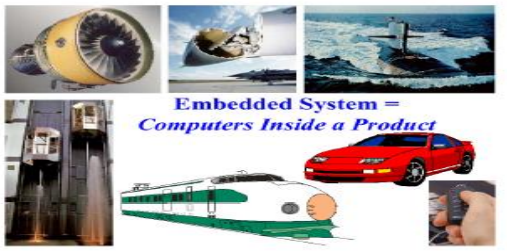
M1-SEM
Processeurs embarqués
Cours 6.2
Superscalaire-VLIW

Année 2020-2021

Pr. R. BOUDOUR



Profondeur de pipeline

- Attacher à chacune des colonnes de caractéristiques : superpipeline et pipeline

* le temps de cycle doit être long car chaque étage est un circuit volumineux qui demande un temps de stabilisation long
* temps de traversée d'une seule instruction demande peu de cycles
* peu de registres de pipeline
* vidange de pipeline rapide

* le temps de cycle peu être court car les étages sont des circuits petits et rapides
* beaucoup de registres de pipeline
* temps de traversée long mais
* débit élevé
* vidange de pipeline pénalisante

Processeur VLIW

- Montrer comment un compilateur planifierait la séquence d'opérations suivante pour une exécution sur un processeur VLIW doté de 3 unités d'exécution. Nous supposerons que toutes les opérations possèdent une latence de 2 cycles et que chacune des unités d'exécution peut exécuter n'importe quel type d'opération.

```
ADD r1, r2, r3
SUB r16, r14, r7
LD r2, (r4)
LD  r14, (r15)
MUL r5, r1, r9
ADD r9, r10, r11
SUB r12, r2, r14
```

Processeur VLIW

- L'opération LD r14, (r15) est placée avant SUB r16, r14, r7 bien que SUB apparait avant elle dans le programme d'origine et lise le registre de destination de LD.
- Les opérations VLIW n'écrasent pas leurs valeurs de registres avant d'être terminées, la valeur précédente r14 reste disponible 2 cycles après l'émission de LD, ce qui permet à SUB de lire l'ancienne valeur de r14 et de générer le résultat correct.
- Cette logique de planification OOO des opérations permet au programme d'être réorganisé en un nombre réduit de fois.

Instruction 1	ADD r1, r2, r3	LD r2, (r4)	LD r14, (r15)
Instruction 2	SUB r16, r14, r7	ADD r9, r10, r11	NOP
Instruction 3	MUL r5, r1, r9	SUB r12, r2, r14	NOP

Processeur VLIW

- Montrer comment un compilateur réorganiserait le code ci-dessous, pour une exécution sur un processeur VLIW doté de 4 unités d'exécution. Les opérations de chargement ont une latence de deux cycles et les autres un cycle.

