Architecture matérielle - Modèle de von Neumann

Introduction

Le principe de fonctionnement des ordinateurs repose sur des travaux réalisés dans les années 40. Les programmes à exécuter étaient stockés au même endroit que les données qu'ils manipulaient, à savoir dans la mémoire de l'ordinateur. L'architecture de von Neumann est née de ce concept, couplé à une *unité arithmétique et logique*, une *unité de contrôle* et des *périphériques d'entrée/sortie*.

I Composants d'un ordinateur

I.1 Unité arithmétique et logique et unité de contrôle

L'unité arithmétique et logique et l'unité de contrôle sont aujourd'hui rassemblés dans un unique composant appelé *Unité contrale de traitement* ou processeur (*Control Processor Unit* en anglais).

- L'unité arithmétique et logique (*ALU* en anglais) est le circuit électronique qui réalise les opérations arithmétique et les opérations sur les bits.
- L'unité de contrôle (*Control Unit* en anglais) est le chef d'orchestre de l'ordinateur. Il s'occupe de la récupération des instructions et des données, mais aussi de l'envoie vers l'ALU.

I.1.1 Unité de contrôle

L'unité de contrôle joue donc le rôle de chef d'orchestre dans un processeur. On y retrouve trois sous-composants : deux *registres* et un *sous-programme*.

- Le *registre d'instruction* (*Instruction Register* ou IR) qui contient les différentes instructions à exécuter (addition, sauts, etc...)
- Le *pointeur d'instruction* (*Instruction Pointer ou IP*) permet d'indiquer l'emplacement de l'espace mémoire de la prochaine instruction à exécuter.
- Le *micro-programme* intégré à l'UC enregistre finalement tous les déplacements d'informations, qu'ils soient à destination de l'ALU ou des entrées/sorties.

I.1.2 Unité arithmétique et logique

L'unité arithmétique et logique va réaliser les calculs à l'intérieur du processeur grâce à des *registres de données*. Les données à exploiter sont appelées *opérandes* et les résultats sont déplacés dans le registre appelé *accumulateur*. L'ALU peut réaliser différentes opérations tel que :

- Des opérations arithmétiques,
- Des opérations logiques,
- Des comparaisons,
- Des opérations de déplacement mémoire.

La nature de l'opération à effectuer est précisé par l'unité de contrôle définie précédemment.

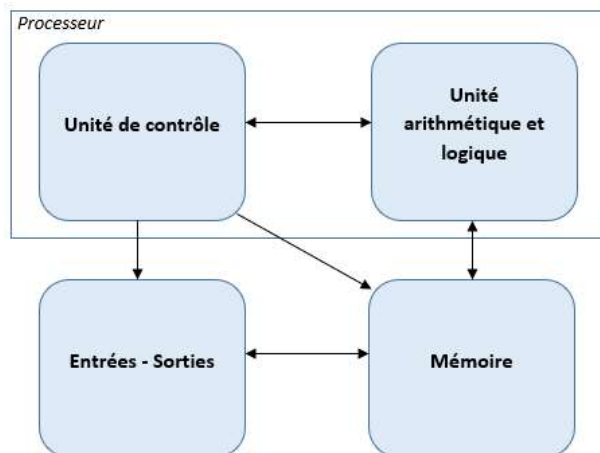
I.2 Mémoire volatile et mémoire non volatile

La mémoire de l'ordinateur contient simultanément les données et les programmes à exécuter. Il en existe deux types :

- La mémoire vive ou volatile (*Random Access Memory* ou *RAM* en anglais) stocke les informations jusqu'à l'extinction de la machine. C'est la mémoire la plus rapide, et qui permet de lire, effacer et déplacer les données.
- La mémoire non volatile s'occupe de stocker les données à conserver lorsque la machine est éteinte. Ce type de mémoire est utilisé pour stocker les données d'initialisation de l'ordinateur (on parle alors de mémoire *ROM* (pour *Read-Only Memory*) ou encore la mémoire *flash* (clé USB par exemple).

