Architectures avancées

Parallélisme d'instructions : processeurs superscalaires et VLIW

Alain MÉRIGOT

Université Paris Saclay

Parallélisme d'instructions: superscalaires et VLIW

Le parallélisme d'instructions permet d'exécuter plusieurs instructions simultanément, et donc de réduire le nombre de cycles par instruction (CPI)

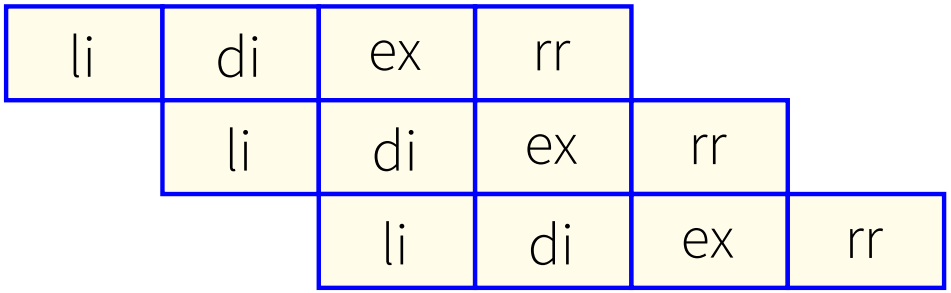
Permet d'utiliser simultanément plusieurs unités fonctionnelles (par exemple l'ALU entière et le multiplieur flottant).

Deux grandes familles :

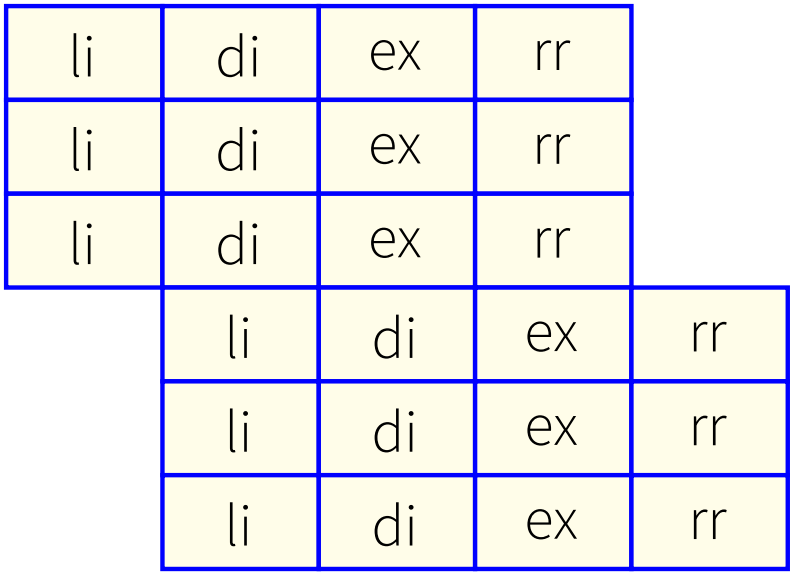
processeurs *superscalaires* La parallélisation est effectuée de manière *dynamique* par le *matériel*

processeurs *VLIW* (*very long instruction word*) La parallélisation est effectuée *statiquement* par le *compilateur*.

Pipeline



Parallélisme d'instructions de degré n (ici 3) : n instructions/cycle



Parallélisme d'instructions: superscalaires et VLIW (cont.)

L'exécution des instructions peut être :

ordonnée (*in-order superscalar*) : exécution par groupe d'instruction respectant l'ordre du compilateur

L'utilisation du parallélisme n'est pas optimal :

- une suspension (due à une dépendance) impacte tout le groupe
- aléa structurels (p. ex. 2 × FP successives)

Premiers pentium, certains processeurs embarqués. Parallélisme max 2-3

non ordonnée (*out-of-order superscalar*) : modification dynamique de l'ordre des instructions pour optimiser le parallélisme en respectant les dépendances.

Processeurs pour serveurs ou de bureau actuels. Parallélisme max 3-6

statiquement ordonné (VLIW *very long instruction word*) : le compilateur extrait le parallélisme du code et organise le programme en groupes d'instructions parallèle.

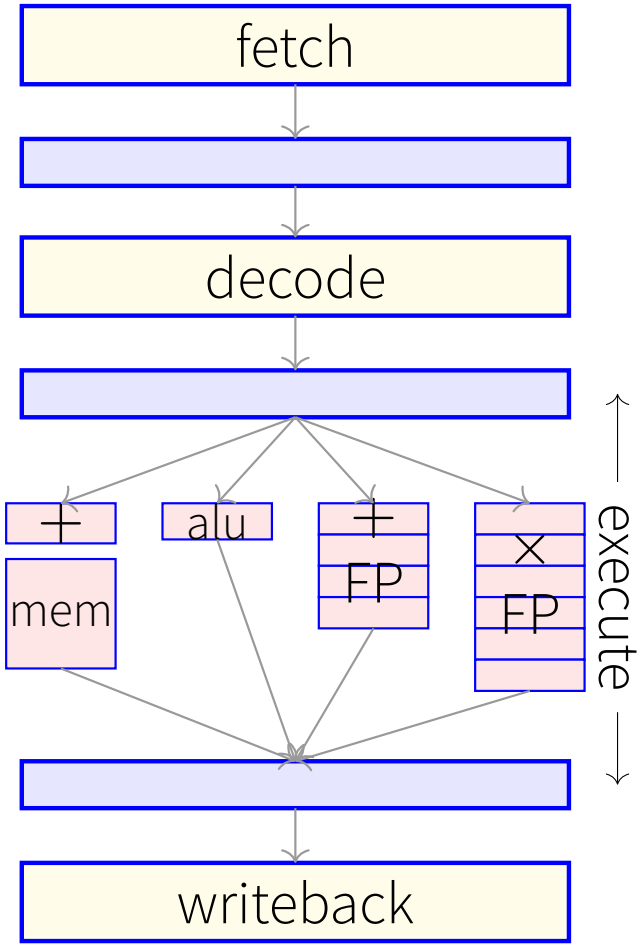
Moins coûteux, mais manque de robustesse lors d'événements dynamiques (défaut de cache).

Code lié aux détails architecturaux (latence des opérateurs)

DSP, processeurs pour serveur. Parallélisme max ≈ 8

Processeurs superscalaires

Les processeurs superscalaires sont une extension du traitement pipeline.

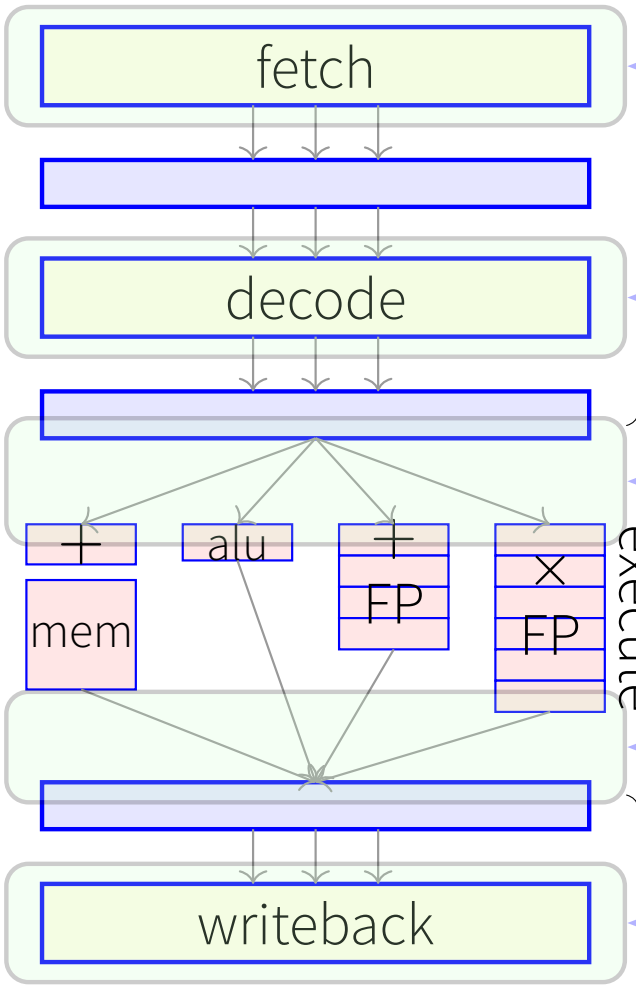


Pipeline RISC classique.

L'accès à la mémoire a été intégré à l'étage d'exécution.
Celui ci peut avoir plusieurs phases pour mettre en oeuvre des opérations complexes (calcul flottant).

Processeurs superscalaires (cont.)

Les processeurs superscalaires sont une extension du traitement pipeline.



Acquisition de plusieurs instructions

Décodage de plusieurs instructions et prise en compte des dépendances

Ajout d'un étage pour répartir les traitements entre unités fontionnelles

Ajout d'un étage pour réordonner les résultats.

Ecriture de plusieurs résultats

Processeurs superscalaires (cont.)

Principaux problèmes :

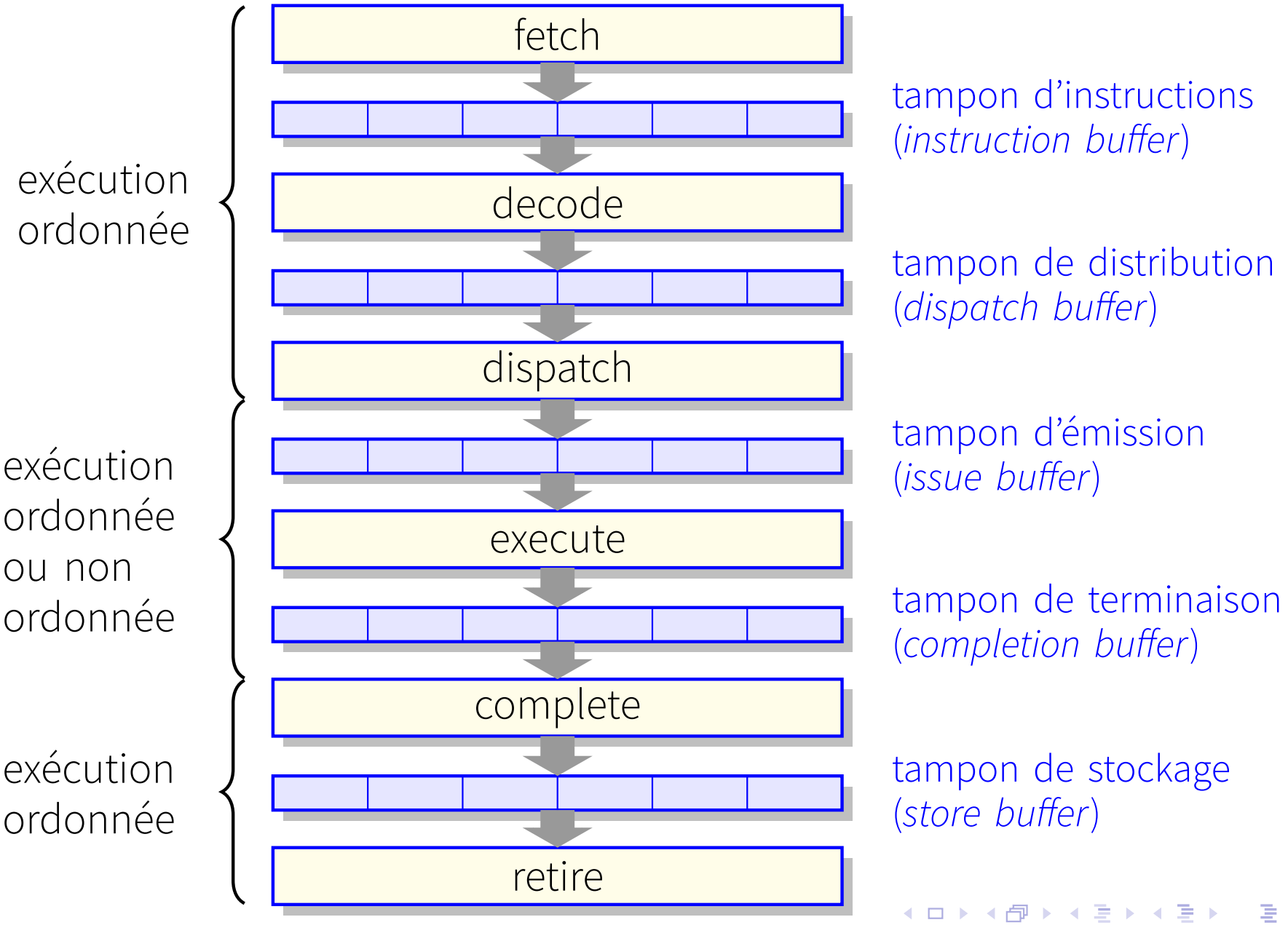
- comment gérer les dépendances entre instructions
  - tableaux de marques
  - mécanisme de Tomasulo
- lecture des instructions en parallèle
- quelles instructions sont parallélisables ?
- exécution ordonnée ou non-ordonnée des instructions ?
  - tampon de réordonnancement
- cas des branchements

Processeurs superscalaires (cont.)

