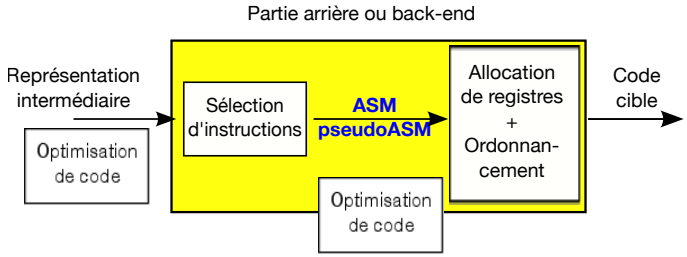
Ordonnancement de code

UE Compilation Avancée
Cours 2.

Karine Heydemann

Contenu du cours 2

Génération de code, allocation de registre et ordonnancement



- La génération de code produit une suite d'instructions (ou pseudo-instr) dans un ordre qui vérifie simplement la sémantique du programme
- Puis étapes clé dépendantes de l'architecture cible
  - L'allocation de registres qui assigne un registre de la cible aux opérandes source et destination des instructions
  - Ordonnancement des instructions qui consiste à changer l'ordre des instructions pour mieux exploiter le matériel disponible

Dépendances entre instruction

- Deux instructions sont dépendantes si l'une doit être exécutée avant l'autre pour maintenir l'exactitude du programme
- 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
- Conflit entre les instructions : deux instructions ont besoin de la même ressource simultanément
- $\Rightarrow$ les conflits dépendent de l'architecture

Classification des dépendances de données

- Read After Write ou RAW ou vraie dépendance : traduit l'utilisation d'un résultat
- Write After Write ou WAW ou dépendance de sortie : traduit la réutilisation d'un même opérande pour un résultat, dépendance de nom
- Write After Read ou WAR ou anti-dépendance : traduit l'utilisation d'un opérande source d'une instruction précédente pour un résultat
- Deux instructions d'accès mémoire sont considérées comme dépendantes si l'on ne peut pas garantir que les adresses accédées sont différentes et ne se recouvrent pas
- Remarque : pour déterminer les dépendances entre les instructions, il faut connaître le sens des opérandes des instructions ou pseudo-instructions

Calcul de dépendances entre deux instructions

- A un sens si dest(i), src1(i), src2(i) existent et sont des registres
- i1 $\rightarrow_{WAW}$ i2
  si dest(i1) = dest(i2) ET dest(i1) $\neq$ r0
- On considère i1 $\rightarrow_{MEM}$ i2
  Si i2 = opmem1 Rx, Imx(Ry) et i1 = opmem2 Rt, Ims(Rs)
  ET si Ry $\neq$ Rs ET pas 2 lectures
  (on ne peut rien dire sans analyse plus poussée)
- Exemples

```
lh $3, 2($4)
lw $2, 0($4)
   ...
addi $2, $12, 10
sw $12, 0($5)
sh $13, 2($4)
sw $14, -4($4)
```

```
lw $2, 0($4)
add $7, $2, $8
...
addi $2, $12, 10
```

Calcul de dépendances entre deux instructions

- A un sens si dest(i), src1(i), src2(i) existent et sont des registres (rappel : r0 vaut toujours 0) :
- i1 $\rightarrow_{RAW}$ i2 :
  Si dest(i1) = src1(i2) ET dest(i1) $\neq$ r0
  Si dest(i1) = src2(i2) ET dest(i1) $\neq$ r0
- i1 $\rightarrow_{WAR}$ i2 :
  Si dest(i2) = src1(i1) ET dest(i2) $\neq$ r0
  Si dest(i2) = src2(i1) ET dest(i2) $\neq$ r0
- Exemples :

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

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

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

Analyse des dépendances

- L'analyse des dépendances est vitale pour l'ordonnancement
- L'analyse peut inclure celle des dépendances de contrôles (selon code)
- L'analyse des dépendances détermine la relation d'ordre entre les instructions du programme
- Cet ordre (partiel) doit être respecté pour exécution correcte du code

Graphe de dépendances

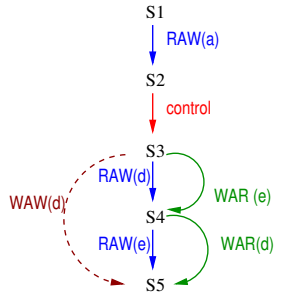
- Un graphe de dépendances représente les dépendances dans une région de code
- Les noeuds sont des *statements*, des pseudo-instructions ou des instructions en fonction du niveau d'analyse
- Les arêtes représentent les dépendances, elles sont annotées avec le type de la dépendance
- Les dépendances de contrôle sont omises sauf si c'est la seule dépendance entre 2 noeuds
- RAW + WAR = WAW $\Rightarrow$ WAW pas nécessaire

```
          a <-- b + c          S1
   if a > 10 goto L1           S2
        d <-- b * e            S3
        e <-- d + 1            S4
L1 : d <-- e/2                 S5
```

