Chapitre 23

Modèle de von Neumann

Notions introduites

- architecture de von Neumann
- composants d'un ordinateur
- organisation de la mémoire
- cycle d'exécution d'une instruction machine
- langage machine

23.1 Composants d'un ordinateur

Les grands principes de fonctionnement des ordinateurs tels que nous les connaissons aujourd'hui reposent sur des travaux réalisés au milieu des années 40 par une équipe de chercheurs de l'université de Pennsylvanie. Ces travaux concernaient la conception d'un ordinateur dans lequel les programmes à exécuter étaient stockés au même endroit que les données qu'ils devaient manipuler, à savoir dans la mémoire de l'ordinateur. Cette idée d'utiliser une zone de stockage unique pour les programmes et les données est toujours utilisée aujourd'hui. Cette architecture est appelée *modèle de von Neumann*, en l'honneur du mathématicien et physicien John von Neumann qui participa à ces travaux et qui publia en 1945 un rapport sur la conception de l'EDVAC, ce nouvel ordinateur basé sur ce modèle de calcul.

Le schéma de la figure 23.1 décrit l'organisation des principaux composants d'un ordinateur selon l'architecture de von Neumann. Ce modèle comporte quatre types de composants : une unité *arithmétique et logique*, une unité de *contrôle*, la *mémoire* de l'ordinateur et les *périphériques d'entrée-sortie*.

Les deux premiers composants sont habituellement rassemblés dans un ensemble de circuits électroniques qu'on appelle *Unité Centrale de Traite-*

ment ou plus simplement *processeur* (*CPU* en anglais, pour *Control Processing Unit*). Lorsqu'un processeur rassemble ces deux unités dans un seul et même circuit, on parle alors de *microprocesseur*.

- L'unité *arithmétique et logique* (*Arithmetic Logic Unit* en anglais ou *ALU*) est un circuit électronique qui effectue des opérations arithmétiques sur les nombres entiers et flottants, et des opérations logiques sur les *bits*.

- L'unité de *contrôle* (*Control Unit* en anglais ou *CU*) joue le rôle de chef d'orchestre de l'ordinateur. C'est ce composant qui se charge de récupérer en mémoire la prochaine instruction à exécuter et les données sur lesquelles elle doit opérer, puis les envoie à l'unité arithmétique et logique.

La *mémoire* de l'ordinateur contient à la fois les programmes et les données. On distingue habituellement deux types de mémoires :

- La mémoire *vive* ou *volatile* est celle qui perd son contenu dès que l'ordinateur est éteint. Les données stockées dans la mémoire vive d'un ordinateur peuvent être lues, effacées ou déplacées comme on le souhaite. Le principal avantage de cette mémoire est la *rapidité* d'accès aux données qu'elle contient, quel que soit l'emplacement mémoire de ces données. On parle souvent de mémoire *RAM* en anglais, pour *Random-access Memory*.

- La mémoire *non volatile* est celle qui conserve ses données quand on coupe l'alimentation électrique de l'ordinateur. Il existe plusieurs types de telles mémoires. Par exemple, la *ROM*, pour *Read-only Memory* en anglais, est une mémoire non modifiable qui contient habituellement des données nécessaires au démarrage d'un ordinateur ou tout autre information dont l'ordinateur a besoin pour fonctionner. La mémoire *flash* est un autre exemple de mémoire non volatile. Contrairement à la ROM, cette mémoire est modifiable (un certain nombre de fois) et les informations qu'elle contient sont accessibles de manière uniforme. Contrairement à la RAM, ces mémoires sont souvent beaucoup plus lentes, soit pour lire les données, soit pour les modifier.

Il existe un très grand nombre de périphériques d'*entrées/sorties* pour un ordinateur. On peut tenter des les classer par familles. Tout d'abord, les *périphériques d'entrée* :

- les dispositifs de saisie comme les claviers ou les souris,
- les manettes de jeu, les lecteurs de code-barres,
- les scanners, les appareils photos, les webcams, etc.

Ensuite, les *périphériques de sortie* comme :

- les écrans et vidéo-projecteurs,
- les imprimantes,
- les haut-parleurs, etc.

Enfin, certains périphériques sont à la fois des dispositifs d'entrée et de sortie, comme par exemple :

- les lecteurs de disques (CD, Blue Ray, etc.),
- les disques durs, les clés USB ou les cartes SD,
- les cartes réseaux (modems), etc.

Les flèches du diagramme de la figure 23.1 décrivent les différentes interactions entre ces composants. Pour les comprendre, nous allons passer rapidement en revue le fonctionnement de chaque composant et ses interactions avec les autres composants de l'ordinateur.

