
# Fonctionnement bas niveau

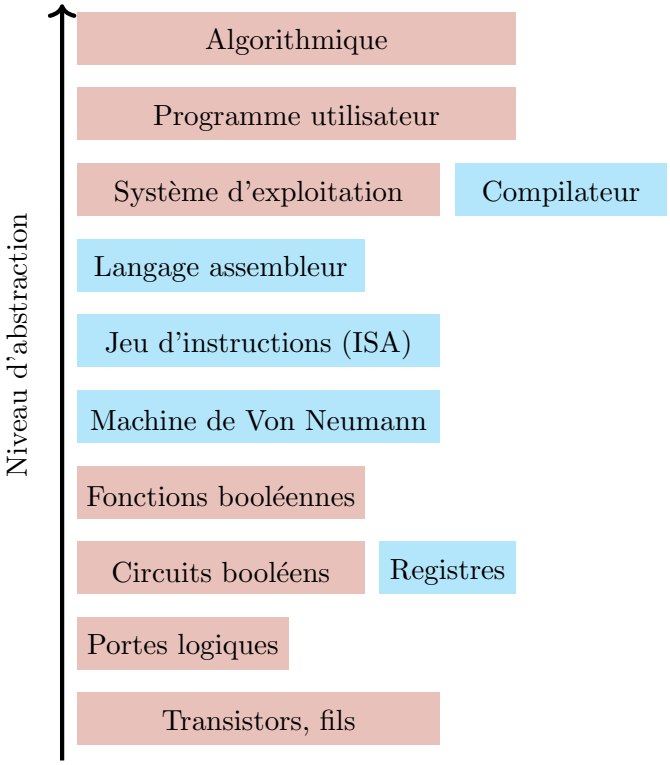
BUT Informatique
Année universitaire 2025–2026

Malory Marin & Rémi Watrigant

# Motivations et Organisation

Ce cours permet de comprendre le fonctionnement de l'objet ubiquitaire qu'est l'ordinateur. En plus de la compréhension de l'informatique de bas niveau (proche de la machine), ce cours permet notamment de comprendre les liens et l'interaction entre le système d'exploitation (et les programmes utilisateurs) avec la composante matérielle de la machine.

**Abstraction** Une notion importante en architecture des ordinateurs est l'**abstraction**. De la même manière qu'on préfère écrire un programme en Python plutôt qu'en binaire, plusieurs niveaux d'abstractions se distinguent dans un ordinateur. Dans cette hiérarchie, le sujet de ce cours sont les niveaux d'abstraction entre le système d'exploitation, et les circuits/fonctions booléens.



# Table des matières

# Chapitre 1

# Architecture bas niveau

## I Architecture matérielle d'un ordinateur

### A Historique

**Historique** Même si l'informatique a explosé au XXème siècle, cette science a débuté avant. Une de premières machine programmable était une machine à tisser perforée. Plus tard, **Ada Lovelace** est à l'origine du premier programme informatique, codé sur la machine analytique de **Charles Babbage**, que l'on peut voir comme l'ancêtre de l'ordinateur. Au XXème siècle, **Turing** et **Von Neumann** ont chacun proposés des modèles de machines programmables et universels, menant aux premiers ordinateurs universels (voir ENIAC). Les avancées technologiques ont menés aux ordinateurs actuels, avec notamment quatre étapes technologiques : les tubes à vide, les transistors, les circuits intégrés et la Very Large Scale Integration.

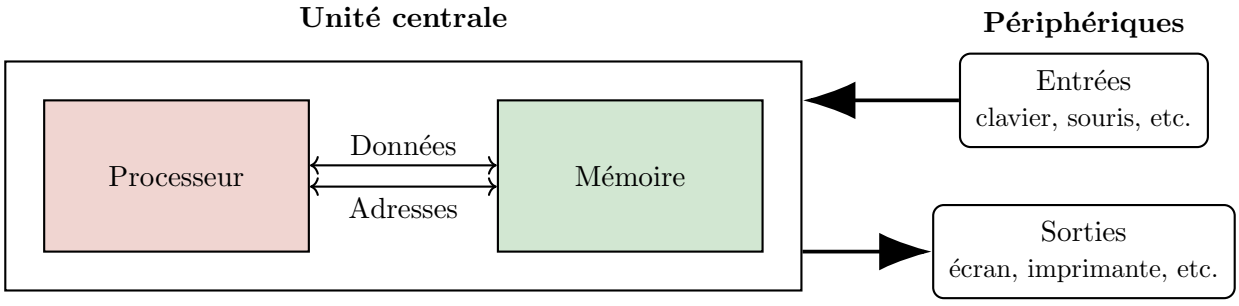
**Explosion des performances** Depuis les premiers ordinateurs, on a pu observer une augmentation exponentielle des puissances de calcul, de la taille des mémoire, etc. La **loi de Moore** (1965) affirme que les ressources des circuits intégrés doublent tous les 18-24 mois. Actuellement, la loi de Moore a atteint ses limites physiques. On tente de la conserver par l'ajout du parallélisme qui permet d'augmenter les performances.

### B Le modèle de Von Neumann

De nos jours, les ordinateurs ont une architecture *type von Neumann*, avec deux composantes principales : la *mémoire* et le *processeur*, formant l'**unité centrale**.

- Une **mémoire** (unité de stockage) qui contient les programmes et les données.
- Un **processeur** (unité de calcul) qui effectue des actions selon les instructions qu'il lit en mémoire.

Avec ces deux principaux éléments, on peut donner le modèle de l'**architecture de Von Neumann** :




**Mémoire adressable.** La mémoire est **adressable** : elle est séparée en cases qui contiennent un nombre fixe de bits (un octet typiquement) et chaque case a une adresse.

Le processeur accède à la mémoire par blocs de 8, 16, 32 ou 64 bits appelés **mots mémoire** (de nos jours, les architectures 64 bits sont les plus courantes).

Pour cet accès, le processeur utilise un registre appelé **compteur ordinal** (*Program Counter* ou PC) qui contient une adresse en mémoire.

**Cycle de von Neumann :**

1. Lire le mot mémoire qui commence à l'adresse PC ;
2. Interpréter ce mot mémoire comme une instruction et l'exécuter ;
3. Augmenter le PC pour passer à l'instruction suivant et recommencer.

Ces échanges processeur/mémoire sont très importants, car les programmes à exécuter et les données sont elles-mêmes dans la mémoire.

On appelle *bus de données* les moyens de circulations entre l'unité centrale et la mémoire.

**Périphériques** Un *périphérique* est un matériel électronique pouvant être raccordé à l'unité centrale par l'intermédiaire d'une interface d'entrée-sortie (par exemple les ports VGA, HDMI, USB, etc). On en distingues plusieurs types :

— Les *périphériques d'entrée* qui envoient des informations à l'unité centrale (par exemple la souris, le clavier, etc).

— Les *périphériques de sortie* qui reçoivent des informations de l'unité centrale (par exemple l'écran, haut-parleur, etc).

— Les *périphériques d'entrée-sortie* qui font les deux à la fois (par exemple une clé USB, un modem, etc).

Ils interagissent avec le système d'exploitation par le biais d'**interruption**.

# II Le processeur

Le **processeur**, ou **Unité centrale de Traitement** (*Computing Processor Unit, CPU*) est le composant électronique chargé de la manipulation et l'exécution des instructions machines codées en binaire (qui sont elles-mêmes dans la mémoire). Il est principalement composé de deux unités :

— l'**ALU** : Unité Arithmétique et Logique, qui réalise les calculs ;

— l'**UC** : Unité de Contrôle, qui cadence et pilote le travail du processeur ;

— mais aussi d'autres unités avancées, comme le FPU (Floating Point Unit), le Memory Managment Unit, etc.

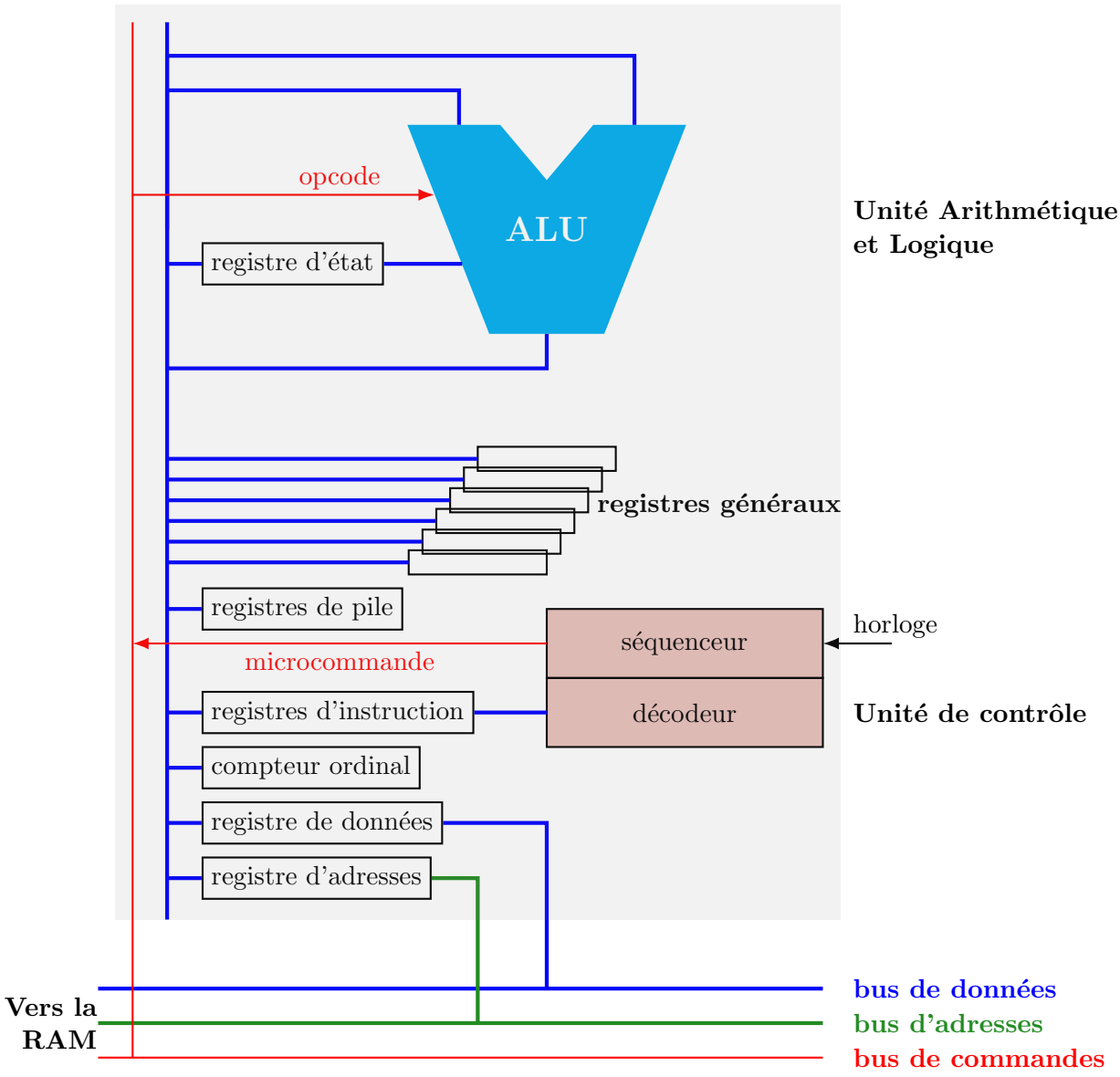
On y trouve notamment :

— des registres, qui stockent des donnée ;

— des bus internes, qui transportent ces données dans les différents endroits du processeur ;

— des circuits booléens (additionneur, décodeur, etc).

Un schéma globale et simplifié d'un processeur est illustré ci-dessous.




## A Les registres

Les registres sont des zones qui permettent de stocker des données en binaire directement dans le processeur. Ils sont de *faible capacité* (32 ou 64 bits, selon le processeur) et de de faible temps d'accès (de l'ordre de la nanoseconde $10^{-9}$s). On distingue deux types de registres :

— les registres **généraux**, qui stockent les données à traiter, les résultats ou les résultats des calculs intermédiaires ;

— les registres **spécifiques**, qui ont une fonction particulière (comme le compteur ordinal PC du cycle de von Neumann).

