La mémoire cache expliquée

I. Introduction

Ce document présente les bases permettant de comprendre la notion de *cache mémoire*. Il en explique la nécessité, le fonctionnement, les différentes solutions possibles, les problèmes afférents et les solutions adoptées dans la pratique.

A. Généralisation de la notion de cache

La notion de "cache" est fréquemment utilisée pour accélérer l'accès à des données disponibles sur un système dont le temps d'accès est trop long pour le dispositif les lisant, soit parce qu'il est intrinsèquement lent, soit parce qu'il se situe à une grande distance.

- Comme nous allons le voir plus loin, le cache peut s'insérer entre le microprocesseur et la mémoire centrale :

  Les données du cache sont en général composées de "blocs" de 64 mots consécutifs de la mémoire centrale.

- Un cache est souvent inséré entre l'unité centrale et les mémoires permanentes (disques durs, SSD, mémoire Flash...) beaucoup plus lentes :

  Les données du cache sont alors des morceaux de fichiers ou de mémoire virtuelles, appelées "pages", de quelques kilo-octets.

- Les navigateurs modernes comportent un cache permettant de sauvegarder les dernières pages web consultées :

  Dans ce cas et le suivant, les données stockées par le cache sont des pages web.

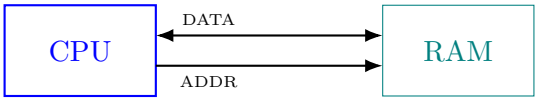
- De même, un cache (un serveur "proxy", par exemple) peut être inséré entre un ou plusieurs ordinateurs en réseau et l'Internet, de manière à conserver les pages souvent accédées :

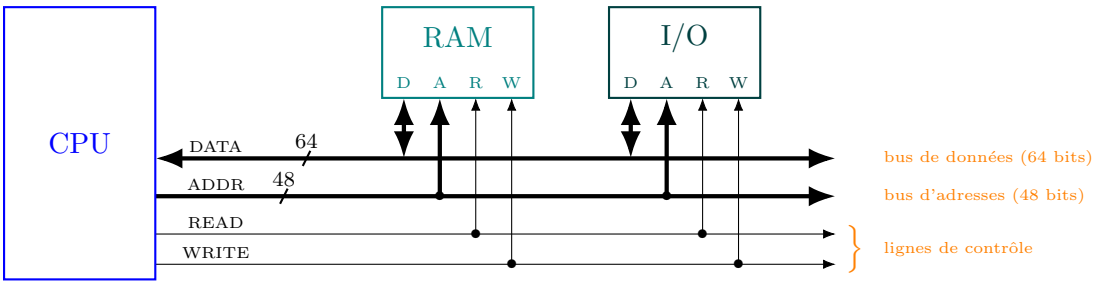
Dans la suite de ce document, nous allons expliquer le fonctionnement du cache entre le microprocesseur et la mémoire principale d'un ordinateur. Nous commencerons par des généralités sur le mode de fonctionnement de l'unité centrale, puis nous détaillerons les principes permettant d'insérer de manière transparente un dispositif accélérant les transferts entre ces deux entités. Nous détaillerons au fur et à mesure les principes utilisés, leurs avantages et inconvénients.

B. Schéma général d'une unité centrale

Voici le schéma-bloc de tout ordinateur, épuré de tout ce qui interagit avec :



Lorsque le microprocesseur (CPU) a besoin d'une donnée, il envoie une adresse sur le bus d'adresse (ADDR), et la mémoire positionne sur le bus de donnée (DATA) la valeur présente à l'adresse donnée.
De même, lorsque le CPU a besoin d'actualiser la mémoire, pour y stocker le résultat d'un calcul par exemple, il envoie une adresse sur le bus d'adresse, ainsi qu'une valeur sur le bus de donnée, et la mémoire stocke à l'adresse donnée la valeur présente sur le bus de donnée.
Soyons un peu plus précis dans la description de ces communications :



On a représenté, en plus de la mémoire (sous la forme d'un bloc, mais il peut y en avoir plusieurs), un bloc d'entrées-sorties, possédant une plage d'adresses particulière.
Le **bus de données** est constitué de 64 fils, permettant de lire des mots de 64 bits en une seule fois. Le **bus d'adresse** est lui constitué de 48 fils, permettant un espace d'adressage de 2^{48} octets, soit 256 péta-octets.
Enfin, des **lignes de contrôle** permettent de préciser les intentions du microprocesseur. Nous en avons indiqué deux, une ligne "READ", qui indique que le processeur veut lire une donnée, et "WRITE", indiquant que le processeur veut écrire une donnée. Mais il y en a beaucoup d'autres.

- Lors d'une opération de lecture [1], le CPU écrit l'adresse sur le bus d'adresses, et active la ligne "READ". La mémoire [2] répond en écrivant sur le bus de données la donnée demandée. Elle active alors une ligne de commande "READY" (non représentée sur le schéma) indiquant que la donnée est disponible, de manière à ce que le CPU puisse la lire.
- Lors d'une opération d'écriture, le CPU écrit l'adresse sur le bus d'adresse, la donnée à écrire sur le bus de données, et active la ligne "WRITE". La mémoire lit l'adresse et stocke la donnée au bon endroit.

Le schéma ci-dessus suppose que le microprocesseur est le seul maître à bord, et qu'il a le contrôle exclusif de tout ce qui se passe sur les bus de données, d'adresse et de contrôle. Dans la pratique, ce contrôle peut parfois être transféré à d'autres dispositif, mais nous conserverons ce schéma simplifié dans la suite de ce document.