L'exécution superscalaire augmente le coût matériel

- bus multiples, accès simultané à plusieurs registres
- augmentation du nombre d'unités fonctionnelles (plusieurs unités flottantes et entières)
- compléxité de la prise en compte des latences différentes des opérateurs et des mécanismes de court circuit (*forwarding*)
- augmentation du débit d'instructions et de données
- complexité de la prise en compte des branchements

Principales étapes d'exécution superscalaire



Principales étapes d'exécution superscalaire (cont.)

L'acquisition des instructions se fait par groupe
Les instructions acquises sont décodées simultanément

L'étape de *dispatch* va répartir les instructions vers les unités fonctionnelles
- en fonction des unités fonctionnelles existantes
  typ : 1 brcht., 1 chargt/rangt, 1 + flot., 1× flot., 1 ou 2 op entières...
- en fonction des dépendances entre instructions

L'étape d'exécution peut être :
- ordonnée : on exécute *toutes* les instructions d'un groupe et on passe au groupe suivant (superscalaire *statique*)
- non ordonnée : l'exécution dépend de la *disponibilité* des opérandes et des unités fonctionnelles. (superscalaire *dynamique*)

La phase de *completion* va assurer l'ordonnancement correct des écritures même en cas d'exécution non ordonnée (tampon de réordonnancement ou *reorder buffer* (ROB))

Principales étapes d'exécution superscalaire (cont.)

superscalaire statique (exécution ordonnée des instructions)

Parallélisation fortement limitée par les dépendances : une instruction bloquée par une dépendance suspend tout le pipeline.
Présent dans certains processeurs embarqués : mips, Arm cortex A15, A53 superscalaire 2 voies (Raspberry Pi3), premiers pentiums (P5 superscalaire 2 voies)

superscalaire dynamique (exécution non ordonnée des instructions)

Meilleure exploitation du parallélisme : des instructions postérieures peuvent s'exécuter, même si une instruction est bloquée.
Présent dans pentium depuis PentiumPro (superscalaire 6 voies), Athlon, et dans plusieurs processeurs Arm depuis V8 Cortex A72 (2 voies), Cortex A75 (3 voies), Cortex A78 (4 voies), Cortex X1 (5 voies), ...

Difficulté de l'exécution non ordonnée : gestion dynamique des dépendances entre instructions.

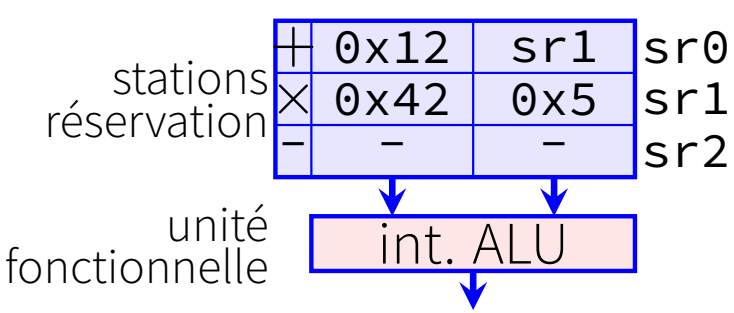
Execution non ordonnée

Deux méthodes principales pour la mise en oeuvre d'un traitement superscalaire dynamique :

- Tableau de marques (*scoreboard*) (Seymour CRAY CDC 6600 1963)
  - tableau centralisé avec informations sur les unités fonctionnelles (occupé, reg. source et dest, unités produisant les données, ...)
  - présente certaines limitations.
    Marche bien pour RAW, mais génère des suspensions pour WAR.
- Algorithme de Tomasulo (Robert TOMASULO IBM 360/91 1966)
  - mécanisme décentralisé
  - utilisé dans la plupart des processeurs superscalaires actuellement

Algorithme de Tomasulo

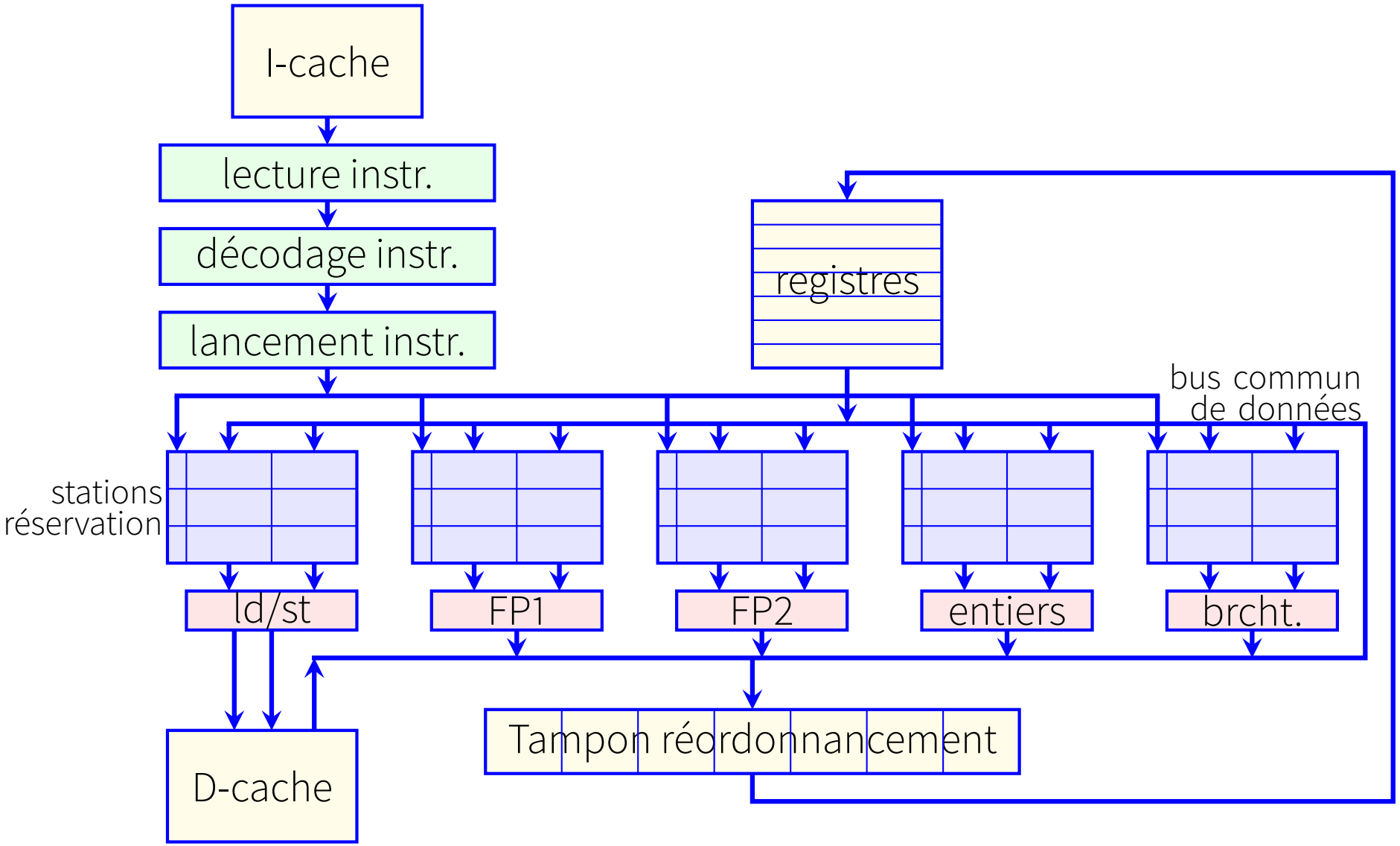
Utilise un mécanisme *distribué* : les *stations de réservation* (SR)
Les SR contiennent les *instructions* à effectuer par les unités fonctionnelles (UF).
Les opérandes des instructions sont soit la valeur si elle est connue, soit un pointeur vers les stations de réservation produisant la donnée.



- l'opérande 2 de `sr0` attend le résultat du calcul décrit par `sr1`
- les deux opérandes de `sr1` sont prêts
- `sr2` est libre et ne contient aucune opération à effectuer

Quand les opérandes d'une SR sont disponibles, l'UF concernée peut effectuer le calcul.
Le résultat produit par une UF est ensuite envoyé sur le *bus commun de données* (CDB) aux registres et aux SR, avec le numéro de la SR qui décrivait l'instruction.
Les SR espionnent le bus et peuvent ainsi récupérer les données dont elles ont besoin.

Algorithme de Tomasulo (cont.)



Algorithme de Tomasulo (cont.)

Les stations de réservation comprennent les informations suivantes :

Op L'opération que doit effectuer l'UF ($+$, $-$, $\times$, etc)

Vj, Vk *valeurs* des opérandes sources (si disponibles)

Qj, Qk Numéro des SR produisant les opérandes source. Une valeur à 0 indique que l'opérande est prêt.

busy Indique que la SR et son UF sont occupées.

dest destination du résultat de l'opération (n° d'entrée dans le tampon de réordonnancement)

Conservation de l'état des registres dans une *table d'état*.
Indique pour chaque registre, l'id **Qi** de la SR qui va écrire le registre.
Mis à jour lors de la phase *dispatch*, si le registre est destination de l'instruction envoyée dans la SR **Qi**.
Si la donnée du registre est à jour **Qi** est à 0

Renommage des registres

Les processeurs comprennent généralement plus de registres que ceux du jeu d'instruction (40–176 registres physiques dans les différentes versions du pentium, contre 8 registres architecturaux)

Lors de la phase de *dispatch*, on teste si le registre destination est en cours d'utilisation (càd va être écrit) en lisant la *table d'état*.
Si **Qi**$\neq$0, on utilise un des registres physiques additionnels (*renommage*).

Permet de supprimer les dépendances WAR et WAW

Le processeur maintient une table des registres libres.
Lors du renommage, on extrait le premier registre libre de la table.

Renommage des registres (cont.)