**Exemple 1.1** *Parmi les registres spécifiques, certains sont d'importance particulière.*

— *Le* ***compteur ordinal*** *(*Program Counter, PC*), qui contient l'adresse mémoire de la prochaine instruction à exécuter.*

— *Le* ***registre d'instruction*** *(*Instruction Register, IR*), qui contient la prochaine instruction à exécuter.*

— *Le* ***registre d'adresse*** *(*Memory Address Register, MAR*), qui contient l'adresse mémoire d'une donnée à récupérer ou stocker.*

— *Le* ***registre de données*** *(*Memory Data Register, MDR*), qui contient les données à échanger entre le processeur et la mémoire.*

— *Le **registre de pile** (*Stack Pointer, SP*), qui pointe vers la zone mémoire où sont stockées (entre autres) les variables du programme en cours d'exécution.*

— *Un **registre d'état** (*Condition Code Register, CCR*) qui contient des informations relatives à la dernière opération effectuée (par exemple une opération arithmétique nulle, dépassement de taille, retenue, etc).*

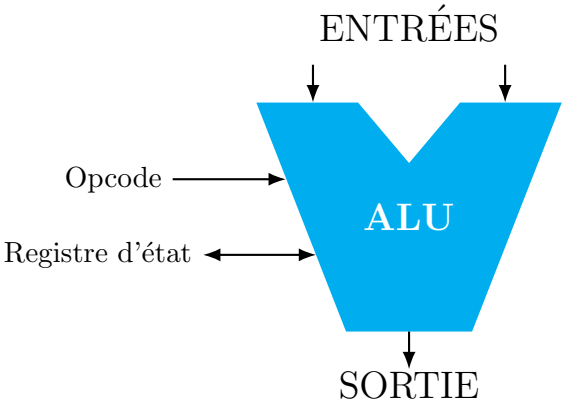
## B L'Unité Arithmétique et Logique

L'**Unité Arithmétique et Logique** (ALU) est constituée de circuits effectuant les principales opérations élémentaires : circuits d'addition, soustraction, comparaison, etc. Elle prend deux opérandes en « entrée » et retourne un résultat en « sortie ».

En parallèle, on lui fournit :

— L'**opcode**, qui est un code spécifiant quelle opération il doit effectuer (qui définira quel circuit utiliser).

— La valeur du **registre d'état** qu'il va mettre à jour.



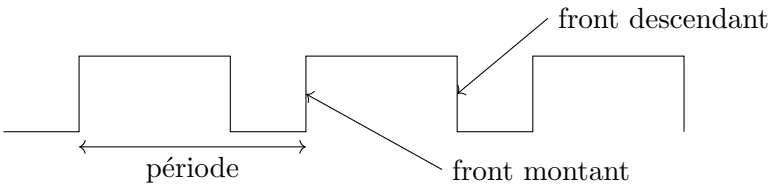

## C L'Unité de Contrôle

**Unité de Contrôle** L'Unité de Contrôle (ou Commande) pilote, au rythme du signal d'horloge, toutes les opérations réalisées par le processeur. Un **cycle d'instruction** (tel que présenté par Hennessy et Patterson), est généralement décomposé en cinq étapes principales :

| | | |
|---|---|---|
| 1) | Lecture de l'instruction en mémoire (PC → mémoire) | Instruction Fetch (IF) |
| 2) | Incrémentation du PC | |
| 3) | Décodage de l'instruction et lecture des registres | Instruction Decode (ID) |
| 4) | Exécution de l'opération (calcul ALU, adresse effective) | Execute (EX) |
| 5) | Accès à la mémoire de données (si nécessaire) | Memory Access (MEM) |
| 6) | Écriture du résultat dans le registre destination | Write Back (WB) |

**Horloge** Une **horloge** est un signal libre ayant une **période** (et donc une **fréquence**) fixée. Une période d'horloge est appelée **cycle d'horloge**, et sa fréquence la **fréquence du processeur**.

On appelle **élément à état** pour tout élément contenant de la mémoire (comme un registre par exemple). Une horloge est nécessaire lorsque un système contient des éléments à état devant être mis à jour.

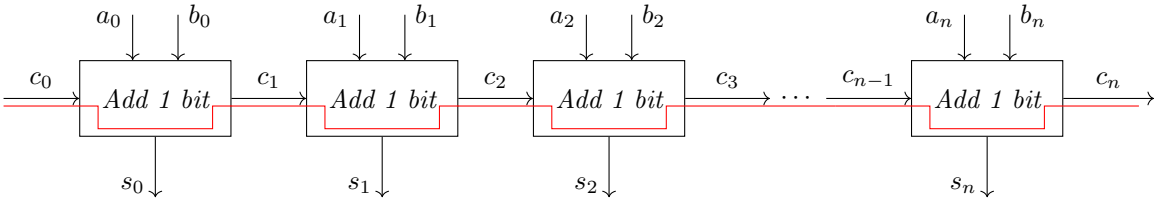



L'état d'un élément à état est mis à jour au **front montant** de l'horloge, et lorsqu'un système contient une unique horloge, on dit qu'il est **synchrone**.

**Stabilité.** Dans un circuit combinatoire, une modification des entrées ne se répercute pas instantanément sur les sorties. Le signal doit se propager à travers les différents composants logiques (portes, multiplexeurs, additionneurs, etc.), chacun introduisant un certain **délai**.

Le temps de propagation d'un circuit combinatoire est défini comme la durée maximale nécessaire pour qu'une variation des entrées se stabilise sur les sorties. Il correspond au plus long chemin de propagation, aussi appelé **chemin critique**, dans le circuit.

**Exemple 1.2** *Un additionneur sur $n$ bits est généralement construit à partir de $n$ circuits élémentaires, appelés* additionneurs 1 bit *(ou* full adders*), connectés en série. Chaque additionneur 1 bit reçoit en entrée deux bits $a_i$ et $b_i$, ainsi qu'une retenue d'entrée $c_i$, et produit en sortie le bit de somme $s_i$ ainsi que la retenue suivante $c_{i+1}$. La retenue se propage ainsi de proche en proche, du bit de poids faible vers le bit de poids fort. Le chemin critique a une longueur de $Kn$, où $K \geqslant 1$ est une constante.*




Dans un processeur synchrone, l'horloge impose un rythme global d'exécution. Pour garantir un fonctionnement correct, la période de l'horloge doit être strictement supérieure au temps de propagation du chemin critique (auquel s'ajoutent les temps de montée, de marge et de synchronisation des registres). Autrement dit, le signal doit avoir le temps de se stabiliser complètement avant d'être échantillonné au front suivant de l'horloge.

Il existe donc un lien direct entre la profondeur logique d'un circuit combinatoire (le nombre de niveaux de portes sur le plus long chemin) et la fréquence maximale de l'horloge : plus le chemin critique est long, plus la fréquence maximale admissible est faible.

**Exercice 1.1** *Supposons que l'on dispose d'un processeur de fréquence 5GHz, et que chaque transistor a un délai de 10ps ($10^{-12}s$) de stabilisation. Pour des soucis de conception, on souhaite que tous les circuits combinatoires soient stabilisé dès lors que 70% de la période d'un cycle d'horloge se soit écoulé. Quelle est la longueur maximale d'un chemin critique que l'on peut utiliser dans un circuit combinatoire dans le processeur ?*

**Solution.** Si le processeur a une fréquence de 5GHz, c'est-à-dire $5 \cdot 10^9 \text{s}^{-1}$, sa période est donc de $\frac{1}{5} \cdot 10^{-9}$s, et ainsi 70% de sa période correspond à $\frac{14}{100} \cdot 10^{-9}$s. Puisqu'un transistor a un délai de $10^{-11}$s, $x$ transistors en séries ont un délai de $x \cdot 10^{-11}$s. On résout l'inégalité

$$x \cdot 10^{-11} \leqslant \frac{14}{100} \cdot 10^{-9}$$

et on obtient $x \leqslant 14$. Ainsi, la longueur maximale d'un chemin critique dans ces conditions est de

14.

Avec l'exercice précédent, on voit bien que des optimisations sont nécessaires (par exemple le chemin critique de l'additionneur en série 32 bits est déjà bien trop important). Des exemples d'optimisations sont :

— les **pipeline** qui permettent de ne pas à avoir à exécuter l'ensemble de la combinatoire sur un seul cycle d'horloge ;

— des conceptions de circuits avec des chemins critiques plus courts ;

**Pipeline** On décrit ici le principe du **pipeline**. Un cycle d'instruction est découpé en plusieurs *étapes* successives, chacune utilisant une partie différente du processeur. Cette organisation permet d'exécuter plusieurs instructions en parallèle : à un instant donné, chaque étape du pipeline traite une instruction différente, comme l'illustre le schéma ci-dessous.

| **Instruction** | **Cycle 1** | **Cycle 2** | **Cycle 3** | **Cycle 4** | **Cycle 5** |
|---|---|---|---|---|---|
| $I_1$ | IF | ID | EX | MEM | WB |
| $I_2$ | | IF | ID | EX | MEM |
| $I_3$ | | | IF | ID | EX |

## D Les bus internes

Les **bus internes** au processeur assurent la communication entre les différents circuits qui le composent. On distingue principalement :

— le **bus de données**, qui transporte les valeurs manipulées par le processeur ;

— le **bus de contrôle**, qui transporte les signaux de commande et de synchronisation.

La **largeur d'un bus** correspond à la quantité de données pouvant être transmise en parallèle et s'exprime en nombre de bits (par exemple : 32 bits, 64 bits).

On distingue également les **bus internes**, utilisés à l'intérieur du processeur, des **bus externes**, qui assurent la communication entre le processeur, la mémoire principale et les périphériques d'entrée/sortie.

## E Exercices

**Exercice 1.2** *Quelle est la durée d'un cycle d'horloge d'un processeur cadencé à 2GHz ?*

**Solution.** La durée d'un cycle d'horloge est l'inverse de sa fréquence, puisque cette durée est exactement la période du signal. Puisque 2GHz correspond à $2 \cdot 10^9 \text{s}^{-1}$, on obtient qu'un cycle dure $0.5 \cdot 10^{-9}$s, soit une demi nanoseconde.

**Exercice 1.3** *On dispose de deux processeurs :*

1. *un processeur A de fréquence 3,2 GHz qui exécute en moyenne 1,2 instructions par cycle d'horloge ;*
2. *un processeur B de fréquence 2,8 GHz, et exécute en moyenne 1,5 instructions par cycle d'horloge.*

*Quel est le processeur qui permet d'exécuter le plus d'instructions par seconde ?*

**Solution.** Notons :

— $f_A$ et $f_B$ les fréquences de $A$ et $B$ ;

— $i_A$ et $i_B$ les nombre d'instructions moyens par seconde de $A$ et $B$.

— $d_A$ et $d_B$ les nombres d'instructions moyens par cycle de $A$ et $B$.

La fréquence est définie comme le nombre de cycle d'horloge par seconde que réalise le processeur.

Donc pour obtenir le nombre de cycle par seconde du processeur, il faut multiplié sa fréquence par le nombre d'instruction par seconde. Ainsi, on a

$$d_A = f_A \times i_A \qquad \text{et} \qquad d_B = f_B \times i_B$$

Puisque $f_A = 3,2{\cdot}10^9 \text{s}^{-1}$ et $i_A = 1,2$, on obtient $d_A = 3,84\text{s}^{-1}$. De manière similaire, $d_B = 4,2\text{s}^{-1}$. Ainsi, $B$ permet d'exécuter le plus d'instructions par seconde.

# III La mémoire

## A Organisation de la mémoire et Unités

On distingue quatre principaux types de mémoire dans un ordinateur :

— les registres,

— la mémoire cache,

— la mémoire centrale (RAM),

— et la mémoire de masse (disque dur).

On définit quelques propriétés importantes d'une mémoire.

— Elle est dite **volatile** lorsqu'elle perd son contenu lorsqu'elle n'est pas alimentée électriquement (RAM, cache, registres), et **permanente** dans le cas contraire (disque dur).

— On appelle **capacité** d'une mémoire la quantité de données (en bits) qu'elle peut stocker.

— Le **temps d'accès** d'une mémoire est la durée de l'intervalle de temps entre la demande de lecture/écriture par le processeur et la fin de cette tâche.

**Unités.** L'unité d'information usuelle est le **bit** (noté b). Dans la mémoire, la plus petite unité « adressable » est le **byte** (noté B). Généralement, un byte vaut 8 bits, qu'on appelle un **octet** (noté o). Très majoritairement, les capacités des mémoires sont des puissances de 2 (32, 64, 128, 256, etc).

