# Architectures des ordinateurs

# VHDL entity

Domaine concurent (architecture):

- declaration de siganl
- instanciation / connexion de sous composants
- affectation de signal
- process

Domaine séquentiel (process)

- déclaration de vaiable
- affectation de variable
- affectation de signal

# Evolutions des architectures de processeurs

Lien du cours : https://moodle-n7.inp-toulouse.fr/pluginfile.php/152004/mod_resource/content/2/Archi2SDN.pdf (https://moodle-n7.inp-toulouse.fr/pluginfile.php/152004/mod_resource/content/2/Archi2SDN.pdf)

- 1977: Apple 2 (gros succès)
- 1978: Intel x86 et microproc 8086
- 1981: IBM PC
- Classification des architectures :
  - SISD : Single Instruction Single Data
    - Un seul coeur de calcul
    - A tout instant, exécution d'une instruction sur un seul flot de données
    - Pas de parallélisme
  - SIMD : Single Instruction Multiple Data
    - Plusieurs coeurs de calcul
    - A tout instant, exécution d'une instruction sur plusieurs flots de données
    - Exemple d'architecture :
      - GPU
      - Processeurs vectoriels
      - Architectures systoliques
    - Exemple d'application :
      - Changement de la luminosité d'une image
      - Correction lexicale de texte
  - MISD : Multiple Instruction Single Data
    - Plusieurs coeurs de calcul
    - A tout instant, exécution d'instructions différentes sur un seul flot de données
    - Architecture rare
    - Peut servir pour de la tolérance aux pannes (systèmes critiques)
      - Comparaison des résultats
  - MIMD : Multiple Instruction Multiple Data
    - Plusieurs coeurs de calcul
    - A tout instant, ex´ecution d'instructions différentes sur des flots de données différents
    - Deux sous groupes :
      - SPMD : Single Program Multiple Data ( ⚠ exécutions indépendantes asynchrones != SIMD)
      - MPMD : différents programmes sur différentes unités
    - Un exemple "ancien" de machines : l'IPSC/2 d'Intel, fin des années 80
      - Machine à topologie hypercube
      - 64 processeurs

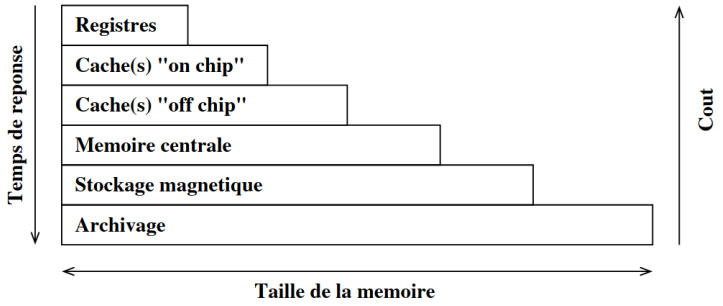
## Mémoire

Critères pour mesurer la rapidité d'une mémoire :

- *Temps de latence* : temps nécessaire pour effectuer une lecture ou une écriture
- *Débit mémoire* : quantité d'informations qui peuvent être lues ou écrites en une seconde dans la mémoire

L'accès la mémoire est *très lent* (typiquement 100~200 cycles).
Pour aller plus vite, on peut prendre des mémoires plus petites. Mais pour garder une grande capacité, on utilise donc une hiérarchie de mémoires.

| Type de mémoire | Temps de latence | Capacité |
|---|---|---|
| Registres | 1 opération UAL | $n \times 10^2$ bits |
| Caches | 10-20 opérations UAL | $n \times 10^4 - 10^6$ bits |
| Mémoires RAM | 100-200 opérations UAL | $n \times 10^9$ bits |
| Mémoires de masse | 1000 opérations UAL | $n \times 10^{11}$ bits |

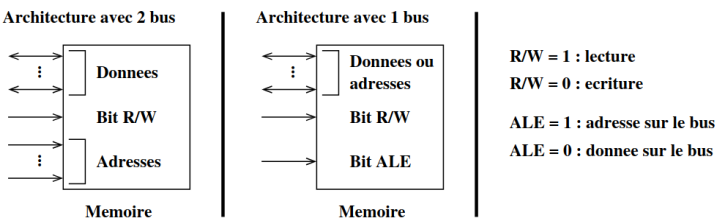



Deux types de mémoires :

