Optimisation de code : Ordonnancement de code & déroulage de boucle

ARCHI 1

Karine Heydemann
karine.heydemann@lip6.fr

Les transparents sont accessibles là:
`http://www-soc.lip6.fr/~heydeman/`

Pipeline

Principe

- Le traitement nécessaire à l'exécution d'une instruction est découpé en étapes
- Un étage de pipeline par étape. 1 étage = 1 cycle et cycle = temps de l'étage le plus long
- Une instruction commence son exécution à chaque cycle : on n'attend pas la fin de l'instruction suivante

Sémantique

- Les étages fonctionnent en parallèle
- Les instructions entrent séquentiellement dedans, conforme à l'ordre d'exécution dicté par le programme
- Les instructions finissent dans le même ordre que celui d'entrée car ordre conservé pour toutes les phases

Idéalement, une instruction finit son éxécution à chaque cycle mais en pratique problème de dépendances.

Dépendances entre instruction

Définition

Deux instructions sont dépendantes si l'une doit être exécutée avant l'autre pour maintenir l'exactitude du programme.

Type des dépendance

- Dépendance de données : opérandes communs entre les 2 instructions.
- Dépendance de contrôle : une instruction est un branchement, l'exécution de l'instruction suivante dépend du résultat du branchement.

Dépendances de données

Définition

- RAW : Read After Write
  $i_1 \rightarrow_{RAW} i_2$
  i_1 écrit Rx et i_2 lit dans Rx
  $\equiv$ utilisation d'un résultat
- WAW : Write After Write
  $i_1 \rightarrow_{WAW} i_2$
  i_1 écrit Rx et i_2 écrit dans Rx
  $\equiv$ réutilisation d'un reg.
- WAR : Write After Read :
  $i_1 \rightarrow_{WAR} i_2$
  i_1 lit Rx et i_2 écrit dans Rx
  $\equiv$ réutilisation d'un reg.

Exemples

```
lw   $2, 0($4)
addi $5, $2, 10
```

```
lw $2, 0($4)
...
addi $2, $7, 10
```

```
sw $2, 0($4)
...
addi $2, $12, 10
```

```
lw $2, 0($4)
...
addi $4, $12, 10
```

Si on inverse les instructions d'un couple dans le code, on change la sémantique du programme

Pour réussir ce cours, il faut connaître les registres lus et écrits par toutes les instructions MIPS !

Dépendances de contrôle

- L'exécution de i_2 dépend du résultat du branchement i_1

```
bne R0, R6, loop
nop
add R2, R3, R4
```

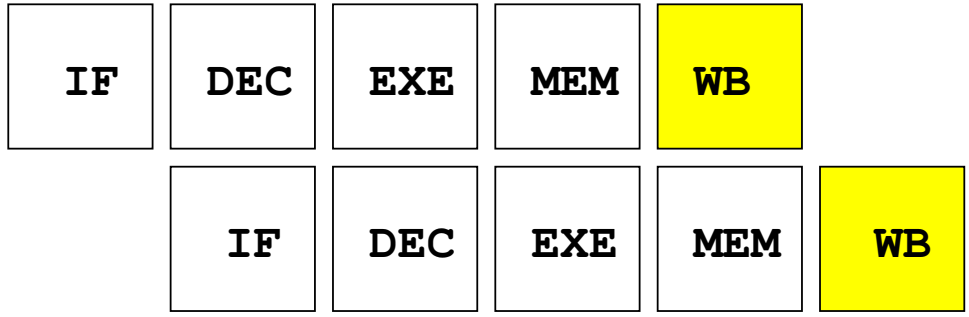
- Delayed slot après chaque branchement : évite d'avoir à annuler l'instruction entrée dans le pipeline avant de connaitre le résultat du branchement
- Les dépendances de controle n'entrainent pas de cycle de gel mais limite la performance (si delayed slot remplit avec NOP)