Graphe de dépendances

- Un graphe de dépendances représente les dépendances dans une région de code
- Les noeuds sont des *statements*, des pseudo-instructions ou des instructions en fonction du niveau d'analyse
- Les arêtes représentent les dépendances, elles sont annotées avec le type de la dépendance
- Les dépendances de contrôle sont omises sauf si c'est la seule dépendance entre 2 noeuds
- RAW + WAR = WAW $\Rightarrow$ WAW pas nécessaire

```
          a <-- b + c          S1
   if a > 10 goto L1           S2
        d <-- b * e            S3
        e <-- d + 1            S4
L1 : d <-- e/2                 S5
```

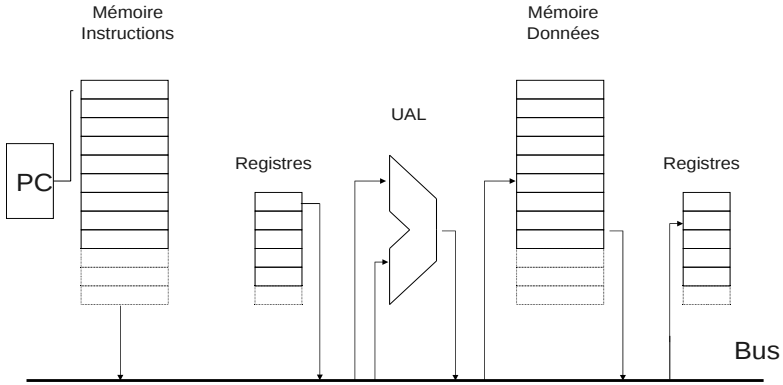


Problème des dépendances

- A l'exécution, certaines dépendances entraînent des aléas (hazards) qui engendrent des pertes de performance.
- Aléas de données lorsqu'une une instruction utilise le résultat d'une autre et attente nécessaire
- Aléas structurel lorsque deux instructions ont besoin de la même ressource simultanément, blocage de l'une des 2
- L'ordonnancement vise à éviter ces aléas ou limiter leurs effets sur les performances
- Besoin de connaître l'architecture cible...

Exécution d'une instruction

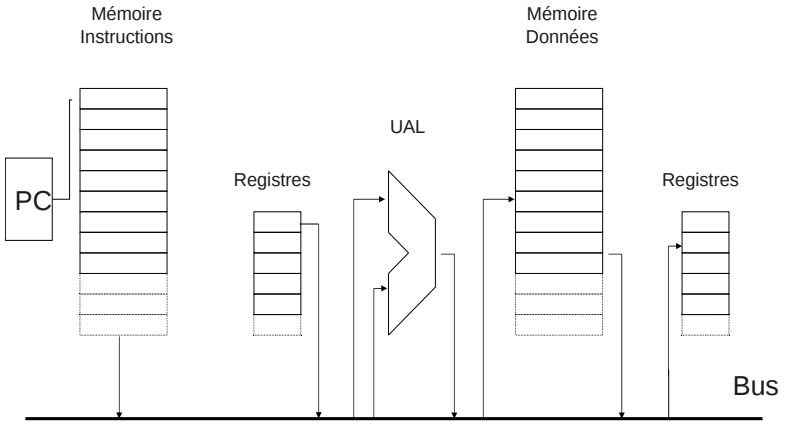
- L'exécution d'une instruction nécessite plusieurs étapes :



- Lecture de l'instruction
- Décodage / lecture des opérandes dans le banc de registres
- Réalisation du traitement demandée :
  - Exécution des opérations arithmétique et logique
  - Lecture/écriture de données en mémoire
- Ecriture du resultat dans le banc de registres
- Calcul de l'adresse de l'instruction suivante

Exécution d'une instruction

- L'exécution d'une instruction nécessite plusieurs étapes :



- A chaque cycle une seule instruction est en cours d'exécution
- Toute instruction est complètement exécutée en un cycle
- Un cycle = temps de traitement de l'instruction la plus longue à réaliser