C. Vitesse relative du processeur et de la mémoire vive

Malheureusement, la mémoire est souvent beaucoup plus lente que le microprocesseur. Pour donner une idée des ordres de grandeur, le CPU fonctionne grosso-modo à la vitesse de l'horloge, donc une fraction de nanoseconde. A contrario, la mémoire actuelle demande un temps d'accès de l'ordre de quelques dizaines de nano-secondes. Ainsi, la mémoire vive classique (DRAM) fonctionne entre 50 et 100 fois plus lentement que le microprocesseur.
Ainsi, si le CPU a besoin de lire et d'écrire souvent en mémoire vive, il va passer la majeure partie de son temps à attendre que la mémoire exécute ses requêtes de lecture et d'écriture. Une équipe n'est pas plus rapide que le plus lent de ses membres.

1. de la part du processeur, puisque la RAM va, elle, écrire sur le bus.
2. ou tout autre circuit branché sur le bus d'adresse et reconnaissant une adresse de son espace réservé.

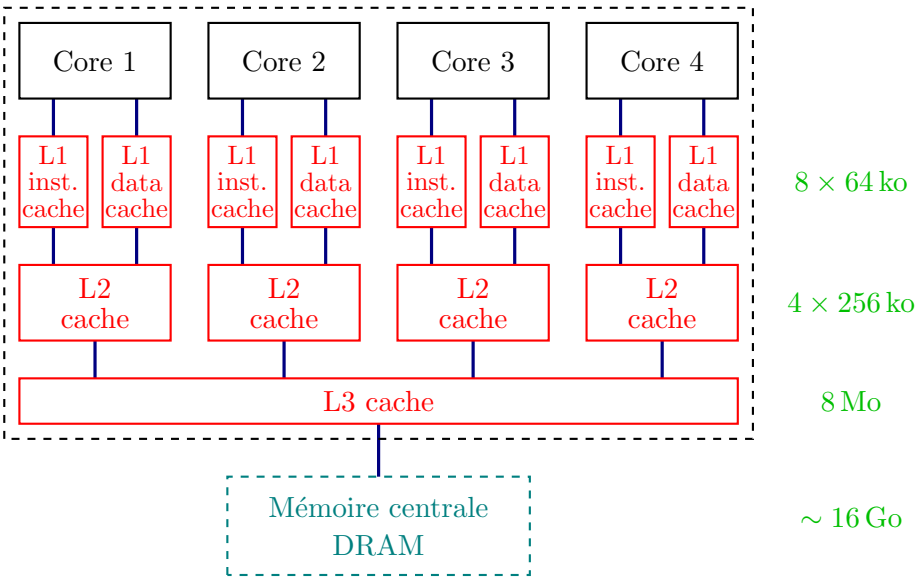
D. Introduction de la notion de cache mémoire

C'est ici qu'intervient la notion de *cache*. La mémoire cache se positionne entre le microprocesseur et la mémoire centrale, de la manière suivante :

La mémoire cache est une mémoire plus petite (entre quelques dizaines de kilo-octets et quelques méga-octets), mais beaucoup plus rapide (entre 10 et 100 fois plus rapide) que la mémoire centrale. Elle coûte beaucoup plus cher à fabriquer, et nécessite plus d'énergie [3], ce qui explique qu'on n'utilise pas les mêmes technologies pour la mémoire centrale. C'est souvent une mémoire *statique* (SRAM), nécessitant plus de transistors par bit stacké, mais ne nécessitant pas de rafraîchissement, contrairement à la mémoire centrale (DRAM), qui est de la mémoire *dynamique*, nécessitant un très petit nombre de transistors par bit stocké, mais devant être régulièrement *rafraichie* pour ne pas perdre la cohérence des donnés stockées.
Lorsque le CPU veut lire une donnée, il écrit une adresse sur le bus ADDR, qui est interceptée par le cache. Si le cache contient la donnée demandée, celle-ci est immédiatement renvoyée au CPU sur le bus de donnée. On parle de *cache hit*. Si la donnée n'est pas présente dans le cache (on parle alors de *cache miss*), celui-ci transfère l'adresse à la mémoire principale, qui renvoie la donnée demandée. Celle-ci est non seulement renvoyée au CPU, mais aussi stockée dans le cache.
On notera que dans ce schéma, du point de vue du CPU, la situation est la même que le cache soit présent ou non : il envoie des requêtes par l'intermédiaire de ses bus, et récupère éventuellement le résultat de ces requêtes sur le bus de donnée. Il en est de même de la mémoire centrale, qui reçoit des requêtes sur les bus, et renvoie les données demandées sur le bus de donnée ou stocke les données présentes.
Ainsi, le cache peut être inséré de manière complètement transparente entre le CPU et la RAM sans modification sensible, et accélère silencieusement les transferts de données entre eux.
Les concepts centraux derrière cette notion de cache sont résumés ci-dessous :

- **Certaines** données sont **importantes**, et **fréquemment utilisées**.
- La **plupart** des données sont **rarement utilisées**.
- La situation idéale consiste donc à **garder les données importantes dans le cache**.
- L'avantage principal est que **ces données fréquemment utilisées sont accessibles rapidement**.
- Les données moins importantes sont **toujours accessibles**, dans une mémoire **plus économique**.
- Du point de vue du CPU et de la RAM, le cache est **totalement transparent**.

E. Exemple d'organisation de caches dans un microprocesseur



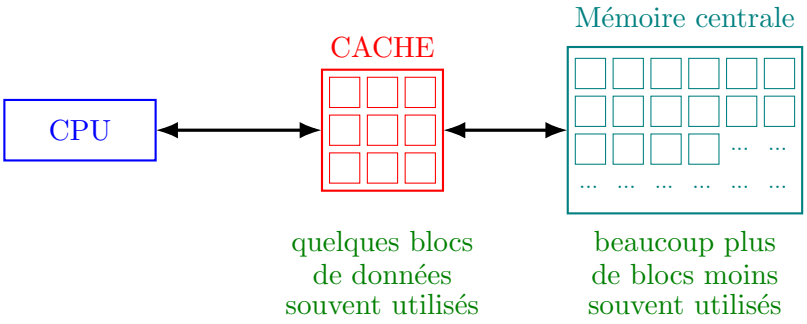
3. que ce soit de manière statique lorsqu'il s'agit simplement de conserver les données, ou de manière dynamique lorsqu'il faut lire ou écrire ces données.