Unité de contrôle. Cette unité est essentiellement constituée de trois sous-composants. Tout d'abord, deux *registres* (mémoires internes très rapides). Le premier est le *registre d'instruction*, dénommé `IR` (car en anglais il se nomme *Instruction Register*), qui contient l'instruction courante à décoder et exécuter. Le second registre est le *pointeur d'instruction*, dénommé `IP` (car en anglais il se nomme *Instruction Pointer*), qui indique l'emplacement mémoire de la prochaine instruction à exécuter. Le troisième sous-composant est un programme particulier, appelé *micro-programme*, qui est exécuté par le CU et qui contrôle presque tous les mouvements de données de la mémoire vers l'ALU (et réciproquement) ou les périphériques d'entrée-sortie. L'unité de contrôle est donc tout naturellement connectée à tous les autres composants de l'ordinateur (voir section 23.3 pour une description plus précise du rôle de cette unité).

Unité arithmétique et logique. Cette unité est composée de plusieurs registres, dits *registres de données*, et d'un registre spécial, appelé *accumulateur*, dans lequel vont s'effectuer tous les calculs. À ces registres s'ajoutent tout un tas de circuits électroniques pour réaliser des opérations arithmétiques (addition, soustraction, etc.), des opérations logiques (et, ou, complément à un, etc.), des comparaisons (égalité, inférieur, supérieur, etc.), des opérations sur les *bits* (décalages, rotations) ou des opérations de déplacements mémoire (copie de ou vers la mémoire). Les entrées d'une ALU sont les données sur lesquelles elle va effectuer une opération (on parle d'*opérandes*). Ces registres sont chargés avec des valeurs venant de la mémoire de l'ordinateur et c'est l'unité de contrôle qui indique quelle opération doit être effectuée. Le résultat d'un calcul (arithmétique ou logique) se trouve dans

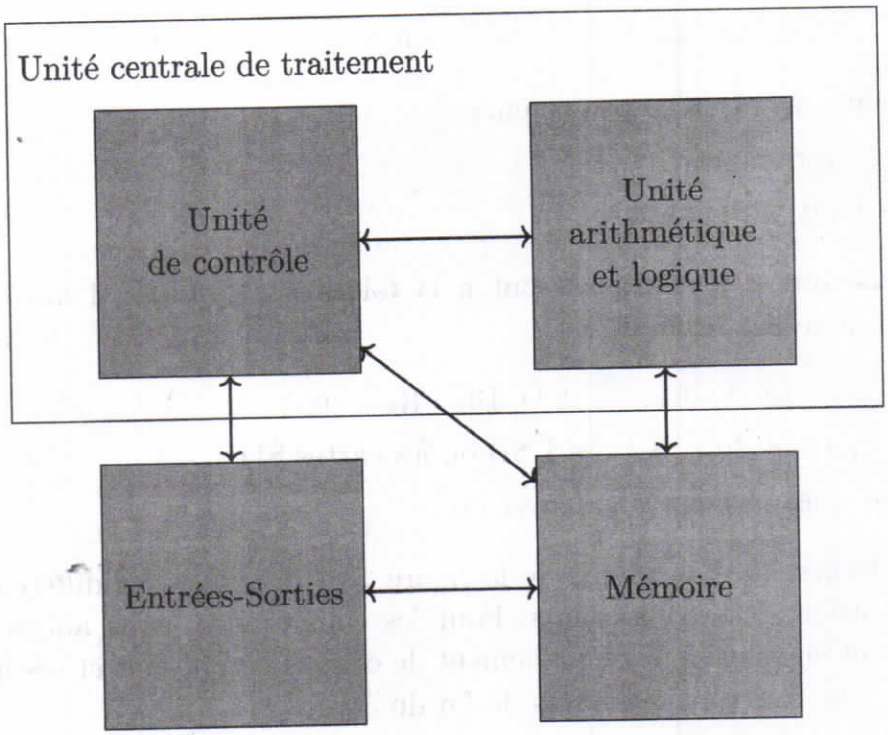

Figure 23.1 – Architecture de von Neumann.

l'accumulateur. Cependant, l'ALU peut également envoyer des signaux pour indiquer des erreurs de calcul (division par zéro, dépassement de la mémoire, etc.) ou des résultats de comparaison (inférieur, supérieur, etc.).

La mémoire. Nous avons vu les mouvements de données entre l'unité centrale de traitement et la mémoire de l'ordinateur. Ces échanges se font à travers un médium de communication appelé *bus*. Mais les périphériques d'entrée-sortie peuvent également lire et écrire directement dans la mémoire à travers ce bus, sans passer par le CPU. Cet accès direct à la mémoire est réalisé par un circuit électronique spécialisé appelé contrôleur DMA, pour *Direct Memory Access* en anglais.

Les dispositifs d'entrée-sortie. Ces composants sont connectés à l'ordinateur par des circuits électroniques appelés *ports* d'entrée-sortie sur lesquels il est possible d'envoyer ou recevoir des données. L'accès à ces ports se fait habituellement à travers des emplacements mémoires à des adresses prédéfinies. Ainsi, l'envoi ou la réception de données revient simplement à lire ou écrire dans ces emplacements réservés. Pour connaître l'état d'un périphérique, le CPU peut soit *périodiquement* lire dans ces emplacements mémoires, mais il peut aussi être directement prévenu par un périphérique d'un changement à travers un mécanisme spécial d'*interruption* prévu à cet effet. Une fois interrompu, le CPU peut aller lire le contenu des ports.

