Chapitre 4 : Bibliothèque numpy et architecture des ordinateurs

Instructions introduites dans ce cours : `numpy,ndarray,shape,zeros,id`.

1 Les tableaux de la bibliothèque numpy

1 A) Notion de bibliothèque : l'exemple de numpy

Tous les langages de programmation répandus proposent des fonctions toutes faites pour la plupart des besoins courants, écrites par les concepteurs du langage eux-mêmes, ou par des utilisateurs, puis intégrées au fil du temps dans la distribution du langage. Ces fonctions sont regroupées dans ce que l'on appelle des « bibliothèques »(encore appelées « paquets » ou « modules complémentaires »). Python n'échappe pas à la règle. Parmi les modules les plus utiles, on peut citer :

- `math` qui contient toutes les fonctions habituellement utilisées en analyse ;
- `numpy` qui fournit des outils variés pour le calcul scientifique ;
- `matplotlib.pyplot` qui permet de tracer des graphiques ;
- `random` qui calcule des nombres pseudo-aléatoires ;
- `io` qui permet de lire et écrire dans les fichiers textes ;
- `os` qui permet de manipuler les fichiers.

Nous nous intéresserons ici à `numpy`, qui est un module de calcul scientifique, permettant notamment de manipuler des tableaux multidimensionnels. Pour utiliser les fonctions d'un module, on commence par importer ce dernier, en début de programme. Par exemple, pour importer le module `numpy`, on écrit :

```
import numpy as np
```

La commande `as` permet de renommer le module `numpy`. Or pour appeler une fonction issue d'un module, on doit préfixer le nom de celle-ci par le nom du module : ici on pourra donc utiliser les fonctions de `numpy` en faisant précéder leur nom de `np`. Citons enfin la commande `dir` qui permet de connaître toutes les fonctions d'une bibliothèque.

1 B) Le type ndarray

Le type associé aux tableaux définis sous `numpy` est `ndarray`. Ces tableaux sont définis grâce à la commande `array` de la bibliothèque `numpy`, qui prend en entrée une liste dont les termes sont des listes (chacune d'entre elles codera une ligne du tableau), et renvoie un objet de type `ndarray`.

```
>>> T=np.array([[1,2],[3,4]])
>>> T
array([[1, 2],
       [3, 4]])
>>> type(T)
<class 'numpy.ndarray'>
```

Remarque 1.1.
Contrairement au cas des listes, tous les éléments d'un tableau de type `ndarray` doivent être du même type.

La commande `shape` permet de calculer la taille d'un tableau (renvoyée sous forme de tuple *(nombre de lignes, nombre de colonnes)*), et la commande `zeros` permet de définir un tableau de taille quelconque ne contenant que des zéros (la taille sera spécifiée sous la forme d'un tuple ou d'une liste).

```
>>> np.shape(T)
(2, 2)
>>> print(np.zeros((2,3)))
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Pour accéder à un élément d'un tableau (que ce soit en lecture ou en écriture), il faut indiquer entre crochets les indices de ligne et de colonne de cet élément, exactement comme pour une liste (et attention les indices de lignes et de colonnes commencent toujours à 0 !).

```
>>> T[1,1]
4
```

On peut extraire la ligne numéro i (resp. la colonne j) d'un tableau T avec la commande `T[i,:]` (`T[:,j]`). On obtient alors un tableau de type `ndarray` de dimension 1.

```
>>> T[0,:]
array([1, 2])
>>> T[:,1]
array([2, 4])
```

Enfin on peut extraire un sous-tableau d'un tableau T avec la commande `T[i1:i2,j1:j2]` : celle-ci renvoie le tableau contenant l'intersection des lignes $i_1, \dots, i_2 - 1$ et des colonnes $j_1, \dots, j_2 - 1$ de T.

