Evolution de l'architecture des processeurs

Synthèse de cours – Par Colin Constans

Avant-propos

Ce document est une synthèse exhaustive du cours théorique d'Architecture des ordinateurs, tenu au S7 par Jean-Luc Scharbar au parcours A. L'intégralité des notions (à la date de décembre 2020) y sont consignés, réorganisées et détaillées pour en faciliter la compréhension. Le document comporte une mise en page compacte de la synthèse (4 dernières pages), imprimable pour le partiel.
Bonne lecture et bon courage.

Introduction : Problématique générale

Augmenter le nombre d'instructions traitées par secondes
→ limites physiques
→ favoriser la duplication à l'augmentation de complexité

I - Classification des architectures

-*Single Instruction Single Data (SISD) :* une instruction sur un cœur (legacy)
-*Signle Instruction Multiple Data (SIMD) :* une même instruction exécutées sur plusieurs cœurs (tâches redondantes, graphiques par ex.)
-*Multiple Instructions Signle Data :* plusieurs instructions différentes sur 1 cœur (très rare)
-*Multiple Instructions Multiple Data :* instructions différentes sur cœurs différents
-SingleProgramMD : un même programme sur plusieurs cœurs (tous indep, d'où les MI)
-MultipleProgramMD : programmes différents en parallèle

II - Mémoire

Débit : quantité max de données lue/écrites en même temps
Latence : temps nécessaire à lecture/écriture
→ causes : longueur des connections, taille mémoire, segmentations (choix du banc long)
→ solutions : hiérarchiser la mémoire :
meilleur tps de rèp. [Registres > Caches > RAM > Hard memory] *meilleure capacité*

A - Types de mémoire

- Non-volatile (pas besoin d'alimentation électrique)
- Volatile statique (besoin d'alim)
- Volatile dynamique (besoin d'alim ET de réécriture régulière)

→ Read-Only Memory (ROM) : non-volatile, seulement réinscriptible en totalité (EPROM : effacement par UV, plusieurs réécritures / EEPROM : idem mais électrique)
→ Read-Write Memory (RWM) : volatile (ex : RAM)

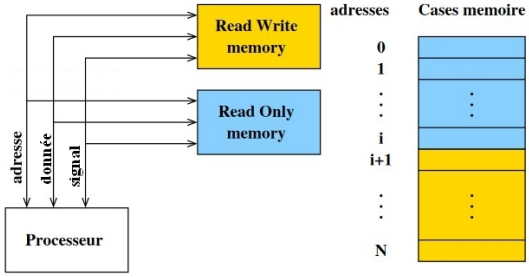
B - Types d'accès

-*Mémoires adressables :* accès à la donnée par adresse (mém. *aléatoire*, ex: RAM) ou à l'adresse par la donnée (mém. *associative*)
-*Mémoire cache :* plusieurs données dans la même case, identifiées par des tags
-*Mémoire séquentielles :* accès dans un ordre défini (files FIFO ou piles LIFO)

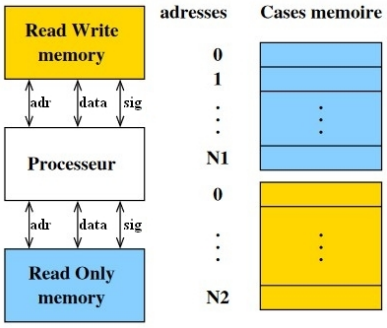
1 - Mémoires adressables

a – Architectures

Architecture de Von Neumann :
RAM et ROM reliées au processeur par 3 bus : adresse, donnée, et signal (précisant lecture ou écriture)



Architecture de Harvard :
idem mais RAM et ROM distinctes (2 jeux d'adresses)



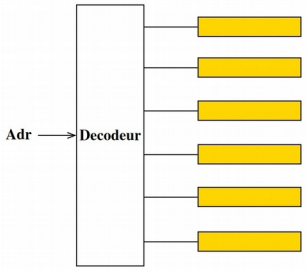
b - Accès

2 principes :
- Architecture double bus (adresse et donnée, comme ci-dessus)
- Architecture simple bus (et un bit, dit ALE, indiquant si le bus présente adresse ou donnée)
  → nécessite 2 étapes pour une écriture

Contrôleur d'accès : sélectionne la case mémoire à lire à partir de l'adresse reçue sur son bus

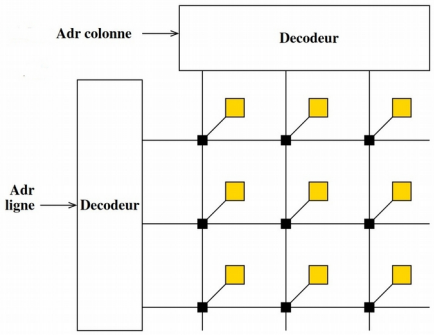
3 types de contrôleurs d'accès :

- *Adressage linéaire :* une case mémoire par ligne, décodeur adresse <=> ligne (→ décodeur très complexe)



- *Adressage par coïncidence :* une case mémoire au croisement d'une ligne et d'une colonne, un décodeur pour chaque (→ décodeurs plus petits/simples), peut nécessiter une ou deux adresses (colonne et ligne séparées)

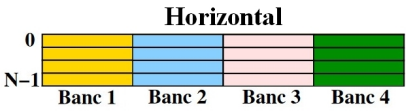
- *Row buffer :* idem que adressage par coïncidence, mais stockage de la ligne sélectionnée dans un buffer, puis lecture de la bonne colonne dans ce buffer