Limitation du modèle de von Neumann Ce modèle impose un va-et-vient constant entre le CPU et la mémoire, soit pour charger la prochaine instruction à exécuter, soit pour récupérer les données sur lesquelles l'instruction courante doit opérer.

Cependant, la différence de vitesse entre les microprocesseurs (nombre d'opérations par seconde) et la mémoire (temps d'accès) est telle qu'aujourd'hui, avec cette architecture, les microprocesseurs modernes passeraient tout leur temps à attendre des données venant de la mémoire, qui, bien qu'ayant aussi gagné en rapidité, reste beaucoup plus lente qu'un CPU. C'est ce qu'on appelle le *goulot d'étranglement* du modèle de von Neumann.

Pour tenter de remédier à ce problème, les fabricants d'ordinateurs ont inventé les *mémoires caches*. Ce sont des composants très rapides (mais très chers) qui s'intercalent entre la mémoire principale et le CPU. L'idée principale pour gagner du temps est qu'une donnée utilisée une fois a de grandes chances d'être utilisée plusieurs fois. Ainsi, la mémoire cache est chargée avec des données provenant de la RAM quand une instruction en a besoin, ceci afin de diminuer le temps d'accès ultérieurs à ces données.

D'autres pistes ont également été explorées. Il s'agit des architectures dites *parallèles*. Le modèle de von Neumann est également appelé *SISD* (*Single Instruction Single Data*, une seule instruction et une seule donnée), le CPU exécutant un seul flot d'instructions sur des données dans une seule mémoire.

- Le modèle *SIMD* (pour *Single Instruction Multiple Data*). Il s'agit d'une architecture avec un seul CPU où une instruction peut être appliquée en parallèle à plusieurs données, pour produire plusieurs résultats en même temps.
- Le modèle *MIMD* (pour *Multiple Instructions Multiple Data*). Il s'agit d'une architecture dotée de plusieurs CPU qui exécutent chacun un programme, de manière indépendante, sur des données différentes.

23.2 Organisation de la mémoire

Comme nous l'avons vu dans la section précédente, la mémoire d'un ordinateur (à ne pas confondre avec les périphériques de stockage comme les disques durs ou les clés USB) contient à la fois les programmes à exécuter et les données que ces programmes doivent manipuler.

En première approximation, la mémoire d'un ordinateur peut être vue comme un tableau de cases mémoires élémentaires, appelées *mot* mémoire. Selon les ordinateurs, la taille de ces mots peut varier de 8 à 64 *bits*. Chaque case possède une adresse unique à laquelle on se réfère pour accéder à son contenu (en écriture ou en lecture). Traditionnellement, ce tableau mémoire

Hiérarchie des mémoires. La vitesse d'accès aux données contenues dans la mémoire est inversement proportionnelle à la quantité de cette mémoire disponible dans un ordinateur, essentiellement parce que le prix d'une mémoire rapide est beaucoup plus élevé que celui d'une mémoire lente.
Voici une liste des différentes mémoires d'un ordinateur.

mémoire	temps d'accès	débit	capacité
registres	1 ns		≈ Kio
mémoire cache	2-3 ns		≈ Mio
mémoire vive	5-60 ns	1 - 20 Gio/s	≈ Gio
disques durs	3-20 ms	10 - 320 Mio/s	≈ Tio
DVD	140 ms	2 - 22 Mio/s	4.6 - 17 Gio

est représenté verticalement comme dans la figure 23.2, avec les premières adresses de la mémoire en bas du schéma.

Le tableau de la figure 23.2 décrit l'organisation de l'espace mémoire d'un programme *actif*, plus généralement appelé *processus*, c'est-à-dire un programme en train d'être exécuté par la machine (sachant qu'il peut y avoir plusieurs programmes actifs en même temps dans la mémoire). Cet espace est découpé en quatre parties, on dit aussi *segments* de mémoire :

- Le segment de *code* qui contient les instructions du programme.
- Le segment de *données* qui contient les données dont l'adresse en mémoire et la valeur sont connues au moment de de l'initialisation de l'espace mémoire du programme. On parle alors de données *statiques*, par opposition aux données dont l'espace mémoire est alloué *dynamiquement*, c'est-à-dire pendant l'exécution du programme. La taille du segment de données statiques est *fixe*, il n'est donc pas possible d'allouer de nouvelles cases mémoire dans cet espace à l'exécution.
- Le segment de *pile*. Ce segment, comme le suivant, contient l'espace mémoire alloué *dynamiquement* par un programme. La pile est utilisée au moment de l'appel de fonctions d'un programme pour stocker les paramètres mais également les variables locales des fonctions. La gestion en pile de ce segment mémoire facilite l'exécution de fonctions récursives[1] (où chaque appel a besoin d'un espace mémoire propre pour être exécuté) mais également la libération de la mémoire au moment où la fonction se termine.
- Le segment du *tas*. Il s'agit de la zone mémoire qui contient toutes les données allouées dynamiquement par un programme. Il peut s'agir de

1. Une fonction récursive est une fonction qui s'appelle elle-même. Cette notion sera abordée dans le programme de terminale uniquement.