Dans un microprocesseur, la mémoire cache est organisée en **plusieurs niveaux**, typiquement trois, nommés L1, L2, L3. Ces différentes mémoires caches sont constituées d'un petit nombre de blocs de 64 octets (de l'ordre de quelques dizaines de kilo-octets à quelques méga-octest), et s'insèrent hiérarchiquement entre le ou les cœurs et la mémoire centrale, dont la taille est typiquement de l'ordre de quelques dizaines de giga-octets. Ces caches sont directement gravés sur la puce du microprocesseur, de manière à être le plus proche possible, et sont constitués de mémoire statique (SRAM), plus rapide, et plus chère que la mémoire dynamique (DRAM) centrale.
Le schéma ci-dessus montre l'organisation de la mémoire cache dans l'Intel Core i7. Au niveau le plus proche des cœurs, on trouve les caches de niveau 1, deux par cœur, un pour les instructions et un pour les données, de 64 ko chacun. Le cache instruction ne nécessite pas de gérer les opérations d'écriture, parce que (sauf paradigme pathologique) le CPU ne modifie pas la liste des instructions exécutées. Il est donc plus simple à concevoir que le cache données, qui lui doit gérer les lectures et les écritures, et donc la mise à jour des données dans la RAM. Les opérations sur ces caches nécessitent de l'ordre de 4 cycles d'horloge.
On trouve ensuite pour chaque cœur un cache de niveau 2 unifié, de 256 ko. Les opérations sur ces caches nécessitent de l'ordre de 11 cycles d'horloge.
Enfin, un cache de niveau 3, beaucoup plus gros puisque de l'ordre d'une dizaine de méga-octets, est commun à tous les cœurs. Les opérations sur ce cache nécessitent de l'ordre de 30 à 40 cycles d'horloge.
À titre de comparaison, les opérations de lecture et d'écriture dans la DRAM nécessitent de 50 à 200 cycles d'horloge.

II. Concepts de base des caches

A. Concept de bloc

Le **bloc** est l'unité élémentaire de transfert entre la mémoire centrale et le cache. Typiquement (dans les processeurs Intel récent notamment), un bloc regroupe 64 octets consécutifs (soit 8 mots de 64 bits dans les CPU modernes, ou 512 bits).



Si le CPU veut récupérer un octet (par exemple) et que celui-ci est présent dans un bloc de la mémoire cache, le cache renvoie très rapidement l'octet demandé.
Par contre, si l'octet n'est pas présent dans le cache, celui-ci va charger **un bloc entier** de 64 octets depuis la mémoire principale et transmettre l'octet demandé au CPU. Cela est en général désirable, étant donné que si le CPU veut lire un octet, il lira souvent des octets qui se trouvent autour (principe de localité, voir plus loin).

B. Concepts de "cache hit", "cache miss" et "éviction"

Supposons que le CPU veuille lire une donnée depuis la mémoire (nous aborderons l'écriture plus tard). Deux choses peuvent se produire :

1. le cache contient la donnée dans un de ses blocs ; l'accès est dans ce cas très rapide, le cache n'a rien d'autre à faire que de renvoyer la donnée demandée ; on parle de **cache hit** ;
2. le cache ne contient pas la donnée dans un de ses blocs ; dans ce cas, le cache récupère le bloc contenant la donnée depuis la mémoire principale, puis transmettre la donnée au CPU ; on parle de **cache miss**.

Dans le second cas, le bloc récupéré sera stocké dans le cache, et la donnée sera alors disponible pour une prochaine lecture (ou écriture), ainsi que les données proches, qui ont de grande chances de servir dans un avenir proche.
Un problème survient : le cache ayant une capacité limitée, il va arriver un moment ou celui-ci sera plein. Pour charger un nouveau bloc, il devra **évincer** un autre bloc. Il choisira en général un bloc qui n'a pas été accédé depuis un long moment.

C. Notion d'"ensemble de travail"

Un programme exécute généralement un grand nombre d'instructions (dans le cas contraire, la notion de rapidité a peu de sens). De plus, les mêmes instructions ont tendance à être exécutées de nombreuses fois, par exemple dans des boucles. La plupart des autres instructions ne sont exécutées qu'un petit nombre de fois.
Ces quelques instructions exécutées un grand nombre de fois (les "corps de boucles") sont les "points chauds" du programme, c'est dans ces instructions qu'il va passer la majeure partie de son temps d'exécution.
Voici un exemple typique :

```
for (i=0, i<N, i++) {
   for (j=0, j<M, j++ ) {
      ... Corps de boucle ...
   }
}
```
Point chaud

Les instructions à l'intérieur du "point chaud", à savoir les quelques instructions dans le corps de boucle, ainsi que le test `j<M` et l'incrémentation `j++`, sont exécutées un grand nombre de fois. On espère que ces quelques instructions sont exécutées le plus rapidement possible.
On peut supposer que le nombre d'instructions est relativement petit,. Il est donc intéressant d'avoir un **cache suffisamment large** pour contenir **l'intégralité de ces instructions** (dans le *cache d'instructions*).
On parle d'**ensemble de travail** (working set en anglais).
Après un moment, les choses changent, l'ensemble de travail se déplace et ces instructions peuvent être retirées du cache d'instruction.
Il en est de même pour les variables :

```
for (i=0, i<N, i++) {
   for (j=0, j<M, j++ ) {
      ... sum += ...
   }
}
```
Point chaud

La variable `sum` est modifiée à chaque passage dans la boucle, y étant lue et écrite, il est donc important qu'elle réside dans le **cache instruction** au moment où ce point chaud est exécuté. Cette variable fait elle aussi partie de l'ensemble de travail, au moins pendant un certain temps.
On résume tout ceci de la manière suivante :

Ensemble de travail

L'ensemble de travail est l'ensemble des instructions et des données utilisées le plus souvent lors de l'exécution d'une partie du code du programme.

