Ecole thématique Archi 2019, Lorient, 20-24 mai

Si le premier ordinateur avait eu plusieurs cœurs ?

Bernard Goossens

Université de Perpignan Via Domitia, DALI-LIRMM

UNIVERSITÉ PERPIGNAN VIA DOMITIA

LIRMM
Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier

Sommaire

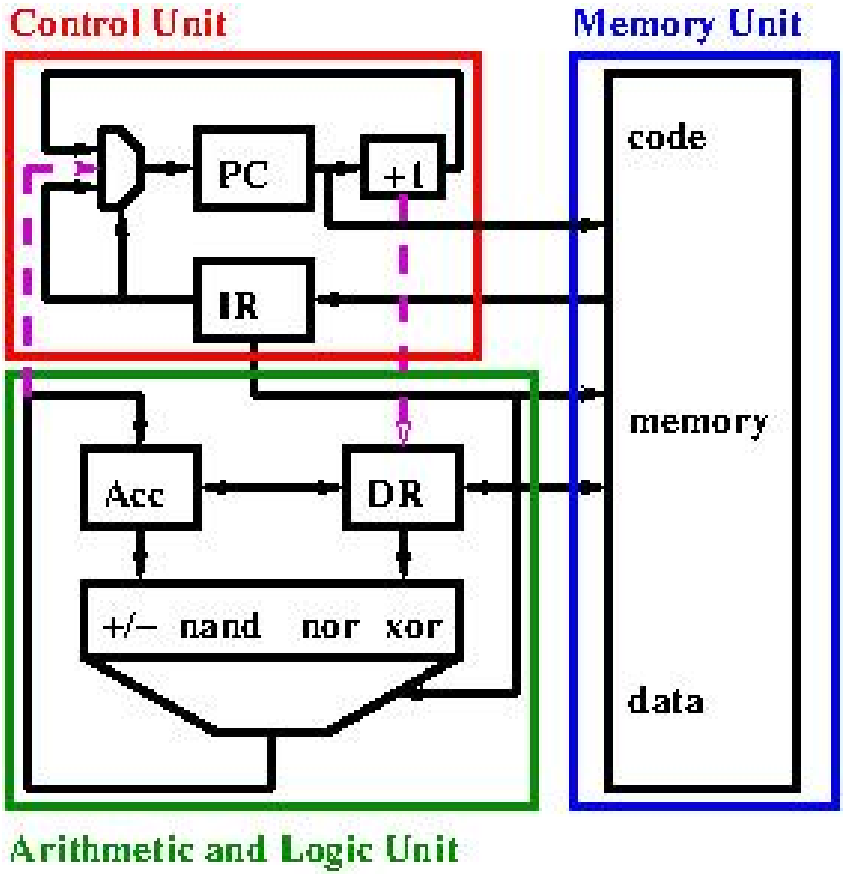
Section 1

Un processeur avec plusieurs PC

Subsection 1

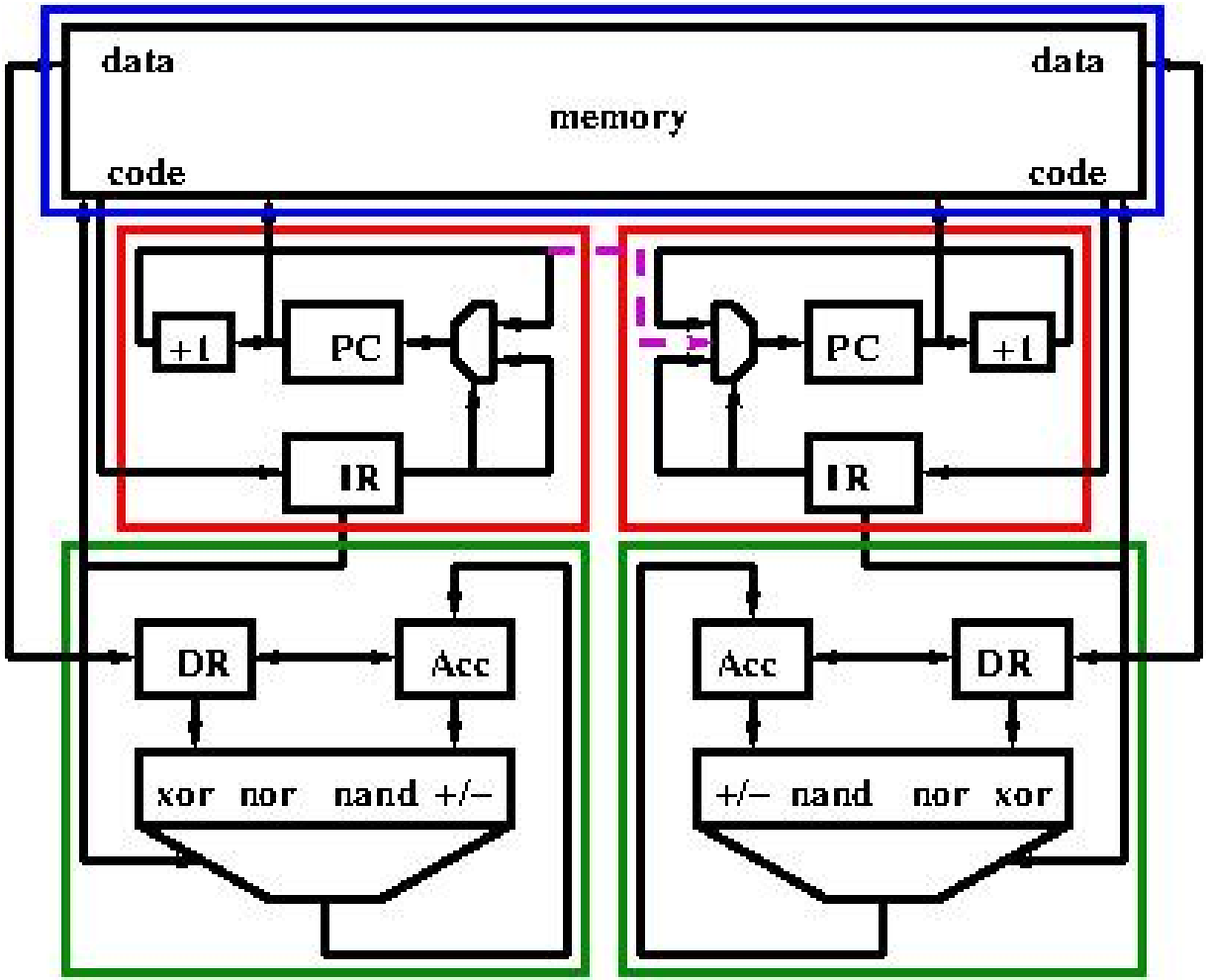
Von Neumann multi-cœur ou multi-thread

L'architecture de Von Neumann modifiée



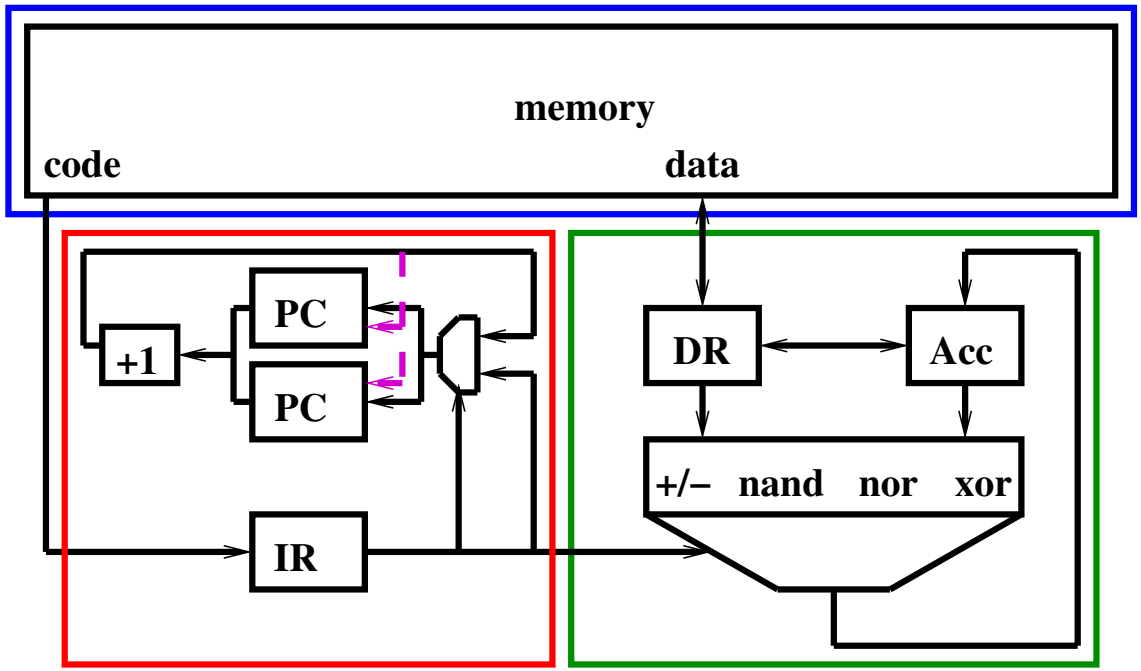
- Sauver le PC suivant en mémoire (empiler l'adresse de retour)
- Restaurer le PC de la mémoire (dépiler et effectuer le retour)

L'architecture de Von Neumann à 2 cœurs



- Envoyer le PC suivant au cœur suivant (envoyer l'adresse de retour)
- Exécuter la fonction et la continuation en parallèle

L'architecture de Von Neumann à 2 *threads*



- Envoyer le PC suivant au *thread* suivant (envoyer l'adresse de retour)
- Exécuter la fonction et la continuation en alternance

