Cours Pipelines

Notre objectif est d'améliorer le CPU, maintenant que nous disposons de caches qui améliorent les accès à la mémoire.

```
poca@localhost:~ (1) $ time rm -rf *.html

real    0m0.005s
user    0m0.001s
sys     0m0.004s
poca@localhost:~ $
```

real : le temps total pour que le job soit exécuté
user : le temps que le CPU exécute les instructions du programme
sys : les appels à l'OS

Clock period

$$\text{CPI} = \text{clock per instruction} = \frac{\text{execution time}}{\text{clock period} \cdot \text{instruction count}}$$

C'est le nombre moyen de cycles par instruction. On définit $\frac{1}{\text{CPI}}$ comme le nombre moyen d'instructions par cycle. *(donc il dépend aussi du cache, si on a un mauvais cache le CPI augmente)*.

$$\text{perf} = \frac{f_{\text{clock}}}{\text{instruction count} \cdot \text{CPI}} = \frac{f_{\text{clock}} \cdot \text{IPC}}{\text{instruction count}}$$

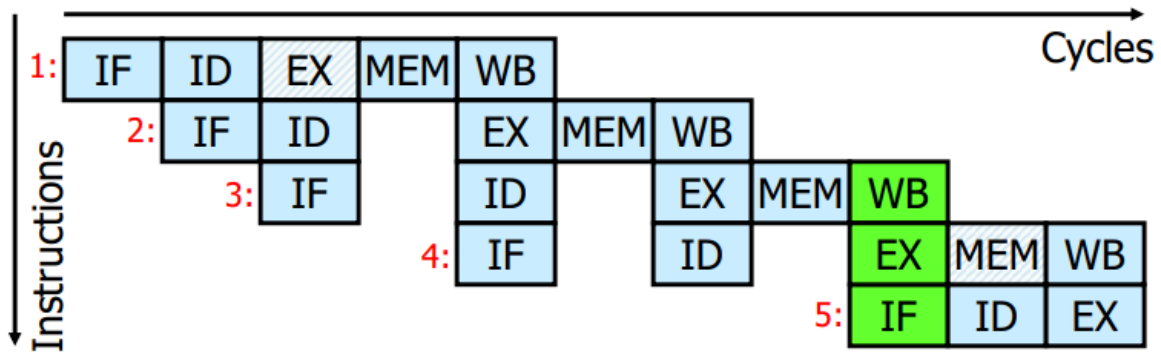
Les moyens d'améliorer la performance :

- réduire le nombre d'instructions ? $\rightarrow$ mais on a besoin d'instructions plus complexes ! et qui ont un CPI plus élevé
- rendre les instructions plus simples (réduire le CPI) $\rightarrow$ mais on a besoin de plus d'instructions pour la même tâche
- augmenter la fréquence de la clock
- exécuter des instructions en parallèle (augmenter le IPC)

Basic pipelining

On commence par ajouter des registers un peu partout dans notre circuit pour réduire le critical path delay. Ça ne semble pas forcément mieux (on ajoute N registers dans notre critical path, on peut diviser notre clock cycle par N, mais on a besoin de N clock cycles pour obtenir le résultat !).

Cependant on peut maintenant utiliser les N-1 combinational paths pendant que notre Nième path est en train d'être calculé ! On a une pipeline → la **latency** (le temps pour sortir le premier résultat reste **N** cycles, mais le **throughput** a changé et est maintenant un nouveau résultat à chaque cycle !)



Simple Multi-Cycle Processor

`FETCH` → `DECODE` → `EXECUTE`

mais maintenant on doit répliquer le hardware! si `FETCH` a besoin d'un adder et que `EXECUTE` a besoin d'un ALU, nous devons répliquer les deux (parce qu'ils sont utilisés en même temps)

CISC VS RISC

(où comment séparer nos longues instructions en plein de petites instructions ?)

si on a un `addi` , on doit passer par `FETCH` → `DECODE` → `EXECUTE` (FSM)
mais si on a un `sw`, on doit passer par `FETCH` → `DECODE` → `LOAD1` → `LOAD2` (FSM)

comment gérer ça dans une pipeline ? un moyen simple de supporter les deux instructions, c'est de passer toujours par tous les states!

`FETCH` → `DECODE` → `EXECUTE` → `LOAD1` → `LOAD2`

but... it increases latency. and throughput.

