# Architecture des ordinateurs

Cours (12x2h00) :

- Frédéric Mallet - fmallet@unice.fr

TP (12x2h00 - 2 groupes) :

- Jean-Pierre Lips - Jean-Pierre.LIPS@unice.fr
- Christophe Delage – Christophe.Delage@sophia.inria.fr

*http://deptinfo.unice.fr/~fmallet/archi*



# Structure du cours

- Performances : évolution et comparaison
- Codage de l 'information
- Fonctions logiques et éléments mémoires
- Systèmes à microprocesseurs
- La famille Intel - 80x86
- Conception d'architectures numériques - VHDL
- Éléments avancés (parallélisme, mémoire, ...)
  - **Hiérarchie mémoire**



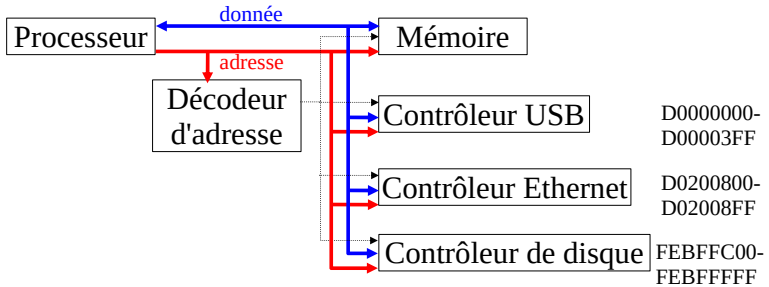
# Mémoire et performance

- **Latence (temps d'accès) – en cycles**
  - le temps nécessaire à une opération de lecture/écriture i.e., le temps qui sépare l'instant auquel l'opération est demandée (exemple : l'adresse est présentée à la mémoire pour une opération de lecture) de l'instant auquel l'opération est terminée (l'information est disponible)
- **Débit – en Moctets/s**
  - la quantité d'informations lues/écrites par unité de temps
- **Bande passante – en Moctets/s**
  - La quantité de données lues/écrites par unité de temps

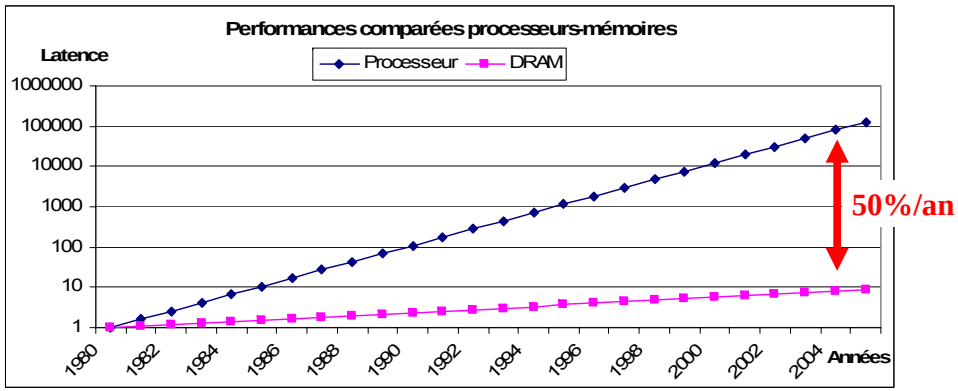


# Mémoire et entrées/sorties

- Une mémoire est un composant auquel on donne une adresse pour obtenir une donnée (**lecture**) ou sur lequel on écrit une donnée à une adresse (**écriture**)
- On peut utiliser un mécanisme similaire pour manipuler des entrées/sorties (périphériques, ...)
  - Il suffit que l'espace d'adressage soit disjoint