Subsection 2

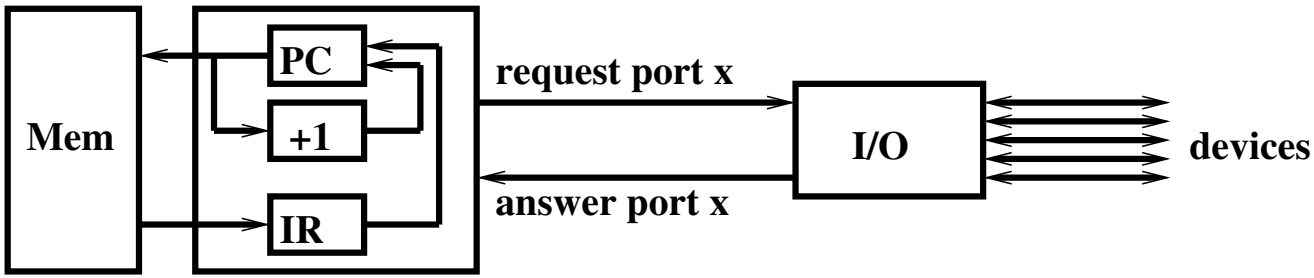
Les interruptions

L'invention des interruptions

- NBS DYSEAC, 1954, premières E/S sur interruption
- Electrologica X-1, 1959, vecteurs d'interruptions (E.W. Dijkstra)
- Dès 1960, la plupart des ordinateurs sont équipés d'interruptions
- "It was a great invention, but also a Box of Pandora" - E.W. Dijkstra

Avant 1954

in–order I/O management

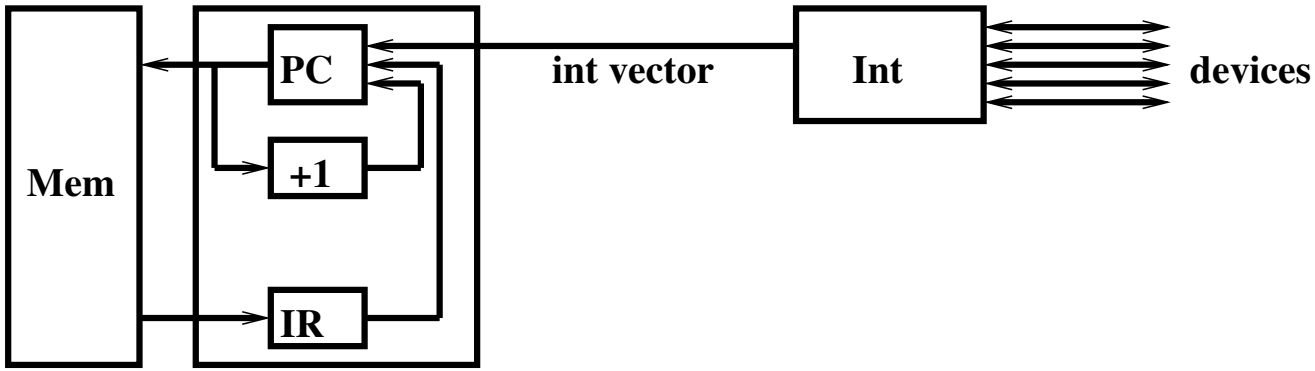


wait from I/O instruction issue until device completion

block PC update until I/O instruction completes

Après 1959

concurrent or interrupt driven I/O management



an I/O is a non blocking sub−program which switches control to an alternative job
two jobs: A and B, A running, B suspended
A enters an I/O sub−program which switches control to B
the I/O device communicates with the interrupt controler while B is running
the interrupt controler interrupts the CPU, i.e. sends an interrupt handler address as the next PC
B is interrupted
the interrupt handler saves the I/O result for the future benefit of A
the interrupt handler marks A as ready and returns to B
when B is suspended, it resumes A at the I/O instruction which is issued

- Ce qui est à l'intérieur de la boîte de Pandore, c'est le non-déterminisme

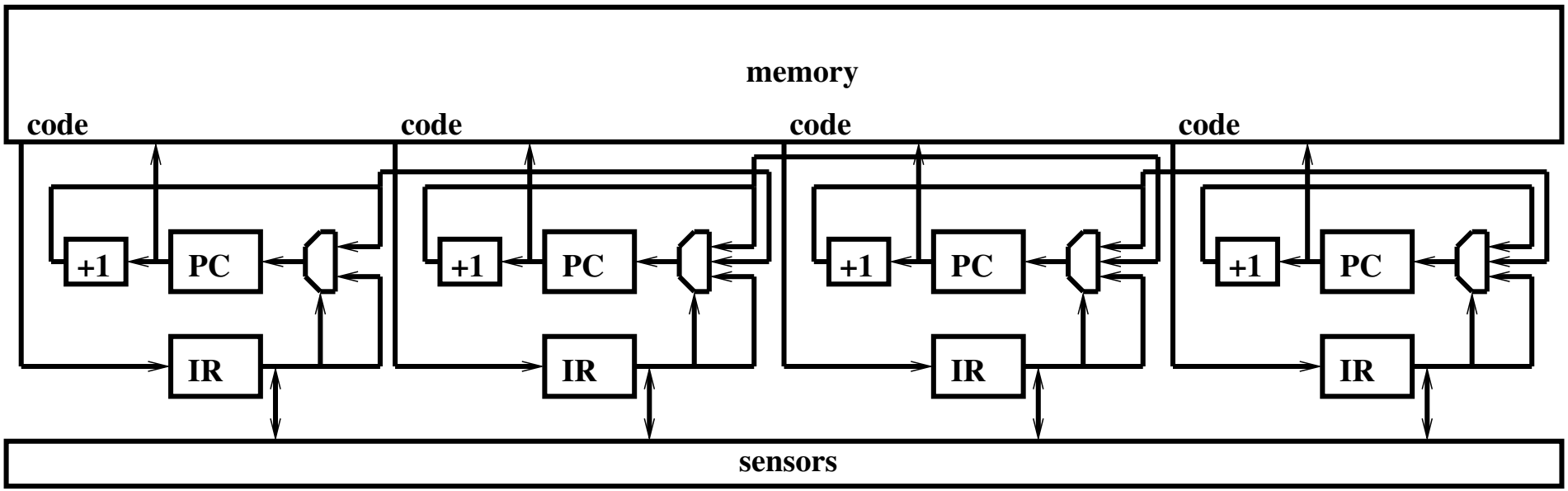
D'où vient le non-déterminisme ?

- La *variabilité des latences* n'implique pas le non-déterminisme
- Un accès mémoire a une latence variable mais cette latence peut être déterministe
- Un saut a une latence variable mais cette latence peut être déterministe
- L'*entrelacement de tâches* n'implique pas le non-déterminisme
- Deux calculs déterministes peuvent être entrelacés de façon déterministe sur un même cœur
- Le non-déterminisme ne peut être introduit que par une *perturbation externe* au processeur
- Par exemple, un accès externe peut modifier le contenu d'un cache ou du prédicteur de sauts, induisant une latence non déterministe d'un accès mémoire ou d'un saut
- Ce non-déterminisme de la latence peut inverser l'ordre de deux opérations, conduisant à un non-déterminisme du calcul
- La source du mal est l'interruption

En quoi le non-déterminisme du matériel est génant ?

- La reproductibilité du résultat nécessite de rendre le calcul déterministe (atomicité, synchronisation, ordonnancement)
- Le calcul n'est pas répétable (p.ex. un *bug* n'est pas répétable)
- Le temps de calcul n'est pas mesurable de façon exacte (perturbations non déterministes en cycles)
- La quantité de travail n'est pas mesurable de façon exacte (perturbations non déterministes en instructions)

Les interruptions auraient-elles été inventées pour une machine multi-threads et multi-cœurs ?



A job has to wait for 4 sensors reacting in a non−deterministic order

The job starts on the first core

It calls an input function for the first sensor and continues on the second core

The second core calls an input function for the second sensor and continues on the third core

The third core calls an input function for the third sensor and continues on the fourth core

The four cores wait for the four sensors inputs (blocking I/O)

Whatever the sensors reaction order, the run is driven according to a unique partial order

This partial order is based on the code sequential referential order and the producer to consumer dependencies

- On retrouve le déterminisme grâce à l'isolation

Subsection 3

Les caches

L'invention des mémoires hiérarchisées

- Ferranti Atlas, Manchester, 1962
- Mémoire à deux niveaux, contrôle matériel
- 96KB rapide, 576KB lente
- Remplacement asynchrone de pages basé sur le mécanisme d'interruption

Les caches auraient-ils été inventés pour une machine multi-threads et multi-cœurs ? (1)

- Une hiérarchie mémoire réduit la latence d'accès
- Un ensemble de *threads* sur un même cœur masque les latences, dont celles des accès mémoire, mais aussi celle des sauts
- L'efficacité de la hiérarchie mémoire vient du principe de localité observé sur des exécutions sur un cœur unique
- Quand on multiplie les cœurs, on distribue le calcul, donc on réduit la localité
- Les itérations d'une boucle sont réparties sur les cœurs, ce qui réduit la localité temporelle (on exécute moins souvent le même code)
- Cela réduit aussi la localité spatiale (à l'extrême, chaque cœur exécute une itération, donc p.ex. accède à un seul élément d'un vecteur au lieu d'une ligne d'éléments adjacents)
- En revanche, augmenter la répartition sur un plus grand nombre de *threads* et de cœurs, cela augmente la capacité à masquer les latences