Plutôt que les préfixes usuelles, appelés **préfixes du SI** (kilo k, méga M, etc), on utilisera plutôt les **préfixes binaires**.

| Nom | Symbole | Facteur |
|---|---|---|
| kibi | ki | $2^{10} = 1\,024$ |
| mébi | Mi | $2^{20} = 1\,048\,576$ |
| gibi | Gi | $2^{30} = 1\,073\,741\,824$ |
| tébi | Ti | $2^{40} = 1\,099\,511\,627\,776$ |

| Nom | Symbole | Facteur |
|---|---|---|
| kilo | k | $10^3$ |
| méga | M | $10^6$ |
| giga | G | $10^9$ |
| téra | T | $10^{12}$ |

**Exercice 1.4** *Un disque dur a une capacité affichée de 500Go, donnez :*

— *le nombre d'octets qu'il contient ;*

— *le nombre de bits qu'il contient ;*

— *le nombre de Gio ;*

— *la différence en pourcentage entre le nombre de Go et de Gio.*

**Solution.** La capacité annoncée du disque dur est de 500 Go, ce qui correspond à

$$500 \times 10^9 = 5 \times 10^{11}$$

octets.

Le disque dur contient $5 \times 10^{11}$ octets.

Un octet étant composé de 8 bits, le nombre total de bits est

$$8 \times 5 \times 10^{11} = 4 \times 10^{12}.$$

Le disque dur contient $4 \times 10^{12}$ bits.

On rappelle qu'un gibioctet correspond à $2^{30}$ octets. Le nombre de gibioctets contenus dans le disque est donc

$$\frac{5 \times 10^{11}}{2^{30}} \approx 465.$$

La capacité réelle du disque est d'environ 465 Gio.

La différence relative entre la capacité annoncée et la capacité réelle se calcule par

$$\frac{500 - 465}{500} \approx 0{,}07.$$

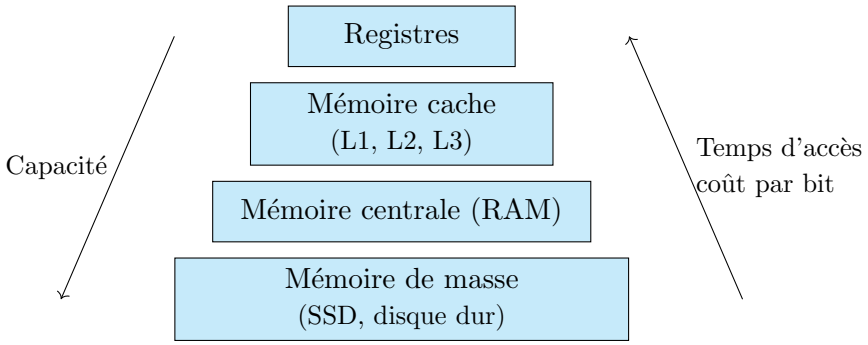
La différence est d'environ 7 %.

**Hiérarchie mémoire.** L'organisation de la mémoire dans un ordinateur est *hiérarchique*. Elle est structurée en plusieurs niveaux, allant des mémoires les plus rapides et les plus proches du processeur aux mémoires les plus lentes mais offrant de grandes capacités de stockage.

Plus une mémoire est proche du processeur :
- plus sa **capacité** est faible ;
- plus son **temps d'accès** est court ;
- plus son **coût par bit** est élevé.

À l'inverse, les mémoires plus éloignées du processeur sont plus lentes, mais offrent une capacité beaucoup plus importante à moindre coût.

Cette organisation permet de concilier deux objectifs contradictoires :
- fournir au processeur des données le plus rapidement possible ;
- disposer d'une grande capacité de stockage à un coût raisonnable.




Les mémoires situées près du processeur sont généralement **volatiles** (leur contenu est perdu hors tension), tandis que les mémoires de masse sont **non volatiles** et assurent le stockage permanent des données.

| Mémoire | Capacité | Temps d'accès | Volatile | Localisation |
|---|---|---|---|---|
| Registres | quelques octets | $\sim$ 1 ns | oui | processeur |
| Cache | Ko à Mo | quelques ns | oui | processeur |
| RAM | Go | $\sim$ 100 ns | oui | carte mère |
| SSD / HDD | Go à To | ms | non | externe |

## B Mémoire cache

La **mémoire cache** est une mémoire rapide située entre le processeur et la mémoire centrale (RAM). Elle sert à conserver temporairement les données et instructions les plus fréquemment utilisées afin de réduire le temps d'accès moyen à la mémoire.

La mémoire cache est généralement organisée en plusieurs **niveaux** :

— **Cache L1** : très faible capacité, mais extrêmement rapide. Il est intégré directement au cœur du processeur.

— **Cache L2** : capacité intermédiaire et temps d'accès plus élevé que le L1. Il est situé à proximité du processeur, souvent sur le même circuit intégré.

— **Cache L3** : capacité plus importante et temps d'accès encore supérieur. Il est généralement partagé entre plusieurs cœurs et peut être situé sur la carte mère ou sur le même circuit que le processeur selon l'architecture.

**Remarque 1.1** *Les processeurs modernes disposent souvent de plusieurs* ***unités de calcul*** *(cœurs). Certains niveaux de cache sont* privés *à un cœur (typiquement le cache L1), tandis que d'autres peuvent être* partagés *entre plusieurs cœurs (souvent le cache L3).*

La technologie utilisée pour la mémoire cache est la **SRAM** (*Static Random Access Memory*). Cette mémoire est *volatile*, mais elle ne nécessite pas de rafraîchissement périodique, contrairement à la DRAM utilisée pour la mémoire centrale. Cette caractéristique explique sa grande rapidité, au prix d'un coût et d'une consommation plus élevés.

## C Mémoire centrale

La **mémoire centrale**, souvent appelée **RAM** (*Random Access Memory*), assure le stockage des informations (données et instructions) qui peuvent être directement manipulées par le processeur lors de l'exécution des programmes.

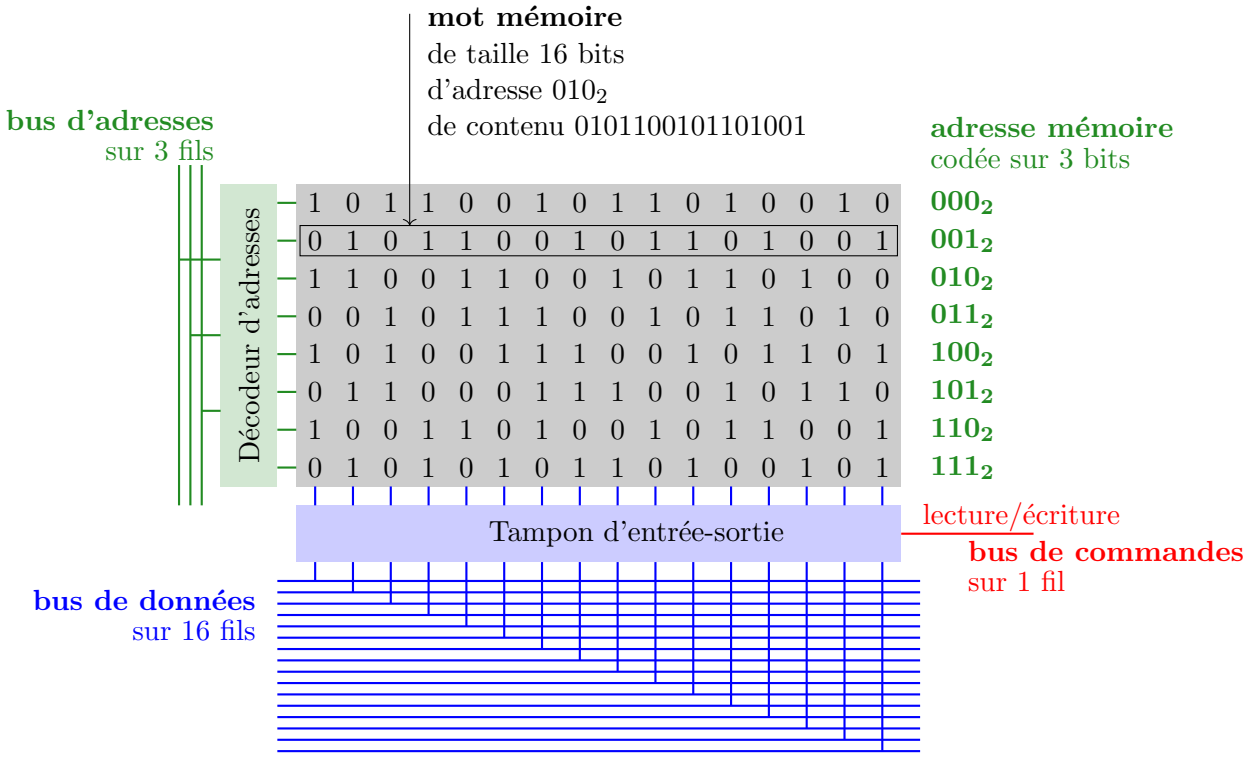
La technologie généralement utilisée est la **DRAM** (*Dynamic Random Access Memory*). Cette mémoire est *volatile* : son contenu est perdu lorsque l'alimentation est coupée. De plus, elle nécessite un *rafraîchissement périodique* afin de conserver les valeurs stockées, ce qui explique un temps d'accès plus élevé que celui de la mémoire cache.

La mémoire centrale est organisée en **mots mémoire**, chacun étant repéré par une **adresse**. Deux caractéristiques la décrivent :

— la **taille des adresses**, qui correspond à la largeur du *bus d'adresses* ;

— la **taille des mots mémoire**, qui correspond à la largeur du *bus de données*.

Si les adresses sont codées sur $n$ bits, alors la mémoire centrale peut contenir $2^n$ adresses distinctes, chacune désignant un mot mémoire.




**Exercice 1.5** *On considère un bus d'adresses de largeur 32 bits.*

— *Quelle est la capacité d'adressage ?*

— *Quelle est la plage des adresses, exprimées en base 16 ?*

*Quelle est la largeur du bus d'adresses nécessaire pour adresser par octet une mémoire centrale d'une capacité de 64 Gio.*

**Solution.** Si le bus d'adresses a une largeur de 32 bits, alors il y aura $2^{32}$ adresses possibles. Ainsi, la plage (en base 2) est l'intervalle suivant :

$$[(\underbrace{0\cdots0}_{32})_2, ..., (\underbrace{1...1}_{32})_2]$$

En base 16, chaque paquet de 8 bits devient un élément de $\{0, ..., 9, A, B, C, D, E, F\}$.

La capacité d'adressage est de $2^{32}$ et la plage d'adresses en base 16 est $[(00000000)_{16}, ..., (FFFFFFFF)_{16}]$.

Si la mémoire centrale a une capacité de 64Gio, c'est-à-dire qu'elle peut contenir $64 \cdot 2^{30} = 2^6 \cdot 2^{30} = 2^{36}$ mots mémoires. Un bus d'adresses de largeur $n$ permet d'obtenir $n$ adresses.

Ainsi, la largeur de bus d'adresses nécessaire est de 36 bits.

## D Mémoire de masse

La **mémoire de masse** est destinée au stockage à long terme des données et des programmes. Il s'agit d'une mémoire dite *permanente* (ou *mémoire morte*) : elle ne nécessite pas d'alimentation électrique pour conserver les informations qu'elle contient.

La mémoire de masse se caractérise par une **très grande capacité** de stockage, mais par un **temps d'accès élevé**. Elle n'est donc pas directement manipulée par le processeur ; les données doivent d'abord être transférées en mémoire centrale avant d'être utilisées.

Parmi les principaux types de mémoire de masse, on distingue notamment :

— les **disques durs magnétiques** (HDD), qui offrent une grande capacité à faible coût, mais avec des temps d'accès relativement importants ;

— la **mémoire flash**, utilisée dans les **clés USB** et les **disques SSD** (*Solid-State Drive*), plus rapide que les disques durs magnétiques et dépourvue de pièces mécaniques.

# IV Bus

Un **bus** permet aux différents composants de l'ordinateur (processeur, mémoire, périphériques) d'échanger des informations. Il peut être vu comme un *ensemble de fils* qui véhiculent les données, les adresses et les signaux de contrôle entre ces composants.

Un bus se caractérise principalement par les éléments suivants :

— la **largeur du bus** (en bits), qui correspond au nombre de fils le composant. Elle détermine la quantité d'informations pouvant être transmise simultanément ;

— la **période du bus**, qui représente le temps minimal entre deux opportunités de transfert de données sur le bus.

