

Evolutions des architectures de processeurs

Jean-Luc Scharbarg - ENSEEIHT - Dpt. SN

Septembre 2024

Le calcul: un besoin millénaire

- Pour le commerce, l'administration, . . .
- Symboles numériques
- Méthodes et outils pour compter et calculer
- Abaque (table à compter): un des outils les plus anciens
 - ▶ Abaques à jetons



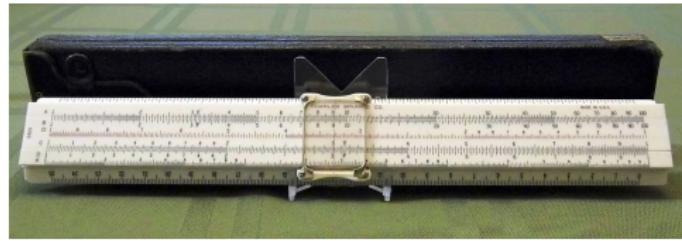
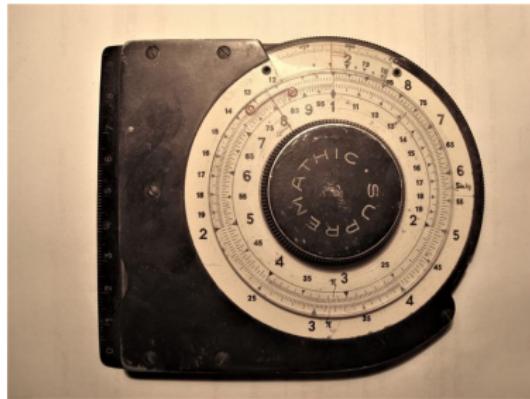
Le calcul: un besoin millénaire

- Pour le commerce, l'administration, ...
- Symboles numériques
- Méthodes et outils pour compter et calculer
- Abaque (table à compter): un des outils les plus anciens
 - ▶ Abaques à boules (bouliers)



Le calcul: un besoin millénaire

- La règle à calcul inventée par William Oughtred (1632)



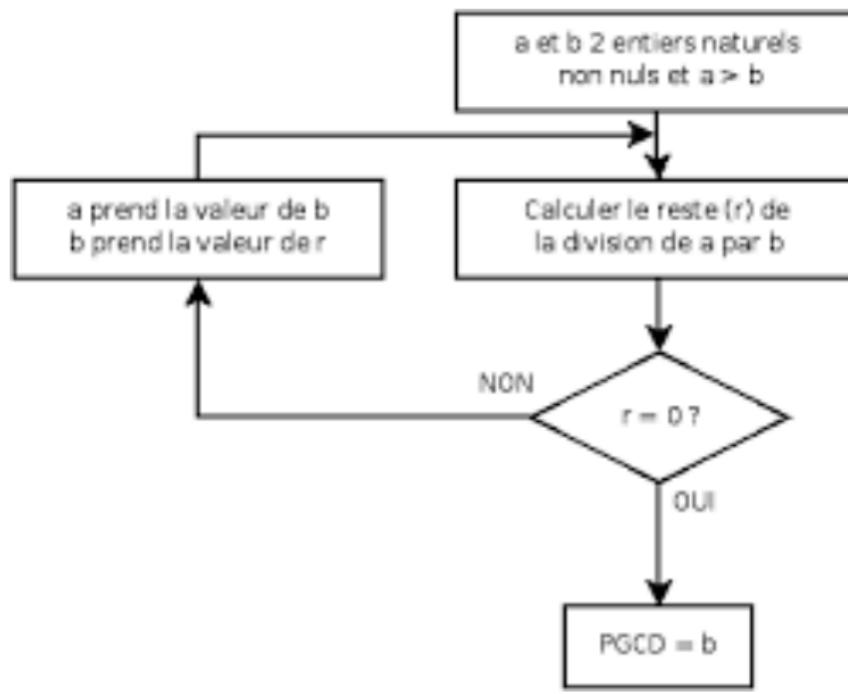
Des algorithmes pour calculer

- Eratosthène: astronome, géographe, philosophe et mathématicien grec (-276, -194)
 - ▶ Crible pour la recherche des nombres premiers

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Des algorithmes pour calculer

- Euclide: mathématicien grec (vers -300)
 - ▶ Algorithme pour le calcul du PGCD de deux entiers



Premiers calculateurs mécaniques

- Calculateur mécanique de Wilhelm Schickard (1623) : additions, soustractions, multiplications, divisions



- Deux machines construites, une brûlée, une perdue
- Ne reste qu'une description dans une lettre

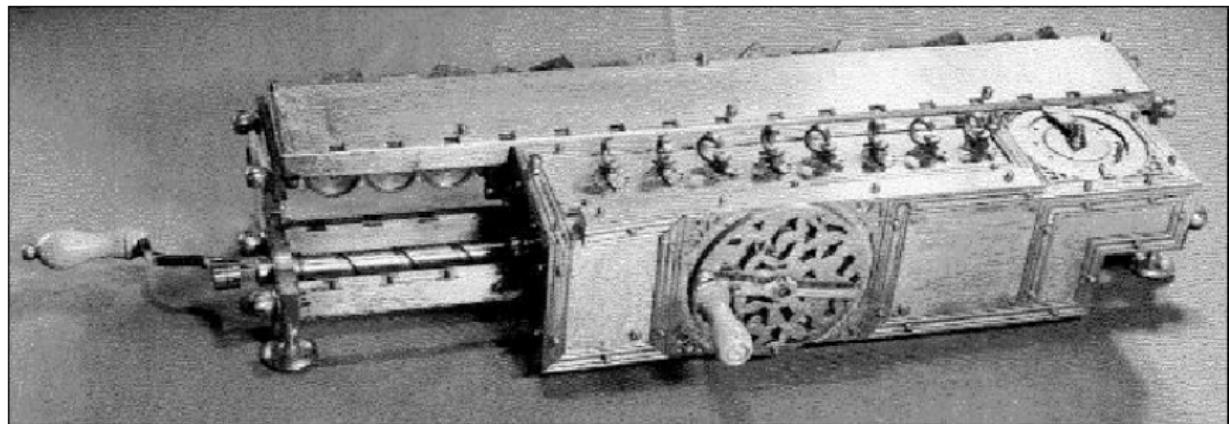
Premiers calculateurs mécaniques

- La pascaline de Blaise Pascal (1642) : additions et soustractions, multiplications, divisions par répétitions



Premiers calculateurs mécaniques

- Stepped Reckoner : amélioration de la pascaline par Gottfried Wilhelm Leibniz (1673) facilitant les multiplications et les divisions (compteur)



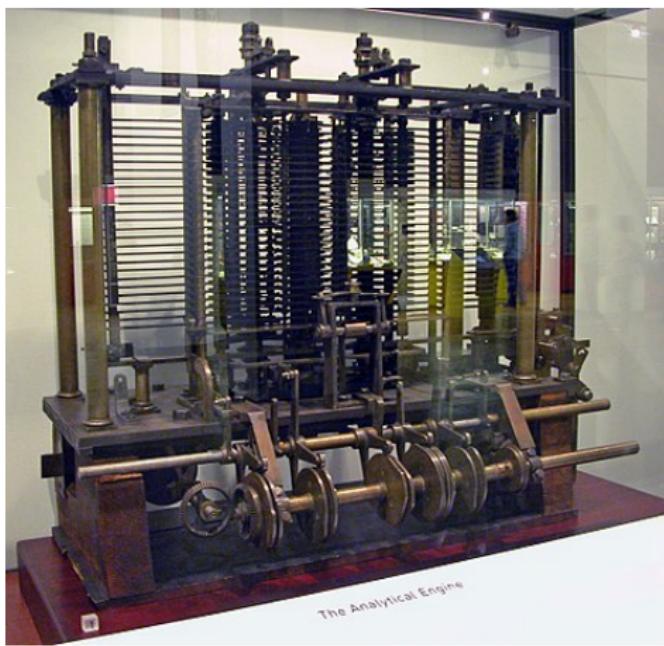
Premières machines programmables

- Machines à tisser automatiques (Jacquart, 1801)
 - ▶ Programmation à l'aide de cartes perforées



Premières machines programmables

- Machine analytique de Charles Babbage (1834), pas construite
 - ▶ Les composantes d'un ordinateur : périphérique d'entrée, unité de commande, unité de calcul, mémoire, périphérique de sortie
 - ▶ Fonctionne à la vapeur avec roues et engrenages mécaniques



Premières machines programmables

- Tabulatrice électromécanique à cartes perforées d'Herman Hollerith (1884) pour le recensement américain



Calculateurs analogiques

- Données et résultats représentés par des valeurs analogiques (courant, tension, ...)
- Considérés comme la solution dans la première moitié du vingtième siècle
- Machines pour la prédition des marées par William Thomson (1872, 1876, 1879)



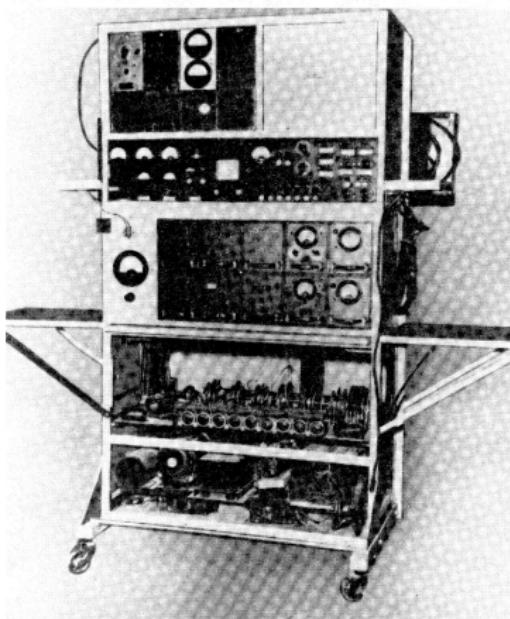
Calculateurs analogiques

- Tir longue distance sur les navires militaires (Arthur Pollen, Frederic Dreyer juste avant la première guerre mondiale)
- Differential Analyzer (Hazen, Bush au MIT en 1927)



Calculateurs analogiques

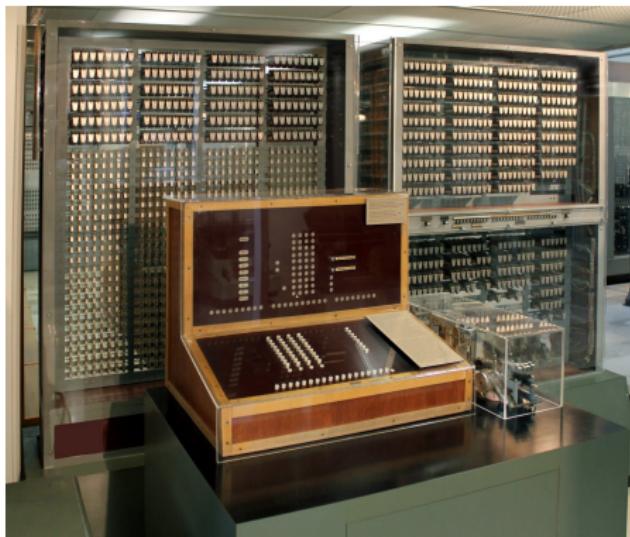
- Calculateur analogique électronique pour la calcul et la simulation de la trajectoire des fusées V2 (Helmut Hölzer, 1942)



F
i
s
t
c
a
c
d
l
S

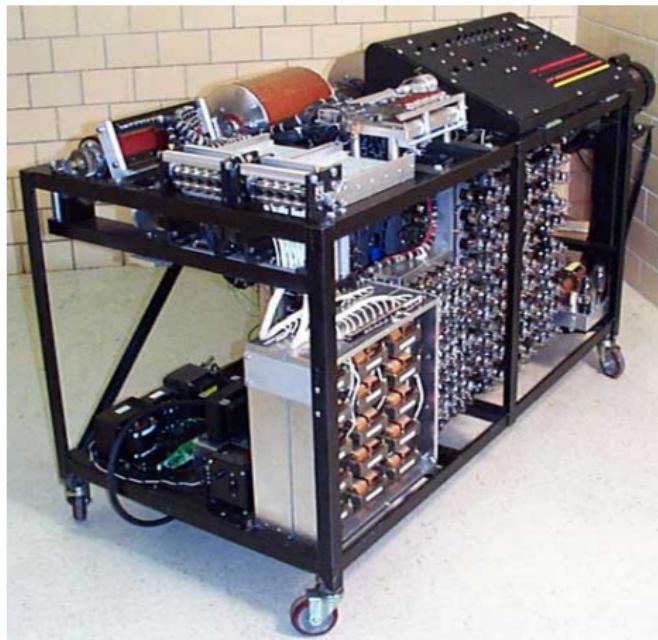
Calculateurs numériques

- Z3 : premier calculateur binaire avec programme enregistré (Konrad Zuse, 1941)
 - ▶ Une console pour l'opérateur
 - ▶ Un lecteur de bandes contenant les instructions à exécuter
 - ▶ 4 additions par seconde et une multiplication en 4 secondes



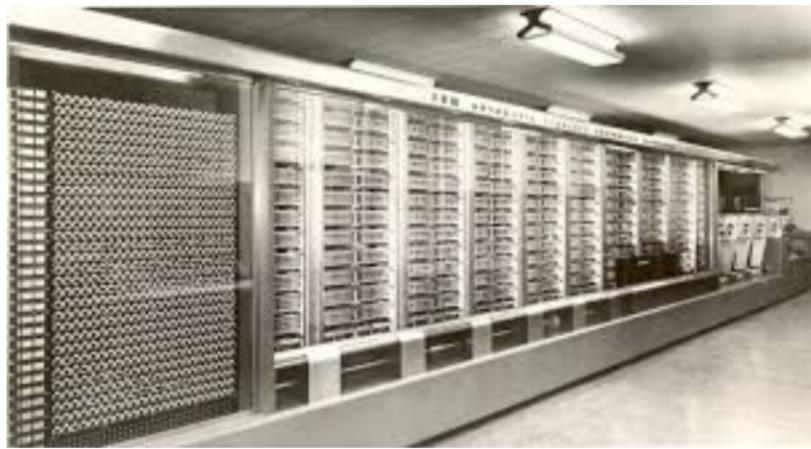
Calculateurs numériques

- ABC (Atanasoff Berry Computer, 1942) : calculateur binaire
 - ▶ 210 lampes, une mémoire de 60 mots de 50 bits, fréquence de 60 Hz



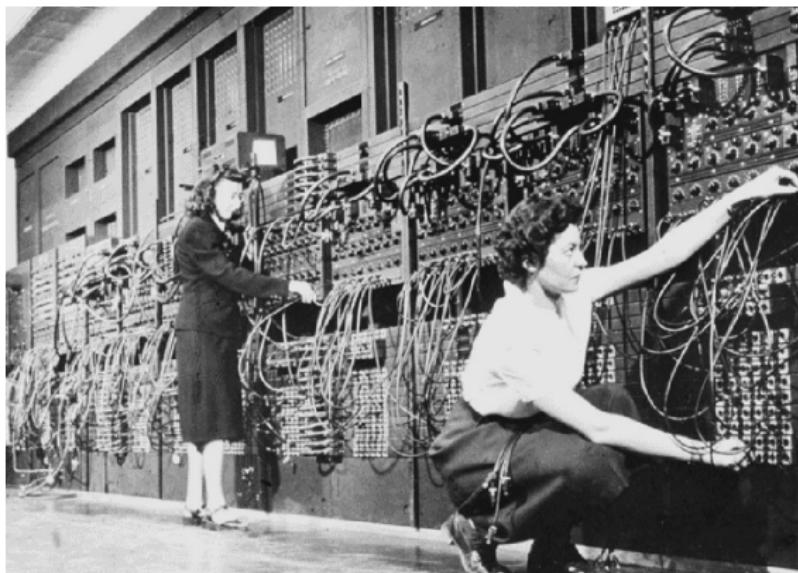
Calculateurs numériques

- IBM ASCC ou Mark 1 (Howard Aiken, 1944)
 - ▶ 5 tonnes, 765000 composants, 800 km de cables
 - ▶ 3 additions/soustractions sur 23 chiffres décimaux par seconde, une multiplication en 6 secondes, une division en 15 secondes, un logarithme en 1 minute
 - ▶ Programmes sur une bande de cartes perforées, données sur une autre bande



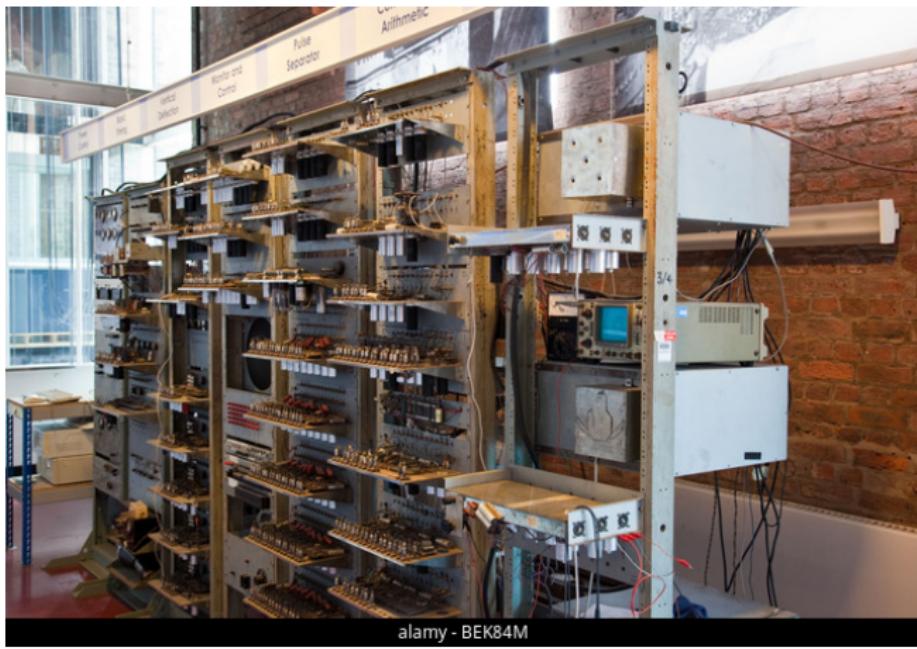
Premiers ordinateurs

- Basés sur le modèle de Von Neuman (1945)
- ENIAC (Electronic Numerical Integrator and Computer, John Eckert et John William Mauchly à partir de 1945)
 - ▶ Von Neuman à partir de 1948
 - ▶ 30 tonnes, 167 m²
 - ▶ 1 multiplication en 3 ms



Premiers ordinateurs

- Université de Manchester : SSEM (1948), Manchester Mark I (1949)
 - ▶ Mémoire composée de tubes cathodiques
 - ▶ Premier programme : nombres premiers de Mersenne (de la forme $2^n - 1$)



alamy - BEK84M

Premiers ordinateurs

- EDSAC, EDVAC : successeurs de l'ENIAC
- Commercialisation de l'UNIVAC (1951)
 - ▶ 46 exemplaires vendus
 - ▶ Mémoire de 1000 mots de 72 bits
 - ▶ 8333 additions ou 555 multiplications par seconde
- Premier ordinateur IBM (1953) : l'IBM 701 pour la défense américaine
 - ▶ Jusqu'à 4096 mots de 36 bits
 - ▶ 16000 additions ou 2200 multiplications par seconde
- Premier ordinateur commercial en série d'IBM : l'IBM 650
 - ▶ Lent, peu fiable, couteux
 - ▶ Premier ordinateur de nombreuses universités américaines
- l'IBM 704 (1955)
 - ▶ Début de l'ère des super ordinateurs dédiés au calcul scientifique (5000 opérations en virgule flottante par seconde)
 - ▶ Machine "très fiable" : une panne par semaine

