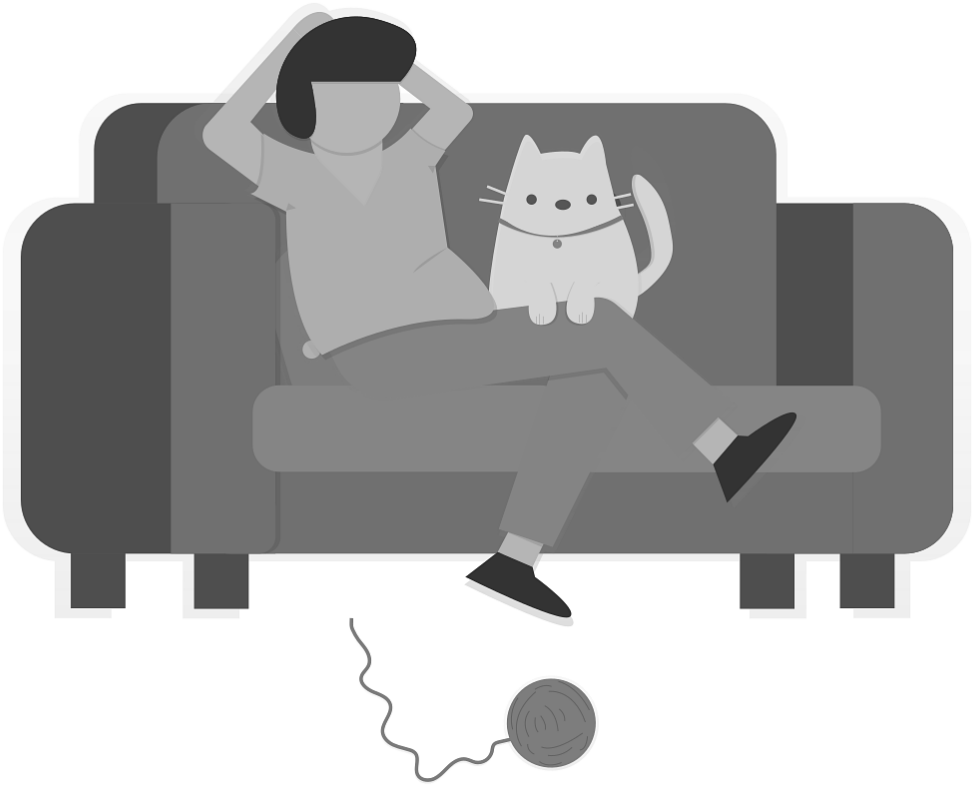
a) C’est un microprocesseur Intel 64 bit, qui n’est pas du tout compatible avec les x86.

b) C’est une Intelligence Artificielle de la 64ème génération.

c) C’est un microprocesseur Intel 64 bit, qui est compatible de façon descendante à la famille des x86.

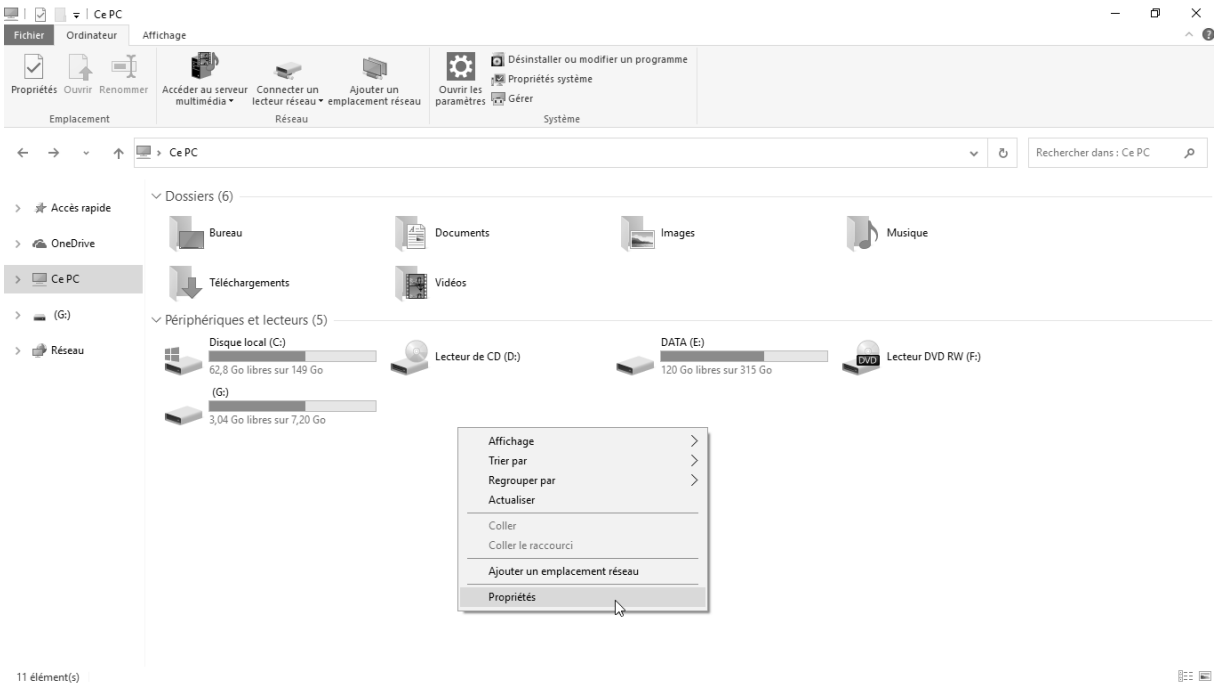
CHAPITRE 3 : UNE PETITE PAUSE, LES OUTILS

Après toutes ces théories plus ou moins harassantes, nous allons prendre une petite pause. En effet, ce chapitre ne demande pratiquement aucun effort. On va juste installer et configurer les logiciels nécessaires à la programmation en ASM.



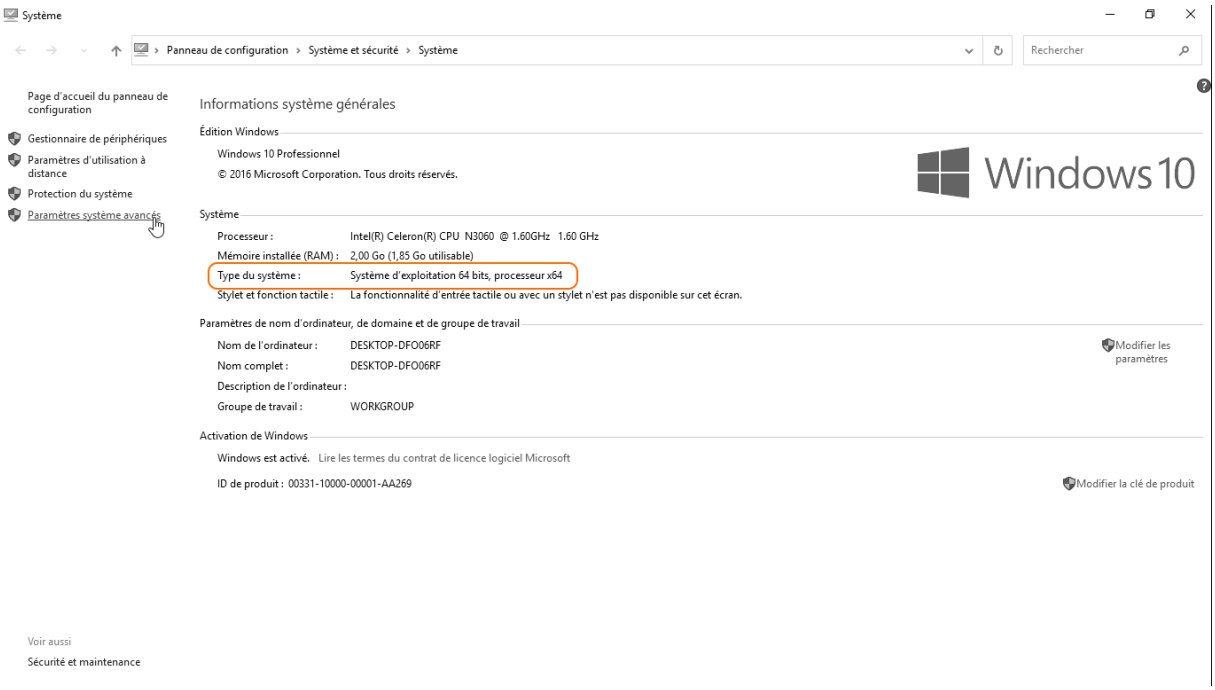
Quel est mon type de système?

Avant de télécharger les outils, nous allons d'abord nous assurer d'identifier le type de système d'exploitation installé sur votre machine. En effet, il faut distinguer ceux qui possèdent un système d'exploitation 64 bits de ceux qui n'en possèdent pas. Pour savoir dans quelle catégorie ouvrez votre explorateur de fichier et rendez-vous dans « Ce PC » ou « ordinateur » selon votre version de Windows. Ensuite cliquez-droit dans le vide. Puis, choisissez « propriétés » dans le menu. La figure suivante illustre ce procédé sous Windows 10 :



Cliquez-droit dans « Ce PC » et choisissez « propriétés »

La capture d'écran suivante montre à peu près la fenêtre qui devrait apparaître et là où vous lirez le type de votre système :

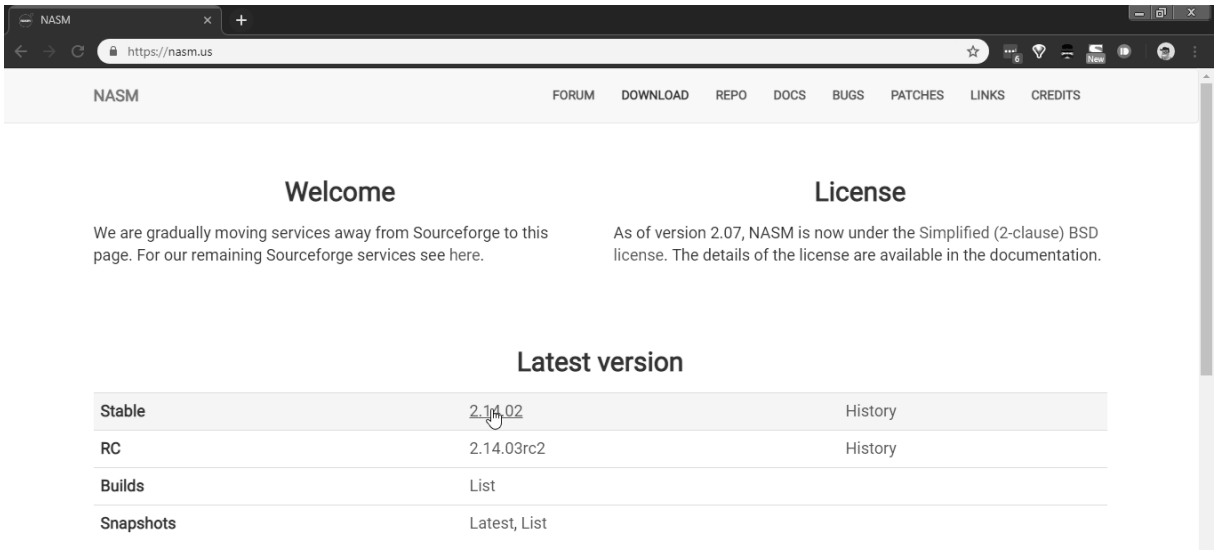


Comme vous pouvez le constater, mon système et mon microprocesseur sont de type 64 bits. Identifiez également, votre type de système pour savoir quelle version des logiciels vous devrez télécharger dans les sections suivantes.