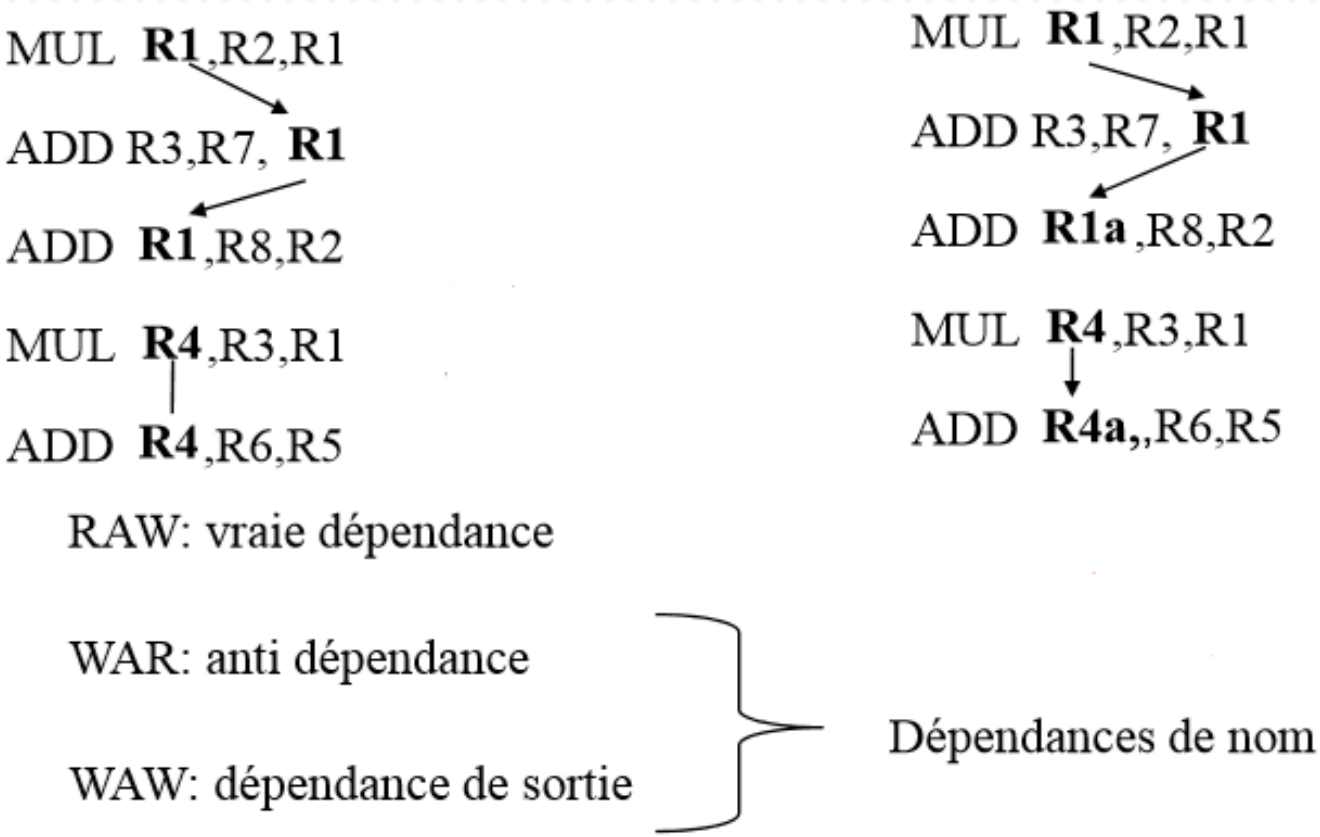
```
ADD  r1, r2, r3
SUB  r5, r4, r5
LD  r4, (r7)
MUL  r4, r4, r4
ST  (r7), r4
LD  r9, (r10)
LD  r11, (r12)
ADD  r11, r11, r12
MUL  r11, r11, r11
ST  (r12), r11
```