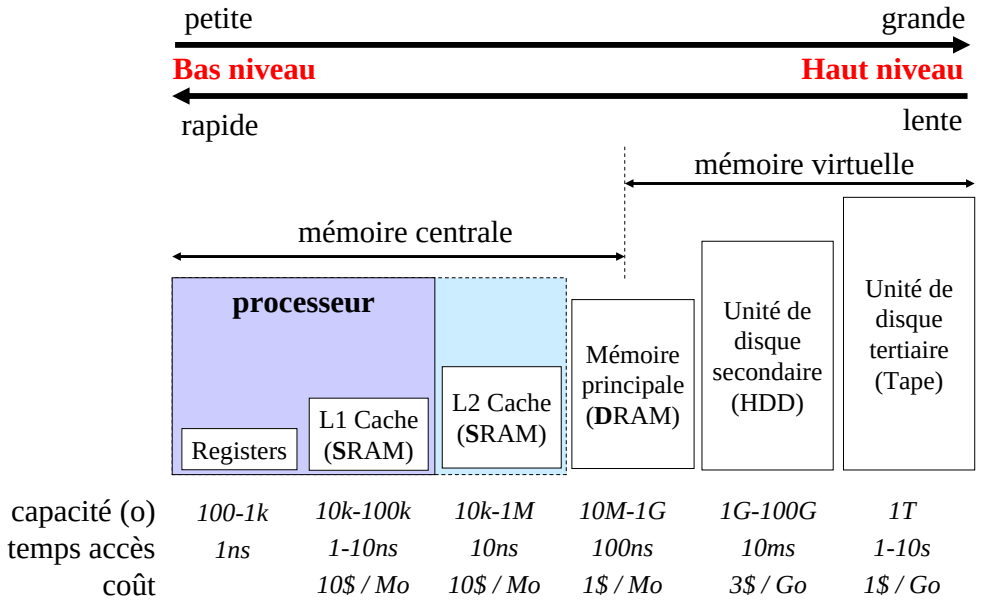
# Importance de la mémoire




- Les performances au cours des années
  - Processeur : 60% / an = x2 / 1.5 ans (Loi de Moore)
  - Mémoire : 9% / an = x2 / 10 ans
  - **Écart : 50% / an**



# Hiérarchie mémoire




| | | | | | | |
|---|---|---|---|---|---|---|
| capacité (o) | *100-1k* | *10k-100k* | *10k-1M* | *10M-1G* | *1G-100G* | *1T* |
| temps accès | *1ns* | *1-10ns* | *10ns* | *100ns* | *10ms* | *1-10s* |
| coût | | *10$ / Mo* | *10$ / Mo* | *1$ / Mo* | *3$ / Go* | *1$ / Go* |



# Mot mémoire, ligne et bloc

- Un **mot** est la plus petite unité manipulée sur le bus de données
  - La taille du mot est déterminée par la largeur du bus
  - e.g. Les mémoires 16 bits ont des mots de 16 bits
- Une **ligne** (ou **bloc**) est la plus petite unité lue ou écrite dans une mémoire cache (**page** pour les mémoires virtuelles)
  - Une ligne peut contenir 1 ou plusieurs mots
  - Dans une mémoire cache, tous les mots d'une même ligne sont liés
- **Capacité** d'un cache(en bits) = taille des mots x nombre de mots
  - On ne compte pas toutes les données supplémentaires qui font l'association entre les lignes du cache et les adresses mémoires
  - Il arrive qu'on exprime la taille d'un cache en nombre de mots



# Principe de localité

- Un programme accède à une zone relativement petite de la mémoire à un instant donné
- **Localité temporelle**
  - Si un mot mémoire est accédé, il y a de grandes chances qu'il soit à nouveau accédé peu après
  - Pourquoi ? boucle, réutilisation, …
- **Localité spatiale**
  - Si un mot mémoire est accédé, les adresses contiguës seront probablement accédées peu après
  - Pourquoi ? Programmes séquentiels, tableaux, …

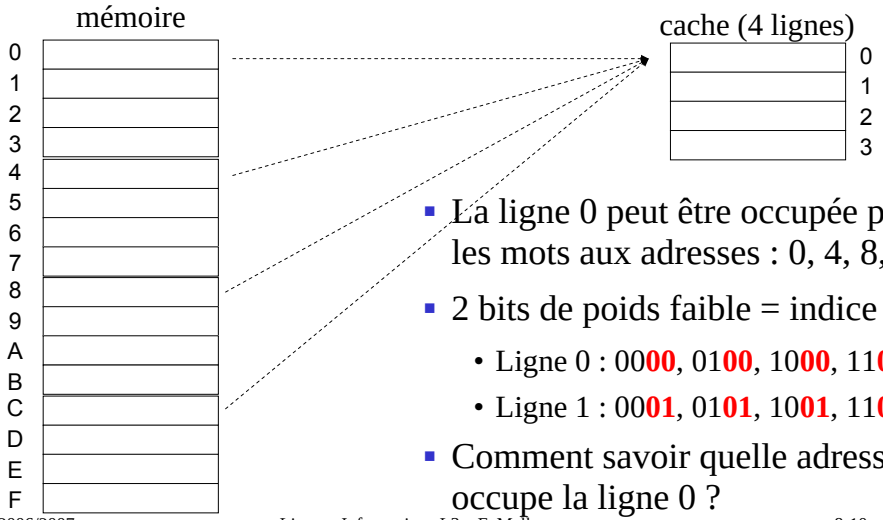
# Mémoire cache (idée générale)

