Migliorare le prestazioni mediante l'uso di pipeline

Paolo Lollini

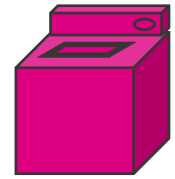
Argomenti

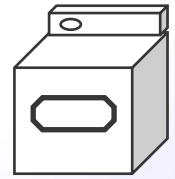
- Panoramica
- Cammino dei dati (semplificato)
- Controllo (semplificato)
- Criticità sui dati
- Criticità sul controllo: salti
- Miglioramenti
- Conclusioni

Panoramica

Introduzione alla pipeline

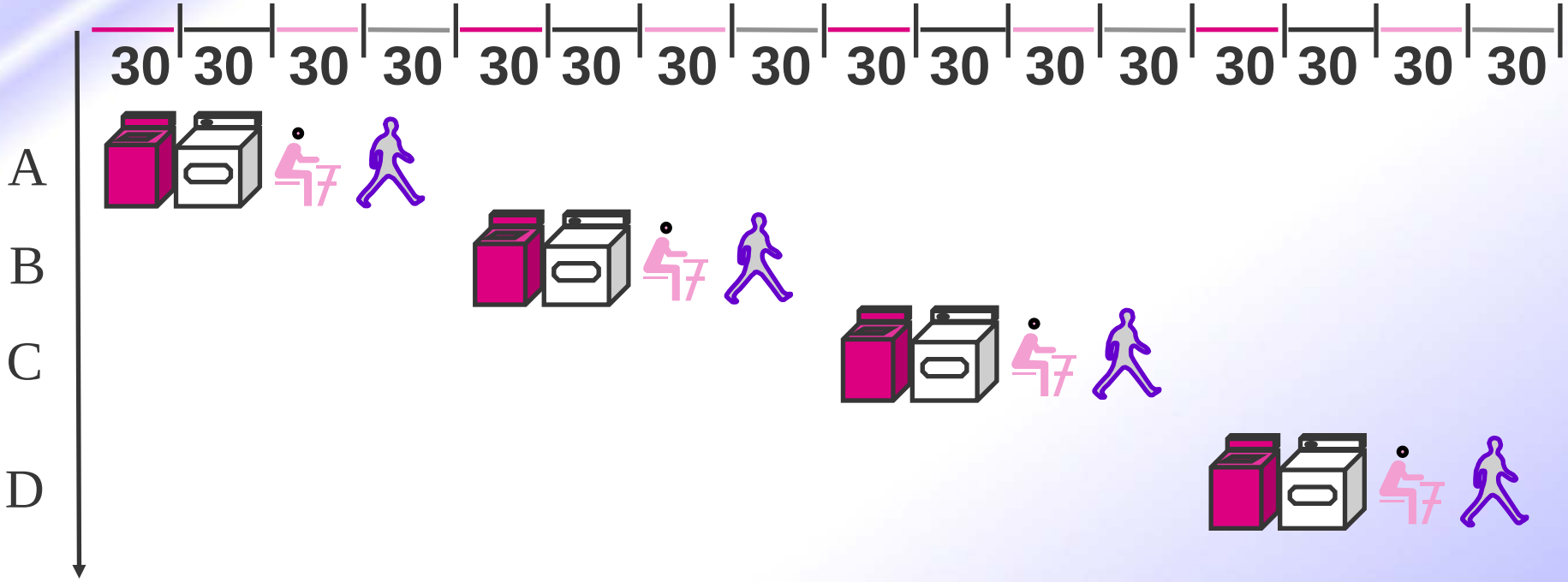
- Anna, Bruno, Carla e Dario devono lavare, asciugare, stirare e riporre un carico di vestiti ciascuno