Processeur VLIW

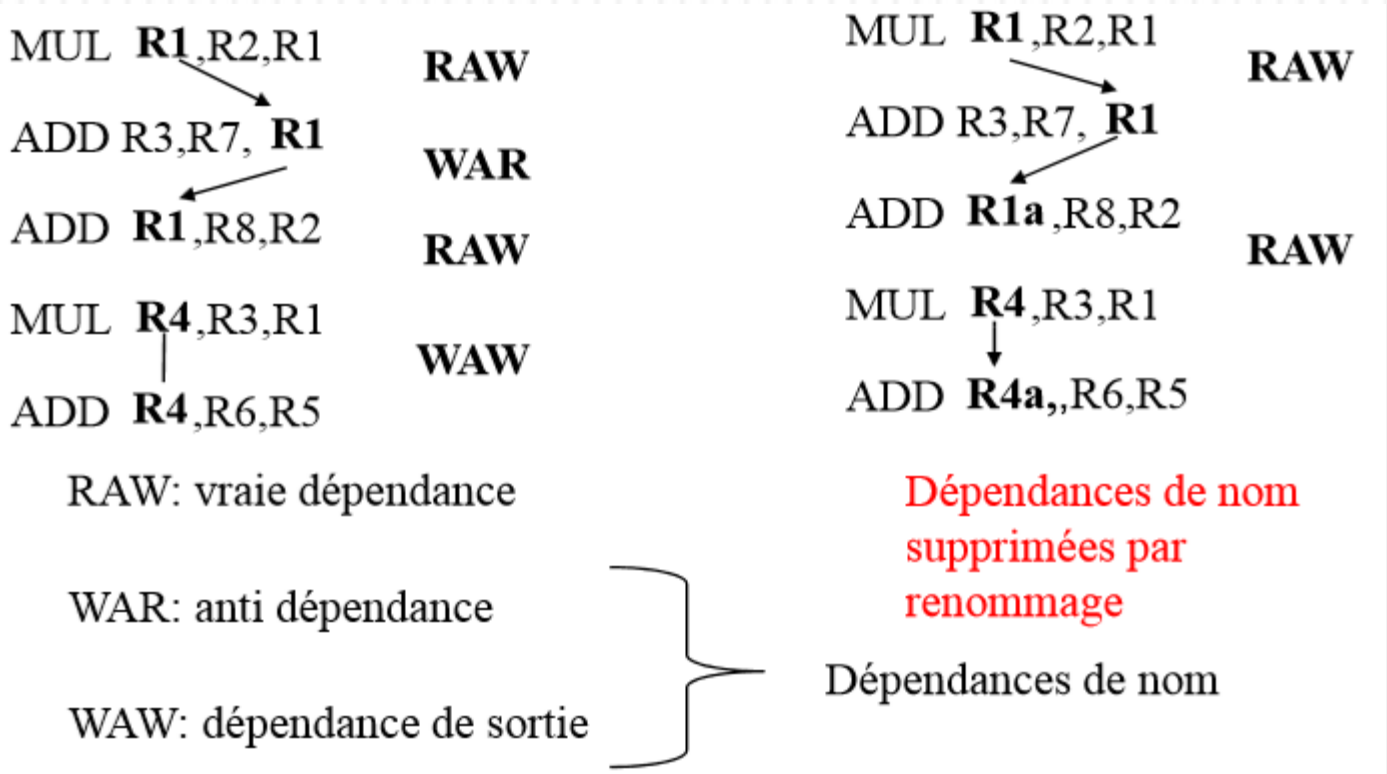
- Le code peut être réorganisé en 5 instructions VLIW;
- La liste ci-dessous présente un exemple d'agencement correct mais d'autres combinaisons d'instructions sont également possibles :

Instruction 1	SUB r5, r4, r5	LD r4, (r7)	LD r9, (r10)	LD r11, (r12)
Instruction 2	ADD r1, r2, r3	NOP	NOP	NOP
Instruction 3	MUL r4, r4, r4	ADD r11, r11, r12	NOP	NOP
Instruction 4	ST (r7), r4	MUL r11, r11, r11	NOP	NOP
Instruction 5	ST (r12), r11	NOP	NOP	NOP