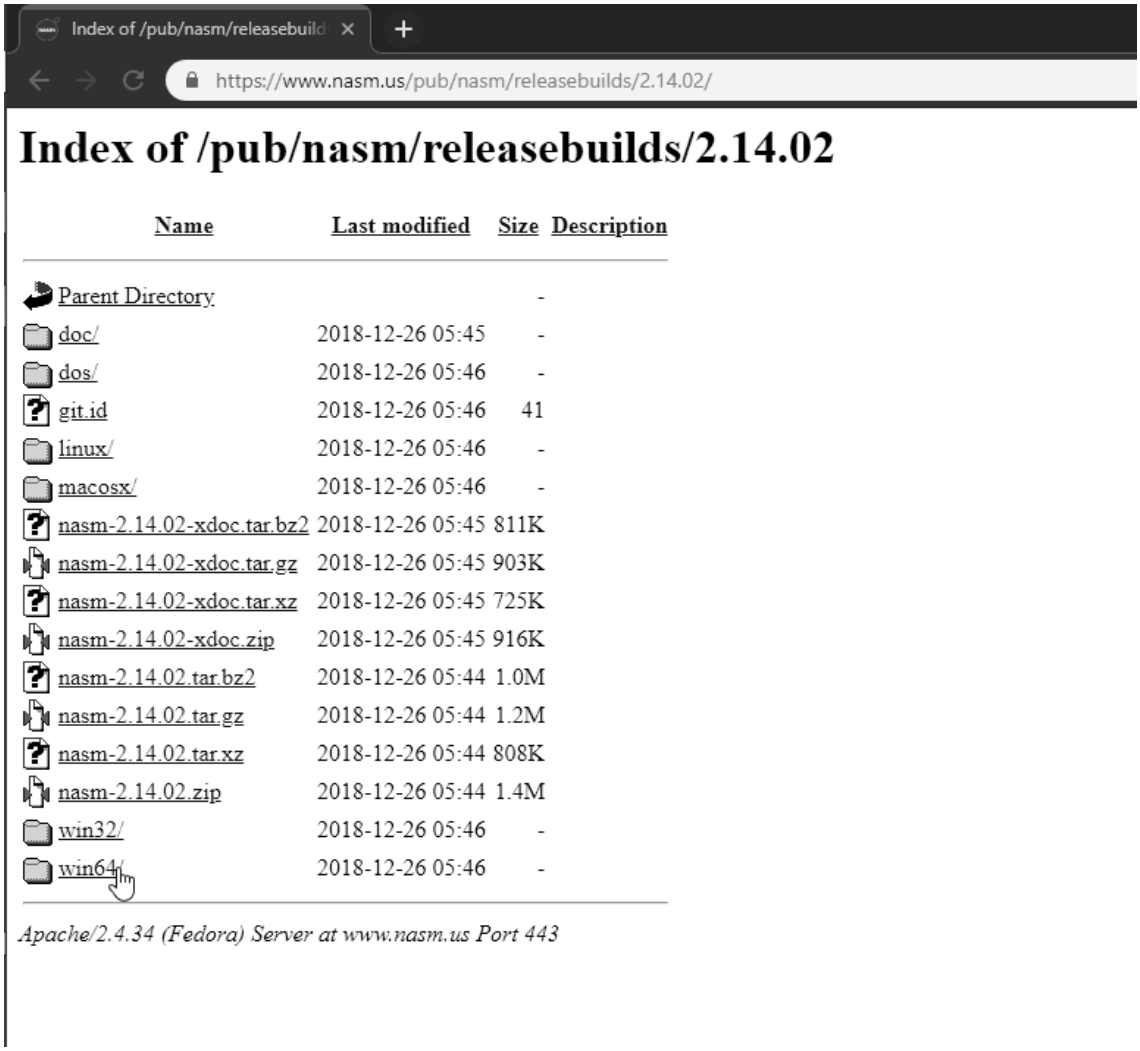
Télécharger et configurer les outils

Nous allons maintenant télécharger et installer les logiciels qu'il nous faut pour bien programmer en ASM[35]. Ensuite, nous apprendrons à nous familiariser avec ces nouveaux logiciels. Il y au moins un outil que j'ai déjà mentionné dans les chapitres précédents, vous voyez lequel ?? Alleeez ...Vous ne devinez pas ? Non ?...Toujours pas ? Bon, je me résigne à vous le dire. Il s'agit de... l'assembleur bien sûr ! Sinon qu'est-ce qui va transformer notre code-source en code machine ? Bon, cela dit, je ne vous apprends rien en disant que l'assembleur qu'on va utiliser est Nasm. Pour le télécharger, rendez-vous sur **le site officiel de Nasm**. Vous devriez avoir une page qui ressemble à peu près à ça :

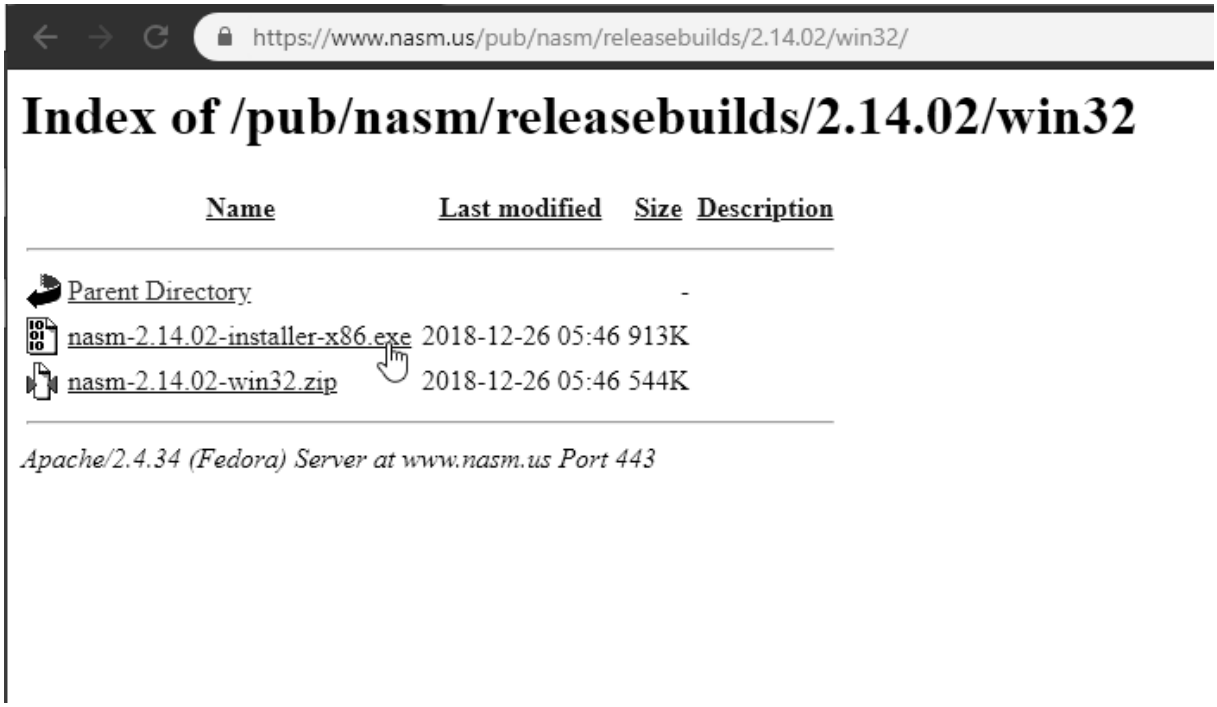
35 Concernant les outils, au lieu de les télécharger 1 à 1 on peut utiliser ce qu'on appelle un EDI (IDE en anglais) c'est-à-dire un Environnement de Développement Intégré. C'est un logiciel qui contient un éditeur, un compilateur/assembleur et un Linker. Nous n'allons pas utiliser d'EDI ici car je veux que vous vous familiarisez le plus possible avec le bas niveau en effectuant chacune des tâches qu'accompli un EDI par vous-même. Pour ceux qui sont intéressés, voici quelques EDI qu'il faudra néanmoins configurer pour qu'ils fonctionnent bien : NASM develop IDE, RadASM, SASM ou NASM IDE.



Sur la page web, vous verrez la section « Latest version ». Choisissez tout comme moi, la version stable et cliquez dessus. Au moment où j'écris ces lignes, la dernière version stable est la **2.14.02**. Une fois que vous aurez cliqué, voici ce que vous verrez :

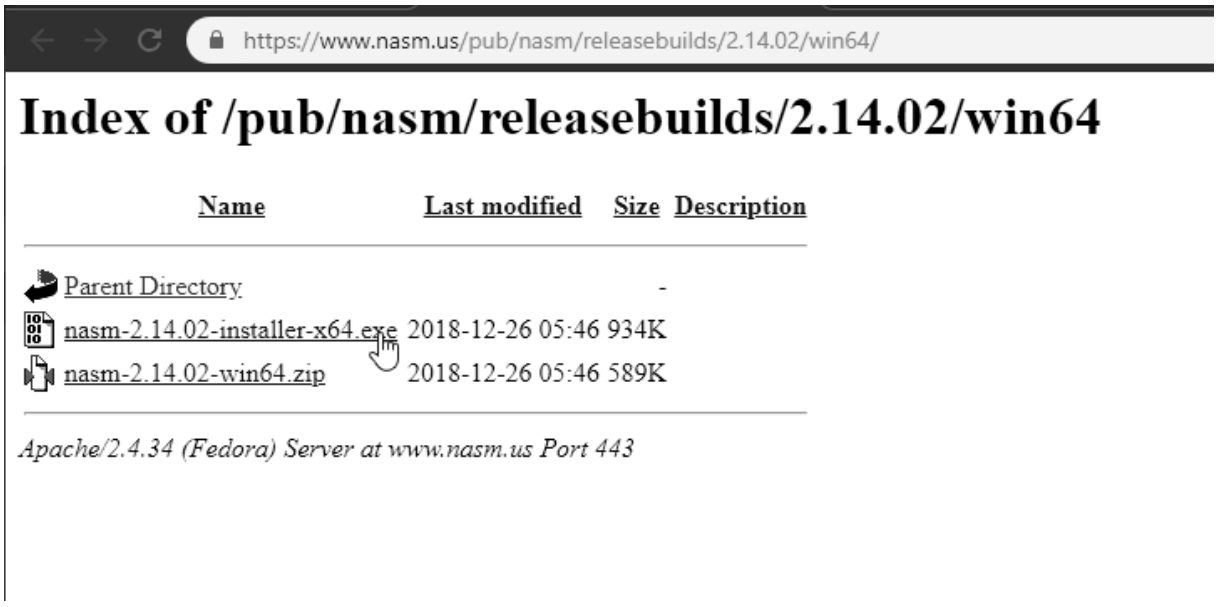


Sous Windows 32 bits
Si vous êtes sous Windows 32 bits, alors cliquez sur le dossier « `win32/` ». Vous devriez donc voir une page qui ressemble à ça :



Dans la page qui s'affiche, cliquez sur le lien avec « installer » dans le nom pour le télécharger.

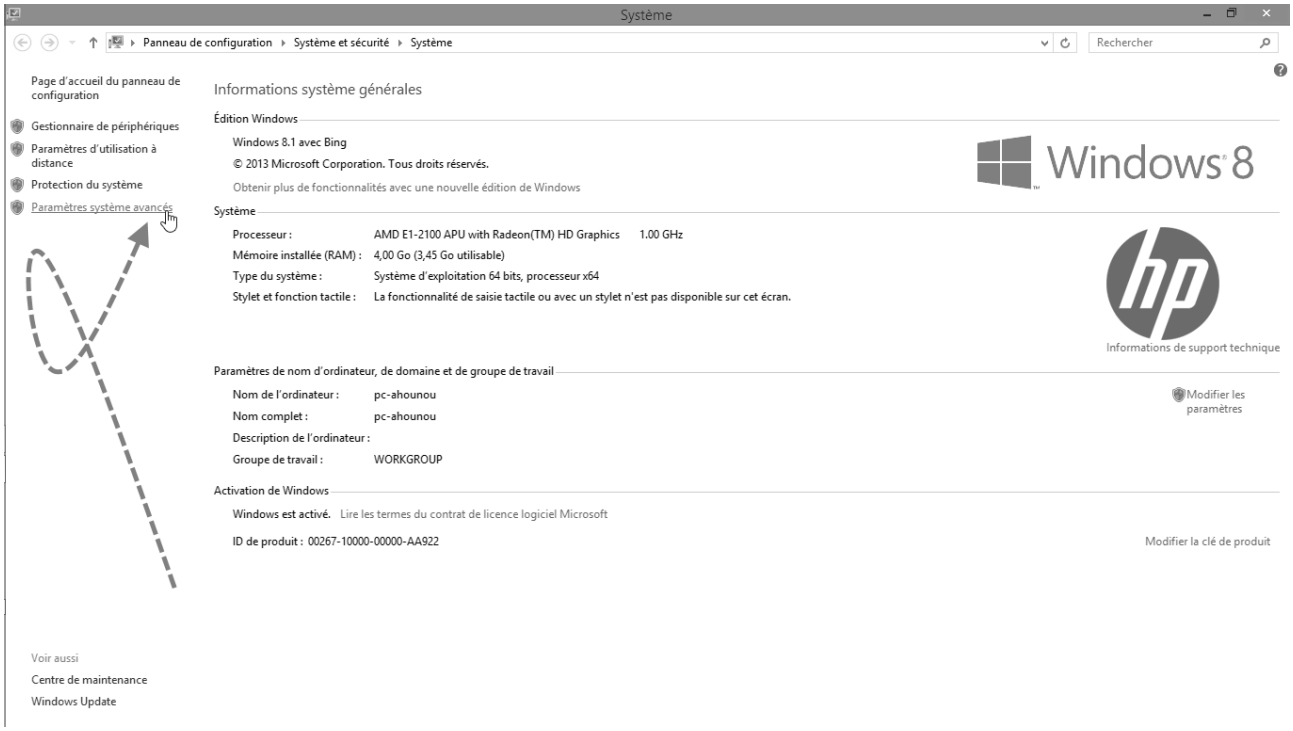
Sous Windows 64 bits
Si vous êtes plutôt sous Windows 64 bits, cliquez sur le dossier « `win64/` ». Vous devriez voir la page suivante :