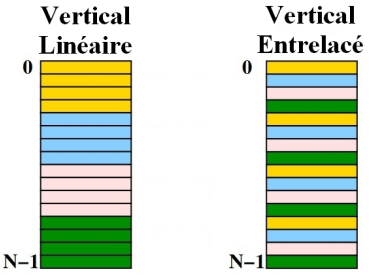
c – Améliorations des performances

- *Mémoire synchrone :* ajout de registres pour maintenir les adresses et signaux en entrée tout le long de l'accès
- *Mémoire synchrone avec pipeline :*
  → séparer l'accès en plusieurs étapes (ex : présentation adr puis lecture)
  → réaliser 1ère étape d'un accès (lecture) en même temps que la dernière de la précédente
  → plus d'accès/sec, mais contrôleur plus complexe
- *Mémoire Dual/Quad Data Rate :* plus de données transmise en même temps (2x/4x), à l'aide d'un bus plus large / de plus haute fréquence

d – Segmentation de la mémoire



Bancs mémoire : unités composant la mémoires, plusieurs arrangements :



- Arrangement *horizontal* : chaque banc contient une partie de la case mémoire → accès simultané à tous
- Arrangement *vertical* : chaque banc contient un set de cases mémoires, sélectionné par bit de poids fort dans l'adr
  → linéaire (lectures successives sur des adr consécutives impossible)
  → entrelacé (lectures successives possibles)

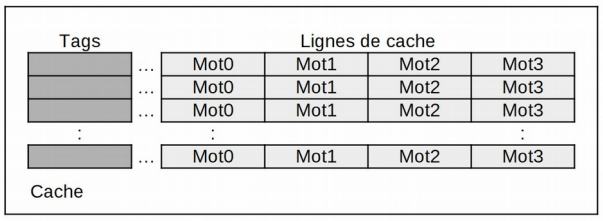
2 - Cache

→ Petite mémoire dans le processeur, stockant données/instructions accédées récemment
→ Lecture à chaque accès mémoire, gain de temps si *Hit* (donnée déjà présente), lecture de la RAM si *Miss*

Intérêt : Localité temporelle → les programmes réutilisent souvent des données/instructions utilisées récemment

a - Structure

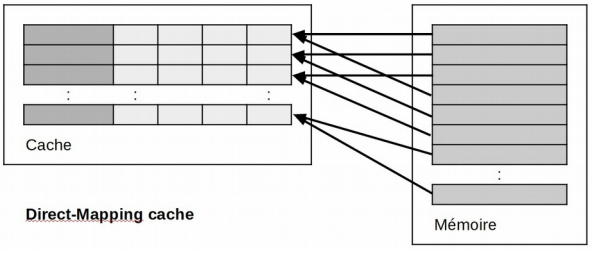
Localité spatiale : les programmes utilisent souvent des données/instructions stockées proches en mémoire



→ *Ligne de cache :* unité de stockage, peut contenir plusieurs mots (bit de poids faible dans adr sélectionne le mot)

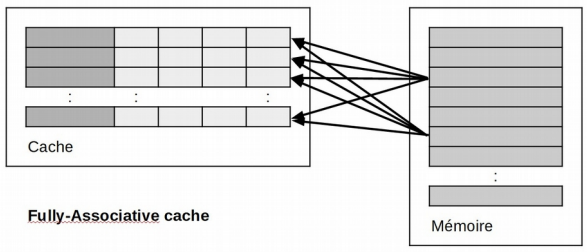
Chaque ligne mémoire (du niveau supérieur) peut potentiellement être stockée dans certaines lignes du cache qui lui sont attribuées, selon sa structure

3 structures :



- *Direct-mapped Cache :* structure simple, chaque ligne mémoire ne peut être stockée que dans un emplacement spécifique du cache → beaucoup de Miss car peu flexible

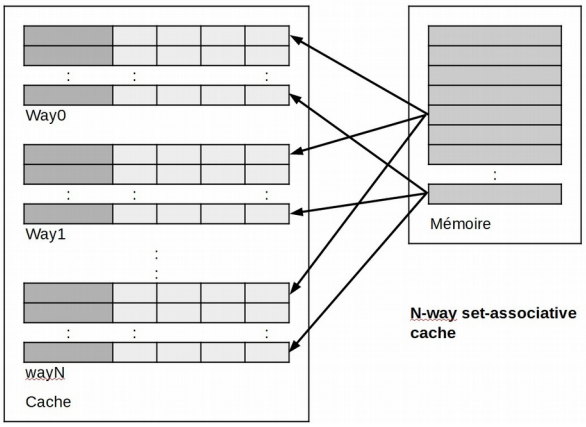
- *Full-associative cache :* chaque ligne cache peut accueillir n'importe quelle ligne mémoire → complexe niveau logique



- *N-way set-associative cache* (mix des deux précédents) :

→ N sets de lignes (way0, way1 …)

→ Chaque ligne mémoire peut être stockée un emplacement de chaque way (donc N possibilités)

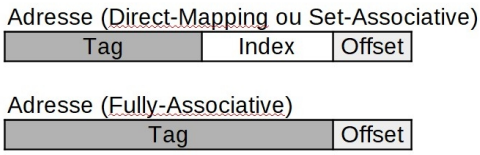


Adressage :

- Tag : « clé » correspondant à une ligne mémoire

- Index (dans le cas non-associatif) : indique à quelle ligne chercher (vérification ensuite avec le tag que c'est la bonne donnée)

- Offset : indique le décalage de lecture pour sélectionner le mot



b – Améliorations

-Plusieurs caches séparés (pour données et instructions) → limite les conflits d'accès
-Caches hiérarchisés (L1, L2, …, plus petit = plus rapide) → optimise l'accès

c - Ecriture (2 méthodes)

- *Cache write-through (écriture transmise) :* envoi systématique de l'ordre d'écriture au niveau suivant de hiérarchie mémoire

→ Hit en lecture : lecture du cache
→ Hit en écriture : Mise à jour du cache
→ Miss en lecture : écriture de la donnée dans le cache depuis niveau supérieur
→ Miss en écriture : 2 options
- allocation en écriture (copie de la ligne manquante dans le cache puis mise à jour)
- sans allocation (mise à jour au niveau suivant sans changement dans le cache)