Ordinateurs électronique à transistors

- Transistor bipolaire à jonction inventé en 1947 au laboratoire BELL
- Permet des ordinateurs plus compacts et plus fiables
- TRADIC (1956) : premier ordinateur à transistors (BELL)



Quelques dates importantes depuis

- 1965 : IBM S/360
 - ▶ Déjà la plupart des techniques utilisées dans les machines actuelles
- 1971 : 1^{er} microprocesseur
 - ▶ Intel 4004
- CRAY 1 : super calculateur
 - ▶ 8 Mo de mémoire vive
 - ▶ 166 millions d'opérations/s
- 1977 : Sortie de l'Apple 2
 - ▶ Domine l'industrie de l'ordinateur personnel de 1977 à 1983
- 1978 : microprocesseur Intel 8086
 - ▶ Jeu d'instruction x86
- 1981 : l'IBM PC
- 1984 : le Macintosh

Problématique générale

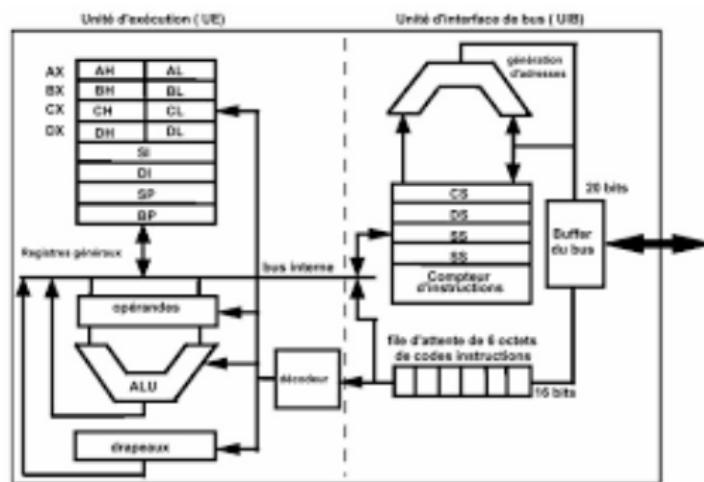
- Objectifs
 - ▶ Augmentation du nombre d'instructions traitées par seconde
 - ▶ Diminution des temps d'exécution
- Limitations
 - ▶ Technique des processeurs
 - ▶ Limite physique de performance
- Tendance
 - ▶ Dupliquer (associer) les capacités matérielles plutôt que d'augmenter la complexité
 - ★ Rapport coût-performance plus favorable

Classification des architectures

- Taxonomie de Flynn
- Quatre catégories
 - ▶ SISD
 - ★ Single Instruction Single Data
 - ▶ SIMD
 - ★ Single Instruction Multiple data
 - ▶ MISD
 - ★ Multiple Instruction Single Data
 - ▶ MIMD
 - ★ Multiple Instruction Multiple Data

Classification des architectures

- SISD : Single Instruction Single Data
 - ▶ Un seul cœur de calcul
 - ▶ A tout instant, exécution d'une instruction sur un seul flot de données
 - ▶ Pas de parallélisme
- Exemple : processeur 8086 d'Intel



Classification des architectures

- SIMD : Single Instruction Multiple Data
 - ▶ Plusieurs cœurs de calcul
 - ▶ A tout instant, exécution d'une instruction sur plusieurs flots de données
 - ▶ Exemples d'architectures
 - ★ GPU
 - ★ Processeurs vectoriels
 - ★ Architectures systoliques
 - ▶ Exemples d'applications
 - ★ Changement de la luminosité d'une image
 - ★ Correction lexicale de texte

Exemple d'un correcteur lexical

- On souhaite corriger des fautes de frappe ou d'orthographe
- On compare un mot T de n caractères avec un ensemble de mots de référence pour déterminer le mot de référence le plus proche de T , i.e. la correction la plus vraisemblable de T
- Méthode de calcul de la distance entre deux mots
 - ▶ Coût d'un caractère x ajouté dans T par rapport au mot de référence R : $C_i(x)$
 - ▶ Coût d'un caractère x omis dans T par rapport au mot de référence R : $C_o(x)$
 - ▶ Coût du remplacement d'un caractère x de R par un caractère y dans T : $C_r(x, y)$
 - ▶ Formule de calcul de la distance $D(i, j)$ entre le mot composé par les i premiers caractères de R ($R[1..i]$) et les j premiers caractères de T ($T[1..j]$)

$$D(i, j) = 0 \quad \text{si } i = 0 \text{ et } j = 0$$

$$D(i, j) = \sup \quad \text{si } i = 0 \text{ xor } j = 0$$

$$\begin{aligned} D(i, j) &= \min(D(i, j - 1) + C_i(T[j]), \\ &\quad D(i - 1, j) + C_o(R[i]), \\ &\quad D(i - 1, j - 1) + C_r(R[i], T[j])) \end{aligned} \quad \text{sinon}$$

Exemple d'un correcteur lexical

- Illustration du calcul avec $C_i(x) = C_o(x) = 1$, $C_r(x, y) = 0$ si $x = y$ et $C_r(x, y) = 1$ sinon

	C	O	R	E	L	T	L	I	O	N
C	0	1	2	3	4	5	6	7	8	9
O	1	0	1	2	3	4	5	6	7	8
R	2	1	0	1	2	3	4	5	6	7
R	3	2	1	1	2	3	4	5	6	7
E	4	3	2	1	2	3	4	5	6	7
C	5	4	3	2	2	3	4	5	6	7
T	6	5	4	3	3	2 → 3	4	5	6	
I	7	6	5	4	4	3	3	3	4	5
O	8	7	6	5	5	4	4	4	3	4
N	9	8	7	6	6	5	5	5	4	3

Exemple d'un correcteur lexical

- On doit enchaîner la comparaison du mot à corriger avec un grand nombre de mots référence
- Mise en œuvre avec un processeur par case du tableau
- Tous les processeurs exécutent le même programme, sur des données différentes
- Gain par rapport à une solution avec un seul processeur
 - ▶ le calcul de la distance entre un mot de n caractères et un mot de m caractères nécessite $n \times m$ calculs de distance “partielles”
 - ▶ Avec un seul processeur, un résultat de comparaison tous les $n \times m$ calculs partiels
 - ▶ Avec $n \times m$ processeurs, un premier résultat après $n \times m$ calculs partiels, puis un résultat à chaque calcul partiel
 - ▶ Attention à la transmission des résultats entre les différents processeurs.

Classification des architectures

- MISD : Multiple Instructions Single Data
 - ▶ Plusieurs cœurs 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

Classification des architectures

- MIMD : Multiple Instructions Multiple Data
 - ▶ Plusieurs cœurs de calcul
 - ▶ A tout instant, exécution d'instructions différentes sur des flots de données différents
 - ▶ Deux sous-groupes
 - ★ SPMD (Single Program Multiple Data) : un même programme exécuté sur différentes unités, de manière indépendante (asynchrone) $\Rightarrow \neq$ SIMD
 - ★ MPMD (Multiple Program Multiple Data) : différents programmes exécutés sur différentes unités
 - ▶ Un exemple "ancien" de machines : l'iPSC/2 d'Intel, fin des années 80
 - ★ Machine à topologie hypercube
 - ★ 64 processeurs

La mémorisation des informations

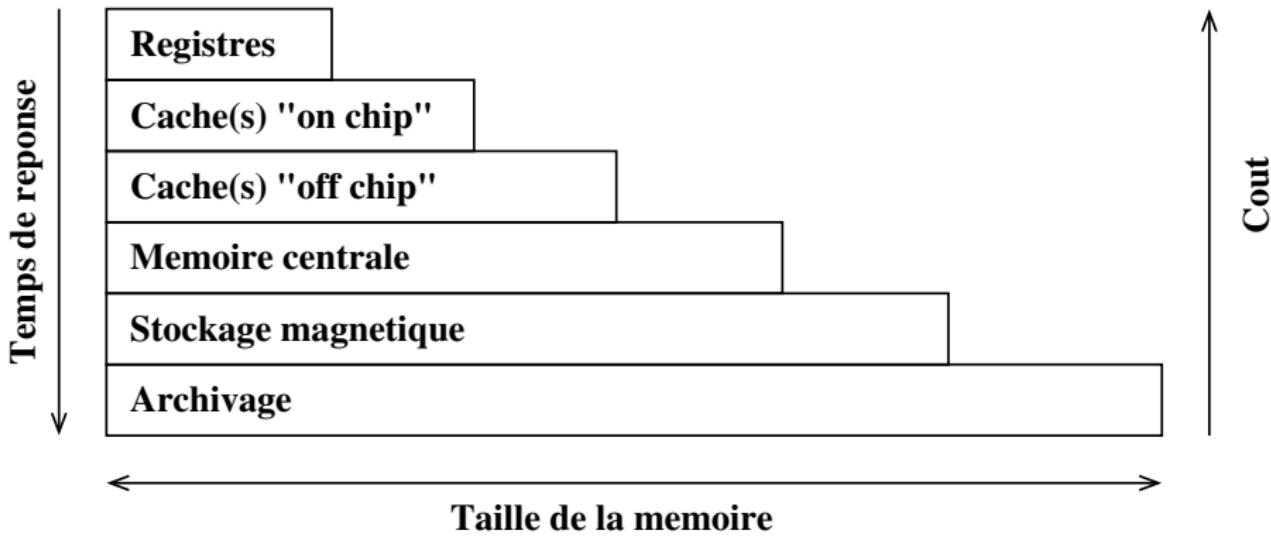
- Mémoire : composant qui mémorise des informations, par exemple les données d'un programme
- Capacité d'une mémoire : quantité d'informations, par exemple en bits, que celle-ci peut contenir
- Critères pour mesurer la rapidité d'une mémoire
 - ▶ Temps de latence : temps nécessaire pour effectuer une lecture ou une écriture
 - ▶ Débit mémoire : quantité d'informations qui peuvent être lues ou écrites en une seconde dans la mémoire
- Plus le temps de latence est faible, plus la mémoire est rapide, mais ça dépend de la quantité d'information par lecture/écriture
- Plus le débit est élevé, plus la mémoire est rapide

Accès mémoire

- Instruction arithmétique (e.g. addition entière sur 64 bits)
 - ▶ Typiquement un cycle d'horloge
- Accès à une mémoire DRAM
 - ▶ Typiquement 100-200 cycles d'horloge
- Sources de la latence mémoire
 - ▶ Connexions longues de plusieurs centimètres ⇒ la vitesse de la lumière a une impact
 - ▶ Une grande mémoire a des temps d'accès plus longs
 - ▶ Plusieurs bancs DRAM ⇒ sélection du banc correspondant à la demande ⇒ latence supplémentaire
- Réduire la latence
 - ▶ Prendre une DRAM plus petite en capacité
 - ★ Besoin toujours plus grand en mémoire
 - ★ Il faudrait trop réduire
 - ▶ Mettre la mémoire sur la puce
 - ★ Mémoire trop grande avec des besoins qui augmentent
- Solution : une hiérarchie de mémoires

Hiérarchie mémoire

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

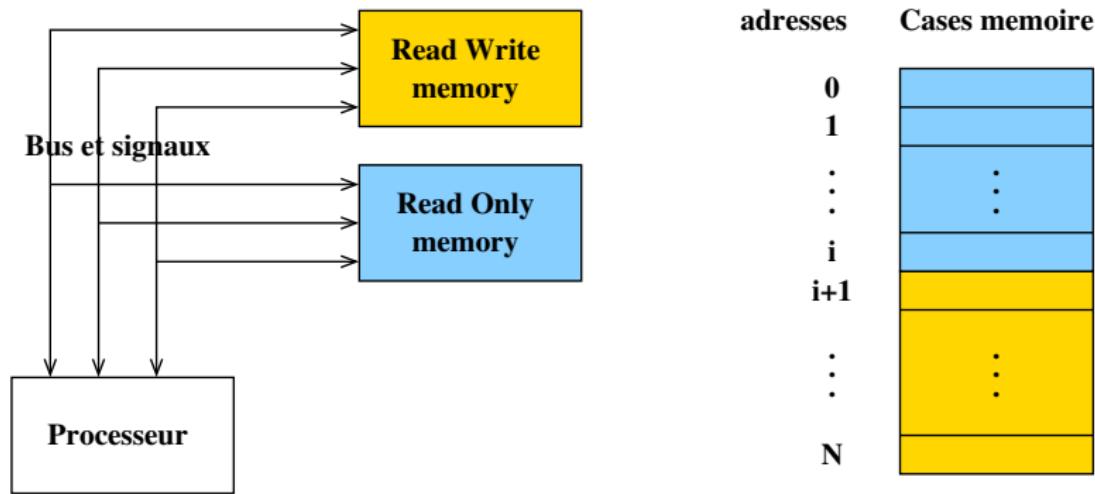


Types de mémoires

- Perennité du contenu de la mémoire
 - ▶ Mémoire non volatile
 - ★ Contenu conservé, y compris en l'absence d'alimentation électrique
 - ▶ Mémoire volatile statique
 - ★ Contenu conservé, sauf en l'absence d'alimentation électrique
 - ▶ Mémoire volatile dynamique
 - ★ Contenu perdu si pas d'alimentation électrique
 - ★ Contenu perdu si pas réécrit régulièrement
- Accès au contenu de la mémoire
 - ▶ Mémoire ROM (Read Only Memory)
 - ★ Mémoire non volatile avec accès en lecture seule
 - ★ Pas d'accès en écriture, mais possibilité de réécrire intégralement le contenu
 - ★ EPROM : effacement avec des rayons UV et reprogrammation possible plusieurs fois
 - ★ EEPROM : effacement par des moyens électriques
 - ▶ Mémoire RWM (Read-Write Memory) en particulier la mémoire RAM (Random Access Memory)
 - ★ Mémoire volatile accessible en lecture et en écriture

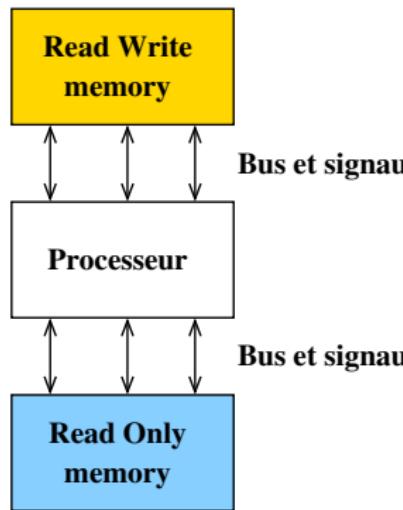
Exemple de type Von Neumann

- La RAM et la ROM sont reliés au processeur via les mêmes bus
 - ▶ Un bus d'adresse : numéro de la case à accéder
 - ▶ Un bus de données : donnée à écrire ou lire
 - ▶ Signal précisant l'opération (lecture ou écriture)
- Adresses distinctes



Exemple de type Harvard

- La RAM et la ROM sont reliés au processeur via des bus distincts
- Une même adresse peut correspondre à la RAM ou à la ROM



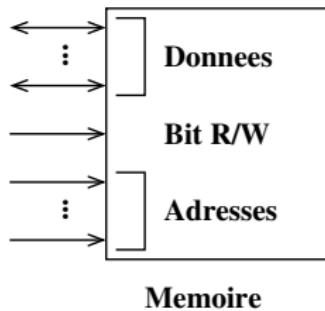
adresses	Cases memoire
0	
1	
:	⋮
N1	
0	
:	⋮
N2	

Adressage et accès

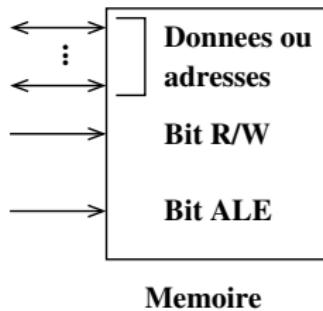
- Mémoires adressables : une adresse pour chaque case mémoire, qui permet de la sélectionner
- Plusieurs types de mémoires adressables
 - ▶ Mémoires à accès aléatoire (typiquement la RAM)
 - ★ Accès par l'adresse de la donnée à lire ou à modifier
 - ▶ Mémoires adressables par contenu (associatives)
 - ★ Fonctionnement inverse de la mémoire à accès aléatoire
 - ★ Envoi de la donnée et le processeur détermine son adresse
- Mémoires caches
 - ▶ Plusieurs données partagent la même case du cache \Rightarrow une information supplémentaire (le tag) pour identifier la donnée présente dans la case
- Mémoires à accès séquentiel : accès uniquement dans un ordre prédéfini
 - ▶ Par exemple une file (FIFO : First In First Out) ou une pile (LIFO : Last In First Out)

Connexion d'une mémoire adressable

Architecture avec 2 bus



Architecture avec 1 bus



R/W = 1 : lecture

R/W = 0 : écriture

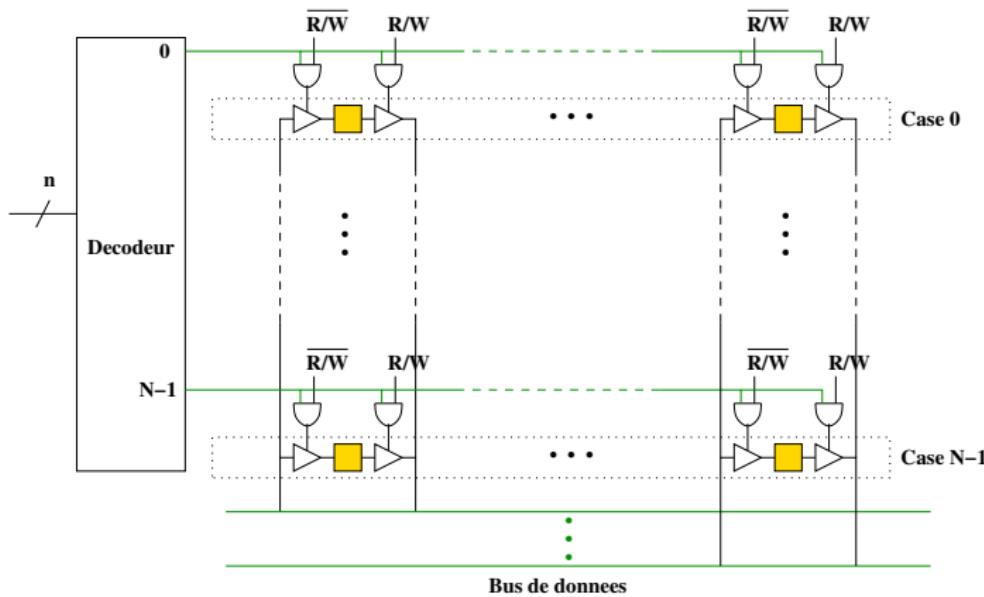
ALE = 1 : adresse sur le bus

ALE = 0 : donnée sur le bus

- Les deux architectures permettent de faire une lecture en une étape
- Avec un seul bus, il faut deux étapes pour une écriture
 - ▶ Etape 1 : adresse de la case mémoire à écrire sur le bus
 - ▶ Etape 2 : valeur à écrire sur le bus