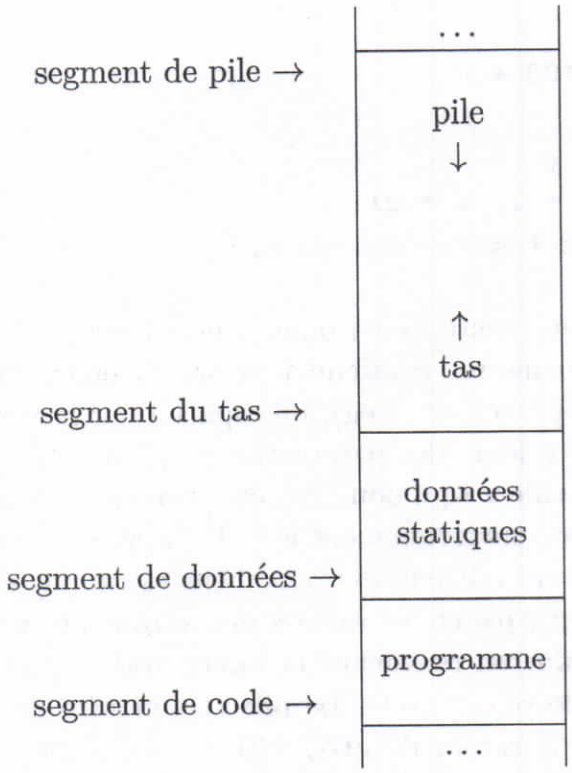

Figure 23.2 – Organisation de la mémoire.

celles dont la durée de vie n'est pas liée à l'exécution des fonctions, ou simplement celles dont le type impose qu'elles soient allouées dans cette zone mémoire, par exemple parce que leur taille peut évoluer (comme des tableaux Python).

Accès aux mots de la mémoire. L'adresse d'une case mémoire s'obtient en additionnant l'adresse du début du segment mémoire où elle se trouve (code, donnée, tas ou pile), qu'on appelle *base*, et la position de ce mot dans le segment, qu'on appelle *déplacement*. Ce mode d'adressage *relatif* à une base permet à un programme d'être exécuté dans n'importe quelle partie de la mémoire d'un ordinateur, sans changer ses instructions d'accès en lecture ou écriture.

Gestion du segment de pile. Le segment de pile permet de gérer facilement l'espace mémoire des fonctions, en empilant successivement leur contexte d'exécution (par exemple pour des fonctions récursives ou des fonctions imbriquées) et en libérant *rapidement* et *automatiquement* cet espace lorsque les fonctions se terminent.

Par exemple, considérons les deux fonctions suivantes en Python et observons la gestion de la pile décrite en figure 23.3 lors de l'appel `f(1, 5)`.

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

La pile (A) représente l'environnement lors de l'appel à `f(1, 5)`. Les premières cases mémoires contiennent respectivement une sauvegarde des registres du microprocesseur (reg), un espace pour la valeur de retour de la fonction (ret), les valeurs des paramètres `x` (1) et `y` (5), ainsi qu'un espace pour la variable locale `z` (qui pour le moment ne contient aucune valeur). La pile (B) représente l'environnement lors de l'appel `g(x - 1, y * 2)`. Pour réaliser cet appel, un nouvel environnement est alloué sur la pile avec une sauvegarde des registres et les valeurs des arguments `x` (0) et `y` (10) de `g`. Cet appel se termine en renvoyant la valeur `100 + y` (110) qui est stockée dans son espace de retour (ret). La pile (C) montre enfin l'environnement de `f(1, 5)` après le retour de `g(0, 10)`. On voit que l'espace alloué pour l'appel à `g` a été supprimé de la pile, que la valeur de retour (110) a été récupérée et stockée dans la case mémoire de la variable locale `u` et que la somme `z + u` (116) est stockée dans l'espace de retour.