I.3 Les périphériques d'entrée/sortie

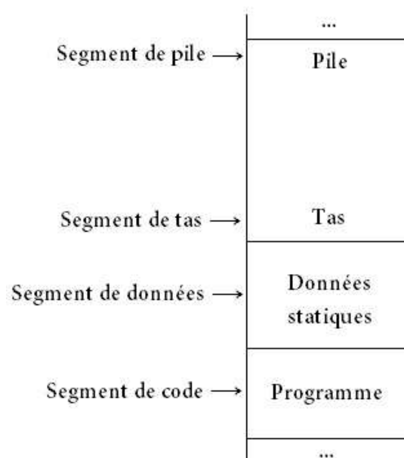
Les périphériques d'entrée/sortie nous permettent d'interagir avec l'ordinateur. Il est difficile d'en faire une liste exhaustive, mais on peut en citer quelques uns. Le clavier et la souris sont des périphériques d'entrée, tout comme une manette ou une webcam, alors qu'un écran ou une imprimante sont des périphériques de sortie. Finalement, certains jouent le double rôle comme les lecteurs CD ou les clés USB.



II Organisation de la mémoire

La mémoire d'un ordinateur est composé d'espaces mémoires élémentaires appelés *mots*. Chaque espace est sous-divisés en quatre segments, contenant diverses informations. Ces segments sont des cases mémoires, de taille allant de 8 à 64 bits, et disposant d'une adresse qui leur est propre. Les segments de l'espace mémoire jouent un rôle différent en fonction de leur tâche :

- Le segment de *code* contient les instructions.
- Le segment de *données* contiennent des valeurs fixées à l'initialisation du programme.
- Le segment de *pile* qui contient les données dynamiques, c'est à dire les données temporaires (paramètres d'une fonction, résultats d'une opération, etc.) qui ont une utilisation à court terme.
- Le segment de *tas* est similaire au segment de pile, mais contient les données qui seront utilisées tout au long de l'exécution (un tableau qui se rempli, les données d'un capteur, etc.).



Pour accéder à un mot mémoire, il suffit d'additionner l'adresse du début du segment et la position du mot en question. Cette opération permet d'atteindre n'importe quel mot mémoire.

II.1 Gestion de la pile

Le segment de pile assure la gestion des données temporaires, en les empilant les unes à la suite des autres. On y stocke, par exemple lors de l'appel d'une fonction, les paramètres, les variables et les résultats. Ces données sont supprimés de l'espace mémoire dès que l'on quitte la fonction. Prenons un exemple pour éclaircir ce mécanisme. Soit la fonction pythons définie comme ci-dessous.

```python
def f(x, y):
        z = x + y
        return z
```

Au moment de l'appel de la fonction f(1, 2), une pile est initialisée. Elle dispose d'une espace mémoire pour le résultat qui sera retourné, et un espace mémoire pour chaque variable soit x, y et z. Les espaces mémoires x et y sont remplis au moment de l'appel de la fonction f, alors que l'espace z sera rempli lors de la résolution de la ligne $z = x + y$. Le résultat z est finalement recopié dans l'espace mémoire de restitution. Lorsque l'appel de la fonction est terminé, la pile est supprimée de la mémoire. C'est le rôle du *ramasse-miettes* (ou *Garbage collector* en anglais).

Info

> Le ramasse-miettes joue le rôle de gestionnaire d'espace. Il alloue et récupère les espaces mémoires en fonction des besoins des programmes.

II.2 Gestion du tas

Contrairement à une pile, un tas permet de maintenir en mémoire les données après l'appel d'une fonction. Un tas est nécessaire par exemple lors de l'initialisation et le remplissage d'un tableau. En python, la taille d'un tableau n'est pas défini à sa création. Encore une fois, c'est le ramasse-miettes qui s'occupe de la tâche difficile de libérer de l'espace. Cette étape reste invisible pour l'utilisateur.

III Langage machine

Les programmes stockés dans la mémoire centrale d'un ordinateur sont constitués d'instructions de bas niveau, interprétables par le processeur. Ce dernier les exécute les unes à la suite des autres, en un cycle de trois étapes appelé *cycle d'instruction*. Les informations sont chargées, décodées puis exécutées.