Dependencies

```
(A) addi r0, r0, 1
(B) sub r2, r0, r1
```

`sub` is getting the wrong value from `r0`!

Types de dépendances :

- `RAW` (B veut lire le registre dans lequel A écrit)
- `WAR` (B veut écrire dans le registre que A lit)
- `WAW` (B veut écrire dans le registre que A écrit)

Stalling

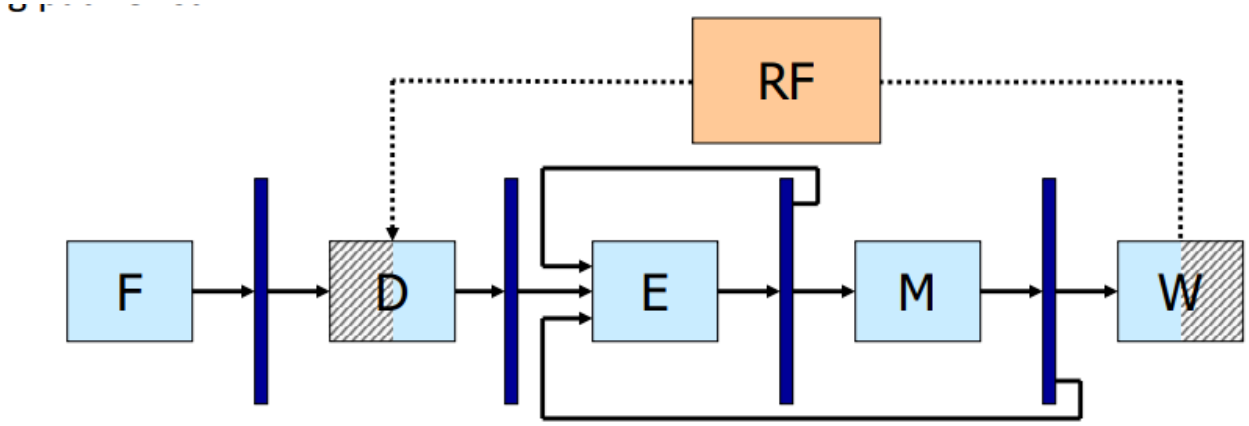
Une solution, le stalling! (on attend que A ait fini ce qu'il doit faire avant de lancer B)
On ajoute du hardware dans notre pipeline.

Une autre solution, c'est que le compilateur ajoute des `nop`s de lui-même dans le code (plus besoin de hardware!) mais cela n'est pas vraiment possible parce qu'on exposerait des détails de **microarchitecture** (multi-cycle/pipelined, FSM ou pipeline structure, ...) dans l'**architecture** (le contrat ISA c-à-d instructions, registers, ...).

Data Hazards (partially solved using forwarding paths)

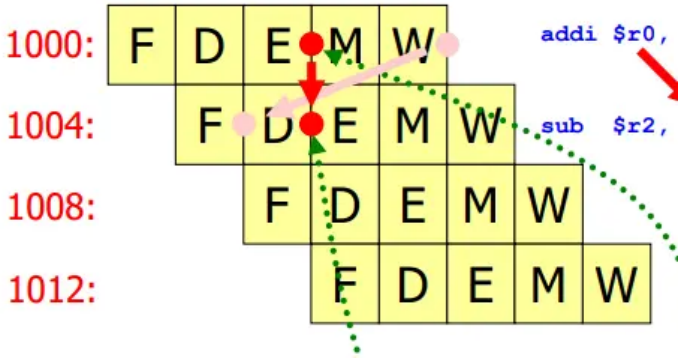
On peut parfois éviter le stalling en transférant des valeurs (par exemple une fois que le `add` a été fait, on n'a pas besoin d'attendre le stockage dans la mémoire pour utiliser le résultat).

Cette solution est assez simple en mise en place, on ajoute juste des forwarding paths.



Entre `W` et `D` c'est assez spécial, le write se fait tôt dans le cycle en cours, tandis que le read (`D`) n'est pris en compte qu'à la fin du cycle (`D` to `E`), donc le forwarding de `W` à `D` se fait en un seul cycle sans qu'on ait besoin de rien de +.

comme montré sur le schéma au dessus, les valeurs doivent arriver au stage juste avant le execute :



Structural hazards

Se produit quand deux instructions veulent utiliser le même stage de la pipeline en même temps. Ça n'arrive pas si on stall tout quand une instruction attend des données, mais si on ne stall **que** cette instruction (et pas les suivantes, qui sont peut-être parfaitement indépendantes), ça peut arriver !