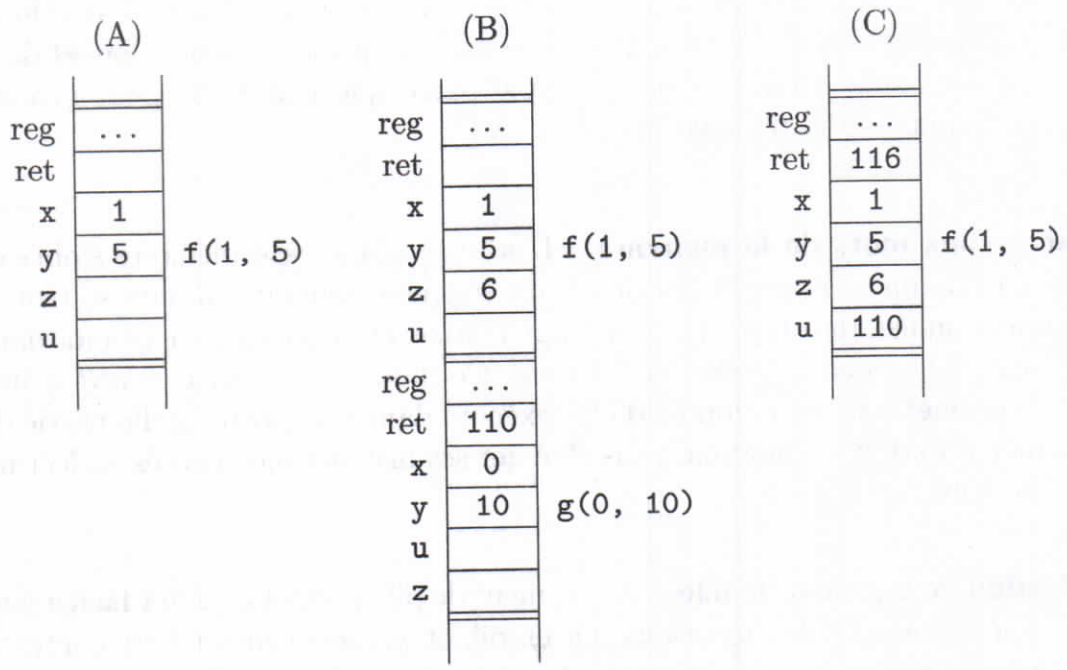

Figure 23.3 – Gestion du segment de pile.

Gestion du tas. La gestion de la mémoire avec une pile n'est pas toujours possible ou souhaitable. Par exemple, si les valeurs allouées dans le corps d'une fonction ont une durée de vie plus grande que l'appel de cette fonction, il n'est pas possible de les allouer dans la pile, sans quoi elles disparaîtraient au retour de la fonction. Prenons par exemple la fonction suivante `allouer` qui renvoie une paire constituée de deux tableaux construits par la fonction.

```
def allouer(x, y):
    t = [y] * x
    u = [x] * y
    return (t, u)
```

Il n'est pas possible d'allouer dans le segment de pile les tableaux `t` et `u`, ni la paire `(t, u)` renvoyée par la fonction, car toutes ces zones mémoires vont disparaître à la fin de l'appel.

Pour cela, ces valeurs doivent être allouées dans le segment du tas qui sert à stocker des valeurs dans une zone mémoire qui n'est jamais effacée. Le tas est simplement vu comme un tableau contigu de cases mémoires. La politique de gestion de cet espace mémoire est beaucoup plus libre que la pile. Pour allouer de la mémoire dans le tas, le programmeur utilise dans certains langages une instruction *explicite* d'allocation (qui porte souvent le nom `malloc`, pour *memory allocation*) ou dans d'autres langages, comme Python, l'allocation est faite automatiquement, sans que le programmeur ne s'en rende compte.

Par exemple, la configuration de la mémoire après l'appel `allouer(2, 3)`, juste au moment où la fonction s'apprête à renvoyer sa valeur, est décrite dans la figure 23.4. Les valeurs des variables `x` et `y` sont allouées dans la pile, mais les tableaux `t` et `u` sont eux alloués dans le tas. Seuls les pointeurs vers ces tableaux sont stockés dans la pile, aux emplacements réservés pour `t` et `u`. La cellule mémoire de la pile qui contient la valeur de retour (`ret`) est elle aussi un pointeur vers une paire dans le tas. Chaque valeur de cette paire est un pointeur vers les zones mémoires allouées pour `t` et `u`.

Ainsi, lorsque la fonction `allouer` termine, toutes les valeurs allouées dans le tas survivent à l'effacement des valeurs dans la pile. La fonction renvoie alors simplement le pointeur de la paire `(t, u)` qui contient bien l'adresse d'une zone mémoire toujours allouée.

La principale difficulté avec la gestion du tas survient au moment où on souhaite libérer des zones mémoires allouées, soit parce que le tas risque de ne plus avoir assez de place, soit tout simplement parce qu'on souhaite libérer de l'espace dès qu'une valeur allouée sur le tas n'est plus utile au programme. Le problème est qu'il n'est pas si simple en général de s'assurer qu'une valeur n'est plus nécessaire. Aussi, si on se trompe en libérant trop tôt une zone mémoire dans le tas, le programme provoquera une erreur à l'exécution dès qu'on tentera d'accéder à cet espace libéré.