Dépendances de données et aléas dans le pipeline

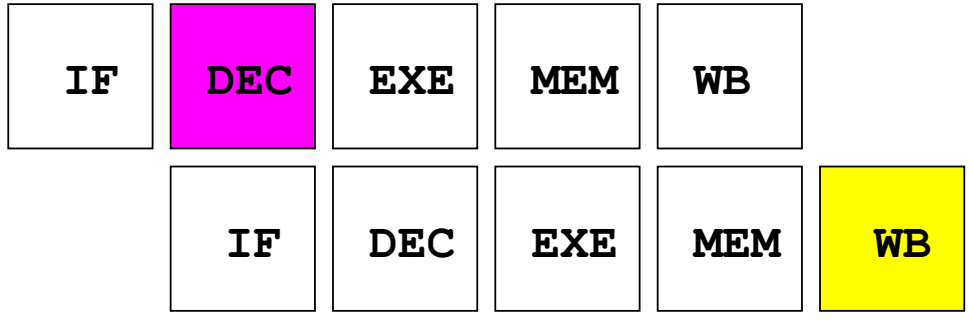
- Les aléas de données surviennent si l'exécution peut changer l'ordre des accès en lecture/écriture de registres correspondant à des opérandes
- Le pipeline doit être gelé pour maintenir une exécution correcte
- MIPS32 : la lecture des opérandes sources dans le banc de registres se fait à l'étage DECODE
- MIPS32 : l'écriture du résultat dans le banc de registre se fait à l'étage WB

Dépendances de données et aléas dans le pipeline

- WAW ou Write After Write correspond à l'écriture de résultat dans un même registre : les écritures ont lieu dans l'ordre du programme, pas de problème dans le pipeline

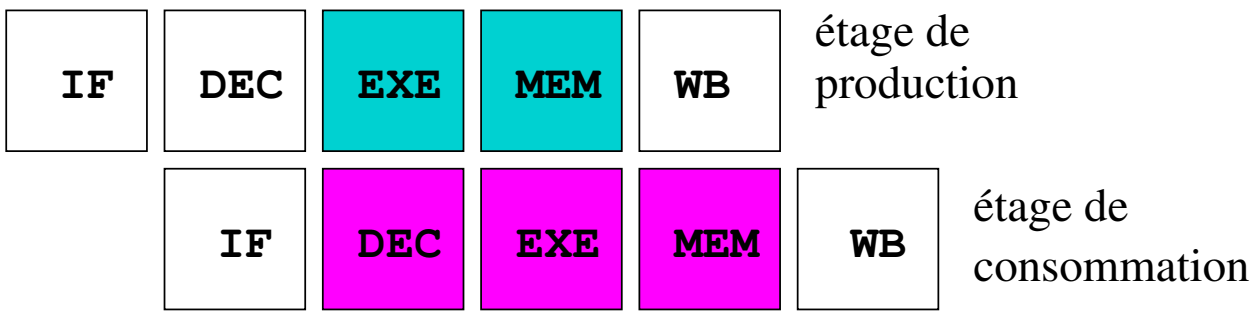


- WAR ou Write After Read traduit l'écriture dans un registre lu par une instruction précédente : la lecture a lieu en DEC donc bien avant que l'instruction dépendante n'écrive son résultat en WB : cette dépendance ne pose pas de problème dans le pipeline



Dépendances de données et aléas dans le pipeline

- RAW ou Read After Write traduit l'utilisation d'un résultat, l'instruction dépendante i2 lit le résultat d'une autre instruction i1 ; il faut que celui-ci soit disponible sinon l'exécution ne sera pas correcte
- Si l'opérande n'est pas disponible dans le dernier étage où il peut être récupéré (peut être différent de l'étage où il est consommé), l'instruction est bloquée en entrée de cet étage et tout le pipeline est bloqué en amont
- Peut induire des cycles de gel dans le pipeline



Dépendance de données et pipeline

- Opération ALU : `opcod Rd, Rs, Rt`
  Production (`Rd`) à la fin de EXE, Consommation (`Rs`, `Rt`) au début de l'étage EXE

```
add $2, $4, $3          IF DEC EXE MEM WB
addi $5, $2, 10             ?????
```

- LOAD : `lw Rt, Imm(Rs)`
  Produit son résultat (`Rt`) à la fin de l'étage MEM, add consomme ses opérandes (`Rs` + `Imm`) à l'étage EXE

```
lw $2, 0($4)          IF DEC EXE MEM WB
addi $5, $2, 10           ?????
```

Dépendance de données et pipeline

- Opération ALU : `opcod Rd, Rs, Rt`
  Production (`Rd`) à la fin de EXE, Consommation (`Rs`, `Rt`) au début de l'étage EXE

```
add $2, $4, $3          IF DEC EXE |MEM WB
addi $5, $2, 10            IF  DEC |EXE MEM WB
```