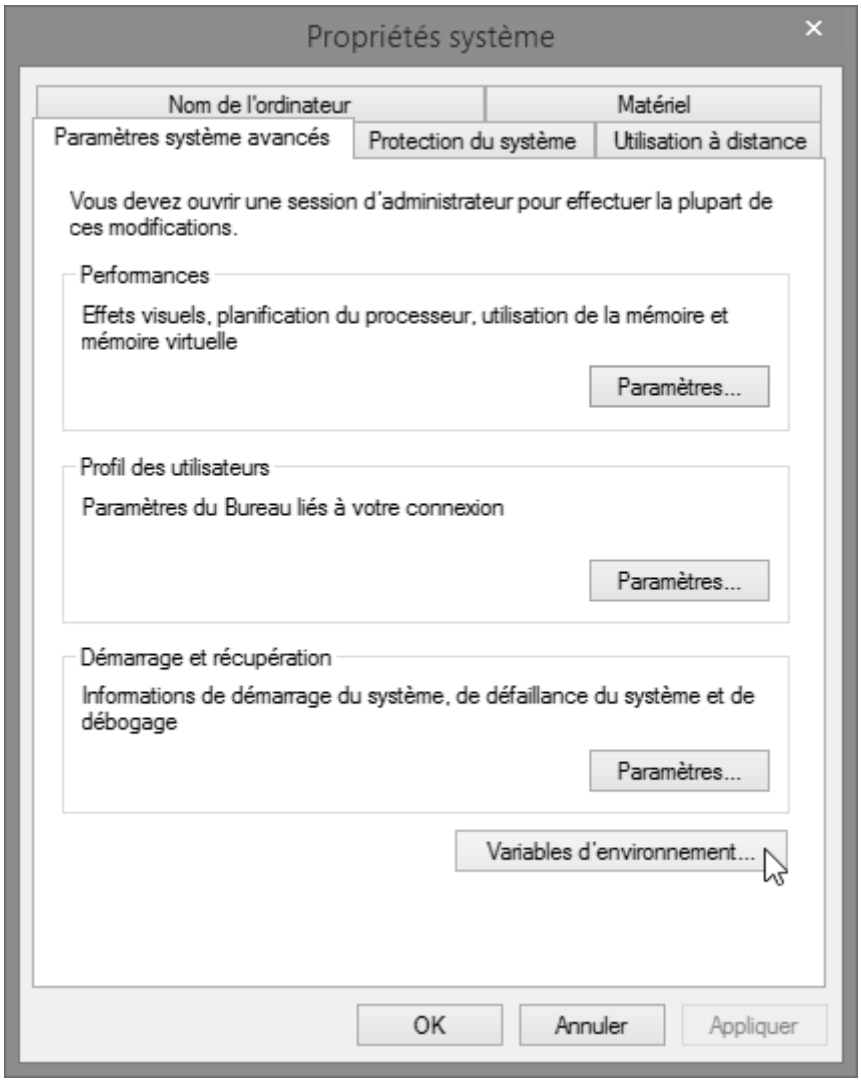
Cliquez sur le fichier avec « installer » dans le nom pour le télécharger. Une fois, NASM téléchargé, lancez l'installation en double-cliquant dessus. Installez-le dans le dossier où les programmes sont installés. Dans mon cas, mon dossier d'installation est `c:\Program Files\NASM`. Laissez les autres options par défaut. Vous avez maintenant un assembleur qui vous permettra d'assembler vos bouts de code.

Configuration de Nasm

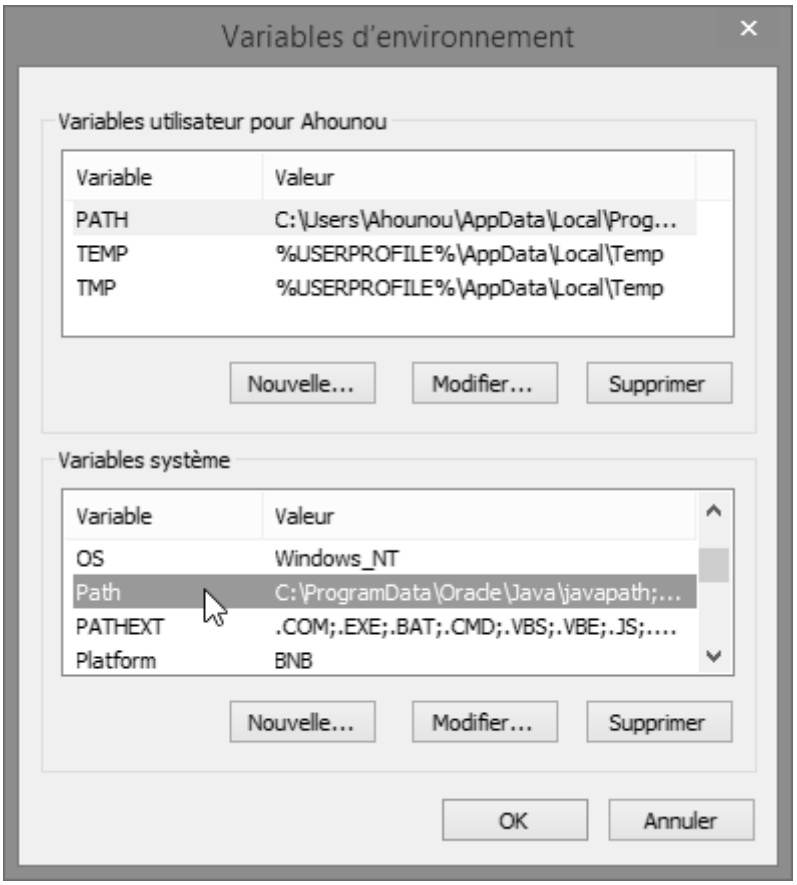
Après avoir installé Nasm, rendez-vous encore dans « Ce PC » ou « ordinateur ». Cliquez-droit dans le vide. Dans le menu qui apparait, choisissez « propriétés ». Là, une fenêtre devrait s'afficher. Regardez à gauche dans la liste, vous verrez « paramètres système avancés ». Voici un exemple sous mon Windows 8.



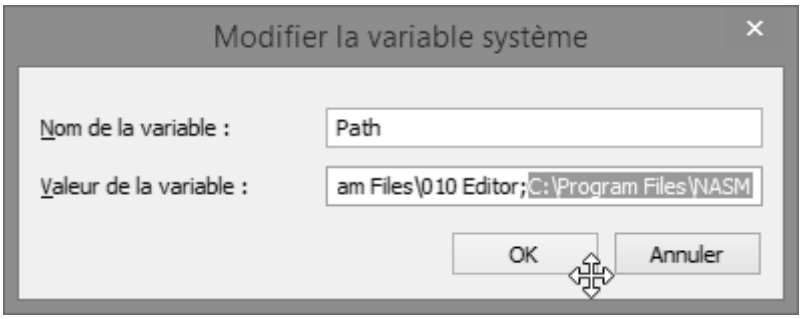
Une fois que vous avez cliqué sur cela, vous devriez avoir une fenêtre qui ressemble à ceci :



Cliquez sur le bouton « variable d'environnement ». Vous devriez voir une fenêtre qui ressemble à peu près ceci :