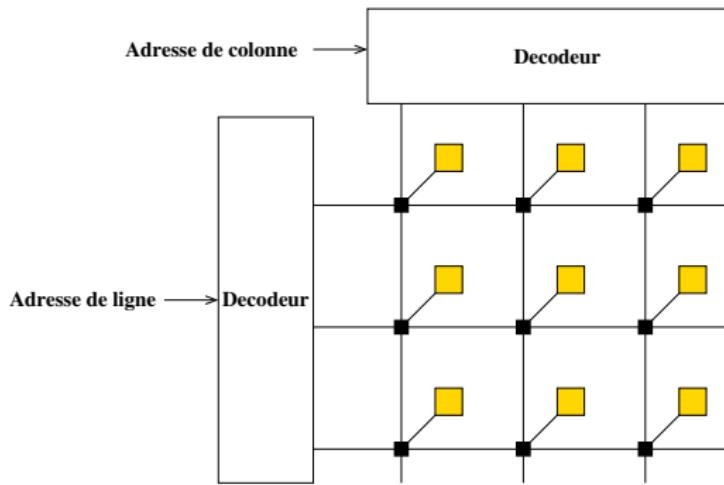
Contrôleur mémoire interne

- Pour la sélection de la case mémoire à lire ou écrire
- Différentes solutions
 - ▶ Mémoire à adressage linéaire
 - ★ Une case mémoire = une ligne
 - ★ Un décodeur mémoire pour relier la bonne ligne au bus



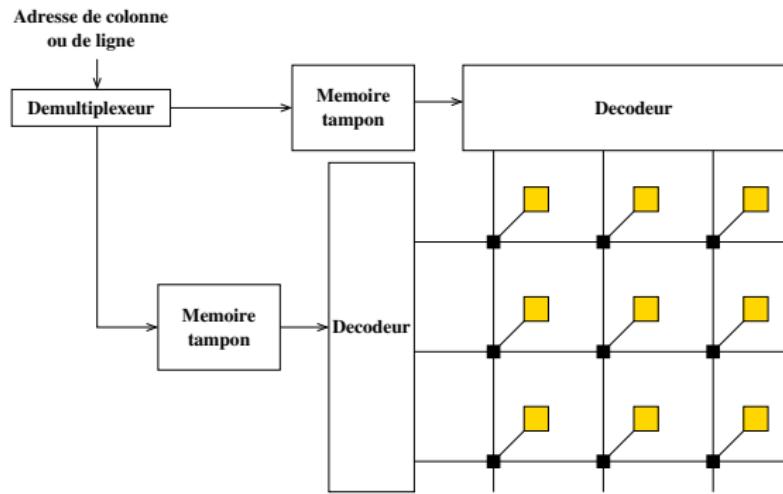
Contrôleur mémoire interne

- Pour la sélection de la case mémoire à lire ou écrire
- Différentes solutions
 - ▶ Mémoire à adressage par coïncidence
 - ★ Plusieurs cases mémoire par ligne, en particulier pour limiter la complexité du décodeur
 - ★ Une case mémoire au croisement d'une ligne et d'une colonne \Rightarrow on peut utiliser 2 décodeurs
 - ★ Solution avec l'adresse en une étape



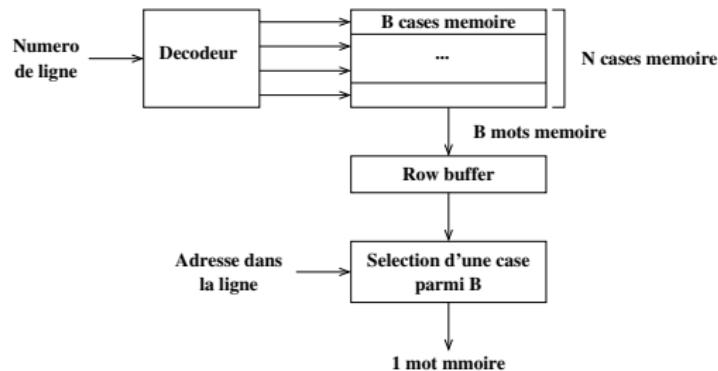
Contrôleur mémoire interne

- Pour la sélection de la case mémoire à lire ou écrire
- Différentes solutions
 - ▶ Mémoire à adressage par coïncidence
 - ★ Plusieurs cases mémoire par ligne, en particulier pour limiter la complexité du décodeur
 - ★ Une case mémoire au croisement d'une ligne et d'une colonne ⇒ on peut utiliser 2 décodeurs
 - ★ Solution avec l'adresse en deux étapes



Contrôleur mémoire interne

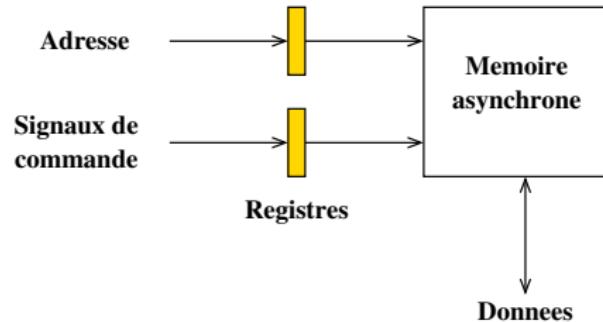
- Pour la sélection de la case mémoire à lire ou écrire
- Différentes solutions
 - ▶ Mémoire à row buffer
 - ★ Plusieurs cases mémoire par ligne,
 - ★ Chargement d'une ligne en une étape, dans le row buffer, puis sélection de la case adressée



Quelques améliorations des mémoires

- Mémoires synchrones

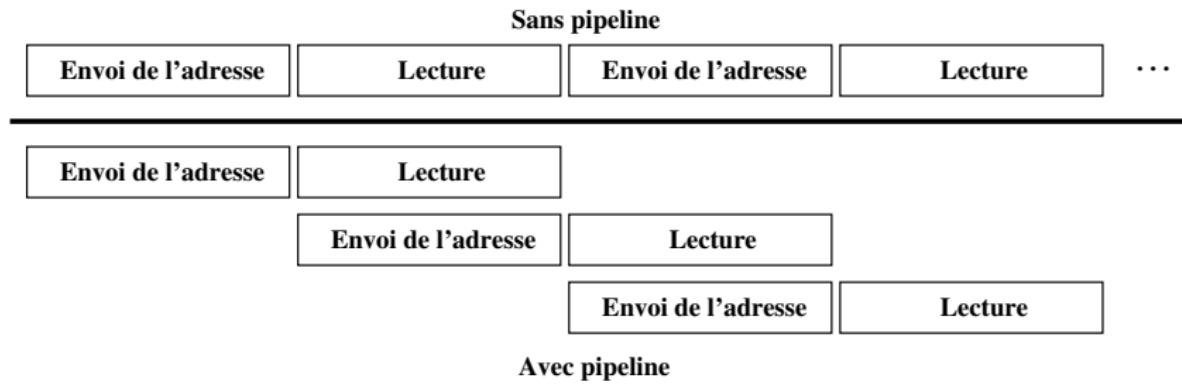
- ▶ Un accès mémoire prend un nombre de cycles déterminé ⇒ le processeur sait quand l'opération sera terminée
- ▶ Ajout de registres, par exemple pour maintenir l'adresse en entrée de la mémoire pendant toute la durée de l'accès



Quelques améliorations des mémoires

- Mémoires synchrones pipelinées

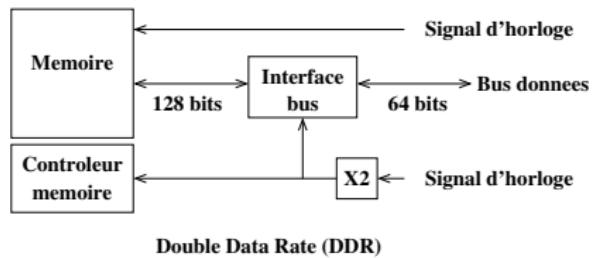
- ▶ Accès mémoire en plusieurs étapes, par exemple envoi de l'adresse, puis lecture de la donnée ⇒ effectuer la première étape d'un accès au même cycle que l'étape suivante de l'accès précédent



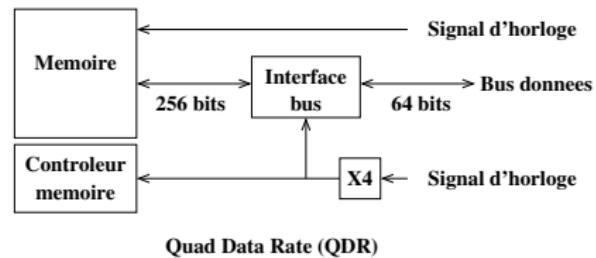
- La durée d'un accès n'est pas réduite mais le nombre d'accès possible par seconde est augmenté
- Contrôleur mémoire interne un peu plus complexe

Quelques améliorations des mémoires

- Mémoires Dual ou Quad Data Rate
 - ▶ La mémoire fournit plus de données en une étape (par exemple deux fois plus ou quatre fois plus)
 - ▶ Ces données sont transmises sur le bus en plusieurs fois ⇒ le bus doit avoir une fréquence plus élevée (deux fois, quatre fois) que la mémoire
 - ▶ Bon compromis pour une mémoire plus rapide à un coût raisonnable



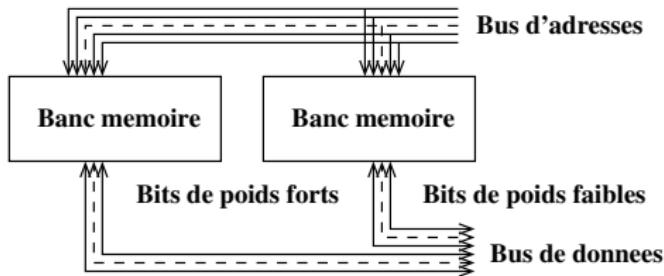
Double Data Rate (DDR)



Quad Data Rate (QDR)

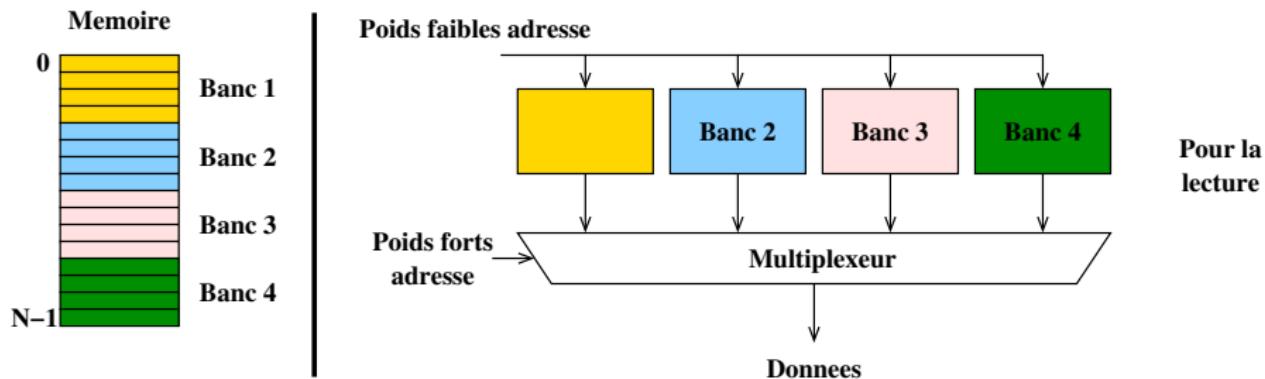
Bancs mémoire

- Une mémoire est un ensemble de bancs mémoire
- Possibilité de rafraîchir en parallèle les différents bancs
- Différentes solutions pour assembler des bancs mémoire
 - ▶ Arrangement horizontal
 - ★ Chaque banc contient une partie de la case mémoire d'une adresse donnée ⇒ accès à tous les bancs en parallèle à chaque accès
 - ★ Exemple avec deux bancs



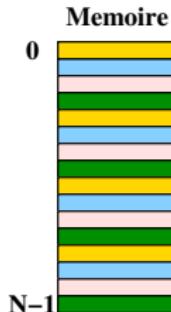
Bancs mémoire

- Une mémoire est un ensemble de bancs mémoire
- Possibilité de rafraîchir en parallèle les différents bancs
- Différentes solutions pour assembler des bancs mémoire
 - ▶ Arrangement vertical
 - ★ Chaque banc contient des cases mémoire différentes
 - ★ Une solution en utilisant les bits de poids forts pour sélectionner le banc
 - ★ Ne permet pas d'enchaîner des accès sur des adresses consécutives



Bancs mémoire

- Une mémoire est un ensemble de bancs mémoire
- Possibilité de rafraîchir en parallèle les différents bancs
- Différentes solutions pour assembler des bancs mémoire
 - ▶ Arrangement vertical
 - ★ Chaque banc contient des cases mémoire différentes
 - ★ Entrelacement des adresses pour permettre d'enchaîner des accès sur des adresses consécutives



Entrelacement simple



Entrelacement avec decalage

Principe des caches

- Principe de localité temporelle
 - ▶ Les programmes réaccèdent souvent des instructions et des données qu'ils ont accédées récemment
 - ▶ Un bon exemple : les itérations
- Mémoire cache : petite mémoire située sur la puce processeur
 - ▶ Par exemple 64 Ko
- Dans le cache : données et instructions accédées récemment
 - ▶ Deux cas lors de l'accès à une donnée ou une instruction
 - ★ Donnée/instruction absente du cache (défaut de cache, cache miss) ⇒ lecture en mémoire centrale et stockage automatique dans le cache
 - ★ Donnée/instruction présente dans le cache (cache hit) ⇒ lecture dans le cache
 - ▶ Mécanisme transparent pour le programmeur et le compilateur

Principe des caches

- Possibilité d'avoir des caches séparés pour les instructions et les données
 - ▶ Limite les conflits d'accès
- Possibilité d'avoir plusieurs niveaux de cache (hiérarchie mémoire)
 - ▶ Niveau 1 (L1) : caches petits et rapides d'accès
 - ★ Par exemple 64 Ko, latence 2 cycle
 - ▶ Niveau 2 (L2) : cache plus gros et plus lent
 - ★ Par exemple 1 Mo, latence 10 cycles
 - ▶ Mémoire : gros volume, très lent
 - ★ Par exemple 2 Go, latence 100-200 cycles

Quelques problèmes à résoudre

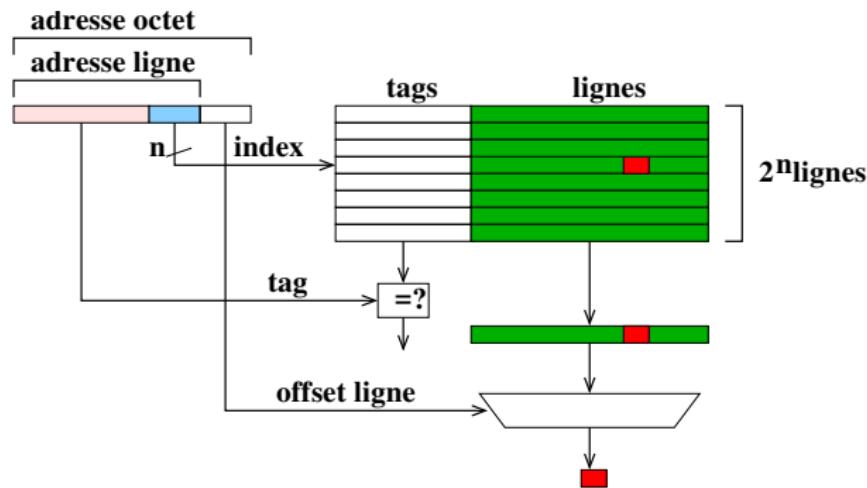
- Comment structurer le cache ?
- Comment gérer les écritures ?
- Comment gérer les remplacements en cas de cache miss ?

Notion de ligne de cache

- Principe de localité spatiale
- 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

Cache à correspondance directe (direct-mapped)

- Structure de cache la plus simple
- Une position pré-déterminée dans le cache pour chaque ligne en mémoire
- Accès à une donnée/instruction dans le cache



Stratégie pour les écritures

- Cache write-through (à écriture transmise)

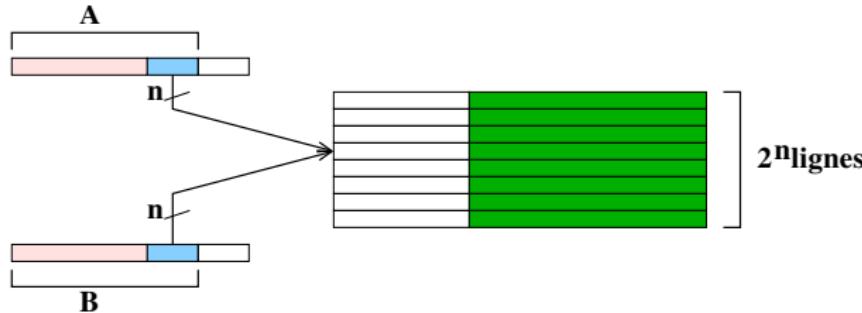
- ▶ Envoi systématique de l'ordre d'écriture au niveau suivant de la hiérarchie mémoire
- ▶ Hit en lecture ⇒ lecture de la donnée dans le cache
- ▶ Hit en écriture ⇒ mise à jour de la donnée dans le cache et dans les niveaux supérieurs
- ▶ Miss en lecture ⇒ écriture de la ligne manquante et de son tag dans le cache, depuis le niveau mémoire supérieur
 - ★ La ligne remplacée est simplement écrasée
- ▶ Miss en écriture ⇒ 2 stratégies possibles
 - ★ Allocation sur écriture : ligne manquante copiée dans le cache, puis même opération que pour un hit en écriture
 - ★ Pas d'allocation sur écriture : transmission de l'écriture au niveau suivant ⇒ cache inchangé

Stratégie pour les écritures

- Cache write-back (à écriture différée)
 - ▶ Un bit *dirty* associé à chaque entrée du cache
 - ★ *dirty* = 1 ⇒ la ligne présente dans le cache a été modifiée
 - ▶ Hit en écriture ⇒ mise à jour de la donnée dans le cache et *dirty* = 1
 - ▶ Hit en lecture ⇒ lecture de la donnée dans le cache
 - ▶ Miss en lecture ⇒ chargement de la ligne manquante dans le cache, à la place de la ligne présente
 - ★ *dirty* = 0 : la ligne n'a pas été modifiée ⇒ on peut l'écraser
 - ★ *dirty* = 1 : la ligne a été modifiée ⇒ on la sauvegarde dans le niveau suivant
 - ▶ Miss en écriture ⇒ copie de la ligne manquante dans le cache (après recopie ou non de la ligne écrasée, en fonction du bit *dirty*), puis même procédure que pour un hit en écriture

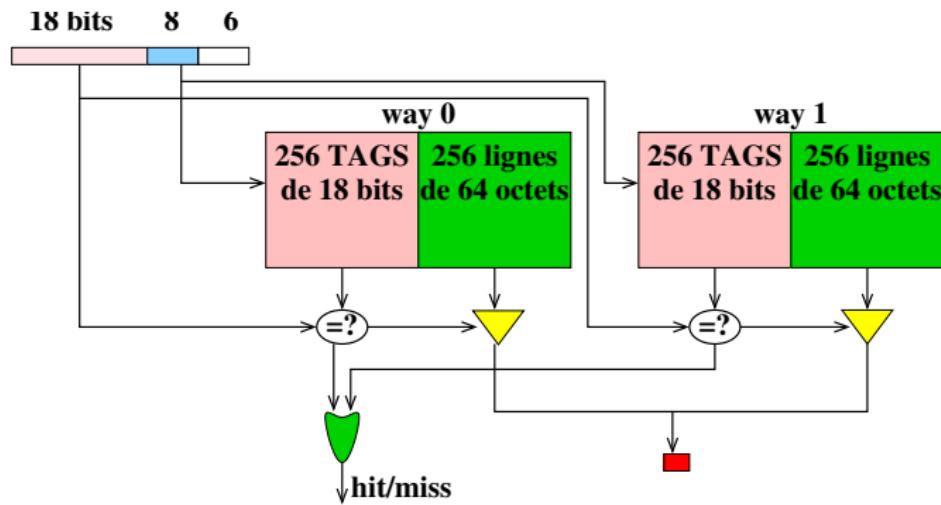
Différents types de miss

- Miss de démarrage
 - ▶ Premier accès à une donnée ou à une instruction
 - ▶ Réduction du nombre de miss de démarrage en prenant des lignes de cache plus longues
- Miss de capacité
 - ▶ Cache trop petit pour contenir tout le programme et/ou toutes ses données
 - ▶ Réduction du nombre de miss de capacité en augmentant la taille du cache
- Miss de conflit
 - ▶ Cache assez grand, mais deux lignes mémoire correspondent à la même entrée du cache