```
>>> T2=np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> T2
array([[1 2 3],
       [4 5 6],
       [7 8 9]])
>>> T2[0:2,0:2]
array([[1, 2],
       [4, 5]])
```

Il faut aussi noter que les tableaux de type `ndarray` sont de taille fixe : il n'existe donc pas d'équivalent de la primitive `append` s'appliquant à ceux-ci.

Enfin, on peut créer des tableaux de dimension n, dans lesquels un terme est associé à un n-uplet $(i_1, \dots, i_n)$. De tels tableaux peuvent être définis avec la commande `np.array`, appliquée à une liste contenant des tableaux (le i-ème tableau correspondra aux termes dont le premier indice est i).

Exercice 1.2. Écrire une fonction Python `crea_tab(L,n)` qui, à partir d'une liste `L` crée un tableau à n colonnes avec les éléments de la liste à la suite. Cette fonction renverra un message lorsque le nombre d'éléments de la liste n'est pas un multiple de n. Par exemple :

```
>>> crea_tab([1,2,3,4,6,7],3)
array([[1 2 3],
       [4 6 7]])
```

1 C) Un premier exemple de parcours des termes d'un tableau

Pour parcourir les termes d'un tableau bidimensionnel, on doit effectuer deux boucles for imbriquées : l'une permettra de parcourir les lignes du tableau, et l'autre ses colonnes. Le programme ci-dessous permet par exemple de calculer la moyenne des termes d'un tableau.

```
def Moyenne(T):
    S=0
    (n,p)=np.shape(T)
    for i in range(n):
        for j in range(p):
            S=S+T[i,j]
    return(S/(n*p))
```

Exercice 1.3.
Écrire une fonction permettant de calculer le nombre de termes nuls d'un tableau.

1 D) Les tableaux 1D

On peut également définir des tableaux de type `ndarray` ne contenant qu'une seule ligne.

```
>>> T=np.array([1,2,3])
>>> T
array([1, 0, 3])
>>> T[0]
1
```

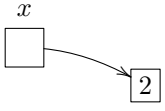
On parle alors de tableaux 1D (à une dimension), par opposition aux tableaux 2D (à deux dimensions) qui contiennent plusieurs lignes. En termes de contenus, ces tableaux sont équivalents à des listes, et on accède par exemple au terme d'indice `k` avec la commande `T[k]`. Mais attention, leur type est toujours `ndarray`, et toutes les fonctions s'appliquant aux listes ne peuvent être utilisées sur ceux-ci, comme par exemple `append`.

```
>>> type(T)
numpy.ndarray
>>> np.shape(T)
(3,)
>>> len(T)
3
>>> T.append(2)
AttributeError: 'numpy.ndarray' object has no attribute 'append'
```

2 Architecture des ordinateurs

2 A) Une seconde approche de l'affectation

Dans le premier cours, nous avions donné une approche intuitive de la notion d'affectation : « la commande `x=2` met pas la valeur `2` dans la boîte portant l'étiquette `x` ».
Une vision des choses plus proche de la réalité est la suivante : lorsque l'on entre la ligne de code `x=2`, ce que Python fait c'est créer une flèche qui relie la lettre `x` au chiffre `2`. Une telle flèche est appelée un *pointeur*, et on dira alors que « `x` et `2` sont reliés. »



En pratique, ce que `x` contiendra c'est l'adresse-mémoire associée à `2` (c'est-à-dire l'emplacement précis de la mémoire où est stockée la valeur `2`). La primitive `id` associe à une variable le numéro de la case mémoire où est stockée la valeur reliée à cette variable.