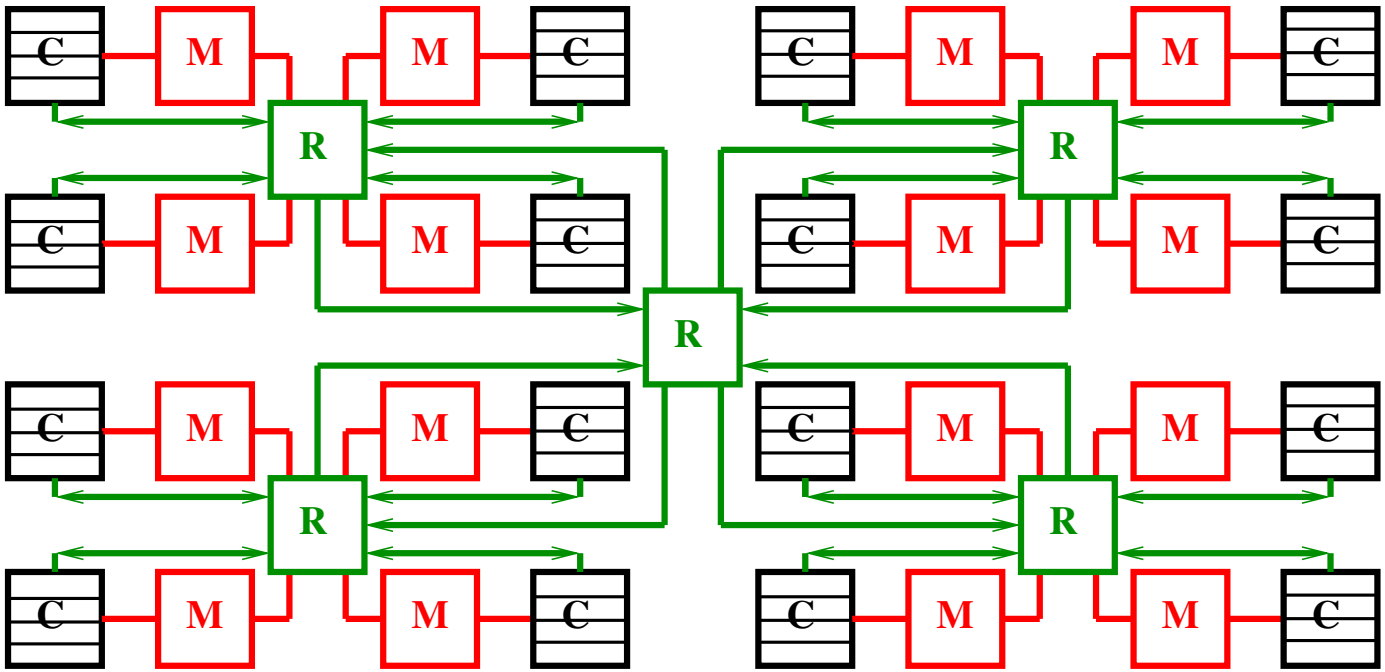
Les caches auraient-ils été inventés pour une machine multi-threads et multi-cœurs ? (2)

- Un cache contient une copie de ce qui est en mémoire
- Cette redondance consomme de la place et de l'énergie
- Dans un environnement multi-cœur, il faut maintenir les caches dans un état cohérent
- Cela nécessite de tenir à jour un répertoire identifiant le contenu de chaque ligne de chaque cache
- Cela nécessite aussi de déplacer une ligne vers le cache qui la modifie et d'invalider les copies
- Plus la localité est faible, plus le déplacement de ligne est inutilement coûteux
- Le matériel d'interconnexion des cœurs doit permettre à chaque cœur d'être un émetteur ou un récepteur de ligne

Les caches pour charger par morceaux

- Un cache permet de charger un extrait (du code, d'une donnée)
- La lecture du code s'effectue à chaque cycle, la lecture des données globales s'effectue plus rarement
- Pour lire le code, soit on dispose de beaucoup de *threads*, soit on charge plusieurs instructions à la fois
- Pour lire/écrire les données globales, le masquage par le *multithreading* suffit
- Il faut pouvoir amener le code par morceaux quand il est volumineux et qu'il ne tient pas en mémoire locale

Une mémoire distribuée sans cache de données



- Pas de cache de données, pas d'étiquettes, pas de répertoire, pas de duplication, cohérence de l'unicité
- Bus de la largeur d'un mot, chemin pipeliné
- Latence variable selon la distance entre le cœur accesseur et le cœur accédé
- Nécessité d'éviter la congestion des routeurs par une répartition des accès (temps et espace)

Le masquage des latences par le *multithreading*

- Au sein d'un cœur, on duplique le PC (p.ex. 4 PC par cœur)
- Les PC sont initialisés par bifurcation (c-à-d *fork*)
- A chaque cycle, on exécute une instruction d'un *thread*
- Avec 4 *threads* actifs effectuant chacun un accès en mémoire distribuée tous les 4 cycles, on masque une latence de 16 cycles
- (en moyenne, 34% d'accès mémoire (SPEC2017 ; max 49%), dont une partie significative est dans la pile, c-à-d en mémoire locale)

Subsection 4

La mémoire virtuelle paginée et le *multiprogramming*

L'invention de la mémoire virtuelle et du *multiprogramming*

- Mémoire virtuelle : Ferranti Atlas, Manchester, 1962
- Mémoire virtuelle paginée
- Traduction d'adresse virtuelle-réelle
- Remplacement asynchrone de pages basé sur le mécanisme d'interruption
- *Multiprogramming* : Leo III, 1961
- *Multiprogramming* et mémoire virtuelle : IBM 360/67, 1965

Les particularités de la mémoire virtuelle paginée

- Trois espaces : adressage (p.ex 64 bits), swap (p.ex. 64GB), physique (p.ex. 16GB)
- Adresse virtuelle statique basée en 0 : région (code, donnée, pile, tas), page virtuelle, *offset*
- Adresse virtuelle dynamique : ajout du numéro de processus
- Adresse physique : page physique, *offset*
- Traducteur matériel (MMU et TLB) + traducteur logiciel (TLB miss = parcours de tables en mémoire)
- Remplacement de page sur interruption (page fault)
- Le chargement de la mémoire principale à partir de la mémoire secondaire est déclenché par le référencement

La mémoire virtuelle aurait-elle été inventée pour une machine multi-threads et multi-cœurs ? (1)

- A cause de la pagination et du mécanisme de remplacement, la contiguïté physique des objets structurés disparaît
- Par exemple, un vecteur est un ensemble de pages éparpillées de la mémoire physique, pointées par une table de références de pages (PTE)
- Nécessité de centraliser le mécanisme de remplacement pour garantir l'unicité des pages virtuelles en mémoire physique
- Le déclenchement d'un remplacement à chaque référence absente n'est pas adapté aux *manycores*
- Le partage de la mémoire paginée par plusieurs applications rompt le principe d'isolation, ce qui est source d'un fort non-déterminisme temporel

La mémoire virtuelle aurait-elle été inventée pour une machine multi-threads et multi-cœurs ? (2)

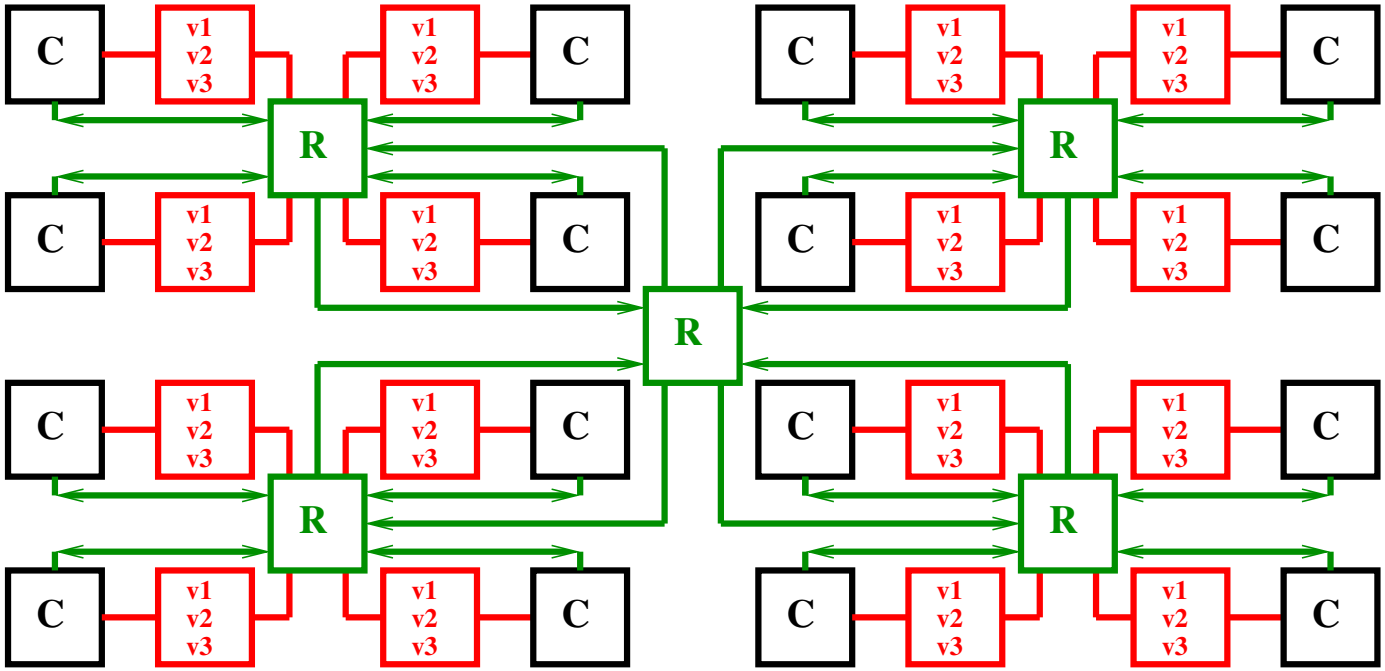
- La mémoire virtuelle sert à isoler l'espace mémoire de chaque application
- La pagination sert à partager l'espace physique entre plusieurs applications
- Dans une machine multi-cœurs à mémoire distribuée, l'isolation et le partage peuvent être obtenus plus simplement par partitionnement des cœurs (p.ex. une application A s'exécute sur la première moitié des cœurs et B sur la seconde moitié ; chaque application utilise la moitié de la mémoire distribuée)
- Dans ce cas, le découpage des objets structurés peut être aligné sur le découpage des cœurs consommateurs
- Amener un code ou une donnée en mémoire par morceaux : dupliquer le code sur les cœurs, distribuer les variables structurées dans les bancs de mémoire globale

La manipulation de vecteurs sur un processeur multi-cœur à mémoire distribuée

```
#include <det_omp.h>
#define LOG_N 20
#define N (1<<LOG_N)
int v1[N], v2[N], v3[N];
void main(){
 int i;
#pragma omp parallel for
 for (i=0; i<N; i++){v1[i]=i; v2[i]=i;}
#pragma omp parallel for
 for (i=0; i<N; i++) v3[i]=v1[i]+v2[i];
}
```