Types d'aléas du pipeline



Traitement des aléas

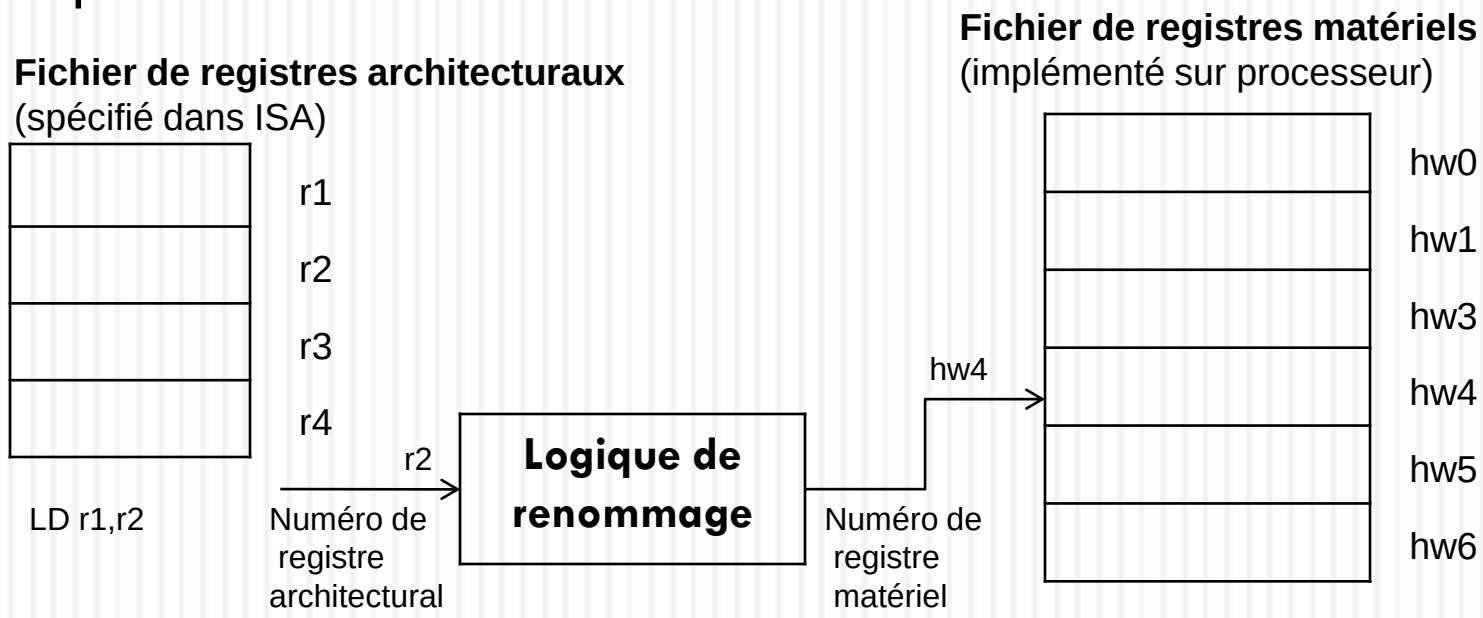


Traitement des aléas

- **Contrôle des dépendances de données**
  - Réalisé par matériel
    - Tableau de marques (scoreboard)
    - Algorithme de Tomasulo
- **Suppression des dépendances de nom**
  - Réalisé par matériel
    - Renommage de registres
    - Tableau de marques (scoreboard)
- **Techniques logicielles pour supprimer les suspensions**
    - Déroulage de boucle
    - Pipeline logiciel

Renommage de registres

- Le renommage de registres est une technique qui vise à minimiser l'impact des aléas EAL et EAE sur le parallélisme d'instructions en assignant dynamiquement chaque valeur produite par un programme à un nouveau registre afin de rompre les aléas EAL et EAE



Renommage de registres

- **Exemple de renommage :**

```
ADD r3, r4, r5                ADD  hw3, hw4, hw5
LD r7, (r3)                   LD  hw7, (hw3)
SUB  r3, r12, r11     ⟶      SUB  hw20, hw12, hw11
ST  (r15), r3                 ST  (hw15), hw20
```

- Grâce au renommage de registres, la première écriture est mappée sur hw3, la seconde sur hw20 (arbitraire)
- Le renommage de registres est plus avantageux pour le processeur OOO (réordonnancement après renommage)

Renommage de registres

- Sur un processeur superscalaire OOO doté de 8 unités d'exécution de la séquence ci-dessous avec et sans renommage de registres si chaque unité d'exécution peut exécuter n'importe quelle instruction et si la latence de toutes les instructions est d'un cycle ?
- Fichier de registres matériel > fichier architecturaux
- Pipeline à 5 étages

Renommage de registres


```
LD  r7, (r8)
MUL  r1, r7, r2
SUB  r7, r4, r5
ADD  r9, r7, r8
LD  r8, (r12)
DIV  r10, r8, r10
```

- **Sans renommage de registres :**
  - LD est émise à n
  - MUL,SUB, émises à n + 1
  - ADD, LD émises à n+2
  - DIV, émise à n+3
- temps total d'émission = 4 cycles
- temps exécution = 5 + 4 – 1 = 8 cycles

```
LD  hw7, (hw8)
MUL  hw1, hw7, hw2
SUB  hw17, hw4, hw5
ADD  hw9, hw17, hw8
LD  hw18, (hw12)
DIV  hw10, hw18, hw10
```