```
>>> x=2
>>> id(x)
3077481168
>>> y=2
>>> id(y)
3077481168
```

```
>>> y=10**8
>>> id(y)
2989587472
>>> x=10**8
>>> id(x)
2989587488
```

L'exemple précédent montre que l'entier 2 se trouve par défaut dans la case mémoire numérotée 3077481168, et donc lorsque l'on affecte à une variable la valeur 2, cette variable contiendra en fait l'adresse 3077481168. Le second exemple est plus surprenant : on y a affecté à deux variables la même valeur, et pourtant ces variables ne contiennent pas le même numéro de case mémoire. Cela vient du fait que nous avons utilisé un entier « trop grand », au sens où il ne fait pas partie des entiers auxquels Python affecte par défaut une case mémoire (car ils sont d'un usage moins fréquent). A chaque fois qu'un tel entier est assigné à une variable z, Python l'enregistrera dans une certaine case mémoire, et mettra le numéro de cette case dans la variable z.

Remarque 2.1.

2 B) Qu'est-ce qu'un ordinateur ?

Alan Turing (1912-1954) est un mathématicien britannique qui dès 1936 définit formellement un *dispositif abstrait* muni d'une mémoire et capable d'effectuer des opérations codées dans cette mémoire. C'est ce concept qui est à la base de tous les ordinateurs construits par la suite (on dit donc qu'un ordinateur est une *réalisation* de ce dispositif).

Si Alan Turing est le père spirituel de l'ordinateur (c'est le premier à en avoir eu l'idée), son géniteur est le mathématicien américano-hongrois John Von Neumann (1903-1957). C'est lui qui a le premier "esquissé le plan" d'une machine concrète qui serait une réalisation de la machine abstraite de Turing, en proposant en 1945 ce que l'on appelle aujourd'hui *l'architecture de Von Neumann*.

Pour finir, signalons que le terme anglais *computer* se traduit en français par *calculateur*. Mais ce terme est bien loin de rendre compte de la grande variété des opérations réalisées par les machines contemporaines : les ordinateurs ne se contentent pas de calculer. Le terme français *ordinateur* a été inventé par le philologue Jacques Perret (1906-1992) en réponse à une demande de la société IBM France, et dérive du terme latin *ordinator* qui signifie « celui qui dispose, qui règle selon son ordre. »

2 C) Machine physique et machine virtuelle