  Il existe deux bypass (`Rs` et `Rt`) EXE $\rightarrow$ EXE : pas de cycle de gel

- LOAD : `lw Rt, Imm(Rs)`
  Produit son résultat (`Rt`) à la fin de l'étage MEM, add consomme ses opérandes (`Rs` + `Imm`) à l'étage EXE

```
lw $2, 0($4)          IF DEC EXE MEM |WB
addi $5, $2, 10          IF  DEC  O  |EXE MEM WB
```

  Il y a deux bypass (`Rs` et `Rt`) MEM $\rightarrow$ EXE : 1 cycle de gel pour obtenir l'opérande

Dépendance de données et pipeline (2)

Branchement : `beq Rs, Rt, etiq`
Consommation de `Rs` et `Rt` au début de DEC

```
add $2, $4, $3        IF   DEC   EXE MEM   WB
beq $5, $2, loop            ????
```

```
lw  $2, 0($3)       IF DEC EXE MEM |WB
beq $5, $2, loop          ????
```

Dépendance de données et pipeline (2)

Branchement : `beq Rs, Rt, etiq`
Consommation de `Rs` et `Rt` au début de DEC

```
add $2, $4, $3        IF  DEC  EXE |MEM  WB
beq $5, $2, loop          IF    0  |DEC  EXE  MEM  WB
```

Il y a des bypass (`Rs` et `Rt`) EXE $\rightarrow$ DEC : 1 cycle de gel pour obtenir l'opérande

```
lw  $2, 0($3)       IF DEC EXE MEM |WB
beq $5, $2, loop       IF   0   0  |DEC EXE MEM WB
```

Il y a des bypass (`Rs` et `Rt`) MEM $\rightarrow$ DEC : 2 cycles de gel pour obtenir l'opérande

Dépendance de données et pipeline (3)

STORE : `sw Rt, Imm (Rs)`
Consommation `Rs` en EXE, `Rt` en MEM

```
addi $2, $3, 1          IF  DEC  EXE  MEM  WB
sw   $2, 0($4)                ??????
```

```
lw   $2, 0($3)        IF DEC EXE MEM WB
sw   $2, 0($4)           ??????
```

Dépendance de données et pipeline (3)

- Bypass nécessite des multiplexeurs et logique pour le contrôle de ceux-ci
- Criticité des étages : (MEM, IF, DEC, EXE)
- L'étage mémoire est le plus critique
- Cela allongerait le temps de cycle d'en mettre un en MEM, utilisé que pour les STORE
- Pas de bypass dans l'étage M
- Un STORE doit récupérer l'opérande à écrire en mémoire en DEC ou EXE
- EXE est donc le dernier étage pour récupérer la valeur à écrire en mémoire, même si opérande consommé en MEM !

Dépendance de données et pipeline (3)

STORE : `sw Rt, Imm (Rs)`
Consommation `Rs` en EXE, `Rt` en MEM

```
addi $2, $3, 1          IF   DEC   EXE  |MEM   WB
sw   $2, 0($4)               IF    DEC  |EXE   MEM WC
```

Bypass EXE $\rightarrow$ EXE (pour `Rt`)

```
lw   $2, 0($3)          IF DEC EXE MEM  |WB
sw   $2, 0($4)             IF   DEC   0   |EXE MEM WB
```

Bypass MEM $\rightarrow$ EXE (pour `Rt`) : 1 cycle de gel

Attention aux STORE !!!

Test

```
lw   $4, 0($3)          IF   DEC   EXE   MEM   WB
lw   $2, 0($4)               IF    ???   ???   ???
```

Test

```
lw   $4, 0($3)          IF   DEC   EXE MEM |WB
lw   $2, 0($4)               IF    DEC   0   |EXE   MEM WB
```

Les adresses sont calculées en EXE...

bypass MEM $\rightarrow$ EXE (pour `Rs` du (pour `sw`))

Exemple de code

Code C :

```
int a[size];
...
for (i = 0; i != size; i++)
  a[i] = 2 * a[i];
```

Code ASM compilé :

```
# i dans R8
# size dans R6, a[] dans R5

    xor R8, R8, R8
    beq R6, R0, suite
     sll R9, R6, 2    # size * 4
    add R9, R9, R5 # @a[size]

loop:
    lw R4, 0(R5)    # R4 <- *tab
     sll R7, R4, 1   # R7 * 2
    sw R7, 0(R5)    # tab[i] <- R7
    addiu R5, R5, 4 # tab++
    bne R9, R5, loop
```

Code ASM exécutable sur MIPS32 ?