- Le chargement permet de récupérer le(s) mot(s) binaire à une adresse spécifique.
- Le décodage assure la compréhension ou l'interprétation de l'instruction chargée en mémoire.
- L'exécution va réaliser l'opération interprétée précédement. Si le décodage a identifier une opération logique ou arithmétique, elle sera réalisée par l'ALU. Autrement, c'est une opération de branchement qui est interprétée et exécutée par l'UC.

Ces différentes opérations sont réalisés avec le langage machine *assembleur*, qui est le langage de bas niveau privilégié lors de l'exécution de tâches par un processeur. Nous allons en voir quelques exemples.

III.1 Les registres

Les processeurs disposent tous d'une mémoire cache, leur permettant de charger et d'exécuter rapidement les instructions appelées. Dans cette mémoire dédiée, on retrouve toujours plusieurs espaces mémoires spécifiques, appelés registres : *EAX*, *EBX*, *ECX*, *EDX*, E*DI*, *ESI*, *EBP*, *ESP* et des drapeaux (*flags*). Chacun de ces espaces a généralement une taille de 32 bits.

- Les registres de travail (*EAX*, *EBX*, *ECX* et *EDX*) permettent de stocker les données.
- Les registres d'adressage d'instruction (*EDI* et *ESI*) mémorisent l'opération à effectuer.
- Les registres de la pile (*EBP* et *ESP*).
- Les drapeaux permettent d'indiquer l'état actuel des activités. Ainsi, une opération arithmétique terminé signalera la fin de son opération en levant un drapeau, indiquant au processeur qu'il peut continuer.

Ces registres sont ensuite manipulés par le biais d'instructions spécifiques du langage assembleur. Ces instructions permettent de réaliser des opérations simples sur les registres. Passons en quelques-unes en revue.

III.2 Quelques instructions exécutables par un processeur

Les processeurs disposent d'un nombre limité de possibilité pour exécuter des actions. Toutes définies par le langage *assembleur*, elles permettent uniquement de réaliser des opérations élémentaires. Ces instrcutions sont exécutées les unes à la suite des autres, en parcourant la mémoire. Sans souhaitez en faire la liste exhaustive, en voici quelques unes primordiales.

III.2.1 Instruction de transfert

Cette instruction est exécutée en avec le mot clé *mov*.

```
mov eax, 42      #Copie de la valeur 42 dans eax
mov eax, ebx     #Copie du contenu de ebx dans eax
mov [x], eax     #Copie du contenu de eax dans x
```

Ces instructions sont élémentaires, et décrivent étapes par étapes les actions à réaliser. Pour que la valeur 42 soit traiter, il est nécessaire qu'elle soit stocké dans la mémoire cache du processeur, particulièrement si l'on doit réaliser une opération sur la valeur.

III.2.2 Instruction de calcul

Si l'on souhaite effectuer une opération sur une valeur, on utilise l'instruction *add*. Il est aussi possible d'utiliser l'outil de comparaison via l'appel de l'instruction *cmp*

```
mov eax, 10     #Copie de la valeur 10 dans eax
mov ebx, 20     #Copie de la valeur 20 dans ebx
add eax, ebx    #Copie de ebx dans eax
mov eax ebp     #Copie de eax dans la pile
```

Ici, les deux premières instructions permettent de charger dans la mémoire cache les valeurs à additionner. Le résultat est stocké dans le registre *eax* (et remplace donc la valeur précédente). Finalement, cette valeur sera récupéré par le programme principale via la pile. De son coté, l'instruction de comparaison vérifie la correspondance entre deux valeurs, et lève un drapeau du registre en fonction du résultat

```
mov eax, 5    #Copie de la valeur 5 dans eax
cmp eax, 5    #Comparaison de eax avec la valeur 5 engraînant le passage de zf à 1.
```

Le résultat de la comparaison est *faux*, ce qui entrainera la levé du drapeau *zf* (dans le registre mémoire, zf=1).