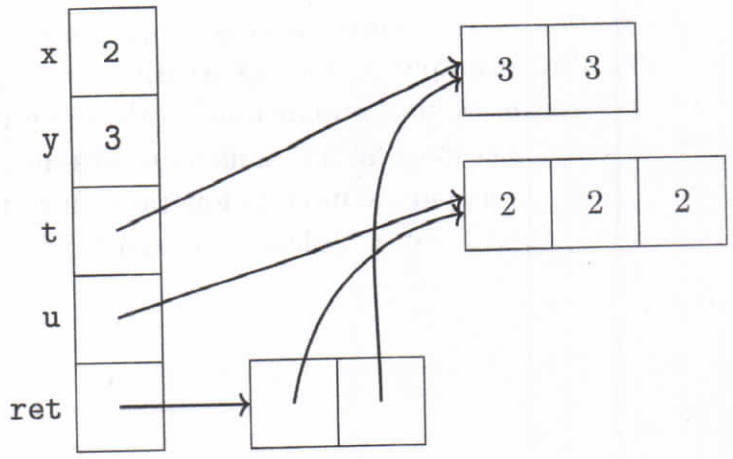

Figure 23.4 – Segments de pile et de tas.

En Python, la libération des zones mémoires est faite automatiquement par l'interpréteur. Il utilise pour cela un algorithme appelé *Glaneur de cellules* ou simplement GC (on dit *Garbage collector* en anglais) qui agit pendant l'exécution d'un programme. Le GC détermine quelles zones mémoires dans le tas sont inutiles et il les libère automatiquement. Ainsi, le programmeur Python n'a jamais à se soucier de la gestion mémoire du tas (allocation ou libération). Tout se fait automatiquement et de manière transparente.

23.3 Langage machine

Les programmes stockés dans la mémoire centrale d'un ordinateur sont constituées d'instructions de bas niveau, directement compréhensibles par le CPU. Il s'agit des instructions du *langage machine*. Lorsqu'elles sont stockées dans la mémoire, ces instructions ne sont ni plus ni moins que de simples nombres binaires, comme les données manipulées par les programmes.

Pour progresser dans l'exécution d'un programme, l'unité de contrôle de l'ordinateur réalise de manière continue, à un rythme imposé par une *horloge* globale, la boucle suivante, appelée *cycle d'exécution d'une instruction*, qui est constituée de trois étapes.

1. **Chargement**. À l'adresse mémoire indiquée par son registre `IP`, l'unité de contrôle va récupérer le mot binaire qui contient la prochaine instruction à exécuter et le stocker dans son registre `IR`.
2. **Décodage**. La suite de *bits* contenue dans le registre `IR` est décodée afin de déduire quelle instruction doit être exécutée et sur quelles données. Cette étape peut nécessiter de lire d'autres mots binaires depuis la mémoire si le format de l'instruction en cours de décodage le nécessite. C'est également à cette étape que sont chargées les données (on

dit aussi *opérandes*) sur lesquelles l'opération va porter (ces données pouvant être dans des registres ou en mémoire).

3. **Exécution**. L'instruction est exécutée, soit par l'ALU, s'il s'agit d'une opération arithmétique ou logique, soit par l'unité de contrôle, s'il s'agit d'une opération de branchement qui va donc modifier la valeur du registre `IP`.

Il est parfois nécessaire d'écrire directement des (morceaux de) programmes en langage machine, par exemple quand un calcul précis doit être réalisé le plus vite possible par la machine. Pour cela, il n'est pas raisonnable (voire possible) d'écrire directement les mots binaires du langage machine. On utilisera alors un langage d'assemblage, appelé aussi *assembleur*, qui est le langage le plus bas niveau d'un ordinateur lisible par un humain. Ce langage fournit un certain nombre de facilités pour programmer, comme des étiquettes symboliques pour identifier des points de programme. À titre d'exemple, nous présentons ci-dessous le langage d'assemblage x86 des microprocesseurs d'Intel (avec la syntaxe NASM) [2].

Sections. Un programme x86 est constitué de trois parties, appelées *sections*. La première section, délimitée par l'instruction `section .data`, définit une mémoire avec des déclarations de constantes, c'est-à-dire des zones de mémoire non modifiables. Chaque constante est déclarée de la manière suivante

`nom` *type* *lval*

où `nom` est l'adresse symbolique de la constante et *type* est le nombre de cases mémoires allouées pour chaque valeur de la liste *lval*. Les types sont des puissances de deux d'octets (`db` pour un octet, `dw` pour deux octets, `dd` pour quatre octets, etc.). Par exemple, la section `.data` suivante initialise à l'adresse mémoire nommée `a` trois mots de 32 *bits* contenant les entiers 101, 278 et 3569. Elle déclare également une constante `b` d'un octet contenant la valeur 42.

```
section .data
a dd 101, 278, 3569
b db 42
```

La section suivante, délimitée par `section .bss`, permet de définir des variables, c'est-à-dire des zones modifiables, initialisées ou non. Les déclarations de variables sont identiques à celles des constantes de la section `.data`, à ceci près que les zones mémoires ne sont pas nécessairement initialisées. Lorsqu'on souhaite simplement réserver de la place mémoire, on utilise les types `resb` (pour un octet), `resw` (pour deux octets), etc. La *valeur* donnée