Exemple de code

Code ASM compilé :

```
# i  dans R8
# size  dans R6, a[]  dans R5

   xor  R8, R8, R8
   beq R6, R0, suite
    sll   R9, R6, 2    # size * 4
   add R9, R9, R5 # @a[size]

loop:
   lw  R4, 0(R5)
    sll   R7, R4, 1
   sw R7, 0(R5)
   addiu R5, R5, 4
   bne R9, R5, loop
```

Code ASM pour MIPS32 :

```
    # i  dans R8
    # size  dans R6, a[]  dans R5
     xor  R8, R8, R8
     beq R6, R0, suite
     NOP       # delayed slot
      sll   R9, R6, 2    # size * 4
     add R9, R9, R5 # @a[size]

loop:
      lw  R4, 0(R5)
       sll   R7, R4, 1
      sw R7, 0(R5)
      addiu R5, R5, 4
      bne R9, R5, loop
      NOP       # delayed slot
```

Analyse du code (au tabelau)

- Schéma simplifié pour connaitre nombre de cycles pour 1 itération
- Calcul du CPI
- Calcul du CPI utile

- Combien de cycles de gel ? Peut on faire mieux ?

Réordonnancement des instructions

- Recouvrir les délais entre instructions dépendantes par l'exécution d'instructions indépendantes
- On peut changer l'ordre des instructions TANT QUE l'on respecte TOUTES les dépendances (RAW, WAR, WAW)
- On peut aussi changer l'ordre des instructions pour limiter les conflits de ressources (délais dus à des conflits pas aux dépendances cf. MIPS SS2)

- Exemple au tableau

Réordonnancement

- Permet de réduire le temps d'exécution/limiter les cycles de gels
- Importance aussi de limiter les delayed slots vides (éviter les nops)

```
# R5 = @tab[i], R9=@tab[size]
loop:
 lw  R4, 0(R5)
  sll  R7, R4, 1
 sw R7, 0(R5)
 addiu R5, R5, 4
 bne R9, R5, loop
 nop
```

- 1 cycle de gel entre le `lw` et `sll`
- 1 cycle de gel entre `addiu` et `bne`
- 1 `nop` inutile
- 6 instructions + 2 cycles de gel = 8 cycles/itération pour 5 instructions utiles

Réordonnancement

- Réordonnancement pour éliminer les cycles de gel + `nop`
- Dépendance WAR entre `addiu` et `sw` MAIS correction possible car incrémentation d'un pas constant (+4) de R5 et immédiat dans `addiu` $\rightarrow$ on peut remonter le `addiu` avant le `sw` (et ajuster immédiat sdu `sw`)
- On peut mettre le `sw` dans le delayed slot

```
# R5 = @tab[i], R9=@tab[size]
loop:
 lw R4, 0(R5)
  sll  R7, R4, 1
 sw R7, 0(R5)
 addiu R5, R5, 4
 bne R9, R5, loop
 nop

# 6 instructions
# 2 cycles de gel, 1 nop
```

```
# R5 = @tab[i], R8=@tab[N]
loop:
 lw R4, 0(R5)
 addiu R5, R5, 4
  sll  R7, R4, 1
 bne R9, R5, loop
 sw R7, -4(R5)
#5 instructions : 0 cycle de gel,
        0 nop
#5 instructions : 2 pour gestion
      de la boucle, 3 pour corps !
```

Le déroulage de boucle

- On peut diminuer le surcoût d'une boucle en la déroulant : partage du coût entre plusieurs itérations
- Le corps de boucle devient plus gros : plus d'opportunités de réordonnancement
- Déroulage d'un facteur `u` = la boucle déroulée `u` fois effectue `u` itérations de la boucle d'origine
- Illustration à haut niveau :

Boucle initiale :