- Le compilateur distribue et entrelace les vecteurs selon le nombre de *threads* OpenMP
- Chaque vecteur est découpé en tranches de $N/OMP\_NUM\_THREADS$
- Le compilateur mappe une tranche par *thread* OpenMP (v1, v2, v3, filler)
- Il construit l'adressage pour tenir compte de l'entrelacement

La manipulation de vecteurs sur un processeur multi-cœur à mémoire distribuée



- v1, v2 et v3 sont entrelacés
- Chaque banc mémoire contient un morceau de $N/16$ octets de chacun des trois vecteurs
- Tous les accès sont locaux

Subsection 5

L'exécution en ordre partiel

L'algorithme de Tomasulo

- Développé pour la FPU de l'IBM 360/91 en 1967
- Basé sur le renommage de registres
- Permet l'exécution en ordre partiel
- Par exemple, une instruction dont les sources sont prêtes peut être exécutée avant une autre dont une source est en cours de calcul par une opération multi-cycles

Un algorithme mésestimé deux fois

- Redécouvert par Yale Patt en 1985 pour HPS (High Performance Substrate)
- "We believe that irregular parallelism in a program exists both locally and globally. Our mechanism exploits the local parallelism, but disregards global parallelism."
- Base des architectures *Out-of-Order* nées dans les années 90 et exploitant l'ILP proche
- Au début des années 2000, on laisse tomber la piste de l'ILP pour s'orienter vers le parallélisme de *threads*

L'ILP est caché

- L'ILP se déduit du graphe de dépendances LAE (registres et mémoire)
- Pour l'ILP local (c-à-d au sein d'une fonction), il faut ajouter les dépendances de contrôle (sauts)
- L'ILP d'une trace brute est inférieur à 2
- L'ILP d'une trace avec réordonnancement local (fenêtre de 100 à 1000 instructions) est inférieur à 6
- L'ILP est caché par des dépendances parasites
- Pour l'ILP distant, il faut éliminer les dépendances LAE sur la pile (mise à jour du pointeur, couples sauvegarde/restauration)
- Cela rend les fonctions indépendantes (sauf lien résultat) et leurs ILP locaux s'ajoutent, ce qui crée un ILP distant massif, qui croît avec la taille de la trace (plus le calcul est long, plus on peut alimenter de cœurs)

Comment capturer l'ILP distant et massif ?

- Pour l'ILP local : Tomasulo
- Pour l'ILP distant : changer de file de registres (c-à-d renommage brutal de tous les registres)
- Exécuter les fonctions sur des *threads* différents avec des files de registres distinctes
- Lier les fonctions (données, résultats) par copie de registres de *thread* à *thread*

L'exécution *Out-of-Order* aurait-elle été implémentée sur une machine multi-threads et multi-cœurs ?

- La disponibilité de plusieurs files de registres (une par *thread*) aurait certainement guidé les chercheurs vers l'exploitation de l'ILP distant
- La parallélisation par bifurcation aurait certainement été préférée à la pile (en conservant un espace privé servant à mémoriser des structures locales)

La capture de l'ILP de deux fonctions

```
#include <det_omp.h>
void f1(int i){...}
void f2(int i){...}
void main(){
 int i=18;
#pragma omp parallel sections
 {
  #pragma omp section
  {f1(i);}
  #pragma omp section
  {f2(i);}
 }
}
```

- Les fonctions sont indépendantes donc leurs ILP s'ajoutent
- Puisqu'elles sont exécutées par deux *threads* différents, il y a deux registres de deux files différentes pour i
- Une trace issue d'une exécution séquentielle aurait fait apparaître de fausses dépendances de contrôle et de pile entre $f1$ et $f2$

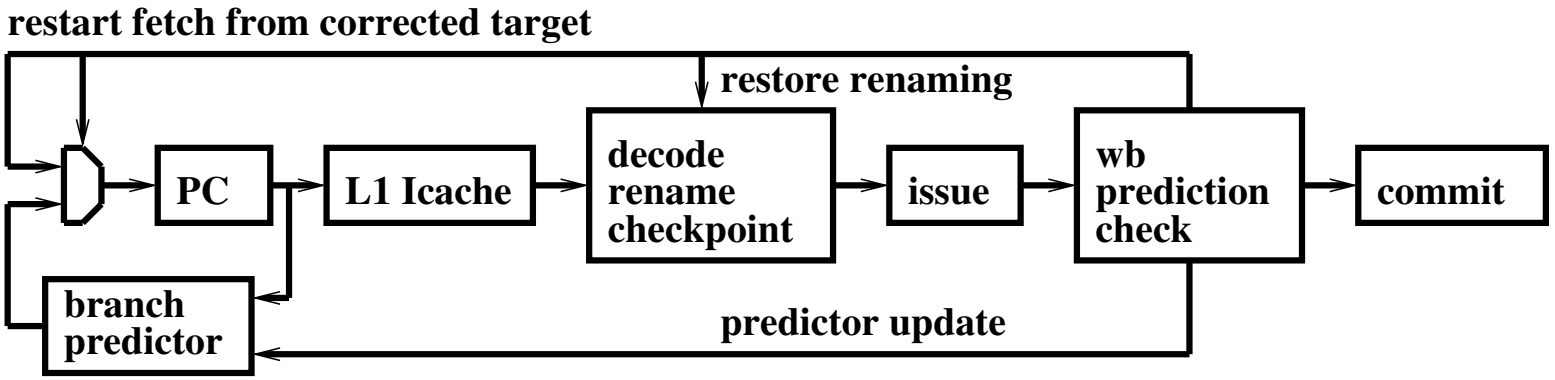
Subsection 6

L'exécution spéculative

La prédiction des sauts

- IBM Stretch, 1961 : prédiction statique non pris
- Lab S-1, 1977 : prédicteur deux bits
- Patt, 1991 : prédicteur à deux niveaux (comportements successifs d'un chemin menant à un saut)
- McFarling, 1993 : prédicteur hybride (sauts conditionnels, sauts indirects, retours de fonctions)
- Seznec, 1999 : gskew predictor Dec Alpha EV8
- Seznec, 2006 : TAGE, 3.3 MPKI
- Seznec, 2014 : TAGE-SC-L, champion du monde (3.3 MPKI pour 32Kbits, 2.4 MPKI pour 256Kbits, 1.8 MPKI nolimit)

Processeur spéculatif

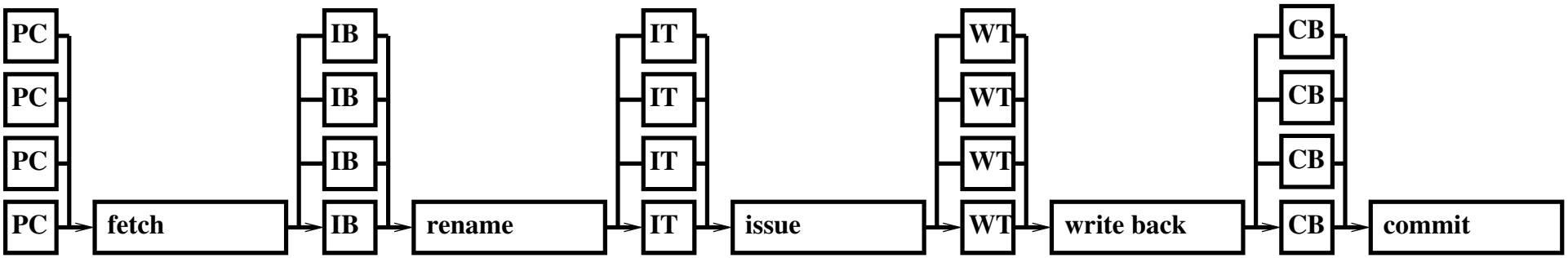


- L'étage de renommage ajoute des instructions spéculatives renommées à la liste des instructions en attente de leurs sources
- L'ensemble des instructions spéculatives renommées en attente forme la fenêtre spéculative
- La fenêtre spéculative est vidée à chaque fausse prédiction
- L'ordre partiel d'exécution est construit dynamiquement à partir des instructions de la fenêtre spéculative
- Remarque : les sauts de deux *threads* (c-à-d deux fonctions) sont en général indépendants

La taille de la fenêtre spéculative

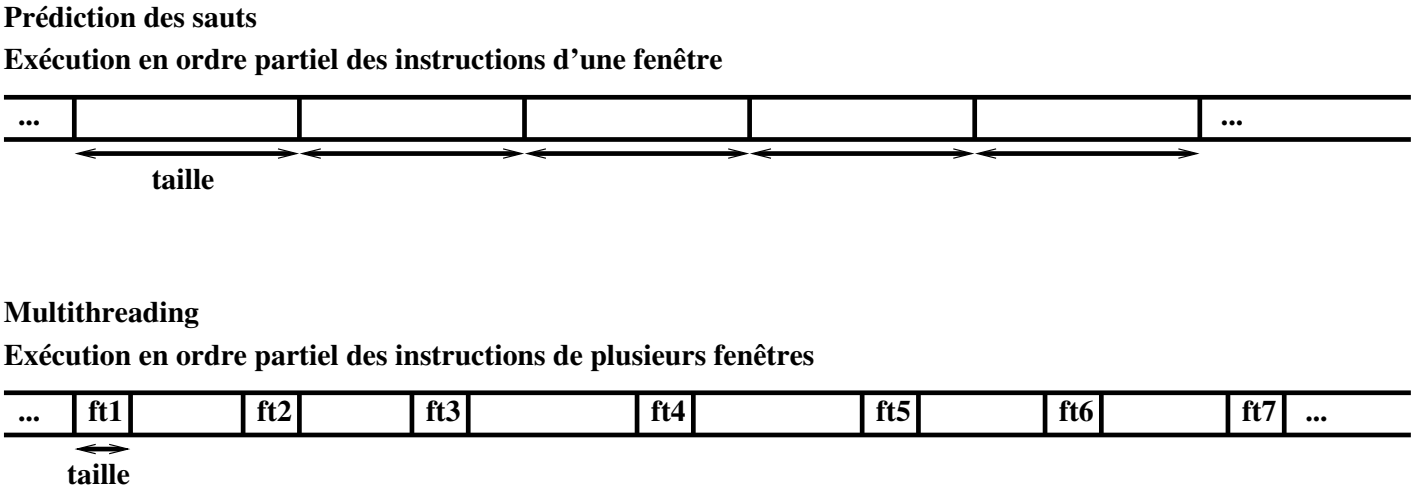
- Environ 20% de sauts
- 97% de bonnes prédictions (Patt 1991) = 6 MPKI = fenêtre de 167 instructions spéculatives
- 3.3 MPKI = fenêtre de 300 instructions spéculatives, budget 4KB par cœur
- 2.4 MPKI = fenêtre de 420 instructions spéculatives, budget 32KB par cœur
- 1.8 MPKI = fenêtre de 560 instructions spéculatives, budget illimité
- Processeurs actuels : fenêtre de 224 entrées (ROB broadwell, skylake, kabylake, cannonlake/coffeelake), agrandie pour SunnyCove/IceLake (256 ?)