- *Cache write-back (écriture différée) :*

→ Hit en lecture : lecture du cache
→ Hit en écriture : Mise à jour du cache et bit « dirty » = 1
→ Miss en lecture : écriture de la donnée dans le cache sur la ligne présente
→ dirty = 0 → ligne non-modifiée → on l'écrase
→ dirty = 1 → ligne modifiée → on la stocke dans le niveau supérieur
→ Miss en écriture : idem que pour write-through, mêmes conditions que Miss Lecture

d – Gestion des Miss

Différents types de Miss :
- Démarrage : premier accès à une donnée
    → solution : prendre des lignes plus longues
- Capacité : cache trop petit pour tout contenir
    → solution : agrandir le cache
- Conflit : 2 lignes mémoire correspondent à la même entrée cache
    → solution : placer les données à des adr proches (au niveau programme)
    → solution : set-associative cache

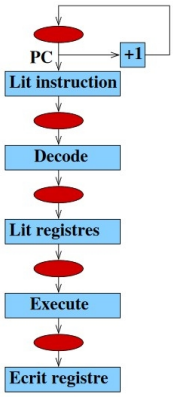
Comment choisir où écrire la ligne manquante ? Plusieurs politiques de remplacement :

-Random : éviction d'une ligne aléatoire parmi celles dans les bonnes entrées
-Last Recently Used (LRU) : éviction de la ligne inutilisée depuis le plus longtemps
    → ex pour un cache 2-way : 1 bit MRU (Most Recently Used) par way,
    MRU = 0 si ligne accédée dans way0, MRU = 1 sinon, et éviction de la ligne dans le way ne correspondant pas à MRU

III – Accélération de l'exécution des programmes

Exécution RISC standard :
lecture instruction > décodage > lecture registres > exécution > écriture résultats > instruction suivante

A – Pipelines



Principe : commencer une instruction sans attendre fin de la précédente (à chaque cycle, une nouvelle instruction commence)

→ Augmente le débit (= nbr d'instructions/sec) mais aussi la latence (durée de l'instruction) car manipulations mémoires supplémentaires

B – Limites

1 – Aléas structurels

Conflits de ressources, par ex lecture et écriture simultanée du registre
→ solution : plusieurs e/s sur les bancs de registres

2 – Aléas d'accès mémoires

Accès mémoire très lent en pratique (plusieurs dizaines de cycles)

3 – Aléas de contrôle

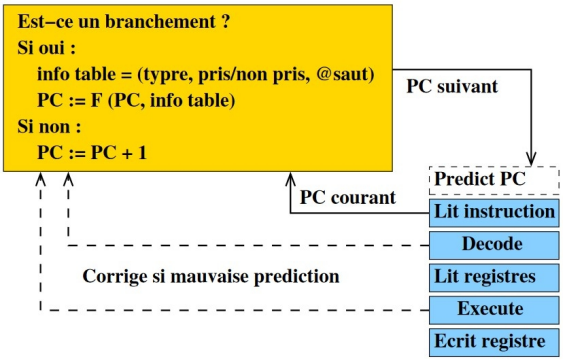
Les branchements (type « jump ») demande le traitement (lecture/décodage/exécution) d'une nouvelle instruction, faisant patienter l'appelante
→ solution : réorganiser les instructions pour que les opérations soient consécutives
→ solution : prédire les branchements à partir des anciens comportements

a - Classification des branchements

- Conditionnel (seulement si condition évaluée vrai à l'étape exécution)
- Inconditionnel (systématique)
- Immédiat (adr connue à la compilation)
- Indirect (adr lue dans un registre)
    - → conditionnel immédiat : for/while, if/then/else, ...
    - → inconditionnel immédiat : appel de fonction, boucles, sortie de if, ...
    - → inconditionnel indirect : retour fonction, pointeur, ...
    - → conditionnel indirect : très rare

b - Prédiction des branchements

Principe : Stockage des instructions de branchement, de leur type, leur cible et leur acceptation (si branch. conditionnel) dans une table
→ permet d'anticiper en spéculant sur le comportement précédent



Cas de mauvais branchement :
On vide le pipeline (perte de temps, dans notre ex : 1 cycle si inconditionnel immédiat, 3 si conditionnel ou indirect), et on utilise ce temps pour corriger l'adr d'instruction courante (registre PC) et la table

Exemple de table : Branch Target Buffer (BTB, ou encore BTAC ou Next-Fetch Predictor)
- Taille variable (entre 8 et 2k entrées), associativité (de 1 à 4)
- Stocke les branchements inconditionnels et conditionnels pris
    - → Lors d'une lecture d'instruction : si présente dans le BTB → saut à l'adr indiquée
    - → Si mauvaise prédiction : correction
        - → saut manqué → ajout dans le BTB
        - → mauvaise adr d'arrivée → correction dans le BTB
        - → branchement prédit « pris » mais non-pris → retrait du BTB

Amélioration : BTB efficace pour branch. inconditionnels immédiats mais pas pour conditionnels/retours de fonctions, car le branch. change souvent
→ solution : compteur incrémenté si branch. pris et décrémenté si non-pris, prédiction si compteur > seuil
(2 pr un compteur 2 bits, le bit de poids fort indique la prédiction)
    - → un compteur par entrée, conserve associativité
    - → compteurs dans table spécifique (Branch History Table, BHT), permet de libérer de l'espace (pas de tags et entrées supprimables dans le BTB si compteur à 1)

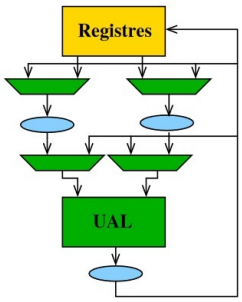
4 – Aléas de dépendance

3 types de dépendance (exécution I1 puis I2):