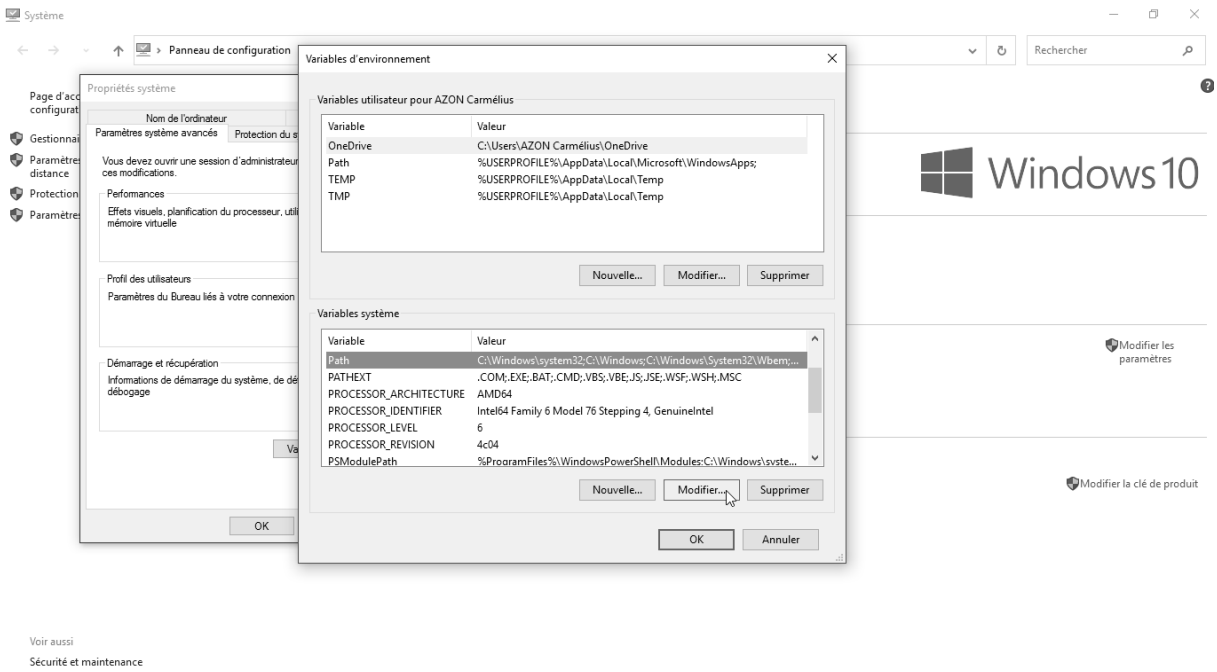


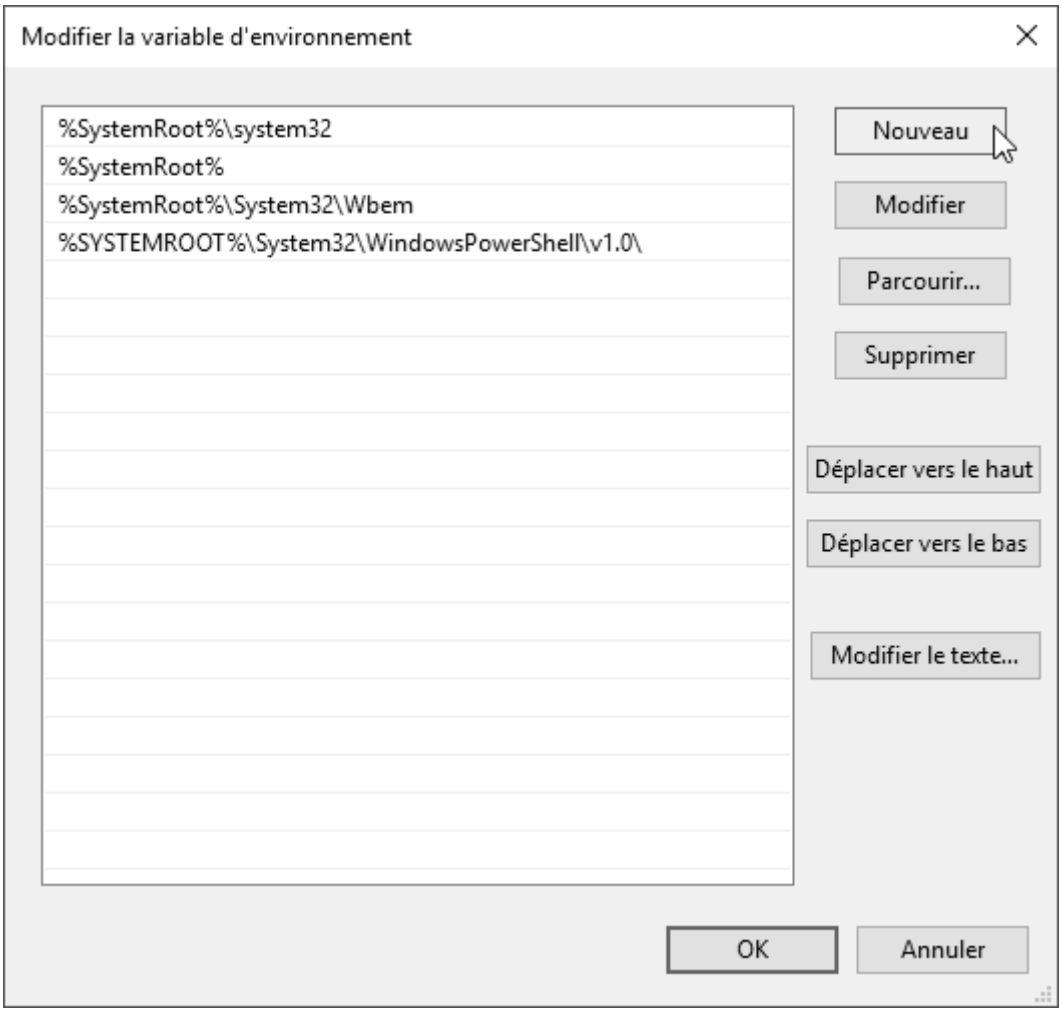
Allez dans la partie « variables systèmes » et de défilez jusqu'à trouver « path ». Cliquez ensuite sur le bouton « Modifier... ». Vous verrez la fenêtre suivante (promis c'est la dernière) :



Comme moi, ajoutez le chemin ou vous avez installé Nasm. Dans mon cas, c'est « `C:\Program Files\NASM;` ». N'oubliez pas **d'ajouter un point-virgule à la fin.** Voilà, nous avons fini de configurer Nasm[36]. **Si vous êtes sous Windows 10**, la dernière étape est différente. Il faudra cliquer sur « Nouveau » afin d'ajouter le chemin de Nasm. Voyez les captures suivantes.

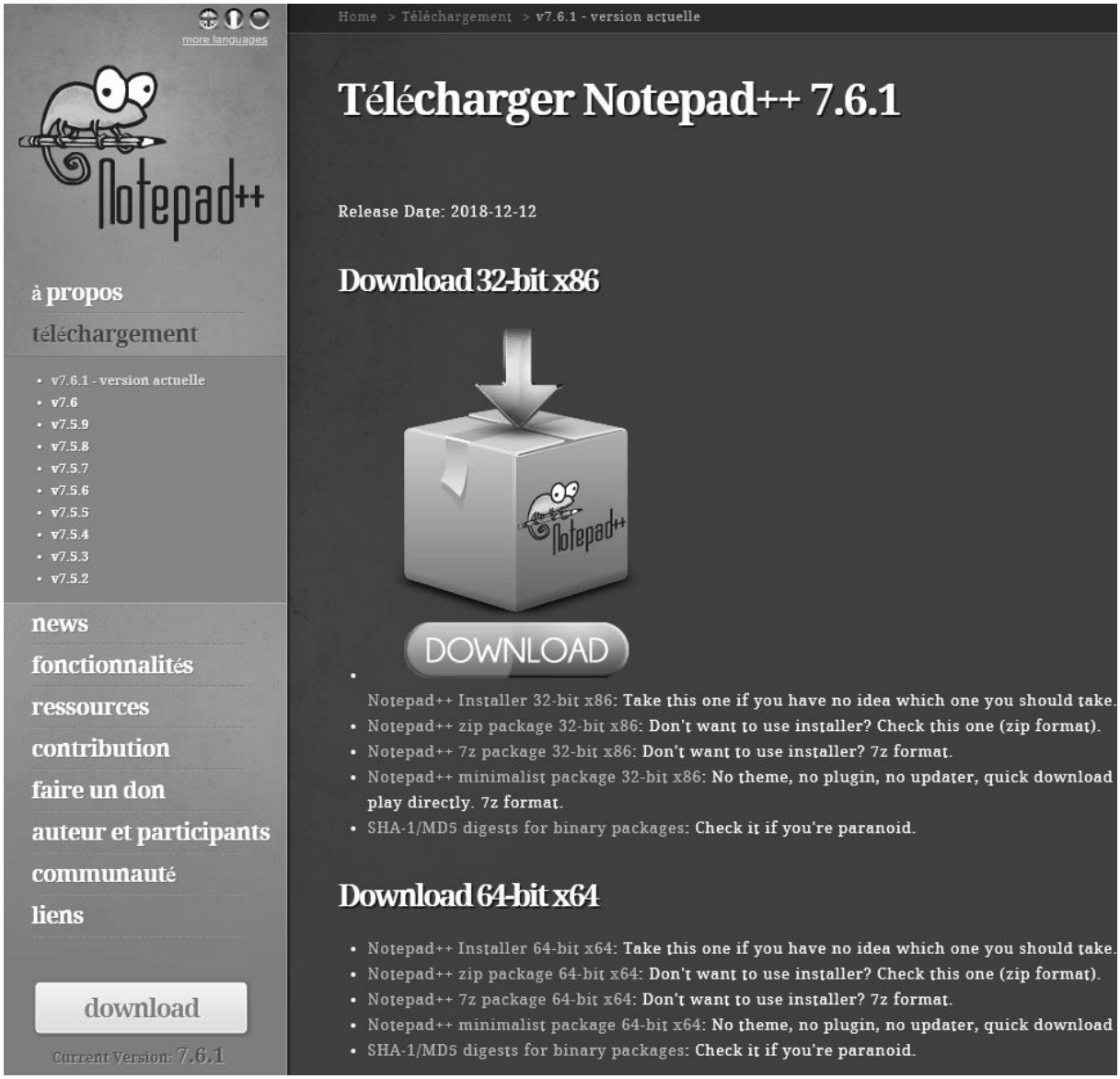
36 Pour ceux qui se demandent à quoi sert tout ce que nous venons de faire, cela nous permet d'utiliser Nasm depuis n'importe quel endroit sans devoir aller dans le dossier d'installation.


Système
Panneau de configuration
Rechercher
Windows 10
Propriétés système
Nom de l'ordinateur
Paramètres système avancés
Protection du s
Performances
Profil des utilisateurs
Paramètres du Bureau liés à votre connexion
Démarrage et récupération
OK
Variables d'environnement
Variables utilisateur pour AZON Carmélius
Variable
Valeur
OneDrive
C:\Users\AZON Carmélius\OneDrive
Path
%USERPROFILE%\AppData\Local\Microsoft\WindowsApps;
TEMP
%USERPROFILE%\AppData\Local\Temp
TMP
%USERPROFILE%\AppData\Local\Temp
Nouvelle...
Modifier...
Supprimer
Variables système
Variable
Valeur
Path
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;...
PATHEXT
.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE
AMD64
PROCESSOR_IDENTIFIER
Intel64 Family 6 Model 76 Stepping 4, GenuineIntel
PROCESSOR_LEVEL
6
PROCESSOR_REVISION
4c04
PSModulePath
%ProgramFiles%\WindowsPowerShell\Modules;C:\Windows\syste...
Nouvelle...
Modifier...
Supprimer
OK
Annuler
Modifier les paramètres
Modifier la clé de produit
Voir aussi
Sécurité et maintenance


Modifier la variable d'environnement
%SystemRoot%\system32
%SystemRoot%
%SystemRoot%\System32\Wbem
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0\
Nouveau
Modifier
Parcourir...
Supprimer
Déplacer vers le haut
Déplacer vers le bas
Modifier le texte...
OK
Annuler

Un éditeur de texte pour nos code-sources

Un éditeur de texte est un logiciel qui permet de saisir du texte brut, c'est-à-dire sans une mise en forme particulière (contrairement à MS Office Word par exemple). C'est lui qui va nous permettre de saisir notre code-source. Il existe une pléthore d'éditeurs de texte, certains sont gratuits et d'autres payants. Je vais vous présenter Notepad++ un logiciel open-source[37] et gratuit qui a plusieurs fonctionnalités intéressantes comme, par exemple, la coloration syntaxique. C'est-à-dire la possibilité de colorer les mots réservés d'un langage de programmation. Permettant ainsi de facilement reconnaître un mot simple d'un mot réservé par le langage. Ce qui réduit considérablement les erreurs dans le code-source. Pour le télécharger, rendez-vous à la page suivante : **page officielle de téléchargement de Notepad++**.

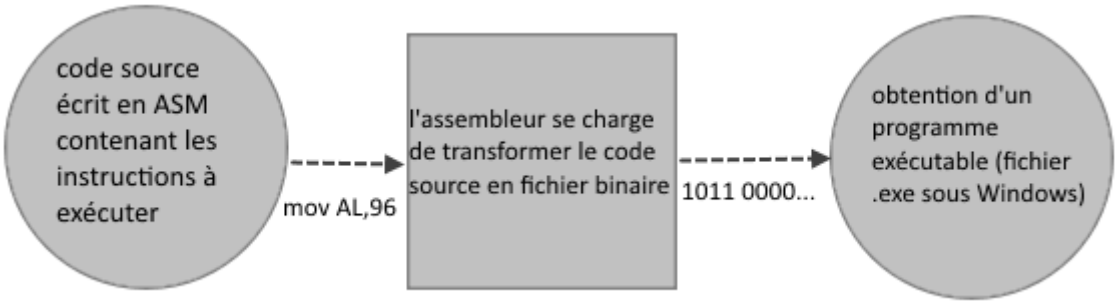


37 Pour ceux qui ignorent ce que signifie le terme open-source, cela veut simplement dit que le code-source du logiciel est « ouvert » c'est-à-dire consultable (et téléchargeable). Pour le dire en terme plus simple, vous avez le droit de connaître la recette de fabrication. C'est toute une idéologie en informatique. Il existe même des organisations comme GNU qui luttent pour cela.

Si vous êtes sous Windows 32 bits, cliquez sur le lien ***Notepad++ Installer 32-bit x86***. Si par contre, vous êtes sous Windows 64 bits, allez à section « download 64-bits x64 ». Cliquez ensuite sur le lien ***Notepad++ installer 64-bits x64***. Le téléchargement devrait débuter.

Le linker

Un Linker ou éditeur de liens est un programme spécial. Pour en comprendre l'utilité, il faudrait que je revienne un peu sur le processus d'assemblage d'un code ASM. Vous vous rappelez de ce schéma.