Très naïvement, on peut dire qu'un ordinateur est un dispositif fait de câbles et d'interrupteurs nécessitant une alimentation électrique. On dispose de deux états pour chaque câble (l'état 1 correspond au fait que le courant passe dans ce câble, et l'état 0 correspond au cas contraire), et grâce à cela, on peut coder de l'information : un alphabet à deux lettres est suffisant pour exprimer toutes les idées possibles. On peut également coder des suites d'opérations structurées (appelées *programmes*), puisque chaque programme peut également s'écrire dans un langage composé de deux lettres.

Les premiers ordinateurs étaient composés de beaucoup plus que des câbles et des interrupteurs : pour programmer une opération il fallait brancher et débrancher un grand nombre de câbles.

Nous ne détaillerons pas la naissance des ordinateurs modernes ici, mais disons simplement qu'arrivèrent ensuite d'une part les microprocesseurs (communément appelés *puces*), et d'autre part les langages de programmation, qui simplifièrent grandement la vie des programmeurs. Désormais, lorsque vous entrez une instruction dans une console, ou lorsque vous cliquez sur une îcone, un certain nombre de processus sont mis en œuvre afin de transformer cette action en un courant électrique qui finit par générer la réponse que vous souhaitez.

Les langages de programmation sont donc autant de langues qui permettent aux humains de communiquer efficacement et rapidement avec la machine afin d'arriver à leurs fins. Entre l'humain et la machine se situent donc un certain nombre de *logiciels* qui assurent en permanence un grand nombre de tâches, au premier rang

desquelles la traduction des instructions données par l'utilisateur en langage machine (c'est-à-dire en courant électrique), puis la traduction en langage utilisateur de la réponse de la machine. Le plus connu de ces logiciels est le *système d'exploitation* dont les rôles sont multiples comme gérer les ressources, fournir une interface simple à l'utilisateur par exemple.

On retiendra en particulier qu'il existe deux machines : la machine physique, faite de tout ce qui se trouve « sous le capot de votre ordinateur », et la machine virtuelle, qui est celle que vous voyez à l'écran et avec laquelle vous interagissez.

2 D) L'architecture matérielle d'un ordinateur