— la **fréquence du bus**, exprimée en Hertz (Hz), qui correspond au rythme auquel le bus peut effectuer des transferts. Il s'agit de l'inverse de la période du bus.

— la **bande passante** du bus (en octets par seconde) correspond au nombre d'octets pouvant être véhiculés par seconde. À chaque période du bus, une quantité de données égale à la largeur du bus peut être transmise. La bande passante dépend donc à la fois de la largeur du bus et de sa période.

$$\text{bande passante} = \frac{\text{largeur du bus}}{8} \times \text{fréquence du bus} = \frac{\text{largeur du bus}}{8} \times \frac{1}{\text{période du bus}}$$

On distingue principalement deux types de bus selon la manière dont l'information est transmise :

— les **bus série**, utilisés pour les communications sur de longues distances. Les bits y sont transmis séquentiellement, les uns à la suite des autres sur un nombre réduit de fils ;

— les **bus parallèles**, utilisés sur de courtes distances (entre le processeur, la mémoire centrale ou certaines interfaces). Chaque bit est transmis sur un fil distinct, ce qui permet un transfert simultané de plusieurs bits.

**Exercice 1.6** *Soit un bus de largeur 32 bits, cadencé à une fréquence de 33 MHz. Calculez la bande passante de ce bus en Mo/s.*

**Solution.** Sa fréquence est de 33 MHz, c'est-à-dire $33 \cdot 10^6$ s$^{-1}$.

Ainsi, sa bande passante est de $\frac{32}{8} \cdot 33 \cdot 10^6 = 132 \cdot 10^6$ o/s, c'est-à-dire 132 Mo/s.

# V Exercices

**Exercice 1.7** *On considère un processeur cadencé à 1 GHz qui n'utilise ni pipeline, ni cache, ni autre optimisation. On a également une mémoire centrale avec un temps d'accès de 20 ns.*

*Supposons que le processeur veuille récupérer deux entiers en mémoire, et les additionner, quel pourcentage de son temps passe-t-il à attendre.*

**Solution.** Dans ce processeur, un cycle d'horloge dure $10^{-9}$ s, c'est-à-dire 1 ns. Le processeur réalise trois opérations : deux lectures et une addition. Chaque lecture est suivant d'une attente de 20 ns. Ainsi, le processeur réalise cette opération en 43 ns, en ayant attendu 40 ns.

Le processeur a passé $\frac{40}{43} \simeq 93\%$ de son temps en attente.

# Chapitre 2

# Programmes et processus

## I Programmes, compilation et assemblage

### A Langages

Un *langage de programmation* permet au programmeur de formuler des algorithmes qui peuvent ensuite produire des programmes exécutables par la machine. On distingue les *langages de bas niveau* et de *haut niveau*.

⚠ Ces concepts ne sont pas bien formellement définis, on propose ici juste une définition possible.

**Langage de haut niveau** Un langage de haut niveau permet au programmeur d'exprimer simplement un algorithme à l'aide de structures de contrôle proches du raisonnement humains. Des exemples de tels langages sont le C, C++, Java ou encore Python.

Un programme source en langage de haut niveau est écrit dans un **fichier texte** où chaque octet représente un caractère ASCII.

| 01101001 | 01101110 | 01110100 | 00100000 | 01111000 | 00111011 |
|---|---|---|---|---|---|
| i | n | t | | x | ; |

Un tel programme n'est pas directement exécutable : **il doit être compilé ou interprété**.

**Langage de bas niveau.** Un langage de bas niveau est un langage spécifique à une « architecture » et chaque processeur en a une différente. On parle aussi de *langage machine*.

Un **programme exécutable** est un ensemble d'instructions en langage machine. Il existe plusieurs formats de programme exécutable :

— GNU/Linux : **ELF** (Executable and Linkable Format)

— Windows : **PE** (Portable Executable)

Pour qu'un humain puisse lire un langage de bas niveau, on peut utiliser un langage d'assemblage (ou langage assembleur), qui représente les instructions du langage machine.

| Suite de 32 bits | Instruction assembleur |
|---|---|
| 1110 0011 1010 0000 0000 0000 0001 0000 | MOV r0 r1 |
| 1110 0000 1000 0000 0000 0000 0001 0001 | ADD r0 r0 r1 |

De manière similaire à un langage de haut niveau, un langage d'assemblage n'est pas directement exécutable par le processeur et est stocké dans un fichier texte.

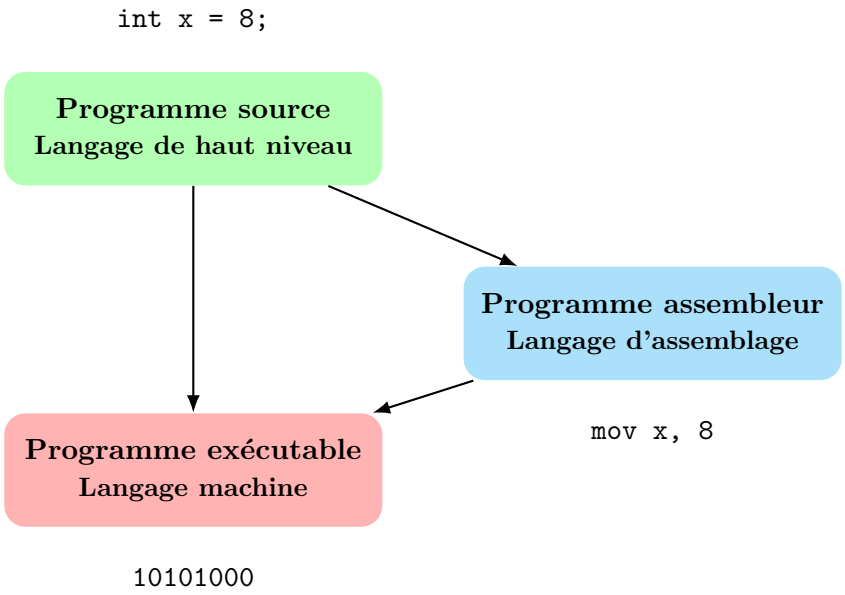
**Remarque 2.1** *On fait souvent la distinction entre « fichier binaire », « fichier texte », etc. À la fin, tout est stocké en binaire.*

— *Un « fichier texte » est un fichier dont chaque octet est interprété comme un caractère ASCII (un nombre entre 0 et 255), et peut être ouvert par tout éditeur de texte.*

— *Un « fichier binaire » possède une structure plus complexe. On peut l'ouvrir avec un éditeur de texte mais il n'aura aucun sens.*

### B Chaîne de production d'un programme exécutable

Un **compilateur** est un programme qui transforme un programme source en un programme exécutable. Un **assembleur** est un programme qui transforme un programme en langage assembleur en un programme exécutable.




La plupart des compilateurs peuvent aussi fournir le code en langage d'assemblage.

**Où sont stockés ces fichiers ?** Tous les programmes (source, en langage d'assemblage et en langage machine) sont stockés dans la mémoire de masse. Le programme en langage machine est prêt à être exécuté, et est donc « **relogeable** ». Toutes les adresses mémoires sont calculées en considérant que l'adresse du premier octet du programme est 0. Pour l'exécuter, on le place en mémoire centrale, et toutes les adresses sont remplacées par les « vraies adresses ».

## II Processus

Un **processus** est un programme en cours d'exécution, c'est-à-dire que le processeur est en train d'exécuter les instructions qu'il contient.

Il y a généralement plusieurs processus en même temps, et même plusieurs processus exécutant le même programme exécutable.

### A Le chargement

Au début de l'exécution, il y a le **chargement** :

— Un espace est alloué en mémoire centrale : **l'espace d'adressage** du processus. Celui-ci est divisé en plusieurs zones distinctes.

— Le code du programme est copié dans l'espace d'adressage et toutes les adresses mémoire sont remplacées par leur « vraie » valeur (par rapport à l'espace d'adressage).

— Le registre du **compteur ordinal (PC)** est initialisé avec l'adresse de la première instruction du programme, puis le processeur exécute ses cycles d'instruction.

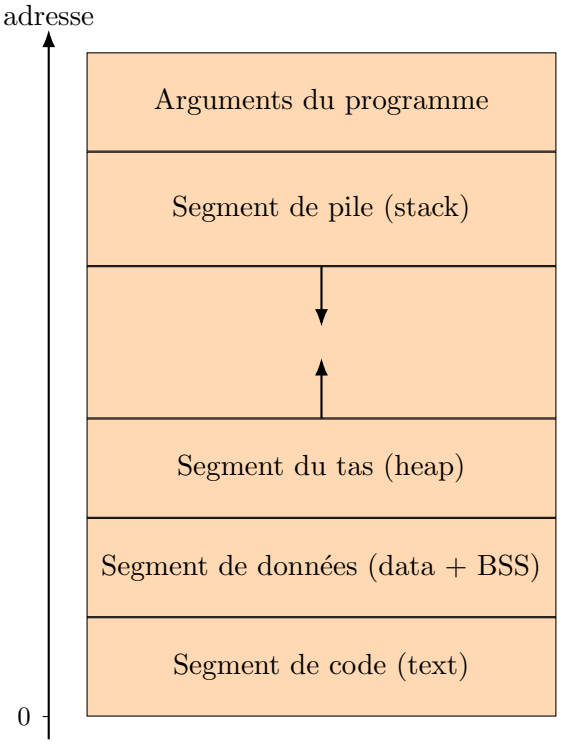
Lorsque le processus se termine, son espace d'adressage est libéré.

| Quoi ? | Langage ? | Où ? | |
|---|---|---|---|
| Programme source | Langage haut niveau | Mémoire de masse | Compilation |
| Programme exécutable | Langage machine relogeable | Mémoire de masse | Chargement |
| Programme exécutable | Langage machine | Mémoire centrale | |

## B Espace d'adressage

L'**espace d'adressage** d'un processus correspond à l'ensemble des adresses mémoire auxquelles ce processus peut accéder dans la mémoire centrale.

Il est organisé en **segments**, chacun occupant une plage contiguë d'adresses et ayant un rôle précis.




— **Segment de code (text)**
Contient les instructions du programme en langage machine ainsi que les constantes. Il s'agit d'un segment de taille fixe (déterminée au chargement), accessible en lecture seule.

— **Segment de données (data)**
Contient les variables globales et les variables statiques du programme. On distingue deux parties :

**BSS :** variables non initialisées ;

**Data :** variables initialisées.

Ce segment est de taille fixe et accessible en lecture et en écriture.

— **Segment du tas (heap)**
Contient les données allouées dynamiquement, notamment via `malloc`, `calloc`, `realloc` en C ou `new` en C++. Sa taille est variable. Il est accessible en lecture et en écriture. Il est utilisé lorsque la taille des données n'est pas connue à la compilation.

— **Segment de pile (stack)**
Sert à gérer les appels de fonctions. Il contient les paramètres d'entrée, les variables locales, la valeur de retour et l'adresse de retour vers la fonction appelante. Sa taille est variable mais limitée. Il est accessible en lecture et en écriture.

— **Arguments du programme**
Correspondent aux données passées lors de l'exécution du programme, par exemple :

```
./monProgramme p1 p2 p3
```

En C, ils sont récupérés via les paramètres `argc` et `argv` de la fonction `main`. Dans cet exemple :
- `argc = 4`
- `argv[0] = "./monProgramme"`,
- `argv[1] = "p1"`,
- `argv[2] = "p2"`,
- `argv[3] = "p3"`

**Exemple 2.1** *Si dans une fonction on écrit :*

```
int* tab = malloc(10*sizeof(int));
```

- *La variable* ***`tab`*** *(un pointeur, généralement 8 octets sur une architecture 64 bits) est stockée dans la* ***pile****.*
- *Le bloc de mémoire alloué (ici* ***`10 * sizeof(int)`*** *octets) est stocké dans le* ***tas****.*

# III Le segment de pile

Dans le segment de pile, l'ordre est **LIFO** (*Last In First Out*). On accède aux données de la pile via l'adresse du **sommet** de la pile, stockée dans le registre de pile **Stack Pointer** (SP).

**Fonctionnement de la pile** Lorsqu'on rentre dans une fonction, plusieurs informations sont contenues dans des registres :
- les valeurs des paramètres en entrées et
- l'adresse de retour (adresse de l'instruction qui a appelé la fonction)

On réserve également de la place sur la pile pour toutes les variables locales de la fonction. À chaque fois que la pile est agrandit, on ajuste la valeur du Stack Pointer.