D. Principe de localisation

La plupart des programmes (à l'exception de quelques cas pathologiques) se comporte de manière standard, et vérifient les hypothèses suivantes :

Principe de localisation

L'ensemble de travail vérifie les hypothèses suivantes :

- **Localisation temporelle** : si un octet a été récemment utilisé, il est probable qu'il sera prochainement réutilisé.
- **Localisation spatiale** : si un octet a été récemment utilisé, il est probable que les octets proches dans la mémoire seront prochainement utilisés.

Ceci est valable à la fois pour les données et les instructions.

On peut déduire de ceci les deux redéfinitions suivantes de l'ensemble de travail :

- c'est l'ensemble des octets **qui ont été utilisés récemment** ;
- c'est l'ensemble des octets **dont on aura besoin prochainement**.

La deuxième définition serait préférable à la première (cache "prédictif"), mais on ne connaît en général pas complètement cet ensemble. Peu importe, pour la plupart des programmes, **ces deux définitions sont très proches l'une de l'autre**.
Tout cela conduit à l'importante conséquence suivante :

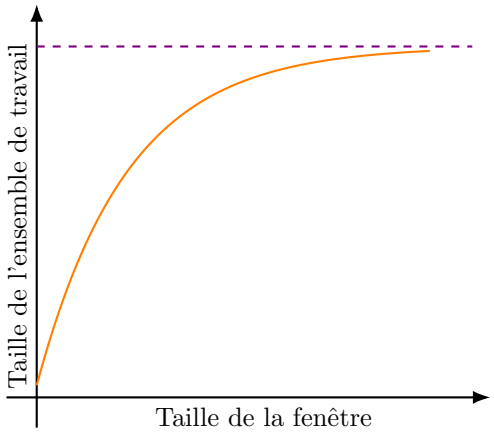
Si l'ensemble de travail peut être conservé dans son intégralité dans le cache, le programme s'éxécutera plus vite !

Déduisons de ceci une stratégie d'intégration des données et des instructions dans le cache :

- quand on insère des octets dans le cache, **on les conserve** ;
- on essaye de garder les octets **le plus longtemps possible** dans le cache, mais on préfère garder les octets **les plus récemment utilisés** ;
- quand on insère des octets dans le cache, **on insère aussi les octets proches** (d'où le concept de *bloc* de 64 octets).

Regardons plus précisément le déroulement d'un programme :

On s'intéresse aux octets (instructions et données) exécutées dans la fenêtre de temps encadrée. C'est l'ensemble de travail. Si ces octets se trouvent dans le cache, alors l'exécution sera rapide (car il n'y aura pas de "cache miss").
La taille de l'ensemble de travail dépend de la largeur de la fenêtre. Plus la fenêtre est large, plus grande sera la proportion de code exécutée, mais plus grand sera l'ensemble de travail.



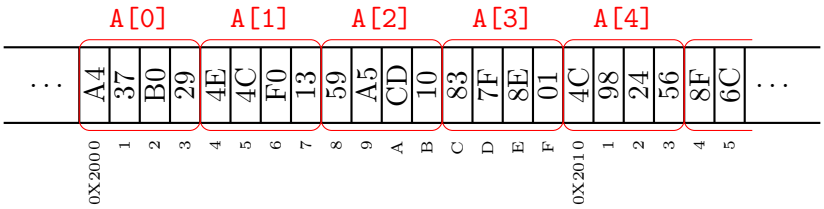
Pour une toute petite fenêtre, l'ensemble de travail ne comportera que quelques instructions et quelques octets de données. Plus la fenêtre sera grande, plus la taille de l'ensemble de travail sera grande, jusqu'à tendre vers la taille totale requise pour le déroulement du programme complet.
Une fenêtre plus grande permet d'avoir un plus gros ensemble de travail, mais nécessite un plus gros cache pour pouvoir exécuter le code plus rapidement. Il n'y a pas de formule magique donnant la taille idéale de la fenêtre, pas plus que la taille idéale du cache, et les questions économiques doivent être prises en compte.

E. Localisation des données

Le principe de localisation des données dit que **des données proches de la donnée traitée sont susceptibles d'être utilisée dans un avenir proche**.
Considérons la boucle suivante :

```
for (i=0, i<N, i++) {
   ... A[i] ...
}
```

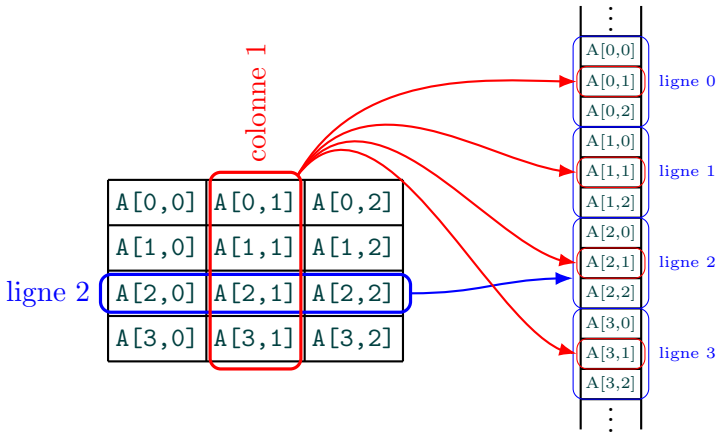
Ce morceau de code accède à tous les éléments du tableau `A`, dans l'ordre. Supposons celui-ci stocké dans la mémoire centrale de la manière suivante [4] :



Cet algorithme a un **pas de** 1, au sens où il accède aux éléments du tableau l'un après l'autre. Si l'on stocke dans le cache des blocs d'octets consécutifs, on peut s'attendre à ce que l'élément `A[2]` soit déjà présent dans le cache au moment où on veut y accéder, étant donné qu'il est proche en mémoire de l'élément `A[1]` auquel on vient d'accéder. Il y a donc de fortes chances que le taux de "cache hit" soit élevé.
A contrario, le morceau de code suivant :

```
for (i=0, i<N, i+=4) {
   ... A[i] ...
}
```

a un pas de 4, et accède donc à des éléments du tableau situés relativement loin les uns des autres. Il y a donc plus de chance que les données ne soient pas présentes dans le cache au moment où on en a besoin.
Le cas des tableaux à deux dimensions est plus fréquent, et plus parlant :



4. On suppose ici que chaque élément du tableau `A` est par exemple un flottant en simple précision, occupant donc 4 octet.