Control hazards

Se produit quand une instruction branche (ou non) sur une autre partie du code, et qu'on ne sait donc pas quelle sera la prochaine instruction.

Solution 1 $\rightarrow$ stall la pipeline.

Solution 2 $\rightarrow$ définir des delay slots (quand on écrit le code en assembleur on doit penser au fait que les deux instructions après un `branch` sont **toujours** exécutées... donc soit on met des `nop` soit on met du code qui était avant -- le code devient très contre-intuitif!) et surtout on expose des détails de microarchitecture donc c'est vraiment pas fou en termes de compatibilité des fichiers binaires.

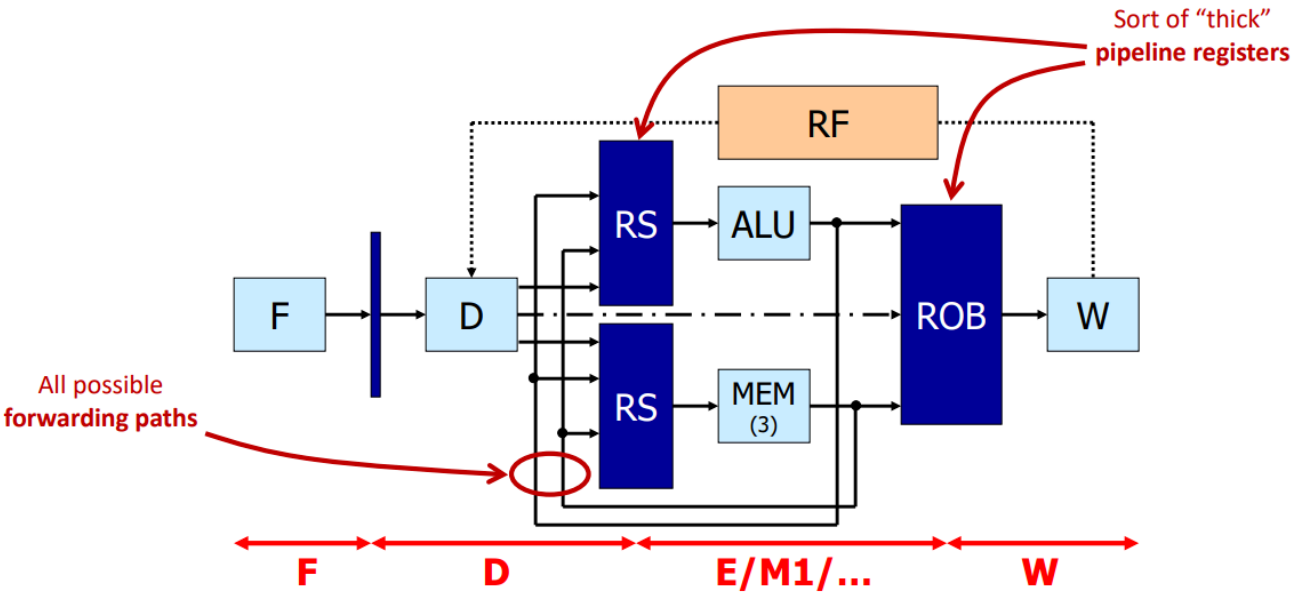
Solution 3 (meilleure) $\rightarrow$ en fait récupérer et décoder une mauvaise instruction, ça ne pose pas de problème ! (on peut kill tant que l'instruction arrive au `EXECUTE`). Donc on peut toujours faire un choix (pris/pas pris) et au pire on est au même niveau que le stall. Certains algorithmes de prédiction parviennent à prédire jusqu'à 95-99% des résultats de branche.

Dynamic pipelining

Jusqu'ici les instructions s'exécutent en parallèle, mais toujours dans l'ordre (c'est-à-dire que pour que l'instruction 3 arrive au stage N de la pipeline, il faut que l'instruction 2 soit au minimum au stage N+1 -- on fait ça en faisant en sorte que chaque instruction passe par **tous** les stages même si elle n'en a pas besoin).