- Il s'agit d'interposer une mémoire plus petite, donc **plus rapide (faible latence)** entre le processeur et la mémoire : mémoire cache
- Le cache étant plus petit, il ne peut pas contenir toute la mémoire :
  - Il faut **associer** plusieurs adresses mémoire à la même **ligne du cache**




# Cache à **accès direct** (*direct-mapped*)

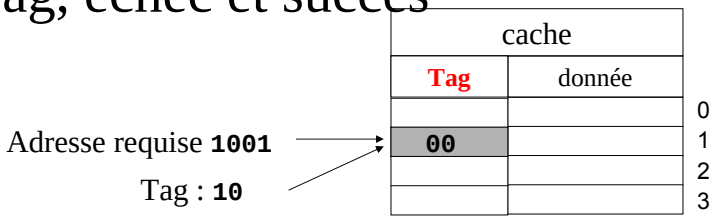
- **Chaque adresse** est associée à **1 ligne** du cache
  - Plusieurs adresses sont associées à la même ligne !




- La ligne 0 peut être occupée par les mots aux adresses : 0, 4, 8, C
- 2 bits de poids faible = indice
  - Ligne 0 : 00**00**, 01**00**, 10**00**, 11**00**
  - Ligne 1 : 00**01**, 01**01**, 10**01**, 11**01**
- Comment savoir quelle adresse occupe la ligne 0 ?



# Tag, échec et succès



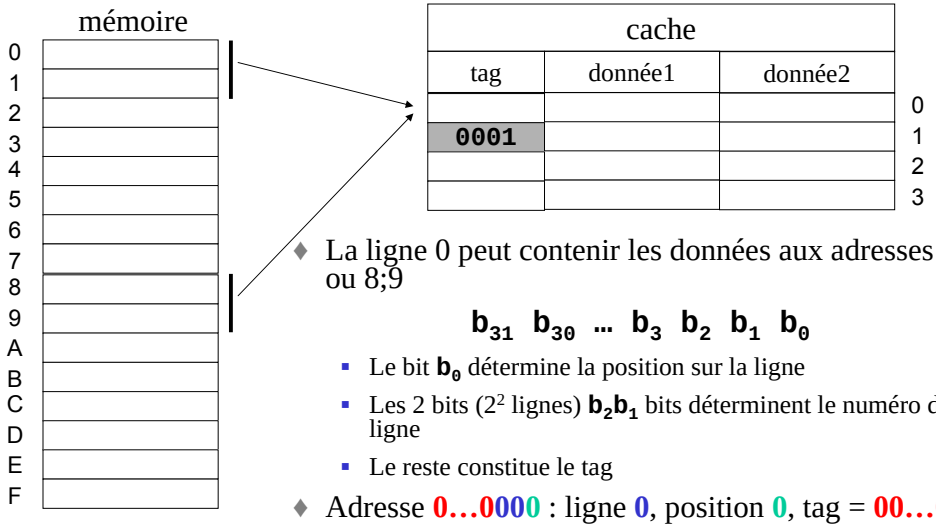

- On veut réduire le **tag** au maximum
  - On enlève les 2 derniers bits de l'adresse (il y a $4=2^2$ lignes)
- On compare l'adresse requise avec le tag
  - Si correspondance : **succès** (*Hit*)
    - On obtient alors la donnée rapidement (latence du cache)
  - Sinon : **échec** / **défaut** (*Miss*)
    - On accède à la mémoire principale (plus lente)