RAT Register Alias Table : indique pour chaque registre du jeu d'instruction, le numéro du registre physique qui contient la version la plus récente de sa donnée.
Peut être un registre du jeu d'instruction ou un registre additionnel.

Table d'état : pour chaque registre

- état (prêt ou en cours d'écriture)
- identificateur **Qi** de la SR qui va écrire dans ce registre.

Lors du lancement de l'instruction, on utilise ces deux tables pour écrire dans chaque opérande source de la SR

- soit la valeur du registre
- soit indiquer la SR qui va produire la donnée.

Renommage des registres (cont.)

Lors de la phase de lancement des instructions

- attendre qu'une station de réservation *r* soit libre
- chercher un registre physique *p* libre
- renommer le registre destination *d* en *p*
  mettre à jour la RAT de *d* (avec *p*) et la table d'état de *p* (avec la SR *r*)
- pour les opérandes sources *s* (*sa* et *sb*)
  - lire la RAT de *s* pour connaitre le registre physique *p* associé à *s*,
  - lire le registre d'état de *p*
    - si *p* est à jour, lire le banc de registres pour avoir sa valeur
    - sinon lire dans la table d'état le numéro de SR qui va écrire *p*
- écrire ces informations dans la SR *r*

Renommage des registres (cont.)

Pendant la phase d'éxécution

- tant que tous les opérandes d'une SR ne sont pas prêts
  - espionner le CDB pour lire les résultats produits par les SR
  - si un résultat correspond pour un opérande, lire sa valeur et indiquer que l'opérande est prêt
- quand tous les opérandes sont disponibles, la SR est prête et peut être traitée par une UF
- l'UF sélectionne une des SR prêtes et effectue l'opération
- une fois le calcul effectué
  - attendre que le CDB soit libre
  - mettre le résultat avec le n° de SR sur le CDB vers le tampon de réordonnancement
  - indiquer que la SR est libre

Le tampon de réordonnancement

Les écritures en registres sont ordonnées par le *tampon de réordonnancement* (*reorder buffer* ROB).

Le ROB est un tampon circulaire (FIFO), contenant pour chaque entrée :

- si l'entrée est valide
- si l'instruction est achevée
- si l'instruction est spéculative
- la valeur du `pc` de l'instruction
- un événement éventuel associé à l'instruction (exception)
- le registre où la donnée doit être réécrite
- la valeur correspondante (si disponible)

Lors du lancement de l'instruction, une entrée est allouée dans le ROB *de manière ordonnée*.
Le n° de l'entrée de ROB est écrit dans la SR de l'instruction.
Ce n° est mis sur le CDB avec le résultat à la fin de la phase d'exécution.

Quand une entrée dans le ROB est en tête du tampon

- si l'instruction est achevée, on écrit le résultat dans le registre destination.

Le tampon de réordonnancement (cont.)

Le ROB permet de :

- mettre en oeuvre des exceptions précises
- gérer les mauvaises prédictions de branchement (*spéculation*)

Une exception (erreur arithmétique, d'accès mémoire, etc) est traitée quand l'instruction arrive en tête du ROB.
En cas d'exception, toutes les instructions qui suivent sont effacées du ROB.
Aucun registre n'est modifié par une instruction suivant l'exception.

Dans le cas d'un branchement, *toutes* les instructions suivant ce branchement sont marquées comme *spéculatives* après prédiction du branchement.

Aucune ne peut modifier un registre tant que l'unité de branchement n'a pas déterminé si la prédiction du branchement était correcte ou erronée.

En cas d'erreur de prédiction, on efface toutes les instructions spéculatives, et on redémarre au nouveau pc.

Le tampon de réordonnancement (cont.)

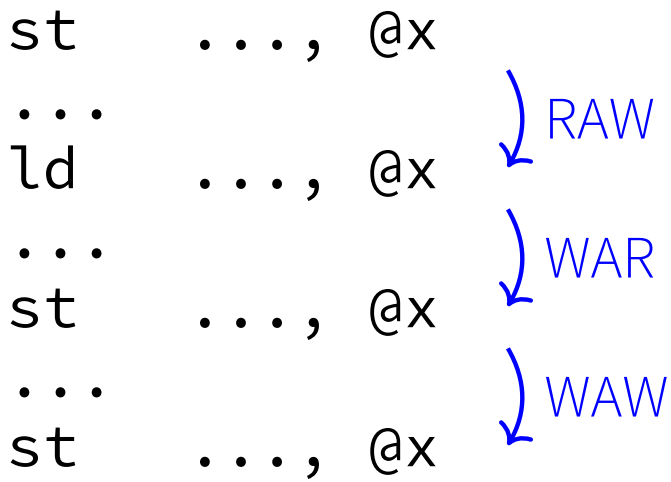
Lors de la phase de retrait des instructions (*retirement*), les valeurs des résultats produits par les instructions sont retirées du ROB et sont copiés dans les registres.

Suivant les architectures :

- le ROB sert pour le renommage des registres.
  Le renommage d'un registre revient à donner le numéro de l'entrée du ROB correspondant.
  Lors de son extraction, une entrée du ROB est directement copiée dans les registres architecturaux.
  Mais on ne peut libérer une entrée de ROB que lors l'extraction du ROB de l'instruction où son registre destination est renommé.
- les registres pour le renommage sont distincts du ROB.
  La recopie des valeurs dans les registres architecturaux se passe alors en deux temps : ROB $\rightarrow$ registres physique de renommage, puis registre de renommage $\rightarrow$ registres architecturaux.

Cas des écritures et lectures mémoire

Aléas possibles lors d'un accès mémoire *à une même adresse* :



Les écritures et lectures mémoire *doivent* être ordonnées.
Il peut arriver

- des aléas RAW (`ld` suivant un `st` à la même adresse)
- des aléas WAR (`st` suivant un `ld` à la même adresse)
- des aléas WAW (`st` successifs à la même adresse mémoire)

Cas des écritures et lectures mémoire (cont.)

De plus :

- les aléas mémoire ne sont pas détectables au décodage des instructions :

```
st   r1, (r4)
...
st   r2, 8(r6)
```

  conduit à un aléa WAW si et seulement si `r4 == r6+8` (*memory aliasing*)

- il n'est pas possible de procéder à un « renommage » pour les adresses mémoire.

Utilisation d'une file d'attente des chargements et rangements (*load/store queue LSQ*)

Cas des écritures et lectures mémoire (cont.)

La LSQ permet :

- d'imposer un ordre pour les lectures écritures (attente de la fin des instructions précédentes avant d'effectuer un `ld/st`)
- d'éviter certains accès mémoire.
  - si on écrit dans le LSQ un `ld` à une adresse présente dans un `st` précédent :
    - on récupère la valeur écrite dans le `st`
  - si on écrit dans la LSQ un `st` à une adresse présente dans un autre `st` :
    - on peut supprimer le `st` le plus ancien
- "*memory disambiguation*"

Entrée de LSQ :

`ld/st`	`pc`	seq	@mem	donnée

Les entrées de LSQ sont crées *de manière ordonnée* dans l'étage de décodage.

Cas des écritures et lectures mémoire (cont.)

Cas des `st`

Les `st` doivent être ordonnés entre eux pour éviter les aléas WAW.

Quand l'adresse et la donnée d'un `st` deviennent connues :

- chercher les `st` plus anciens à la même adresse pour éviter de les effectuer
- chercher les `ld` plus récents pour leur transmettre la donnée (*forwarding*).

Cas des écritures et lectures mémoire (cont.)

Cas des `ld`

Par contre, les `ld` n'ont pas besoin d'être ordonnés entre eux, ni avec les `st`, si ils concernent des adresses mémoire différentes des `st`.

Un `ld` est généralement considéré comme critique car bloquant les instructions ayant besoin de la donnée.

On cherche donc à exécuter un `ld` au plus tôt.

L/S	PC	Seq	@mm	don.
S	1432	5234	1234	xxx
S	1256	5235	5678	xxx
S	1472	5236	90AB	xxx
L	4432	5237	CDEF	–

1/ Aucun problème.
Envoyer le `ld` au cache pour exécution au plus vite

L/S	PC	Seq	@mm	don.
S	1432	5234	1234	xxx
S	1256	5235	5678	xxx
S	1472	5236	90AB	xxx
L	4432	5237	1234	–

2/ Un `st` à la même adresse que le `ld` inséré.
On récupère la donnée immédiatement.

L/S	PC	Seq	@mm	don.
S	1432	5234	1234	????
S	1256	5235	5678	xxx
S	1472	5236	90AB	xxx
L	7432	5237	1234	–

3/ La donnée du `st` de même adresse que le `ld` est encore inconnue. Attendre sa disponibilité

L/S	PC	Seq	@mm	don.
S	1432	5234	ABCD	xxx
S	1256	5235	????	xxx
S	1472	5236	90AB	xxx
L	5237	1234	CDEF	–

4/ Avant le `ld`, il y a un `st` à une adresse encore inconnue.
Problème...

Cas des écritures et lectures mémoire (cont.)

Dans le cas 4/ d'un `st` à une adresse encore inconnue avant un `ld`, on peut :

- bloquer le `ld` et attendre que l'adresse du `st` soit connue.
  Simple, mais peut créer beaucoup d'attentes inutiles pour les `ld`.
- exécuter le `ld` en le rendant *spéculatif*.
  Quand l'adresse du `st` sera connue, on pourra lever la spéculation :
  - Si l'adresse est différente, pas de problème
  - Si les deux adresses sont identiques :
    - On efface le ROB et on reprend l'exécution du programme à l'adresse du `ld`. Très pénalisant quand cela arrive.
    - On ne réexécute que les instructions impactées par le `ld`. Très compliqué, mais existe sur certaines architectures.

La LSQ est rendue plus complexe avec accès mémoire de taille variable et adresses non alignées.

`st.byte...,@1238`
`...`
`ld.byte...,@1236`
Pas de problème, cases mémoire différentes

`st.long...,@1236`
`...`
`ld.byte...,@1238`
Recouvrement partiel, mais possibilité de lire l'octet du `ld`

`st.byte...,@1238`
`...`
`ld.long...,@1236`
Recouvrement. Le `ld` lit `long` @1236 et modifie l'octet @1238 du `st`.

`st.long...,@1238`
`...`
`ld.long...,@1236`
`long`s non alignés sur des adresses $\times 4$. Recouvrement entre les accès mémoire.