Exécution d'une instruction

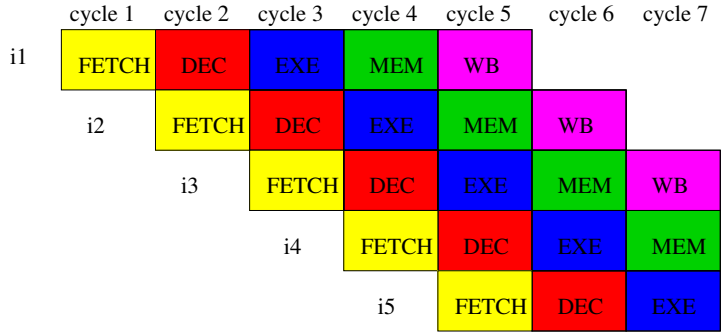
- L'exécution d'une instruction nécessite plusieurs étapes :
  - Lecture de l'instruction
  - Décodage / lecture des opérandes dans le banc de registres
  - Réalisation du traitement demandée :
    - Exécution des opérations de type arithmétique et logique
    - Lecture/écriture de données en mémoire
  - Ecriture du résultat dans le banc de registres
  - Calcul de l'adresse de l'instruction suivante
- Dépendances entre les étapes $\neq$ d'exécution d'une instruction
- Indépendances des étapes d'instructions différentes
- Démarrage d'une instruction avant la fin de l'exécution de la précédente possible
- Seul le matériel pose des problèmes de conflit : chaque étape doit disposer de son matériel

Pipeline d'instructions

- L'exécution d'une instruction peut être découpée en X étapes de longueur égale
- On peut commencer l'étape i d'une instruction lorsque la précédente commence l'étape $i + 1$
- C'est le principe du pipeline d'instructions à X étages, un étage = 1 sous étape de l'exécution
- A un instant donné, plusieurs instructions sont en cours d'exécution mais à des étages différents du pipeline
- Un cycle = temps de traversée d'un étage (importance de l'équilibre entre les étages)
- Le pipeline n'attend pas la fin d'une instruction pour démarrer la suivante

Pipeline d'instructions : exemple

- 5 étages : Fetch, Decode, Execute, Memory, Write Back
  - Fetch : lecture de l'instruction
  - Decode / calcul @ : décodage, lecture des opérandes (banc de registres), calcul de l'@ de l'instruction suivante
  - Execute : réalisation des opérations nécessitant l'ALU
  - Memory : lecture/écriture mémoire
  - Write Back : écriture du résultat dans le banc de registres



- Augmente le debit des instructions : une instruction/cycle mais temps de cycle plus petit !

Dépendances de données et aléas dans le pipeline

- Les aléas de données surviennent si l'exécution peut changer l'ordre des accès lecture/écriture de registres correspondant à des opérandes
- Le pipeline doit alors gelé pour maintenir une exécution correcte
- La lecture des opérandes sources dans le banc de registres se fait à l'étage DECODE (hyp)
- L'écriture du résultat dans le banc de registre se fait à l'étage WB (hyp)

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 lit le résultat d'une autre instruction ; il faut que celui-ci soit disponible sinon l'exécution ne sera pas correcte

Dépendances de données et aléas dans le pipeline

- Les dépendances RAW peuvent engendrer des cycles de gel dans le pipeline, l'instruction est bloquée dans l'étage où elle a besoin au plus tard de l'opérande, le pipeline gelé en amont, et ce jusqu'à ce que l'opérande soit disponible
- Exemple : lw produit son résultat à la fin de l'étage MEM, add consomme ses opérande à l'étage EXE

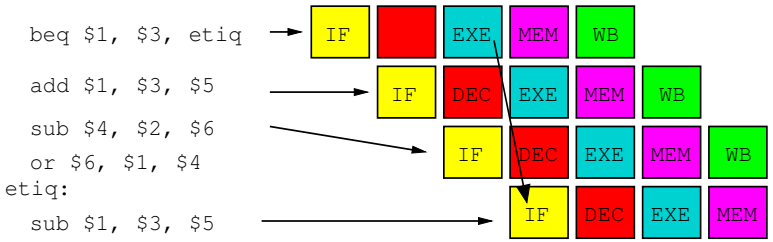
```
lw $2, 0($4)        IF  DEC  EXE  MEM  WB
addi $5, $2, 10         IF   DEC   O   EXE  MEM  WB
```

Il y a un cycle de gel dû à la dépendance. Sans gel, le résultat serait faux.

Dépendances de contrôle et aléas dans le pipeline

- L'adresse de l'instruction suivante n'est connue qu'une fois le branchement exécuté (condition évaluée et @cible calculée)

```
Supposons que le calcul de l'@ a lieu à l'étage EXE,
comme la résolution du saut (pris ou non pris)
```




```
i2 et i3 ont commencé leur exécution quand on connait
la cible du saut et s'il est pris.

Il y a donc 2 cycles de délai entre un saut et l'inst à
exécuter ensuite.
Si 20% de saut et CPI = 1 sans saut, alors CPI = 1.4 avec
Perte de 40% !
```

