Renommage et ILP

Le compilateur mappe la mémoire sur les registres architecturaux.
Les registres architecturaux étant peu nombreux, sont fortement réutilisés.
Cela crée de fausses dépendances entre instructions: deux instructions qui écrivent deux valeurs indépendantes dans le même registre par exemple.

Renommage et ILP

Dans une machine à registres, on distingue trois types de dépendances:

- dépendance Lecture Après Ecriture ou LAE
- dépendance Ecriture Après Lecture ou EAL
- dépendance Ecriture Après Ecriture ou EAE

Renommage et ILP

Par exemple:

R1 = R2 + R3
R4 = R1 + R0 // dépendance LAE (R1)
R4 = R2 + R0 // dépendance EAE (R4)
R2 = R1 + R0 // dépendance EAL (R2)

Renommage et ILP

Les dépendances LAE sont des dépendances de données.
Les dépendances EAE et EAL sont des dépendances de ressources: c'est la réutilisation du registre qui crée la dépendance.
En re-mappant les registres architecturaux sur un ensemble plus large de registres de renommage, le matériel peut éliminer les dépendances de ressource et favoriser l'ILP.

Réordonnancement dynamique

```
racine( unsigned x) {
int i = –1;
unsigned s = 0;
while (s<x) {
   i+=2;
   s+=i;
 }
return (i/2+1);

}
```

```
                /* x dans R1 */
                R0 = –1
                R9 = 0
                si R0>= R1 vers f
e:              R0 = 2 + R0
                R9 = R9 + R0
                si R0 < R1 vers e
f:              R0 = R0 / 2
                R0 = R0 + 1
                RET
                /* résultat dans R0 */
```

On sait extraire les instructions en //

Peut-on les exécuter en //?

Il faut construire l'ordre partiel des instructions

Dépendances LAE, EAE et EAL

code source

```
     /* x dans R1 */
     R0 = –1
     R9 = 0
     si R0>= R1 vers f
e:   R0 = 2 + R0
     R9 = R9 + R0
     si R0 < R1 vers e
f:   R0 = R0 / 2
     R0 = R0 + 1
     RET
     /* résultat dans R0 */
```

fragment d'exécution

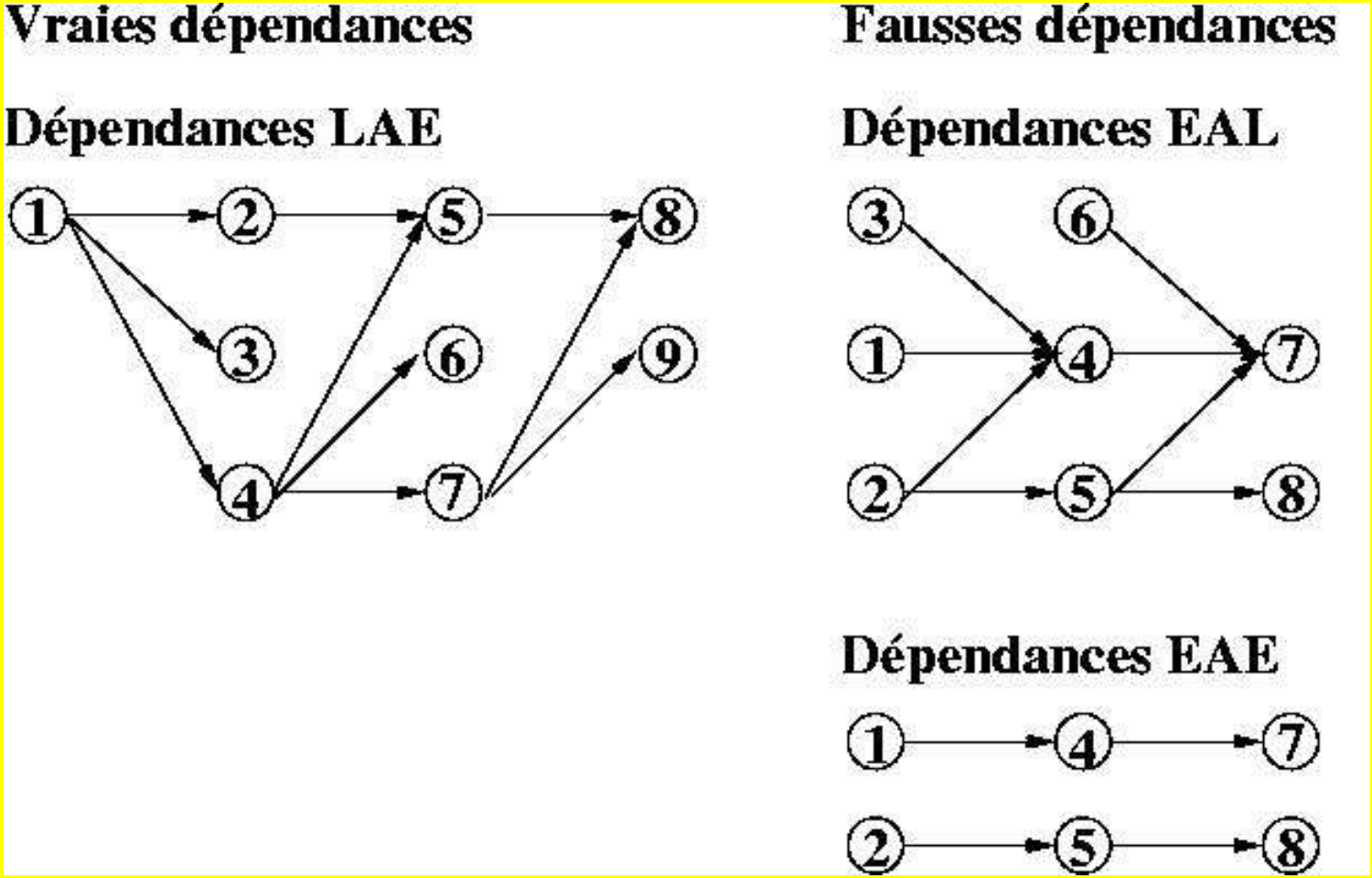
```
R0 = 2 + R0
R9 = R9 + R0
si R0 < R1 vers e
R0 = 2 + R0
R9 = R9 + R0
si R0 < R1 vers e
R0 = 2 + R0
R9 = R9 + R0
si R0 < R1 vers e
```