Le *multithreading* : une alternative aux prédicteurs



- L'étage de *fetch* choisit un PC actif
- Le PC choisi est suspendu jusqu'à ce que le PC suivant ait été calculé (au moins un cycle)
- Il faut au moins deux *threads* pour remplir le pipeline
- Pas de spéculation (menace Spectre éliminée)
- Exceptions gérées comme des sauts (menace Meltdown éliminée)

Prédiction vs masquage



- Le prédicteur réduit la latence des sauts, le *multithreading* la masque
- Le prédicteur permet de créer des fenêtres de quelques centaines d'instructions (ILP = 6)
- Le *multithreading* permet de créer autant de petites fenêtres que de *threads*
- La complexité du réveil des instructions prêtes est proportionnel à la taille de la fenêtre
- 256 *threads* et fenêtre de 32 instructions = fenêtre cumulée de 8K instructions
- 256 *threads* = ILP cumulé supérieur à 512

L'exécution spéculative aurait-elle été implémentée sur une machine multi-threads et multi-cœurs ?

- La disponibilité de plusieurs *threads* aurait certainement guidé les chercheurs vers le masquage de la latence des sauts
- Les prédicteurs n'auraient sans doute pas été nécessaires

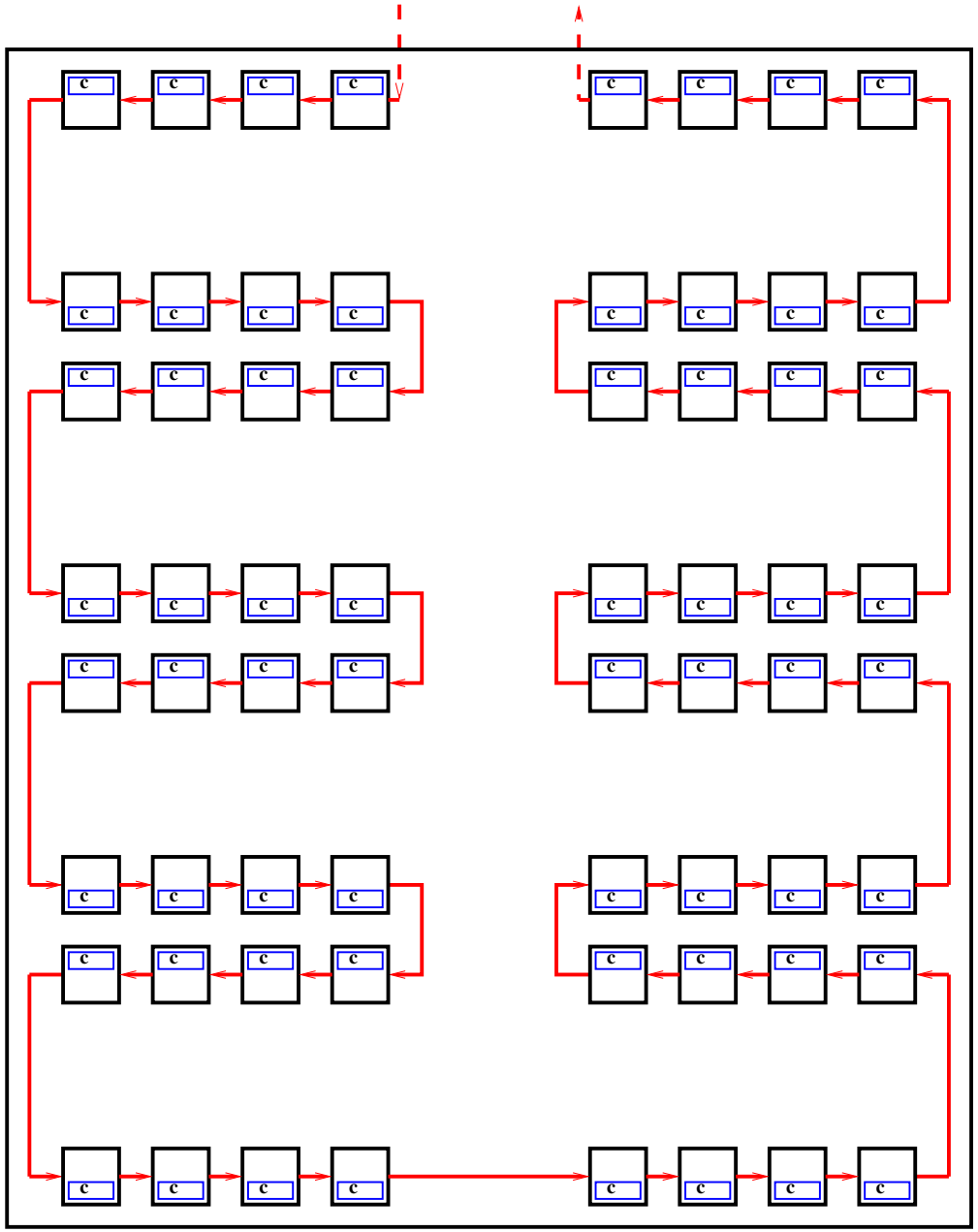
Section 2

Le processeur LBP (*The Little Big Processor*)

Un processeur à beaucoup de cœurs parallélisant

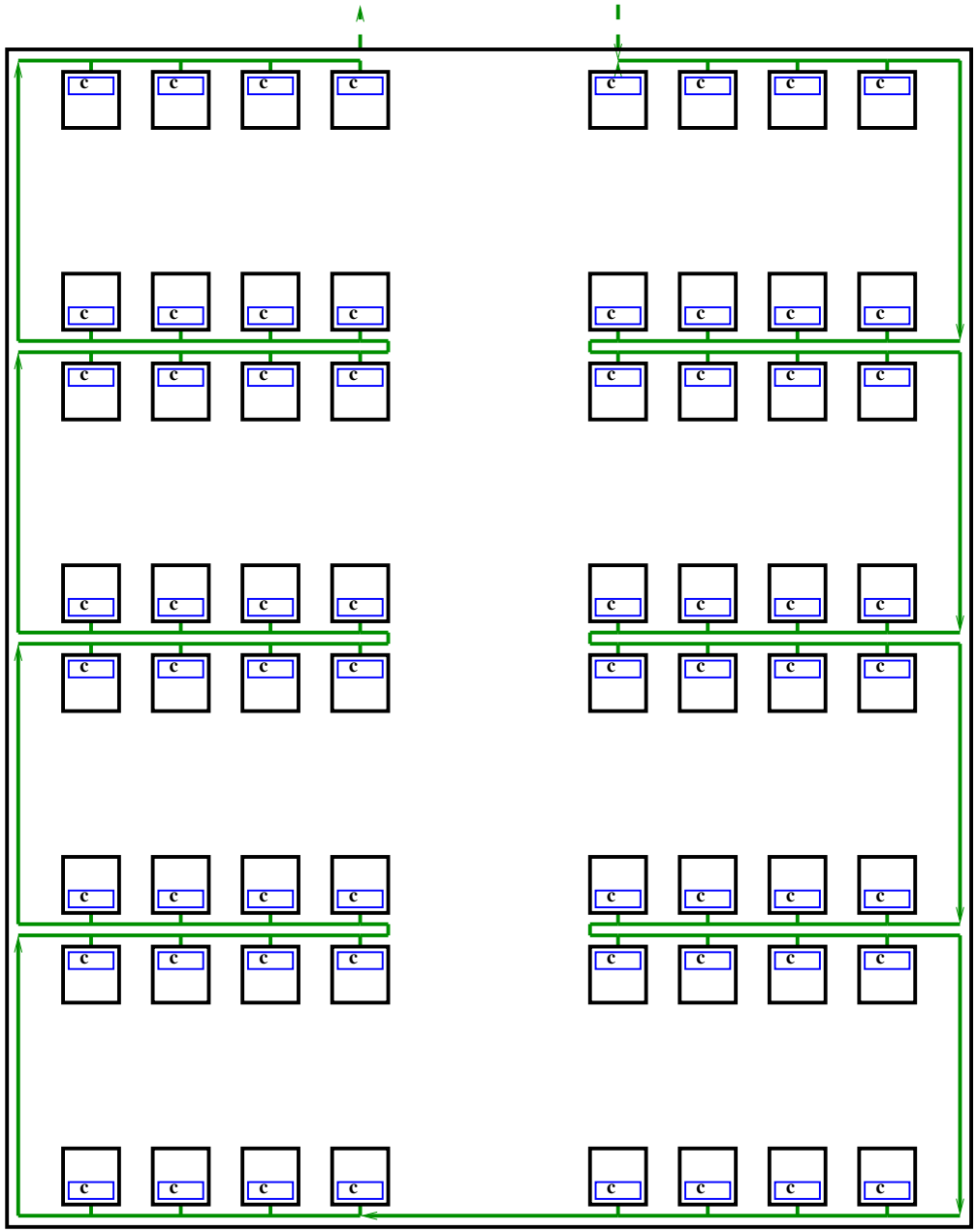
- LBP : The Little Big Processor (ARM Big Little ; Little big man)
- Un processeur pour exploiter le parallélisme d'instructions (ILP) : parallélisation par bifurcation matérielle plutôt que par l'OS
- Exécution des *threads* en ordre partiel plutôt que concurrente
- Un ensemble extensible de cœurs (LBP 4, 16, 64, ... cœurs)
- Chaque cœur peut héberger jusqu'à 4 *harts* (*hardware thread*)
- Une mémoire locale privée pour chaque cœur (code et données locales)
- Une mémoire globale distribuée et partagée (données globales) avec un banc par cœur
- Un ISA RISC-V étendu avec X_PAR (instructions de parallélisation)