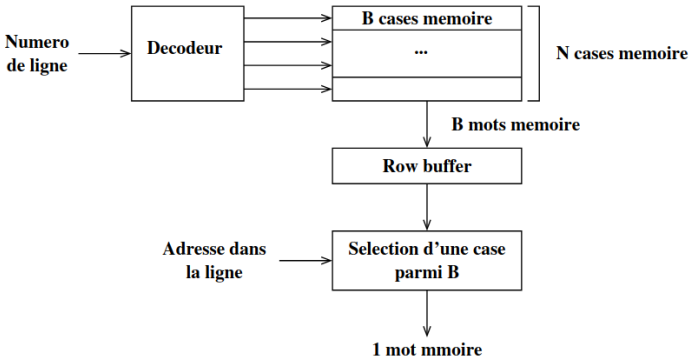
- ROM (Read Only Memory)
- RWM (Read Write Memory) : la RAM (Random Access Memory) par exemple

**Mémoire cache** : plusieurs données partagent le même case du cache => une information supplémentaire (le tag) pour identifier la données présente dans la case.
**Mémoires à accès séquentiel** : FIFO ou LIFO.

Connexion d'une mémoire adressable :



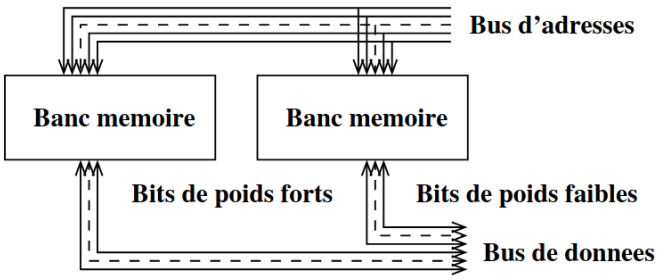

**Mémoire à row buffer (tampon de ligne)** : utile dans les caches où on lit les programmes linéairement. On sélectionne la ligne puis la colonne. On peut ainsi souvent garder la même ligne et juste changer la colonne.

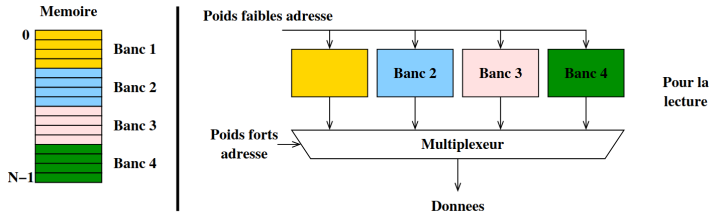


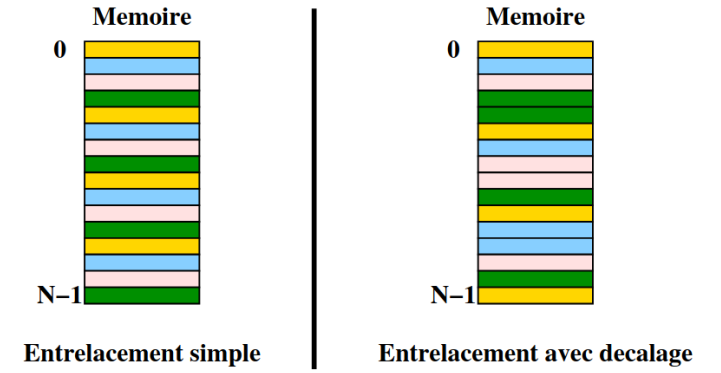

**Mémoires synchrones pipelinées** : accès en plusieurs étapes, opérations qui se chevauchent (ex: envoit l'@ de la prochaine lecture pendant la lecture des données de l'opération précedante)

**Bancs mémoire** : une mémoire est une ensemble de bancs mémorie. On a la possibilité de rafraîchir les bancs mémoires en parrallèle.
Plusieurs possibilités:

- Les mots mémoires sont découpés et répartie sur les bancs (*Arrangement horizontal*)
- Les adresses sont réparties sur plusieurs bancs (*Arrangement Vertical*)
- Entrelacé: le mélange des deux






## Cache

**Principe de localité (temporelle)** : les programmes accèdent souvents aux instructions et données récemment utilisées (fort heureusement !)

