Lezione 41 pipeline 2

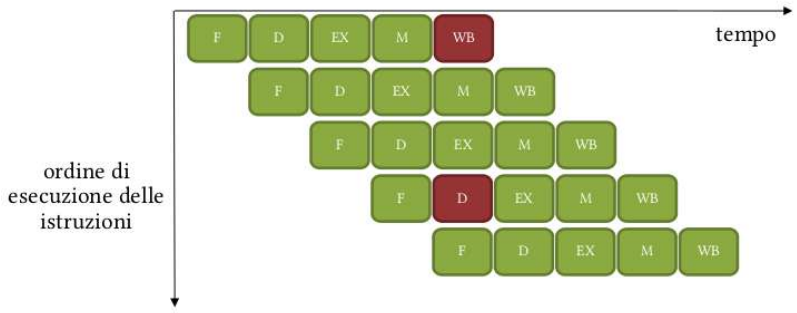
lunedì 11 marzo 2024 15:10

Come già detto , la propagazione delle istruzioni con i relativi segnali nei registri tamponi , potrebbe portare a delle problematiche , come per esempio il risultato di un'operazione potrebbe influenzare le operazioni successive, oppure lo stesso risultato potrebbe non essere disponibile quando effettivamente serve. Vediamo un esempio :

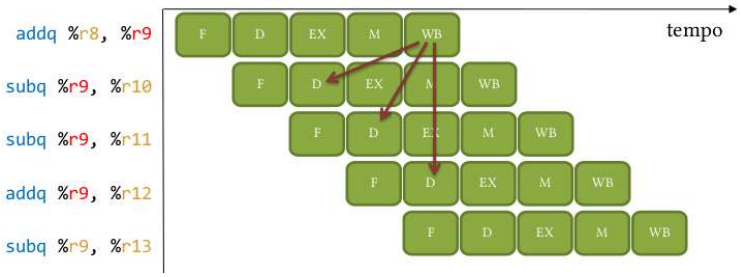
```
addq %rax, %rbx
movq %rbx, 0xabcd(%rdx)
```

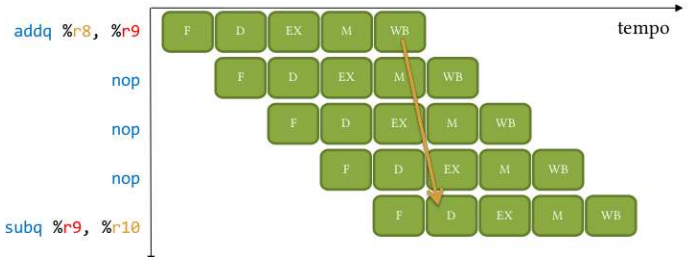
In generale questi conflitti (alee) sorgono nei processori pipeline , quando non è possibile eseguire un'istruzione nel ciclo immediatamente successivo senza dovere cambiare la semantica del programma eseguita in una struttura /processore multi ciclo.

Principalmente le criticità vengono suddivise in 3 categorie :

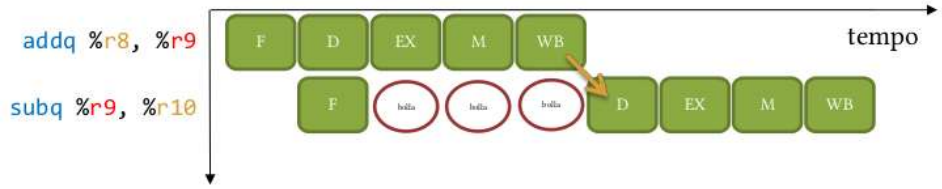
1. Strutturale
   a. Diverse istruzioni cercano di usare la stessa risorsa hardware , nello stesso ciclo di clock
   b. Mediato dalla struttura Harvard , la quale , usando la cache di primo livello (vicina al processore), elimina automaticamente la criticità strutturale per accesso a memoria dati e memoria istruzioni.
   c. Un componente che ne soffre è il banco dei registri , in quanto potrebbe capitare che differenti istruzioni lo usano , ma semantica diversa
   d. 

   e. Quindi una possibile soluzione a criticità di questo livello è fare scrittura dei dati sul fronte di salita del clock, mentre la lettura del dato viene effettuata sul fronte di discesa .
2. Su dati
   a. Criticità nella quale si cerca di usare il dato prima che sia disponibile (magari in seguito al verificarsi o meno di una condizione)
   b. In dettagli si ha quando un'istruzione nella pipe dipende da una precedente che ancora non ha finito
   c. Si dividono in due sotto categorie : **define use e load use**
   d. Vediamo la prima :
      i.
```
addq %rax, %rbx
subq %rax, %rbx
```
      ii. Qui si nota che lo stesso operando viene usato da due istruzioni : la seconda lo vuole usare quando ancora la prima istruzione non è stata ritirata dalla pipe
      iii. Andiamo a vedere un esempio pratico:
         1. Possibile coinvolgimento di più istruzioni
```
addq %r8, %r9
subq %r9, %r10
```
         2.
```
subq %r9, %r11
addq %r9, %r12
subq %r9, %r13
```
         3. Che dal punto di vista del processore diventa :

4. 