Dans la suite, nous ne considérerons que les micro-ordinateurs, c'est-à-dire les ordinateurs proposés aux particuliers dans le commerce, et nous allons passer en revue la nature et le rôle de leurs différents composants matériels (ce que les anglo-saxons appellent le *hardware*).

Les périphériques

Il faut d'abord distinguer l'ordinateur à proprement parler de ses *périphériques*. L'ordinateur est l'entité constituée par :

Les autres composants d'un ordinateur sont appelés les *périphériques*, et ils sont répartis en deux classes :

Cette structure est synthétisée par la figure suivante :

L'architecture de Von Neumann

Les deux idées principales de Von Neumann sont les suivantes :

— décomposer l'unité centrale en deux parties, en ajoutant à la partie qui effectue les calculs (l'*unité de traitement*) une *unité de commande* ou *UC*, qui donne des ordres et synchronise les opérations à effectuer ;

— doter la machine d'une mémoire centrale interne, qui permettra de stocker des données mais également des programmes.

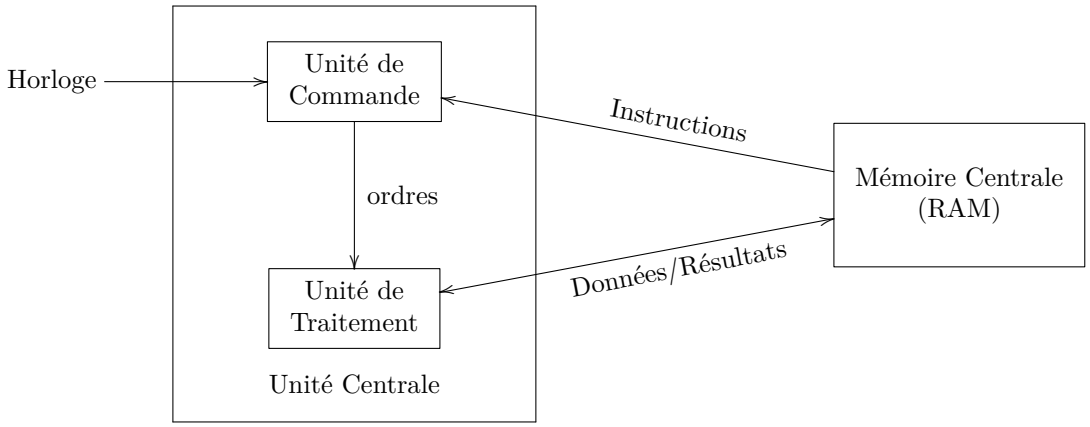


Figure 2.2 - L'architecture de Von Neumann

La mémoire centrale interne (RAM)

La mémoire vive d'un ordinateur peut être vue comme un espace découpé en une multitude de *cases*, chacune d'elle étant en fait une suite de bits dont la longueur est fixée. Afin de pouvoir s'y retrouver, on a numéroté ces cases : chacune d'elle possède ainsi une *adresse*, de sorte que lorsque l'on connaît l'adresse d'une case, on peut directement aller consulter son contenu (et éventuellement le modifier).

L'adresse d'une case est elle aussi une suite de bits, et le nombre de bits préalablement réservé au stockage de l'adresse doit bien entendu être suffisant pour pouvoir associer une adresse à chaque case. Considérons par exemple une mémoire composée de 16 cases de 2 octets [1] chacune. Chaque case a donc pour adresse un entier compris entre 1 et 16, ce qui peut être codé sur 4 bits (voir le cours n° 6 sur la représentation binaire des entiers). Cette mémoire est représentée à la figure **??**.

adresse	1er octet	2ème octet
0000		
0001		
0010		
0011		
0100		
0101		
0110		
0111		

adresse	1er octet	2ème octet
1000		
1001		
1010		
1011		
1100		
1101		
1110		
1111		

Figure 2.3 - Représentation d'un exemple de mémoire centrale interne