-Read After Write (RAW, ou dep. vraie) : I2 utilise le résultat de I1
→ solution : changer l'ordre des instructions
→ solution : mécanisme de byPass (utiliser le résultat sans attendre l'écriture)
-Write After Read (WAR, ou anti-dep.) : I2 écrit dans registre lu par I1
-Write After Write (WAW, ou dep de sortie) : I1 et I2 écrivent dans le même registre

a – Mécanisme de byPass

Principe : utiliser le résultat sans attendre l'écriture, en l'envoyant par un byPass parallèle et l'intégrant à l'aide de multiplexeurs



b – Instructions de longeurs différentes

- La plus longue dicte la fréquence
- Problème si pipelines sur opérateurs (les opérations se font simultanément), ex : si I1 plus longue que I2 : I2 va écrire avant I1
→ solution : faire attendre I2 entre exécution et écriture (mais byPass plus complexe, et problème si différence de longueur trop importante)

C - Parallélisation

1 - Processeurs SuperScalaires

Processeurs pouvant décoder plusieurs instructions (du même thread) par cycle
→ SuperScalaire de degré n : max n instructions décodées en un cycle
→ Efficacité fonction du parallélisme d'instruction du programme (ie de la proportion d'instructions pouvant être exécutées en parallèle)
→ Complexité matérielle proportionnelle au degré (limite courante : n=6, causes : nbr de ports sur bancs de registres, byPass trop complexe, parallélisme des programmes actuels)

Plusieurs types de fonctionnement :
- SuperScalaire *in-order* : instructions exécutées dans l'ordre du programme
- SuperScalaire *out-of-order* : instructions lancées dans un ordre différente

Exemple : in-order de degré 2 → on double tous les équipements (décodeur, ports de registre, UAL …) fréquemment utilisés

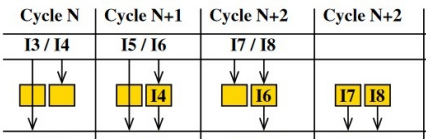
Problème : Si deux instructions parallèles dépendantes : on retarde l'une d'entre elles, éventuellement en la couplant avec une des instructions suivante (qui elle est indépendante, technique du décalage de groupements)

Exemple : I2 dépend de I1 : retard, et couplage avec I3 indép de I2

	Cycle N	Cycle N+1	Cycle N+2	
Lit instruction	I3 / I4	I5 / I4	I7 / I6	
Decode	I1 / I2	I3 / I2	I5 / I4	
Lit registres		I1	I3 / I2	
Execute			I1	
Ecrit registre				

Mise en œuvre : buffer qui mémorise l'instruction dans le cas d'une dépendance (ex : I2 dépend de I1 et I5 de I4)

	Cycle N	Cycle N+1	Cycle N+2	Cycle N+2
Lit instruction	I3 / I4	I5 / I6	I7 / I8	
Decode	I1 / I2	I3 / I2	I5 / I4	I5 / I6
Lit registres		I1	I3 / I2	I4
Execute			I1	I3 / I2
Ecrit registre				I1



2 - Processeurs multi-cœurs

Principe : Permettre le parallélisme de threads (1 cœur, 1 thread, chacun son compteur d'instruction mais mémoire partagée, gérée par le programmeur)
→ Caches locaux (un par cœur) ou partagés

3 - Processeur SMT (Simultaneous Multi-Threading)

Principe : Plusieurs threads par cœur
→ Plusieurs compteurs d'instructions :
- Lecture d'une instruction d'un thread lors d'un cycle,
- Lecture d'une instruction d'un second thread lors du cycle suivant
- etc …

→ ressources partagés : caches, opérateurs, mais aussi mémoire (même espace d'adressage)

Problème : nécessite de définir un mécanisme de cohérence
→ différents niveaux de caches (ex : L1 locaux, L2 partagés)
→ Lecture d'une ligne : copie sur chaque cœur
→ écriture : invalidation de toutes les copies avant modification
→ Cas de caches locaux write-back → vérification des caches des autres cœurs, et lecture d'une éventuelle copie « dirty » de l'un d'eux

→ Application possible sur processeurs multi-cœurs
→ Application possible sur processeurs superscalaires mais difficile en pratique d'exécuter autant d'instruction/cycle que son degré

4 - Processeurs vectoriels

Principe : processeurs adaptés au calcul sur tableaux, matrices, vecteurs
→ opérations sur des vecteurs = données de même taille et type

Accès mémoire :
-instructions de lecture/écriture de vecteurs (type load/store)
-stockage dans registres vectoriels
-adressage absolu ou avec *stride* (saut de cases) selon contiguïté ou non des données du vecteur

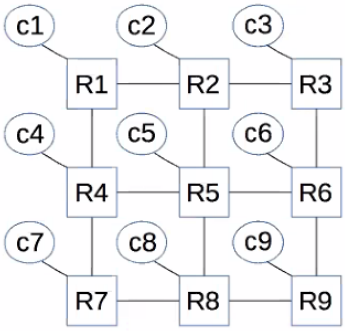
Mémoire organisée pour faciliter lecture/écriture de vecteurs :
- plusieurs bancs → accès parallèle
- mémoire entrelacée → adr consécutives dans bancs consécutifs
- peu adapté au stride → organisation plus complexe

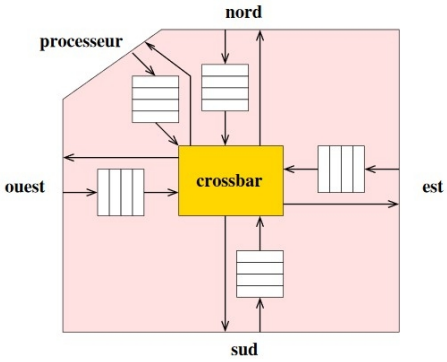
Vectorisation (exemple) : duplication du corps d'une boucle (si instructions indépendantes) pour réduire le nombre de tours nécessaire (ex : 25 tours traitant chacun 4x l'opération en parallèle plutôt que 100 tours traitant une seule fois l'opération)

5 - Architectures many-cœurs

Principe : Grand nombre de cœurs, de complexité moindre, pour gagner en puissance
→ problème d'interconnexion : utilisation de mesh 2D (bus insuffisants, et structures complexes (tores, hypercubes) trop techniques)