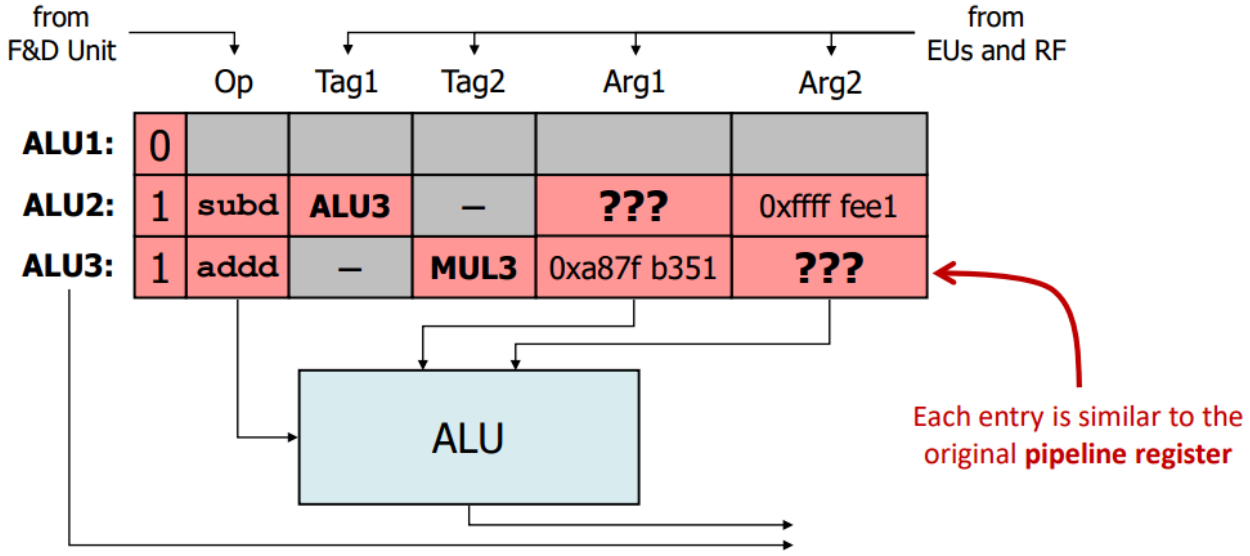
C'est dommage, parce que parfois deux instructions sont dépendantes mais une troisième après pourrait avancer plus vite (voire finir avant la précédente !). Pour ça, ce serait bien qu'on puisse continuer à récupérer et décoder même si une instruction se bloque en attendant un résultat (pour savoir si elle est indépendante !).

On commence par ajouter des **Reservation Stations** et un **Reorder Buffer**.



Les **Reservation Stations** se trouvent juste avant l'exécution. Elles vérifient que les opérandes sont bien déjà dispos (elles les reçoivent depuis les forwarding paths), puis lancent l'exécution quand l'unité demandée est disponible. Elles ont une certaine taille (= la taille de la file d'attente, ici 3 par exemple).