Les compilateurs rangent usuellement les tableaux ligne par ligne. Donc les éléments d'une même ligne sont dans à des positions consécutives de la mémoire, alors que des éléments d'une même colonne sont séparés par un nombre d'octets égal à la place occupée par une ligne complète.
Les deux codes suivants vont avoir des comportements très différents :

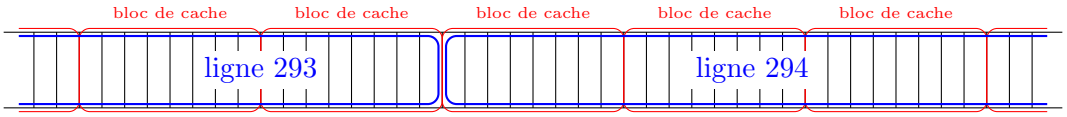
```
for (i=0, i<4, i++) {
   for (j=0, j<3, j++) {
      ... A[i,j] = ...
   }
}
```
pas de 1

```
for (j=0, j<3, j++) {
   for (i=0, i<4, i++) {
      ... A[i,j] = ...
   }
}
```
pas de 4

Celui de gauche a un pas de 1, son ensemble de travail traite des données consécutives, qui ont donc de grandes chances d'être présentes dans le cache au moment de leur accès.
Par contre, celui de droite a un pas de 4, et accède donc a des données non consécutives. On peut imaginer qu'un plus grand tableau provoquerait régulièrement des "cache miss", et des évictions en cascade.

F. Calcul du taux de cache hit, temps moyen d'accès à la mémoire

Regardons ce qui se passe dans le cas d'un gros tableau. Prenons l'exemple d'un cache de 64 octets, soit de 8 flottants en double précision.



ON voit sur le schéma ci-dessus la fin de la ligne 293 de notre tableau, et le début de la ligne 294. On suppose les blocs de cache bien alignés, mais cela n'a pas beaucoup d'importance pour notre démonstration.
Dans le cas d'un algorithme traitant le tableau ligne par ligne, donc avec un **pas de** 1, à chaque lecture d'un nouvel flottant, on place cet élément et les 7 suivants dans le cache. On a donc 1 "cache miss" pour 7 "cache hits". Ainsi :

- le taux de cache hits est $\frac{7}{8} \approx 87\%$,
- le taux de cache miss est $\frac{1}{8} \approx 12\%$.

Si l'on reprend les valeurs de références mentionnées plus haut, et qu'on considère qu'un cache hit renvoie la donnée en 4 cycles d'horloges, et qu'un cache miss charge le cache et renvoie la donnée en 100 cycles d'horloge, le nombre moyen de cycles d'horloge pour accéder à une donnée est

$$\frac{7}{8} \times 4 + \frac{1}{8} \times 100 = 16$$

soit 16 cycles d'horloge en moyenne pour chaque lecture d'un flottant [5]. On a donc multiplié l'efficacité de la mémoire vive par un peu plus de 6.
Dans le cas d'un algorithme traitant le tableau colonne par colonne, on peut supposer, si les lignes sont suffisamment longues, que chaque lecture d'un flottant conduira à un cache miss. Dans ce cas, on aura 100% de cache miss, et donc un temps moyen d'accès à la mémoire de 100 cycles d'horloge.
On voit ici tout l'intérêt de cette étude : une bonne compréhension des mécanismes internes au microprocesseur permet d'améliorer l'implémentation pratique des algorithmes de manière significative.

III. Différents types de caches

Les caches se distinguent par les stratégies qu'ils adoptent, dans l'organisation du stockage des données et dans les transferts avec la mémoire centrale.
Commençons par étudier les stratégies de réécriture dans le cache.

5. Le calcul est un peu plus complexe que cela, mais cela donne une bonne idée des ordres de grandeur.

A. Stratégies d'écriture dans le cache

Lorsque le microprocesseur doit écrire une donnée dans la mémoire, il va en fait l'écrire sur son bus de données, et écrire l'adresse à laquelle il souhaite la stocker sur son bus d'adresses. Ces informations sont interceptées par le cache. Deux choses peuvent se produire :

1. la donnée est présente dans un bloc du cache ; on parle alors de "**write hit**" ;
2. la donnée n'est pas présente dans un bloc du cache ; on parle alors de "**write miss**".

Les caches se distinguent par leurs réactions à ces différents cas.

- en cas de "write hit", il peut
  - écrire directement la donnée dans la mémoire vive : on parle de cache "**write-through**",
  - attendre que le bloc soit évincé du cache pour actualiser la donnée dans la mémoire principale : on parle de cache "**write-back**" ;
- en cas de "write miss", il peut
  - lire le bloc contenant la donnée depuis la mémoire centrale, et actualiser la donnée dans le cache : on parle de cache "**write-allocate**",
  - ou bien transmettre l'écriture directement à la mémoire centrale, sans charger le bloc correspondant : on parle de cache "**write-no-allocate**".

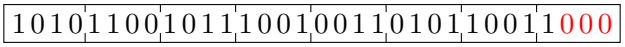
On peut donc concevoir quatre type de cache différents en panachant une option de "write hit" et une de "write miss". En général, les caches "write-back" sont aussi "write-allocate", les caches "write-through" sont aussi "write-no-allocate" [6].
Voici un approche fréquemment rencontrée dans les CPU modernes.