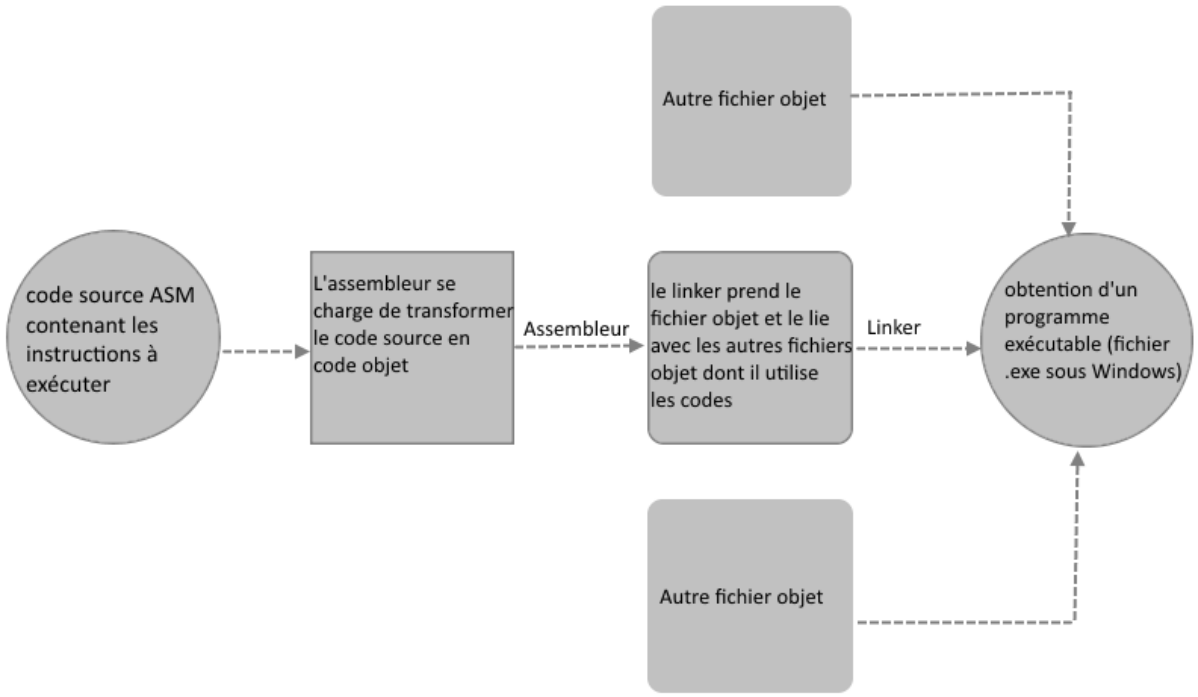


Eh bien, comme je l'avais dit, c'est un schéma **simplifié.** En réalité, il y a une étape avant la génération de l'exécutable[38]. Il s'agit de l'**édition de liens.** L'édition des liens est un processus dont la finalité est de référencer les symboles non résolues lors de l'assemblage afin de générer un fichier exécutable final.

Je me doute bien que dit comme cela, ça ressemble à du charabia. Je vais donc réexpliquer. En fait, lorsque vous créez un programme, vous aurez souvent besoin d'utiliser des bibliothèques. Les bibliothèques sont des bouts de codes tout prêts qui ont été créés par d'autres programmeurs afin de vous éviter le recodage de certaines parties dont la plupart des programmes ont besoin. Il s'agit, par exemple, l'affichage de texte, de la réception de frappes du clavier ou encore de la création de fenêtre graphique ou l'interaction avec le réseau.

En effet, imaginez que chaque fois que vous devriez faire ces tâches, vous devriez réécrire tous ces codes. Les bibliothèques sont là pour aider. Mais le problème, c'est que quand vous utilisez les bibliothèques, vous ne savez pas exactement où se situent ces bouts de codes que vous appelez. Mais, l'éditeur de liens, lui grâce au processus de **résolution d'adresses** va regarder dans les bibliothèques fournies et retrouver les bons bouts de codes. Ensuite, il va noter leurs adresses. C'est ce processus de résolution de symboles en adresses concrètes qu'on appelle l'édition de liens. Ainsi, **si vous utilisez des bouts de codes externes, il est impératif d'utiliser un éditeur de liens**. Voici un schéma qui montre l'éditeur de liens en action.

38 Si vous voulez en savoir plus sur le processus de génération d'exécutables et leur chargement en mémoire, ces trois liens vous seront utiles : l'article de Guy GRAVE (FR). Le livre gratuit de David Salomon (EN) et le livre de John R. Levine (version manuscrite gratuite - EN).

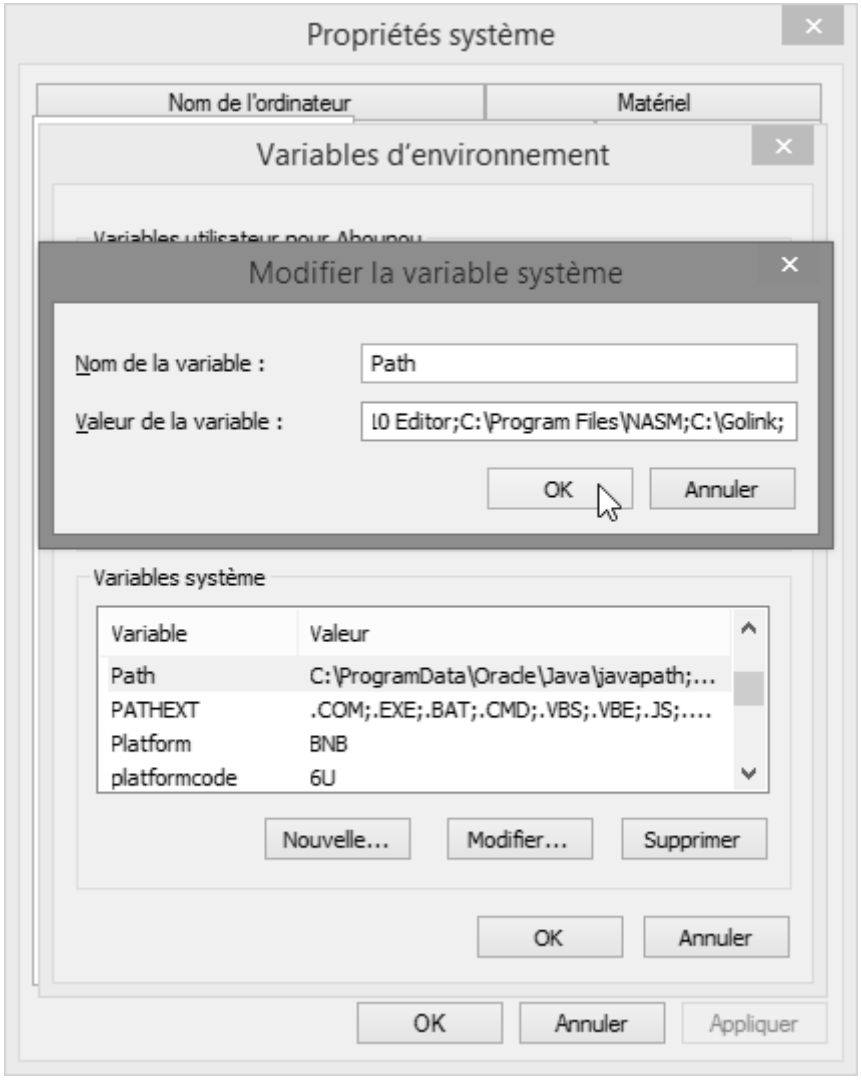


Le linker en action

Nous allons maintenant télécharger un éditeur de liens. L'éditeur de lien que j'ai choisi est GoLink. Il peut être téléchargé à l'adresse web suivante : **Télécharger GoLink**. C'est un fichier compressé. Ce qui est intéressant, c'est qu'il fonctionne autant pour un système 32 bits que 64 bits.

Une fois téléchargé, il faut extraire le contenu de l'archive dans un dossier. De préférence, extrayez-le dans le dossier `c:\golink`. Ensuite, **vous allez ajouter ce dossier à la variable d'environnement** `PATH` **juste comme nous l'avions fait pour Nasm** et pour les mêmes raisons.

Ainsi, vous suivrez les mêmes étapes. Mais, à la dernière minute, vous introduirez le chemin où vous aviez extrait GoLink. C'est-à-dire `C:\Golink` (si vous avez bien suivi mes recommandations). Voici ce que cela donne sous Windows 8.

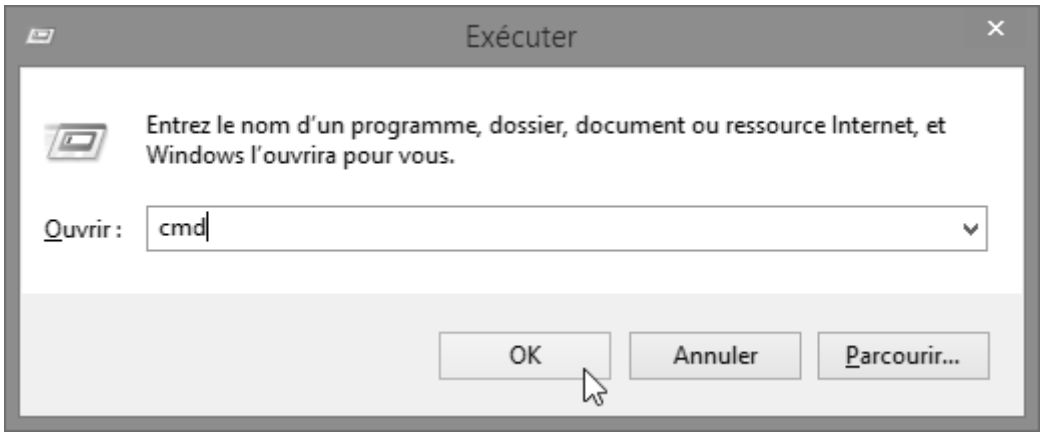


N'oubliez pas le point-virgule à la fin et cliquez sur « ok ». Voilà vous avez un linker prêt à l'emploi.

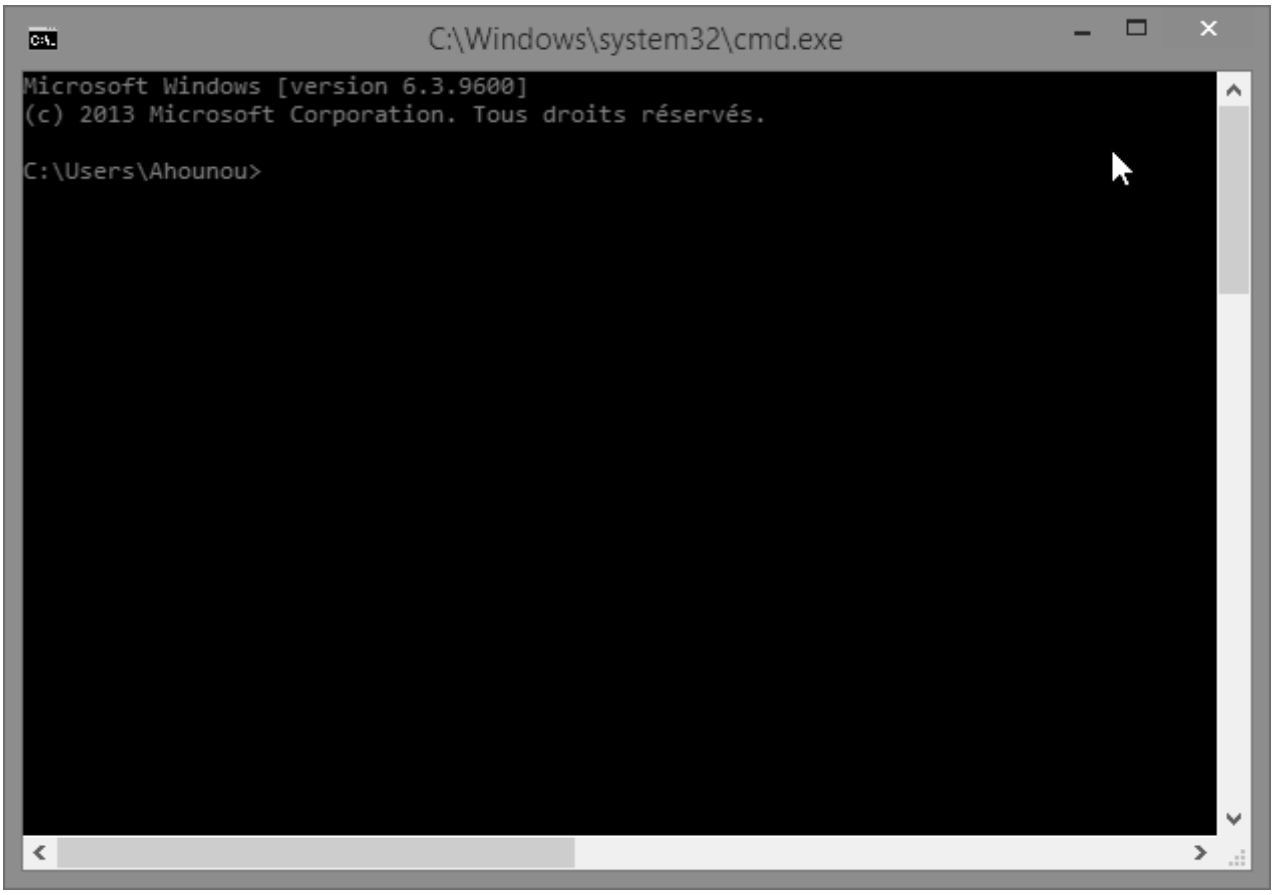
Utilisation de NASM

NASM est un programme qui n'est pas très intuitif. Il ne s'utilise pas avec une souris et n'a pour interface qu'une petite fenêtre à fond noir avec du texte écrit en blanc. Pour pouvoir l'utiliser, on doit lui envoyer des commandes grâce à notre clavier. Je vais donc vous présenter les commandes usuelles de NASM.