Elimination/limitation des conflits

- Au niveau du programme
 - ▶ Placer les données/instructions à des adresses mémoire proches
- Au niveau du matériel : cache associatif par ensembles (set-associatif)
 - ▶ Plusieurs entrées par ligne de cache
 - ★ N-way set-associatif \Rightarrow N entrées par ligne de cache
 - ★ Full-associatif \Rightarrow une seule ligne avec toutes les entrées



Politique de remplacement

- N entrées possibles pour une ligne mémoire dans un cache N-way associatif \Rightarrow politique de remplacement permettant d'en choisir une
- Exemple de politiques de remplacement
 - ▶ Random : éviction d'une ligne au hasard parmi celles qui sont dans les bonnes entrées
 - ▶ LRU (Least Recently Used) : éviction de la ligne qui n'a pas été utilisée depuis le temps le plus long parmi celles qui sont dans les bonnes entrées
- LRU est une politique assez efficace
 - ▶ Ligne pas utilisée depuis longtemps \Rightarrow probablement pas réutilisée avant longtemps
- LRU est simple à implémenter pour un cache 2-way
 - ▶ 1 bit MRU (Most Recently Used) par ensemble de lignes
 - ▶ Une ligne de l'ensemble accédée \Rightarrow $MRU = 0$ si la ligne est dans le way 0, $MRU = 1$ sinon
 - ▶ Eviction de la ligne dans le way ne correspondant pas à la valeur de MRU
- Associativité plus élevée \Rightarrow mise en œuvre plus complexe

Accélérer l'exécution des programmes

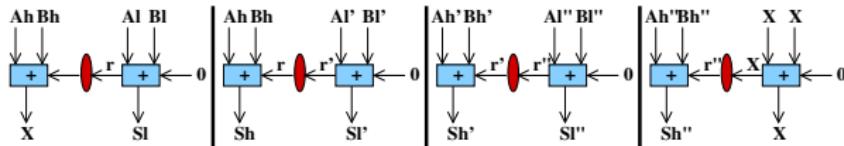
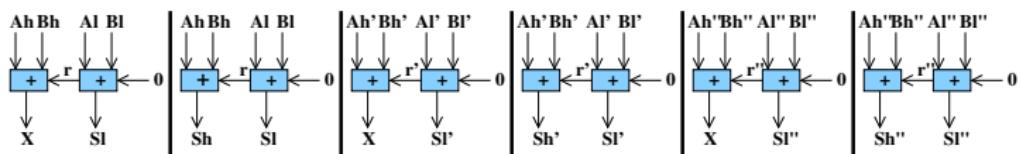
- Etapes de l'exécution d'une instruction RISC
 - ▶ Lecture de l'instruction stockée en mémoire à l'adresse PC
 - ▶ Décodage de l'instruction
 - ★ En particulier extraction du type d'opération et des registres accédés
 - ▶ Lecture des registres
 - ▶ Exécution de l'opération
 - ▶ Ecriture du résultat dans les registres
 - ▶ Passage à l'instruction suivante
- Durée d'exécution d'une instruction : somme des durées de chaque étape
- Certaines étapes n'ont pas de ressources en commun avec d'autres étapes ⇒ pourraient fonctionner en parallèle
- Ne pas attendre qu'une instruction soit terminée pour commencer à exécuter l'instruction suivante ⇒ technique du pipeline

Principe du pipeline

- Exemple

- ▶ Additionneur 64 bits constitué de deux additionneurs 32 bits
- ▶ Calcul de $A + B$, puis $A' + B'$, puis $A'' + B''$
- ▶ Durée d'une addition 64 bits : $2T$
- ▶ Durée d'une addition 32 bits : T

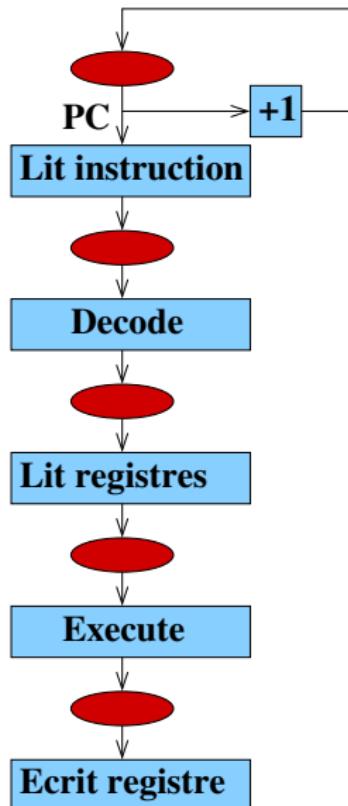
SANS PIPELINE



AVEC PIPELINE

Pipeline d'instructions

- A chaque cycle, une nouvelle instruction peut entrer dans le pipeline



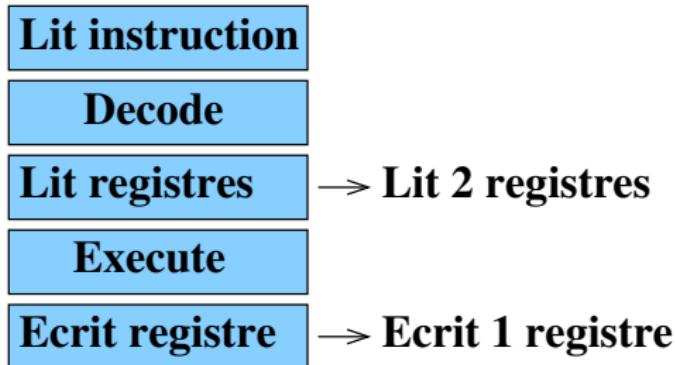
Débit versus latence

- Latence d'une instruction
 - ▶ Temps entre le début et la fin de l'instruction
- Débit
 - ▶ Nombre d'instructions exécutées pendant un temps fixe
- Impact du pipeline
 - ▶ Augmentation du débit
 - ★ En théorie une instruction par cycle
 - ▶ Légère augmentation de la latence
 - ★ Il faut traverser des points de mémorisation
- En pratique, l'augmentation du débit est souvent moindre
 - ▶ Aléas structurels : conflits de resources
 - ▶ Aléas de contrôle : instructions de branchements
 - ▶ Aléas de dépendance : dépendances entre instructions
 - ▶ Accès mémoire : beaucoup plus lent qu'un calcul

En théorie une instruction par cycle

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

Aléas structurels : registres



- Deux lectures et une écriture en même temps dans le banc de registres
- Une solution : banc de registres avec deux ports de lectures et un port d'écriture

Aléa de contrôle : branchements

	cycle N	cycle N+1	cycle N+2	...
Lit instruction	I4	I29	I30	...
Decode	I3	(yellow oval)	I29	...
Lit registres	I2	(yellow oval)	(yellow oval)	...
Execute	I1:jmp I29	(yellow oval)	(yellow oval)	...
Ecrit registre		I1	(yellow oval)	...

- Il faut attendre de savoir
 - ▶ Si le branchement est pris
 - ▶ Si oui, vers quelle adresse
- Si on ne fait rien de particulier, trois cycles perdus à chaque branchement

Aléa de contrôle : Une “vieille” solution

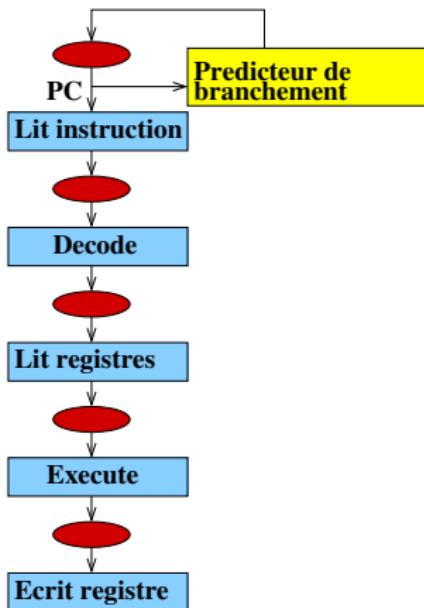
- Le branchement différé

- ▶ Mis en œuvre sur MIPS, Sparc
- ▶ Branchement après exécution d'une ou plusieurs autres instructions ⇒ revient à écrire après l'instruction de branchement des instructions qui auraient du (pu) s'exécuter avant
- ▶ Exemple de programme

```
I1:jmp I5  
I2  
I3  
I4  
I5
```

- ▶ Exécution de I1, puis I2, puis I5
- ▶ Permet d'éviter des bulles

Aléa de contrôle : prédicteur de branchement



- Branchements spéculatifs en fonction du comportement passé des branchements
- Mauvaise prédition \Rightarrow aléa de contrôle

Aléa de dépendance



I2 : r2:= r1 sub 56

I1 : r1:= r1 add 3

- I2 utilise la valeur produite par I1
- I2 doit attendre que I1 ait écrit r1

Trois types de dépendances

- Dépendance Read After Write (RAW)

I1: $r1 := r1 \text{ add } 1$

I2: $r2 := \text{load } r1 + \text{dep}$

- ▶ Résultat de I1 utilisé par I2
- ▶ Aussi appelée dépendance vraie

- Dépendance Write After Read (WAR)

I1: $r2 := \text{load } r1 + \text{dep}$

I2: $r1 := r1 \text{ add } 1$

- ▶ I2 écrit dans un registre lu par I1
- ▶ Aussi appelée anti-dépendance
- ▶ Pas de problème si on exécute les instructions dans l'ordre

- Dépendance Write After Write (WAW)

I1: $r2 := \text{load } r1 + \text{dep}$

I2: $r2 := \text{load } r3 + \text{dep}$

- ▶ I2 écrit dans le même registre que I1
- ▶ Aussi appelée dépendance de sortie
- ▶ Pas de problème si on exécute les instructions dans l'ordre

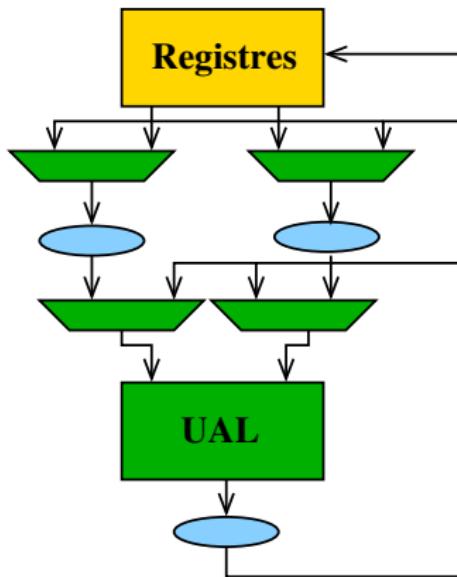
Aléa Read After Write

	cycle N	N+1	N+2	N+3
Lit instruction				
Decode				
Lit registres	I2:r1:=r1+1	I2	I2	
Execute	I1:r1:=r2+r3			I2
Ecrit registre		I1		

- Quelques solutions

- ▶ Changement de l'ordre des instructions
- ▶ Mécanisme de bypass
 - ★ Utiliser le résultat d'une instruction sans attendre qu'il soit écrit dans le registre

Mécanisme de bypass



- Deux instructions avec une dépendance Read After Write peuvent s'exécuter dans deux cycles consécutifs
- Multiplexeurs commandés par des comparaisons sur les numéros de registres

Illustration du mécanisme de bypass

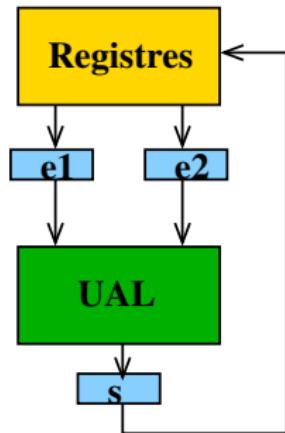
- Calcul dans S de la somme des N premiers entiers

```
add    %r0,%r0,%r1
add    %r0,1,%r2
set    N,%r3
ld     [%r3],%r3
add  %r0,%r0,%r4
tq:   add  %r1,%r4,%r1
      add  %r4,%r2,%r4
      subcc %r4,%r3,%r0
      ble   tq
      set   S,%r5
      st    %r1,[%r5]
fin:  ba   fin
N:    .word 10
S:    .word 0
```

- Exécution des trois instructions en gras

Illustration du mécanisme de bypass

- Exécution sans mécanisme de bypass



A executer : $\%rd \leftarrow \%rs1 \text{ op } \%rs2$

Etage 1 du pipeline

$\%ri1 \leftarrow \text{mem}[\%pc]$

Etage 2 du pipeline

$e1 \leftarrow \%rs1(\%ri1)$

$e2 \leftarrow \%rs2(\%ri1)$

$\%ri2 \leftarrow \%ri1$

Etage 3 du pipeline

$s \leftarrow e1 \text{ op } (\%ri2) e2$

$\%ri3 \leftarrow \%ri2$

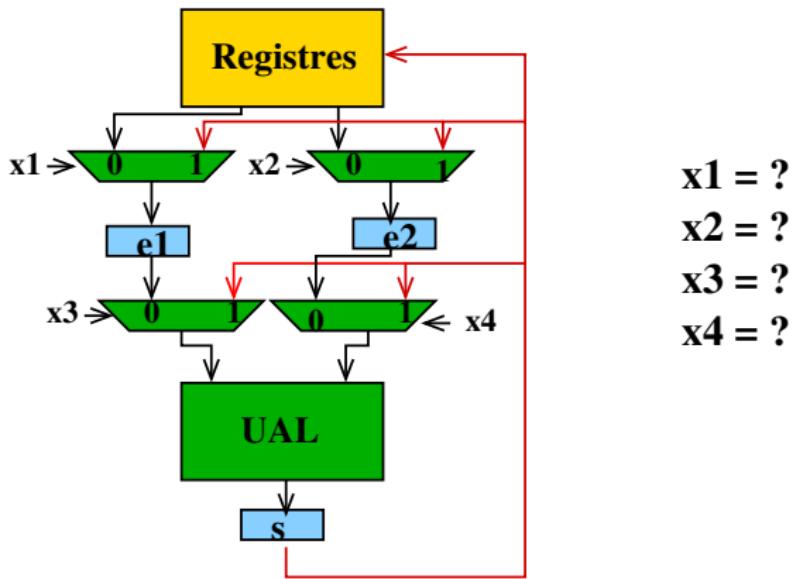
Etage 4 du pipeline

$\%rd(\%ri3) \leftarrow s$

- Déroulement de l'exécution des 3 instructions

Illustration du mécanisme de bypass

- Exécution avec mécanisme de bypass

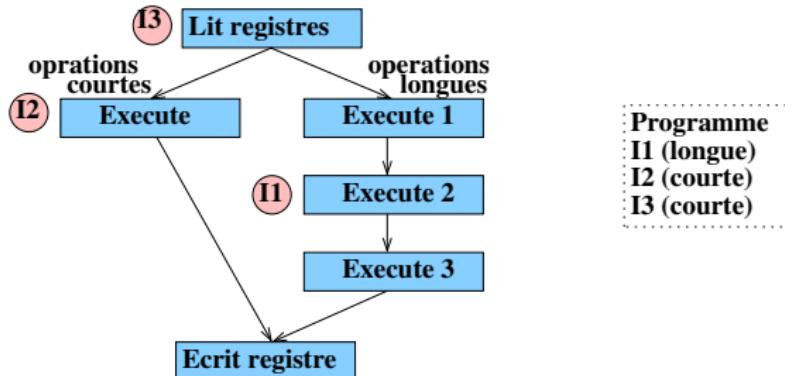


- Déroulement de l'exécution des 3 instructions

Instructions de durées différentes

- Des instructions plus longues à exécuter que d'autres
 - ▶ Load/store : temps d'accès à la mémoire
 - ▶ Division et multiplication entières : plus complexes que l'addition
 - ▶ Opérations en virgule flottante
 - ▶ ...
- L'opérateur le plus lent dicte la fréquence d'horloge
 - ▶ Durée d'exécution d'une instruction : durée de l'instruction la plus lente
- Possibilité de pipeliner les opérateurs

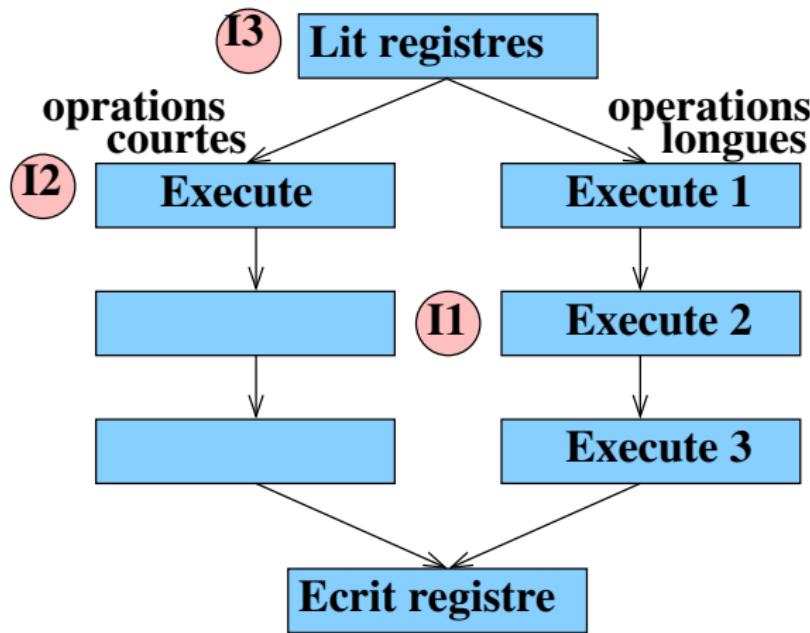
Pipeliner les opérateurs : problèmes



	Cycle N	Cycle N+1	Cycle N+2	Cycle N+3	Cycle N+4
Lit registres	I1	I2	I3		
Execute			I2	I3	
Execute 1		I1			
Execute 2			I1		
Execute 3				I1	
Ecrit registre				I2	I1,I3

- I1 et I3 écrivent en même temps
- I2 écrit avant I1

Pipeliner les opérateurs : une solution

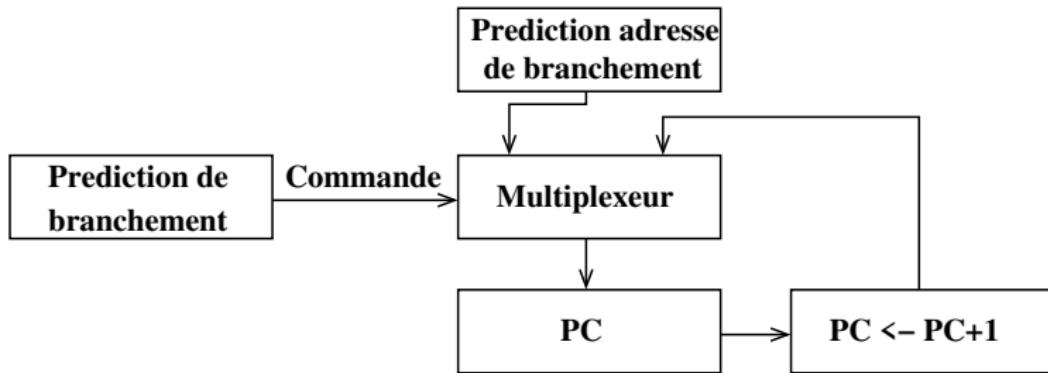


Programme
I1 (longue)
I2 (courte)
I3 (courte)