**Mémoire cache** : petite mémoire située sur la puce processeur (64 Ko par exemple)

On trouve dans le cache les données et instructions accédées récemment.
On a la possibilité d'avoir des caches séparés pour les instructions et les données afin de limiter les conflits d'accès.

**Problèmes à résoudre** :

- Comment structurer le cache ?
- Comment gérer les écritures ?
- Comment gérer les remplacements en cas de cache miss ?

**Ligne de cache** :

- Principe de localité spatiale

  => augmenter la taille du cache
- Miss de conflit : cache assez grand mais deux ligne correspondant à la même entrée du cache (cf. Cache à correspondance directe)

Pour limiter les conflits, on peut jouer sur :

- Le programme : changer l'organisation mémoire
- Matériel du cache: cache N-way associatif (il y a plusieurs ligne pouvant stocker un même ensemble de ligne mémoire)

> On peut aussi faire un cache Full-associatif : chaque ligne peut être n'importe ou dans le cache

**Politique de remplacement** :

- Random 🎲
- Least Recently Used (LRU) : éviction de la ligne non utilisée depuis le plus longtemps. Stratégie assez efficace.
- Least Frequently Used (LFU)

# Execution des programmes

Exemple RISC:

- lecture instruction en mémoire à @PC
- décodage instruction
- lecture des registres
- Execution de l'OP
- écriture résultats dans registres
- PC <- PC+1

## Pipeline

On peut effectuer en parallèles les étapes n'utilisant pas de ressources en commun. Ainsi, comme pour les mémoires on peut chevaucher des étapes : execution pipelinée (mais nécessite l'ajout de buffer pour sauvegarder les résultats intermédiaires + faut réussir à enchaîner les instructions pour garder le pipeline "nourri"). A chaque cycle, on débute l'éxecution d'une instruction suivante. En théorie, grosse augmentation de débit mais augmentation (parfois petite) de la latence.

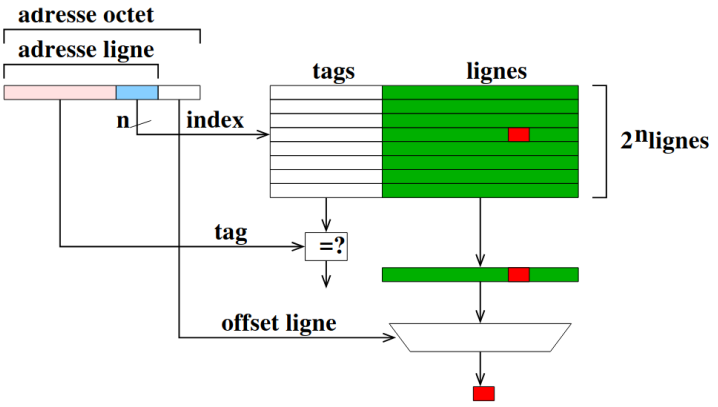
**Problèmes:**

- accès aux registres (par exemple 2 lectures + 1 ecriture)
- aléa de branchement -> on doit attendre les résultats des calculs précédents pour savoir que faire, donc on insère des **noop**… pas génial !

- Données et instructions accédées par un programme dans un intervalle de temps court souvent à des adresses proches
- Ligne : unité de stockage dans le cache (par exemple 64 octets)
- Sélection de la donnée/instruction dans la ligne par les bits de poids faibles de l'adresse

Quand on charge une instruction, on va probablement charger la suivante dans un futur très proche, donc autant le faire à l'avance. On le regroupe donc en ligne et on charge des lignes enières d'un coup

**Correspondance cache / mémoire** :
On a pas encore discuté de comment on accède à une ligne mémoire dans le cache: il faut savoir si elle y est et si oui où.
*Cache à correspondance directe* : chaque ligne mémoie ne peut être que dans une seule ligne de cache spécifique (via format adresses qui imposent la ligne de cache), ce qui permet de tester super vite si une ligne est dans la cache ou non.