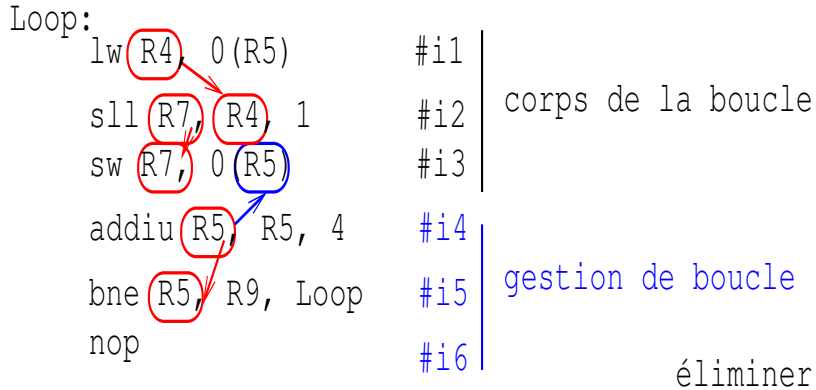
```
for(i = 0 ; i<N ; i++)
  tab[i] = tab[i]*2;
```

Boucle déroulée 2 fois
+ boucle de reste :

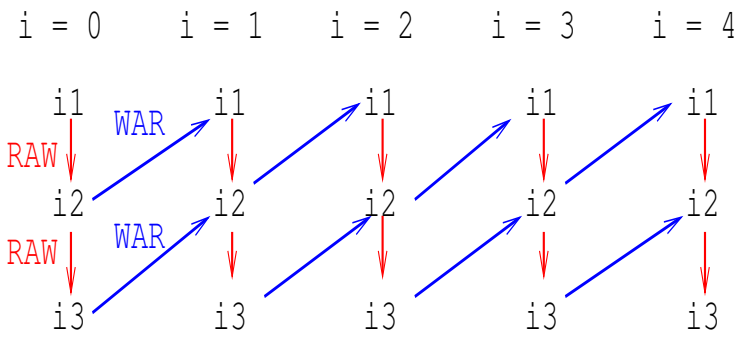
```
for(i = 0 ; i+1<N ; i+=2){
   tab[i] = tab[i]*2;
   tab[i+1] = tab[i+1]*2;
 }

 for(     ; i< N; i++)
   tab[i] = tab[i]*2;
```

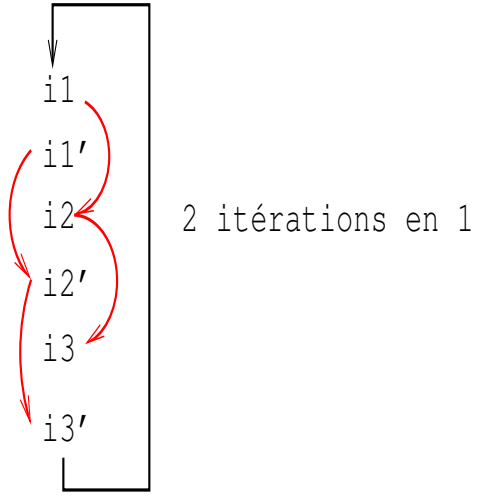
Le déroulage de boucle à bas niveau

Déroulage de boucle : autre vision



Déroulage : renommage de registres pour
éliminer les dépendances WAR entre 2 (ou plus) itérations
pour entrelacer les instructions des itérations





Optimisation et flot de contrôle

```
# R5 = @tab[i], R7=@tab[N]
loop:
        lw  R8, 0(R5)       # lecture tab[i]
        bgez R8, endif      # si tab[i] positif ou nul suite a endif
        nop
        sub R9, R0, R8      # R9 = -R8
        sw  R9, 0(R5)       # tab[i] = -tab[i]
endif:
        addiu R5, R5, 4     # @tab[i]++ adresse element suivant
        bne R7, R5, loop    # si @tab[i] != @tab[N] boucler
        nop
```

- branchement dans la boucle... traduit un if...then
- problème pour ordonnancement ?
- problème pour le déroulage de boucle ?
- Besoin d'analyser le flot de contrôle...

Bloc de base : définition et calcul

Définition

- Bloc de base : séquence d'instructions comportant un seul point d'entrée (1er inst) et un seul point de sortie (dernière) : si la 1ère est exécutée, elles le sont toutes

Détermination des BB

1. Déterminer les entêtes
   - première instruction d'une fonction/d'un bout de code
   - instruction qui suit le (dernier) delayed slot après un saut
   - instruction cible d'un saut
2. Déterminer les BBs linéairement : d'une entête à la suivante exclue.