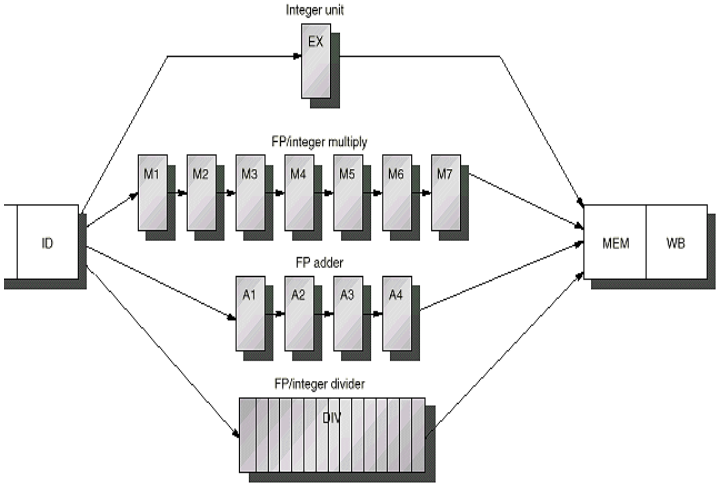
Gestion des dépendances de contrôle dans le pipeline

- Pour gérer les instructions entrées dans le pipeline lors de la résolution des saut, deux approches possibles :
  - Annuler les instructions entrées dans le pipeline en cas de saut pris
  - Considèrer N delayed slots : les N instructions qui suivent un branchement seront exécutées que le saut soit pris ou non pris → impact sur le compilateur qui doit remplir ces delayed slots avec des instructions correctes
- Les sauts limitent la performance des processeurs (delayed slots contenant des nops ou annulation des instructions) : en moyenne il y a un branchement toutes les 5 instructions
- Le code pour le processeur MIPS considéré dans ce cours a 1 delayed slot

Opérations de durée variable

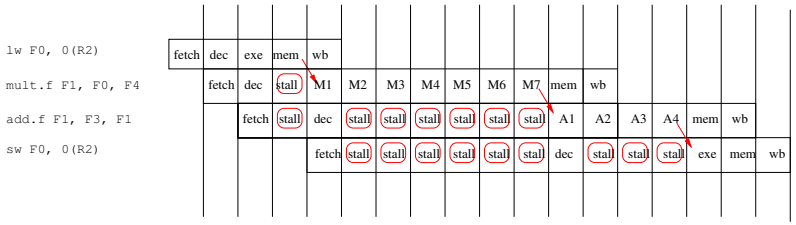
- Les opérations n'ont pas toutes la même durée : les opérations flottantes et les opérations de multiplication/division nécessitent plus d'un cycle.
- Plusieurs pipelines d'exécution dans le processeur, un pipeline par type d'opération arithmetique et logique.
- Délai : nombre de cycles minimal entre le numéro du cycle du démarrage d'une instruction qui produit un résultat et celui d'une instruction qui utilise ce résultat pour éviter des cycles de gel.
- Latence : nombre de cycles pour produire un résultat

Opérations flottantes et pipeline



- Unité entière latence de 1 cycle
- Multiplication flottante pipelinée : latence de 7 cycles
- Addition flottante pipelinée : latence 4 cycles
- Division flottante non pipelinée : latence 25 cycles

Impact des dépendances RAW : exemple



- Délai entre 2 instructions = distance entre étage de production et étage de consommation/récupération possible de l'opérande
- Dépend des instructions en jeu
- Lors d'un stall, le pipeline est gelé en amont de l'instruction bloquante
- Nombre de cycles de gel = delai - (nb instructions entre les 2 dépendantes + cycles induits par ces instructions)
- IMPACT de l'ordre des instructions sur le temps d'exécution : importance de l'ordonnancement à la compilation !

Ordonnancement des instructions

- Recouvrir les délais entre instructions dépendantes par l'exécution d'instructions indépendantes
- Trouver un ordre d'exécution des instructions qui respecte TOUTES les dépendances
- Trouver un ordre d'exécution des instructions qui limite les conflits de ressources (délais dus à des conflits pas aux dépendances)
- Ordonnancement :
  - local à un bloc de base : par définition des BB, on peut changer l'ordre des instructions modulo les dépendances/conflits sans s'occuper du flot.
  - global c-a-d entre plusieurs BB : plus complexe car il faut prendre en compte le flot de contrôle, nécessite du code de compensation/gestion.