Ainsi, si l'on dispose d'une adresse mémoire, par exemple 0011, on peut accéder en lecture au contenu des 2 octets correspondant à cette case, et ces octets peuvent indifféremment coder :

— une donnée (par exemple un numéro de téléphone) ;

— une adresse mémoire (par exemple 0111, car c'est dans cette case que se trouve le numéro de téléphone que l'on recherche) ;

— un programme (par exemple un programme permettant de créer une base de données avec ses contacts téléphoniques).

1. Un *octet* est une suite de 8 bits ; par exemple 01001011 est un octet.

L'unité de commande

L'unité de commande est composée de deux *registres*, c'est-à-dire de deux petites mémoires vives à accès rapide (l'unité de commande n'est donc rien d'autre qu'un ensemble de bits). Ces deux registres sont :

— le *compteur ordinal* (ou CO), qui contient à chaque instant l'adresse de la case de la mémoire centrale où se trouve enregistrée l'instruction qui est en train d'être exécutée (appelée *l'instruction courante*). Sa taille coïncide donc avec la taille des adresses-mémoires (4 bits dans notre exemple).

— le *registre d'instruction* (ou RI), qui contient à chaque instant l'instruction qui est en train d'être exécutée (appelée *instruction courante*). Sa taille est donc la même que celle d'une case mémoire (2 octets dans notre exemple).

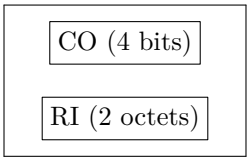


Figure 2.4 - L'unité de commande

On voit ainsi que l'unité de commande fait en quelque sorte office d' « organe de lecture à deux têtes », communément appelé *tête de lecture*.

Il faut encore comprendre une chose ici ; lorsque l'on dit que le *registre d'instruction contient une instruction* c'est évidemment un abus de langage : en réalité le registre d'instruction contient le code binaire d'une instruction (c'est-à-dire une suite de 0 et de 1 qui correspond à l'instruction).

Expliquons maintenant comment est ainsi codée une instruction élémentaire (dans cet exemple on se limite aux quatre opérations arithmétiques usuelles, $+, -, *, /$). Le code d'une instruction élementaire se décompose en quatre parties (voir la figure 2.5), chacune correspondant à $\frac{1}{2}$ octet soit 4 bits (car dans notre exemple chaque case mémoire fait 2 octets).

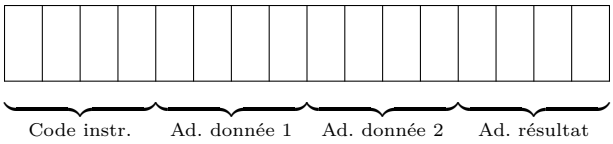


Figure 2.5 - Un exemple de codage des instructions élémentaires