- Mécanisme de bypass plus complexe
- Problème si différence très grande

La prédition de branchement

- La plupart des programmes ont un comportement répétitif
- On stocke des informations sur les instructions de contrôle et leur comportement passé ⇒ prédicteur de branchement
- On utilise ces informations pour effectuer les sauts spéculativement
- En cas de mauvaise prédition, on vide le pipeline et on corrige le PC



Classification des branchements

- Conditionnel / inconditionnel
 - ▶ Branchement conditionnel
 - ★ Branchement pris si condition vraie
 - ★ Branchement non pris sinon
 - ★ Evaluation de la condition en général à l'étage d'exécution
 - ▶ Branchement inconditionnel
 - ★ Branchement systématique
- Immédiat / indirect
 - ▶ Branchement immédiat.
 - ★ Adresse cible connue à la compilation \Rightarrow peut être calculée au décodage
 - ▶ Branchement indirect
 - ★ Adresse cible lue dans un registre \Rightarrow connue seulement à l'étage d'exécution

Utilisation classique des branchements

- Branchement conditionnel immédiat
 - ▶ Boucles pour et tantque
 - ▶ Conditionnelle si ... alors ... sinon
 - ▶ Conditionnelle switch case
- Branchement inconditionnel immédiat
 - ▶ Tout type de boucles
 - ▶ Sortie de conditionnelle
 - ▶ Appel de fonction
- Branchement inconditionnel indirect
 - ▶ Retour de fonction
 - ▶ Pointeur de fonction
 - ▶ Saut longue distance
- Branchement conditionnel indirect
 - ▶ Très rare

Pour illustrer la prédition de branchement

- Recherche d'une valeur V dans un tableau trié T de N entiers

$bi \leftarrow 0; bs \leftarrow N-1;$

 tq $bi \neq bs$ faire

$m \leftarrow (bi+bs)/2;$

 si $T[m] < V$ alors

$bi \leftarrow m+1;$

 sinon si $T[m] > V$ alors

$bs \leftarrow m;$

 sinon

$bi \leftarrow m; bs \leftarrow m;$

 fsi;

 ftq;

 si $T[bi] = V$ alors

 retourner (bi);

 sinon

 retourner (-1);

 fsi;

Pour illustrer la prédition de branchement

set V, %r5	add %r4,1,%r2
ld [%r5],%r5	ba fsi
set T,%r6	sinon1: add %r4,%r0,%r3
set N,%r1	ba fsi
ld [%r1],%r1	sinon2: add %r4,%r0,%r2
add %r0,%r0,%r2	add %r4,%r0,%r3
sub %r1,1,%r3	ba tq
tq: subcc %r3,%r2,%r0	ftq: ld [%r6+%r2],%r7
be ftq	subcc %r7,%r5,%r0
add %r2,%r3,%r4	be fin
slr %r4,1,%r4	sub %r0,1,%r2
ld [%r6+%r4],%r7	ba fin
subcc %r7,%r5,%r0	V: .word 1
bg sinon1	N: .word 10
be sinon2	T: .word 1,3,4,7,9,12,15,20,24,30

- Branchements pris/pas pris ?

Pour illustrer la prédition de branchement

set V, %r5	add %r4,1,%r2
ld [%r5],%r5	ba fsi
set T,%r6	sinon1: add %r4,%r0,%r3
set N,%r1	ba fsi
ld [%r1],%r1	sinon2: add %r4,%r0,%r2
add %r0,%r0,%r2	add %r4,%r0,%r3
sub %r1,1,%r3	ba tq
tq: subcc %r3,%r2,%r0	ftq: ld [%r6+%r2],%r7
be ftq	subcc %r7,%r5,%r0
add %r2,%r3,%r4	be fin
slr %r4,1,%r4	sub %r0,1,%r2
ld [%r6+%r4],%r7	ba fin
subcc %r7,%r5,%r0	V: .word 14
bg sinon1	N: .word 10
be sinon2	T: .word 1,3,4,7,9,12,15,20,24,30

- Branchements pris/pas pris ?

Pénalité de mauvaise prédition

- Pipeline à six étages vu précédemment
 - ▶ 1 bulle dans le cas d'un branchement inconditionnel immédiat
 - ▶ 3 bulles dans le cas d'un branchement conditionnel ou indirect
- Dépend de la longueur du pipeline
 - ▶ Intel core ⇒ 14 cycles de pénalité de mauvaise prédition sur les branchements conditionnels

Principe de la prédition de branchement

- Table de prédition accédé en même temps que le cache d'instructions
- Permet de
 - ▶ Prédire qu'à l'adresse indiquée par le compteur de programme se trouve une instruction de branchement
 - ▶ Identifier le type de branchement
 - ★ Branchement conditionnel ou inconditionnel
 - ★ Appel de fonction
 - ★ Retour de fonction
 - ▶ Prédire l'adresse de saut
 - ▶ Pour les branchements conditionnels, prédire si le branchement est pris ou non
- Mauvaise prédition \Rightarrow profiter du temps de réparation du pipeline pour corriger l'information stockée dans la table de prédition

Table de prédition

Est-ce un branchement ?

Si oui :

info table = (typre, pris/non pris, @saut)

PC := F (PC, info table)

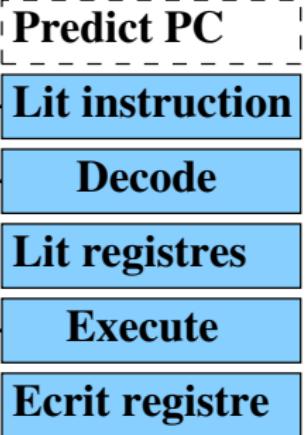
Si non :

PC := PC + 1

PC suivant



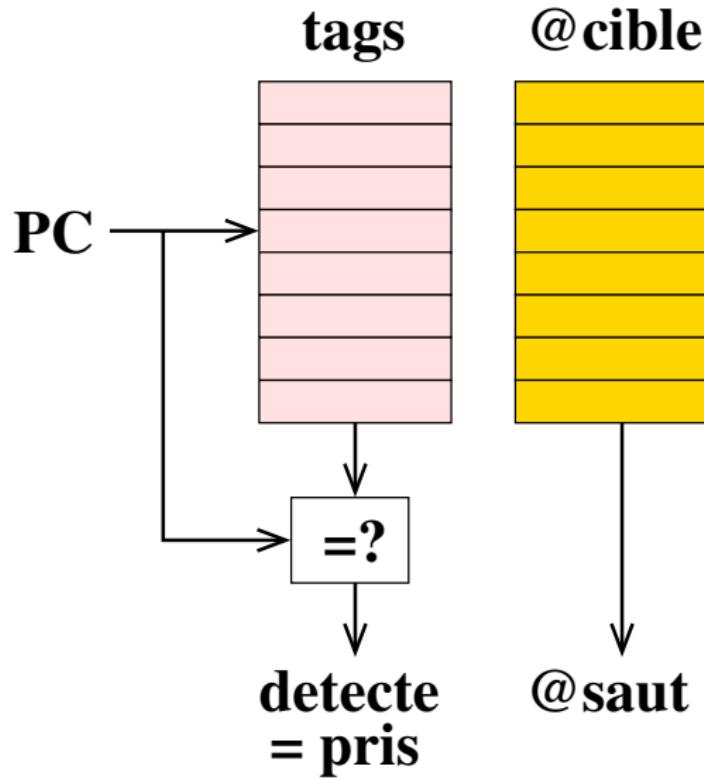
PC courant



Corrige si mauvaise prediction

Cache de branchements

- BTB : Branch Target Buffer



Utilisation du BTB

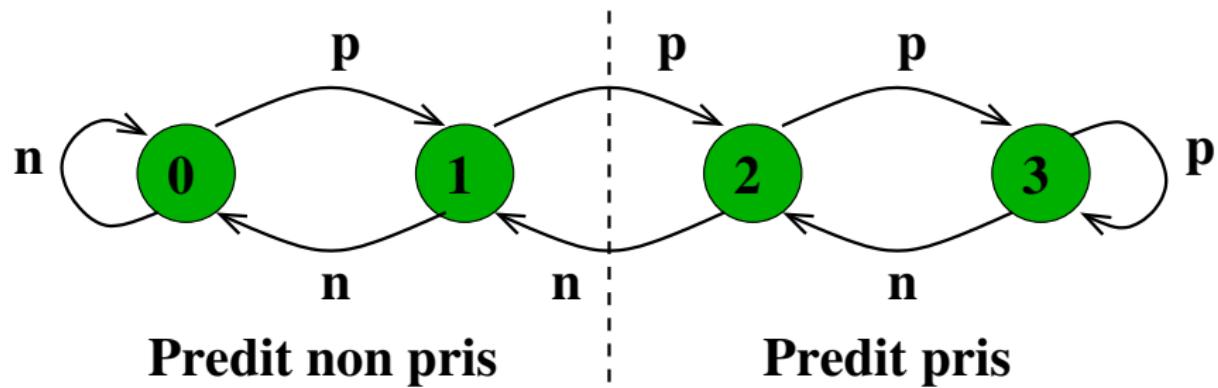
- Stockage dans le BTB
 - ▶ Des branchements inconditionnels
 - ▶ Des branchements conditionnels pris
- Entrée présente dans le BTB \Rightarrow saut à l'adresse indiquée par le BTB
- Entrée absente du BTB \Rightarrow pas un branchement ou branchement conditionnel non pris
- Mauvaise prédition \Rightarrow correction du BTB
 - ▶ Saut manqué \Rightarrow ajout de l'entrée manquante
 - ▶ Branchement prévu pris, mais non pris \Rightarrow retrait de l'entrée du BTB
 - ▶ Saut à une mauvaise adresse \Rightarrow correction de l'adresse dans le BTB

Remarques sur BTB

- Autres noms
 - ▶ BTAC : Branch Target Address Cache
 - ▶ Next-Fetch Predictor
 - ▶ ...
- Taille très variable
 - ▶ 8 entrées sur certains processeurs
 - ▶ 2k entrées sur d'autres
- Associativité
 - ▶ de 1 (direct-map) à 4
- Prédiction "comme la dernière fois"
 - ▶ Performant pour les branchements inconditionnels immédiats
 - ▶ Pas très performant pour les branchements conditionnels et les retours de fonction

Prédiction avec compteur 2 bits

- Incrémentation du compteur lorsque le branchement est pris (p)
- Décrémentation du compteur lorsque le branchement est non pris (n)
- Prédiction pris ssi valeur du compteur ≥ 2

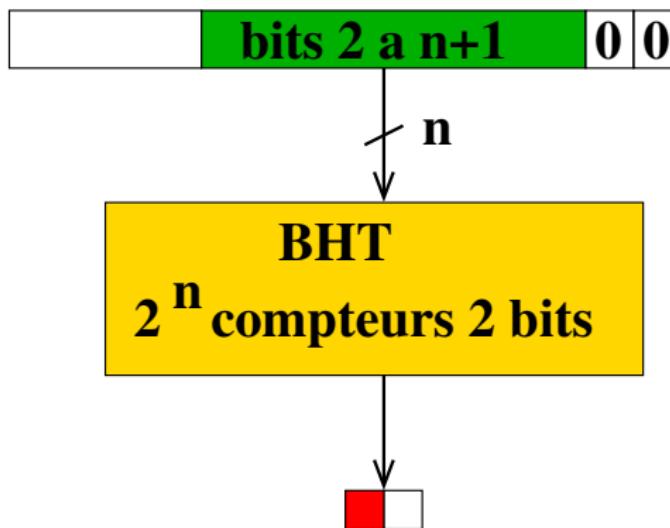


Mise en œuvre du compteur 2 bits

- Solution 1
 - ▶ Ajout d'un compteur 2 bits dans chaque entrée du BTB
 - ▶ Permet de bénéficier de l'associativité du BTB
 - ▶ Branchement prédit pris et non pris ⇒
 - ★ Décrémentation du compteur
 - ★ L'entrée reste dans le BTB
- Solution 2
 - ▶ Stockage des compteurs 2 bits dans une table spécifique, la BHT (Branch History Table)
 - ▶ Pas de tags, seulement les 2 bits des compteurs dans la table
 - ▶ Eventuellement plus d'entrées que dans le BTB
 - ▶ Un compteur passe de l'état 2 à l'état 1 ⇒ entrée correspondante dans le BTB plus vraiment nécessaire

BHT : prédicteur bimodal

PC du branchement conditionnel



- Bit de poids fort du compteur donne la prédiction
- Pas de tags \Rightarrow pas d'associativité
- Interférences potentielles entre branchements

Processeurs superscalaires

- Processeur pouvant décoder plusieurs instructions du même thread (~ programme) par cycle
- Efficacité dépend du parallélisme d'instructions des programmes
 - ▶ Notion qualitative caractérisant la proportion d'instructions d'un programme pouvant être exécutées en parallèle
 - ▶ Programme comportant un nombre important de suites d'instructions consécutives indépendantes ⇒ parallélisme d'instructions élevé
 - ▶ Programme comportant beaucoup d'instructions consécutives dépendantes (e.g. longue chaîne de dépendance) ⇒ parallélisme d'instructions faible

Différents types de superscalaires

- Superscalaire in-order
 - ▶ Instructions lancées à l'exécution dans l'ordre séquentiel, c'est-à-dire l'ordre du programme
- Superscalaire out-of-order
 - ▶ Possibilité de lancer les instructions dans un ordre différent de l'ordre séquentiel
- Superscalaire de différents degrés
 - ▶ Superscalaire de degré $n \Rightarrow$ décodage d'au plus n instructions par cycle
- Augmentation de la complexité matérielle avec l'augmentation du degré
 - ▶ Aujourd'hui en général au plus 6
 - ▶ Limiteurs principaux
 - ★ Nombre de ports sur les bancs de registres
 - ★ Complexité du mécanisme de bypass
 - ★ Parallélisme d'instructions des programmes

Le plus simple : in-order de degré 2

- Doublement de la largeur du pipeline
 - ▶ Lecture de deux instructions par cycle : $PC \leftarrow PC + 2$
 - ▶ Décodage de deux instructions par cycle \Rightarrow deux décodeurs
 - ▶ Lecture registres : 4 ports de lecture
 - ▶ Exécution : duplication des opérateurs
 - ▶ Cache de données : deux ports de lecture/écriture
 - ▶ Ecriture registres : deux ports d'écriture
- En pratique, on ne double que les ressources utilisées fréquemment
 - ▶ un seul diviseur au lieu de deux
 - ▶ Cache de données : un seul port de lecture/écriture

Exemple

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

- I2 dépend de I1 \Rightarrow on retarde le lancement de I2

Principe du décalage des groupements

	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			

- I2 dépend de I1 et I3 ne dépend pas de I2 \Rightarrow lancement de I2 et I3 ensemble

Mise en œuvre du décalage des groupements

	Cycle N	Cycle N+1	Cycle N+2	Cycle N+2	Cycle N+2
Lit instruction	I3 / I4	I5 / I6	I7 / I8		I9 / I10
Decode	I1 / I2	I3 / I2	I5 / I4	I5 / I6	I7 / I8
Lit registres		I1	I3 / I2	I4	I5 / I6
Execute			I1	I3 / I2	I4
Ecrit registre				I1	I3 / I2

The diagram illustrates the execution flow of multiple instructions across five cycles. The columns represent Cycles N through N+4. The rows represent different stages of the pipeline: Loading Instructions (Lit instruction), Decoding (Decode), Loading Registers (Lit registres), Executing (Execute), and Writing Registers (Ecrit registre). Instructions are represented by yellow boxes labeled with their identifiers (I1 through I10). Dependencies are shown by arrows pointing from earlier instructions to later ones. For example, in Cycle N+2, there are two downward arrows from I3 and I4 to I7, indicating that both I3 and I4 must be completed before I7 can begin.

- I2 dépend de I1
- I5 dépend de I4

Processeurs multi-cœurs

- Plusieurs processeurs (cœurs) sur une même puce de silicium
- Les caches peuvent être locaux (un pour chaque cœur) ou partagés
- Exemples
 - ▶ Intel Core Duo → 2 cœurs
 - ▶ AMD Phenom → 4 cœurs
 - ▶ Sun Ultrasparc T2 → 8 cœurs

Parallélisme de thread

- Exploitation de la performance potentielle d'un multi-cœur ⇒ exécution de plusieurs threads simultanément
- Exemple : 2 cœurs, 2 threads ⇒ 1 thread s'exécute sur chaque cœur
- Chaque thread a son propre compteur de programme et ses propres registres
- Les threads se partagent le même espace d'adressage et communiquent en accédant à des variables partagées
- C'est au programmeur d'exposer du parallélisme de thread

Processeur SMT

- Simultaneous MultiThreading (SMT) : une autre manière d'exploiter le parallélisme de thread sur une même puce
 - ▶ Processeur superscalaire : rare de réussir à exécuter à chaque cycle un nombre d'instructions égal au degré superscalaire
 - ★ Dépendances entre instructions, miss de cache, ...
 - ▶ Processeur SMT capable d'exécuter plusieurs threads simultanément
 - ★ Plusieurs PC (e.g. cycle N lit instructions thread 1, cycle $N + 1$ lit instructions thread 2, ...)
 - ★ Instructions de plusieurs threads dans le pipeline d'instructions à un instant donné
 - ★ Partage de resources (caches, opérateurs, ...) entre les threads
 - ★ Chaque thread a ses propres registres
- Un processeur peut être à la fois multicœur et SMT
- Exemples de processeurs SMT
 - ▶ IBM Power 5 : 2 threads par cœur
 - ▶ Sun Ultrasparc T2 : 8 threads par cœur

Architecture à mémoire partagée

- Multi-cœurs généralistes : souvent architectures à mémoire partagée
 - ▶ Peuvent exécuter simultanément plusieurs threads ayant le même espace d'adressage
- Définition d'un modèle de comportement de la mémoire partagée
 - ▶ Consistency model
 - ▶ Contrat avec le programmeur
 - ▶ Modèle le plus connu : Sequential Consistency
- Nécessité d'un mécanisme de cohérence

Mécanisme de cohérence

- Niveaux de cache locaux et niveaux de cache partagés
 - ▶ Un exemple : L1 locaux, niveaux suivants partagés
 - ▶ Autre exemple : L1 et L2 locaux, niveaux suivants partagés
- Plusieurs cœurs accèdent en lecture à une même ligne de cache ⇒ une copie de la ligne stockée sur chaque cœur
- Un des cœurs veut modifier la ligne ⇒ invalider les autres copies avant d'autoriser la ligne à être modifiée
 - ▶ Ecriture atomique effectuée quand on est sûr que plus aucun cœur ne peut lire l'ancienne valeur
- Miss avec des caches locaux write-back (écriture différée) ⇒ regarder dans les caches locaux des autres cœurs
 - ▶ Existence d'une copie dirty de la ligne ⇒ lecture de cette copie
- Différents protocoles existent

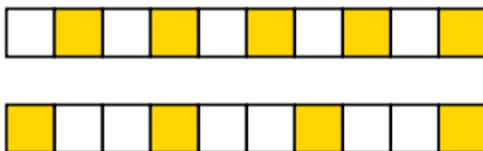
Processeurs vectoriels

- Fonctionnalités adaptées aux calculs sur des tableaux, des matrices, des vecteurs
- Instructions de calcul vectoriel
 - ▶ Effectuent des opérations sur des vecteurs : données de même taille et de même type
 - ▶ Exemple d'une addition

$$\begin{array}{|c|c|c|c|} \hline 17 & 32 & 20 & 45 \\ \hline \end{array} \text{ Vecteur 1}$$
$$+$$
$$\begin{array}{|c|c|c|c|} \hline 25 & 18 & 52 & 48 \\ \hline \end{array} \text{ Vecteur 2}$$
$$=$$
$$\begin{array}{|c|c|c|c|} \hline 42 & 50 & 72 & 93 \\ \hline \end{array} \text{ Vecteur résultat}$$

Processeurs vectoriels

- Instructions d'accès mémoire
 - ▶ Au moins une instruction de lecture de vecteurs
 - ▶ Au moins une instruction d'écriture de vecteurs
 - ▶ Le plus souvent architecture de type load/store
 - ▶ Stockage des vecteurs dans des registres vectoriels
 - ▶ Différents modes d'adressage possibles
 - ★ Adressage absolu lorsque les données du vecteur sont contigües
 - ★ Accès en stride lorsque les données sont séparées par un intervalle régulier
 - ★ ...