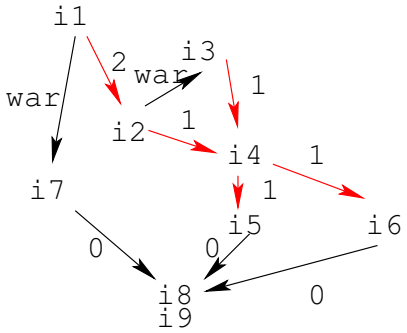
Ordonnancement local : DAG associé à un BB

- Ordonnancement local basé sur le graphe de dépendance des instructions du BB
- Graphe des dépendances sans cycle ou DAG/Directed Acyclic Graph, pas de dépendance de contrôle
- Arêtes annotées avec le type de la dépendance et le délai induit par la dépendance
- Cas d'un saut en fin de BB : dans le DAG, toutes les instructions sans successeur deviennent prédécesseurs du saut.
- Cas de delayed slots :
  - l'ordonnanceur tente de trouver des instructions utiles à mettre dedans avant la construction du DAG, si pas d'instruction, des nops (ou recherche plus complexe)
  - ces instructions sont aggrégées avec le noeud correspondant au saut dans le DAG
  - ces instructions sont ajoutées après l'ordonnancement du reste du bloc : extension du noeud du branchement contenant les instructions de ses delayed slots

DAG associé à un BB, exemple

- En supposant que les delais sont de
  - 0 pour les dépendances WAW et WAR
  - 2 pour les dépendances RAW avec une lecture mémoire
  - 1 pour les dépendances RAW sinon
- 1 cycle comme temps d'exécution pour toutes les instructions

```
i1   : lw $4, 0($2)
i2   : add  $5, $4, 10
i3   : addi $4, $11, 8
i4   : xor $8, $5, $4
i5   : sw $8, 0($3)
i6   : sw $8, 4($3)
i7   : add $2, $2, 10
i8   : j suite
i9   : nop
```



Chemin critique

- Correspond à la durée d'exécution min (longueur du chemin le plus long) des instructions d'un DAG
- Calcul du chemin critique en parcourant les noeuds triés dans un ordre topologique inverse (les noeuds sans successeur d'abord) et en leur associant un poids :

```
for all noeud(i)
    node(i).weight = 0
for all node(i) in a topological postorder
  if leaf(node(i))
    node(i).weight = ExecTime(i)
  else
    for all node(j) in succ(node(i))
      node(i).weight = max (node(i).weight,
      delai(node(i),node(j)) + node(j).weight)
criticalpath = 0
for all node(k) sans predesseur
  criticalpath = max(criticalpath, node(k).weight)
```

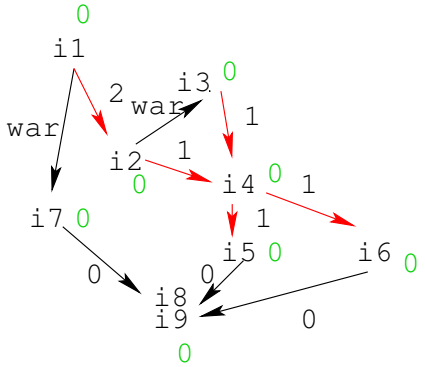
- avec delai(node(i), node(j)) = délai entre node(i) et node(j)
- avec ExecTime(i) temps d'exécution du noeud i

Chemin critique : en pratique

- Ordre topologique inverse en pratique :
  - Avec une liste de noeuds : initialisée avec les noeuds sans successeur, ajout des prédecesseurs d'un noeud traité si tous ses successeurs ont été traités
  - Avec une approche itérative cherchant un point fixe (utilisation d'un booléen) qui traite tous les noeuds dans un ordre quelconque

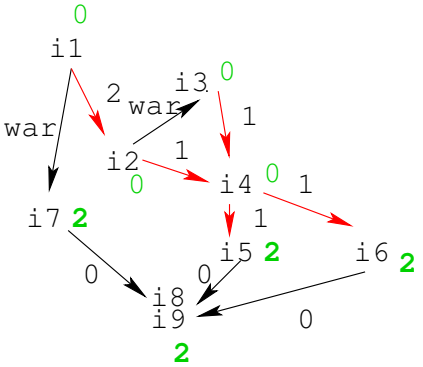
Chemin critique : exemple

```
i1  : lw $4, 0($2)
i2  : add  $5, $4, 10
i3  : addi $4, $11, 8
i4  : xor $8, $5, $4
i5  : sw $8, 0($3)
i6  : sw $8, 4($3)
i7  : add $2, $2, 10
i8  : j suite
i9  : nop
```



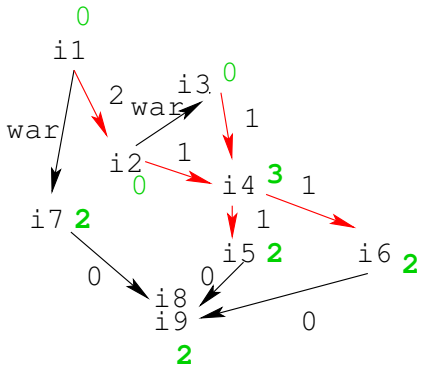
Chemin critique : exemple

```
i1  : lw $4, 0($2)
i2  : add  $5, $4, 10
i3  : addi $4, $11, 8
i4  : xor $8, $5, $4
i5  : sw $8, 0($3)
i6  : sw $8, 4($3)
i7  : add $2, $2, 10
i8  : j suite
i9  : nop
```