- En cas de "write hit", adopter la stratégie "write-back" : actualiser la donnée dans le cache, et ne la recopier dans la mémoire principale que lorsque cela est absolument nécessaire. Lee cache stocke un bit d'information supplémentaire, nommé "**dirty bit**", indiquant si le bloc a été modifié depuis son chargement (auquel cas il faut le réécrire en mémoire) ou pas (auquel cas on peut se passer de l'opération de réécriture).
- En cas de "write miss", adopter la stratégie "write-allocate" : charger le bloc depuis la mémoire principale, écrire dans le cache la donnée modifiée, et positionner le "dirty bit" à 1. La mémoire principale n'est pas immédiatement mise à jour, elle le sera au moment de l'éviction du bloc.

Ces stratégies sont cohérentes avec la notion de localisation spatiale. Elle assure une efficacité maximale, mais engendre des incohérences temporaires de la mémoire principale qui devront être corrigées à un moment ou à un autre.
L'autre approche, à savoir "write through/write-no-allocate", semble moins efficace, mais elle est parfois nécessaire, par exemple lorsque plusieurs processeurs partagent une mémoire centrale commune : il ne faudrait pas qu'un processeur travaille avec une valeur obsolète simplement parce que celle-ci n'a pas été mise à jour par un autre processeur...

B. Implémentation d'un bloc dans le cache

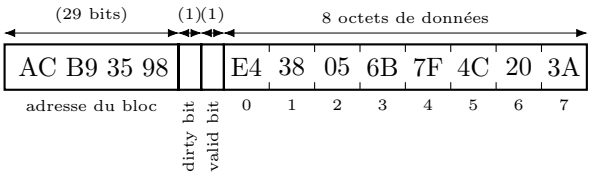
On suppose dans cette section que les blocs sont constitués de 8 octets (et non 64), et que les adresses sont sur 32 bits.
On s'arrange pour que les adresses des départs de blocs correspondent à des adresses mémoires multiples de 8. Ainsi, l'adresse d'un bloc se termine par trois [7] "0" :



La ligne de cache est représentée ci-dessous :

6. Il semblerait (à vérifier) que les CPU Intel utilisent une stratégie "write-back", alors que les CPU AMD utilisent plutôt une stratégie "write-through". Mais bien entendu, avec les technologies modernes, les choses sont loin d'être aussi simples que cela.

7. Un bloc de 64 octets se terminerait par 6 zéros.

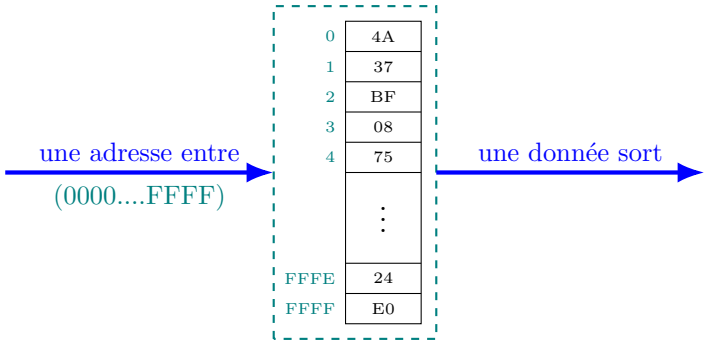


Après les 29 bits significatifs de l'adresse du bloc (les trois bits nuls en fin d'adresse ne sont pas implémentés), on trouve le "**dirty bit**" indiquant si le bloc a été modifié depuis son insertion dans le cache, puis le "**valid bit**" indiquant si le bloc est complet ou si les données n'ont pas de signification (par exemple à la mise sous tension de l'ordinateur), puis les 8 octets de données en cache.
On dit que cette ligne a une taille de 8 octets, même si elle contient des informations supplémentaires.

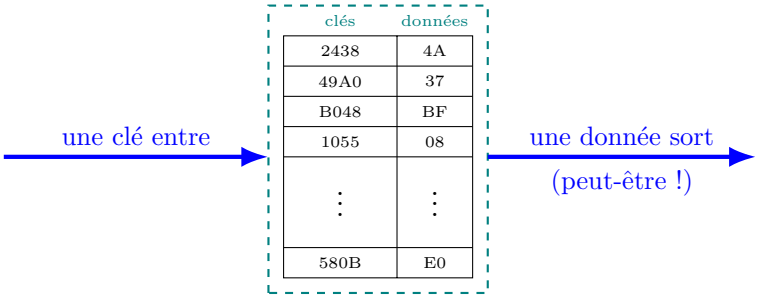
C. Cache associatif

1. Notion de mémoire associative

Traditionnellement, l'adressage mémoire est direct : on rentre une adresse, et la mémoire renvoie la donnée stockée à cette adresse (si elle existe matériellement) :



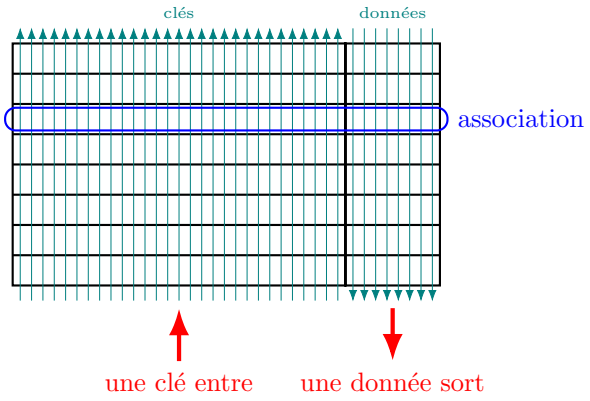
Lorsqu'on rentre une adresse sur le bus, une valeur sort de la mémoire (ici, l'une des 65536 valeurs stockées).
Une **mémoire associative** est implémentée de manière différente : on stocke en mémoire un petit nombre de paires composées d'une valeur et d'une **clé**.