Maintenez la touche Windows et enfoncez la touche R. Une boite de dialogue devrait s'afficher :



Tapez à l'intérieur « cmd ». La fenêtre suivante devrait s'afficher.



Si vous avez bien configuré Nasm, les commandes suivantes devraient bien s'exécuter. Tapez donc dans la fenêtre noire la commande suivante et validez avec la touche entrée :

```
nasm -v
```

C'est une commande qui permet de savoir quelle version de Nasm vous exécutez. Voici ce que cette commande donne chez moi.

```
C:\Users\milo> nasm –v
NASM version 2.14.02 compiled on Dec 26 2018
```

Comme vous pouvez le constater, cette commande me donne la version de Nasm que j'exécute et la date à laquelle elle a été compilée. Si vous avez le même résultat (la date et la version peuvent être différentes), cela veut dire que vous aviez bien configuré Nasm. Si ce n'est pas le cas, retournez plus haut pour bien configurer Nasm.

Commandes pour assembler vos codes

Ici, nous allons voir les commandes que vous devrez utiliser afin d'assembler vos codes.

Sous Windows 64 bits

```
nasm –f win64 chemin_de_votre_code_source.asm
```

Sous Windows 32 bits

```
nasm -f win32 chemin_de_votre_code_source.asm
```

Ce sont les commandes que vous allez utiliser le plus souvent. Ces commandes méritent néanmoins quelques explications.

11) **-f** signifie format. Il est suivi du format choisi. Selon votre système, vous choisirez un format win32 (Windows 32-bit) ou win64 (Windows 64-bit) ;

12) **chemin_de_votre_code_source.asm**, n'est rien d'autre que l'endroit où vous avez enregistré votre code source suivie du nom du fichier. Par exemple, si vous aviez enregistré votre code sur votre *bureau*, et que le nom du fichier source est *nom_code_source.asm*, On aurait comme chemin, *c:\Users\[compte]\Desktop\nom_code_source.asm*.

Après l'utilisation de telles commandes, Nasm génère un fichier objet dans le dossier où vous exécutez la commande. Le fichier objet porte le même nom que le fichier source. En suivant l'exemple précédent, on aurait donc sur le bureau un fichier généré qui sera nommé *nom_code_source.obj*.

Utilisation de GoLink

GoLink est également un programme sans interface graphique. Il faut donc écrire ses commandes au clavier afin de l'utiliser. La commande que nous allons utiliser pour le moment est celle-ci :

```
GoLink /type_de_programme liste_fichiers_objets liste_bibliotheques
```

Le paramètre `/type_de_programme` permet de spécifier le type d'exécutable qu'on veut créer. Ce paramètre peut prendre les valeurs comme `/console` si on veut créer des programmes de type console ou encore `/dll` si c'est des dll qu'on veut créer. J'y reviendrai lorsque nous créerons notre premier programme.

Le paramètre `liste_fichiers_objets` n'est rien d'autre qu'une liste de chemins vers des fichiers objets. En effet, vous pouvez lier plus d'un seul fichier objet. Mais, pour l'instant, nous n'irons pas plus loin qu'un seul fichier objet.

Enfin, le dernier paramètre `liste_fichiers_bibliotheques` permet de spécifier la liste des bibliothèques ou autres fichiers objets dont les codes ont été utilisés dans notre code-source. Voici un exemple de commande à GoLink qui crée un programme console à partir d'un seul fichier objet nommé mon_prog.obj qui se trouve dans le dossier courant.

```
GoLink /console mon_prog.obj
```

Le fichier généré sera *`mon_prog.exe`*. Vous voyez que GoLink est assez facile à utiliser. Nous allons voir tout cela en action dès le prochain chapitre.

Un dossier pour nos codes-sources

Pour terminer, nous allons maintenant créer un dossier afin de rassembler nos codes-sources en un seul endroit. Rendez-vous donc dans `C:\Users\[votre compte]\Documents\` et créez un dossier et nommez-le `mes_codes`. C'est dans ce dossier que vous enregistrerez tous vos codes-sources. Voilà c'est déjà fini. Dès le prochain chapitre, nous allons créer notre premier programme.

En résumé

Nous avons vu dans ce chapitre, les outils qu'il faut pour programmer en ASM. Nous avons donc, installé et configuré l'assembleur Nasm. Et ensuite, nous avons installé l'éditeur de texte Notepad++ pour écrire nos codes-sources. Pour terminer, nous avons également installé le linker GoLink afin de lier nos fichiers objet obtenus après assemblage à d'autres fichiers objets (bibliothèques). Les bibliothèques sont des bouts de codes tout prêts à l'emploi.

QCM

1) Quel est l'outil qui nous permet d'assembler les codes-sources écrits en ASM ?

a) C'est le microprocesseur.
b) C'est le linker.
c) C'est l'assembleur.

2) Qu'est-ce qu'un éditeur de texte ?

a) C'est un programme qui permet de saisir du texte brut.
b) C'est un programme qui ne sert qu'à écrire des codes-sources.
c) C'est un programme qui permet de créer des documents de type texte et de mettre des parties en gras, en italique etc…

3) Qu'est-ce qu'un linker ?

a) C'est un programme qui lie les fichiers objets obtenus après assemblage à d'autres fichiers objets ou à des bibliothèques.
b) C'est un programme dont le rôle est de prendre un fichier source et de le transformer en exécutable.
c) C'est un programme dont le but est de lier tous les codes-sources ensemble.

4) Quelle commande permet d'assembler du code sous Windows 64 bits ?

a) `nasm -a win64 fichier_source.asm`
b) `nasm -f win64 chemin_de_votre_code_source.asm`
c) `nasm -f windows64 chemin_de_votre_code_source.asm`

5) Quelle commande permet d'assembler du code sous Windows 32 bits ?

a) `nasm -f win32 chemin_de_votre_code_source.asm`
b) `nasm -f windows32 chemin_de_votre_code_source.asm`
c) `nasm -a windows32 chemin_de_votre_code_source.asm`
d) `nasm -a win32 chemin_de_votre_code_source.asm`

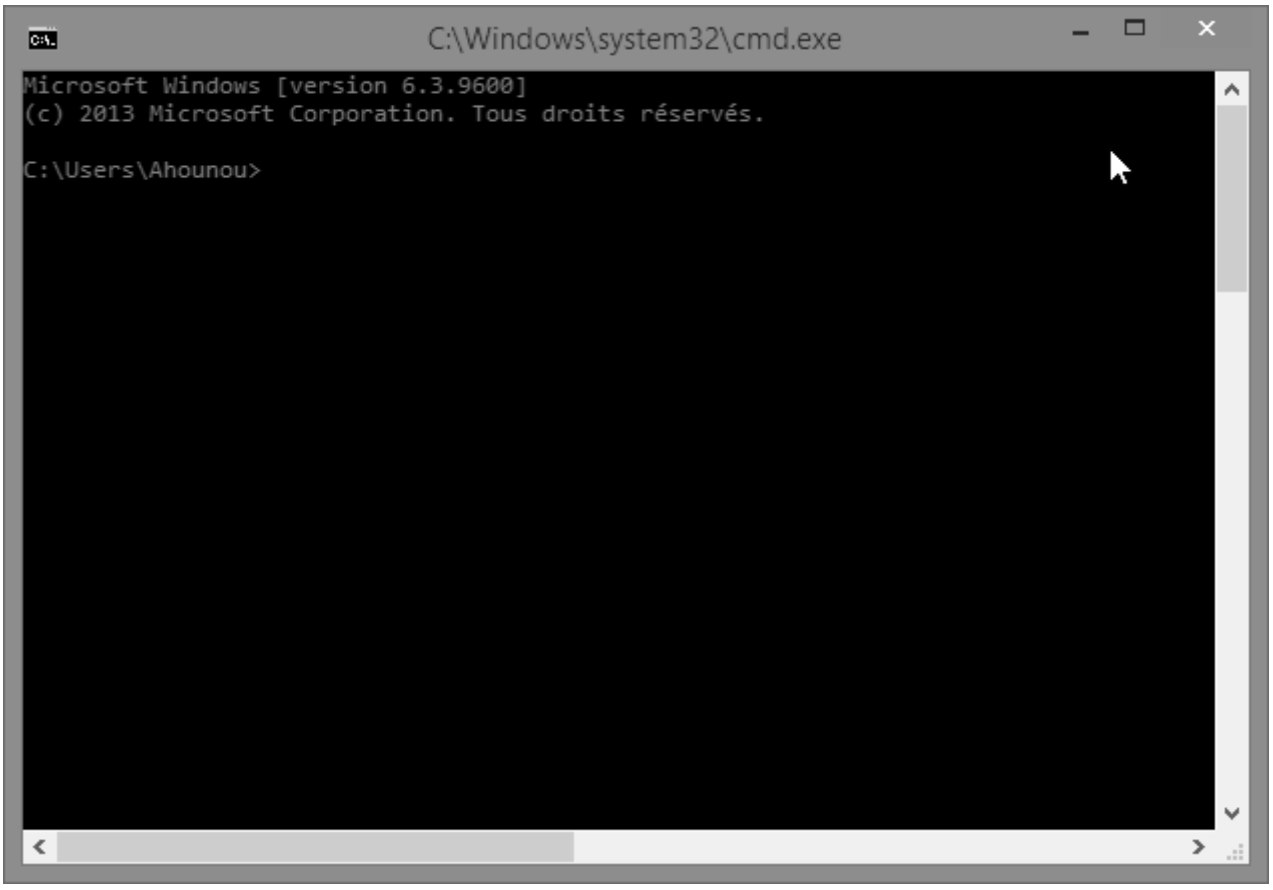
CHAPITRE 4 : VOTRE PREMIER PROGRAMME

Au cours des chapitres précédents, nous avons appris les bases du monde de l'ASM, revue en bref l'architecture des ordinateurs, installé et configuré notre environnement de travail. Il est maintenant temps de mettre toutes ces notions en pratique pour notre premier programme. Dans ce chapitre, je vous aiderai à créer votre premier programme de toutes pièces. C'est vrai que, pour cette première pratique, nous allons juste créer un programme qui déplace des données en mémoire. Mais, croyez-moi, ce sera largement suffisant pour que vous en soyez fiers.

Console ou fenêtre ?

Avant de créer notre premier programme, on va d'abord parler de la notion de programme en « console » et de programme en « fenêtre ». Surtout que les accros aux jeux vidéo ne s'excitent pas. Quand je parle de console je ne fais aucunement allusion aux jeux vidéo. Pour mieux comprendre la notion de console et de fenêtre, faisons une virée très brève dans le passé.

Il n'y a pas si longtemps, les programmes qui étaient créés n'étaient pas aussi « beaux » que ceux d'aujourd'hui. Ils étaient sous forme d'une fenêtre ayant, un fond noir avec du texte en blanc écrit dessus. Les informaticiens devaient taper des commandes, à l'aide du clavier (il n'y avait pas de souris), pour exécuter un programme. C'est cette fenêtre qu'on appelle une console. C'était bien loin de nos belles fenêtres graphiques d'aujourd'hui et de notre souris dont on ne peut plus se passer pour cliquer ici et là. L'image suivante va vous donner une idée de ce que c'est qu'une console.



Vous vous demandez sûrement quel est le rapport avec le programme que nous allons créer. Eh bien, c'est ce genre de programme que nous allons créer. Oui, je sais que la console a un aspect un peu rebutant mais, vous vous y ferrez vite. Et quand vous aurez acquis assez d'expériences, nous allons nous intéresser aux programmes avec des interfaces graphiques : les fenêtres[39]. Voilà à quoi ressemble plutôt un programme à interface graphique :

39 Ce ne sera pas dans ce volume.

Cette image vous est familière n'est-ce pas ? Mais, patience. Maîtrisez d'abord la console et ensuite, bien plus tard, on apprendra à créer des fenêtres.