Si la fonction fait elle-même appel à une autre fonction :
- l'adresse de l'instruction qui a fait l'appel est placé dans le registre dédié ;
- la fonction appelée « empile » ses données au dessus.

**Exemple de programme** Considérons la fonction suivante :

```
int f(int x, int y){
        int z = 0 ;
        for (int i= 0 ; i<x + y ; i++){
                z+=i ;
        }
        return z ;
}
```

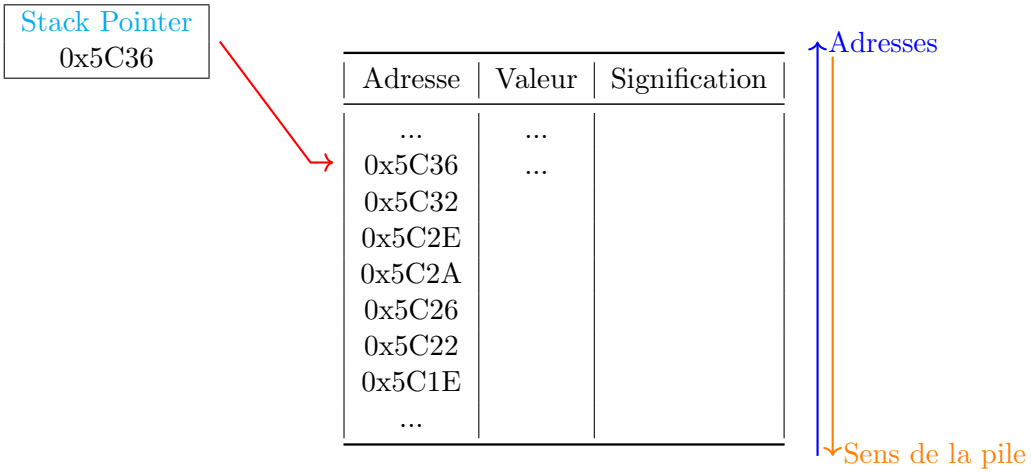
**Exercice 2.1** *Que calcule la fonction* `f` *ci-dessus ?*

**Solution.** Étant donné $x$ et $y$ en entrée, la sortie $f(x, y)$ somme les entiers $i$ allant de 0 à $x+y-1$.

Ainsi,

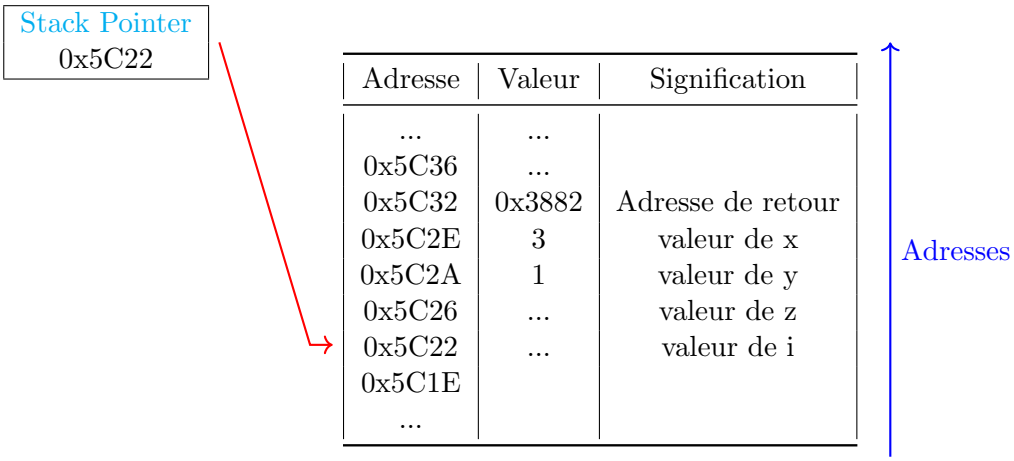
$$f(x,y) = \sum_{i=0}^{x+y-1} i = \frac{(x+y-1)(x+y)}{2}$$

On réalise un appel à `f(3,1)` depuis l'instruction d'adresse 0x3882, avec l'état de la pile suivant :



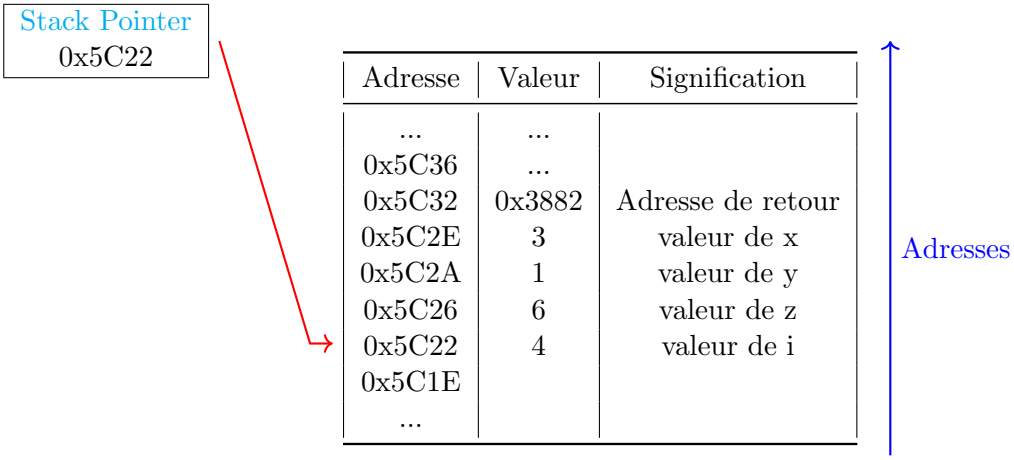

| Adresse | Valeur | Signification |
|---|---|---|
| ... | ... | |
| 0x5C36 | ... | |
| 0x5C32 | | |
| 0x5C2E | | |
| 0x5C2A | | |
| 0x5C26 | | |
| 0x5C22 | | |
| 0x5C1E | | |
| ... | | |

Au début de la fonction, l'état de la pile devient le suivant :



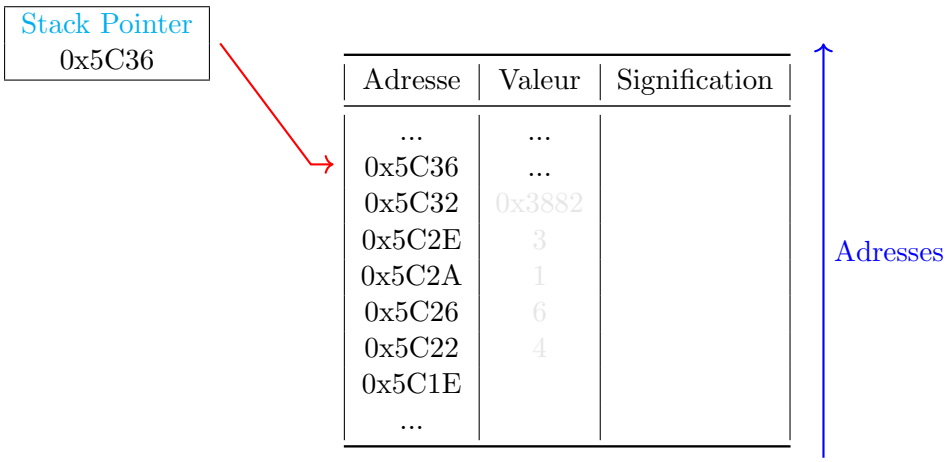

| Adresse | Valeur | Signification |
|---|---|---|
| ... | ... | |
| 0x5C36 | ... | |
| 0x5C32 | 0x3882 | Adresse de retour |
| 0x5C2E | 3 | valeur de x |
| 0x5C2A | 1 | valeur de y |
| 0x5C26 | ... | valeur de z |
| 0x5C22 | ... | valeur de i |
| 0x5C1E | | |
| ... | | |

À la fin de l'exécution de la fonction, on est dans l'état suivant :




| Adresse | Valeur | Signification |
|---|---|---|
| ... | ... | |
| 0x5C36 | ... | |
| 0x5C32 | 0x3882 | Adresse de retour |
| 0x5C2E | 3 | valeur de x |
| 0x5C2A | 1 | valeur de y |
| 0x5C26 | 6 | valeur de z |
| 0x5C22 | 4 | valeur de i |
| 0x5C1E | | |
| ... | | |

Lorsqu'on termine une fonction, on récupère l'adresse de retour qui avait été empilée, et on supprime les données de la pile que notre fonction a utilisé en mettant à jour le Stack Pointer. Sur l'exemple précédent, on obtient :




# IV Variables et portée

On distingue au moins trois types de variables dans un programme C :

— Les **variables locales** qui sont déclarées dans une fonction. Une telle variable a une portée seulement dans la fonction où elle est déclarée, et se situe dans la pile.

— Les **variables globales** sont déclarées en dehors de toute fonction. Leur portée couvre tout le programme après leur déclaration, et elles se situent dans le segment de données.

— Une **variable statique** peut être déclarée dans une fonction. Sa portée est limitée à cette fonction, mais sa durée de vie est celle du programme, et elle se situe dans le segment de données.

```c
int variable_globale = 5;

int une_fonction() {
    int variable_locale = 8;
    return variable_locale;
}

int autre_fonction() {
    static int variable_statique = 0;
    variable_statique++;
    printf("%d\n", variable_statique);
    return variable_statique;
}

int main() {
    autre_fonction();
    autre_fonction();
    return 0;
}
```

**Exercice 2.2** *Quel sera le résultat du programme ci-dessus ?*

**Solution.** Ce programme affiche successivement 1 puis 2 car `variable_statique` conserve sa valeur entre les appels à la fonction.

# Chapitre 3

# Langage assembleur ARM simplifié

## I Architectures de jeu d'instructions

Un *jeu d'instructions* (Instruction Set Architecture, ISA) est l'ensemble des instructions en langage machine qu'un processeur peut exécuter. On parle également d'*architecture de jeu d'instructions*. Ces instructions permettent d'effectuer des opérations élémentaires (addition, opérations logiques comme le OU, chargement et stockage en mémoire, etc.).

On distingue principalement deux grandes familles d'architectures : les architectures **RISC** (*Reduced Instruction Set Computer*) et les architectures **CISC** (*Complex Instruction Set Computer*).

— Dans les architectures **CISC**, le jeu d'instructions est riche et étendu. Il existe de nombreux modes d'adressage (différentes manières d'accéder aux données en mémoire) et certaines instructions peuvent nécessiter plusieurs cycles d'horloge pour s'exécuter.

— Dans les architectures **RISC**, le jeu d'instructions est plus réduit et plus simple. Les instructions sont généralement de taille fixe et conçues pour s'exécuter en un seul cycle d'horloge. En pratique, cette simplicité facilite le pipeline et permet souvent de concevoir des processeurs plus efficaces énergétiquement.

Parmi les processeurs de type CISC, les plus connus appartiennent à la famille **x86** (Intel : Pentium, Celeron, Xeon, Core i3/i5/i7, etc., ainsi que leurs équivalents AMD). Ces processeurs ont longtemps dominé le marché des ordinateurs personnels.

Les architectures RISC ont longtemps été principalement utilisées dans les systèmes embarqués, notamment avec les architectures **ARM** et **MIPS**. On peut également citer la gamme **PowerPC** d'IBM, qui a équipé les ordinateurs Apple entre 1994 et 2006.

**ARM** (*Advanced RISC Machines*) est une entreprise britannique fondée en 1990. Avec l'essor des smartphones et des tablettes, l'architecture ARM est devenue l'une des plus répandues au monde. ARM ne fabrique pas directement les processeurs : l'entreprise conçoit l'architecture et vend des licences que d'autres fabricants utilisent pour produire leurs propres puces. Par exemple, les processeurs *Qualcomm Snapdragon* équipent de nombreux smartphones et tablettes.

Plus récemment, Apple a conçu ses propres processeurs basés sur l'architecture ARM, connus sous le nom d'*Apple Silicon*, qui équipent désormais ses ordinateurs portables et de bureau.

**Remarque 3.1** *En 2020, l'entreprise américaine Nvidia, principalement connue pour ses cartes graphiques, a annoncé son intention de racheter ARM pour environ 40 milliards de dollars. Face aux préoccupations des autorités de régulation de la concurrence (États-Unis, Royaume-Uni, Union européenne), le projet a finalement été abandonné en 2022.*

## II Description d'ARM

Nous décrivons ici de manière simplifiée le jeu d'instructions d'une architecture ARM 32 bits.