LBP à 64 cœurs : paralléliser les *threads*



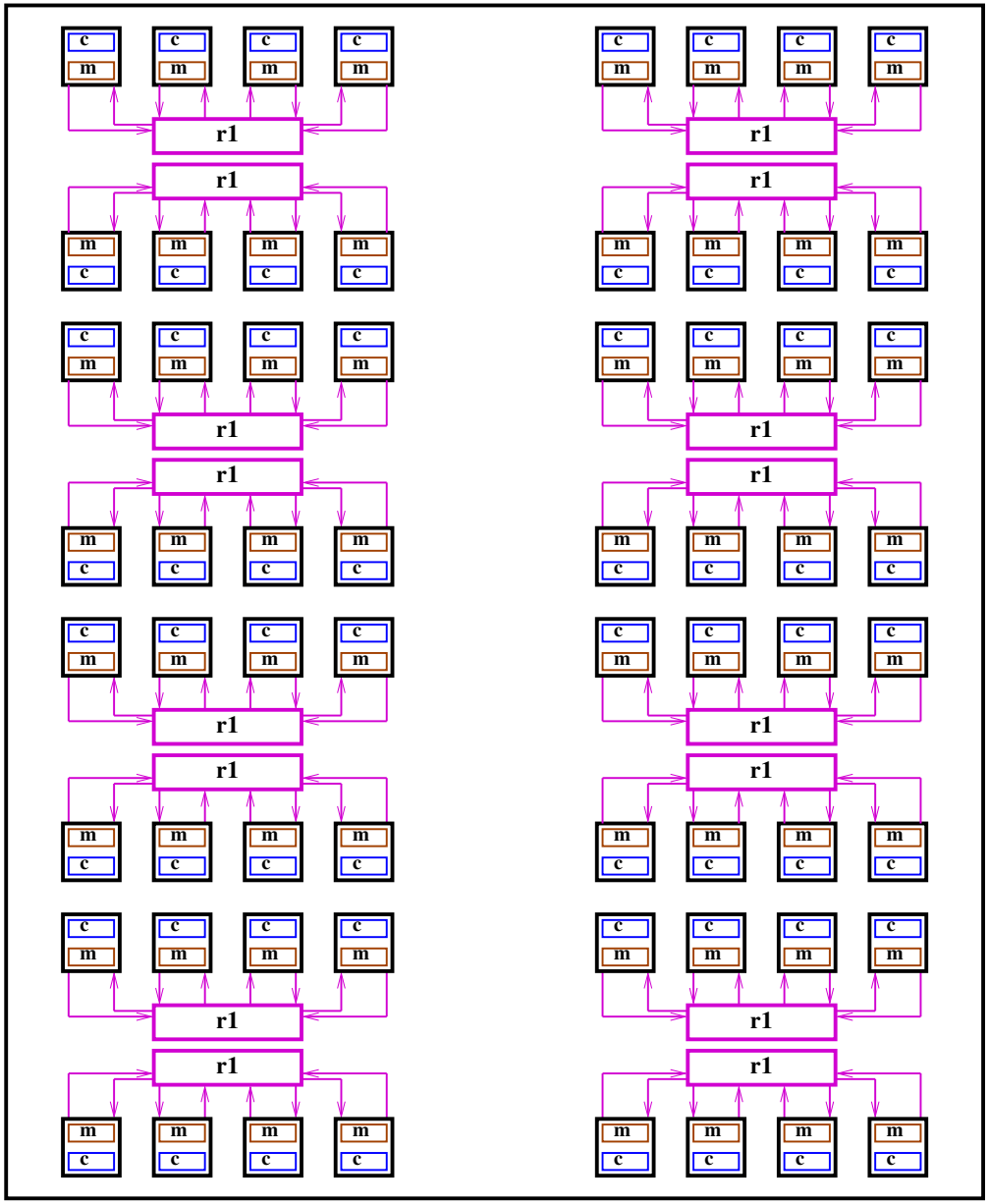
- Topologie de ligne (extensible par connexion de composants)
- Le lien rouge permet à un cœur (c) de lancer un *thread* sur le cœur successeur et de lui transmettre des registres (allocation d'un *hart*)
- Bus de connexion directe entre deux cœurs adjacents, largeur d'un mot

LBP à 64 cœurs : joindre les *threads*



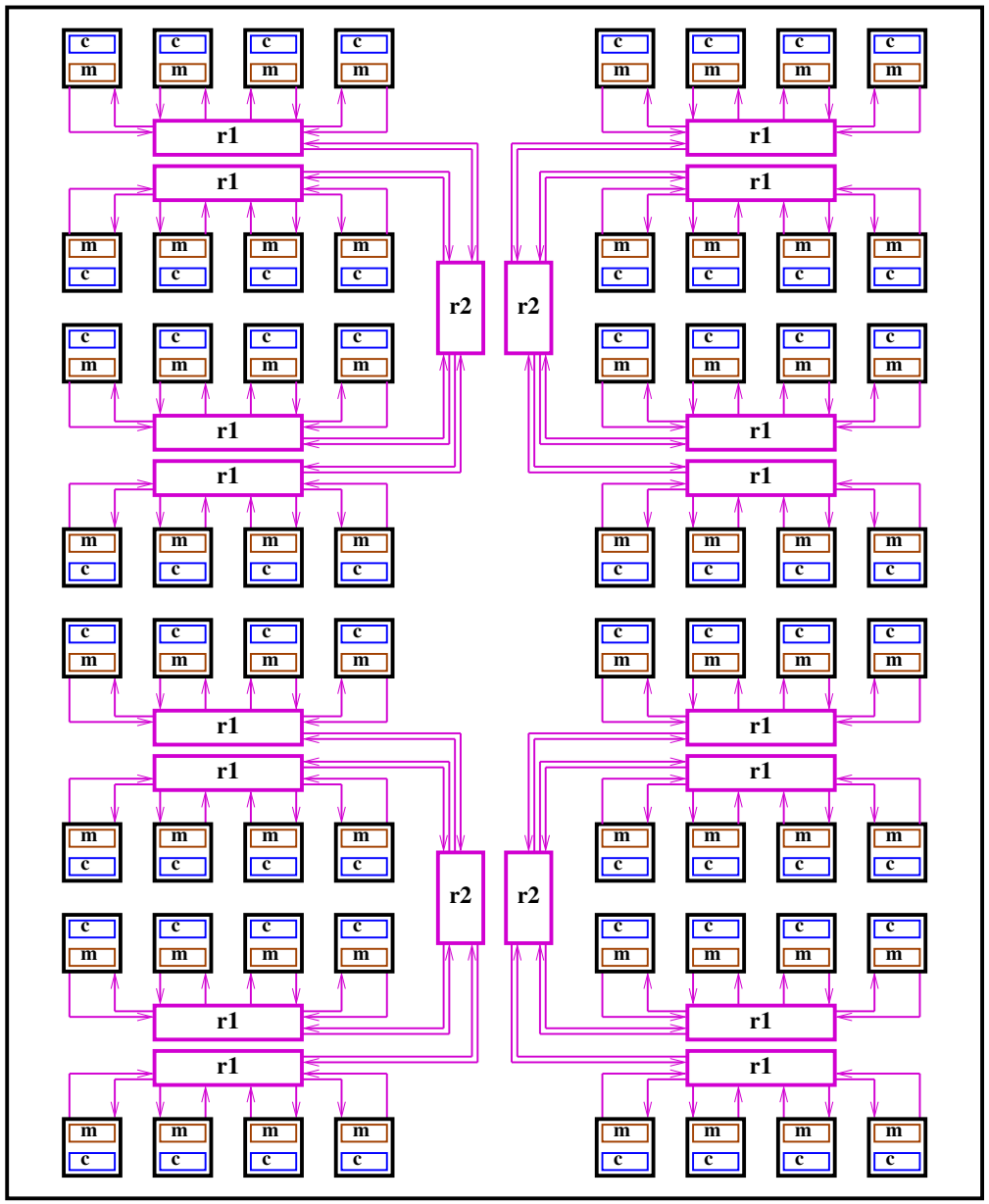
- Le lien vert permet de transmettre un registre ou une adresse de continuation à un cœur prédécesseur (terminaison du dernier *hart* d'un *team* de *threads* parallèles)
- Résultat d'une fonction
- Adresse de jonction des *threads* parallèles (continuation séquentielle)
- Bus de largeur un mot, *latch* entre deux nœuds adjacents