Architecture ci-contre : C=cœur / R = routeur





Routeur : route l'information (routage X-Y : transmission jusqu'à la bonne colonne puis jusqu'à la bonne ligne)

Problème : les routeurs ont très peu de mémoire
→ solution : « *wormhole switching* » : paquets découpés en flits, (le buffer d'un routeur ne peut stocker qu'un seul flit), le premier détermine le routage, les autres suivent en flot

→ Entraîne des *blocages* (quand 2 flots se croisent sur un routeur) :
- Directs : un flot occupe le buffer, les autres flots sont bloqués tant que le premier n'a pas libéré la place
- Indirects : un flot est bloqué par un autre, lui même bloqué par un troisième (le troisième bloque indirectement le premier)

Evolution de l'architecture des processeurs

Synthèse de cours – par Colin Constans

Introduction : Problématique générale

Augmenter le nombre d'instructions traitées par secondes
→ limites physiques
→ favoriser la duplication à l'augmentation de complexité

I - Classification des architectures

-*Single Instruction Single Data (SISD) :* une instruction sur un cœur (legacy)
-*Signle Instruction Multiple Data (SIMD) :* une même instruction exécutées sur plusieurs cœurs (tâches redondantes, graphiques par ex.)
-*Multiple Instructions Signle Data :* plusieurs instructions diff. Sur 1 cœur (très rare)
-*Multiple Instructions Multiple Data :* instructions différentes sur cœurs différents
-SingleProgramMD : un même prgrm sur pls cœurs indep
-MultipleProgramMD : programmes différents en parallèle

II - Mémoire

Débit : quantité max de données lue/écrites en même temps
Latence : temps nécessaire à lecture/écriture
→ causes : longueur des connections, taille mémoire, segmentations (choix du banc long)
→ solutions : hiérarchiser la mémoire :
Tps de rèp. [Registres>Caches>RAM>Hard memory] C*apacité*

A - Types de mémoire

- Non-volatile (pas besoin d'alim élec)
- Volatile statique (besoin d'alim)
- Volatile dynamique (besoin d'alim ET de réécriture régulière)

→ Read-Only Memory (ROM) : non-volatile, seulement réinscriptible en totalité (EPROM : effacement par UV, plusieurs réécritures / EEPROM : idem mais électrique)
→ Read-Write Memory (RWM) : volatile (ex : RAM)

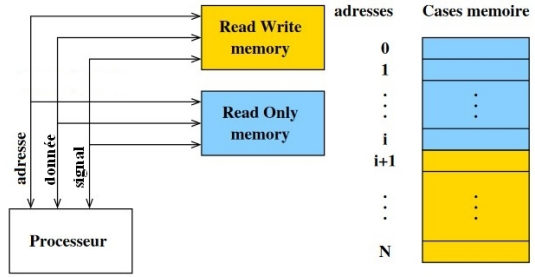
B - Types d'accès

-*Mémoires adressables :* accès à la donnée par adresse (mém. *aléatoire*, ex: RAM) ou à adresse par donnée (mém. *associative*)
-*Mémoire cache :* plusieurs données dans la même case, identifiées par des tags
-*Mémoire séquentielles :* accès dans un ordre défini (files FIFO ou piles LIFO)

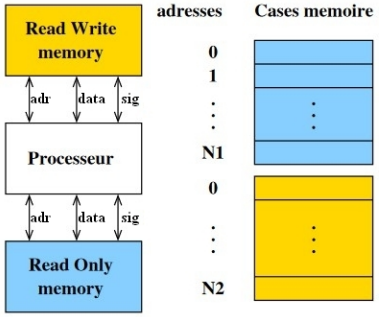
1 - Mémoires adressables

a – Architectures

- *Architecture de Von Neumann :*RAM et ROM reliées au processeur par 3 bus : adresse, donnée, et signal (précisant lecture ou écriture)



- *Architecture de Harvard :* idem mais RAM et ROM distinctes (2 jeux d'adresses)



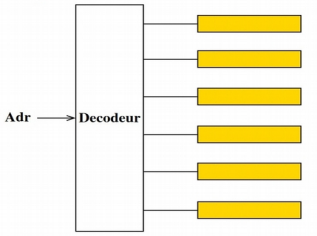
b - Accès

2 principes :
- Architecture double bus (adresse et donnée, comme ci-dessus)
- Architecture simple bus (et un bit, dit ALE, indiquant si le bus présente adresse ou donnée)
→ nécessite 2 étapes pour une écriture

Contrôleur d'accès : sélectionne la case mémoire à lire à partir de l'adresse reçue sur son bus

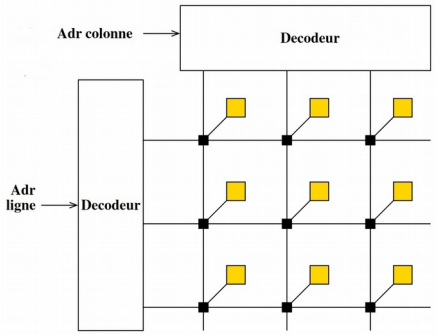
3 types de contrôleurs d'accès :

- *Adressage linéaire :* une case mémoire par ligne, décodeur adresse <=> ligne (→ décodeur très complexe)



- *Adressage par coïncidence :* une case mémoire au croisement d'une ligne et d'une colonne, un décodeur pour chaque (→ décodeurs plus petits/simples), peut nécessiter une ou deux adresses (colonne et ligne séparées)

- *Row buffer :* idem que adressage par coïncidence, mais stockage de la ligne sélectionnée dans un buffer, puis lecture de la bonne colonne dans ce buffer