- **Avec renommage de registres :**
  - LD, SUB, LD émises à n
  - MUL, ADD, DIV émises à n + 1
- temps total d'émission = 2 cycles
- temps exécution = 5 + 2 – 1 = 6 cycles

Renommage de registres

Combien de registres matériels sont requis pour permettre au renommage de registres de rompre toutes les dépendances EAL et EAE du jeu d'instruction suivant ?

```
LD r1, (r2)
ADD r3, r4, r1
SUB r4, r5, r6
MUL r7, r4, r8
OR r8, r9, r10
SUB r11, r8, r12
DIV r12, r13, r14
ST (r15), r12
```

Renommage de registres

Le fragment de code utilise 15 registres architecturaux. En outre, il existe trois dépendances EAL entre :

ADD r3, **r4**, r1 et SUB **r4**, r5, r6,

MUL r7, r4, **r8** et OR **r8**, r9, r10,

SUB r11, r8, **r12** et DIV **r12**, r13, r14.

Il n'y a pas de dépendance EAE dans ce code.

*En conséquence, **18 registres matériels sont nécessaires** pour que le renommage des registres puisse rompre toutes les dépendances du programme :*

15 pour les 15 registres architecturaux

*et **3 pour renommer chacun des registres concernés par les dépendances EAL).***

Déroulage de boucle

- Le déroulage est une technique classique d'optimisation de boucle.
- Le principe est de répliquer le corps de la boucle de manière à :
  - diminuer le coût de gestion de la boucle
  - et d'en augmenter le parallélisme d'instructions potentiellement exploitable.

Déroulage de boucle

Exemple 1 :

Boucle originale

```
pour (i=0; i<100; i++){
        a[i] = b[i] + c[i] ;
}
```

Boucle déroulée

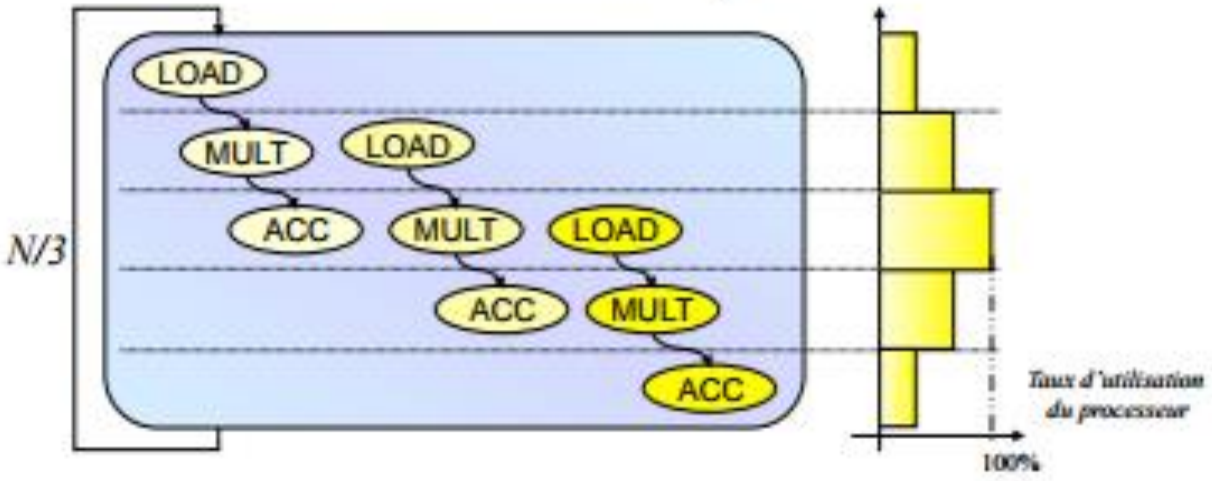
```
pour (i=0; i<100; i+=2){
        a[i] = b[i] + c[i] ;
        a[i+1] = b[i+1] + c[i+1] ;
}
```