CFG : graphe de contrôle de flot

- Les liens entre les BB traduisent le flot de contrôle du programme
- CFG : graphe reflétant le flot de contrôle
- Il y a un arc entre deux blocs BB1 et BB2 si
  - il y a un saut de BB1 vers BB2
  - si BB2 suit BB1 dans le l'ordre du programme et BB1 ne se termine pas par un saut inconditionnel (forme j etiquette)

CFG : exemple

```
# R5 = @tab[i],  R7=@tab[N]

loop:———————— BB1 ————————
        lw  R8, 0(R5)
        bgez R8, endif
        nop
        ———————— BB2 ————————
        sub R9, R0, R8
        sw  R9, 0(R5)
endif:———————— BB3 ————————
        addiu R5, R5, 4
        bne R7, R5, loop
        nop
```

Optimisations et flot de contrôle

- Ordonnancement : attention au flot de contrôle !
- Si un seul bloc de base : par définition des BB, on peut changer l'ordre des instructions (avec respect des dépendances) sans s'occuper du flot
- Si plusieurs BB : plus complexe car il faut prendre en compte le flot de contrôle

Ordonnancement et flot de contrôle : exemple

```
# R5 = @tab[i], R7=@tab[N]

loop:———————— BB1 ————————
        lw  R8, 0(R5)
        bgez R8, endif
        nop
        ———————— BB2 ————————
        sub R9, R0, R8
        sw  R9, 0(R5)
endif:———————— BB3 ————————
        addiu R5, R5, 4
        bne R7, R5, loop
        nop
```

- BB1 et BB3 toujours exécutés : ok de descendre des instructions de BB1 dans BB3, ou remonter des instructions de BB3 vers BB1 (si on respecte les dépendances)
- BB2 pas forcément exécuté : on ne peut descendre une instruction de BB1 vers BB2, ni remonter une instruction de BB3 dans BB2 sinon pas d'exécution possible !
- On peut remonter des instructions de BB2 vers BB2, descendre des instructions de BB2 vers BB3 si on s'assure que cela ne change pas la sémantique d'une itération...

Déroulage de boucle avec du contrôle interne

```
# R5 = @tab[i],  R7=@tab[N]

loop:————————— BB1 —————————
        lw  R8, 0(R5)
        bgez R8, endif
        nop
        ————————— BB2 —————————
        sub R9, R0, R8
        sw  R9, 0(R5)
endif:————————— BB3 —————————
        addiu R5, R5, 4
        bne R7, R5, loop
        nop
```

- Comment dérouler la boucle ?
- Deux chemins d'exécution possibles par itération
- Le déroulage de boucle doit conserver les différents combinaisons possibles des chemins dans le corps de la boucle déroulée !
- Copie des étiquettes, sérialisation d'une partie des traitements...
- Exemple au tableau

Déroulage de boucle et (in)dépendance des itérations

```
for (i = 0 ; i < N ; i++)
  tab[i] = tab2[i] + tab[i];
```

```
for (i = 0; i < N ; i++)
  tab[i] = tab[i−1]+ 2∗tab[i] + tab[i+1];
```

```
for (i = 0; i+1 < N ; i += 2) {
  tab[i]   = tab2[i]   + tab[i];
  tab[i+1] = tab2[i+1] + tab[i+1];
}
```

```
for (i = 0 ; i+2 < N ; i += 3) {
   tab[i]   = tab[i−1] + 2∗tab[i]   + tab[i+1];
   tab[i+1] = tab[i]   + 2∗tab[i+1] + tab[i+2];
   tab[i+2] = tab[i+1] + 2∗tab[i+2] + tab[i+3];
}
```

- Indépendance des itérations versus dépendance
- Duplication et entrelacement versus réutilisation des calculs / correction de la version déroulée
- Le déroulage de boucle doit assurer l'utilisation des bonnes valeurs (tab[i-1] modifié, tab[i] et tab[i+1] non modifiés !)
- Pas de mise en parallèle des traitements mais "pipeline" et réutilisation des valeurs disponibles (nouvelle valeur de tab[i-1] par exemple)
- Exemple au tableau