LBP à 64 cœurs : la mémoire globale distribuée



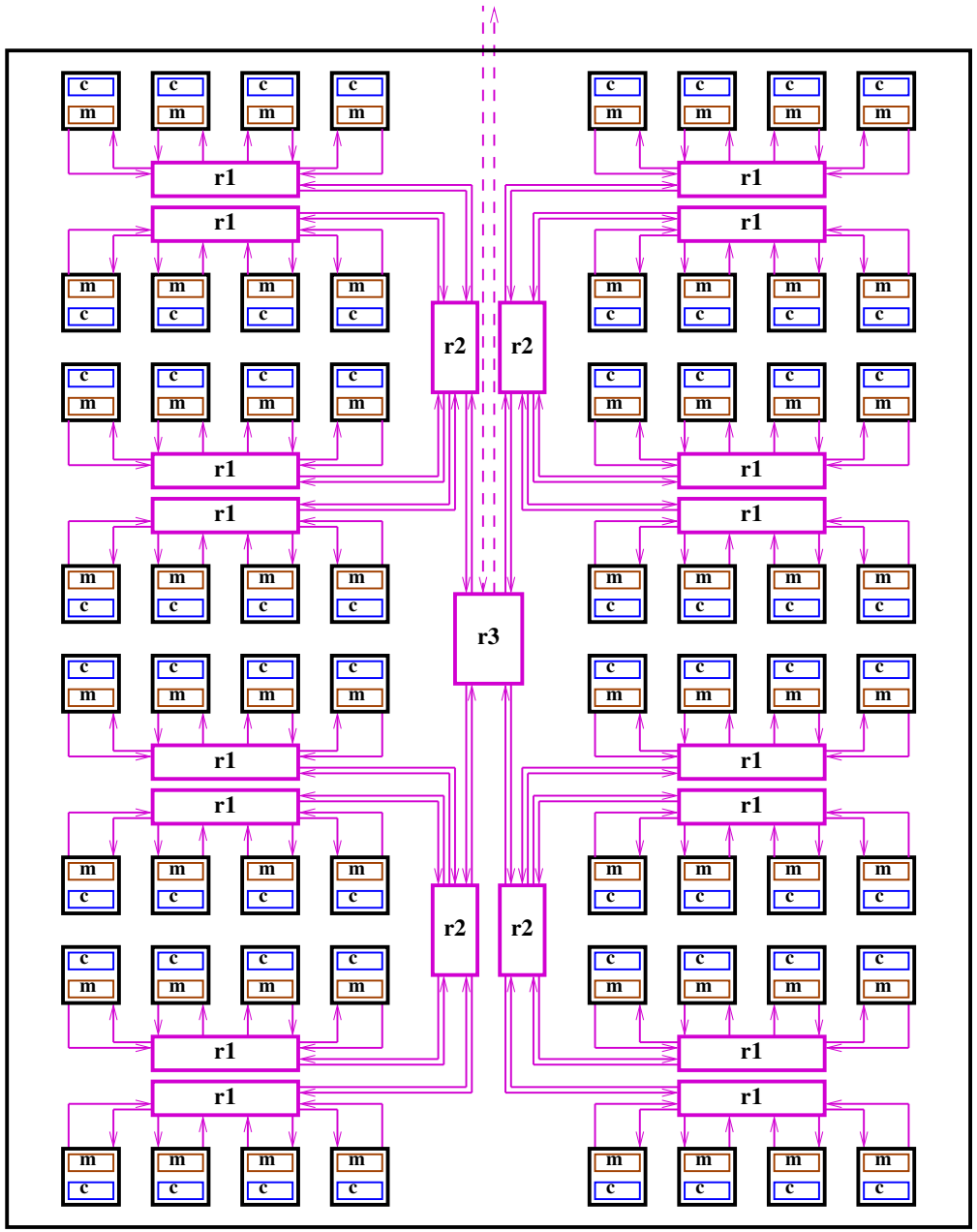
- Le lien magenta permet à un cœur d'accéder à la mémoire (m) d'un autre cœur du même groupe de 4 via un routeur de premier niveau (r1)
- Chaque banc mémoire est à double accès : accès privé du cœur local, accès du routeur de groupe r1
- Chaque cœur a 4 liens avec son routeur r1 pour acheminer 4 requêtes de 4 *harts*
- Chaque routeur r1 a 16 liens entrants et 16 liens sortants
- L'accès distant est pipeliné

LBP à 64 cœurs : la mémoire globale distribuée



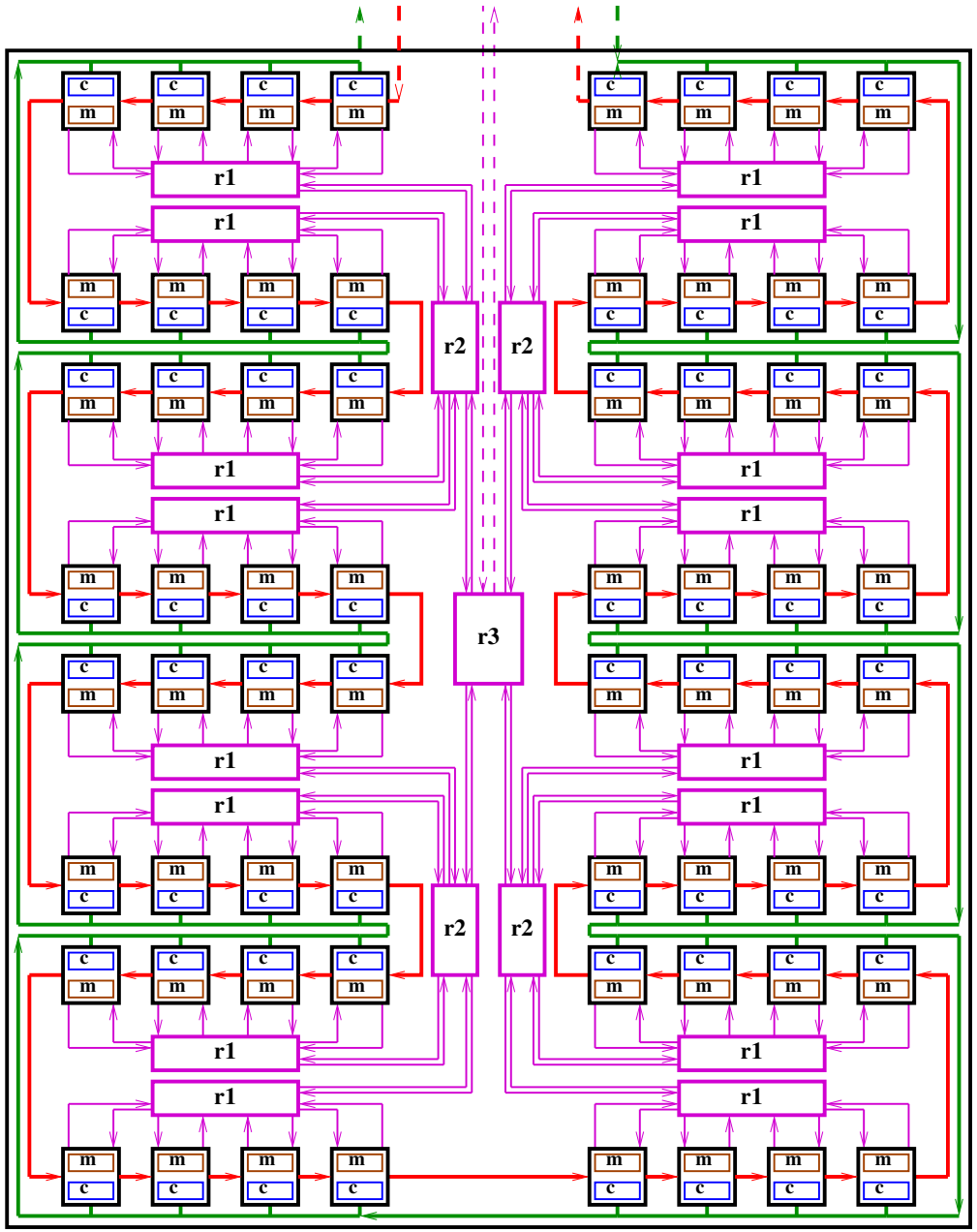
- Un routeur r2 relie entre eux 4 routeurs r1 (4 liens par routeur r1)
- Chaque cœur peut accéder aux 16 bancs mémoires voisins (chemin cœur, r1, r2, r1, banc mémoire, r1, r2, r1, cœur)
- Le chemin est pipeliné

LBP à 64 cœurs : la mémoire globale distribuée



- Un routeur r3 relie entre eux 4 routeurs r2 (4 liens par routeur r2)
- Chaque cœur peut accéder aux 64 bancs mémoires voisins (chemin cœur, r1, r2, r3, r2, r1, banc mémoire, r1, r2, r3, r2, r1, cœur)
- Le chemin est pipeliné
- Le routeur r3 peut être relié à un routeur r4 par extension externe au composant LBP

LBP à 64 cœurs

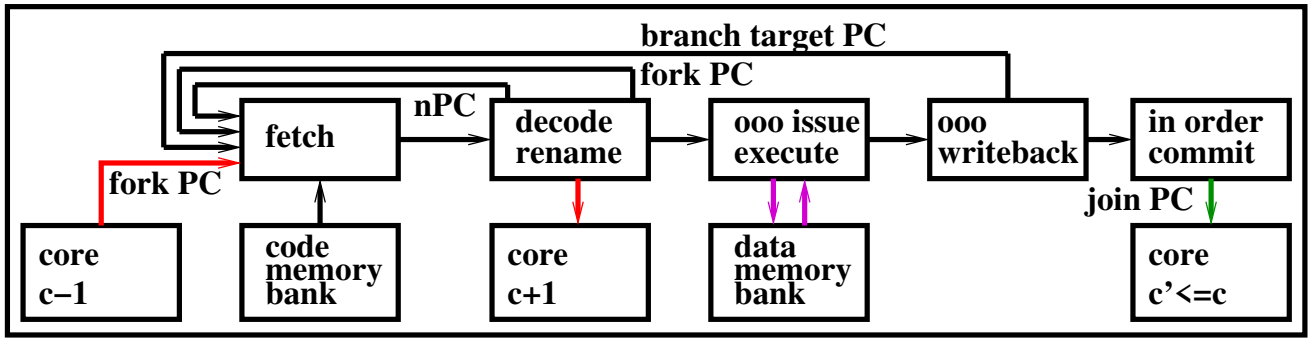


- Le cœur initial déploie un *team* de *threads* parallèles en suivant le lien rouge
- Il occupe d'abord les 4 *harts* locaux avant de s'étendre sur le cœur suivant
- Les *threads* du *team* se terminent en ordre (mais s'exécutent en parallèle)
- Chacun transmet un signal de terminaison à son successeur (lien interne ou lien rouge)
- Le dernier du *team* transmet au premier l'adresse de continuation par le lien vert (sorte de retour distant)

Les cœurs de LBP

- Des cœurs les plus simples possibles : pas de prédicteur, pas de cache, pas d'unités vectorielles, pas plus d'une opération par cycle
- Le plus de cœurs possible sur le composant
- Des cœurs à 4 *harts* (on masque les latences par le *multithreading*)
- Des cœurs scalaires à exécution *out-of-order* (pipeline à 5 étages)