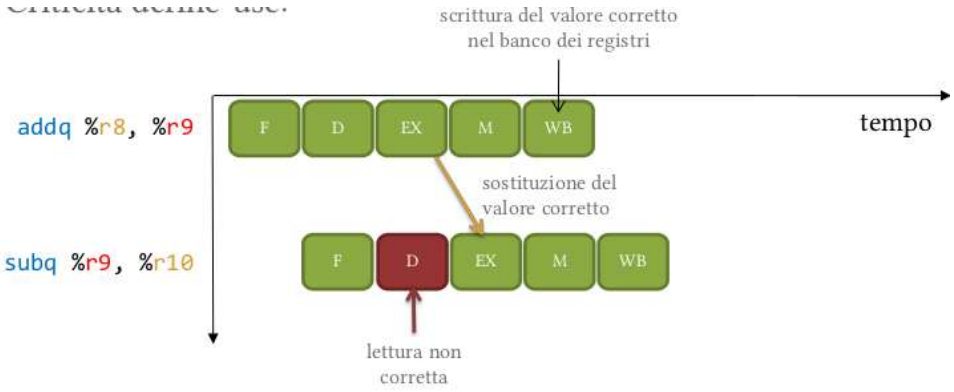
5. Questo conflitto può essere risolto in modo sw oppure hw: nel primo inserisco dei tempi morti (aumentando il throughput), oppure sposto le istruzioni (magari vedendo quelle innocue), mentre nella seconda inserisco delle bolle , propagando alla fase successiva i dati appena disponibili
6. **Vediamo quelle sw:**
   a. Inserimento istruzione nop
   b. 

   c. In generale se criticità strutturale sul banco di registri , bastano 2 nop
   d. Spostamento delle istruzioni innocue : istruzioni correlate troppo vicine
   e. 

7. **Vediamo ora quelle hw:**
   a. Vediamo l'inserimento di bolle : si blocca il flusso nella pipeline , finche' la criticità non viene risolta. Vediamo un esempio :
   b. 

   c. <u>Quindi le bolle bloccano la fase di fetch , facendo propagare(nella fase di decode) il codice operativo della nop invece che quello dell'istruzione</u>
   d. Come si individua un conflitto : grazie ad un circuito combinatorio hazard detenction unit , il quale verifica l'identificativo dei registri di src e dst e verifica che istruzione in uno stato più avanti abbia una fase di write back.
   e. Vediamo ora la propagazione in avanti : il risultato generato dalla alu viene messo a disposizione dell'istruzione che segue , sostituendo l'operatore letto dal banco durante il decode

f.



e. Vediamo la seconda :

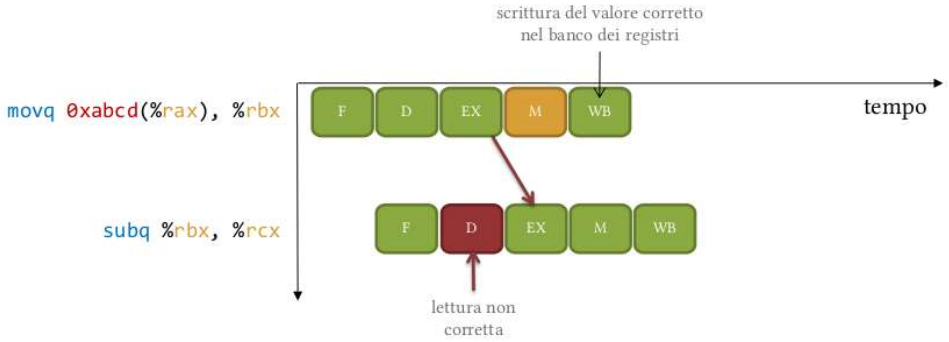
i.
```
movq 0xabcd(%rax), %rbx
subq %rbx, %rcx
```

ii. Qui uno degli operandi della seconda istruzione dipende dal valore letto dalla memoria , la quale ancora non è stata acceduta