Premier baptême du feu

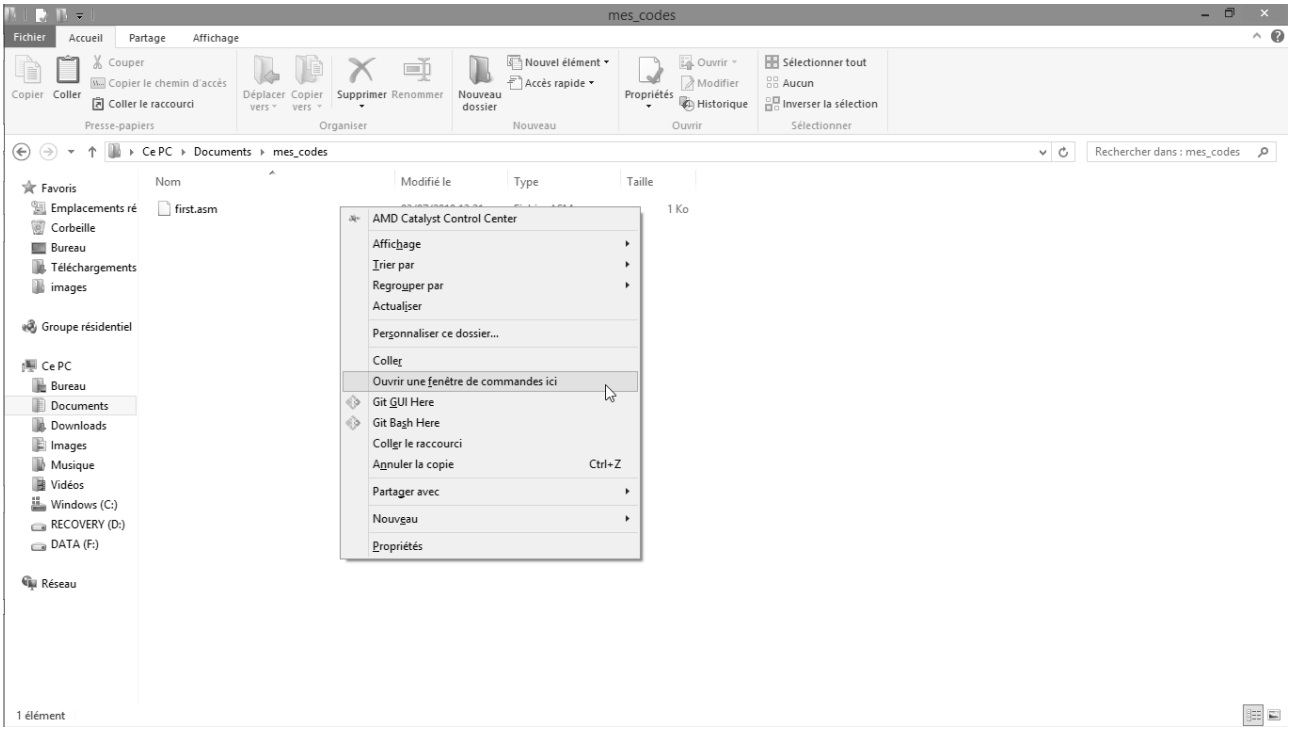
Dans presque tous les langages de programmation qu'on apprend nouvellement, le premier programme qu'on écrit est le fameux « hello world » qui veut simplement dire « bonjour tout le monde ». Nous, nous allons plutôt créer un simple programme qui déplace des données entre différents emplacements mémoire. Ouvrez donc votre éditeur de texte Notepad++. Ensuite, cliquez en haut dans le menu « langage ». Puis, allez sur « A » et choisissez « assembly » pour activer la coloration du code. Que vous soyez sous Windows 32 ou 64 bits, commencez à écrire les 4 lignes suivantes :

```
section .text
Start:
      mov EAX, 80000000
      ret
```

J'imagine que pour vous ces lignes de code n'ont absolument aucune signification et qu'essayer d'en comprendre le contenu tout seul vous donne d'affreuses migraines. Mais, ne vous inquiétez pas nous allons analyser chaque ligne et comprendre ce que ça veut réellement dire. Mais, avant d'expliquer tous ces codes, nous allons d'abord assembler le programme et voir le résultat. Pour commencer, enregistrez le fichier sous le nom `first.asm` dans le dossier `mes_codes` que je vous avais fait créer dans `Documents`.

Exécutons le code

Ouvrez le dossier `mes_codes`. Une fois dans le dossier, enfoncez la touche Majuscule (celle avec la flèche qui pointe vers le haut) et pendant qu'elle est enfoncée, cliquez-droit. Vous devriez voir un menu comme celui-ci :



Choisissez « Ouvrir une fenêtre de commandes ici » Ensuite, à l'intérieur de la fenêtre de commande, si comme moi, votre fichier s'appelle `first.asm`, tapez ceci :

Sous Windows 64bit

```
nasm -f win64 first.asm
```

Sous Windows 32bit

```
nasm -f win32 first.asm
```

Si aucune phrase n'apparaît à l'écran c'est que vous avez réussi à créer le fichier objet. Il est temps d'utiliser le linker pour créer l'exécutable final[40]. Que vous soyez **sous Windows 32 ou 64 bits**, tapez donc :

```
Golink /console first.obj
```

Vous devriez avoir à peu près les écrits suivants dans votre fenêtre de commande :

```
GoLink.Exe Version 1.0.3.0 Copyright Jeremy Gordon 2002-2018   info@goprog.com
Output file: first.exe
Format: X64   Size: 1,024 bytes
```

Cela signifie qu'un exécutable nommé first.exe a été créé au format 64 bits (si vous êtes sous Windows 32 bits, ça doit être différent). Pour voir de quoi notre programme est capable, il suffit juste de taper dans la fenêtre de commande, « first.exe » et valider en appuyant sur la touche « entrée ». Vous devriez obtenir un écran à peu près comme celui-ci :

```
C:\Users\milo\Documents\mes_codes>first.exe
```

Si tout se passe bien, … Rien ne s'affichera ! Et, c'est tout à fait normal. Car, je n'ai écrit aucune instruction qui dit d'afficher quoi que ce soit. J'ai juste utilisé une instruction qui copie une donnée dans une zone mémoires de l'ordinateur. Il faut avouer que l'ASM ne dispose pas de bibliothèques internes pour afficher des choses. Mais, pas de panique. Nous allons utiliser les fonctions du système sur lequel on programme. En l'occurrence, Windows.

Mais, euh… Pas tout de suite. En effet, le problème avec les bibliothèques du système Windows est qu'elles sont un peu particulières et demandent qu'on leur fournisse des paramètres et qu'on utilise leurs bouts de codes d'une manière bien spécifique[41]. Si je le faisais tout de suite, vous ne comprendriez rien du tout. Je vais donc d'abord approfondir votre connaissance de « l'ASM pur » avant de commencer à manipuler des bibliothèques tierces[42]. Mais ne vous inquiétez pas. Vous n'allez pas vous ennuyer car on va tout de même beaucoup nous amuser

40 Remarquez ici, qu'on n'a utilisé aucune bibliothèque (explicitement). Mais, nous avons néanmoins, besoin du linker afin qu'il crée la structure interne du fichier exécutable pour que le système le reconnaisse. Après tout, un exécutable moderne ne contient pas que du code et des données mais aussi un ensemble de métadonnées qui aident le système d'exploitation à le reconnaitre et le charger en mémoire.

41 Il s'agit de l'ensemble des conventions qu'un programme doit suivre sous un système d'exploitation donné pour être viable.

42 C'est l'un des sujets d'étude du volume 2 de ce livre. Mais, on va en toucher un bout vers la fin du présent volume.

avec les instructions de base de l'ASM. Passons donc à l'explication du code précédent.

Explication du code

Afin de mieux comprendre le code, nous allons d'abord, voir la structure d'une ligne code typique en ASM.

Les instructions : c'est un ordre, exécution !

Pour programmer, il faut commencer par écrire du code. La ligne suivante montre à quoi ressemble une ligne de code typique en ASM.

```
[label :] instruction [opérande(s)] [; commentaire]
```

Les parties mises entre crochets sont optionnelles et la partie sans crochets est obligatoire. En d'autres mots, une ligne de code en ASM est constituée de quatre champs dont trois optionnels et un obligatoire. Les champs optionnels sont ceux qu'on n'est pas obligé de fournir.

Le premier champ appelé `label` est un identificateur pour identifier une ligne de code. On n'est pas obligé de fournir un label systématiquement à chaque ligne de code ASM qu'on écrira.

Le deuxième champ est l'instruction elle-même. Étant donné qu'une instruction donnée au microprocesseur doit être au moins constituée… D'une instruction, vous comprendrez le caractère obligatoire de ce champ.

Le troisième champ est le champ d'un ou des opérandes. En effet, une instruction à souvent besoin d'opérandes. Prenons l'exemple d'une instruction d'addition. Il lui faut au moins deux opérandes : les nombres à additionner. Dans le cas où l'instruction a besoin de plusieurs opérandes, chaque opérande est séparé du prochain par une virgule. Mais, il peut arriver qu'une instruction n'ait pas besoin d'opérande(s). Par exemple, l'instruction *`RET`* n'a pas (forcément) besoin d'opérande. Puisque toutes les instructions n'ont pas besoin d'opérandes, on dira que ce champ est optionnel.

Enfin, nous avons le dernier champ : il s'agit des commentaires. Un commentaire sert à annoter nos codes-sources afin de nous rappeler plus tard, à quoi sert telle chose ou telle autre. Il commence par un point-virgule (;). Les commentaires sont invisibles à notre assembleur. Il faut dire que ça ne lui sert à rien. C'est plutôt à nous que ça sert de point de repère. Ce champ est également optionnel.

L'ossature d'un code ASM

On vient de parler d'une ligne de code en ASM. Parlons maintenant du fichier source en entier. C'est-à-dire l'ossature ou l'organisation d'un code ASM. Quand on parle d'ossature, il s'agit des parties fondamentales que tous les codes ASM

ont en commun[43]. En effet, tout code ASM est souvent constitué de trois sections ou segments spécifiques. Il s'agit de :

- La section des données initialisées nommée `.data` ;
- La section des données non initialisées nommée `.bss` ;
- La section du code nommée `.text` (ou `.code` parfois).

En fait, comme nous l'avons déjà vu, lorsque vous exécutez un programme, le microprocesseur, lui, ne voit qu'un ensemble de 0 et de 1 qu'il doit interpréter afin d'exécuter une tâche précise. Mais, vu que tout n'est que 0 et 1 dans un ordinateur, il faudrait un moyen pour que le microprocesseur distingue les données du code qui les manipule. Sinon, si le microprocesseur tombe sur les données, il essayera de les interpréter comme des instructions et cela risque de mal se passer.

Une section ou un segment est donc un regroupement logique d'informations similaires. Les codes sont regroupées ensemble dans la section `.text`, les données initialisées aussi sont regroupées ensemble dans la section `.data`, et les données non-initialisées aussi sont regroupées ensemble… Ainsi de suite. Selon le contenu de la section (du code, des données…), le système d'exploitation, lors de l'exécution du programme, renforce la protection de l'espace mémoire où elle est chargée à travers des attributs. Par exemple, dans un segment de code, le système va activer l'attribut qui dira que la portion de mémoire est en lecture seul mais est exécutable. Cela signifie que théoriquement aucun programme ne peut essayer de modifier cette portion de mémoire.

En plus, si par exemple, vous exécutez plusieurs instances d'un même programme, la partie en lecture seule permet au système de l'optimiser car il sait que cette partie ne sera pas modifiée. Il peut ainsi avoir une seule section code en mémoire et plusieurs sections data. Chacune étant liée à l'instance en cours d'exécution. Par exemple, si vous ouvrez plusieurs fenêtres de VLC, pour jouer différentes vidéos. Le système va, grâce aux sections, allouer une nouvelle place pour chaque donnée (les vidéos à jouer) à chaque nouvelle instance et laisser le code (VLC) une seule fois en mémoire évitant ainsi le gaspillage de mémoire.