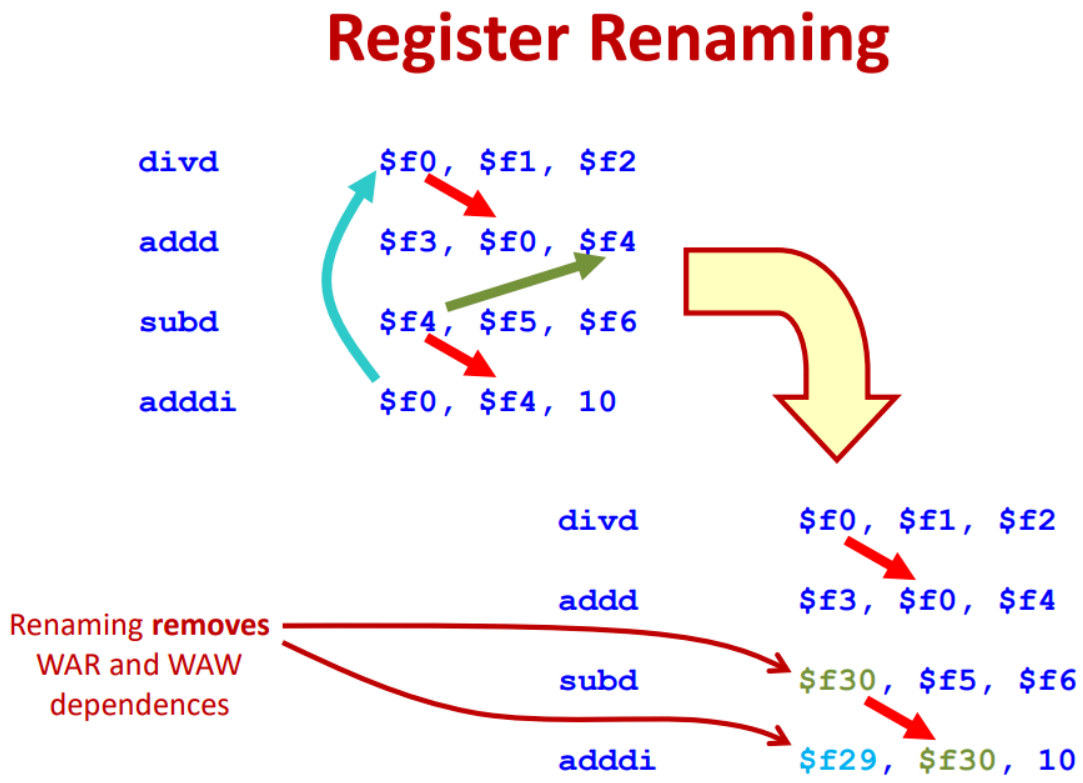
Les tags, ce sont des références vers la ligne de la reservation station de l'instruction qui est capable de donner l'opérande manquante. Pour les trouver, on va voir dans le ROB! (voir plus bas)
Si la valeur est déjà calculée, on peut la lire depuis le ROB, sinon on peut prendre le tag du ROB (le plus récent !).



Par exemple on vient ici que **subd** est dans la reservation station, prête à être lancée mais qu'elle dépend d'un tag qui n'est pas encore disponible. Elle le recevra depuis les autres Execution Unit. Là, on a résolu le problème RAW.

Maintenant qu'est-ce qu'on fait si notre subd peut s'exécuter et qu'il écrit dans le registre `s9` alors qu'après on avait un `li s9, 1` ? (qui, comme étant indépendant, s'est exécuté avant ?). Ce problème ne se posait pas en simple pipelining.

En fait, pour les dépendances de type WAW ou WAR, c'est une "fausse dépendance" (pour reprendre l'exemple d'avant, il n'y a pas d'intérêt à écrire dans `s9` puis écraser par un zéro, ce qui devait se passer c'est qu'entre le subd et le `li`, il y avait une instruction qui lisait le résultat du subd, donc il "suffit" de renommer le registre `s9` en `s10` dans les deux lignes et c'est résolu !)
C'est ce qui est montré dans l'exemple dessous.