Ces 4 parties ont alors la signification suivante :

L'horloge

L'unité centrale est munie d'une *horloge*, sorte de métronome électronique qui lance ce que l'on appelle communément des *tops* à intervalles de temps réguliers. Ces tops donnent la cadence à laquelle travaille l'ordinateur, et permettent aux différents composants de l'unité centrale de se synchroniser.

En pratique, plus les tops d'horloge sont rapprochés dans le temps, plus l'ordinateur est rapide. La fréquence de l'horloge est mesurée en MegaHertz (MHz). Si en 1985, les ordinateurs personnels les plus rapides étaient cadencés à 8 MHz, on trouve aujourd'hui des machines cadencées à plus de 5GHz.

L'unité de traitement

L'unité de traitement est le composant de l'ordinateur qui exécute les calculs. On peut résumer son fonctionnement par le schéma suivant :

L'UAL est ainsi la « calculatrice » de l'ordinateur. Elle est en quelque sorte « programmée physiquement », c'est-à-dire que les opérations qu'elle effectue sont « cablées », et elle ne peut donc faire que des opérations élémentaires (dans notre exemple les seules opérations arithmétiques, mais en réalité elle sait encore effectuer les opérations logiques élémentaires - comme et, ou, non, plus quelques autres - et des comparaisons entre entiers). D'un point de vue concret, on peut voir l'UAL comme un *circuit intégré* composé de différentes *portes*, tel que si l'on envoie des données et le code d'une opération à l'UAL sous la forme de courants électriques, alors celle-ci renvoie (toujours sous forme de courant) le résultat de cette opération.

Les bus

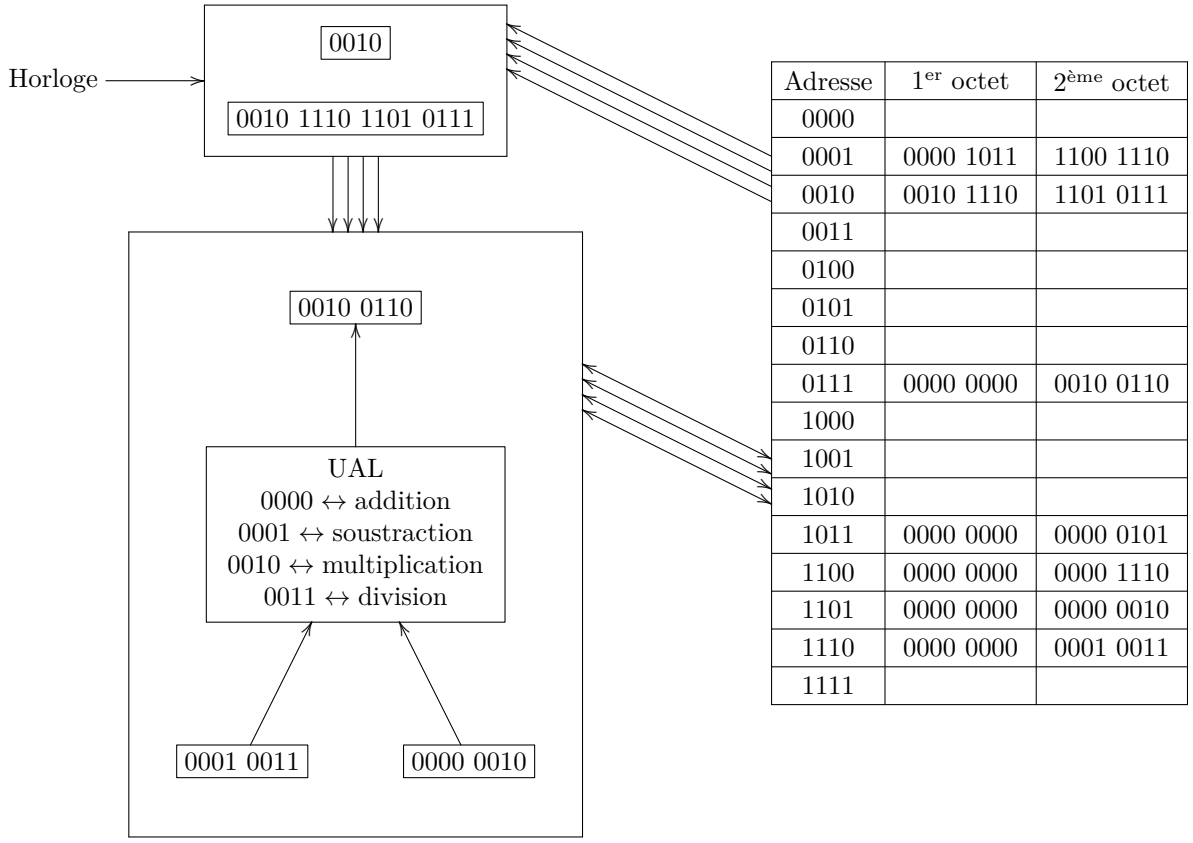
Les flèches reliant sur nos schémas les différents composants de l'ordinateur entre eux sont en fait des ensembles de fils électriques permettant de transporter simultanément plusieurs bits : pour cette raison, ils sont appelés des *bus*. Sur la figure 2.2 sont ainsi représentés trois bus, dont les noms désignent le type de données qu'ils transportent :

— le bus *ordres* sert à transmettre des demandes d'exécution d'opérations de l'unité de commande vers l'unité de traitement ;
— le bus *instructions* sert à transmettre les instructions élémentaires des cases mémoires vers le registre d'instruction de l'unité de commande ;