III.2.3 Instruction de saut

L'instruction de saut permet de se déplacer dans les adresses mémoires en spécifiant la destination voulue. On en distingue plusieurs types :

- *jmp* assure un saut vers une adresse mémoire spécifique, sans condition
- *je* et *jne* sont similaire à *jmp*, mais permettent un saut suite à une comparaison. *je* permet le saut quand *zf* vaut 1 et *jne* permet le saut quand *zf* vaut 0.
- *jg* et *jl* permettent d'effectuer un saut si la comparaison précédente à donné un résultat plus grand ou plus petit.

```
        mov eax, 16     #Copie de 16 dans eax
        mov ebx, 2      #Copie de 2 dans ebx
        add eax, ebx    #Ajout de ebx dans eax
        com eax, 19     #Comparaison de eax et de 19
        je add1         #Saut si la comparaison précédente a trouvé une égalité
        jg add2         #Saut si la comparaison précédente a trouvé que 19>eax
```

Les saut réalisés se font à destination d'un espace mémoire. Pour s'y retrouver plus facilement, on leur associe un nom unique (appelé *label*). Pour notre comparaison précédente, le saut sera effectué par l'instruction *jg* à destination du *label* add2.

IV Annexe - Gestion de la pile et du tas

Le segment de pile assure la gestion des données temporaires, en les empilant les unes à la suite des autres. On y stocke, par exemple lors de l'appel d'une fonction, les paramètres, les variables et les résultats. Ces données sont supprimés de l'espace mémoire dès que l'on quitte la fonction. Contrairement à une pile, un tas permet de maintenir en mémoire les données après l'appel d'une fonction. Un tas est nécessaire par exemple lors de l'initialisation et le remplissage d'un tableau. En python, la taille d'un tableau n'est pas défini à sa création. Encore une fois, c'est le ramasse-miettes qui s'occupe de la tâche difficile de libérer de l'espace. Cette étape reste invisible pour l'utilisateur.

IV.1 Gestion du segment de pile

Pour illustrer de fonctionnement d'une pile, prenons en exemple les deux fonctions ci-dessous décrites en python.

```python
def g(x, y):
        return 100 + y
def f(x, y):
        z = x + y
        u = g(x-1, y * 2)
```

A l'appel de la fonction f(1, 5), une première pile s'initialise comme ci-dessous. On y retrouve nos deux espaces mémoires pour x et y, ainsi que deux autres espaces :

- *reg* pour *registres* qui contient une sauvegarde de l'état du programme.
- *ret* pour *return* qui recevra la valeur à retourner.

La première illustration représente l'état de la mémoire à l'appel de la fonction *f(1, 5)*. Les variables associées à x et à y sont stockés dans les espaces mémoires. On note aussi que les espaces mémoires pour z et u ont été alloués.

reg	...
ret	
x	1
y	5
z	
u	

Sur cette deuxième illustration, on observe l'organisation de la mémoire lorsque *f(1, 5)* doit appeler la fonction *g(x, y)*. Un nouvel espace mémoire est alors créé (sur la droite de l'image), d'après les arguments passés à la fonction lors de son appel.

reg	...
ret	
x	1
y	5
z	6
u	

reg	...
ret	
x	110
y	0
z	10
u	

Finalement, la pile associé à l'appel de la fonction *g(x, y)* se termine et retourne la valeur 110 qui est stocké dans la variable u, dans la pile de *f(1, 5)*. La valeur finalement retourné est 116.

reg	...
ret	116
x	1
y	5
z	6
u	110

Exercice

Exercice 1

D'après les fonctions définies ci-dessous, décrire les différents états de la pile lorsque :

- L'appel de *g(4)*
- Lorsque *f(x)* s'apprête à renvoyer sa valeur
- Lorsque *g(4)* s'apprête à renvoyer son résultat

```
def f(x):
        z = x * x
        return z - x
def g(y):
        x = y + 1
        t = f(x)
        return x + y + x
```

Exercice

Exercice 2

Décrire le fonctionnement du programme en assembleur ci-dessous :

```
        mov eax, 0
        mov ecx, 100
ici:
        cmp ecx, 0
        je la
        add eax, ecx
        sub ecx, 1
        jmp ici
la:
```

Exercice

Exercice 3

Traduire e, langage assembleur le programme ci-dessous :

```
x = y + 42
if x == y:
        z = 1
else:
        z = 2
```