Pour nous les programmeurs, les sections se matérialisent dans notre code ASM sous forme de directive qu'on donne à notre assembleur qui lui, se chargera de dire au système le type de section dont il s'agit. Une directive n'est pas une instruction destinée au microprocesseur mais juste une commande que comprend notre assembleur afin de réaliser l'opération désirée. Donc une directive n'a aucune garantie de fonctionner sur un autre assembleur. Avec Nasm, il faut utiliser la directive `section` ou `segment` suivit du nom du type de section. La directive doit être utilisée avant le début de l'écriture du contenu de la section. C'est-à-dire qu'il faut déclarer la section avant de commencer à écrire ce qui y restera. Comme dit plus haut, les sections les plus courantes

43 Toutes les sections ne sont pas toujours présentes dans un code ASM. Il peut y avoir des cas où on a besoin que d'une seule section. Par contre la section de code est toujours présente. En effet, un programme est tout au moins constitué de codes.

sont `.text`, `.data` et `.bss`[44]. En bref, de nos jours voilà ce à quoi ressemble l'ossature d'un code ASM :

```
section .data
 ; ici le programmeur déclarera ses données avec une valeur initiale

section .bss
 ; ici le programmeur déclarera des données sans valeurs initiales

section .text
 ; ici le programmeur écrira son code
```

Dans le code-source, l'ordre des sections n'a aucune importance[45] et on n'est pas obligé de déclarer systématiquement toutes les sections. D'ailleurs, dans le cas de `first.asm`, je n'utilise que la section `.text`. Une dernière chose : durant la phase d'édition de liens de votre programme, le linker peut ajouter des sections afin de ranger certaines parties de votre programme au mieux. Mais, cela n'a souvent aucune incidence sur votre façon d'écrire du code. Pour résumer donc, **une section ou segment est une unité de rangement des informations similaires**.

Maintenant que vous avez compris les sections, revenons donc à notre premier programme afin de mieux l'expliquer. Débutons donc par les deux premières lignes :

```
section .text
Start:
```

La première ligne déclare la section de code et je commence avec le code. La deuxième ligne n'est pas une instruction mais plutôt un label. Je reviendrai sur les labels mais, ici c'est un label spécial qui signale le point d'entrée de notre code. En effet, notre code possède en quelques sortes… Une porte d'entrée. C'est l'équivalent de la fonction `main()` dans certains langages de haut niveau.

Voici une analogie assez parlante. Imaginez un appartement avec plein de portes. Une porte qui tombe sur la cuisine, une autre sur les toilettes, encore une qui donne sur la chambre des enfants….Bref chaque porte donne sur différentes zones. Il faut donc une sorte de pancarte sur les portes pour permettre à un visiteur étranger de savoir d'avance où il s'engage à aller en entrant à travers telle porte ou telle autre.

---

44 BSS signifie Block Started by Symbol. L'article Wikipédia sur le sujet explique pourquoi cela.

45 En réalité, Il est normalement de la responsabilité du linker d'ordonner les sections dans le fichier exécutable finale. Mais, il peut arriver que le linker soit configuré pour accepter d'ordonner les sections selon leur ordre de déclaration dans le code-source.

Notre programme aussi possède plusieurs *possibles* points d'entrée. En effet, n'oubliez pas que notre linker va lier notre code avec d'autres codes et effectuer aussi quelques ajustements. Afin d'aider le microprocesseur (l'étranger) à trouver où débute les instructions de notre programme, il faut lui indiquer un point d'entrée. `Start:` est le nom du point d'entrée dans notre cas[46].

Passons maintenant à la prochaine ligne de notre programme.

```
    mov EAX, 80000000
```

L'instruction `MOV` est suivie de deux opérandes séparés par des virgules. L'instruction `MOV` permet de copier une donnée vers un emplacement mémoire. Donc la première ligne du code précédent, copie 80000000 dans EAX. EAX, est un espace mémoire spéciale qu'on l'on nomme registre. Les registres seront étudiés en profondeurs dès le prochain chapitre.

La dernière ligne de code de notre programme est assez simple :

```
ret
```

Il faut dire qu'elle n'est pas plus complexe qu'elle en a l'air. Cette instruction dit juste au microprocesseur d'arrêter[47] le programme en cours d'exécution.

Pour résumer donc, notre programme copie une donnée dans un registre puis s'arrête. Voilà ! Nous avons fini de décortiquer le code de notre premier programme. Je sais qu'il y a encore des parties qui sont assez floues. Mais, vous allez très vite vous habituer et mieux comprendre grâce aux chapitres qui suivront.

Commentez vos programmes !

Avant de terminer cette initiation à la programmation en ASM, je vais vous faire découvrir ce qu'on appelle les commentaires et leur utilité. Quel que soit le langage de programmation, on a la possibilité d'utiliser les commentaires. L'ASM n'y échappe pas. Les commentaires, en ASM, sont en fait des lignes de texte qui viennent après des points virgules dans le code-source.

```
; Ceci est un commentaire
```

Ces lignes de texte décrivent ce à quoi servira la ligne sur laquelle elles se trouvent ou celle qui suivra. En effet, c'est exactement ce à quoi servent les commentaires : à décrire ce à quoi sert un bout de code. J'entends déjà certains se dire que ce

46 En réalité ce label de point d'entrée est automatiquement configuré par Nasm pour que le linker le retrouve. Sinon j'aurais dû utiliser la directive `global` qui exporte le nom du point d'entrée. Mais, je ne voulais pas compliquer le code inutilement.

47 En fait, elle effectue une opération plus complexe que cela. Nous verrons cela au 11ème chapitre.

n’est pas très utile. Mais, laissez-moi vous dire que les commentaires sont très utiles et en plus ils ne coûtent rien, ils sont gratuits. Ne vous en privez donc pas. Mettez-en dès que vous sentez qu'il vous a fallu réfléchir énormément pour écrire un bout de code. Sinon des semaines ou des mois plus tard, vous aurez des difficultés à comprendre un code que vous avez vous-même écrit.

Les commentaires peuvent aussi servir de point de repère à un autre programmeur qui lit votre code-source et qui n'était pas là lorsque vous l'écriviez. D'ailleurs sur de nombreux forums, lorsque vous avez des problèmes et que votre code ne marche pas, si vous le postez sans le commenter, vous serez reçu avec des réponses qui sont, pour le moins qu'on puisse dire, très déplaisantes. Par ailleurs, pour vous obliger à prendre l'habitude de commenter vos codes, je vais être le premier à suivre cette directive en me servant des commentaires pour expliquer les futurs codes que j'écrirai.

Mais, l'excès en toutes choses nuit. Donc, trop de commentaires, peut rendre votre code illisible. Bien qu'à l'origine, les commentaires aient été créés pour rendre le code facile à comprendre, ne commentez pas à tout bout de champ. Surtout lorsque le code-source que vous avez en face fait quelque chose de vraiment évident. Commentez donc, seulement lorsque c'est utile. Voici un petit bonus pour terminer : pour automatiquement transformer une ligne en commentaire dans Notepad++ il suffit de bien positionner son curseur sur la ligne en question et d’utiliser le raccourci clavier **ctrl + Q**.

Quelques conventions

Nous allons parler ici d'une chose qui au prime abord peut sembler banale mais qui a une importance capitale : La présentation de votre code source. En effet, il ne suffit pas d'écrire un code qui marche pour être respecté dans la communauté informatique. Il faut aussi savoir le présenter. Je suis sûr que vous allez, tout comme moi, finir par adopter une manière de coder qui sera un peu comme votre signature.

Bon, c'est parti. Alors, comment présente-ton son code-source ? En réalité, cela dépend de votre goût car il y a pléthore de styles. Je vais vous présenter les règles de bases. La première règle à observer est d'indenter et d’ajouter des sauts de lignes pour aérer votre code-source.

Indenter, c’est le fait d'insérer de l'espace dans votre code-source. Une indentation se fait à l'aide de la touche tabulation ou espace. Il faut indenter un code chaque fois qu’on aborde une partie du code contenue dans la précédente. Un peu comme une arborescence. Je vous donne un exemple.
Voici un code sans indentation :

```
section .text
Start:
mov EAX, 80000000
ret
```

En voici un autre avec indentation et sauts de ligne :

```
section .text
Start:
      mov EAX, 0x80000000
      ret
```

Quelle différence ? Me direz-vous. La différence, c'est la lisibilité. Il est vrai que pour un bout de code comme celui-ci, il est difficile de remarquer l'avantage qu'apporte l'indentation. Mais, pour des codes plus importants qui feront plusieurs centaines de lignes, vous allez sûrement me remercier de vous avoir montré cette astuce. Elle vous permettra de ne pas vous perdre dans votre code. Répétez donc après moi: « à partir de maintenant, je vais toujours bien indenter mon code ».

La deuxième règle pour bien présenter son code, c'est les commentaires. Nous n'allons pas nous attarder dessus car nous en avons déjà parlé : il s'agit de commenter son code source. Si nous prenons les codes-sources ci-dessus, je suis sûr que vous êtes d'accord avec moi que sans commentaire, ils sont plus ou moins difficiles à comprendre (surtout pour les débutants). Maintenant prenons les mêmes codes-source et commentons-les :

```
section .text                ; début de la section de code
Start :                      ; point d'entrée du programme
       mov EAX, 80000000     ; on passe une valeur au registre EAX
       ret                   ; arrêt du programme et retour à la console
```

J'ai exagéré à dessein, le nombre de commentaires. En effet, trop en mettre embrouille la compréhension du code. Enfin, pour terminer, La dernière règle, c'est de donner des noms explicites à vos labels (on verra plus en profondeur les labels bientôt). En effet, vous allez peut-être trouver cela contradictoire mais sachez qu'un bon code est un code qui peut se passer de commentaires en étant le plus explicite possible. C'est-à-dire, rien qu'en le lisant, il s'explique tout seul sans qu'on ait besoin d'intenses efforts de réflexion.

Obtenir un tel code se fait à l'aide de noms explicites pour les labels, des indentions et un style de codage propre. Bon, je crois que je vais m'arrêter là pour ce qui concerne la présentation de vos codes-sources. Ces quelques règles devraient suffire pour l'instant. De toute façon, si une autre règle devrait vous permettre d'être plus productif et plus professionnel, je vous la présenterai.

En résumé

Ça y est ! Vous avez franchi le cap. Félicitations ! En effet, vous avez enfin créé votre premier programme et par la même occasion, vous avez débuté le voyage pour devenir des programmeurs hors pairs. Nous avons eu un aperçu des formats exécutables, découvert l'ossature d'un fichier source en ASM et comment se structure chaque ligne de code. Nous savons donc qu'une ligne de code ASM est constituée de quatre champs dont trois sont optionnelles et une obligatoire. Nous avons également vu que dans un fichier source en ASM, qu'il faut regrouper les informations similaires dans des sections ou segments. Les sections standard les plus courants sont la section de codes `.text`, la section de données initialisées `.data` et la section de données non-initialisées `.bss`. Nous avons également vu que les commentaires servent à annoter nos codes-sources pour nous-y retrouver plus tard. On a terminé par la manière de présenter nos code-sources. Vous pouvez donc désormais, vous considérer comme des programmeurs ASM... en herbe. Vous en voulez plus ? Suivez le guide.

QCM

1) C'est quoi un programme en mode console ?

a) C'est un programme qui fonctionne sans souris et qui répond grâce des commandes saisies depuis le clavier.
b) C'est un programme qui ne sert qu'à créer des jeux vidéo.
c) C'est un programme qui ne se contrôle qu'avec des manettes de jeu vidéo

2) Qu'est-ce qu'une section ou segment ?

a) C'est un regroupement d'informations similaires dans un programme
b) C'est un programme qui ne sert qu'à écrire des codes-sources.
c) C'est un programme qui découpe notre code en plusieurs parties

3) Quelle est la différence entre une directive et une instruction ?

a) Une directive est une instruction sans opérandes.
b) Une instruction est exécutée par le microprocesseur et peut fonctionner quel que soit l'assembleur. Une directive est propre à l'assembleur et ne procure aucune garantie de fonctionner sur un autre assembleur.
c) Une instruction est un ordre donné à l'assembleur et une directive est une vague direction sans précision au microprocesseur

4) À quoi sert la directive « section » ?

a) Elle marque le début d'une section ou segment dans le code ASM
b) Elle permet de créer un code plus propre.
c) Elle ordonne à l'assembleur de mieux optimiser le programme généré.

5) Que contient la section .text ?

a) Elle contient le code du programme.
b) Elle contient du texte que manipulera le programme

6) À quoi servent les commentaires dans un code ?

c) À annoter nos codes afin qu'on s'y retrouve plus tard
d) À expliquer à l'assembleur ce que le code est censé faire