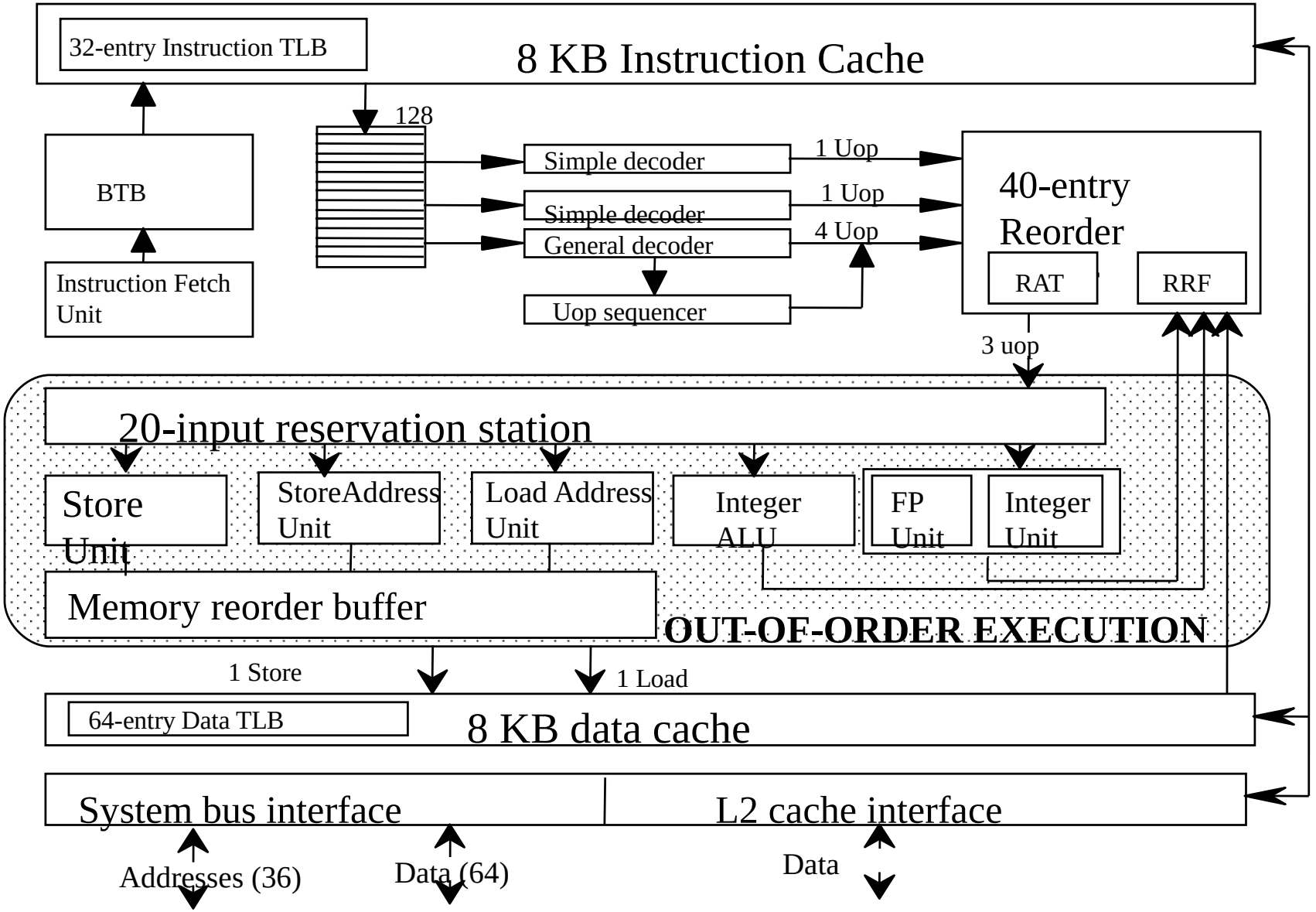
Cas des écritures et lectures mémoire (cont.)

Caches non bloquants

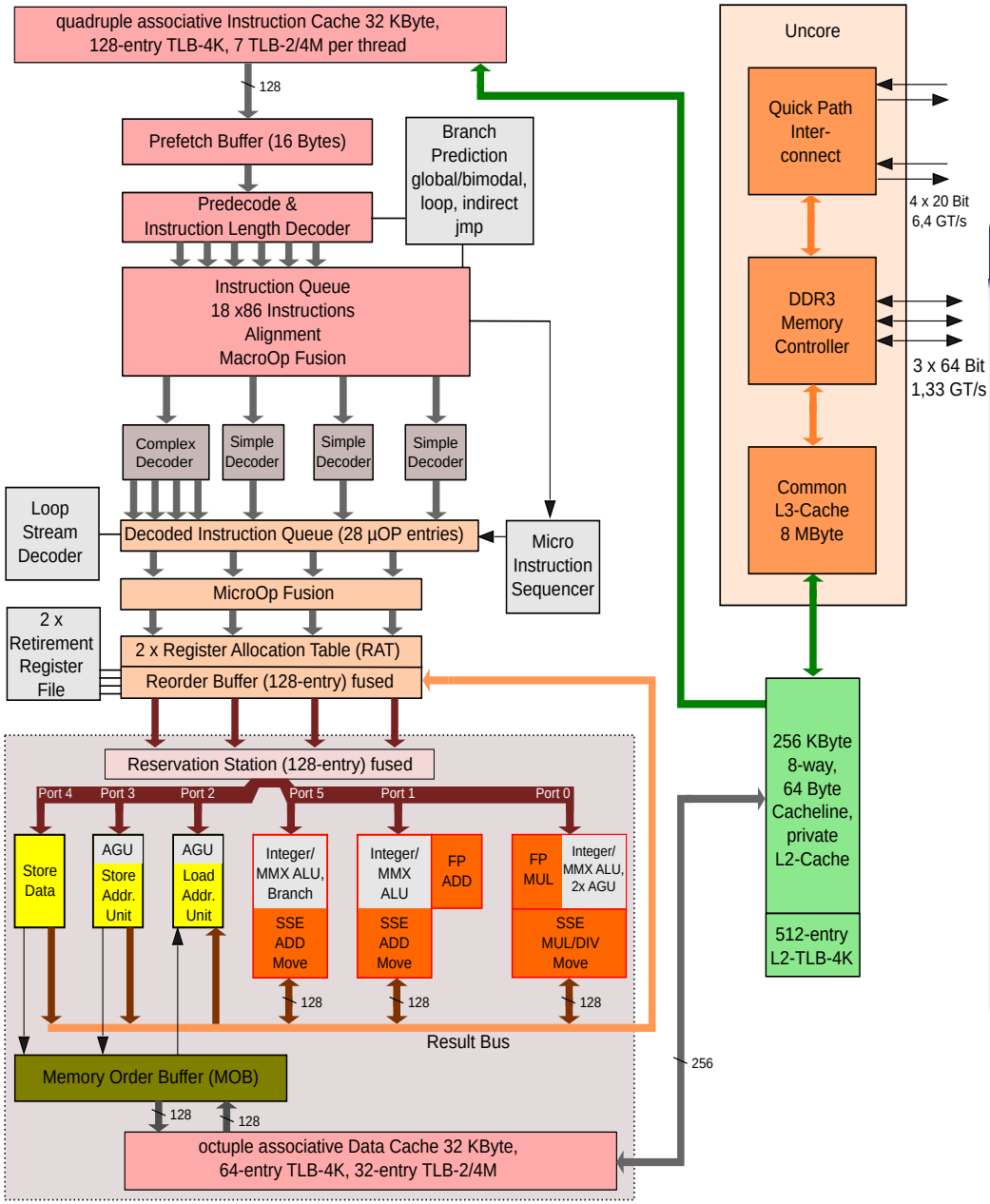
Pour utiliser au mieux l'exécution non ordonnée des instructions, le processeur doit pouvoir continuer à exécuter des `ld` alors qu'un `ld` précédent a provoqué un défaut de cache.

Un cache non bloquant permet plusieurs accès cache simultanés, et notamment de traiter des succès pendant le traitement d'un échec (*hit under miss*).

Schéma fonctionnel du pentium PRO



Intel Nehalem

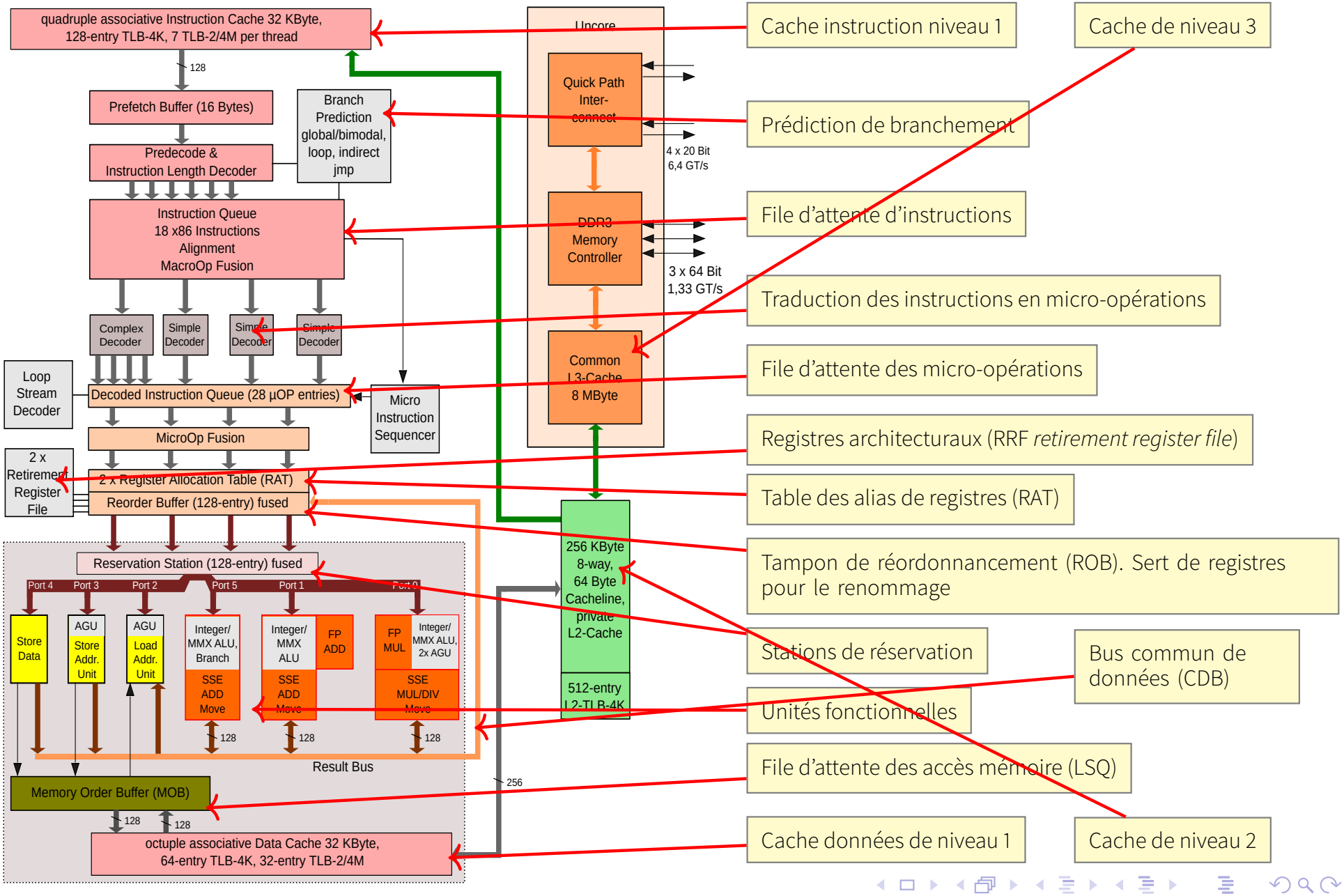


Nehalem

Microarchitecture du pentium introduite en 2008
Inclut notamment :

- *hyperthreading*
- 3 niveaux de cache
- contrôleur mémoire intégré
- instruction SSE4
- bus système QPI
- etc...