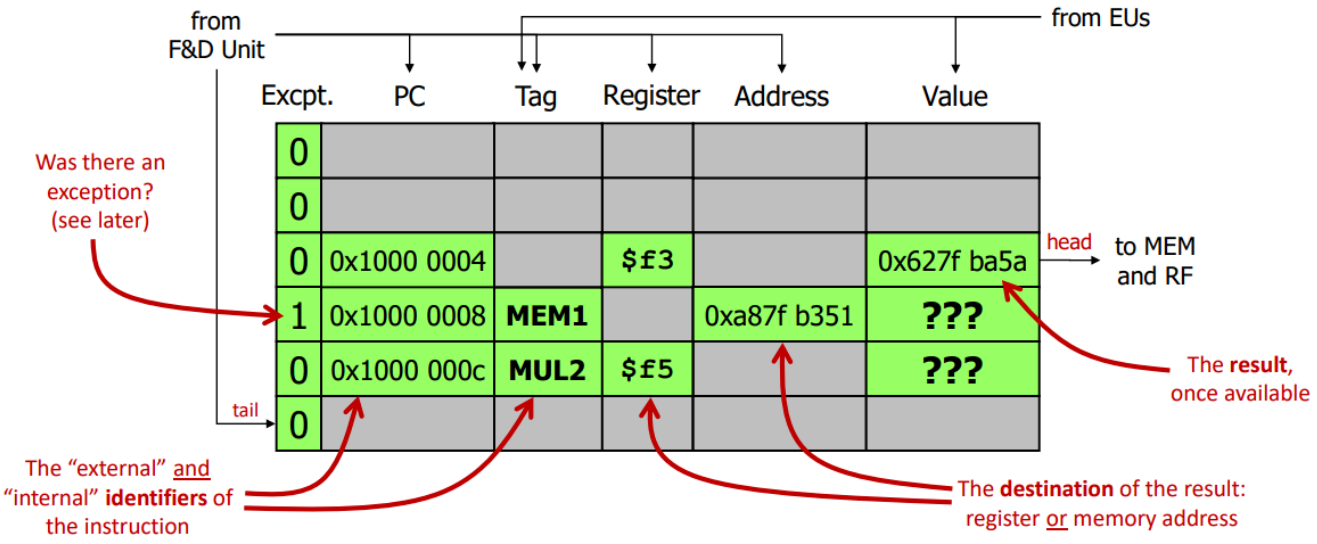


Et en fait le problème disparaît tout seul avec les reservation station ! En effet, on se fiche d'écrire forcément **après** avoir lu puisqu'on ne lit plus directement depuis le register file, mais c'est la reservation station qui détient une référence (le tag, unique) vers ce qu'elle veut lire !

Et les exceptions ? Et la mémoire ?

Maintenant que tout est correct, rapide, mais dans le désordre, comment gère-t-on les exceptions ? et le fait qu'on écrive dans le bon ordre dans la mémoire ?

C'est là que le **Reorder Buffer** (ROB) est utile !



Le ROB sait :

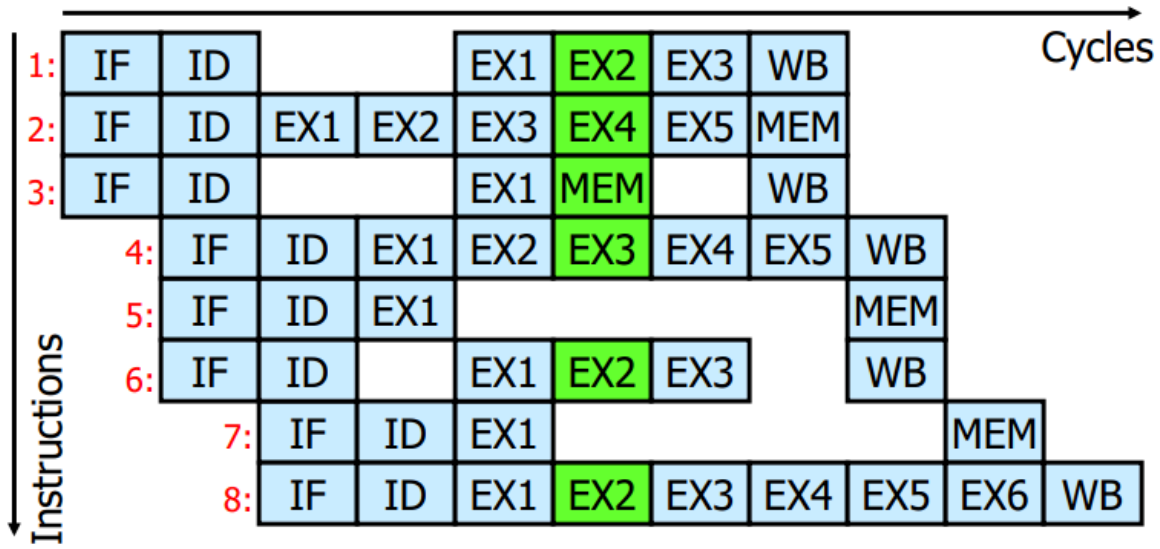
- où écrire le résultat (dans un registre ou à une adresse dans la mémoire)
- quelle est la ligne de l'instruction (PC)
- quelle est la ligne de la réservation station de l'instruction
- quelle est la valeur à écrire
- s'il y a une exception en attente pour cette instruction

Et dans l'ordre original, il va écrire dans le registre ou dans la mémoire la valeur quand elle est disponible. On appelle ça commit l'instruction.

Si une instruction génère une exception, on attend que toutes les instructions précédentes soient commits puis on lève l'exception. On vide ensuite le ROB. et on reprend.

Superscalar processor

Ces processeurs fetch (et commit) plusieurs instructions à la fois !



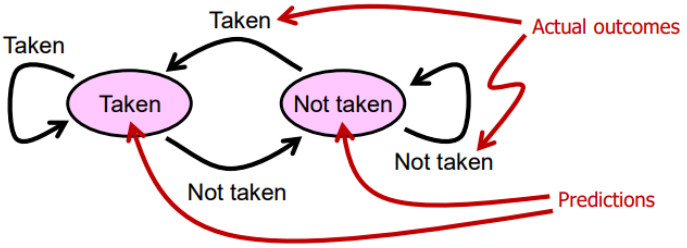
Les dernières finitions...