# Exemple – cache 64Ko = $2^{16}$ octets

- Mot mémoire de 16 bits = $2^1$ octets
- Espace d'adressage de 4Go = adresse de 32 bits
- Un cache de capacité 64Ko contient $2^{15}$ lignes
  - $2^{16-1} = 2^{15}$ lignes = $2^{15}$ mots
- Le tag est alors de 32-15 = **17** bits
- Chaque ligne du cache occupe 16+**17** = 33 bits
- Ce cache de capacité 64ko occupe en mémoire $33*2^{15=10+2+3}$ bits, c'est-à-dire 132Ko
- **La pénalité due au tag est énorme !**

# 2 mots par lignes ! Accès direct




- La ligne 0 peut contenir les données aux adresses 0;1 ou 8;9

$$b_{31}\ b_{30}\ \ldots\ b_3\ b_2\ b_1\ b_0$$

  - Le bit $b_0$ détermine la position sur la ligne
  - Les 2 bits ($2^2$ lignes) $b_2b_1$ bits déterminent le numéro de ligne
  - Le reste constitue le tag
- Adresse **0…0000** : ligne **0**, position **0**, tag = **00…0**
- Adresse **0…1001** : ligne **0**, position **1**, tag = **00…1**



# E.g. – cache 64Ko, 2 mots par ligne

- Mot mémoire de **16** bits = $2^1$ octets
- Espace d'adressage de 4Go = adresse de **32** bits
- Chaque ligne contient 2 mots = $2^2$ octets
- Un cache de capacité 64Ko=$2^{16}$octets contient $2^{14}$ lignes
  - $2^{16-1} = 2^{15}$ mots et $2^{16-2} = 2^{14}$ lignes
- Le tag est alors de 32-15 = **17** bits
- Chaque ligne du cache occupe **16**+**16**+**17** = 49 bits
- Ce cache de capacité 64ko occupe en mémoire 49*$2^{14}$ bits, soit 49*$2^{11}$ octets, c'est-à-dire 98Ko
- **La pénalité due au tag est réduite !**



# Accès direct – Taille effective

- De façon générale
  - Pour un espace d'adressage de $2^A$ octets
  - Pour des mots de donnée de $2^D$ octets
  - Avec un cache de $2^L$ lignes et $2^M$ mots par ligne
- La capacité du cache est : $2^{L+M}$ mots = $2^{L+M+D+3}$ bits
- Le tag occupe : A – M = T bits
- Chaque ligne contient T + $2^{D+M+3}$ bits
- Le cache occupe un espace mémoire de
  - $2^L$ * (T + $2^{D+M+3}$) bits



# Taux d'échec et taux de succès

- **Succès (*Hit*) :** donnée présente dans le cache
  - **Taux de succès (*Hit_rate*) :** fraction des accès mémoires qui ont provoqués un succès (doit être grand)
  - **Temps pour succès (*Hit_time*)** : temps total pour obtenir une donnée lors d'un succès = accès à la SRAM + temps pour déterminer si il y a succès
- **Échec/défaut (*Miss*) :** donnée absente
  - **Taux d'échecs (*Miss_rate*) :** `miss_rate=1-hit_rate`
  - **Défaut de cache (*Miss_penalty*)** : temps pour remplacer le mot dans le cache + temps accès le mot dans le niveau inférieur + temps de transfert vers le niveau courant
- **Temps accès moyen** :
  - `Hit_time + Miss_rate * Miss_penalty (ns ou cycles)`

# Les échecs en lecture

- Lorsqu'une lecture du cache provoque un échec
  - La donnée est lue dans la mémoire principale
  - Puis copiée (**chargée**) dans le cache
  - Lorsque la ligne du cache est déjà occupée
    - Il y un **conflit** de lecture
    - La nouvelle donnée remplace l'ancienne (**victime**) et le tag est mis à jours
  - Temps de traitement = temps d'accès à la mémoire principale plus le temps de transfert pour chargée la ligne (**proportionnel à la taille de ligne!**)
    - Si la ligne contient plusieurs mots, il faut lire tous les mots de la ligne !