Dépendance LAE: 2 dépend de 1

Dépendance EAE: 4 dépend de 1

Dépendance EAL: 4 dépend de 3

Graphe des dépendances LAE, EAE et EAL



Les dépendances EAE et EAL sont de fausses dépendances. On les élimine en dédoublant le registre de destination: c'est le renommage.

Renommage des registres

fragment d'exécution	fragment renommé
R0 = 2 + R0	RR10 = 2 + RR00
R9 = R9 + R0	RR19 = RR09 + RR10
si R0 < R1 vers e	si RR10 < RR01 vers e
R0 = 2 + R0	RR20 = 2 + RR10
R9 = R9 + R0	RR29 = RR19 + RR20
si R0 < R1 vers e	si RR20 < RR01 vers e
R0 = 2 + R0	RR30 = 2 + RR20
R9 = R9 + R0	RR39 = RR29 + RR30
si R0 < R1 vers e	si RR30 < RR01 vers e

Pour matérialiser les r registres de l'architecture on dispose de q (q>r) registres de renommage

Chaque destination est associée à un registre de renommage disponible (allocation)

Renommage

On voit sur la boucle précédente que le renommage a permis de mapper chaque tour de boucle sur un jeu de registres distinct.

Il ne reste plus que la dépendance de donnée sur R0, due à la nature de la boucle qui accumule son résultat dans R0.

Renommage

```
saxpy(float *x, float *y,
float a, int n) {
 int i;
 for (i=0;i<n;i++) y[i]+=x[i]*a;
}
```

```
// R1: x, R2: y
// F1: a, R3: n
// R8: i
     R8 = 0
e:  F8 = M[R1 + 4*R8] // x[i]
    F8 = F8 * F1 // x[i] * a
    F9 = M[R2 + 4*R8] // y[i]
    F9 = F9 + F8 // y[i] + a*x[i]
    M[R2 + 4*R8] = F9 // y[i] = F9
    R8 = R8 + 1
    si (R8<R3) vers e // if (i<n) goto e
```

Renommage

```
// R1: x, R2: y
// F1: a, R3: n
// R8: i
     R8 = 0
e:  F8 = M[R1 + 4*R8] // x[i]
     F8 = F8 * F1 // x[i] * a
     F9 = M[R2 + 4*R8] // y[i]
     F9 = F9 + F8 // y[i] + a*x[i]
     M[R2 + 4*R8] = F9 // y[i] = F9
     R8 = R8 + 1
     si (R8<R3) vers e // if (i<n) goto e
```

```
RR08 = 0
RF10 = M[RR01 + 4*RR08]
RF11 = RF10 * RF01
RF12 = M[RR02 + 4*RR08]
RF13 = RF12 + RF11
M[RR02 + 4*RR08] = RF13
RR18 = RR08 + 1
si (RR18 < RR03) vers e
RF20 = M[RR01 + 4*RR18]
RF21 = RF20 * RF01
RF22 = M[RR02 + 4*RR18]
RF23 = RF22 + RF21
M[RR02 + 4*RR18] = RF23
RR28 = RR18 + 1
...
```

Renommage

Le matériel, en réordonnant les calculs, peut produire plus tôt les différentes valeurs de i (mais au mieux une par cycle).
Chaque nouvelle valeur permet de lancer la chaîne des calculs du tour de boucle associé.
L'ILP maximal obtenu est le nombre d'instructions du corps de boucle.
Pour augmenter l'ILP, dérouler les boucles.