Exemples d'accès en stride

Processeurs vectoriels

- Vectorisation
 - ▶ Déroulage de boucles
 - ★ Indépendance des exécutions successives du corps d'une boucle simple
⇒ réplication du corps de la boucle pour exécuter en parallèle plusieurs tours

```
for (i=0;i<100;i++)  
{  
    a[i] = b[i]*2;  
}
```

devient

```
for (i=0;i<100;i+=4)  
{  
    a[i] = b[i]*2;  
    a[i+1] = b[i+1]*2;  
    a[i+2] = b[i+2]*2;  
    a[i+3] = b[i+3]*2;  
}
```

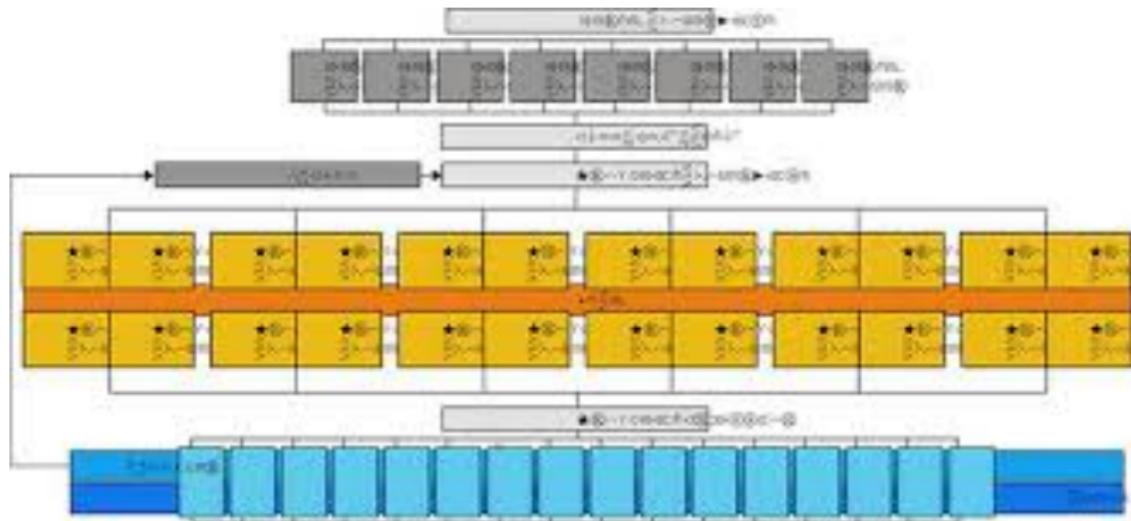
Processeurs vectoriels

- Quelques éléments d'architecture
 - ▶ Mémoires caches pour les instructions vectorielles
 - ▶ Organisation mémoire pour faciliter la lecture ou l'écriture des vecteurs
 - ★ Mémoire découpée en bancs \Rightarrow accès en parallèle aux différents bancs
 - ★ Mémoire entrelacée \Rightarrow adresses consécutives dans des bancs mémoire consécutifs
 - ★ Peu adapté au stride \Rightarrow il existe des organisations plus complexes
 - ▶ Unité de calcul vectorielle vectorisée

General-purpose GPU

- GPU : Graphics Processing Unit, co-processeur graphique
- Structure hautement parallèle ⇒ très efficace pour les tâches graphiques
 - ▶ Rendu 3D
 - ▶ Décompression Mpeg
 - ▶ ...
- Grande capacité de calcul pour des applications hautement parallèles avec peu de structures de contrôle
- Nvidia : inventeur du GPU en 1999
- Illustration de l'évolution des GPU nvidia pour faire autre chose que du graphique

GPU Nvidia jusqu'en 2006



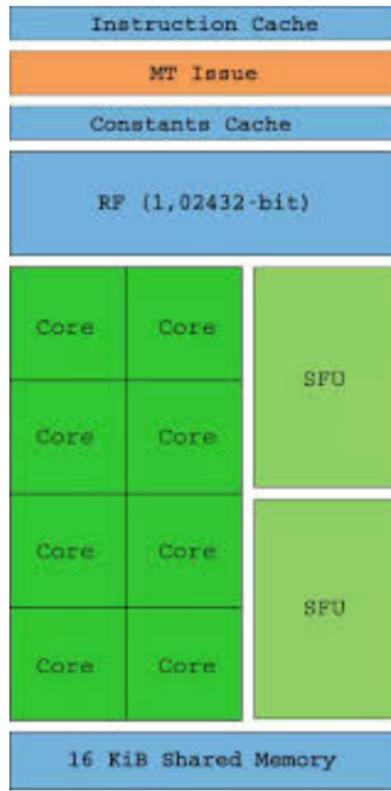
- Processeurs dédiés pour le traitement de sommet, la rastérisation et la fusion des fragments
- Ne correspond pas nécessairement au besoin d'une application non graphique

2006 : Nvidia introduit l'architecture Tesla



- Plus de processeur dédié
- Exécution de chaînes de 32 threads
 - ▶ SIMT (Single Instruction Multiple Threads) : tous les threads de la chaîne exécutent la même instruction sur des données différentes
- Langage de programmation CUDA (Compute Unified Device Architecture)

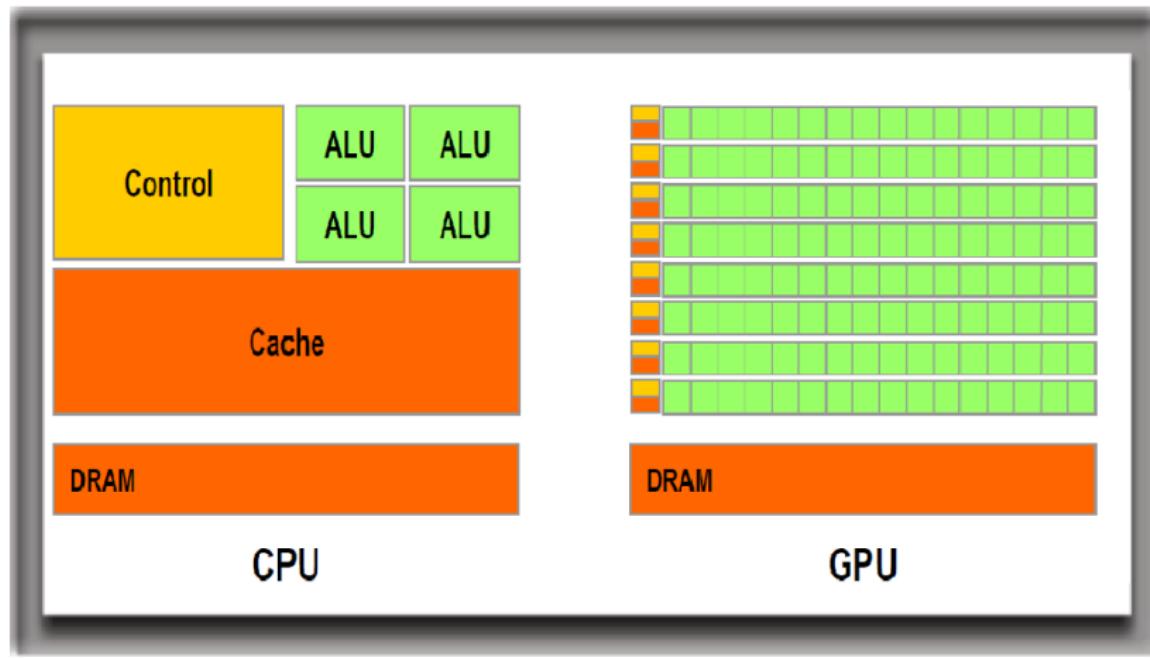
Nvidia Tesla : Stream Multiprocessor



- 8 cœurs de calcul
- 2 unités pour le calcul flottant : 4 fois moins que de cœurs \Rightarrow chaîne exécutée 4 fois moins vite
- Toujours avoir des chaînes exécutables
- RF (Register File) : stockage de l'état des threads, threads trop gourmands pénalisants
- MT (Multi-Threaded instruction unit) : autorise/bloque l'exécution des threads de la chaîne s'ils convergent/divergent

Accès mémoire et GPU

- Pas de hiérarchie mémoire complexe
- Haut débit avec six liens bi-directionnels vers la DRAM

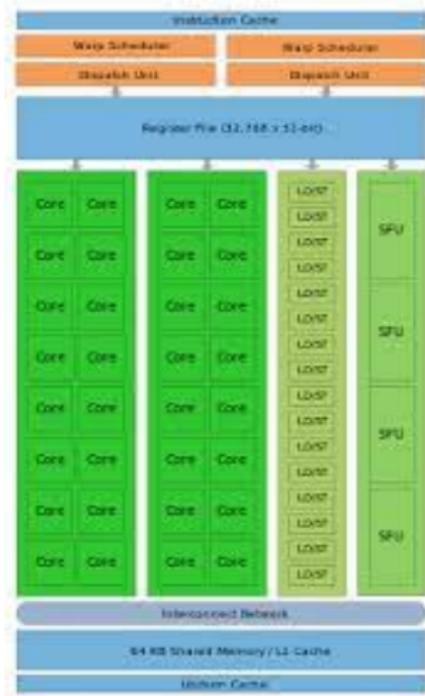


2010 : Nvidia introduit l'architecture Fermi



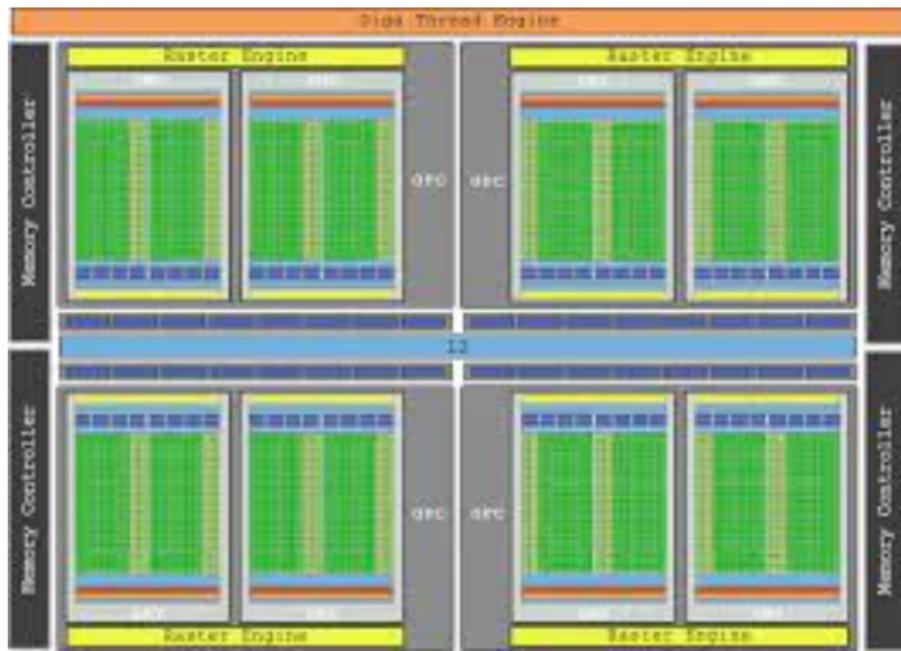
- 512 cœurs : 4 Graphics Processor Clusters de 4×32 cœurs

Nvidia Fermi : Stream Multiprocessor



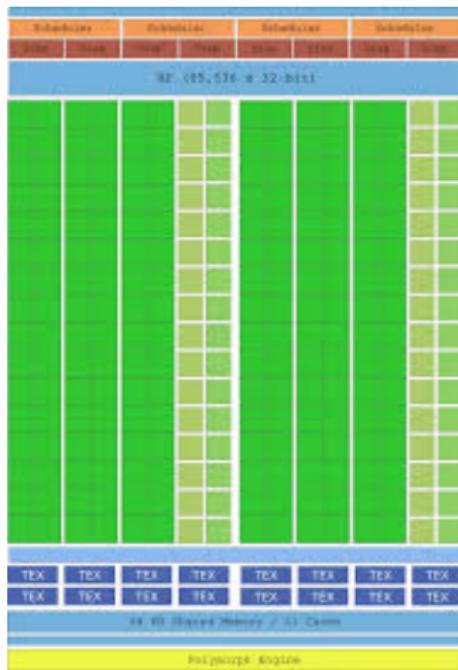
- 32 coeurs de calcul
- 4 unités pour le calcul flottant
- Accélération des opérations flottantes sur 64 bits

2012 : Nvidia introduit l'architecture Kepler



- Plus efficace en énergie (horloges plus lentes)
- Pas de perte de performance, car plus de SM avec un design amélioré

Nvidia kepler : Stream Multiprocessor



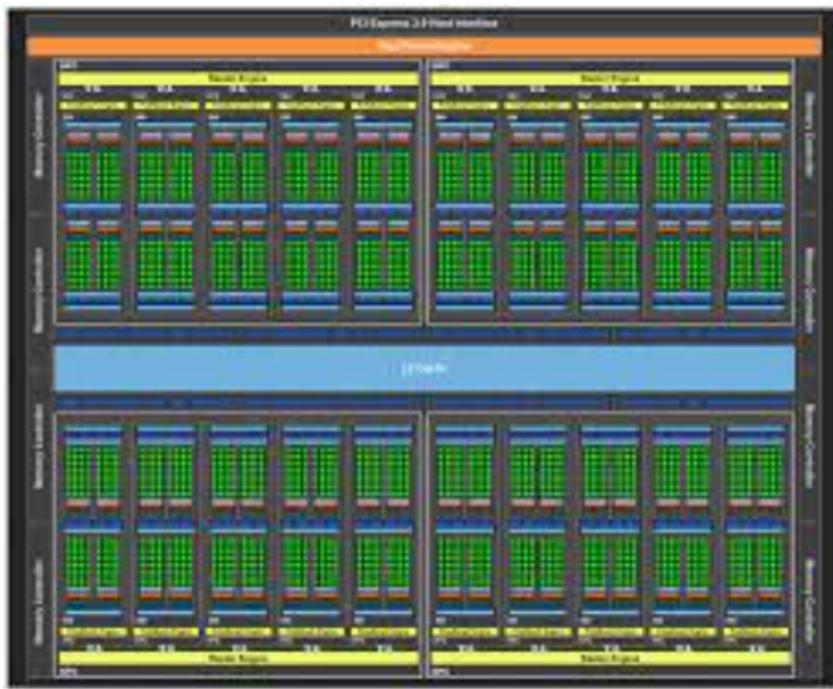
- 196 cœurs de calcul
- 4 ordonneurs de chaînes
- 2 instructions de la même chaîne peuvent être exécutées en parallèle si elles sont indépendantes

2014 : Nvidia introduit l'architecture Maxwell



- Moins de coeurs par SM (128), mais plus de SM, donc plus de coeurs au total (3072)

2016 : Nvidia introduit l'architecture Pascal



- Pas très différent de Maxwell
- Une technologie plus compacte et un débit mémoire plus important

2018 : Nvidia introduit l'architecture Turing

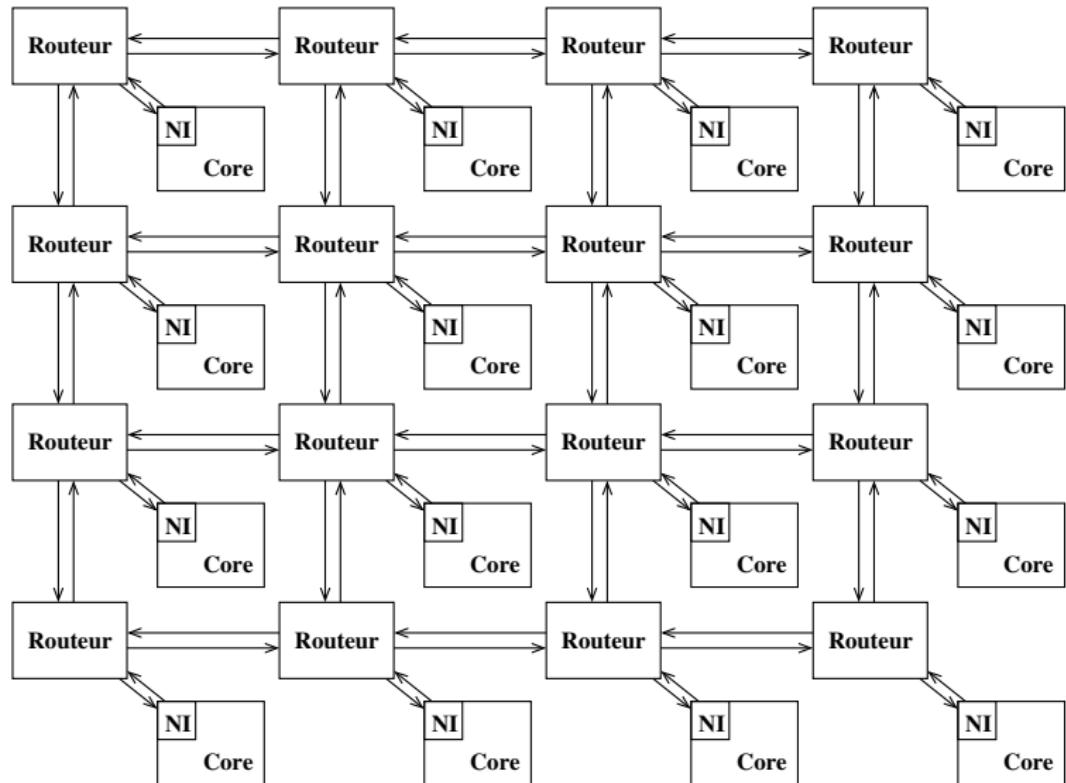


- Architecture totalement différente avec le retour de coeurs spécialisés

Les architectures many-cœurs

- Processeurs multi-cœurs complexes
 - ▶ Chaque cœur implémente des mécanismes complexes
- Idée de départ des many-cœurs
 - ▶ Un grand nombre de cœurs moins complexes sur une puce
 - ▶ Chaque cœur peut disposer d'un OS comme Linux
 - ▶ Possibilité d'avoir un OS multiprocesseur sur plusieurs cœurs
 - ▶ Le nombre de cœurs amène la puissance
- Problème d'interconnexion de ces cœurs
 - ▶ Bus insuffisant
 - ▶ Structures complexes (tores, hypercubes, ...) peu/pas utilisées
 - ▶ En général réseaux mesh 2D (passe bien à l'échelle)
 - ▶ Exemple du Tilera Tile64 : 5 réseaux mesh 2D