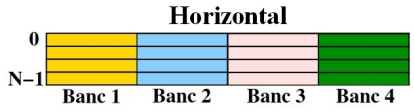
c – Améliorations des performances

- *Mémoire synchrone :* ajout de registres pour maintenir les adresses et signaux en entrée tout le long de l'accès
- *Mémoire synchrone avec pipeline :*
→ séparer l'accès en plusieurs étapes (ex : présentation adr puis lecture)
→ réaliser 1ère étape d'un accès (lecture) en même temps que la dernière de la précédente
→ plus d'accès/sec, mais contrôleur plus complexe
- *Mémoire Dual/Quad Data Rate :* plus de données transmise en même temps (2x/4x), à l'aide d'un bus plus large / de plus haute fréquence

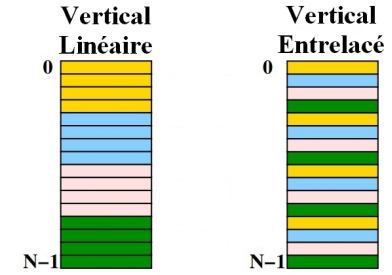
d – Segmentation de la mémoire

Bancs : unités composant la mémoires, plusieurs arrangements :

- Arrangement *horizontal* : chaque banc contient une partie de la case mémoire
→ accès simultané à tous



- Arrangement *vertical* : chaque banc contient un set de cases mémoires, sélectionné par bit de poids fort dans l'adr
→ linéaire (lectures successives sur des adr consécutives impossible)
→ entrelacé (lectures successives possibles)



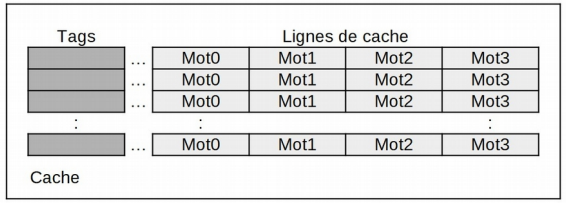
2 - Cache

→ Petite mémoire dans le processeur, stockant données/instructions accédées récemment
→ Lecture à chaque accès mémoire, gain de temps si *Hit* (donnée déjà présente), lecture de la RAM si *Miss*

Intérêt : Localité temporelle → les programmes réutilisent souvent des données/instructions utilisées récemment

a - Structure

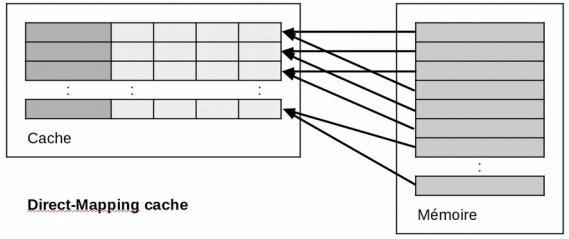
Localité spatiale : les programmes utilisent souvent des données/instructions stockées proches en mémoire
→ *Ligne de cache :* unité de stockage, peut contenir plusieurs mots (bit de poids faible dans adr sélectionne le mot)



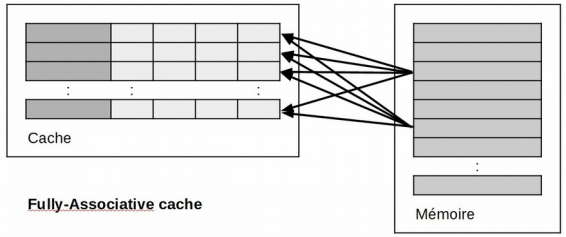
Chaque ligne mémoire (du niveau supérieur) peut potentiellement être stockée dans certaines lignes du cache qui lui sont attribuées, selon sa structure

3 structures :

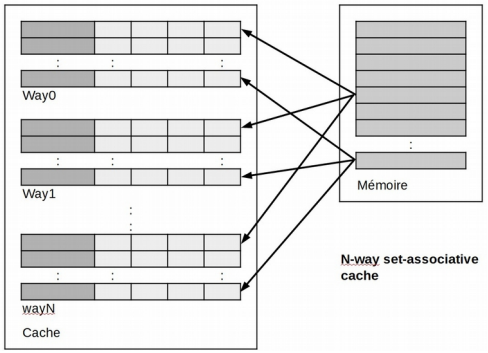
- *Direct-mapped Cache :* structure simple, chaque ligne mémoire ne peut être stockée que dans un emplacement spécifique du cache → beaucoup de Miss car peu flexible



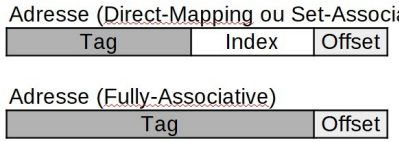
- *Full-associative cache :* chaque ligne cache peut accueillir n'importe quelle ligne mémoire → complexe niveau logique



- *N-way set-associative cache* (mix des deux précédents) :
  → N sets de lignes (way0, way1 …)
  → Chaque ligne mémoire peut être stockée un emplacement de chaque way (donc N possibilités)



Adressage :
- Tag : « clé » correspondant à une ligne mémoire
- Index (dans le cas non-associatif) : indique à quelle ligne chercher (vérification ensuite avec le tag que c'est la bonne donnée)
- Offset : indique le décalage de lecture pour sélectionner le mot



b – Améliorations

-Plusieurs caches séparés (pour données et instructions)
→ limite les conflits d'accès
-Caches hiérarchisés (L1, L2, …, plus petit = plus rapide)
→ optimise l'accès

c - Ecriture (2 méthodes)

- *Cache write-through (écriture transmise) :* envoi systématique de l'ordre d'écriture au niveau suivant de hiérarchie mémoire
  → Hit en lecture : lecture du cache
  → Hit en écriture : Mise à jour cache
  → Miss en lecture : écriture de la donnée dans le cache depuis niveau supérieur
  → Miss en écriture : 2 options
    - allocation en écriture (copie de la ligne manquante dans le cache puis mise à jour)
    - sans allocation (mise à jour au niveau suivant sans changement dans le cache)