On a encore deux problèmes :

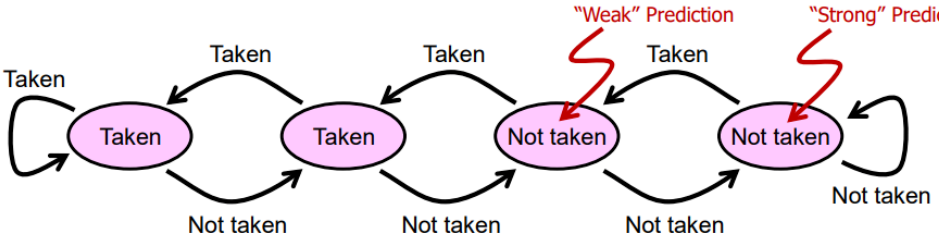
- data hazard: il y a toujours des dépendances (mais on ne peut pas y faire grand choses)
- control hazard: on peut améliorer la prise de décision !

Dynamic branch prediction

- Simplest one-bit predictor: “do the same as last time”



- Two-bit predictor (saturating counter): adding some “inertia” or “take some time to change your mind”



on retient les résultats passés pour prédire les suivants (ajouter de l'inertie)

On peut encore faire mieux ! On peut aller plus loin que simplement fetch et decode les instructions dont on est pas sûr, on peut aussi les exécuter ! on les met ensuite dans le

ROB. Et le ROB n'a le droit de commit les instructions après une branche qu'une fois le résultat de la branche connu.

Simultaneous Multithreading (SMT)

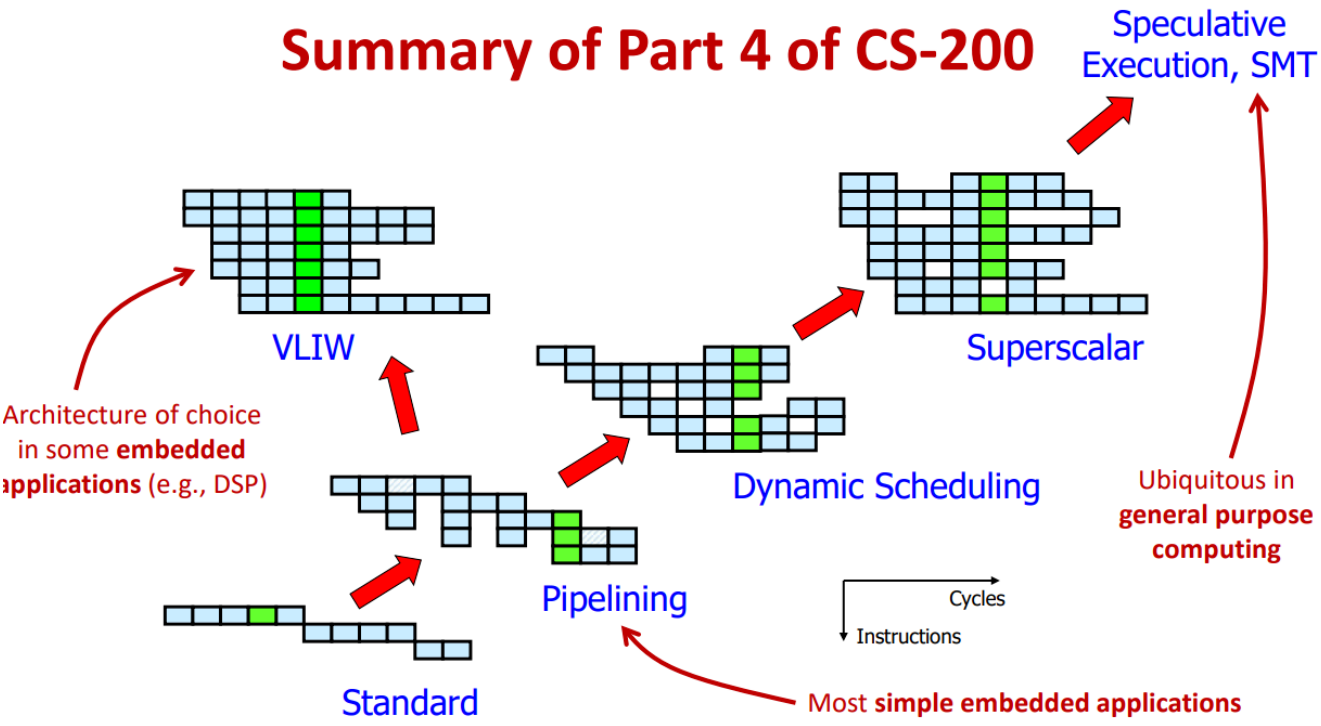
En fait ça ne coûte rien d'exécuter plein de programmes en même temps! (pour occuper nos execution unit). pour ça, on ajoute plusieurs program counters, plusieurs robs, et on garde les même execution units et les mêmes reservation stations (elles n'ont pas besoin de savoir quel thread a fait la demande ! de toute façon le tag est unique pour tout le monde).