Déroulage de boucle

- Déroulage des boucles : augmente l'ILP

```
For(i=0;i<N;i++)
{
  ACC=ACC + x[i].h[i]
}
```

```
For(i=0;i<N;i+=3)
{
  ACC=ACC + x[i].h[i]
  ACC=ACC + x[i+1].h[i+1]
  ACC=ACC + x[i+2].h[i+2]
}
```



Déroulage de boucle (N impair)


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

```
for(int i=0; i+1<n; i=i+2){
      a[i] = 2 * b[i] + c[i]
      a[i+1] = 2 * b[i+1] + c[i+1]
}
if(n est impair) a[n-1] = 2 * b[n-1] + c[n-1]
```

```
        ...
DO I = 1, N
  A[I] = B[I]*C[I];
ENDDO
        ...
```

```
        ...
DO I = 1, N, 2
  A[I] = B[I]*C[I];
  A[I+1] = B[I+1]*C[I+1];
ENDDO
IF N MOD 2 = 1
  THEN A[N]=B[N]*C[N];
ENDIF
        ...
```

Déroulage de boucle

Exemple 2 : Déroulement de boucle inégal

Boucle originale

```
pour (i=0; i<100; i++){
        a[i] =b[i] + c[i] ;
}
```

i = o .. 95 ⇔

i = 96, 97, 98, 99 ⇔

Boucle déroulée

```
pour (i=0; i<100; i+=8){
        a[i] = b[i] + c[i] ;
        a[i+1] = b[i+1] + c[i+1] ;
        a[i+2] = b[i+2] + c[i+2] ;
        a[i+3] = b[i+3] + c[i+3] ;
        a[i+4] = b[i+4] + c[i+4] ;
        a[i+5] = b[i+5] + c[i+5] ;
        a[i+6] = b[i+6] + c[i+6] ;
        a[i+7] = b[i+7] + c[i+7] ;
 }
pour ((i=100/8)x8; i<100;i++){
        a[i] = b[i] + c[i] ;
}
```

Pipeline logiciel

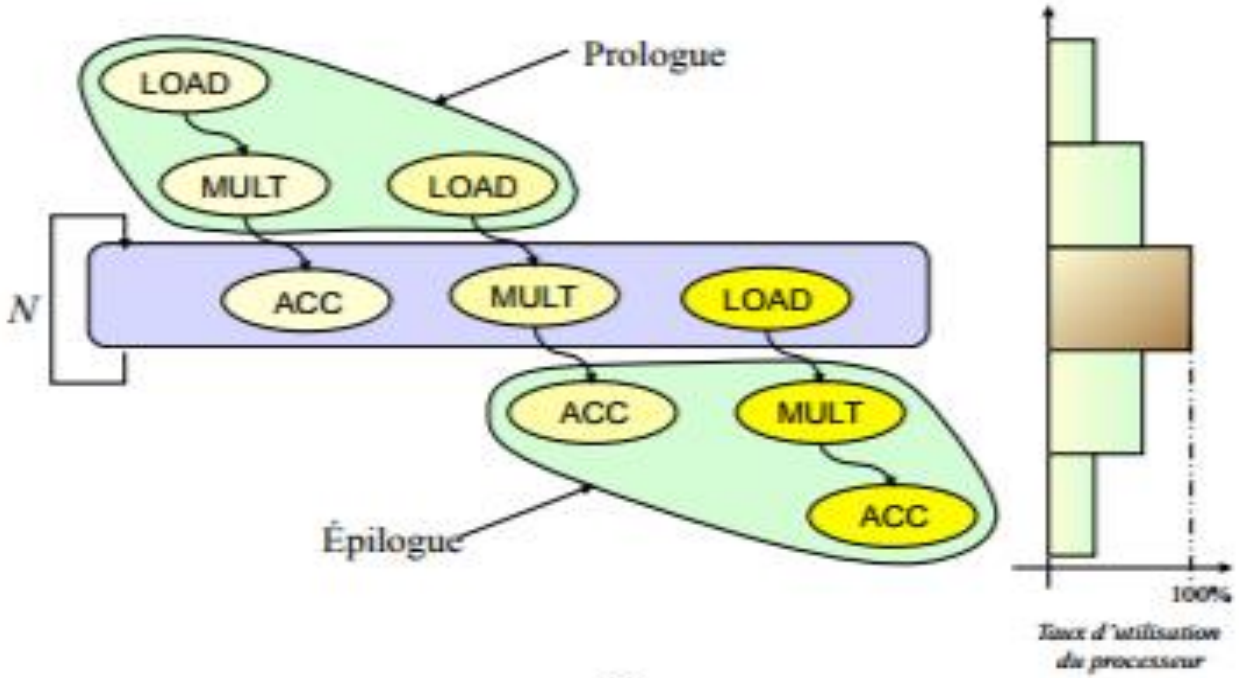
- **Le déroulage de boucle** a ses limites :
  - Approcher des performances optimales $\Rightarrow$ facteurs de déroulage élevés
    - Taille de code trop importante pour être acceptable
- **Recouvrir l'exécution de plusieurs itérations**
  - Mais sans dérouler la boucle !
  - C'est ce qu'on appelle le pipeline logiciel
- **Fonctionnement**
  - A un instant donné, on aura plusieurs itérations en cours d'excution par la machine
  - On commencera éventuellement l'exécution de l'itération i+1 avant que les opérations de l'itération i n'aient été achevées
- **Mise oeuvre**
  - Il faut trouver un motif de calcul couvrant plusieurs itérations et dont l'exécution peut être répétée.