- *Cache write-back (écriture différée) :*
  → Hit en lecture : lecture du cache
  → Hit en écriture : Mise à jour cache et bit « dirty » = 1
  → Miss en lecture : écriture de la donnée dans le cache sur la ligne courante
    → dirty=0 → ligne non-modifiée → on l'écrase
    → dirty=1 → ligne modifiée → on la stocke dans le niveau supérieur
  → Miss en écriture : idem que pour write-through, mêmes conditions que Miss Lecture

d – Gestion des Miss

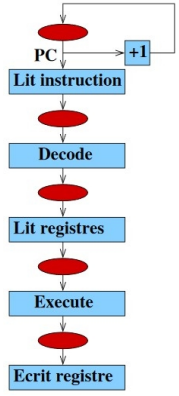
Différents types de Miss :
- Démarrage : premier accès à une donnée
  → solution : prendre des lignes plus longues
- Capacité : cache trop petit pour tout contenir
  → solution : agrandir le cache
- Conflit : 2 lignes mémoire correspondent à la mm entrée cache
  → solution : placer les données à des adr proches (au niveau programme)
  → solution : set-associative cache

Comment choisir où écrire la ligne manquante ?
Plusieurs politiques de remplacement :

-Random : éviction d'une ligne aléatoire parmi celles dans les bonnes entrées
-Last Recently Used (LRU) : éviction de la ligne inutilisée depuis le plus longtemps
→ ex pour un cache 2-way : 1 bit MRU (Most Recently Used) par way,
→ MRU=0 si ligne accédée dans way0, MRU=1 sinon
→ Eviction de la ligne du way ≠ MRU

III – Accélération de l'exécution des programmes

Exécution RISC standard :
lecture instruction > décodage > lecture registres > exécution > écriture résultats > instruction suivante



A – Pipelines

Principe : commencer une instruction sans attendre fin de la précédente (à chaque cycle, une nouvelle instruction commence)

→ Augmente le débit (=nbr d'instructions/sec) mais aussi la latence (durée de l'instruction) car manipulations mémoires supplémentaires

B – Limites

1 – Aléas structurels

Conflits de ressources, par ex lecture et écriture simultanée du registre
→ solution : plusieurs e/s sur les bancs de registres

2 – Aléas d'accès mémoires

Accès mémoire très lent en pratique (plusieurs dizaines de cycles)

3 – Aléas de contrôle

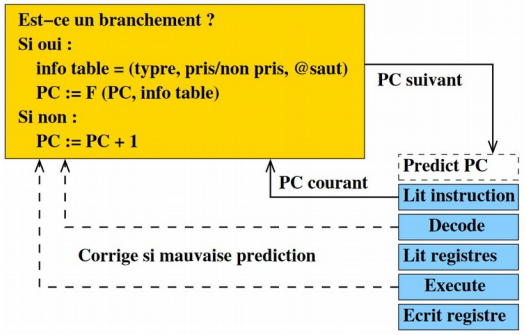
Les branchements (type « jump ») demande le traitement (lecture/décodage/exécution) d'une nouvelle instruction, faisant patienter l'appelante
→ solution : réorganiser les instructions pour que les opérations soient consécutives
→ solution : prédire les branchements à partir des anciens comportements

a - Classification des branchements

- Conditionnel (seulement si condition évaluée vrai à l'étape exécution)
- Inconditionnel (systématique)
- Immédiat (adr connue à la compilation)
- Indirect (adr lue dans un registre)
  - → conditionnel immédiat : for/while, if/then/else, ...
  - → inconditionnel immédiat : appel de fonction, boucles, sortie de if, ...
  - → inconditionnel indirect : retour fonction, pointeur, ...
  - → conditionnel indirect : très rare

b - Prédiction des branchements

Principe : Stockage des instructions de branch, de leur type, leur cible et état d'acceptation (si conditionnel) dans une table
→ permet d'anticiper en spéculant sur le comportement précédent



Cas de mauvais branchement :
On vide le pipeline (perte de temps, dans notre ex : 1 cycle si inconditionnel immédiat, 3 si conditionnel ou indirect), et on utilise ce temps pour corriger l'adr d'instruction courante (registre PC) et la table

Exemple de table : Branch Target Buffer (BTB, ou encore BTAC ou Next-Fetch Predictor)
- Taille variable (entre 8 et 2k entrées), associativité (de 1 à 4)
- Stocke les branchements inconditionnels et conditionnels pris
  - → Lors d'une lecture d'instruction : si présente dans le BTB → saut à l'adr indiquée
  - → Si mauvaise prédiction : correction
    - → saut manqué → ajout dans le BTB
    - → mauvaise adr d'arrivée → correction BTB
    - → branchement prédit « pris » mais non-pris → retrait du BTB

Amélioration : BTB efficace pour branch. inconditionnels immédiats mais pas pour conditionnels/retours de fonctions, car le branch. change souvent
→ solution : compteur incrémenté si pris et décrémenté si non-pris, prédiction si compteur > seuil (2 pr un compteur 2 bits, bit de poids fort indique prédiction). 2 versions :
- → un compteur par entrée, conserve associativité
- → compteurs dans table spécifique (Branch History Table, BHT), libère de l'espace (pas de tags et entrées supprimables dans le BTB si compteur à 1)

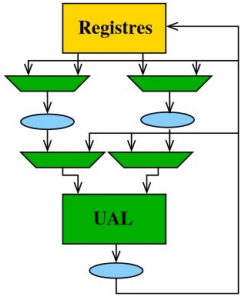
4 – Aléas de dépendance

3 types de dépendance (exécution I1 puis I2):
-Read After Write (RAW, ou dep. vraie) : I2 utilise résultat de I1
- → solution : changer l'ordre des instructions
- → solution : mécanisme de byPass

-Write After Read (WAR, ou anti-dep.) : I2 écrit reg. lu par I1
-Write After Write (WAW, ou dep de sortie) : I1 et I2 écrivent dans le même registre

a – Mécanisme de byPass

Principe : utiliser le résultat sans attendre l'écriture, en l'envoyant par un byPass parallèle et l'intégrant à l'aide de multiplexeurs



b – Instructions de longeurs différentes

- La plus longue dicte la fréquence
- Problème si pipelines sur opérateurs (les opérations se font simultanément), ex: si I1 plus longue que I2, I2 va écrire avant I1
  - → solution : faire attendre I2 entre exécution et écriture