## A Les registres

Un tel processeur ARM possède 16 registres de 32 bits, qui sont dans un espace mémoire propre au processeur. On distingue les registres suivants :

— 13 registres de mémoires ; **R0**, **R1**, ..., **R12** qui peuvent stocker des valeurs afin de faire des calculs ;

— 1 registre **SP** Stack Pointer, qui stocke l'adresse du haut de la pile ;

— 1 registre **LR** Link Register, qui stocke l'adresse de l'instruction où revenir lorsqu'on fait un appel de fonction ;

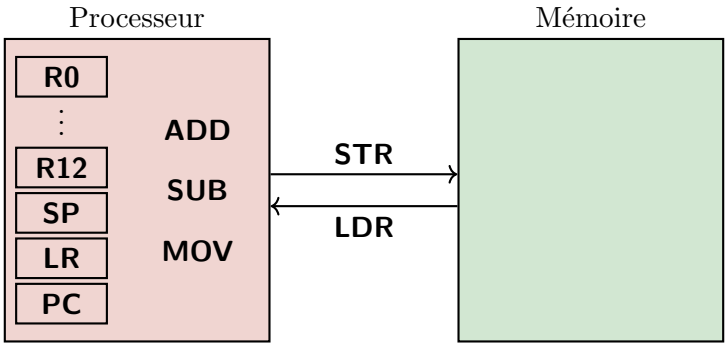
— 1 registre **PC** Program Counter, qui stocke l'adresse de la prochaine instruction à exécuter.

## B Les principales instructions assembleurs

On distingue deux types d'instructions :

— celles qui permettent d'interagir avec la RAM ;

— celles qui se cantonnent au processeur et ses registres.




### 1 Au sein du processeur

**MOV** : permet de copier une valeur dans un registre.

**Utilisation 1** Depuis un autre registre, par exemple **MOV R2, R1** copie la valeur du registre **R1** dans le registre **R2**.

**Utilisation 2** Depuis une valeur constante (immédiate), par exemple **MOV R1, #133** copie la constante décimale `133` dans le registre **R1**.

**ADD** : permet d'effectuer des additions.

**Utilisation 1** Addition entre registres, par exemple **ADD R3, R1, R2** calcule **R1 + R2** et place le résultat dans **R3**.

**Utilisation 2** Addition en plaçant le résultat dans un des registres opérandes, par exemple : **ADD R1, R1, R2** remplace la valeur de **R1** par **R1 + R2**.

**Utilisation 3** Addition avec une constante (opérande immédiate), par exemple : **ADD R1, R2, #5** calcule **R2 + 5** et place le résultat dans **R1**.

**Exercice 3.1** *Placer l'entier 36 dans* **R0**, *l'entier 24 en* **R1**, *et l'addition des deux dans* **R2**.

**Solution.**
```
MOV R0, #36
MOV R1, #24
ADD R2, R0, R1
```

**SUB** : permet d'effectuer des soustractions.

**Utilisation 1** Soustraction entre registres, par exemple **SUB R3, R1, R2** calcule **R1 - R2** et place le résultat dans **R3**.

**Utilisation 2** Soustraction en plaçant le résultat dans un des registres opérandes, par exemple **SUB R1, R1, R2** remplace la valeur de **R1** par **R1 - R2**.

**Utilisation 3** Soustraction avec une constante (opérande immédiate), par exemple **SUB R1, R2, #10** calcule **R2 - 10** et place le résultat dans **R1**.

### 2 Entre le processeur et la RAM

**LDR** : permet de récupérer une valeur dans la RAM (*LoaD Register*).

**Utilisation 1** Récupération en spécifiant une adresse, par exemple **LDR R0, 27832** lit la case mémoire (en RAM) d'adresse 278332, et la place la valeur dans **R0**.

**STR** : permet de stocker une valeur dans la RAM (*STore Register*).

**Utilisation 1** Stockage en spécifiant une adresse, par exemple **STR R2, 3644** place la valeur de **R2** dans la case mémoire (en RAM) d'adresse 3644.

**Exercice 3.2** *Incrémentez de 1 la valeur contenue dans la RAM à l'adresse mémoire 60504.*

**Solution.**
```
LDR R1, 60504
ADD R1, R1, #1
STR R1, 60504
```

**Exercice 3.3** *Ajoutez les valeurs contenues dans la RAM aux adresses 36200 et 36204, et placez le résultat à l'adresse 36208.*

**Solution.**
```
LDR R1, 36200
LDR R2, 36204
ADD R1, R1, R2
STR R1, 36208
```

## III L'adressage indirect

Si un certain registre contient une adresse dans la RAM, alors on peut l'utiliser dans **LDR** et **STR**. On appelle cela **l'adressage indirect**. Par exemple, si **R1** contient une adresse dans la RAM, alors l'instruction **LDR R0, [R1]** permet d'aller récupérer la valeur stockée à cette adresse, et de la copier dans **R0**.

⚠**Erreur fréquente :** Écrire **LDR R0, R1** n'a pas de sens puisque le second paramètre de **LDR** (et **STR**) doit être une adresse et non un registre.

De même, si **R1** contient une adresse dans la RAM, alors l'instruction **STR R0, [R1]** permet de stocker la valeur dans le registre **R0** à l'adresse contenue dans **R1**.

**Exercice 3.4** *Écrire la suite d'instructions permettant d'incrémenter de 5 l'entier stocké à l'adresse RAM 36152, en utilisant l'adressage indirect. On pourra d'abord placer l'entier 36152 dans un registre.*

**Solution.**
```
MOV R0, #36152
LDR R1, [R0]
ADD R1, R1, #5
```

```
STR R1, [R0]
```

**Remarque 3.2** *En pratique, on ne connaît pas la vraie adresse des variables avant l'exécution, on ne rencontrera ainsi jamais ce genre de code.*

Nous allons voir maintenant comment définir des variables locales et comment récupérer leur adresse.

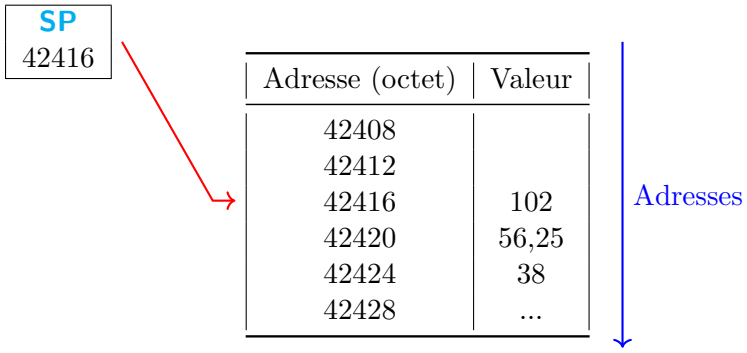
# IV La pile

On rappelle que la pile est l'espace mémoire d'un processus qui permet de stocker ses variables locales, les paramètres de fonctions et les adresses de retour des fonctions. On veut être capable de faire deux opérations sur cette pile :

**Empiler ou Push** placer une nouvelle valeur en haut de la pile ;

**Dépiler ou Pop** retirer la valeur du haut de la pile.

Dans une architecture ARM, la pile est stockée dans un espace contiguë de la RAM. Ainsi, les adresses des cases se suivent, et quand on empile, les adresses diminuent. Le registre **SP** du processeur contient l'adresse de la dernière valeur ajoutée dans la pile.




Lorsqu'on déclare une variable en C (par exemple un `int`), on décroît la valeur de **SP** du nombre nécessaire d'octets (4 car un `int` est codé sur 32 bits).

| code C | assembleur |
|---|---|
| `int x;` | **SUB, SP, SP, #4** |

Dans le reste du code, on sait que la valeur de notre variable sera stockée en RAM à l'adresse qui contenue dans le registre **SP**.

**Exercice 3.5** *Traduire en assembleur le bout de code C suivant :*

```
1 int x ;
2 x= 33 ;
```

**Solution.**

```
SUB SP, SP, #4
MOV R0, #33
STR R0, [SP]
```

Ainsi, si on déclare deux entiers, il faudra décroître **SP** de 8, si on déclare trois entiers, il faudra décroître **SP** de 12, etc.

**Remarque 3.3** *Le compilateur analyse la totalité de la fonction et toutes les déclarations de variables locales y figurent, afin d'allouer la quantité nécessaire de mémoire dans la pile au tout début de la fonction. Par exemple, si des variables ne sont déclarées qu'à l'intérieur d'un* `if`*, leur espace est quand même alloué, même si le code n'est jamais exécuté. De même, si une variable est déclarée à l'intérieur d'une boucle, la pile n'est agrandie qu'au début de la fonction, et non pas à chaque tour de boucle.*

# V Adressage indirect avec décalage

Lors de l'adressage indirect, on suppose qu'un registre contient comme valeur une adresse mémoire, et c'est à cette adresse que l'on veut charger ou stocker une valeur. Il est aussi possible de spécifier un décalage de l'adresse stockée dans ce registre (typiquement le Stack Pointer). Par exemple, si **R1** contient une adresse dans la RAM, alors l'instruction **LDR R0, [R1 #x]** permet d'aller récupérer la valeur stockée à cette adresse auquel on ajoute $x$ octets, et de la copier dans **R0**. Le fonctionnement pour l'instruction **STR R0, [R1 #x]** est similaire.

Par exemple, on considère la situation suivante :

Processeur

| R0 | R1 |
|---|---|
| 33 | 8616 |

RAM

| Adresse (en octets) | Valeur |
|---|---|
| ... | ... |
| 8624 | 28 |
| 8620 | 56,25 |
| 8616 | 102 |
| 8612 | 45 |
| 8608 | 8 |
| ... | .. |

Adresses

et on exécute l'instruction **STR R0, [R1 #4]**, la situation devient :

Processeur

| R0 | R1 |
|---|---|
| 33 | 8616 |

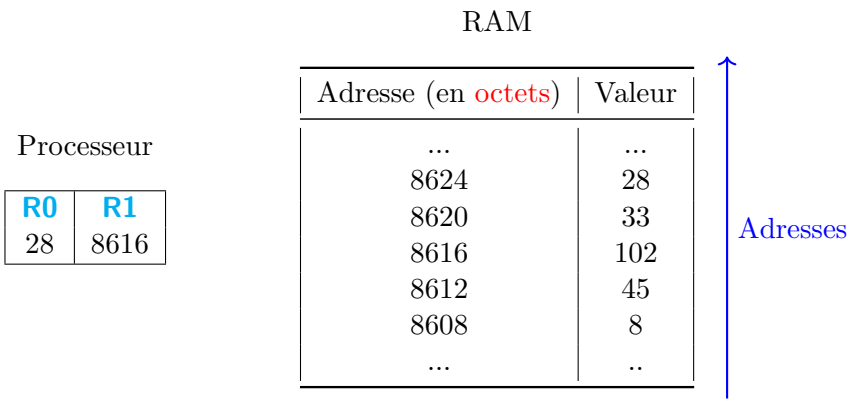
RAM

| Adresse (en octets) | Valeur |
|---|---|
| ... | ... |
| 8624 | 28 |
| 8620 | 33 |
| 8616 | 102 |
| 8612 | 45 |
| 8608 | 8 |
| ... | .. |

Adresses

La valeur contenue dans **R0**, c'est-à-dire l'entier 33, a été chargé dans la case mémoire d'adresse `[R1]` (c'est-à-dire 8616), auquel on ajoute 4 octets (donc 8620).

Pour continuer l'exemple, après l'instruction **LDR R0, [R1 #8]**, la situation devient :




La valeur de la case mémoire d'adresse **[R1]** plus 8 octets, c'est-à-dire l'adresse 8624 et la valeur 28, a été chargé dans le registre **R0**.

**Exercice 3.6** *Traduire en assembleur le bout de code C suivant :*

```
1 int x,y,z;
2 x=3;
3 y=5;
4 z=x+y;
```

**Solution.** L'idée générale consiste à diminuer de 12 octets l'adresse du stack pointer **SP** pour indiquer qu'on empile trois entiers $x$, $y$ et $z$ (chacun sur 4 octets). Cela se fait via l'instruction **SUB SP SP 12**. Ensuite, on peut modifier les valeurs de $x$, $y$ et $z$ en utilisant l'adressage indirect avec décalage. on pensera à traduire chaque instruction C indépendamment, en supposant qu'on oublie le contenue de chaque registre entre chaque instruction.