Pipeline logiciel

- Le **pipeline logiciel** est une technique utilisée par les compilateurs pour optimiser l'exécution des boucles. Elle consiste à superposer l'exécution de plusieurs itérations de la boucle, comme un pipeline matériel le fait pour les différentes phases des instructions : une itération du code généré contiendra en fait des instructions provenant de plusieurs itérations d'origine.
- Le but est de réorganiser les instructions de manière à pouvoir exécuter des instructions provenant d'autres itérations pendant que des instructions coûteuses s'exécutent.
- Contrairement au déroulage de boucle, le compilateur cherche à optimiser les interactions entre plusieurs itérations de la boucle compilée, là où le déroulage ne permet que des optimisations locales après avoir dupliqué le code, qui sont limitées par le nombre de duplications.

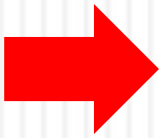
Pipeline logiciel

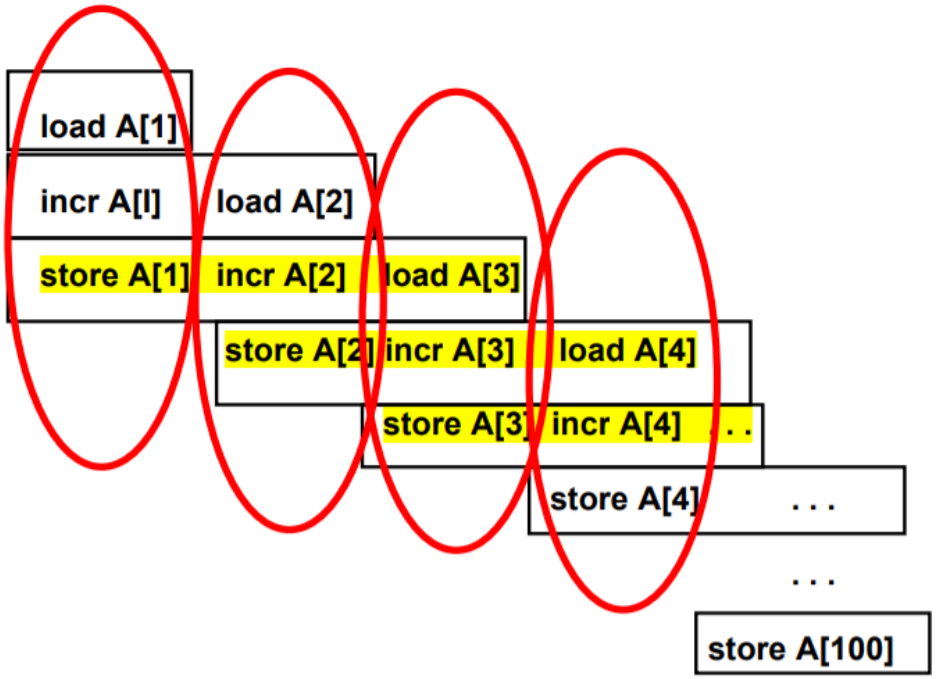
- Pipeline logiciel : ILP maximum
  - Optimisation du code assembleur