NoC : architecture générale



NoC avec garanties de Qualité de service

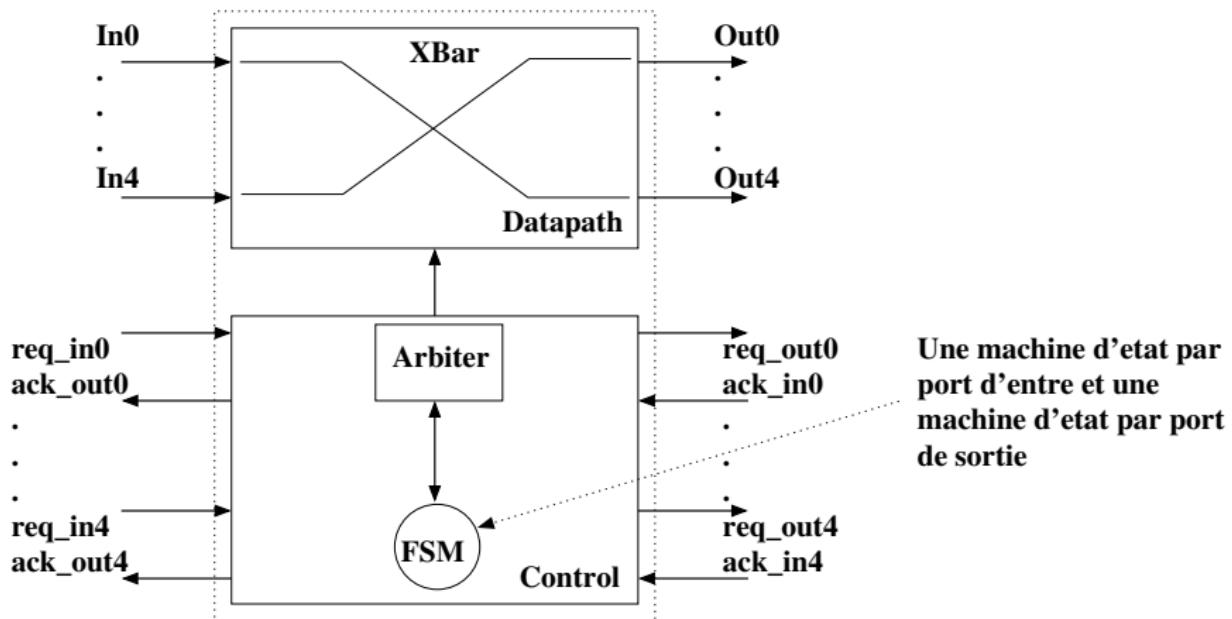
- Garantir des contraintes spécifiées par des applications
 - ▶ Perte de paquets
 - ▶ Délai, gigue
 - ▶ Débit disponible
- Grandes classes de NoC pour y répondre
 - ▶ Dimensionnement ad hoc en fonction des applications
 - ★ Ne garantit pas l'absence de congestion
 - ★ Inadapté lorsque les applications varient
 - ▶ Commutation de circuit
 - ▶ Commutation de paquets
 - ▶ Multiplexage temporel
 - ▶ Combinaison des approches précédentes

NoC à commutation de circuit

- La transmission d'un ensemble de paquets se déroule en trois étapes
 - ▶ Réservation des ressources sur le chemin de la source à la destination
 - ▶ Transmission des paquets
 - ▶ Libération des ressources
- Deux parties dans un routeur
 - ▶ La partie en charge du contrôle
 - ★ Une machine d'état par port d'entrée et une machine d'état par port de sortie
 - ★ Un arbitre qui décide pour chaque requête et configure le crossbar
 - ▶ La partie en charge de la transmission des données
 - ★ Un crossbar reliant les ports d'entrée aux ports de sortie

NoC à commutation de circuit

- Architecture d'un routeur



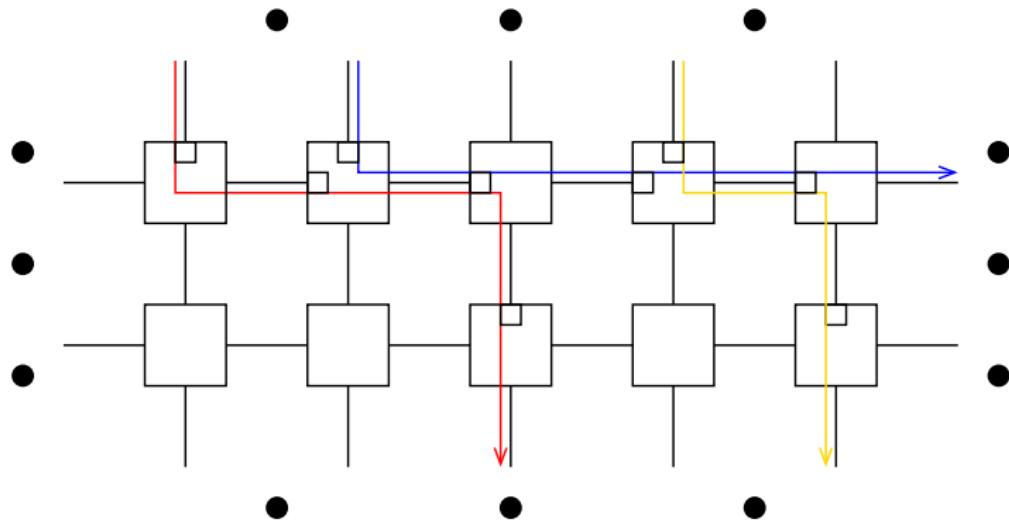
NoC à commutation de circuit

- Transmission sans contention \Rightarrow durée constante, calculable
 - ▶ Exemple : transmission d'un message de z flits entre une source S et une destination D séparées par $h_{S \rightarrow D}$ sauts
$$(h_{S \rightarrow D} + z - 1) \times T_{data}$$
- Réservation des ressources : dépend des autres flux et des autres requêtes
 - ▶ Sur le même exemple, avec FWD cycles pour un traitement réussi de la requête sur un routeur
$$(FWD + 1) \times T_{ctrl} T_{blocking}$$
 - ▶ Comment réduire/borner $T_{blocking}$?
 - ★ Echec au niveau d'un routeur \Rightarrow retour arrière et la requête suit un autre chemin (les plus courts chemins sont considérés)
 - ★ Envoi de la requête sur plusieurs chemins en parallèle

NoC à commutation de paquet

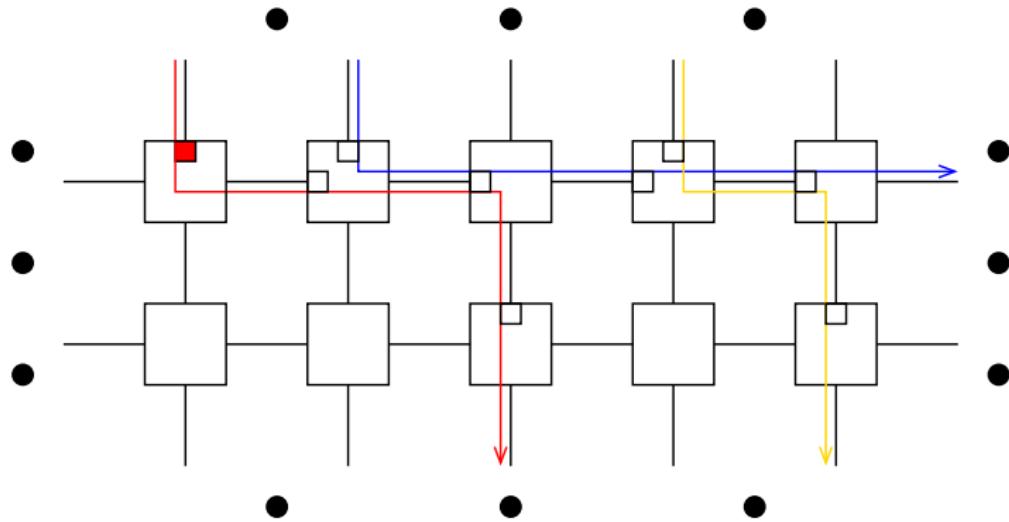
- Pour améliorer l'utilisation de la bande passante et des ressources réseau
- Ordonnancement dynamique des flux au niveau de chaque routeur
- Fondé sur le wormhole switching
 - ▶ Très peu de mémoire au niveau des routeurs
 - ▶ Paquet découpé en flits
 - ★ Premier flit détermine le routage et réserve le chemin
 - ★ Les flits suivants suivent sans être interrompués
 - ★ Entraîne des blocages directs et des blocages indirects
 - ★ Tourniquet entre les ports d'entrée
 - ▶ Très difficile de borner le délai
 - ★ Blocages directs
 - ★ Blocages indirects entre des flux qui ne partagent aucun lien

Wormhole switching



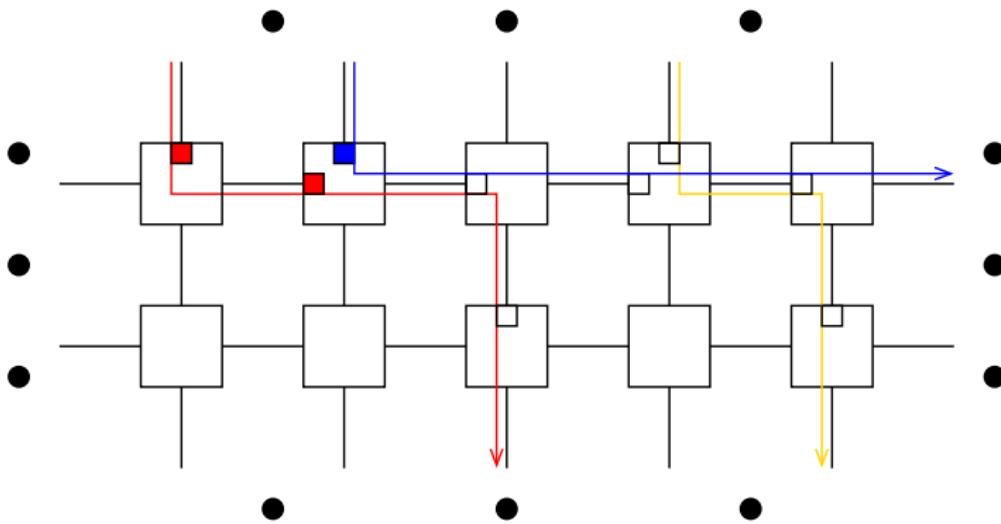
- Three flows, three flit packets

Wormhole switching



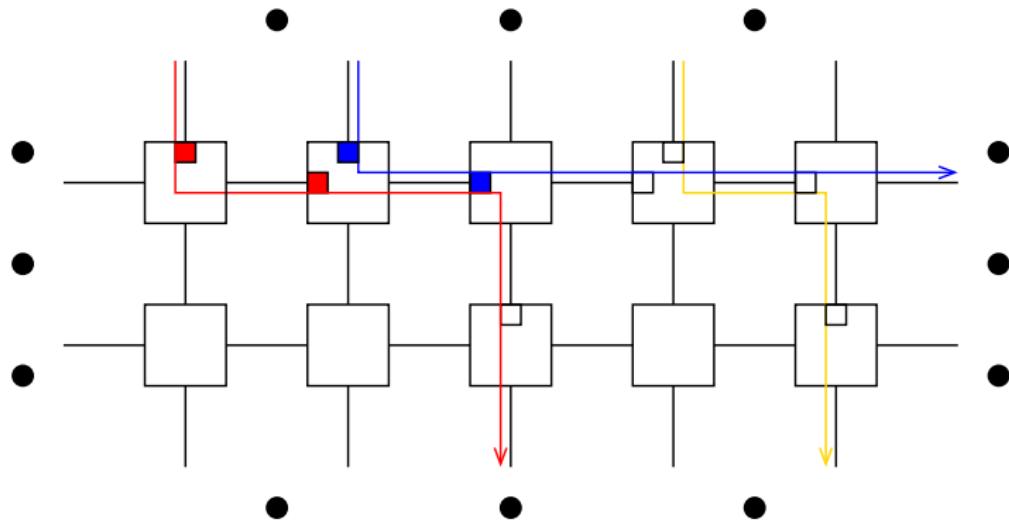
- Red flow: first flit in first router on its path

Wormhole switching



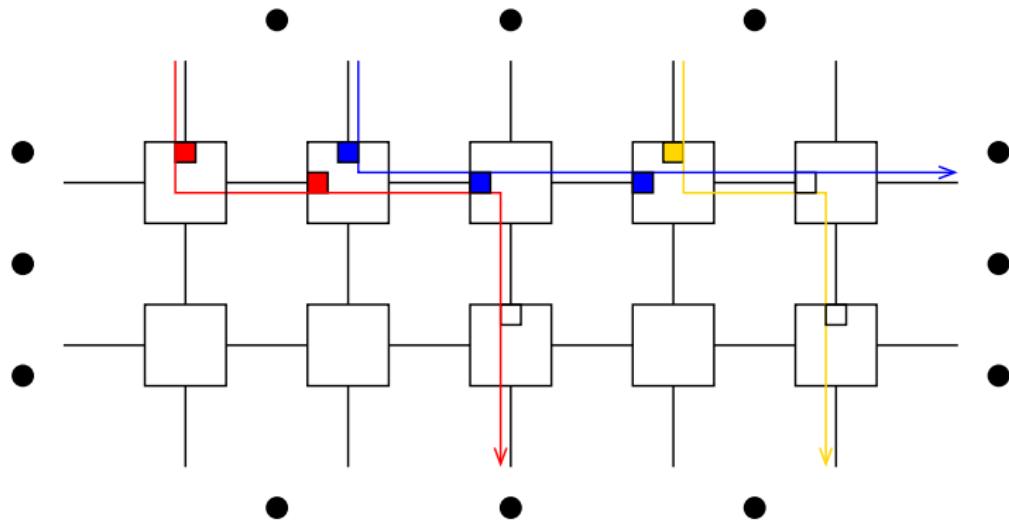
- Red flow: two first flits in two first routers on its path
- Blue flow: First flit in first router on its path

Wormhole switching



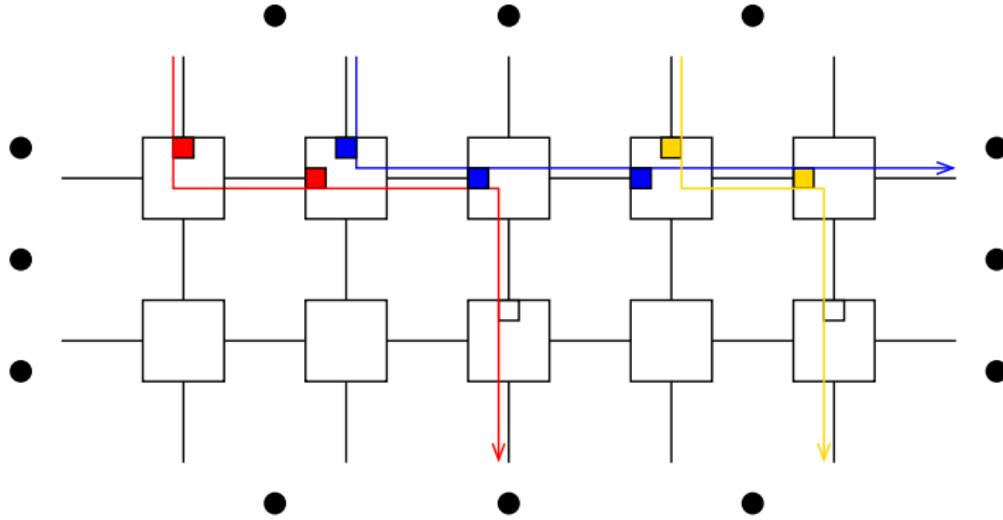
- Red flow directly blocked by blue flow

Wormhole switching



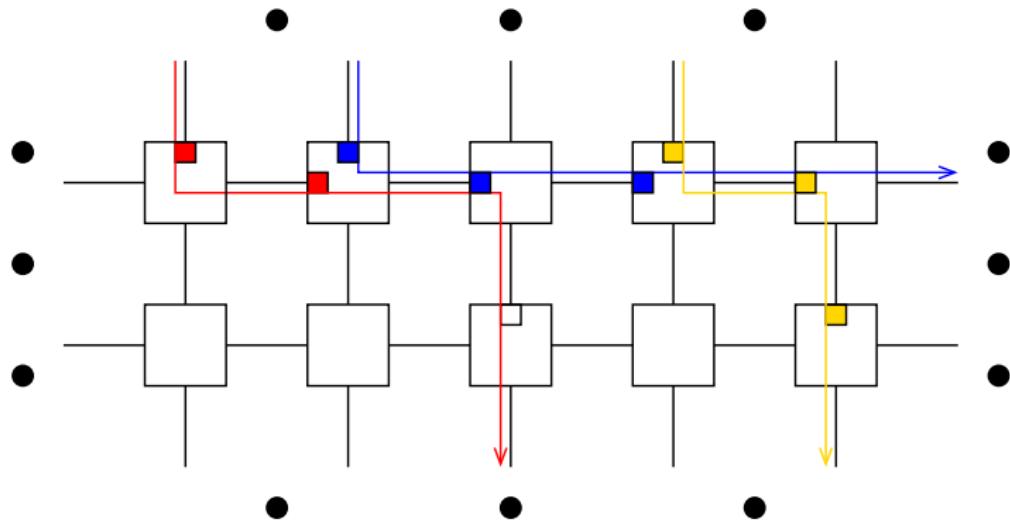
- blue flow progresses to next router on its path

Wormhole switching



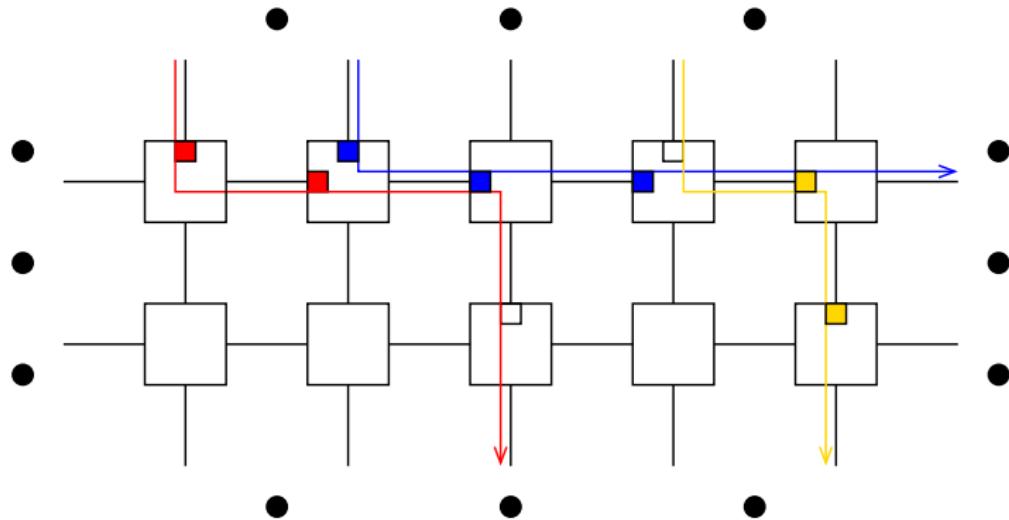
- blue flow directly blocked by yellow flow \Rightarrow red flow indirectly blocked by yellow flow