Intel Nehalem



Cache instruction niveau 1

Cache de niveau 3

Prédiction de branchement

File d'attente d'instructions

Traduction des instructions en micro-opérations

File d'attente des micro-opérations

Registres architecturaux (RRF *retirement register file*)

Table des alias de registres (RAT)

Tampon de réordonnancement (ROB). Sert de registres pour le renommage

Stations de réservation

Bus commun de données (CDB)

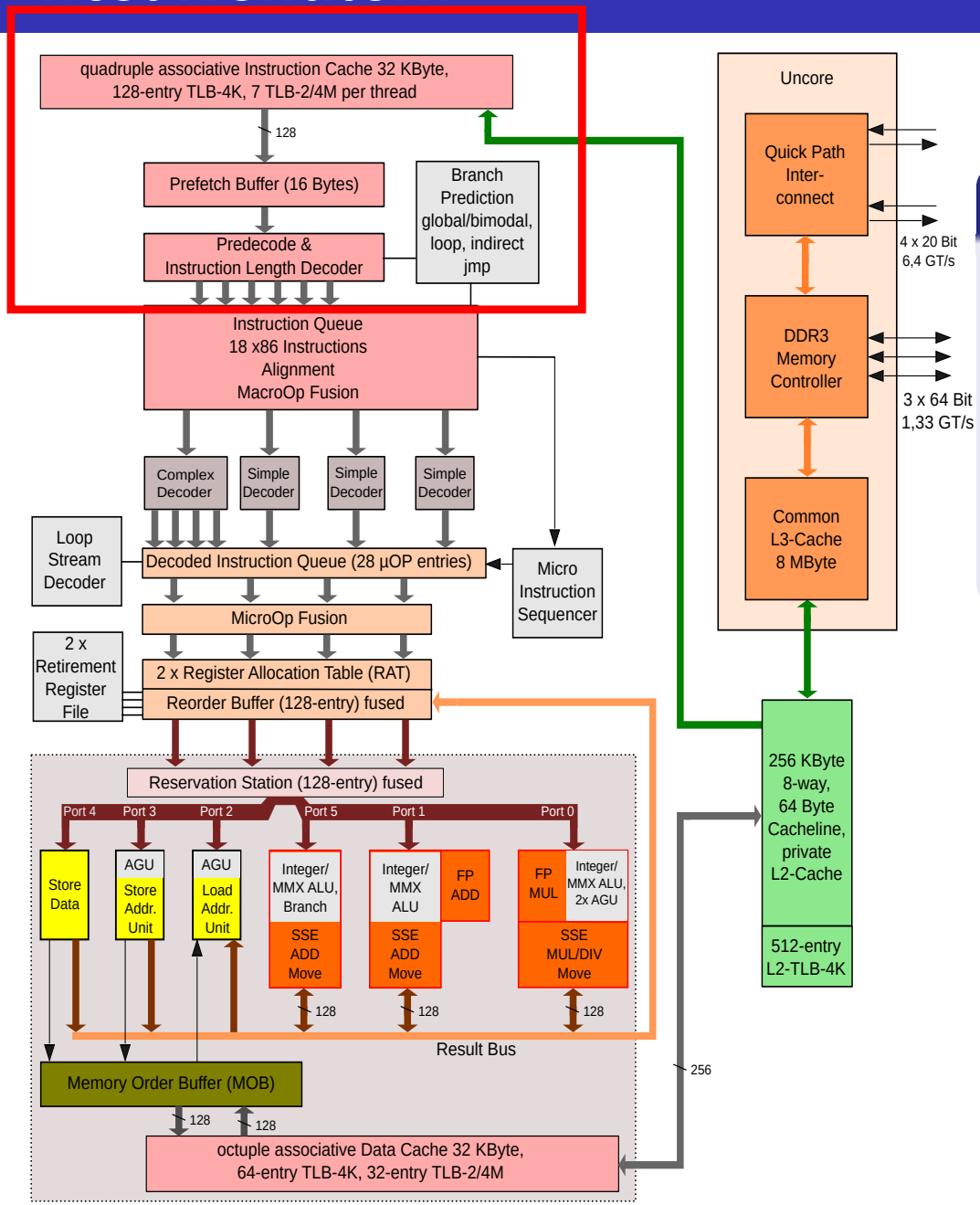
Unités fonctionnelles

File d'attente des accès mémoire (LSQ)

Cache données de niveau 1

Cache de niveau 2

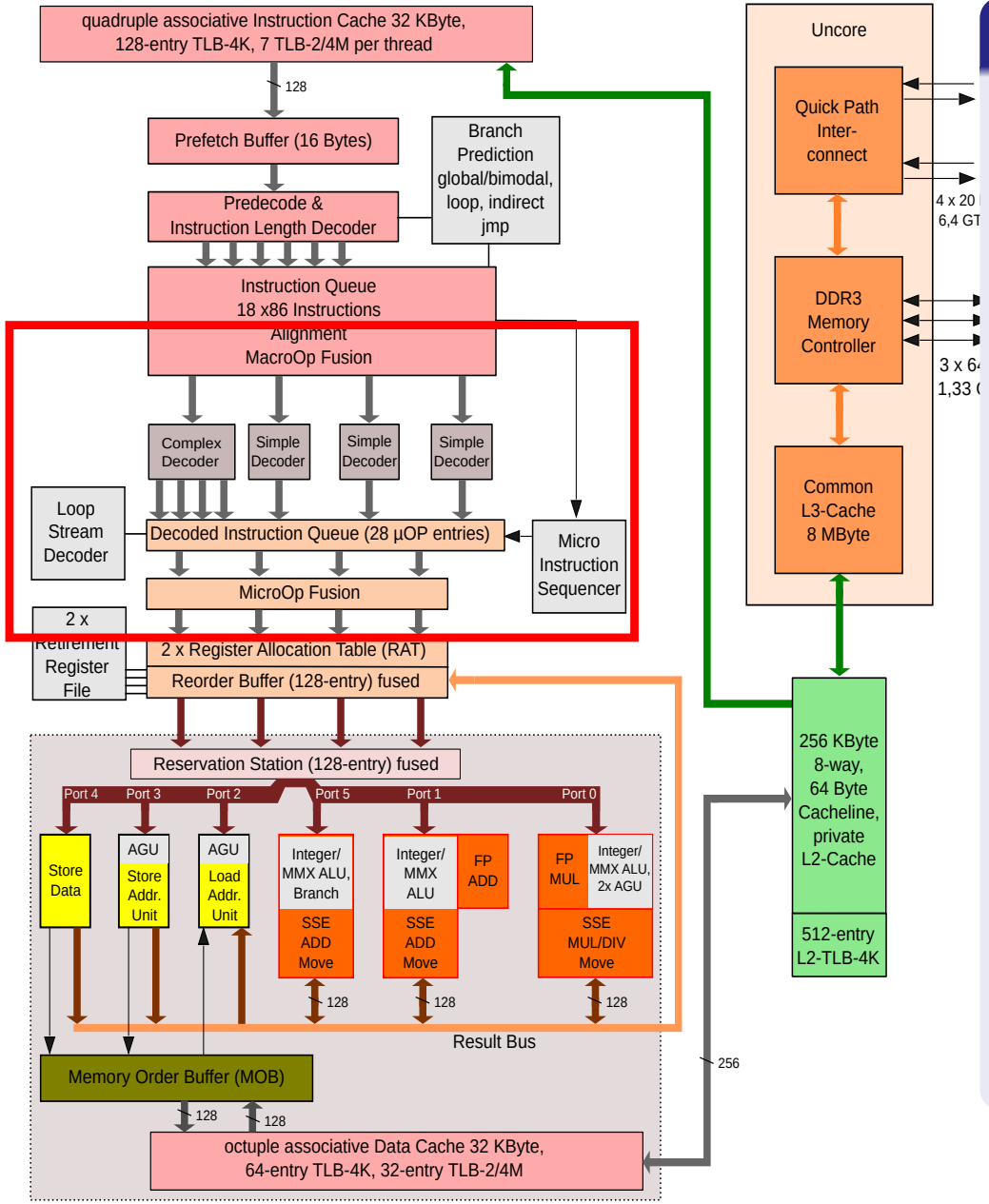
Intel Nehalem (cont.)



Instruction fetch

- aller chercher *n* instructions consécutives dans le cache instructions
- nécessite une prédiction des branchements

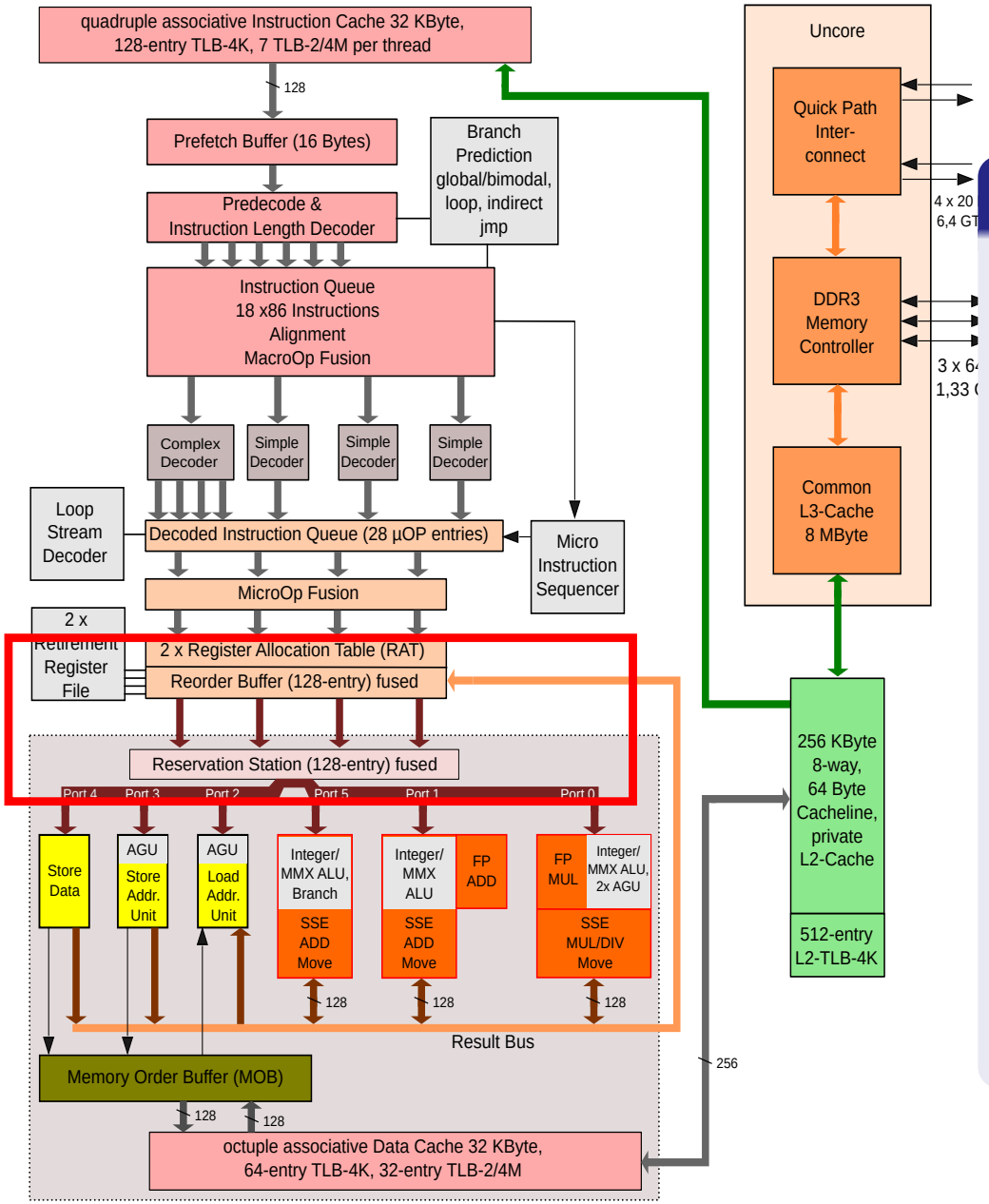
Intel Nehalem (cont.)