Exemple 1


```
for (i=1, i<100, i++) {
        x := A[i];
        x := x+1;
        A[i] := x
        }
```





Exemple 1

Maintenant groupons ces instructions horizontalement telles qu'elles sont visualisées dans les rectangles.
Après ces 3 premières instructions, la boucle peut être restructurée comme suit :

```
for (i=1, i<98, i++) {
        store A[i]
        incr A[i+1]
        load A[i+2]
}
```

Ces instructions peuvent être exécutées en parallèle
Pas d'aléas de données

A compléter par les instructions manquantes

Exemple 2

Unrolled 3 times

```
1   LD    F0,0(R1)
2   ADDD  F4,F0,F2
3   SD    0(R1),F4
4   LD    F6,-8(R1)
5   ADDD  F8,F6,F2
6   SD    -8(R1),F8
7   LD    F10,-16(R1)
8   ADDD  F12,F10,F2
9   SD    -16(R1),F12
10  SUBI  R1,R1,#24
11  BNEZ  R1,LOOP
```

Software Pipelined

```
    LD    F0,0(R1)
    ADDD  F4,F0,F2
    LD    F0,-8(R1)
1   SD    0(R1),F4;    Stores M[i]
2   ADDD  F4,F0,F2;    Adds to M[i-1]
3   LD    F0,-16(R1);  loads M[i-2]
4   SUBI  R1,R1,#8
5   BNEZ  R1,LOOP
    SD    0(R1),F4
    ADDD  F4,F0,F2
    SD    -8(R1),F4
```

Exercice

I. Quelle est la plus longue chaîne d'opérations dépendantes du fragment de programme suivant ?

```
LD r7, (r8)
SUB r10, r11, r12
MUL r13, r7, r11
ST (r9), r13
ADD r13, r2, r1
LD r5, (r6)
SUB r3, r4, r5
```

Réponse

La chaîne de dépendances la plus longue fait quatre instructions de long.

```
LD r7, (r8)
MUL r13, r7, r11
ST (r9), r13
ADD r13, r2, r1
```

Exercice

II. Si le fragment de code de la question I était exécuté sur un processeur superscalaire doté d'un nombre infini d'unités d'exécution avec une latence d'un cycle pour toutes les opérations, combien de temps faudrait-il pour émettre toutes les instructions de la séquence ? La question peut aussi se formuler ainsi : quelles limitations les dépendances du fragment de programme apportent-elles au temps d'émission de ses instructions ?

Réponse

Avec un nombre infini d'unités d'exécution, la capacité du processeur à émettre les instructions en parallèle n'est plus limitée que par la longueur des chaînes d'instructions dépendantes du programme.

La plus longue chaîne de dépendances a été identifiée dans la question précédente et la deuxième chaîne la plus longue est une chaîne de deux instructions.

Si toutes les dépendances dans la plus longue chaîne étaient des dépendances LAE, les instructions dans la chaîne devraient être émises en séquence, ce qui donnerait un temps d'émission de quatre cycles.

Toutefois, l'une des dépendances correspond à une dépendance EAL, or les instructions entretenant une dépendance EAL peuvent être émises durant le même cycle. Ceci permet aux instructions ST (r9), r13 et ADD r13, r2, r1 d'être émises au même moment et réduit le temps d'émission total à 3 cycles


THANK YOU