Wormhole switching



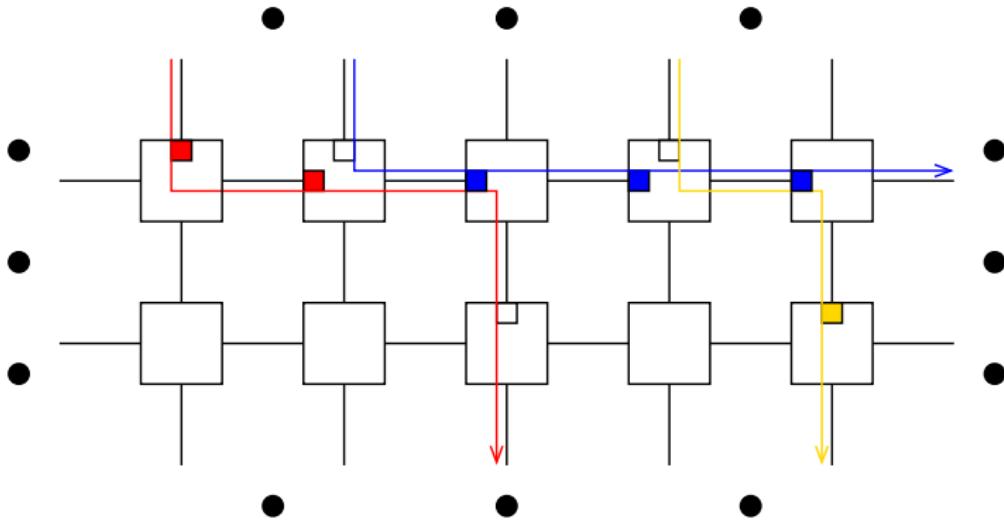
- One step for yellow flow

Wormhole switching



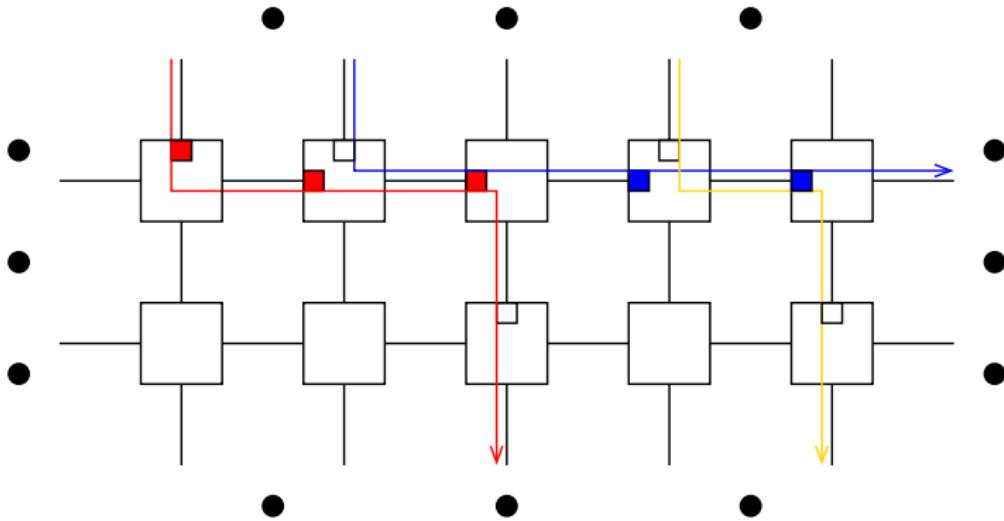
- One step for yellow flow

Wormhole switching



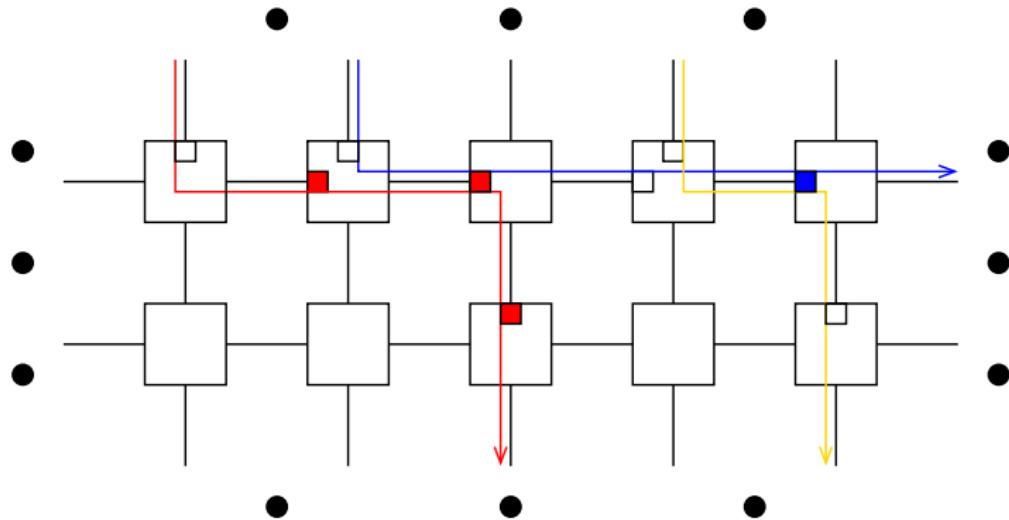
- One step for yellow flow
- One step for blue flow (no more blocked)

Wormhole switching



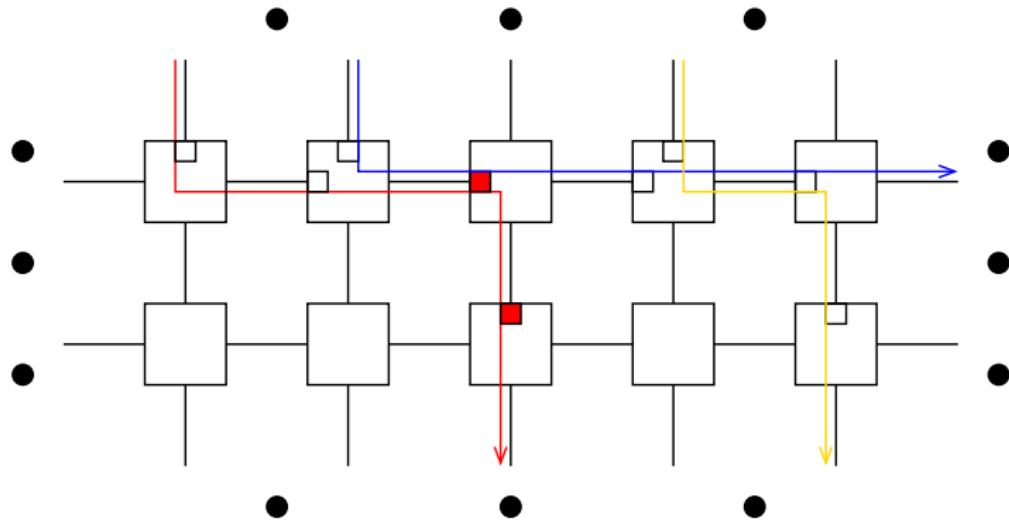
- One step for blue flow
- One step for red flow (no more blocked)

Wormhole switching



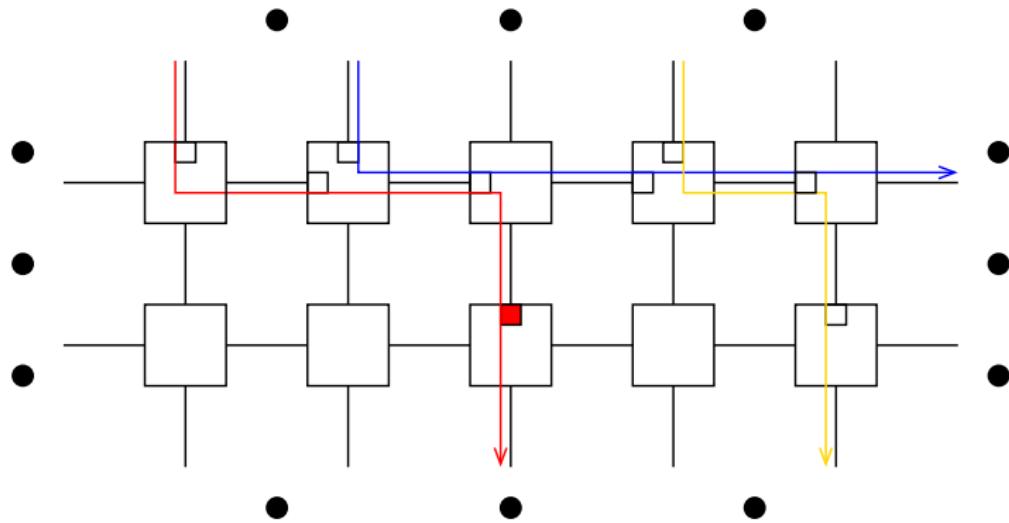
- One step for red flow

Wormhole switching



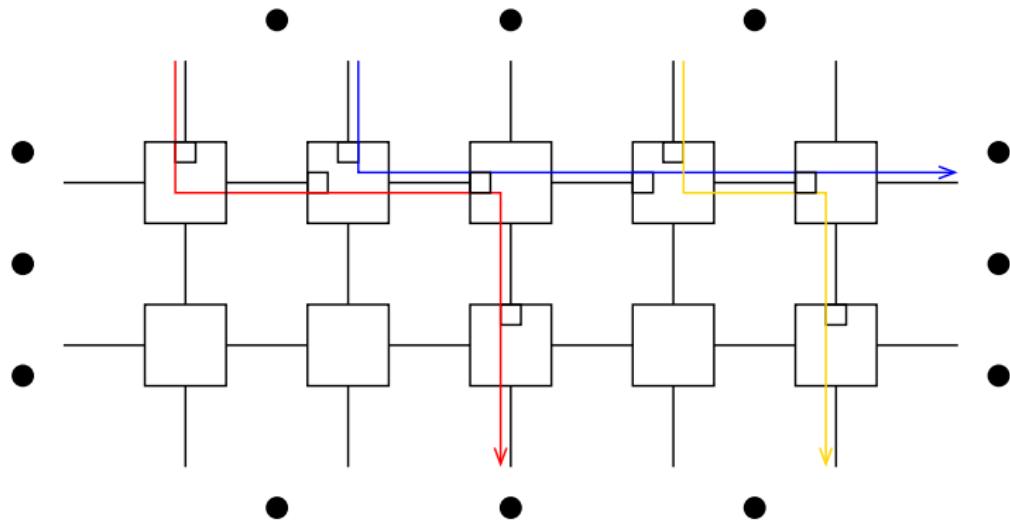
- One step for red flow

Wormhole switching



- One step for red flow

Wormhole switching



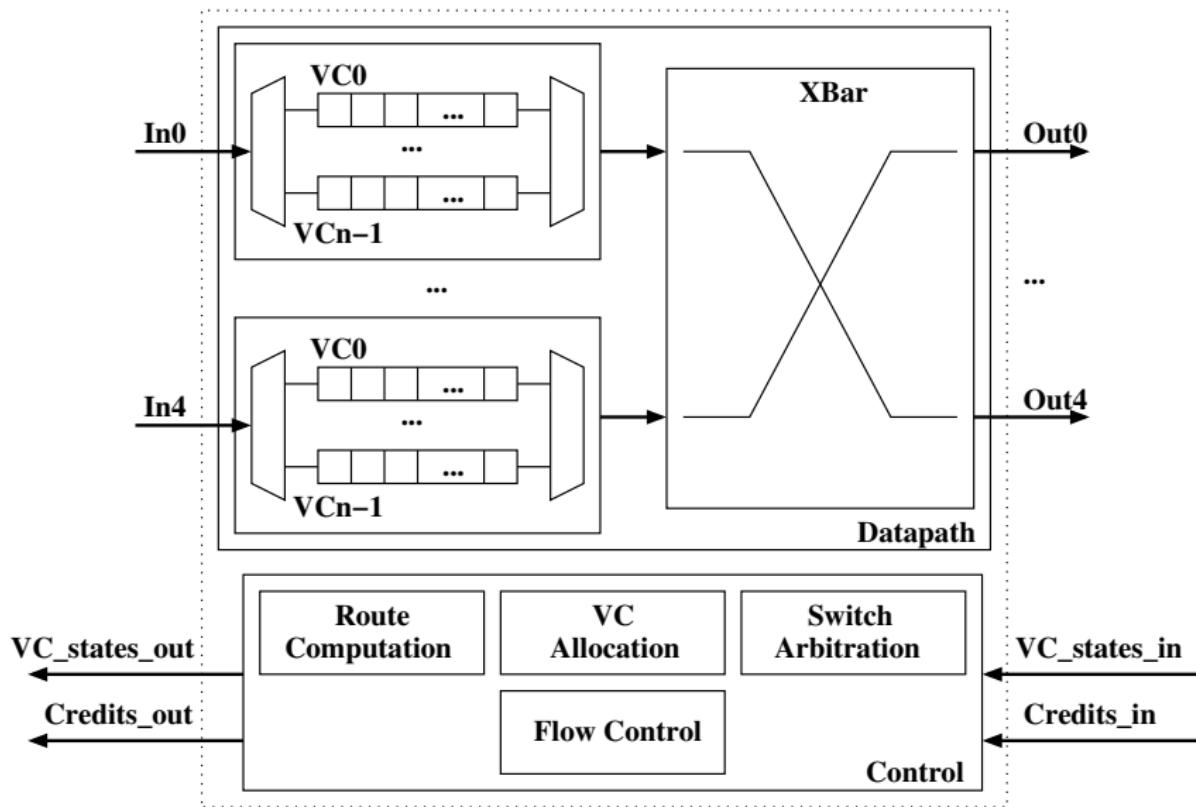
- One step for red flow

NoC à commutation de paquet

- Classement des flux par priorité
 - ▶ Intéressant quand des applications hétérogènes partagent le NoC
 - ▶ Limite le blocage des flux les plus critiques
- Possibilité de faire de la préemption
 - ▶ Un paquet moins prioritaire en cours de transmission laisse passer un paquet plus prioritaire
 - ▶ Les buffers doivent être différenciés en fonction des classes ⇒ Virtual Channels

NoC à commutation de paquet

- Architecture d'un routeur



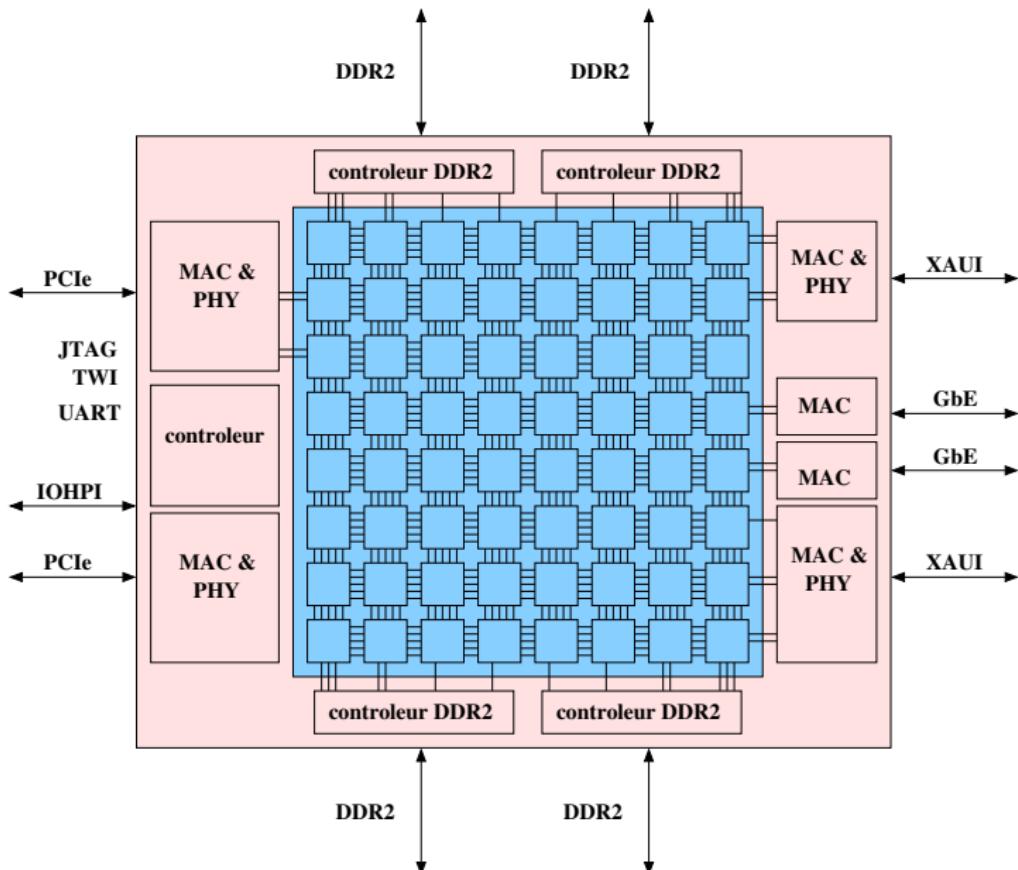
NoC à commutation de paquet

- Délai fortement impacté par
 - ▶ l'allocation des tâches aux différents cœurs de calcul
 - ▶ l'allocation des priorités
- Différentes méthodes pour l'analyse pire cas
 - ▶ Network calculus, recursive calculus, ...
- Exemples de NoC à commutation de paquet
 - ▶ QNoC
 - ★ Quatre classes de trafic, priorité fixe
 - ★ Round Robin entre les flux de même priorité
 - ★ Possibilité de shapers pour éviter les situations de famine
 - ▶ BiNoC
 - ★ Possibilité de configurer le sens des liens

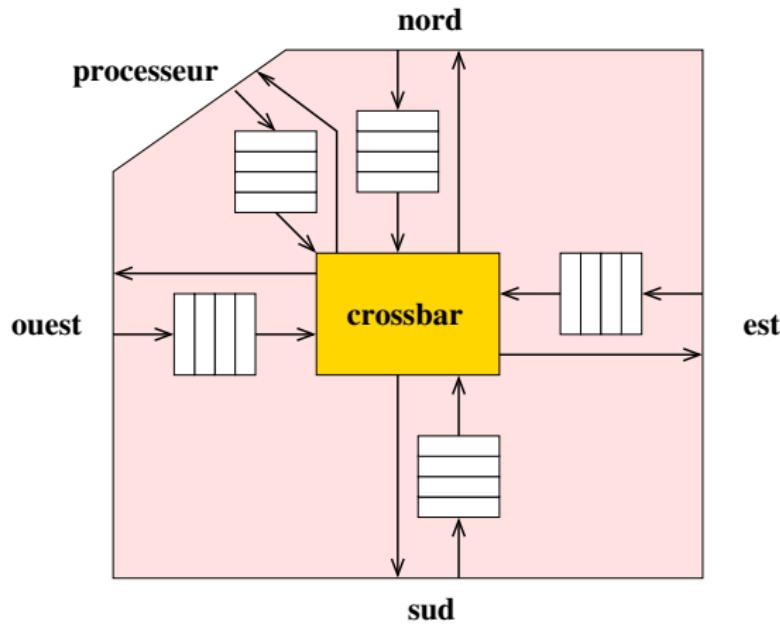
NoC à multiplexage temporel

- Ordonnancement global des transmissions par un multiplexage temporel
- Calcul statique ou dynamique de cet ordonnancement global
- Deux possibilités d mise en œuvre
 - ▶ Table dans chaque routeur ⇒ routage distribué
 - ▶ Seulement au niveau des interfaces réseau ⇒ routage par la source
- Les routeurs doivent être synchronisés
- La construction de l'ordonnancement est un problème difficile

Tilera Tile64



Architecture d'un crossbar



Organisation des transmissions

- Routage X-Y
 - ▶ Transmission horizontale jusqu'à la bonne colonne
 - ▶ Transmission verticale jusqu'à la destination
- Très peu de mémoire au niveau des routeurs ⇒ wormhole switching
 - ▶ Paquet découpé en flits
 - ★ Premier flit détermine le routage et réserve le chemin
 - ★ Les flits suivants suivent sans être interrompués
 - ★ Entraîne des blocages directs et des blocages indirects
 - ★ Tourniquet entre les ports d'entrée