Instruction decode

- (pour le pentium, traduire les instructions en μopérations)
- pour chaque instruction (ou μopération),
  - pour chaque registre source, chercher dans la RAT le registre (RRB ou ROB) associé et éventuellement la SR qui va produire la donnée
  - créer une entrée pour l'instruction dans le ROB et faire le renommage de l'opérande destination dans la RAT
  - pour les accès mémoire, créer une entrée pour l'instruction dans la LSQ (MOB)

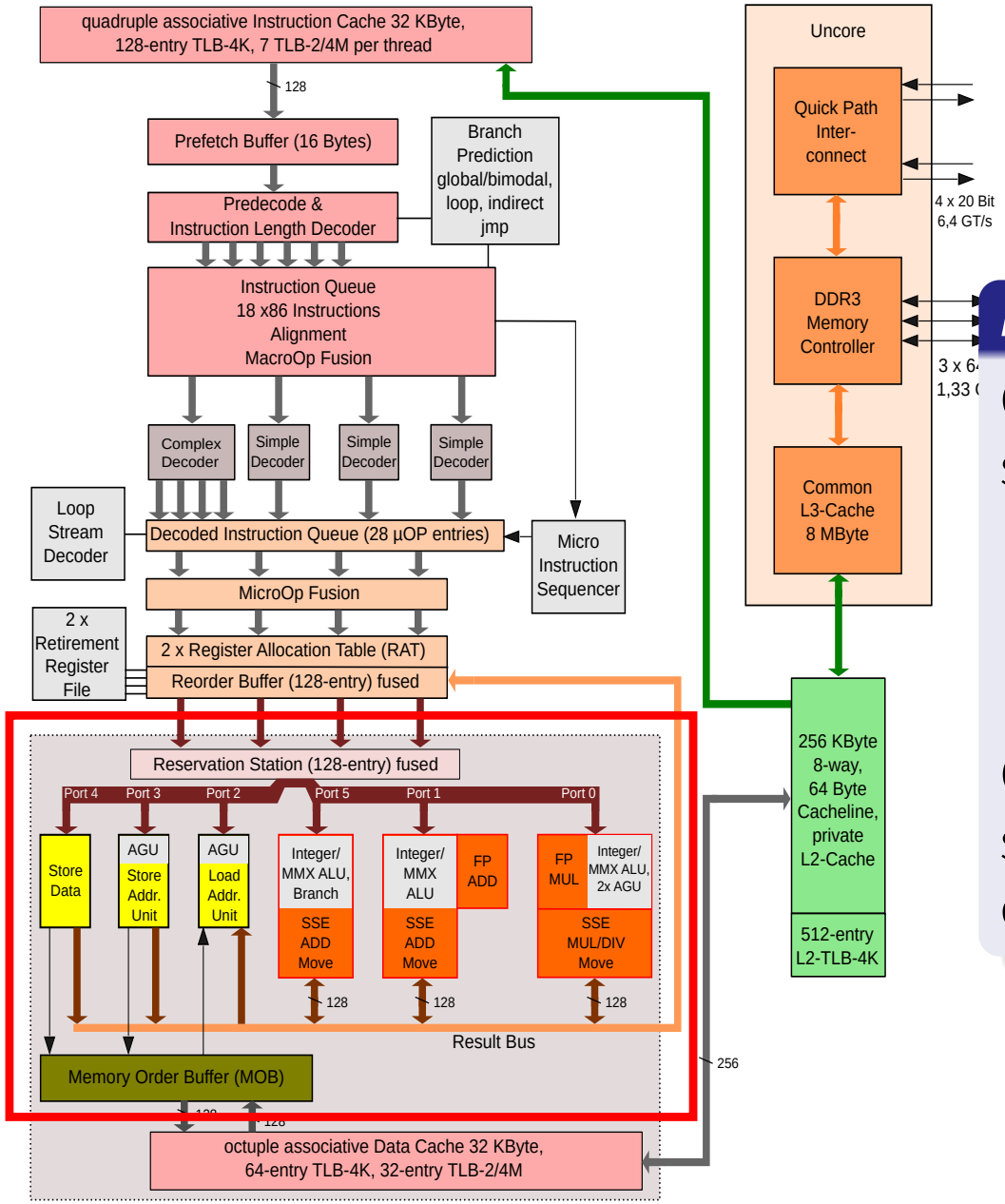
Intel Nehalem (cont.)



Instruction dispatch

- prendre une ou plusieurs instructions en attente
- chercher si les opérandes sont disponibles dans la RRF ou le ROB
- chercher s'il y a une SR disponible pour l'UF correspondante
  - si oui : distribuer l'instruction et ses opérandes vers la SR
  - si non : attente qu'une SR se libère (aléa structurel)

Intel Nehalem (cont.)



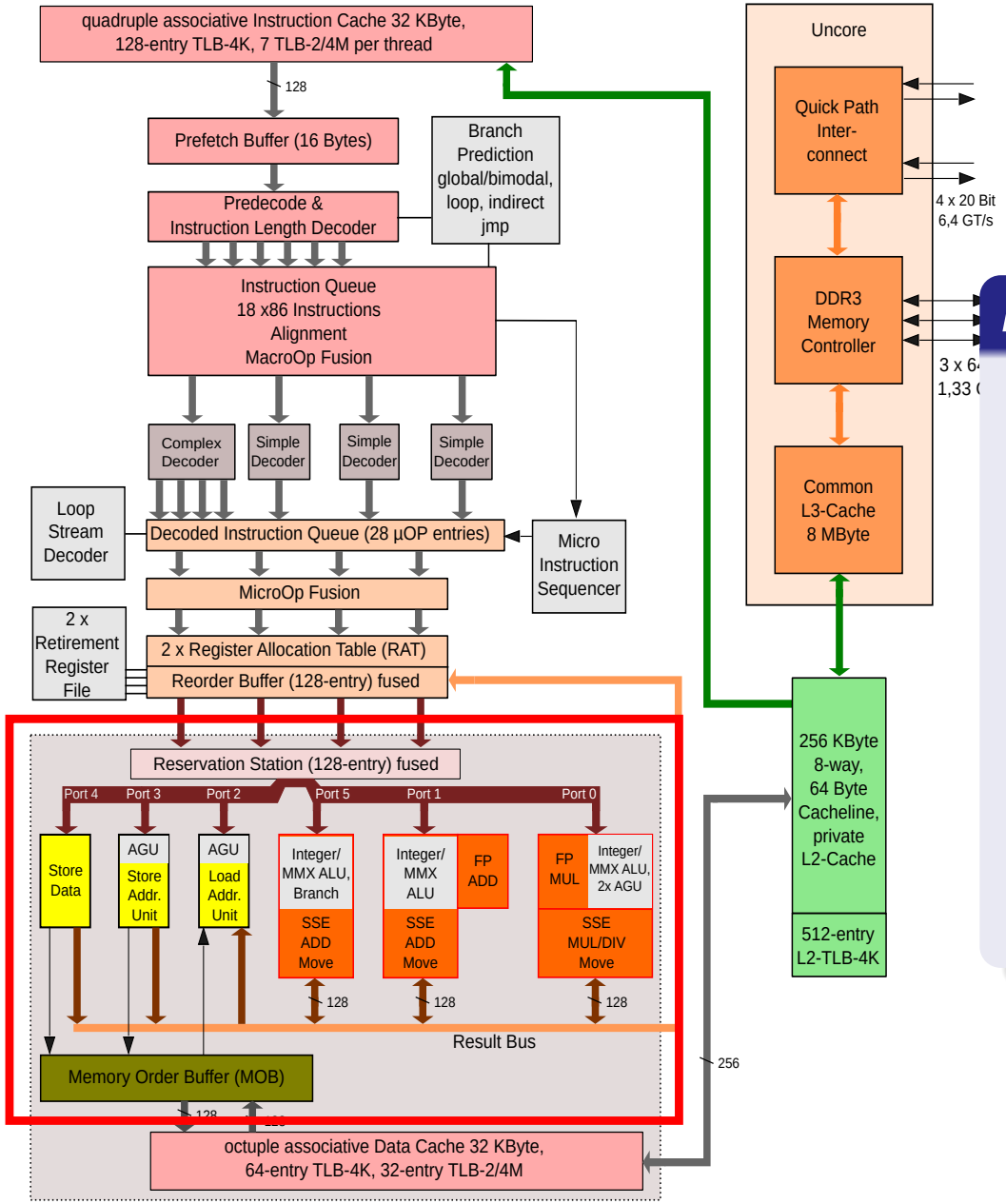
Instruction execute

Chaque UF cherche dans les stations de réservation

- si tous les opérandes source de la SR sont disponibles, lancer l'exécution

Chaque station de réservation surveille le CDB pour ses différents opérandes

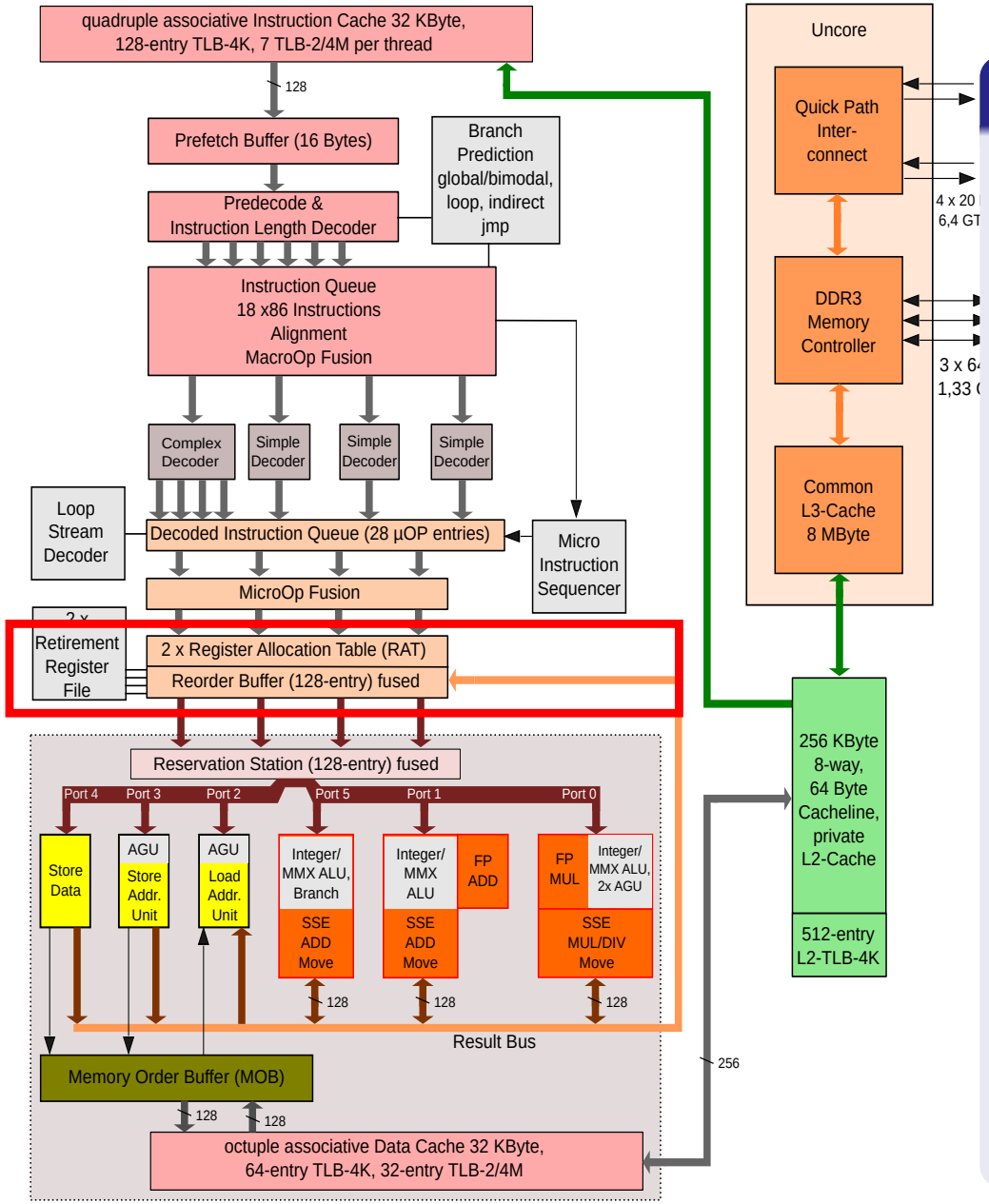
Intel Nehalem (cont.)