**Stratégie pour les écritures** :
Quand on écrit en cache, est-ce qu'on répercute en mémoire ou pas ?

- *Cache write-through* : si on a un hit en écriture, on écrit aussi en mémoire. Si on miss en lecture, alors c'est super simple il suffit d'aller chercher aux niveaux supérieurs car les données y ont été MàJ.
- *Cache Write-back* : on écrit aux niveaux supérieurs que quand on écrase une ligne de cache qui a été modifiée (bit `dirty` qui indique si une ligne de cache a été modifiée).

**Différents types de cache-miss** :

- Miss de démarrage : premier accès à une donnée/instruction
- Miss de capacité : cache trop petit pour contenir tout le programme et/ou toutes ses données

- Une des solution c'est d'anticiper: à la place des *noop*, on met des instructions qui auraient pu s'éxecuter avant
- Une autre solution: **la prédiction de branchement (branchement spéculatif)**.
- Aléa de dépendance: instruction 2 dépend du résultat de l'instruction 1. Trois type de dépendance:
  - Read After Write: i1 écrit qqchose que i2 lit
  - Write after Read: i1 lit qqchose que i2 va remplacer
  - Write after Write: on modifie 2 fois de suite le registre
- Solution: changer l'ordre des instructions quand on peut pour laisser du temps entre deux instructions dépendantes, bypass le cache (ne pas attendre que le résultat y soit écrit, mécanisme matériel)

## Bypass

Réponse slide 76 :
x1 = %rs1(%ri1)=%rd(%ri3)
x2 = %rs2(%ri1)=%rd(%ri3)
x3 = %rs1(%ri2)=%rd(%ri3)
x4 = %rs2(%ri2)=%rd(%ri3)

# Plein d'autres piste d'améliorations de performance en vrac

## BTB, BHT, Processeurs superscalaires (décode plusieurs instructions par cycle)

Le + simple : in order de degré 2 (on lis 2 instructions à chaque cycle)
mécanisme "tampon" d'instructions pour minimiser les problèmes de dépendances entre les instructions

## Processeurs Multicoeurs

Plusieurs coeurs ce calcul sur une même puce
Chaque coeur dispose de son propre cache + du cache partagé
Chaque coeur execute un thread (chaque thread ayant son *Program Counter* et ses *registres*)
(c'est au programmeur de construire le programme pour ça)

- Problèmes: cohérence mémoire entre les coeurs

## SMT (Simultaneous MultiThreading)

Processeur superscalaire: peu efficace car les instructions d'un programme dépendent très souvent de la précédante -> on execute des instrutions de 2 threads/programmes différents en même temps pour éviter ces soucis de dépendances inter-instructions

**Mémoire partagée**

Modèle de comportement permettant la cohérence: copie locale à chaque coeur qui lit une ligne de cache, puis quand on essaye d'écrire qqpart, on invalide les copies dans *tout* les autres coeurs, on attend que les autres coeurs soit tous au courrant avant d'écrire la nouvelle valeur

## Processeurs vectoriels

Instructions sur vecteur, plusieurs fois la même opération d'un coup (par exemple addition).
Adressage mémoire absolue / en stride (xyz…x'y'z'…x"y"z"… = lire un élement sur 2,3,n).

## GPGPU (General Purpose Graphics Processing Unit)

Structure **très hautement parallèle**, peu de structure de contrôle.
2006 : nVidia introduit l'architecture Tesla (SIMT) et CUDA

## Architecture Many-coeurs

Idée de départ: beaucoup de coeurs peu complexes, chaque coeur pouvant avoir son propre OS, mais aussi un OS multiprocesseurs sur plusieurs coeurs.
Problèmes: interconnexion entre les coeurs (bus insuffisant, structures complexes [tores, hypercubes] peu/pas utilisées -> en général mesh 2D)

#### Network On Chip (NoC)

- Commutation de paquet
    - Wormhole switching (besoin de très peu de mémoire dans les routeurs mais calcul délais dans pire cas très compliqué à cause des bloquages indirects (cf slide 134)) : on découpe les données en "flits"
    - Classement des flux par priorité, possibilité de faire de la préemption (on remplace un paquet peu important par un plus prioritaire), Virtuel Channels
    - Délai fortement impacté par l'allocation des tâches aux différents coeurs et les priorités
    - QNoC, BiNoC
- Multiplexage Temporel (construction de l'ordonnancement très compliqué, pb NP-complet + synchronisation de tout les routeurs)