Renommage

Le renommage permet d'éliminer les fausses dépendances (dépendances EAE et EAL)
pour ne conserver que l'ordre partiel d'exécution imposé par les vraies dépendances (dépendances LAE)

Les destinations EAE et EAL sont dédoublées, ce qui permet une exécution dans un ordre quelconque

L'association des sources avec la destination dont elles dépendent (LAE) doit se faire après son renommage

Renommage

Une table RAT associe chaque registre architectural à son renommage le plus récent.

Pour n instructions extraites et renommées solidairement (ligne extraite):

Renommage des destinations: on alloue n registres pris dans une liste de registres libres. La table RAT est mise à jour.

Renommage des sources (dépendance intra–ligne): comparer les deux sources de l'instruction i aux (i–1) destinations des instructions précédentes.

Renommage des sources (dépendance inter–ligne): la table RAT donne le nom du renommage le plus récent.

Renommage des destinations

Quatre instructions extraites et renommées solidairement:

Code extrait:	Allocation:	RAT:
R0 = 2 + R0	RR47 alloué	RAT[0]=47
R9 = R9 + R0	RR48 alloué	RAT[9]=48
si (R0<R1) vers e	-	-
R0 = 2 + R0	RR49 alloué	RAT[0]=49

Renommage des sources

Code extrait:

R0 = 2 + R0

R9 = R9 + R0

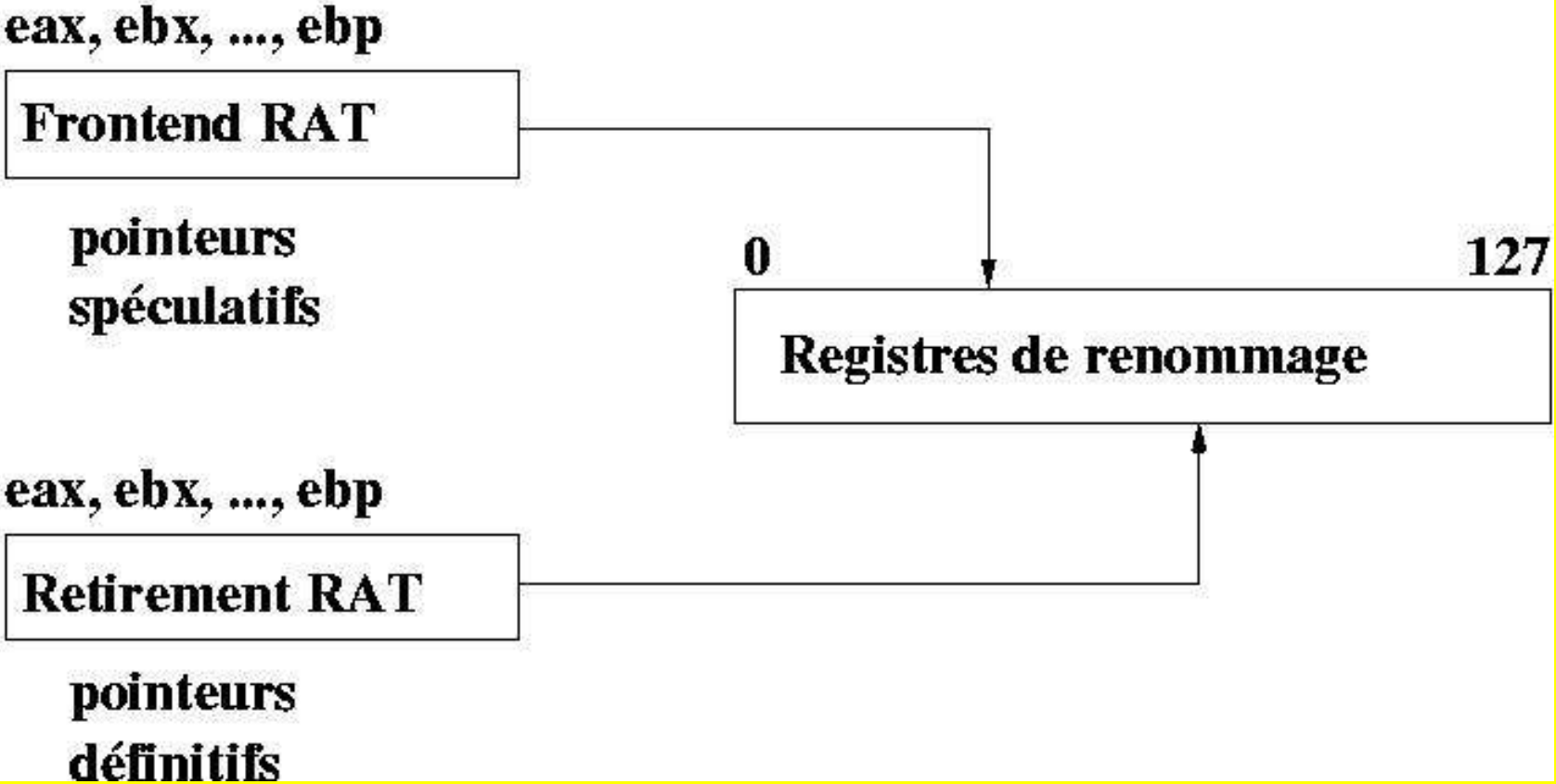
si (R0<R1) vers e

R0 = 2 + R0

Comparaisons (source gauche)	Renommage (source gauche)	Comparaisons (source droite)	Renommage (source droite)
-	-	-	RAT[0]
R9/R0	RAT[9]	R0/R0	RR47
R0/R9, R0/R0	RR47	R1/R9, R1/R0	RAT[1]
-	-	R0/-, R0/R9, R0/R0	RR47

Renommage et validation:

Renommage du Pentium 4 (NetBurst): pointeurs



Renommage de l'instruction: F-RAT[d] = RR alloué

Validation de l'instruction: R-RAT[d] = RR validé

Renommage Netburst (P4)

RRr, renommant Rd, est libéré quand R-RAT[d] == r est modifiée par la validation d'une instruction de destination Rd renommée RRr'.
R-RAT[d] reçoit r' et r est libéré.
A ce moment, le registre RRr' devient le représentant de Rd.

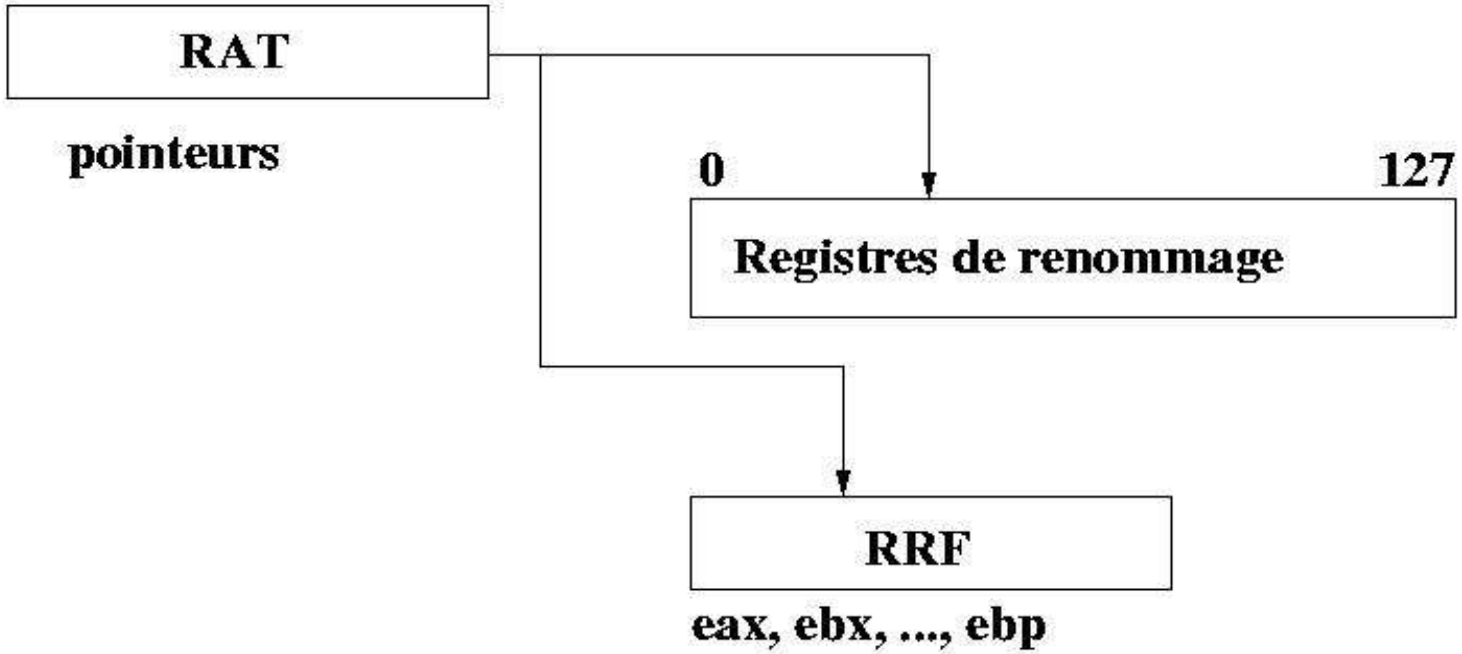
Renommage spéculatif P4

```
Si (cond) Vers e
R0 = ...              // renommage spéculatif
```