2. Nous utilisons ici l'assembleur x86 32 bits, de l'architecture IA-32.

pour initialiser la zone mémoire représente alors le nombre de cases à réserver. Par exemple, la section suivante réserve deux mots de 16 *bits* à l'adresse `x` et un mot de 32 *bits* à l'adresse `y`, initialisé avec la valeur 7637.

```
section .bss
x resw 2
y dd 7637
```

Enfin, la section `.text` contient les instructions du programme. Chaque instruction peut être précédée d'une étiquette qui représente de manière symbolique l'adresse de cette instruction. En particulier, la section doit indiquer avec la directive `global` l'étiquette de l'instruction de départ du programme.

Registres. Le langage x86 a huit registres nommés `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `esp` et `ebp`. À ces registres s'ajoutent ceux pour manipuler les segments mémoires : `cs` et `eip` (segment de code et pointeur d'instruction), `ds` (segment de données), `ss` et `esp` (segment de pile et sommet de pile).

Le jeu d'instructions du langage x86 se compose d'instructions de transfert de données, d'opérations de calcul et d'instructions de saut. En voici ci-dessous une liste non-exhaustive.

Instructions de transfert. La principale instruction pour transférer des données entre des registres et/ou la mémoire est `mov`. Elle a la forme suivante.

`mov` destination, source

La source ou la destination peuvent être des registres ou des emplacements en mémoire. Par exemple :

```
mov eax, 42          ; copie la constante 42 dans eax
mov eax, ebx         ; copie le contenu de ebx dans eax
mov [x], eax         ; copie eax dans la variable x
```

mais les copies de la mémoire vers la mémoire ne sont pas possibles directement (il faut passer par un registre).

Des instructions particulières permettent également de manipuler des données sur la pile, comme par exemple :

```
push  source       ; copie source au somment de la pile
pop   destination  ; copie le sommet de la pile dans
                     destination
```

La source ou la destination de ces instructions peuvent être soit des registres, soit des adresses de variables.

```
push 42          ; empile la constante 42
push eax         ; empile le contenu de eax
pop  [x]         ; dépile à l'adresse de x
pop  ebx         ; dépile dans ebx
```

Instructions de calcul. L'assembleur x86 permet d'écrire des opérations arithmétiques élémentaires (addition, soustraction, etc.), mais également des opérations logiques (et, ou, etc.). Par exemple, l'opération d'addition a la forme suivante.

`add` *destination*, *source*

Cette opération ajoute la valeur contenue dans *source* à celle contenue dans *destination* et stocke le résultat dans *destination*. Comme pour l'opération `mov`, la source et la destination peuvent être des registres ou des adresses, mais l'addition entre le contenu de deux adresses mémoire n'est pas possible. Par exemple :

```
add eax, 10    ; ajoute 10 à eax
add eax, ebx   ; ajoute ebx à eax
add [x], eax   ; ajoute eax à x
add ebx, [x]   ; ajoute x à ebx
```

L'instruction de soustraction `sub` ainsi que les opérations logiques (`and`, `or`, etc.) sont similaires. Seules les instructions de multiplication et de division (`mul` et `div`) sont plus restreintes car elles effectuent uniquement leur calcul entre le registre `eax` et un autre registre, case mémoire ou constante.

Instructions de saut. Pour contrôler le flot d'exécution, le langage x86 dispose d'instructions de comparaison et instructions de saut. L'instruction de comparaison a la forme suivante.

`cmp` *source1*, *source2*

L'opération effectue la soustraction *source1* - *source2*, mais ne stocke pas son résultat. Au lieu de cela, elle indique le résultat de la comparaison dans des registres spéciaux, appelés *drapeaux* (en anglais, *flag*). Si le drapeau `zf` vaut 1, c'est que *source1* est égal à *source2*. Si le drapeau `sf` vaut 1, c'est que *source1* est inférieur à *source2*.

Ces drapeaux sont ensuite utilisés par les fonctions de saut. Ainsi, l'instruction de saut conditionnel `je` (pour *jump if equal*), de la forme suivante,

`je` *adr*

permet de sauter à l'adresse *adr* si le drapeau `zf` vaut 1. De même, l'instruction de même forme `jne` saute à l'adresse indiquée si `zf` vaut 0. On trouve également les instructions `jg` et `jl` pour *jump greater* et *jump less*.

```
    section .data
tab dd 11,2,32,40,6

    section .bss
sum dd 0

    section .text
    global _start

_start:
    MOV EAX, 0
    MOV EBX, tab
    MOV ECX, 5
bcl:
    CMP ECX, 0
    JE  fin
    ADD EAX, [EBX]
    ADD EBX, 4
    SUB ECX, 1
    JMP bcl
fin:
    MOV [sum], EAX
```

Figure 23.5 – Exemple de programme x86 (syntaxe NASM).