# Les succès en écriture

- **Écriture simultanée (*Write through*)**
  - La donnée est écrite à la fois dans le cache ET dans la mémoire principale : cohérence maximale
  - Cela peut coûter cher si la donnée est écriture plusieurs fois à la suite (dans une boucle) sans être lue
- **Ré-écriture** (*Write back*)
  - La donnée est écrite seulement dans le cache
  - Un **bit spécial** (*dirty bit*) est positionné pour indiquer que la ligne du cache ne correspond pas à la mémoire
  - Lorsque la ligne devient victime, si le bit spécial est positionné, la mémoire principale est mise à jour
    - Peut entraîner des problèmes de cohérence dans les systèmes distribués.



# Les échecs en écriture

- **Écriture allouée (*Write allocate*)**
  - Une ligne est allouée dans le cache, la donnée est écrite à la fois dans le cache ET dans la mémoire principale
- **Écriture non allouée (*Write no allocate*)**
  - La donnée est seulement écrite dans la mémoire principale
  - Le principe de localité est moins vrai sur les écritures
- Toutes les combinaisons de stratégies sont possibles
  - *Write Through allocate* ou *no allocate*
  - *Write Back allocate* ou *no allocate*



# Associativité

- L'**associativité** est le nombre de lignes du cache associées à une adresse
  - L'associativité est la même pour toutes les adresses
  - L'ensemble des lignes associées à une adresse s'appelle un **ensemble** (***set***)
- Cache à correspondance (accès) **directe** (***Direct-mapped***)
  - Associativité = 1, la correspondance est rapide
- Cache à correspondance **complètement associative** (*Fully-associative*)
  - Associativité = nombre de lignes du cache
- Cache à correspondance associative par ensemble (*Set-associative*)

# *Fully-Associative*

- Une adresse est associée à n'importe quelle ligne du cache
  - Chaque ligne peut contenir toutes les adresses
  - La première ligne libre est utilisée
  - Le tag est nécessairement l'adresse complète (divisée par le nombre de mots par ligne)
- Lors d'un accès mémoire (lecture/écriture)
  - Il faut parcourir TOUT le cache pour trouver une correspondance
  - Permet d'utiliser tout le cache, mais cela augment le temps de correspondance



# *Fully-Associative*

| Ligne du cache | Adresse | Donnée |
|---|---|---|

- La ligne du cache contient
  - La donnée
  - L'adresse complète à laquelle cette donnée se trouve dans la mémoire
- Ce type de cache est très coûteux
  - e.g. 32 bits d'adresse, 16 bits de données
  - Chaque ligne pèse 48 bits
  - Il faut 192Ko de mémoire pour un cache de taille 64Ko
- Exercice : Quelle est l'associativité d'un tel cache ?



# Cache à correspondance directe

- **Associativité = 1**
  - Chaque adresse est associée à UNE seule ligne du cache
- Pour établir la correspondance, il suffit de regarder à la ligne associée
  - Bits de poids faible de l'adresse = numéro de ligne
- Mais si on utilise fréquemment des adresses associées à la même ligne, on perd l'intérêt d'utiliser un cache
  - **`READ 0`**
  - **`READ 4`**
  - **`READ 0`**
  - **`READ 4`**



# Associatif par ensemble

| Ligne du cache | Tag | Donnée |
|---|---|---|

- Chaque adresse est associée à 1 ensemble '*set*' (plusieurs lignes) mais pas à toutes
  - Les bits de poids faible servent à identifier le numéro de l'ensemble
  - Les bits de poids fort de l'adresse constituent le *tag*
  - C'est une solution intermédiaire qui ne coûte pas autant que le *Fully-Associative* mais a beaucoup de ses avantages.

| Adresse mémoire | Tag | Numéro de *set* dans le cache |
|---|---|---|

# Associatif par ensemble

- Exemple
  - Bus d'adresses = 32 bits  Un mot = 16 bits
  - Un cache set-associative de 64Ko = $2^{16}$ octets = $2^{15}$ lignes
  - L'associativité est $2^{12}$ :
    - Chaque adresse peut être associée à $2^{12}$ lignes, les *set*s font $2^{12}$ lignes
    - Il y a $2^{15-12}=2^3=8$ *sets*, 3 bits pour coder le numéro du *set*
    - `adr%8` indique le *set* auquel est associée l'adresse `adr`
  - A l'intérieur du set, on peut utiliser n'importe quelle ligne !
  - Quelle est la taille du *tag* nécessaire ? 32-3 = 29 bits
  - Chaque ligne fait donc 29+16=45 bits
  - La mémoire nécessaire pour le cache est donc : 180Ko contre 132Ko pour le *direct-mapped* et 192Ko pour le *Fully-Associative*