Le renommage spéculatif est enregistré dans Frontend-RAT[0].
Si la prédiction de saut est fausse, on récupère le renommage antérieur de Retirement-RAT[0].
Quand une instruction i est validée, son renommage spéculatif r devient définitif (ici, r porté par i, renommant R0 est copié dans Retirement-RAT[0]; F-RAT[0] peut être != r).

Renommage et validation:

Solution du pentium iii (P6): copie



Renommage de i, dest=d: RAT[d] = r (RRr)

Validation de i: RRF[d] = RRr (r porté par i)

(attention: on peut avoir RAT[d]!=RRr)

Renommage PIII

RRr est alloué au renommage et libéré à la validation.
Le représentant de Rd est soit RRF[d] si Rd n'est pas renommé, soit RR[r] si Rd est renommé RRr.
L'allocation et la libération des registres de renommage fonctionnent en file d'attente.

Renommage spéculatif PIII

```
Si (cond) Vers e
R0 = ...              // renommage spéculatif
```

Le renommage est enregistré dans RAT[0].

Les renommages sont spéculatifs.

Si la prédiction de saut est fausse, on annule tous les renommages spéculatifs en vidant la RAT.

Tous les registres sont représentés par leur RRF.

	pointeur (P4)	copie (PIII)
lecture	pointeur spéculatif ou si vide pointeur définitif puis registre de renommage	pointeur spéculatif puis registre de renommage ou si vide registre architectural
validation	copie du pointeur spéculatif dans le pointeur définitif	copie du registre de renommage dans le registre architectural
libération	pointeur définitif modifié	validation
structure	table à trous	file

Références mémoires et ILP

La détection des dépendances LAE, EAE et EAL entre références mémoires ne peut intervenir qu'une fois les adresses calculées.

```
M[R3 + R4] = R8
...
R0 = M[R1 + R2] // dépendance LAE?
M[R5 + R6] = R0 // dépendance EAL?
                // dépendance EAE?
```

Références mémoires et ILP

Les calculs des adresses peuvent s'effectuer en désordre (tout ordre est valide).
Les accès eux-mêmes peuvent s'effectuer en désordre (certains ordres sont invalides).
Un chargement à l'adresse a ne peut passer devant un rangement impliquant a (LAE).
Un rangement en a ne peut passer devant un chargement ou un rangement impliquant a (EAL ou EAE).

Références mémoires et ILP

L'accès d'un chargement ne peut être démarré que si:

. son adresse a est calculée,

. les adresses de tous les rangements antérieurs sont calculées,

. les accès de tous les rangements antérieurs impliquant a sont effectués.

Références mémoires et ILP

L'accès d'un rangement ne peut être démarré que si:

. son adresse **a** est calculée,

. les adresses de tous les chargements et rangements antérieurs sont calculées,

. les accès de tous les chargements et rangements antérieurs impliquant **a** sont effectués.

Accès mémoire spéculatif

```
Si (cond) Vers e
M[R0+R1] = ...          // accès mémoire spéculatif
```

Les chargements spéculatifs ne sont pas problématiques. Ils peuvent avoir un effet de préchargement du cache de données.

Les rangements spéculatifs sont placés dans une file d'attente spéculative: le *store buffer*.

A la validation du rangement, l'écriture est effectuée dans le cache.

Si la prédiction de saut est fausse, on vide le *store buffer*.

Accès mémoire spéculatif

Le *store buffer* devient le premier niveau de la hiérarchie mémoire pour les chargements. Il a une structure de file pour l'entrée et la sortie des rangements et de cache pour les recherches des chargements.