Lorsqu'on fournit une clé sur le bus, une valeur sort de la mémoire **si** la clé est valide. L'espace des clés est en général beaucoup plus gros que la taille de la mémoire (les emplacements associés à chaque clé sont calculés à l'aide d'une *fonction de hachage*), donc il se peut que la clé fournie n'existe pas dans l'espace mémoire.
Ainsi, dans une mémoire traditionnelle, l'adresse a une taille de N bits, on peut stocker 2^N données.
Dans une mémoire associative, les clés peuvent avoir n'importe quelle taille (par exemple 29 bits), et la mémoire comporte un petit nombre de lignes de paires clé/valeur.

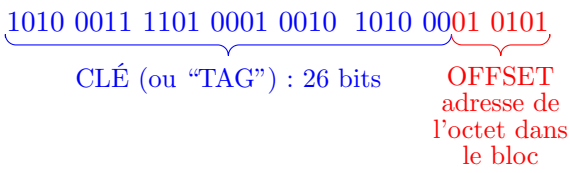
2. Fonctionnement d'un cache complètement associatif

Pour un **cache associatif**, les clés correspondent aux adresses des blocs, les valeurs sont les données stockées.



La clé est transmise en parallèle à toutes les cellules de la mémoire cache. Chaque cellule compare sa clé à celle présente sur l'entrée. Si une cellule a une clé identique, elle transmet son bloc de donnée sur la sortie. Pour fixer les données, voici les caractéristique d'un cache complètement associatif :

- La taille des blocs (B) est de 64 octets,
- Le cache contient 512 lignes (L),
- La taille du cache est donc [8] $S = B \times L = 32\,\text{Ko}$,
- il y a un seul ensemble de lignes (voir plus loin),
- les adresses occupent 32 bits, chaque adresse est donc décomposée de la manière suivante :



Voici alors le fonctionnement détaillé du cache (tout est codé *matériellement*, donc s'exécute très rapidement, en quelques cycles d'horloge) :

- en entrée, on trouve une adresse sur 32 bits ;
- l'adresse est découpée en un tag de 26 bits et un offset de 6 bits,
- le tag est propagé sur l'entrée de la mémoire associative ;
  - si il y a correspondance du tag avec l'une des clés stockées, le bloc de 64 octets correspondant est transmis par la mémoire associative en sortie, et on utilise l'offset pour extraire l'octet demandé ; celui-ci est alors transmis au CPU ;
  - si aucune ligne ne répond au tag, on sélectionne un bloc à évincer du cache [9], le tag est envoyé à la mémoire centrale, qui renvoie les 64 octets correspondants ; la ligne de cache est mise à jour (clé, données, dirty et valid bits...), et ce bloc est transmis sur la sortie de la mémoire associative, pour extraction de l'octet demandé à l'aide de l'offset qui est envoyé au CPU.

D. Cache à adressage directe

De par son fonctionnement, une mémoire associative est soit petite, soit lente (il faut transmettre beaucoup de données en parallèle à toutes les lignes du cache ; plus il y a de ligne, plus cela prend de temps et d'énergie). Une autre approche est utilisée : la notion de **mémoire à adressage directe**.

8. on compte uniquement la taille de l'espace occupé par les données, mais il ne faut pas oublier l'espace supplémentaire occupé par les clés, et les bits d'états.

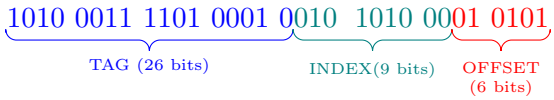
9. des bits supplémentaires sont nécessaire pour implémenter la stratégie de détermination du bloc non accédé depuis le plus longtemps.

1. Mémoire à adressage directe

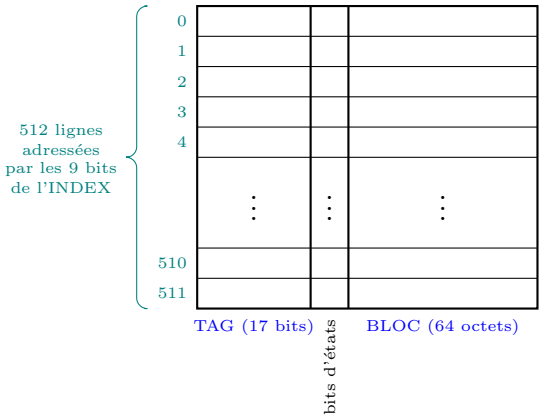
Contrairement à une mémoire associative dans laquelle n'importe quel bloc peut être stocké dans n'importe quelle ligne, dans une mémoire à adressage directe, chaque bloc ne peut être stocké que dans une ligne du cache.
Par exemple, le bloc de l'adresse 514 ne peut aller qu'à la ligne 2. Cette ligne ne peut stocker que le contenu des blocs aux adresses 2, 514, 1026...
Chaque ligne doit toujours contenir des informations concernant le bloc qu'elle stocke. Par contre, lorsqu'on recherche un bloc particulier, **on sait déjà où il peut se trouver**, il suffit donc d'interroger une ligne pour savoir si elle contient la donnée cherchée.

2. Fonctionnement d'un cache à adressage direct

Reprenons notre exemple d'une mémoire de 34 Ko, constituée de 512 lignes de blocs de 64 octets.
Comme le cache contient $512 = 2^9$ lignes, une ligne particulière est repérée par son adresse sur 9 bits. Une adresse mémoire est maintenant découpée de la manière suivante :



La mémoire est organisée de la manière suivante :



Pour lire une donnée dans ce cache :

- on découpe l'adresse en trois morceaux, le tag de 17 bits, l'index de 9 bits et l'offset de 6 bits,
- on utilise l'index pour sélectionner la bonne ligne du cache,
- on compare le tag lu sur cette ligne au tag calculé avec l'adresse, si ils concordent le bloc correspondant est transmis,
- on sélectionne le bon octet dans le bloc à l'aide de l'offset.