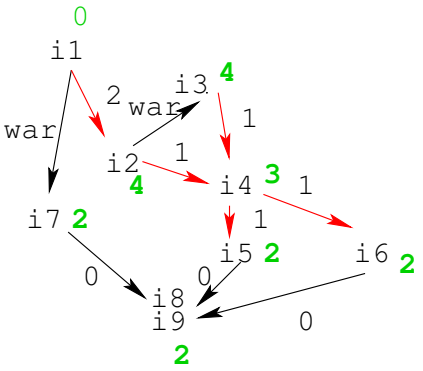
Chemin critique : exemple

```
i1  : lw $4, 0($2)
i2  : add  $5, $4, 10
i3  : addi $4, $11, 8
i4  : xor $8, $5, $4
i5  : sw $8, 0($3)
i6  : sw $8, 4($3)
i7  : add $2, $2, 10
i8  : j suite
i9  : nop
```



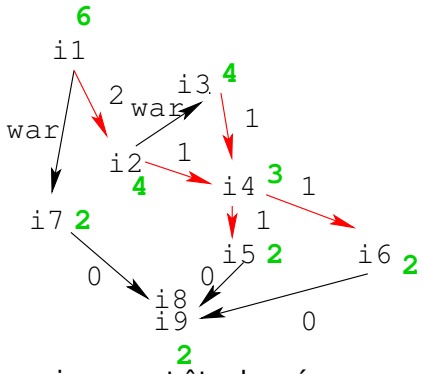
Chemin critique : exemple

```
i1  : lw $4, 0($2)
i2  : add  $5, $4, 10
i3  : addi $4, $11, 8
i4  : xor $8, $5, $4
i5  : sw $8, 0($3)
i6  : sw $8, 4($3)
i7  : add $2, $2, 10
i8  : j suite
i9  : nop
```



Chemin critique : exemple

```
i1  : lw $4, 0($2)
i2  : add  $5, $4, 10
i3  : addi $4, $11, 8
i4  : xor $8, $5, $4
i5  : sw $8, 0($3)
i6  : sw $8, 4($3)
i7  : add $2, $2, 10
i8  : j suite
i9  : nop
```



Quel que soit le nombre d'instructions qui peuvent être lancées par cycle, quel que soit l'ordonnancement des instructions, il faut 6 cycles minimum pour exécuter les instructions du BB

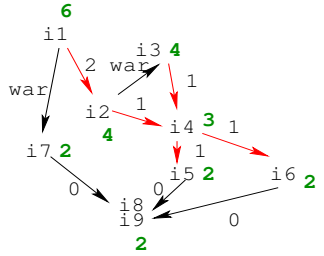
Ordonnancement par liste : principe

- Maintien d'une liste `ReadyInst` des instructions qui peuvent être ordonnancées : celles dont les prédécesseurs dans le DAG sont ordonnancés
- Choix dans cette liste d'une instruction (ou N si processeur de degré N) avec des priorités
- `ETime(i)` indiquant le cycle où l'instruction du noeud `i` peut être lancée sans causer de gel dans le pipeline, mis à jour quand un prédécesseur est ordonnancée
- `ExecTime(i)` temps d'exécution de(s) instruction(s) du noeud `i`
- Maintient du temps courant `CurTime`, commençant à 1 et mis à jour à chaque ordonnancement de N instructions (N degré du processeur) ou moins si pas autant

Ordonnancement par liste : exemple de priorité

- Exemple de priorités et ordre possible d'application
  1. Instructions qui ne causeront pas de gel du pipeline
  2. Instructions qui ont le poids le plus élevé dans calcul chemin critique
  3. Instructions qui ont le plus de successeurs
  4. Instructions qui ont la plus grande latence
  5. Instructins qui ont le plus de descendants
  6. Instruction qui apparait la première dans l'ordre d'origine
  7. ...
- Pas d'ordre optimal des priorités

Ordonnancement par liste : exemple



- processeur de degré 1
- `ExecTime(i)` vaut 1 pour toute instruction `i`
- couples `(i, Etime(i))` manipulés