Le pipeline à 5 étages



- 5 machines concurrentes : Fetch, Rename, Execute, Writeback, Commit
- Fetch : lit une instruction d'un *hart* actif
- Rename : décode et renomme une instruction d'un *hart*
- Execute : effectue l'opération d'une instruction prête
- Writeback : écrit le résultat d'une instruction terminée pour chaque *hart*
- Commit : libère les registres de renommages, termine les *harts* et transmet la continuation séquentielle (fin d'une structure *omp parallel*)

LBP : un processeur déterministe

- Entrelacements gérés par priorités tournantes (mécanisme déterministe)
- Aucun entrelacement externe (pas d'interruption)
- E/S par attente d'exécution (instruction bloquée dans l'étage d'exécution)
- Une entrée survenant à un moment variable ne perturbe pas l'ordre partiel d'exécution : tout ce qui en dépend est exécuté après.
- Une application isolée de l'extérieur a une exécution déterministe cycle à cycle
- Une application dépendante d'événements externes non datés a une exécution en ordre partiel déterministe (déterminisme temporel)

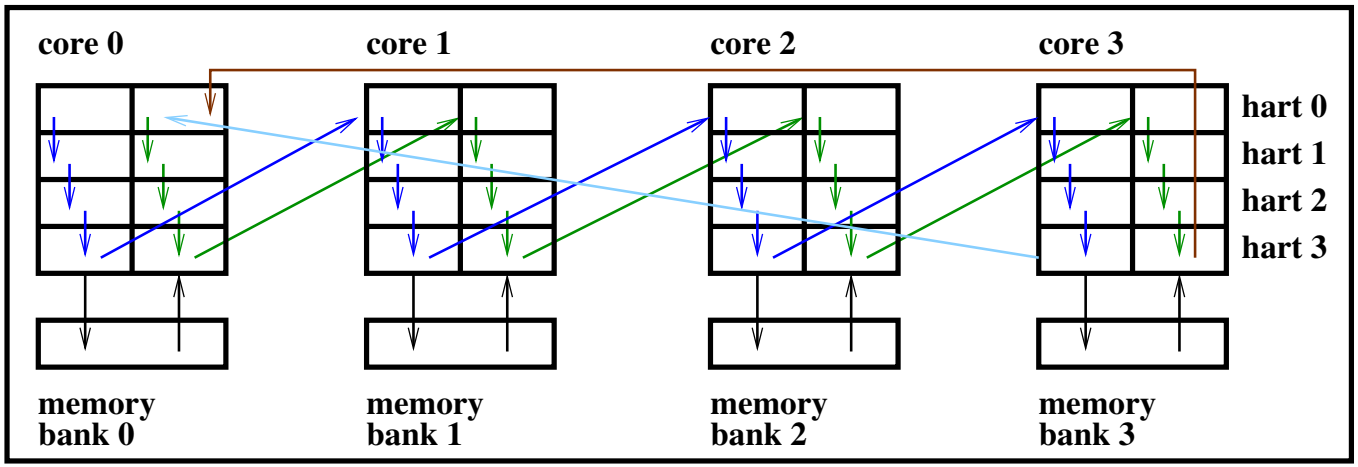
Deterministic OpenMP

- Deterministic OpenMP est l'API OpenMP standard associée à un nouveau *runtime* ("det_omp.h" remplace "omp.h")
- Le *runtime* de Deterministic OpenMP lie un code source aux mécanismes du processeur LBP permettant de paralléliser une exécution sur ses ressources de calculs interconnectées
- Deterministic OpenMP est aussi basé sur le modèle *fork/join* : une exécution Deterministic OpenMP est une succession de bifurcations/jonctions entièrement gérées par le matériel
- Chaque bifurcation crée un *team* de *threads* matériels qui s'exécutent en ordre partiel des dépendances producteur/consommateur et se rejoignent à leur terminaison

Une exécution synchronisée et alignée

```
#include <det_omp.h>
#define NUM_HART 16
#define SIZE (1<<16)
int v[SIZE];
void thread_set(int v[], int t){
 /*init chunk t of v*/
}
void thread_get(int v[], int t){
 /*use chunk t of v*/
}
```

```
void main(){
 int t;
 omp_set_num_threads(NUM_HART);
#pragma omp parallel for
 for (t=0; t<NUM_HART; t++)
  thread_set(v,t);
#pragma omp parallel for
 for (t=0; t<NUM_HART; t++)
  thread_get(v,t);
 /*sequential part*/
}
```



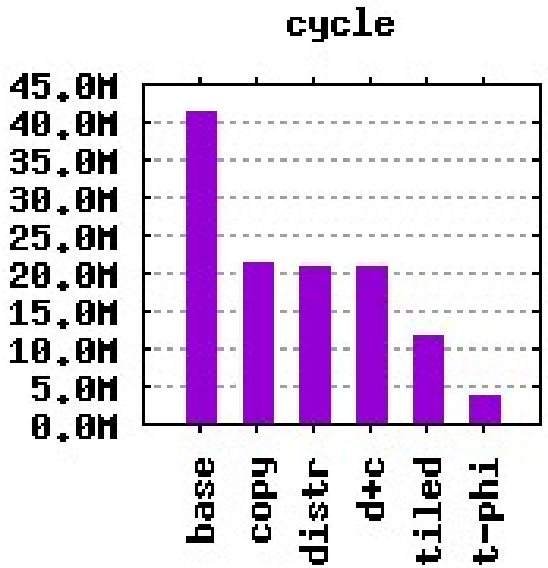
Une multiplication de matrices entières (base)

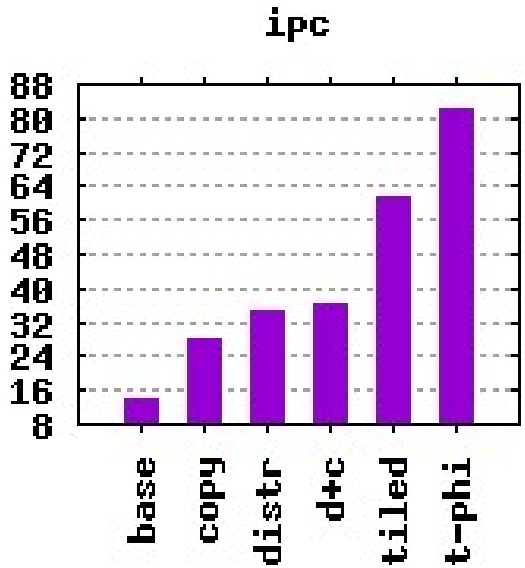
```
#include <stdio.h>
#include <det_omp.h>
#define LINE_X      16
#define COLUMN_X    8
#define LINE_Y      COLUMN_X
#define COLUMN_Y    16
#define LINE_Z      LINE_X
#define COLUMN_Z    COLUMN_Y
#define NUM_HART    16
int X[LINE_X*COLUMN_X]={[0...LINE_X*COLUMN_X-1]=1};
int Y[LINE_Y*COLUMN_Y]={[0...LINE_Y*COLUMN_Y-1]=1};
int Z[LINE_Z*COLUMN_Z];
```

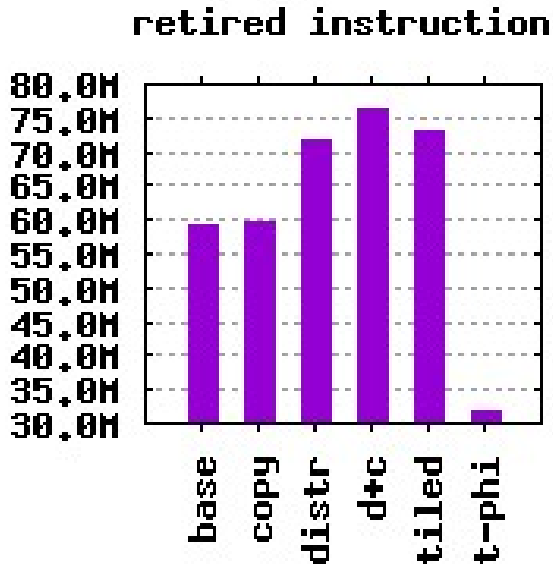
```
void thread(int t){
 int i, j, k, l, tmp;
 for (l=0, i=t*LINE_Z/NUM_HART; l<LINE_Z/NUM_HART; l++, i++)
  for (j=0; j<COLUMN_Z; j++)
   tmp=0;
   for (k=0; k<COLUMN_X; k++)
    tmp+=*(X+(i*COLUMN_X+k)) * *(Y+(k*COLUMN_Y+j));
   *(Z+(i*COLUMN_Z+j))=tmp;
  }
}
void main(){
 int t;
 omp_set_num_threads(NUM_HART);
#pragma omp parallel for
 for (t=0; t<NUM_HART; t++) thread(t);
}
```

- 5 algorithmes
- "base"
- "copy" : copie une ligne de X en mémoire locale
- "distr" : distribue Y uniformément (entrelacement de X, Y et Z)
- "c+d" : copie X et distribue Y
- "tiled" : calcul partitionné

Résultats pour LBP à 64 cœurs







LBP aujourd'hui

- Code HLS Xilinx simulé (processeurs 4, 16 et 64 cœurs + routeurs mémoire)
- Testé sur carte Pynq (version à 8 cœurs sans routeur)
- Version à 64 cœurs en cours d'implantation sur ZCU102
- ISA RISC-V 32 bits entier avec multiplieur et diviseur (pas de flottant)
- Construction manuelle du runtime "det_omp.h"
- Démarrage d'une thèse en octobre pour construire l'IP LBP à 64 cœurs et son environnement de développement Deterministic OpenMP

Section 3

Conclusion

Remarque sur la recherche

Il y a 4 sortes de chercheurs (en ordre décroissant de difficulté à publier) :

- Les pionniers (inventeurs d'une théorie ; p.ex. Turing/Eckert-Mauchly)
- Les réformateurs (modification profonde d'une théorie existante ; p.ex. Hennessy-Patterson, Tomasulo)
- Les perfectionneurs (amélioration substantielle d'une théorie existante ; p.ex. Patt, Seznec)
- Les évaluateurs (évaluation d'une théorie existante ; p.ex. comparaison CISC/RISC par mesures sur SPEC, détection de menaces sur l'exécution spéculative)

Tout est utile un jour ou l'autre