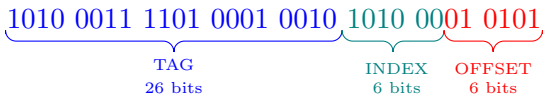
Il y a un problème potentiel associé à cette configuration : comme la ligne 2 peut contenir un seul des blocs 2, 514, 1026..., si l'ensemble de travail inclut *à la fois* ces deux blocs, à chaque fois qu'on voudra accéder au bloc qui ne sera pas présentement dans le cache, il faudra l'évincer et le remplacer par le bon bloc (les anglais parlent de "cache trashing").
Cela est rare, mais si cela se produit, cela peut affecter les performances du cache et en limiter l'intérêt.

IV. Le meilleur des deux mondes

Pour palier les inconvénients des deux types de cache (cache associatif limité en taille, cache à adressage direct augmentant les possibilités de "trashing"), on combine les deux approches de la manière suivante : on combine un grand nombre de petites mémoires associatives, chacune étant indexée par un index calculé à partir de l'adresse mémoire.
Considérons l'exemple d'une mémoire cache de 32 Ko, "8 way set-associative" :

- chaque mémoire associative ne contient que 8 lignes (L),
- chaque ligne contient 64 octets (B),
- on combine 64 telles mémoires (S).

On retrouve la taille de la mémoire en faisant le produit $S \times L \times B$.
On découpe comme dans l'exemple vu pour les mémoires à adressage direct l'adresse en trois parties :



Lorsque le CPU veut accéder (en lecture ou en écriture) à une donnée en mémoire, le cache commence par découper l'adresse en un tag, un index et un offset. Il sélectionne la mémoire associative correspondant à l'index, recherche le tag dans cette mémoire. Si il est présent, le bloc de données correspondant est renvoyé par la mémoire, et l'octet demandé est sélectionné par son offset dans le bloc.
Tout se passe comme si on avait une mémoire à adressage direct, dans laquelle les conflits d'occupation étaient possibles [10], dans une certaine mesure.

V. Résumé

Ci-dessous sont résumés les principes généraux d'organisation des caches.

- La mémoire cache est en général de l'ordre de 10 fois **plus rapide** que la mémoire centrale. Elle est constituée de mémoire **statique**, contrairement à la mémoire centrale qui est constituée de mémoire **dynamique**. Elle est **plus chère** à produire, et consomme **plus d'énergie**, on ne peut donc la substituer à la RAM classique.
- Si la donnée demandée par le processeur est dans le cache, on parle de **cache hit**. Sinon, on parle de **cache miss**. Le ratio du nombre de cache hits sur le nombre total d'accès est appelé le **ratio de hit**. On définit de même le **ratio de miss**.
  L'objectif est bien entendu d'avoir un ratio de hit le plus grand possible.
- Les cache miss proviennent de trois causes principales :
  — les manquements **obligatoires** se produisent lorsque le cache n'a pas encore été chargé, par exemple au lancement du programme,
  — les manquements par **conflit** se produisent lorsque deux blocs occupent la même place dans la mémoire cache,
  — enfin les manquements de **capacité** se produisent en raison de la taille nécessairement limitée de la mémoire cache.
- Le cache stocke des **lignes** de **blocs** d'octets consécutifs, ainsi que des informations sur la localisation en mémoire des blocs stockés, et des informations sur la cohérence des données stockées (**dirty bit**, **valid bit**...).
- Le fait de stocker non seulement les données utilisées dans le cache, mais aussi les données autour, profite du **principe de localisation** :
  — si une donnée a été utilisée récemment, elle a de grande chance d'être utilisée dans un temps proche (**localisation temporelle**,
  — si une donnée a été utilisée récemment, les données proches risquent d'être utilisée prochainement (**localisation spatiale**).

10. On peut comparer cette situation à celle d'un hachage de données dans lequel les conflits sont réglés en enfilant les données correspondant au même hachage dans une liste dont on peut espérer qu'elle sera de taille bien inférieure à la taille totale de l'ensemble de données.

- Il y a deux méthodes pour associer un bloc à une position dans le cache. Dans les **mémoires associatives**, les blocs peuvent être stockés n'importe où. Dans les **mémoires à adressage direct**, chaque adresse mémoire ne peut aller que dans un emplacement du cache.

  En pratique, on combine ces deux techniques : l'adressage direct permet de stocker de nombreux blocs, l'associativité compense les conflits.
- Les stratégies d'écriture en mémoire se distinguent en quatre catégories.
  - Lorsque la donnée à écrire se trouve dans le cache (**write hit**), on peut transférer directement cette donnée dans la mémoire centrale (**write through**), ou ne pas actualiser la mémoire jusqu'à ce que le bloc doivent être évincé.
  - Lorsque la donnée à écrire ne se trouve pas dans le cache (**write miss**), on peut charger le bloc en cache et mettre à jour la donnée modifiée (**write-allocate**), soit mettre directement à jour la mémoire principale sans actualiser le cache (**write-no-allocate**).

Les avantages principaux des caches sont résumés ci-dessous.

- La mémoire cache fonctionne au moins 10 fois plus vite que la mémoire centrale. Elle augmente donc drastiquement la rapidité des programmes, si elle est utilisée de façon efficace.
- Tous les programmes bénéficient de la présence d'un cache, indépendamment de la façon dont le programmeur travaille.
- Par contre, une bonne connaissance des principes exposés permet au programmeur de construire des programmes mieux adaptés, et profitant mieux de la présence du cache, de manière à maximiser le hit ratio.

VI. Bibliographie

Ces notes de cours sont très fortement inspirées des vidéos de Harry H. Porter. Ces vidéos sont visibles sur Youtube, et ce papier reprend plus ou moins fidèlement son plan et ses schémas.
J'ai aussi utilisé un cours de Quentin L. Meunier pour approfondir les connaissances sur le sujet.