Instruction complete

- si un résultat est disponible dans une UF
  - si le CDB est disponible, envoyer le résultat vers le ROB par le CDB.
  - si non, attendre que le CDB se libère.

Intel Nehalem (cont.)



Instruction retire

- Si toutes les instructions précédentes du ROB (dans l'ordre d'exécution du programme) sont retirées (ou peuvent être retirées dans le même cycle)
- et s'il n'y a pas eu d'interruption avant ou pendant l'exécution
- et si l'instruction n'est pas (ou plus) spéculative
- alors :
  - recopier la valeur dans le banc de registres (RRB)
  - marquer l'entrée de ROB comme achevée
    elle sera retirée du ROB dès que l'on n'aura plus besoin de sa valeur

Bilan superscalaire

En pratique, le parallélisme d'instruction obtenu est relativement limité.

Sur les pentium récents 1.3 – 1.8 instructions par cycle

Les principales limitations viennent :

- de la prédiction des branchements
- des accès à la mémoire

Le surcoût du parallélisme d'instruction superscalaire est important (comparable à l'ensemble des unités fonctionnelles).

Les processeurs VLIW

Very Long Instruction Word

Principes :

- Ordonnancement statique des instructions par le compilateur avec prise en compte des
  - aléas de données
  - aléas de contrôle
- exécution pipelinée des instructions
- exécution parallèle de N instructions dans N unités fonctionnelles

Présent dans des processeurs de traitement de signal (C6x, trimedia) et certains processeurs pour serveurs (IA-64).

Toute la parallélisation est faite par le compilateur

- graphe de dépendances locales : bloc de base
- graphe de dépendance du programme : ordonnancement de traces

Les processeurs VLIW (cont.)

Objectifs :

- matériel simplifié
- performances améliorées
- consommation réduite

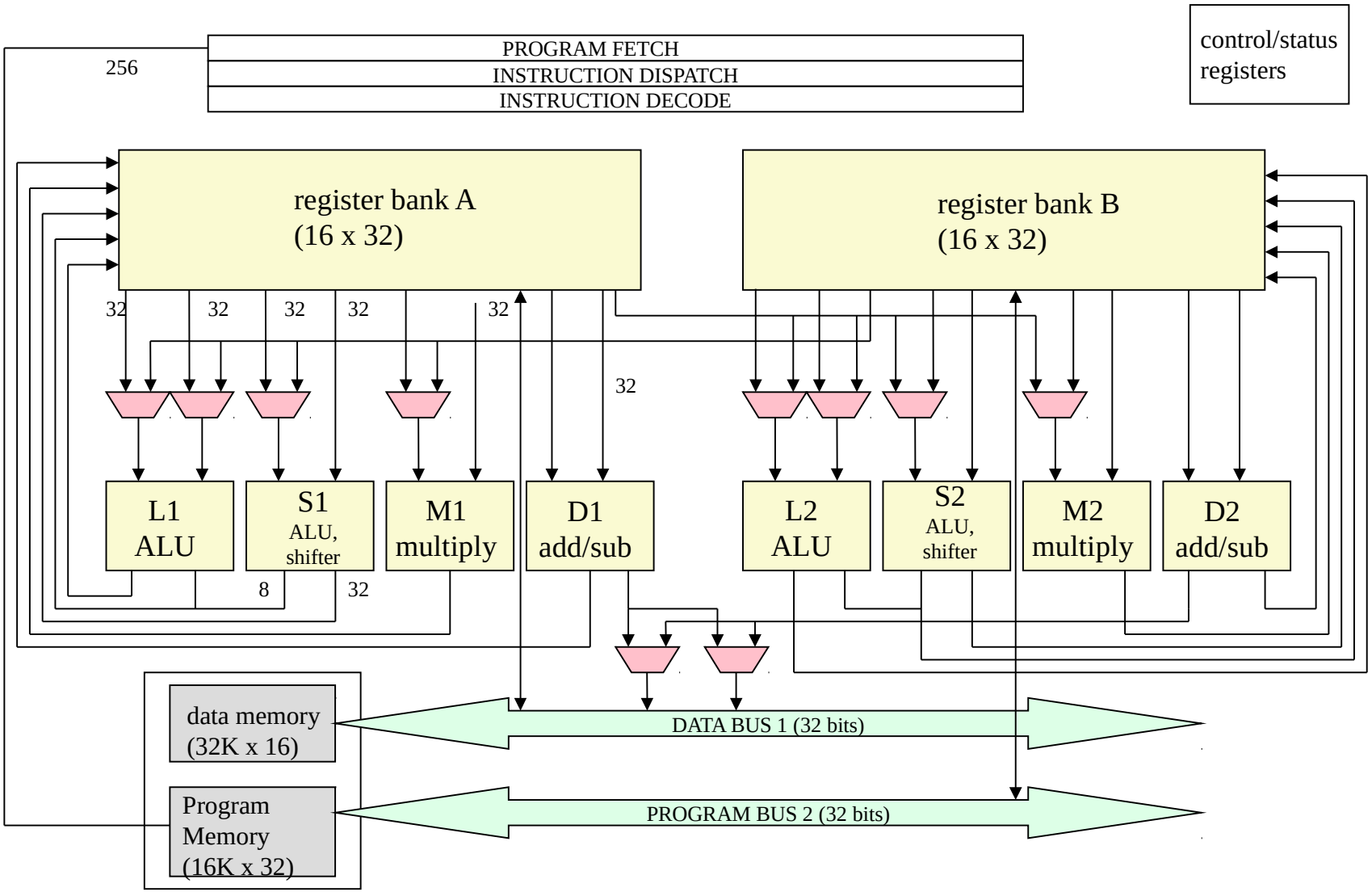
La réduction de complexité matérielle est possible car :

- moins de problèmes pour lancer plusieurs instructions : les dépendances sont déja résolues
- pas d'exécution dans le désordre
- pas de détection dynamique d'aléas

Tout est fait par le compilateur qui peut potentiellement extraire un parallélisme plus important.
Inconvénients :

- la compatibilité binaire entre générations successives est difficile ou impossible
- mauvaise prise en compte des délais dynamiques (défauts de cache)

Schéma fonctionnel du C6x (Texas)



La compilation VLIW

Le compilateur génère (jusqu'à) 8 instructions éxécutées à chaque cycle et correspondant à chacune des unité fonctionnelles et les groupe dans une « instruction très longue ».

Nombreuses d'opérations nécessaires pour accroitre le parallélisme d'instructions en détectant les aléas et en masquant la latence :

- aléas structurels :
  - éviter 2 opérations sur la même unité fonctionnelles, deux accès au même banc mémoire, etc
- aléas de données
  - évite les aléas de données dans un groupe d'instructions
- aléas de contrôle
  - prédiction statique des branchements
  - instructions avec prédicat pour éviter certains branchements
- masquage de la latence
  - préacquisition des données

La compilation VLIW (cont.)

Utilisation systématique de techniques de compilation avancées

- déroulage de boucles
- *inlining* des fonctions
- pipeline logiciel : génère des instructions correspondant à des itérations différentes
- *ordonnancement de traces* : génère simultanément des instructions potentiellement éloignées dans le code
  - on sélectionne une *trace* (séquence d'instructions principale)
  - on optimise la trace en ajustant les instructions pour gérer les problèmes d'entrée et sortie de la trace
  - on ajoute des blocs pour connecter les autres chemins à la trace

Instructions du C6x

- VLIW : 8 x 32 bits
- Instructions 32 bits, correspondant à 4 x 2 unités fonctionnelles
- 2 ensemble A et B de 16 registres généraux
- Instructions à exécution conditionnelle
  5 registres généraux `A1`, `A2`, `B0`, `B1`, `B2` peuvent contenir la condition ($\neq 0$ ou $=0$)

Modes d'adressage du C6x

Existence de modes d'adressage sophistiqués orientés traitement de signal

Type d'adressage	Sans modification registre adresse	Préincrement ou décrement registre adresse	Postincrément ou décrement registre adresse
Registre Indirect	`*R`	`*++R` `*--R`	`*R++` `*R--`
Registre + Déplacement	`*+R[ucst5]` `*-R[ucst5]`	`*++R[ucst5]` `*--R[ucst5]`	`*R++[ucst5]` `*R--[ucst5]`
Registre + Index	`*+R[offsetR]` `*-R[offsetR]`	`*++R[offsetR]` `*--R[offsetR]`	`*R++[offsetR]` `*R--[offsetR]`

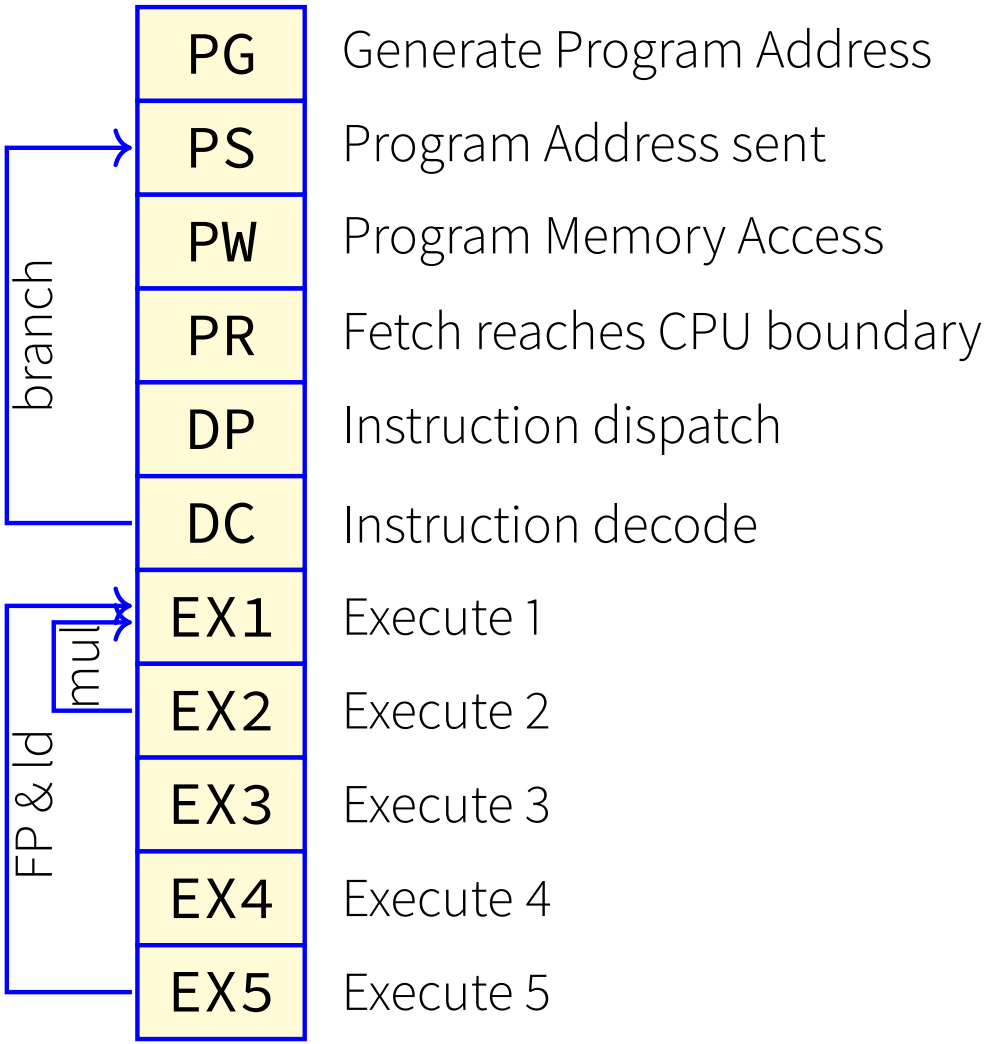
Unités fonctionnelles et instructions associées