CurTime = 1, ReadyInst = {(i1,1)} Sched = {}

ReadyInst = {(i2,3), (i7,1)}

ReadyInst = {(i2,3)}

ReadyInst = {(i3,3)}

ReadyInst = {(i4,5)}

ReadyInst = {(i5,5), (i6, 5)}

ReadyInst = {(i6,5)}

ReadyInst = {(i8,8)}

=> Sched = {i1} CurTime = 2

=> Sched = {i1, i7} CurTime = 3

=> Sched = {i1, i7, i2} CurTime = 4

=> Sched = {i1, i7, i2, i3} CurTime = 5

=> Sched = {i1, i7, i2, i3, i4} CurTime = 6

=> Sched = {i1, i7, i2, i3, i4, i5} CurTime = 7

=> Sched = {i1, i7, i2, i3, i4, i5, i6} CurTime = 8

=> Sched = {i1, i7, i2, i3, i4, i5, i6, i8} CurTime = 9

Ordonnancement et fausses dépendances

- Les fausses dépendances (WAR et WAW) sont des dépendances de nom pas de données
- Elles limitent les possibilités de l'ordonnanceur
- En renommant les registres concernés on peut les éliminer :
  - Utiliser un autre registre
  - Propager le nouveau nom dans ses utilisations
- Quel(s) registre(s) utiliser ?
- $\rightarrow$ un registre disponible sur toute la durée nécessaire
- Besoin d'analyser la vivacité des registres/variables
- C'est une analyse de flot de données : étude des définitions/utilisations des variables dans le code

Vivacité

- Un registre est vivant en un point du programme (entre 2 instructions) s'il existe un chemin jusqu'à la fin du programme le long duquel ce registre est utilisé avant d'être potentiellement redéfini
- Il est dit mort sinon
- On peut calculer les registres vivants en tous les points d'un programme, sur une plage du programme ou leur durée de vie
- $\Rightarrow$ besoin de connaitre les définitions et utilisations des registres

Définition et utilisations d'un registre

- Un registre est défini lorsqu'une instruction écrit dedans
- Un registre est utilisé lorsqu'il est lu par une instruction
- Un registre peut être utilisé et défini par une instruction
- Si `inst` = `add r1, r2, r3`

  `Def(inst)` =

  `Use(inst)` =
- Si inst `lw r1, 0(r1)`

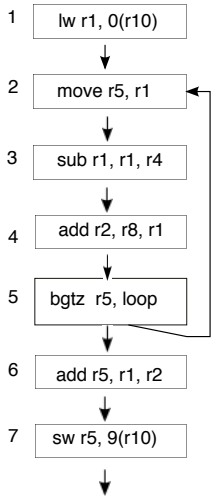
  `Def(inst)` =

  `Use(inst)` =

Registres vivants en entrée et sortie d'une instruction

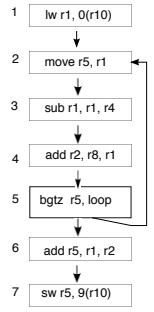
- Pour une instruction `i`, on note
  - `use(i)` = ensemble des registres lus par `i`
  - `def(i)` = ensemble des registres écrit par `i`
  - `succ(i)` = successeurs immédiats de `i`
  - $\texttt{live}_{\texttt{in}}(\texttt{i})$ = ensemble des registres vivants en entrée de `i`
  - $\texttt{live}_{\texttt{out}}(\texttt{i})$ = ensemble des registres vivants en sortie de `i`
- $\texttt{live}_{\texttt{out}}(\texttt{i}) = \bigcup_{\texttt{i'} \in \texttt{Succ(i)}} \texttt{live}_{\texttt{in}}(\texttt{i'})$
- $\texttt{live}_{\texttt{in}}(\texttt{i}) = \texttt{use(i)} \bigcup (\texttt{live}_{\texttt{out}}(\texttt{i}) \setminus \texttt{def(i)})$
- Un registre est mort en entrée/sortie de `i` s'il n'est pas dans $\texttt{live}_{\texttt{in/out}}(\texttt{i})$

Registres vivants en entrée et sortie d'une instruction



inst	def	use	succ	$live_{out}$	$live_{in}$
1	r1	r10	2		
2	r5	r1	3		
3	r1	r1, r4	4		
4	r2	r1, r8	5		
5		r5	6, 2		
6	r5	r1, r2	7		
7		r5, r10	∅	∅	

Registres vivants en entrée et sortie d'une instruction

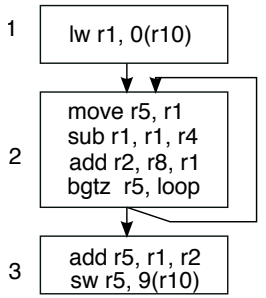


inst	def	use	succ	$live_{out}$	$live_{in}$
1	r1	r10	2	r1, r10, r8, r4	r10, r8, r4
2	r5	r1	3	r1, r10, r5, r8, r4	r1, r10, r8, r4
3	r1	r1, r4	4	r1, r10, r5, r8, r4	r1, r10, r5, r8, r4
4	r2	r1, r8	5	r2, r1, r10, r5, r8, r4	r1, r10, r5, r8, r4
5		r5	6, 2	r2, r1, r10, r8, r4	r2, r1, r10, r5, r8, r4
6	r5	r2, r1	7	r5, r10	r2, r1, r10
7		r5, r10	∅	∅	r5, r10