# Généralisation

- Si il y a 1 ligne par *set* ?
  - Autant de *set* que de ligne
  - Associativité = 1, *direct-mapped*
- Si il y a 1 seul *set* ?
  - Le *set* a la taille du cache
  - Associativité = taille du cache, *fully-associative*
- Comment calculer la taille du tag ?
  - Taille du tag = taille adresse – $\log_2$(nombre de set)
  - Taille de la ligne = taille du mot + taille du tag
    - Si il y a un seul mot par ligne !



# Plusieurs mots par ligne – *Fully-associative*

- Si il y a plusieurs mots par ligne le coût lié au tag est amorti
  - Un mot (16 bits) par ligne, tag de 32 bits, les lignes pèsent 48 bits
    - Rapport 1/3 : cache de 64Ko → 192Ko de mémoire
  - Deux mots par ligne, tag de 32 bits, les lignes pèsent 64 bits
    - Rapport ½ : cache de 64Ko → 128Ko de mémoire
  - Huit mots par ligne, tag de 32 bits, les lignes pèsent 160 bits
    - Rapport 4/5 : cache de 64Ko → 80Ko de mémoire
- Mais attention, la ligne est la plus petite unité lue/écrite dans le cache
  - A chaque fois qu'on écrit une ligne dans le cache, il faut charger tous les mots de la ligne : avantages et inconvénients, voir TP



# Choix des victimes

- Lors d'un échec avec allocation
  - Pour un cache *direct-mapped*, 1 seule ligne par *set*
    - L'élection est facile
- Stratégie d'élection d'une victime
  - Aléatoire : une ligne est choisie au hasard dans le *set*
  - *FIFO (First In First Out)* : la première ligne allouée dans le *set* est choisie comme victime
  - LRU (Least Recently Used) : la ligne la moins récemment utilisée dans le set est choisie comme victime

# FIFO/LRU – format de la ligne

- Pour implémenter ces deux politiques (plus équitables), il faut conserver l'ordre d'entrée (resp. d'accès) des lignes du cache
- Il faut donc ajouter de l'information dans la ligne
  - L'information supplémentaire nécessaire est proportionnelle aux nombres de lignes dans le *set*.
    - Pour un cache *fully-associative* avec $2^{16}$ lignes, il faut 16 bits supplémentaires par ligne
  - Cette information supplémentaire est une extension du *tag*



# Bilan – Correspondance associative

- Comment savoir si une adresse a été chargée dans le cache ?
  - Il FAUT parcourir tout le cache : dans le pire des cas = elle n'y est pas
- Complexité pour élire une victime ?
  - Dépend de la politique d'élection : Random = O(1),
  - LRU/FIFO = O(n), il faut parcourir tout le cache pour trouver la victime !
- Le *Fully-Associative* est très coûteux en mémoire et a un temps d'accès élevé
  - Seulement réalisable pour les petits caches : niveau 1
  - Taux de succès (*Hit_rate*) = très élevé !



# Bilan – Correspondance directe

- Comment savoir si une adresse a été chargée dans le cache ?
  - Il SUFFIT de lire le tag de LA ligne associée : O(1)
- Complexité pour élire une victime ?
  - Pas le choix, une seule ligne possible : O(1)
- Ce type de cache est rapide et peu gourmand en mémoire
  - Taux de succès (Hit_rate) = très faible dans de nombreux cas !
  - Exemple : 8 lignes dans le cache, je lis en boucle plusieurs fois, les données aux adresses 0 et 8 ?



# Bilan – correspondance par ensemble

- Comment savoir si une adresse a été chargée dans le cache ?
  - Il FAUT parcourir tout le *set* : O(taille du *set*)
- Complexité pour élire une victime ?
  - Random : O(1)
  - FIFO/LRU : O(taille du set)
- *Set-Associative* est un très bon compris, mais il faut bien choisir tous les paramètres
  - Taille du *set*, nombre de mots par ligne, stratégie d'élection de victime, stratégie sur écriture, … : **Bancs de tests**