iii. Stesse soluzioni del define use

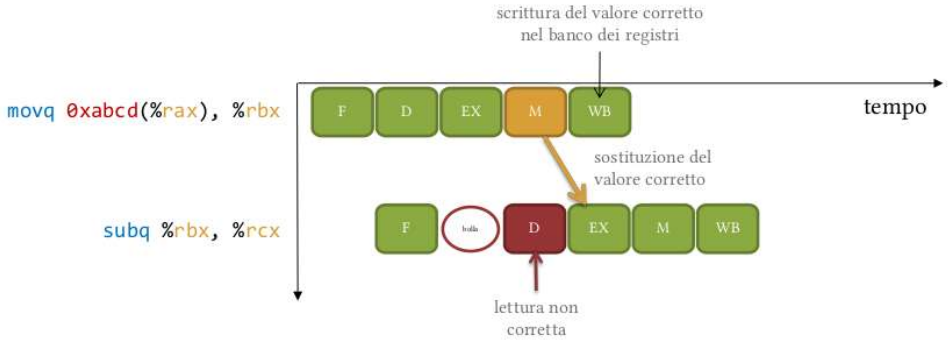
iv. Nel caso di bolle la soluzione è identica , mentre quella per la propagazione in avanti è differente : il dato corretto è disponibile solo al termine della fase di memory:

v.



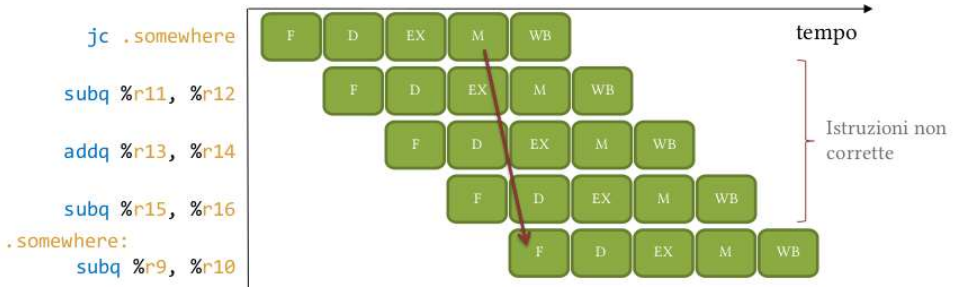
vi. Da notare che qui il risultato non è corretto : il dato viaggia tra entrambe le fasi di execute : quindi devo inserire una bolla :

vii.



3. Su controllo
   a. Si cerca di prevedere quale sia la prossima istruzione da eseguire, senza avere valutato effettivamente la condizione.
   b. In questo tipo di criticità , la pipeline viene alimentata effettuando il fetch della prossima istruzione (RIP->RIP+8), e se vi è presente un salto condizionato RIP viene aggiornato durante la fase di memory :

c.



d. Per risolvere questa criticità si usano 3 tipi di approcci :
   i. Pessimistico:
      1. Non si eseguono istruzioni significative fino al completamento della scelta , quindi si ha un'implementazione più semplice , ma vista l'attesa , si ha un aumento del numero di cicli di clock (CPI aumenta)
   ii. Ottimistico :
      1. Si prospetta che il salto non venga eseguito : quindi se il salto non verrà eseguito aumentano le prestazioni , altrimenti si ha la condizione del pessimistico. Invece come contro si deve annullare (per forza) gli effetti delle istruzioni eseguite dopo il salto : aumenta quindi la complessità dell'unità di calcolo . Vediamo un esempio : il salto viene effettuato: quindi si deve annullare (nella pipeline) l'esecuzione delle istruzioni entrate nella pipe : flash della pipe, quindi rispristino dei registri , simulando cosi' una nop , **ed in particolare si deve usare altro registro tampone per salvare la copia del registro flags.**

e. Quindi in ottica di ottimizzare le prestazioni (sempre caso del salto) : se salto si cerca di prevedere se il salto sarà effettuato o meno : se effettuato si massimizzano le prestazioni , mentre nel secondo si effettua il flush della pipe . In generale due tipi di predizioni sulla condizione :**dinamica e statica.** Vediamoli in dettaglio : il primo si ha quando il valore previsto cambia durante l'esecuzione , ma necessita di implementazione hardware (branch prediction unit) , e questo salto sarà effettuato basandosi sullo storico ; mentre per la seconda il salto verrà effettuato verso e solo parti precedenti del codice , e questo tipo predizione viene guidata dal compilatore e dal programmatore. Quindi riassumendo: per la predizione del salto , in generale si usa la cosiddetta **tabella di predizione del salto** : ovvero piccola memoria indicizzata dal bit meno significativo dell'indirizzo dell'istruzione di salto condizionato , nella quale ad ogni istruzione contenuta nella memoria è associata una previsione: effettuo il salto oppure no?? : nel primo caso il salto si effettua e si paga un solo ciclo di clock in più , mentre se nella seconda non si fa , quindi non si sprecano cicli , mentre nel caso peggiore (previsione errata), la si deve modificare , si deve effettuare il flush della pipe e per quanto riguarda la penalità è la stessa se non vi fosse predizione. In generale per prevedere se il salto si farà , si potrebbe usare una macchina a stati finiti : la quale effettua il cambio di previsione dopo due previsioni errate :

i.