**SUB SP, SP, 12** valeur de $z$ à **SP**, celle de $y$ à **SP+4** et celle de $x$ à **SP+8**

**MOV R0 #3** on place la constante 3 dans **R0**
**STR R0, [SP #8]** on stocke la valeur contenue dans **R0** dans la zone mémoire de $x$

**MOV R0 #5**
**STR R0, [SP #4]**

**LDR R0, [SP #8]** on place la valeur de $x$ dans **R0**
**LDR R1, [SP #4]** on place la valeur de $y$ dans **R1**
**ADD R2, R0, R1** on place la valeur de $x + y$ dans **R2**
**STR R2, [SP]** on stocke cette valeur dans la zone mémoire de $z$

**Exercice 3.7** *Traduire en assembleur le bout de code C suivant :*

```
1 int a,b,c;
2 a=11;
3 b=a-22;
4 c=33;
5 int d;
6 d= b + 44 + c + a;
```

**Solution.**
**SUB SP, SP, 16** zone de $d$ à **SP**, zone de $c$ à **SP+4**, zone de $b$ à **SP+8** et zone de $a$ à **SP+12**

**MOV R0, #11**
**STR R0, [SP #12]**

```
LDR R0, [SP #12]
SUB R0, R0, #22
STR R0, [SP #8]

MOV R0, #33
STR R0, [SP #4]

LDR R0, [SP #8]
ADD R1, R0, #44
LDR R0, [SP #4]
ADD R1, R1, R0
LDR R0, [SP #12]
ADD R1, R1, R0
STR R1, [SP]
```

Remarquons qu'on voit qu'on peut faire une addition de taille arbitraire avec seulement un nombre constant de registre.

**Exercice 3.8** *Traduire en assembleur le bout de code C suivant :*

```c
int x;
int* px;

x=33;
px = &x;
*px=44;
```

**Solution.**

**SUB SP, SP, #8** zone de $px$ à **SP** et zone de $x$ à **SP+4**

```
MOV R0, #33
STR R0, [SP #4]

MOV R0, SP
ADD R0, R0, #4
STR R0, [SP]

MOV R0, #44
LDR R1, [SP]
STR R0, [R1]
```

**MOV R0, SP** on récupère l'adresse du stack pointer dans **R0**
**ADD R0, R0, #4** **R0** contient l'adresse de la zone de $x$
**STR R0, [SP]** on stocke cette adresse dans la zone de $px$

**MOV R0, #44**
**LDR R1, [SP]** on charge la valeur de $px$ dans **R1**, qui est l'adresse de $x$
**STR R0, [R1]** on stocke la valeur de **R0** (44) dans la zone mémoire d'adresse `[R1]`, qui est exactement l'adresse de la zone de $x$

# VI Les conditions (`if-else`)

On ne traite ici que les cas où les conditions sont des expressions arithmétiques de base (égalité ou inégalités). Pour écrire un `if-else` en assembleur, nous avons besoin de deux instructions supplémentaires : une instruction de compraison, et des instructions de saut.

## A Instruction de comparaison

L'instruction **CMP** permet de faire une comparaison et de modifier des « flags » selon le résultat.

**Utilisation 1** Comparaison de deux registres, par exemple **CMP R1, R2** compare la valeur du registre **R1** avec celle du registre **R2**.

**utilisation 2** Comparaison avec une constante, par exemple **CMP R1, #4** compare la valeur du registre **R1** avec la constante 4.

Les "flags", qui contiennent les informations relatives au résultat de la comparaison telles que le résultat ou la détection d'un débordement, sont contenus dans le registre **CPSR** (*Current Program Status Register*).

## B Instructions de saut.

Ces instructions permettent de se déplacer à un autre endroit du code assembleur selon la valeur des flags de comparaison. Un « autre endroit du code » est désigné par une **étiquette**. Une étiquette est une chaîne de caractère placée avant une instruction. Celle-ci sera, au début de l'exécution du programme, remplacée par **l'adresse mémoire** de l'instruction suivante partout où elle apparaît. Par exemple, pour définir une étiquette, on écrit :

```
mon-étiquette :
MOV R0, R1
```

Pour utiliser ces étiquettes, nous utiliserons une instruction de saut. On distingue les sauts **conditionnels** (qui dépendent du résultat d'une comparaison par exemple) et les sauts **inconditionnels** (qui ont lieu dans tous les cas).

**Saut inconditionnel.** Un saut **inconditionnel** se fait grâce à l'instruction **BRA** (pour *BRAnch*), suivie d'une étiquette :

```
BRA mon-étiquette
```

Lorsque cette instruction est exécutée, l'unité de commande du processeur place l'adresse de l'instruction située après l'étiquette **mon-étiquette** dans le registre `PC`. Ainsi, la prochaine instruction exécutée sera celle située après l'étiquette. Ce mécanisme ressemble aux « goto » dans le langage C.

**Saut conditionnel.** Un instruction de saut conditionnel permet de faire un saut selon le résultat de la **dernière comparaison**. Supposons que la dernière invocation de **CMP** est **CMP R0,R1**. On distingue six instructions de saut conditionnel :

| Nom | Acronyme | Signification |
|---|---|---|
| **BEQ** | Branch if EQual | **R0=R1** |
| **BNE** | Branch if Not Equal | **R0≠ R1** |
| **BLE** | Branch if Less or Equal | **R0≤ R1** |
| **BLT** | Branch if Less Than | **R0< R1** |
| **BGE** | Branch if Greater or Equal | **R0≥ R1** |
| **BNE** | Branch if Greater Than | **R0> R1** |

**Exemple 3.1** *Considérons le code suivant :*

```
    BLT toto
    MOV R0, #8

toto :
    MOV R0, #6
```

*L'instruction* **BLT toto** *permet de sauter directement à* **toto** *si dans la dernière invocation de* **CMP**, *le premier paramètre était strictement inférieur au second. Si c'est le cas, l'instruction* **MOV R0, #8** *est ainsi sautée et on exécutera directement l'instruction* **MOV R0, #6**.

**Exercice 3.9** *Traduire en assembleur le bout de code C suivant :*

```
int x,y;

if (x<10){
        y=44;
}
x=77;
```

**Solution.**

**SUB SP, SP, #8** on alloue l'espace pour x et y

**LDR R0, [SP #4]** R0 contient la valeur de $x$

**CMP R0, #10** la valeur de x est comparée à la constante 10
**BGE fin-if** on va à l'étiquette fin-if si $x \geq 10$

**MOV R0, #44**
**STR R0, [SP]** on charge 44 à l'adresse mémoire de y

**fin-if :**

**MOV R0, #77**
**STR R0, [SP #4]** on charge 77 à l'adresse mémoire de x L'idée est de sauter directement à l'étiquette **fin-if** si le test « x¡10 » est faux, c'est-à-dire $x \geq 10$. On utilise pour cela l'instruction BGE.

**Remarque 3.4** *Il est de donné pratique de mettre des noms d'étiquettes qui ont un sens :* **fin-if**, **else**, **fin-for**, *etc. En pratique, le compilateur a un compteur et les étiquettes sont L1, L2, L3, etc (L pour « Label »).*

Pour implémenter les *else*, on utilisera l'instruction **BRA** vue précédemment.

**Exercice 3.10** *Traduire en assembleur le bout de code C suivant :*

```
int x,y;

if (x!=y){
        y=44;
} else {
        y = 66;
}
x=77;
```

**Solution.**

**SUB SP, SP, #8** on alloue l'espace pour x et y

**LDR R0, [SP #4]** R0 contient la valeur de x
**LDR R1, [SP ]** R1 contient la valeur de y

**CMP R0, R1** la valeur de x est comparée à la valeur de y
**BEQ else** on va à l'étiquette au else si x=y

**MOV R0, #44**
**STR R0, [SP]** on charge 44 à l'adresse mémoire de y

**BRA fin-if** on saute à la fin du if

**else :**
**MOV R0, #66**
**STR R0, [SP]** on charge 66 à l'adresse mémoire de y
**fin-if :**
**MOV R0, #77**
**STR R0, [SP #4]** on charge 77 à l'adresse mémoire de x

# VII Les boucles

En C, une boucle while suit généralement le schéma suivant :

```
\\ Bloc A
while (Condition) {
        \\ Bloc B
}
Bloc C
```

En assembleur, cela donnera :
instructions Bloc A

**début-while :**
instructions pour réaliser la comparaison avec **CMP**
**Bxx fin-while** xx correspond à la négation de la condition
instructions Bloc B
**BRA début-while**

**fin-while :**
instructions Bloc C

**Exercice 3.11** *Traduire en assembleur le bout de code C suivant :*

```
int x, y, z;
x = 0;
while (x<100){
        y = y + z;
        x += 3 ;
}
z = y + 9 ;
```

**Solution.**
**SUB SP, SP, #12** on alloue l'espace pour x, y, z

**MOV R0, #0**
**STR R0, [SP #8]** on charge 0 à l'adresse mémoire de x

**début-while :**
**LDR R0, [SP, #8]** on charge la valeur de x dans R0
**CMP R0, #100** comparaison de x avec 100
**BGE fin-while**
**LDR R1, [SP, #4]** on charge la valeur de y dans R1
**LDR R2, [SP]** on charge la valeur de z dans R2
**ADD R1, R1, R2**

**STR R1, [SP, #4]** on charge la valeur de R1 à l'adresse de y
**LDR R0, [SP, #8]** on charge la valeur de x dans R0
**ADD R0, R0, #3**
**STR R0, [SP, #8]** on charge la de R0 à l'adresse de x
**BRA début-while**

**fin-while :**
**LDR R0, [SP #4]**
**ADD R0, R0, #9**
**STR R0, [SP]**

En C, une boucle for suit généralement le schéma suivant :

```
\\ Bloc A
for (Instruction 1 ; Condition ; Instruction 2) {
        \\ Bloc B
}
Bloc C
```

On remarque qu'on peut transformer une telle boucle en une boucle while :

```
\\ Bloc A
\\ Instruction 1
while (Condition) {
        \\ Bloc B
        \\ Instruction 2
}
Bloc C
```

Ainsi, quand on veut traduire une boucle for en assembleur, on commence d'abord par la transformer en boucle while.

**Exercice 3.12** *Traduire en assembleur le bout de code C suivant :*

```
int x = 8;

for (int i = 0 ; i !=1000 ; i++){
        x = x + i ;
}
int z = x + 3 ;
```

**Solution.** Avec une boucle while, on obtient le code C suivant :

```
int x = 8;
int i = 0;
while (i!= 1000){
        x = x + i;
        i++;
}
int z = x + 3 ;
```

On obtient le code assembleur suivant.

**SUB SP, SP, #12** on alloue l'espace pour x, i, z

**MOV R0, #8**
**STR R0, [SP #8]** on charge 8 à l'adresse mémoire de x
**MOV R0, #0**
**STR R0, [SP #4]** on charge 0 à l'adresse mémoire de i

**début-for :**
**LDR R0, [SP, #8]** on charge la valeur de x dans R0
**CMP R0, #100** comparaison de x avec 100
**BEQ fin-for**
**LDR R0, [SP, #8]** on charge la valeur de x dans R0
**LDR R1, [SP, #4]** on charge la valeur de i dans R1
**ADD R0, R0, R1**
**STR R0, [SP, #8]** on charge la valeur de R0 à l'adresse de x
**LDR R0, [SP, #4]** on charge la valeur de i dans R0
**ADD R0, R0, #1**
**STR R0, [SP, #4]** on charge la valeur de R0 à l'adresse de x
**BRA début-for**

**fin-for :**
**LDR R0, [SP #8]**
**LDR R1, [SP]**
**ADD R1, R0, #3**
**STR R1, [SP]**

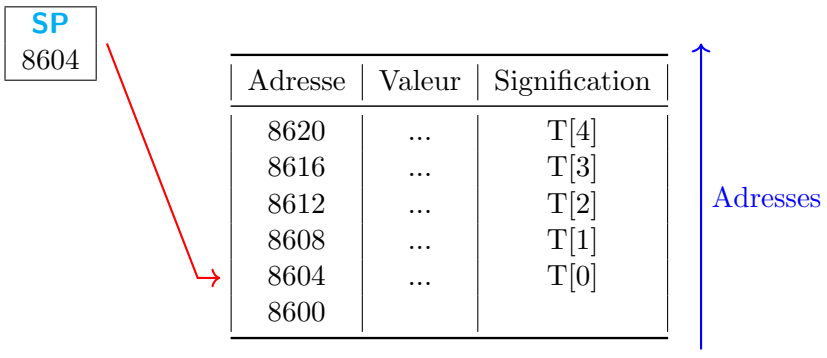
# VIII Les tableaux

Nous traiterons ici que de tableaux statiques : on connaît leur taille à la compilation. De tels tableaux sont entièrement stockés sur la pile. Pour récupérer/stocker les valeurs d'un tableau, on distinguera selon si la valeur de l'indice est une constante, ou bien stockée dans une variable et/ou le résultat d'un calcul impliquant des variables.