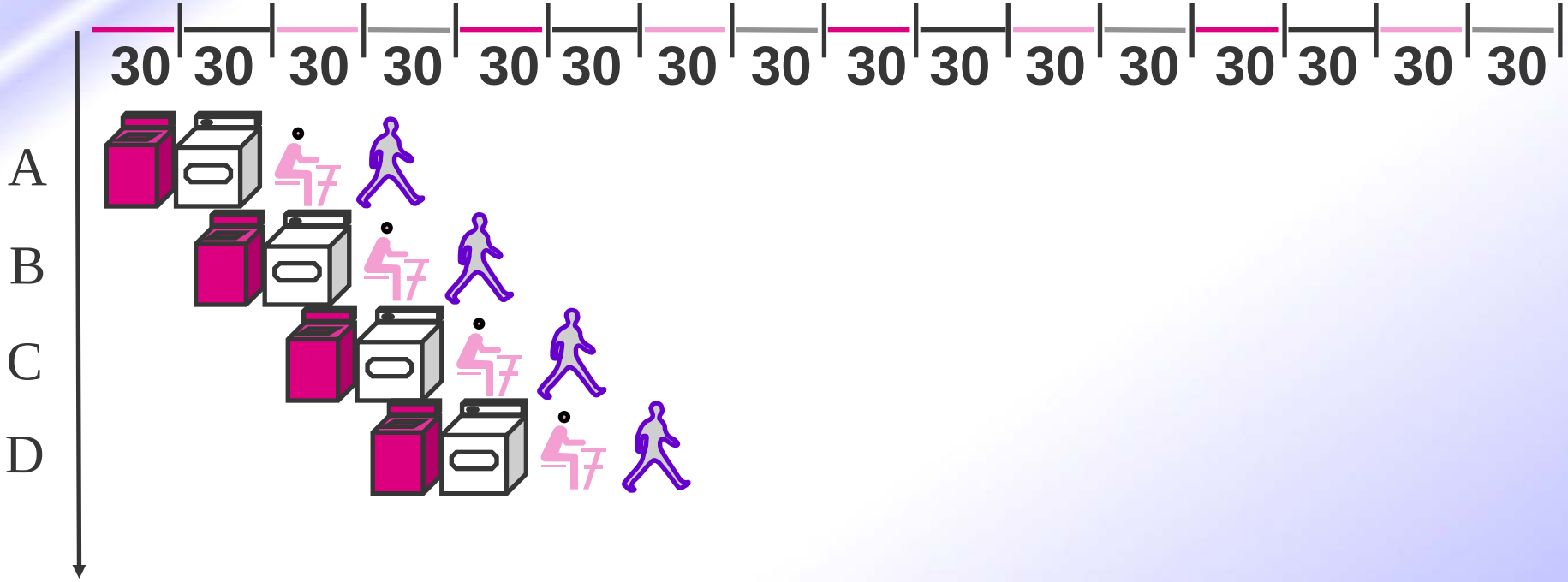
- La lavatrice richiede 30 minuti
- L'asciugatrice richiede 30 minuti
- Stirare richiede 30 minuti
- Riporre i panni richiede 30 minuti

La lavanderia sequenziale



- Richiede 8 ore per 4 carichi

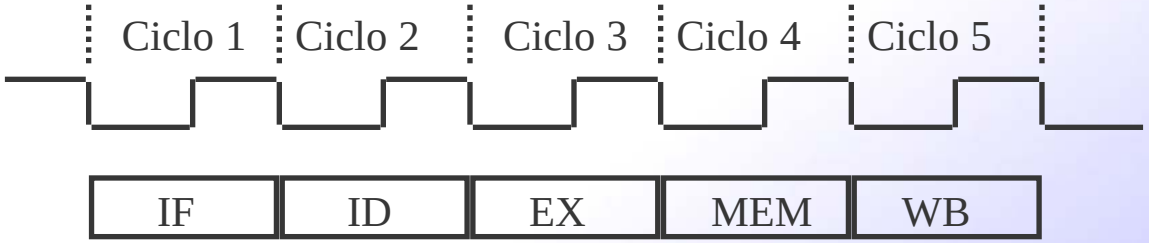
La lavanderia con pipeline



- Richiede 3,5 ore per 4 carichi!

I cinque passi di un'istruzione

- IF (Instruction Fetch): prelievo istruzione dalla memoria istruzioni
- ID (Instruction Decode): lettura registri e decodifica istruzione
- EX (EXecution): esecuzione operazione o calcolo indirizzo
- MEM (MEMory access): accesso alla memoria dati
- WB (Write Back): scrittura in un registro