Enfin, il y a également une instruction de saut non conditionnel `jmp` qui permet de sauter à une adresse. Dans ces instructions, les adresses peuvent être contenues dans des registres, à des adresses mémoires ou être des constantes.

Exemple. La figure 23.5 contient un petit exemple de programme x86 qui effectue la somme des éléments d'un tableau.

Le programme commence par allouer un zone mémoire constante contenant un tableau `tab` de mots de 32 *bits* initialisé avec les valeurs 11, 2, 32, 40 et 6. Puis, dans la section `.bss`, une variable `sum` de 32 *bits* est initialisée à 0. C'est cette case mémoire qui va contenir la somme finale des éléments du tableau à la fin du programme.

```
    section .data
tab dd 11,2,32,40,6
    section .bss
sum dd 0
```

Les instructions sont données dans la section `.text` et le point de départ du programme est fixé à l'étiquette `_start`. Le programme commence par

initialiser le registre EAX à 0. Ce registre va servir d'accumulateur pour la somme des éléments de tab. Le registre EBX, initialisé avec l'adresse de tab, va être utilisé pour accéder successivement aux éléments du tableau. Enfin, le registre ECX est initialisé avec la valeur 5 et va jouer le rôle de la variable de boucle.

```
    section .text
    global _start

_start:
    MOV EAX, 0
    MOV EBX, tab
    MOV ECX, 5
```

Le corps du programme est une boucle qui s'arrête quand le compteur (le registre ECX) arrive à 0.

Pour écrire une boucle, on commence par positionner une étiquette (bcl) devant la condition d'arrêt de la boucle, qui consiste dans notre programme à tester si ECX vaut 0.

```
bcl:
    CMP ECX, 0
    JE  fin
```

Si la condition est vérifiée, le programme saute à la fin de la boucle représentée ici par l'étiquette fin. Sinon, on exécute le corps de la boucle et on termine cette suite d'instructions par un saut non conditionnel à l'étiquette de la boucle.

```
    ...
    JMP bcl
```

Le corps de la boucle consiste tout d'abord à ajouter dans l'accumulateur EAX le contenu de la case mémoire pointée par EBX. Cela est réalisé par l'accès à la case mémoire [EBX].

```
    ADD EAX, [EBX]
```

L'instruction suivante fait alors pointer le registre EBX sur la prochaine case du tableau. Puisque ce dernier contient des mots de 32 *bits*, il faut ajouter 4 à EBX pour que l'adresse corresponde au prochain mot mémoire:

```
    ADD EBX, 4
```

Enfin, on soustrait 1 à ECX et on retourne à l'étiquette de début de la boucle bcl.

```
    SUB ECX, 1
    JMP bcl
```

Le programme se termine en chargeant la variable `sum` avec le contenu de l'accumulateur.

```
fin:
    MOV [sum], EAX
```

À retenir. Les ordinateurs d'aujourd'hui suivent le **modèle de von Neumann**, dans lequel la mémoire stocke à la fois des données et des programmes. Un ordinateur est composé d'une unité **arithmétique et logique** (ALU), d'une unité de **contrôle**, d'une **mémoire** et de **périphériques d'entrée-sortie**. La mémoire prend différentes formes : registres, mémoire cache, mémoire vive, disques durs, etc. La mémoire vive est organisée en différents **segments** : pile, tas, données, code. Un microprocesseur exécute des instructions écrites dans un langage très bas niveau appelé **langage machine**. Le langage **assembleur** est un langage de programmation pour programmer au niveau du langage machine.

Exercices

Exercice 238 Étant données les fonctions Python suivantes.

```
def f(x):
    z = x * x
    return z - x

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

Décrire la configuration de la pile pour l'appel à `g(4)`, au moment (A) où l'appel interne `f(x)` s'apprête à renvoyer sa valeur et (B) où la fonction `g` s'apprête à renvoyer son résultat.

Solution page 493 □

Exercice 239 Étant données les fonctions Python suivantes.

```
def f1(x):
    t = [x]
    u = [t, t]
    return u

def f2(x):
    a = f1(x)
    return a[1]
```

Quelles zones mémoires ont été allouées sur le tas pendant l'exécution de `f2(10)` ? Quelles zones peuvent être libérées ?

Solution page 493 □

Exercice 240 Que fait le programme assembleur suivant ?

```
    MOV EAX, 0
    MOV ECX, 100
ici:
    CMP ECX, 0
    JE  la
    ADD EAX, ECX
    SUB ECX, 1
    JMP ici
la:
```

Solution page 493 □

Exercice 241 Traduire les instructions suivantes en assembleur.

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

Pour cela, on complètera la section `.text` du code assembleur suivant.

```
        section .bss
  x:    dd 5
  y:    dd 10
  z:    dd 0
        section .text
        global _start
_start:
```

Solution page 493

□