Registres vivants en entrée et sortie d'un bloc de base

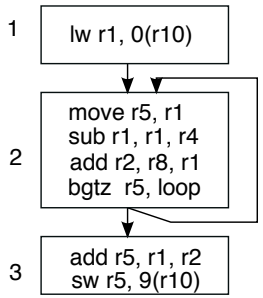
- Un BB peut être vu comme une MACRO instruction
  - `use(bb)` = ensemble des registres lus par `bb` AVANT d'être définis potentiellement dans le `bb`
  - `def(bb)` = ensemble des registres écrits par `bb`
  - `succ(bb)` = successeurs immédiats de `bb`
  - $\texttt{live}_{\texttt{in}}\texttt{(bb)}$ = ensemble des registres vivants en entrée de `bb`
  - $\texttt{live}_{\texttt{out}}\texttt{(bb)}$ = ensemble des registres vivants en sortie de `bb`
- $\texttt{live}_{\texttt{out}}\texttt{(bb)} = \bigcup_{\texttt{bb'} \in \texttt{Succ(bb)}} \texttt{live}_{\texttt{in}}\texttt{(bb')}$
- $\texttt{live}_{\texttt{in}}\texttt{(bb)} = \texttt{use(bb)} \bigcup (\texttt{live}_{\texttt{out}}\texttt{(bb)}) \setminus \texttt{def(bb)})$
- Un registre est mort en entrée/sortie de `bb` s'il n'est pas dans $\texttt{live}_{\texttt{in/out}}\texttt{(bb)}$
- Un registre peut être mort en entrée et en sortie MAIS apparaitre dans `bb`
  → `x` mort dans `bb` ssi $\texttt{x} \notin \texttt{def(x)}$ et $\texttt{x} \notin \texttt{live}_{\texttt{in}}\texttt{(bb)}$!!

Registres vivants en entrée et sortie d'un bloc



bb	def	use	succ	*live*$_{out}$	*live*$_{in}$
1	r1	r10	2		
2	r1, r2, r5	r1, r4, r8	2, 3		
3	r5	r2, r1, r10			

Registres vivants en entrée et sortie d'un bloc



bb	def	use	succ	*live*$_{out}$	*live*$_{in}$
1	r1	r10	2	r1, r10, r4, r8	r10
2	r1, r2, r5	r1, r4, r8	2, 3	r2, r1, r10, r4, r8	r1, r10, r4, r8
3	r5	r2, r1, r10		$\emptyset$	r2, r1, r10

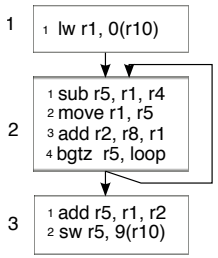
Renommage dans un bloc de base

- Soit $\texttt{defInst(bb)}$ l'ensemble des `(r,inst)` tel que `inst` définit `r` dans `bb`
- Soit $\texttt{defLive}_{\texttt{out}}\texttt{(bb)} \in \texttt{defInst(bb)}$ le sous-ensemble des `(r, inst)` avec `r` vivant en sortie provenant de la définition faite par `inst`
- On peut renommer dans `bb` de manière sure les registres `r` dans $\texttt{R} = \texttt{defInst(bb)} \setminus \texttt{defLive}_{\texttt{out}}\texttt{(bb)}$

1. Calcul des utilisations de `r` avec $\texttt{(r,inst)} \in \texttt{R}$ à ses utilisations dans `bb`
   ⇒ correspond aux dépendances RAW de l'instruction `inst` dans `bb`
2. Choisir un registre mort en entrée de `bb` et n'apparaissant pas dans `def(bb)`
3. Renommer le registre dans sa définition (`inst`) et propager dans ses utilisations

⇒ Utile pour l'ordonnancement si les utilisations ont une dépendance WAR avec une autre instruction de `bb` !

Renommage dans un bloc



bb	defInst	defLive	$live_{in}$	$live_{out}$
1	(r1,1)	(r1,1)	r4, r8, r10	r1, r4, r8, r10
2	(r5,1) (r1,2) (r2,3)	(r1, 2) (r2, 3)	r1, r4, r8, r10	r1, r2, r4, r8, r10
3	(r5,1)		r1, r2, r10	$\emptyset$

bb	R	used by inst
2	(r5,1)	(r5,1) $\rightarrow$ 2, 4