Non-blocking cache

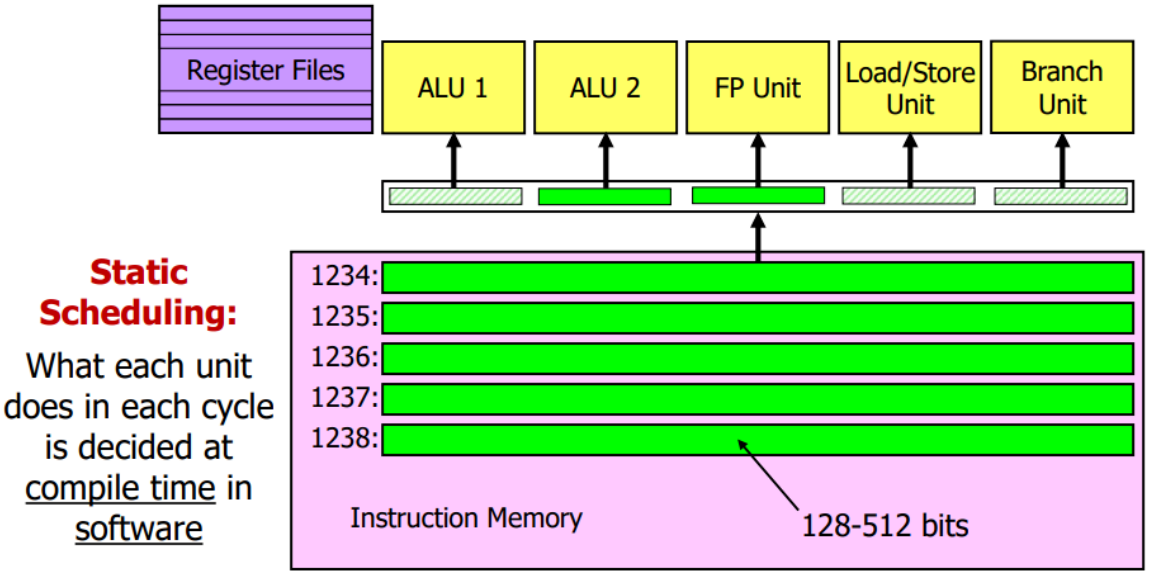
Pour que la pipeline fonctionne bien il nous faut un cache capable de :

- donner une valeur de son stockage tout en attendant une autre valeur de la part de la mémoire (on veut donc un cache **hit under** une opération de cache **miss**)
- faire une nouvelle requête à la mémoire tout en attendant une autre valeur de la part de la mémoire (**miss under miss**)

Very-Long Instruction Word



C'est une tentative pour décider de tout ce qui est fait en parallèle en software (rien n'est géré dynamiquement par le hardware -- on appelle ça du **static scheduling**). Donc c'est cool, car moins de hardware. Mais c'est peu compatible.



Donc pas de ROB, etc. tout est statique.

Problèmes :

- on a pas toutes les informations au compile time (comment détecter les dépendances ? quand on ne connait pas forcément les adresses contenues dans les registres)
- on ne peut plus faire de register renaming dynamique ! ça pose notamment problème dans les boucles, on ne peut plus faire les prédictions qu'on faisait avant (occuper nos exec units avec des calculs qui ne servaient peut-être pas). Ce qu'on peut faire, en supposant p. ex que l'on sache que notre nombre d'itérations sera un multiple de 5, c'est du loop unrolling :