**Indices constants.** On utilisera l'adressage indirect comme vu précédemment. Si on déclare un tableau de 5 entiers

```
1 int T[5];
```

Alors en assembleur, on augmentera la pile de $5 \times 4 = 20$ octets via l'instruction **SUB SP, SP, # 20**. L'adresse de $T[0]$ sera à **SP**, $T[1]$ sera à **SP+4**, etc.




**Exercice 3.13** *Traduire le code C suivant :*

```
1 int T[10] ;
2
3 T[3] = 5 ;
4 T[0] = 9 ;
5 T[1] = T[9] ;
```

**Solution.**

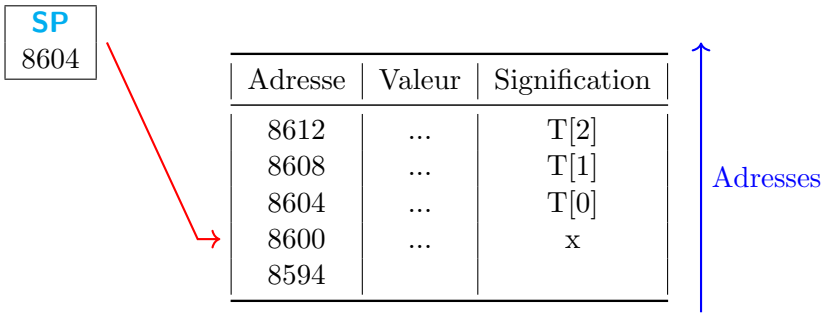
```
SUB SP, SP, #20     T[0] est à SP, T[1] est à SP+4, etc

MOV R0, #5
STR R0, [SP #12]
MOV R0, #9
STR R0, [SP]
MOV R0, #5
LDR R0, [SP #36]
STR R0, [SP #4]
```

Dans le cas où il y a plusieurs variables locales, il faudra adapter le "saut" dans l'adressage indirect. Par exemple, dans le code suivant :

```
1 int T[3];
2 int x;
```

Il faudra agrandir la pile en fonction : **SUB SP, SP, #16**. On réserve $3 \times 4 = 12$ octets pour le tableau, et 4 octets en plus pour x. De la même manière que plusieurs variables locales sont définies, x sera empilé en dernier :




**Exercice 3.14** *Traduire le code C suivant :*

```
1 int a ;
2 int T[5] ;
3 int b, c ;
4
5 T[0] = a + b ;
6 c = T[3] + T[4];
```

**Solution.** On alloue sur la pile $5 \times 4 = 20$ octets pour le tableau T, et 12 octets pour a,b et c.

```
SUB SP, SP, #32     c est à SP, b est à SP+4, T[0] est à SP + 8, T[1] est à SP+12
                    T[2] est à SP+16, T[3] est à SP+20, T[4] est à SP+24, a est à SP+28
LDR R0, [SP #28]
LDR R1, [SP #4]
ADD R0, R0, R1
STR R0 [SP #8]

LDR R0, [SP #20]
LDR R1, [SP #24]
ADD R0, R0, R1
STR R0 [SP]
```

**Indices non constants.** Pour lire/stocker une case dont l'indice est contenu dans une variable ou un calcul impliquant des variables, par exemple :

```
1 T[x]=20;
```

Il faudra d'abord stocker la valeur de x dans un registre, par exemple **R1**. Ensuite, pour accéder à `T[x]`, il faut se décaler de 4x octets par rapport au début du tableau. Pour cela, on utilisera l'instruction **LSL** pour *Logical Shift Left* qui permet de faire un décalage logique à gauche. Par exemple, l'instruction

**LSL R0, R1, #2**

prend les 32 bits situés dans **R1**, leur applique un décalage de 2 bits vers la gauche, et place le résultat dans **R0** (cela ne modifie pas **R1**). Remarquons que décaler un nombre binaire de $i$ bits vers la gauche revient à multiplier ce nombre par $2^i$.

| Binaire | Décimal |
|---|---|
| 1 0 0 1 1 | 19 (16+2+1) |
| 1 0 0 1 1 0 | 38 (32+4+2) |
| 1 0 0 1 1 0 0 | 76 (64+8+4) |

C'est la raison pour laquelle les tailles de chaque type sont des puissances de 2 (int, char, float sur 4 octets, double sur 8 octets) : il est plus simple de faire un décalage de bits qu'une multiplication. L'instruction **LSL** s'utilise similairement à **ADD** et **SUB** : le troisième paramètre peut être un registre ou une constante, et le premier et le second paramètre peuvent être le même registre.

Dans le cas où il y a plus d'une variable locale, on chargera d'abord l'adresse de T[0] dans un registre, et on avancera ensuite le nombre qu'il faut à l'aide de **LSL**.

On utilisera aussi l'**adressage indirect avec registre** pour les instructions **LDR** et **STR**. Par exemple, l'instruction

**LDR** R0, [R1, R2]

permet de récupérer en mémoire la valeur à l'adresse **[R1]**+**[R2]** et de la mettre dans R0.

**Exercice 3.15** *Traduire le code C suivant :*

```
1 int T[100];
2 int x = 33;
3 T[x]= T[x+8] ;
```

**Solution.**

```
SUB SP, SP, #404     T[99] est à SP+400, T[0] à SP+4, x à SP

MOV R0, #33
STR R0, [SP]

LDR R0, [SP]     R0 contient x
ADD R0, R0, #8     R0 contient x+8
LSL R0, R0, #2     R0 contient 4(x+8)
MOV R1, SP
ADD R1, R1, #4     R1 contient l'adresse de T
LDR R0, [R1, R0]     R0 contient T[x+8]
LDR R1, [SP]     R1 contient x
LSL R1, R1, #2     R1 contient 4x
ADD R2, SP, #4     R2 contient l'adresse de T
```

**STR R0, [R2, R1]** T[x]=T[x+8]

# IX Les appels de fonctions

## A Appeler une fonction

Une fonction sera repérée par une étiquette, qui portera son nom, et sera placée juste avant son code assembleur. Lorsqu'on appelle une fonction, on sautera alors vers son étiquette. Pour cela, on utilisera l'instruction **BL** (*Branch and Link*) : **BL foo** où **foo** est une étiquette. Cette instruction prend une étiquette en paramètre, et consiste en deux parties :

1. affecter à PC l'adresse de l'instruction correspondant à l'étiquette (c'est le « saut ») ;
2. affecter à LR l'adresse de l'instruction suivant l'instruction BL. De cette manière, la fonction appelée connaît l'endroit où il faut revenir lorsqu'elle se termine.

Avant d'appeler une fonction, on place les éventuels paramètres dans les registres **R0**, **R1**, **R2** et **R3**. S'il y en a plus que quatre, il faut les placer sur la pile (nous ne verrons pas cela dans ce cours).

Apprès l'appel de fonction, l'instruction suivant l'instruction **BL** est exécutée. Par convention, la valeur de retour de cette fonction est placée dans le registre **R0**.

**Remarque 3.5** *Par convention, (PCS :* Procedure Call Standard*), la fonction appelante est assurée que les valeurs courantes des registres* **R4**,..., **R11** *seront préservées après l'appel de la fonction. Ce n'est en revanche pas le cas des registres* **R0**, **R1**, **R2** *et* **R3**.

**Exercice 3.16** *On suppose que la fonction* `foo` *est déjà écrite quelque part : c'est une fonction qui prend en entrée deux entiers, et retourne un entier. Traduisez en assembleur ARM le bout de code C suivant :*

```
1 int x ;
2 x = foo (22, 33) + 44 ;
```

**Solution.**

**SUB SP, SP, #4**

**MOV R0, 22**
**MOV R1, 33**
les paramètres sont dans **R0** et **R1**
**BL foo** Branch and Link vers foo
**ADD R0, R0, #44**
**STR R0, [SP]**

## B Définir une fonction

Le code assembleur d'une fonction est précédée par son étiquette. Ensuite, nous trouvons, dans l'ordre :

1. un ensemble d'instructions appelé le *préambule* ;
2. l'ensemble des instructions de la fonction à proprement parler ;
3. un ensemble d'instructions appelé le *prologue*.

Le préambule consiste à :

— sauvegarder le registre LR dans la pile ;

— éventuellement sauvegarder les registres **R4**, ..., **R11** dans la pile s'ils sont utilisés dans le reste de la fonction ( cf Remarque 3.5) ;

— réserver la place nécessaire dans la pile pour les paramètres et les variables locales ;
— sauvegarder les paramètres (situées dans les registres **R0**, **R1**, **R2** et **R3**) dans la pile.

Le prologue consiste à :

— supprimer de la pile les variables locales et les paramètres (on augmente pour cela la valeur de SP) ;
— rétablir éventuellement les valeurs des registres **R4**,..., **R11** s'ils avaient été sauvegardés précédemment ;
— récupérer la valeur du registre LR sauvegardé précédemment pour le placer dans le registre PC.

Pour sauvegarder des registres dans la pile, il faut augmenter la taille de celle-ci (avec **SUB**), et copier les données (avec **STR**). Une instruction permet de faire cela en une fois : **PUSH**. Par exemple,

**PUSH** {**R0, R1, R2**}

permet de stocker la valeur des registres **R0**, **R1** et **R2** directement sur la pile, en augmentant automatiquement SP.

L'opération inverse est possible : **POP**. Par exemple,

**POP** {**R0, R1**}

place la valeur située en haut de la pile dans **R1**, celle située juste en dessous dans **R0**, et ajuste automatiquement la valeur de SP.

Ainsi, pour la dernière tâche du prologue qui consiste à récupérer LR sauvegardée en préambule, et la placer dans PC, on pourra directement écrire **POP** {**PC**}.

**Exercice 3.17** *Traduire le code C suivant :*

```c
int foo(int x, int y){
        int z = x + y + 88 ;
        return z ;
}
```

**Solution.**
**foo :**

```
PUSH {LR}
SUB SP, SP, #12     z est à SP, y à SP+4 et x à SP+8
STR R0, [SP, #8]    on place l'argument x à l'adresse SP+8
STR R1, [SP, #4]    on place l'argument y à l'adresse SP+4

LDR R0, [SP, #8]
LDR R1, [SP, #4]
ADD R0, R0, R1
ADD R0, R0, #88
STR R0, [SP]
LDR R0, [SP]        on met la valeur de retour dans R0

ADD SP, SP, #12     on supprime les variables locales et paramètres
POP {PC}            on dépile LR dans la pile et on le met dans PC
```

**Exercice 3.18** *Traduire le code C suivant :*

```c
int fibo(int n){
        if (n==0)
                return 0 ;
        if (n==1)
                return 1 ;
        return fibo(n-1)+fibo(n-2);
}
```

**Solution.** On utilisera ici le registre **R4** afin de récupérer la valeur d'un des appels de fonction. On pensera alors à le sauvegarder dans le préambule.

```
fibo :
        PUSH {LR, PC}
        SUB SP, SP, #4     n est à SP
        STR R0, [SP]       on place l'argument n à l'adresse SP

        LDR R0, [SP]
        CMP R0,#0
        BNE fin-if-1
        MOV R0, #0
        BRA fin-fibo

fin-if-1 :
        LDR R0, [SP]
        CMP R0, #1
        BNE fin-if-2
        MOV R0, #1
        BRA fin-fibo

fin-if-2 :
        LDR R0, [SP]
        SUB R0, R0, #2     R0 contient n-2
        BL fibo     appel de fibo(n-2)
        MOV R4, R0     R4 contient fibo(n-2)
        LDR R0, [SP]
        SUB R0, R0, #1     R0 contient n-1
        BL fibo     appel de fibo(n-1)
        ADD R0, R0, R4     R0 contient fibo(n-1)+fibo(n-2)

fin-fibo :
        ADD SP, SP, #4     on supprime les variables locales et paramètres
        POP {PC, R4}     on dépile R4 et LR dans la pile et on le met dans R4 et PC
```