— le bus *données/résultats* fait circuler (dans les deux sens) le contenu des cases mémoires entre la mémoire et les registres de l'unité de traitement.

Différents bus peuvent contenir différents nombres de fils. Signalons que le nombre de fils du bus *données/résultats* détermine la capacité du microprocesseur : un *processeur 32 bits* contiendra en fait un bus *données/résultats* composé de 32 fils.

Le cycle d'exécution d'une instruction

En guise de synthèse, décrivons le cyle d'exécution d'un programme vu à travers l'architecture matérielle de l'ordinateur. Supposons donc que l'utilisateur (via le système d'exploitation) commande l'exécution d'un programme. L'adresse de la case où se trouve stockée la première instruction du programme est alors transmise au compteur ordinal (CO), puis le programme est exécuté, instruction par instruction. L'exécution de chacune de ces instructions élémentaires, codée suivant la convention donnée à la figure 2.5, se fait suivant un cycle contenant trois étapes :



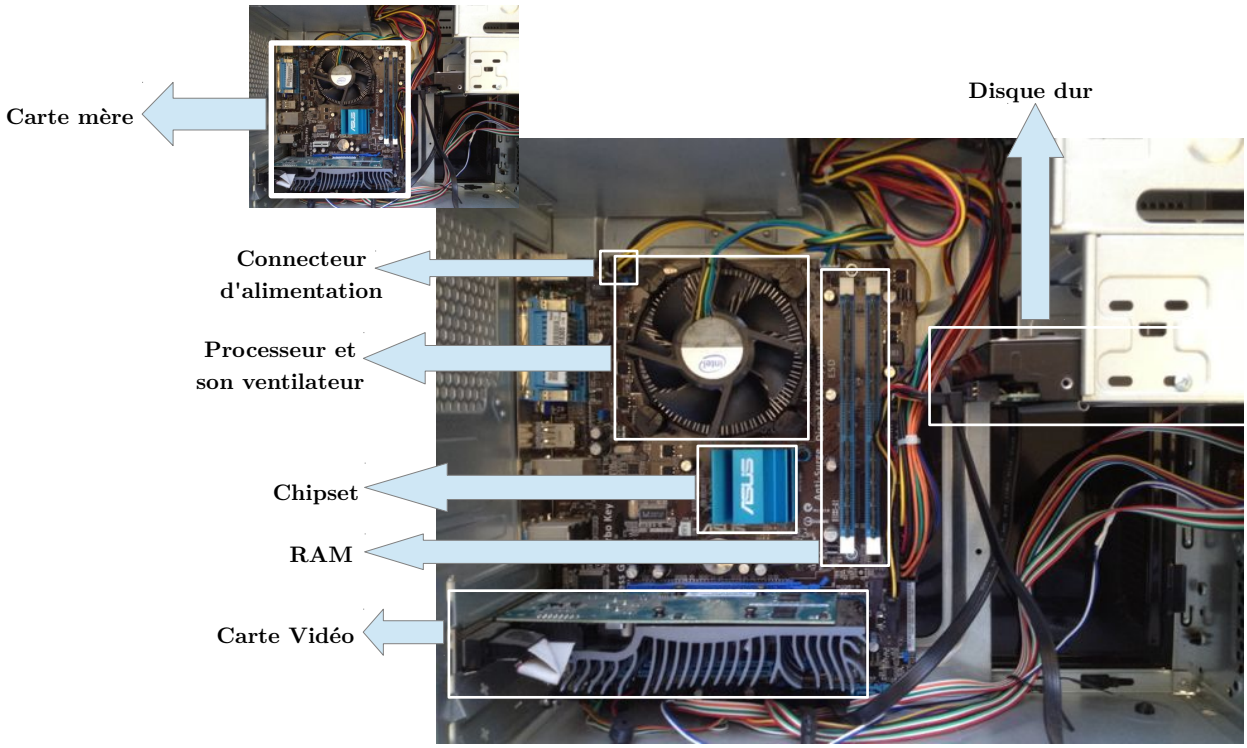
Adresse	1er octet	2ème octet
0000		
0001	0000 1011	1100 1110
0010	0010 1110	1101 0111
0011		
0100		
0101		
0110		
0111	0000 0000	0010 0110
1000		
1001		
1010		
1011	0000 0000	0000 0101
1100	0000 0000	0000 1110
1101	0000 0000	0000 0010
1110	0000 0000	0001 0011
1111		

Figure 2.6 - Schéma Bilan : fin du cycle d'exécution d'un programme

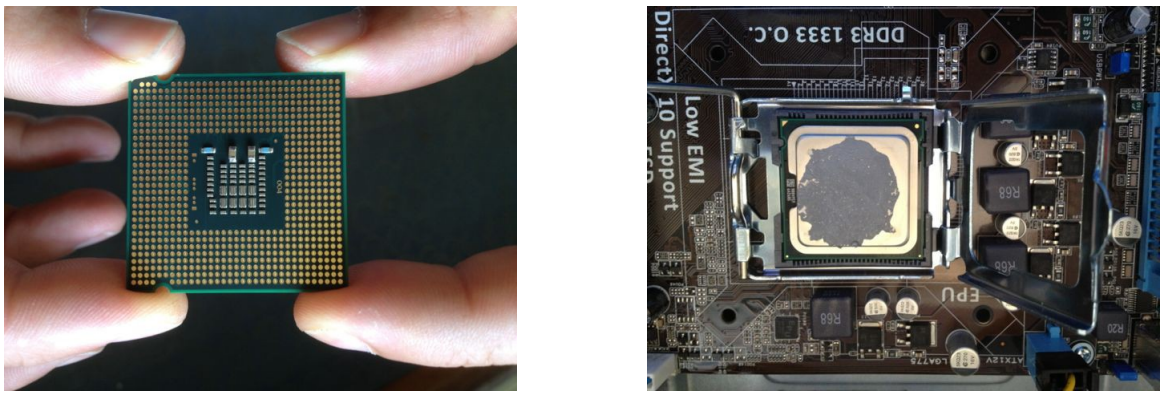
Le Schéma Bilan ci-dessus représente l'état de la machine après avoir exécuté un programme, dont les instructions sont codées dans les cases mémoires d'adresse 0001 et 0010 : l'adresse de la case 0011 vient d'arriver dans le compteur ordinal, et on suppose que cette case redonnera la main au système d'exploitation. Ce programme a consisté à effectuer le calcul $(5 + 14) * 2 = 38$, ou plus précisemment à appliquer la fonction $(x, y, z) \mapsto (x + y) * z$ aux données contenues dans les cases mémoires numéro $1011, 1100$ et 1101. On voit ainsi comment l'enchaînement de plusieurs opérations élémentaires peut aboutir à la réalisation d'une opération plus complexe.

Remarque 2.7.

3 Annexe

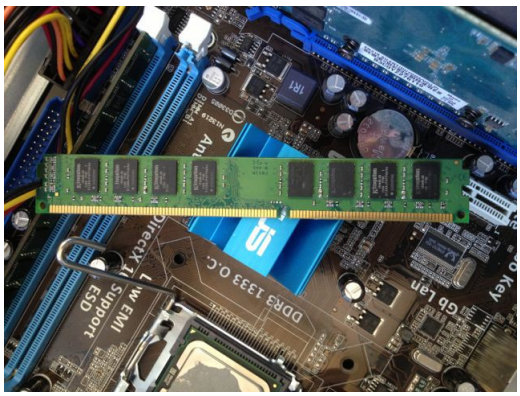


Sous le capot d'un ordinateur personnel : les principaux composants.



Un microprocesseur et son réceptacle (*socket* en anglais)

Un disque dur



Une barrette mémoire et un bus