(mais byPass plus complexe, et pb si longueurs diffèrent trop)

C - Parallélisation

1 - Processeurs SuperScalaires

Principe : un processeur SuperScalaire de degré n peut décoder n instructions (du même thread) par cycle

→ Efficacité fonction du parallélisme d'instruction du programme (ie % d'instructions exécutables en parallèle)
→ Complexité matérielle proportionnelle au degré
(limite courante : n=6, causes : nbr de ports sur bancs de registres, byPass trop complexe, parallélisme des prgms actuels)

Plusieurs types de fonctionnement :
- *in-order* : instructions exécutées dans l'ordre du programme
- *out-of-order* : instructions lancées dans un ordre différente

Exemple : in-order de degré 2 → on double tous les équipements (décodeur, ports de registre, UAL …) fréquemment utilisés

Problème : Si deux instructions parallèles dépendantes : on retarde l'une d'entre elles, éventuellement en la couplant avec une des instructions suivante (qui elle est indépendante, technique du décalage de groupements)

Ex : I2 dépend de I1 : retard / couplage avec I3 indép de I2

	Cycle N	Cycle N+1	Cycle N+2	
Lit instruction	I3 / I4	I5 / I4	I7 / I6	
Decode	I1 / I2	I3 / I2	I5 / I4	
Lit registres		I1	I3 / I2	
Execute			I1	
Ecrit registre				

Mise en œuvre : buffer qui mémorise l'instruction dans le cas d'une dépendance (ex : I2 dépend de I1 et I5 de I4)

	Cycle N	Cycle N+1	Cycle N+2	Cycle N+2
Lit instruction	I3 / I4	I5 / I6	I7 / I8	
		I4	I6	I7 I8
Decode	I1 / I2	I3 / I2	I5 / I4	I5 / I6
Lit registres		I1	I3 / I2	I4
Execute			I1	I3 / I2
Ecrit registre				I1

2 - *Processeurs multi-cœurs*

Principe : Permettre le parallélisme de threads (1 cœur, 1 thread, chacun son compteur d'instruction mais mémoire partagée, gérée par le programmeur)
→ Caches locaux (un par cœur) ou partagés

3 - *Processeur SMT (Simultaneous Multi-Threading)*

Principe : Plusieurs threads par cœur
→ Plusieurs compteurs d'instructions :
- Lecture d'une instruction d'un thread lors d'un cycle,
- Lecture d'une instruction d'un 2nd thread lors du suivant
- etc …
  → ressources partagés : caches, opérateurs, mais aussi mémoire (même espace d'adressage)

Problème : nécessite de définir un mécanisme de cohérence
→ différents niveaux de caches (ex : L1 locaux, L2 partagés)
→ Lecture d'une ligne : copie sur chaque cœur
→ écriture : invalidation de toutes les copies avant modif.
→ Cas de caches locaux write-back → vérification des caches des autres cœurs, et lecture d'une éventuelle copie « dirty » de l'un d'eux

→ Application possible sur processeurs multi-cœurs
→ Application possible sur processeurs superscalaires mais difficile en pratique d'exécuter autant d'instruction/cycle que son degré

4 - *Processeurs vectoriels*

Principe : processeurs adaptés au calcul sur tableaux, matrices, vecteurs
→ opérations sur des vecteurs = données de même taille et type

Accès mémoire :
-instructions de lecture/écriture de vecteurs (type load/store)
-stockage dans registres vectoriels
-adressage absolu ou avec *stride* (saut de cases) selon contiguïté ou non des données du vecteur

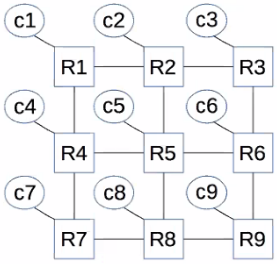
Mémoire organisée pour faciliter lecture/écriture de vecteurs :
- plusieurs bancs → accès parallèle
- mémoire entrelacée → adr consécutives dans bancs consécutifs
- peu adapté au stride → organisation plus complexe
Vectorisation (exemple) : duplication du corps d'une boucle (si instructions indépendantes) pour réduire le nombre de tours nécessaire (ex : 25 tours traitant chacun 4x l'opération en parallèle plutôt que 100 tours traitant une seule fois l'opération)

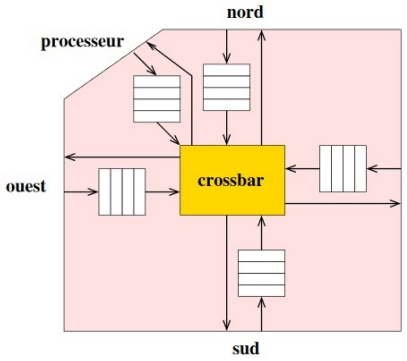
5 - *Architectures many-cœurs*

Principe : Grand nombre de cœurs, de complexité moindre, pour gagner en puissance

→ problème d'interconnexion : utilisation de mesh 2D (bus insuffisants, et structures complexes (tores, hypercubes) trop techniques)

Architecture ci-contre :
C=cœur / R = routeur





Routeur : route l'information (routage X-Y : transmission jusqu'à la bonne colonne puis jusqu'à la bonne ligne)

Problème : les routeurs ont très peu de mémoire
→ solution : « *wormhole switching* » : paquets découpés en flits, (le buffer d'un routeur ne peut en stocker qu'un seul), le 1er détermine le routage, les autres suivent en flot

→ Entraîne des *blocages* (qd 2 flots se croisent sur un routeur) :
- Directs : un flot occupe le buffer, les autres flots sont bloqués tant que le premier n'a pas libéré la place
- Indirects : un flot est bloqué par un autre, lui même bloqué par un troisième (le troisième bloque indirectement le premier)