Unités fonctionnelles : .L ALU .M Multiplication .S Shift .D Mémoire

.L unit		.M unit	.S unit		.D unit
abs	sadd	mpy	add	mvkh	add
add	sat	smpy	addk	neg	adda
and	ssub	mpyh	add2	not	ld mem
cmpeq	sub		and	or	ld mem (15 bit) (D2)
c.pgt	subc		b disp	set	mv
cmpgtu	xor		b irp	shl	neg
cmplt	zero		b nrp	shr	st mem
cmpltu			breg	shru	st mem (15 bit) (D2)
lmbd			clr	sshl	sub
mv			ext	sub	suba
neg			extu	sub2	zero
norm			mvc (S2)	xor	
not			mv	zero	
or			mvk		

Pipeline du C6x

Pipeline 11 étages



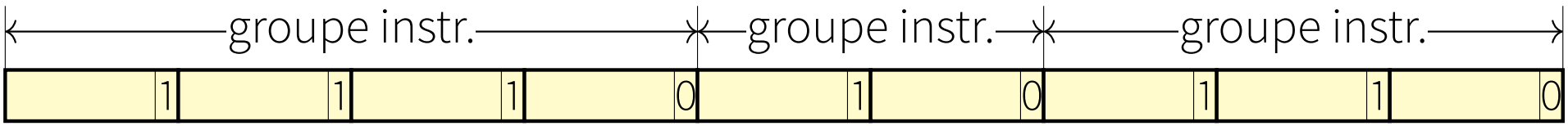
Type instruction	Latence
NOP	1
Store	1
Instr. simple (add)	1
Mult. (16x16)	2
Load	5
Branchement	6
Addition SP	5
Multiplication SP	5

Acquisition des instructions

Acquisition des instructions en parallèle par groupe jusqu'à 8

Mécanisme de compression des instructions :

le bit de poids faible de chaque instruction indique si l'instruction qui suit peut être exécutée en parallèle ou séquentiellement.



Produit scalaire sur C6x

```
int dotp(short[a], short[b])
{
  int sum, i;
  sum=0;
  for (i=0; i<100;i++)
    sum+=a[i]*b[i];
  return (sum);
}
```

```
;;; @ai->A4 @bi->A3 ai->A2 bi->A5 sum->A7 cntr(i)->A1
        MVK    .S1    100,A1       ; set up loop counter
        ZERO   .L1    A7           ; zero out accumulator
LOOP:   LDH    .D1    *A4++,A2     ; load ai in A2
        LDH    .D1    *A3++,A5     ; load bi in A5
        MPY    .M1    A2,A5,A6     ; ai*bi in A6
        ADD    .L1    A6,A7,A7     ; sum+=(ai*bi) in A7
        SUB    .S1    A1,1,A1      ; decrement loop counter
[A1]    B      .S2    LOOP         ; branch to loop if A1!=0
```

Produit scalaire sur C6x (cont.)

Prise en compte des délais des opérateurs

```
       MVK    .S1    100,A1        ; set up loop counter
       ZERO   .L1    A7            ; zero out accumulator
LOOP:
       LDH    .D1    *A4++,A2      ; load ai
       LDH    .D1    *A3++,A5      ; load bi
       NOP    4                    ; delay slots for LDH
       MPY    .M1    A2,A5,A6      ; ai*bi
       NOP    1                    ; delay slot for MPY
       ADD    .L1    A6,A7,A7      ; sum+=(ai*bi)
       SUB    .S1    A1,1,A1       ; decrement loop counter
[A1]   B      .S2    LOOP          ; branch to loop
       NOP    5                    ; delay slots for branch
```

16 cycles/itération

Produit scalaire sur C6x (cont.)

Réordonnancement d'instructions et exécution d'instructions en parallèle :

- *bi* mis dans banc de registre B (permet LDW en parallèle)
- SUB et B avancés (les branchements sont *retardés*)

```
        MVK     .S1     100,A1    ; set up loop counter
||      ZERO    .L1     A7        ; zero out accumulator
LOOP:
        LDH     .D1     *A4++,A2  ; load ai
||      LDH     .D2     *B4++,B2  ; load bi
        SUB     .S1     A1,1,A1   ; decrement loop counter
[A1]    B       .S2     LOOP      ; branch to loop
        NOP     2                 ; delay slots for LDH
        MPY     .M1X    A2,B2,A6  ; ai*bi
        NOP     1                 ; delay slot for MPY
        ADD     .L1     A6,A7,A7  ; sum+=(ai*bi)
; branch occurs here
```

8 cycles/itération

Produit scalaire sur C6x (cont.)

Modification de l'algorithme : déroulage et utilisation de deux accumulateurs

```
int dotp(short[a], short[b])
{
  int sum, sum0, sum1, i;
  sum0=0;
  sum1=0
  for (i=0; i<100;i+=2){
    sum0+=a[i]*b[i];
    sum1+=a[i+1]*b[i+1];
  }
  sum=sum0+sum1;
  return (sum);
}
```

Produit scalaire sur C6x (cont.)

Assembleur correspondant
NB : les registres (32 bits) `A2` et `B2` contiennent 2 `shorts` (16 bits).

```
        MVK    .S1     100,A1     ; set up loop counter
        ZERO   .L1     A7         ; zero out sum0
        ZERO   .L2     B7         ; zero out sum1
LOOP:
        LDW    .D1     *A4++,A2   ; load ai and ai+1
        LDW    .D2     *B4++,B2   ; load bi and bi+1
        MPY    .M1X    A2,B2,A6   ; ai*bi
        MPYH   .M2X    A2,B2,B6   ; ai+1*bi+1
        ADD    .L1     A6,A7,A7   ; sum0+=(ai*bi)
        ADD    .L2     B6,B7,B7   ; sum1+=(ai+1*bi+1)
        SUB    .S1     A1,1,A1    ; decrement loop counter
[A1]    B      .S2     LOOP       ; branch to loop
; branch occurs here
        ADD    .L1X    A7,B7,A4    ; sum=sum0+sum1
```

Produit scalaire sur C6x (cont.)

Mise en paralléle d'instructions, introduction des délais et réordonnancement

```
        MVK     .S1     100,A1    ; set up loop counter
||      ZERO    .L1     A7        ; zero out sum0
||      ZERO    .L2     B7        ; zero out sum1
LOOP:
        LDW     .D1     *A4++,A2  ; load ai and ai+1
||      LDW     .D2     *B4++,B2  ; load bi and bi+1
        SUB     .S1     A1,1,A1   ; decrement loop counter
[A1]    B       .S2     LOOP      ; branch to loop
        NOP     2                 ; delay slots for LDH
        MPY     .M1X    A2,B2,A6  ; ai*bi
||      MPYH    .M2X    A2,B2,B6  ; ai+1*bi+1
        NOP     1                 ; delay slot for MPY
        ADD     .L1     A6,A7,A7  ; sum0+=(ai*bi)
||      ADD     .L2     B6,B7,B7  ; sum1+=(ai+1*bi+1)
; branch occurs here
        ADD     .L1X    A7,B7,A4  ; sum=sum0+sum1
```

8 cycles/2 itérations

Produit scalaire sur C6x (cont.)

Table des intervalles entre itérations

	0	1	2	3	4	5	6	7
.D1	LDW							
.D2	LDW							
.M1						MPY		
.M2						MPYH		
.L1								ADD
.L2								ADD
.S1		SUB						
.S2			B					

Avec deux itérations

	0	1	2	3	4	5	6	7	8
.D1	LDW	LDW1							
.D2	LDW	LDW1							
.M1						MPY	MPY1		
.M2						MPYH	MPYH1		
.L1								ADD	ADD1
.L2								ADD	ADD1
.S1		SUB	SUB1						
.S2			B	B1					

Produit scalaire sur C6x (cont.)

Avec 8 itérations

	0	1	2	3	4	5	6	7	8
.D1	LDW	LDW1	LDW2	LDW3	LDW4	LDW5	LDW6	LDW7	LDW8
.D2	LDW	LDW1	LDW2	LDW3	LDW4	LDW5	LDW6	LDW7	LDW8
.M1						MPY	MPY1	MPY2	MPY3
.M2						MPYH	MPYH1	MPYH2	MPYH3
.L1								ADD	ADD1
.L2								ADD	ADD1
.S1		SUB	SUB1	SUB2	SUB3	SUB4	SUB5	SUB6	SUB7
.S2			B	B1	B2	B3	B4	B5	B6

prologue boucle épilogue

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
.D1	0	1	2	3	4	5	6	7	8							
.D2	0	1	2	3	4	5	6	7	8							
.M1						0	1	2	3	4	5	6	7	8		
.M2						0	1	2	3	4	5	6	7	8		
.L1								0	1	2	3	4	5	6	7	8
.L2								0	1	2	3	4	5	6	7	8
.S1		0	1	2	3	4	5	6	7	8						
.S2			0	1	2	3	4	5	6	7	8					

Produit scalaire sur C6x (cont.)

Utilisation de pipeline logiciel

```
;;; initialization
;;; prologue
LOOP:
        LDW    .D1     *A4++,A2 ; load ai and ai+1    (+7)
||      LDW    .D2     *B4++,B2 ; load bi and bi+1    (+7)
||[A1]  SUB    .S1     A1,1,A1  ; decr. loop counter  (+6)
||[A1]  B      .S2     LOOP     ; branch to loop      (+5)
||      MPY    .M1X    A2,B2,A6 ; ai*bi                (+2)
||      MPYH   .M2X    A2,B2,B6 ; ai+1*bi+1            (+2)
||      ADD    .L1     A6,A7,A7 ; sum0+=(ai*bi)        (+0)
||      ADD    .L2     B6,B7,B7 ; sum1+=(ai+1*bi+1)    (+0)
; branch occurs here
;;; epilogue
;;; final addition
        ADD    .L1X    A7,B7,A4 ; sum=sum0+sum1
```

1 cycle/2 itérations (au lieu de 16 cy/itération pour version initiale !)
+ prologue/épilogue