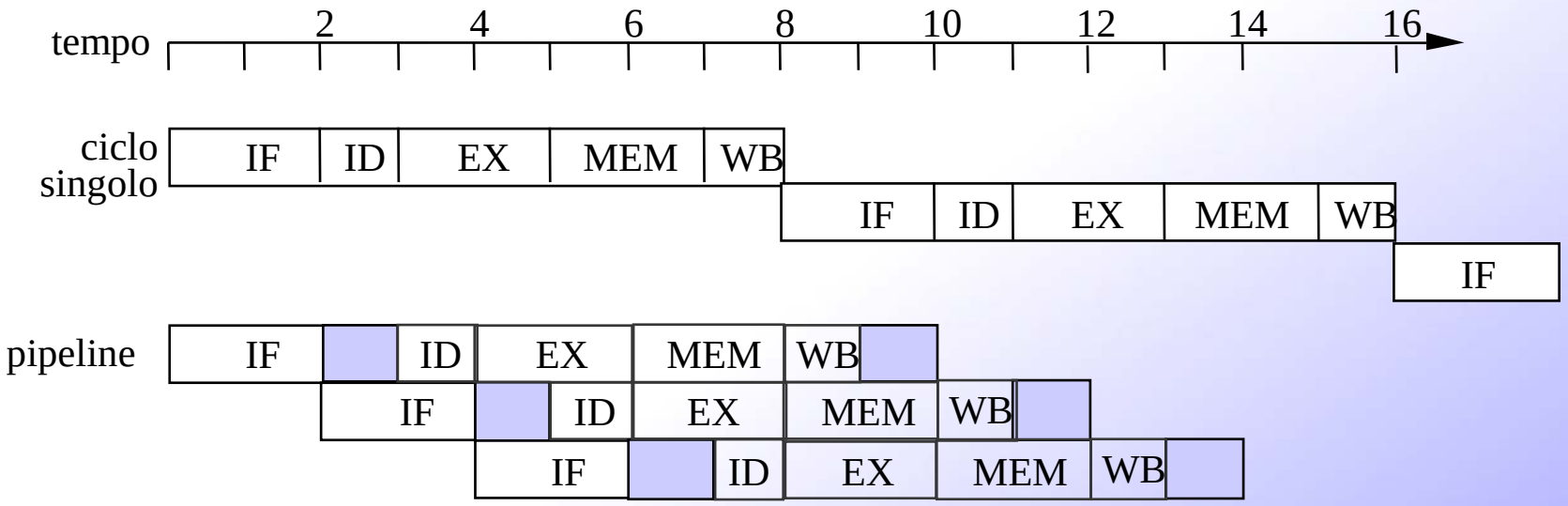
Ciclo singolo vs. Pipeline

- IF: 2ns, ID: 1ns, EX: 2ns, MEM: 2ns, WB: 1ns

Tre lw: lw $s1, 100($s0)

lw $s2, 200($s0)

lw $s3, 300($s0)



Pipeline speedup

- T = tempo tra il completamento di una istruzione e la successiva

$$T_{pipe} = \frac{T_{seq}}{\text{N. stadi}}$$

Il T_{pipe} così calcolato è il minimo teorico in *condizioni ideali*, assumendo cioè il bilanciamento degli stadi, che ci sia abbastanza lavoro da fare e che la pipeline non introduca ritardi aggiuntivi (vedi lucido seguente).

Alcune considerazioni

- La pipeline NON migliora il tempo del singolo compito (tempo di esecuzione della singola istruzione), ma l'esecuzione dell'intero carico lavorativo (**throughput**)
- **Speedup potenziale**: numero di passi della pipeline.

Fattori limitativi:

- Più compiti eseguiti contemporaneamente richiedono l'uso di risorse diverse
- Velocità della pipeline limitata dalla fase di pipeline più lenta (es. 2 ns). Fasi con lunghezze sbilanciate riducono lo speedup
- Tempo per avviare e concludere la pipeline riduce speedup
- Criticità

Criticità *(hazards)*

- Situazioni in cui l'istruzione successiva da eseguire non può essere eseguita nel ciclo di clock immediatamente successivo
- Criticità strutturali:
  - tentativo di utilizzare la stessa risorsa in due modi diversi allo stesso tempo
    - un unica memoria dati/istruzione sarebbe un azzardo strutturale
- Criticità sul controllo:
  - tentativo di prendere una decisione prima che la condizione sia valutata
    - salto condizionato
- Criticità sui dati:
  - tentativo di utilizzare un oggetto prima che sia pronto
    - istruzione che dipende dal risultato di un'istruzione precedente

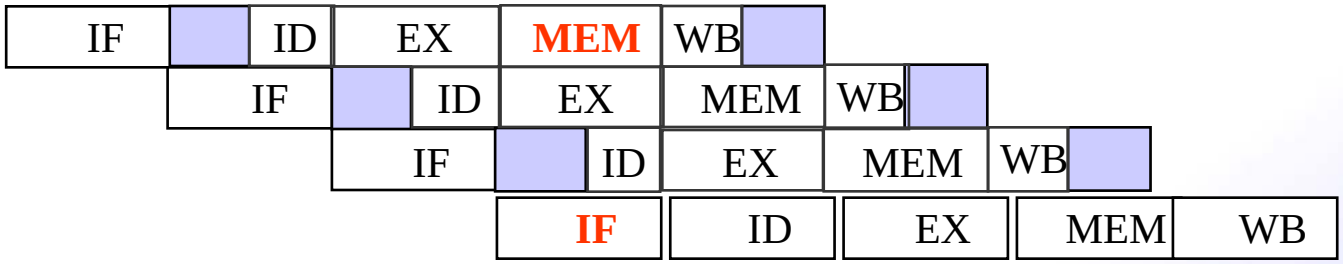
Criticità Strutturali

- Tentativo di utilizzare la stessa risorsa in due modi diversi allo stesso tempo
- Es.

```
lw $s1, 100($s0)
lw $s2, 200($s0)
lw $s3, 300($s0)
add $s4, $s5, $s6
```

Con una memoria dati/istruzioni, la 4^ istruzione sarebbe un problema...

Criticità Strutturale su Memoria



- Soluzioni (hw):
  - riprogettare il set di istruzioni, o
  - hw extra (es. una memoria dati + una memoria istruzioni)

Criticità sul Controllo

- Tentativo di prendere una decisione prima che la condizione sia valutata: beq
- Es:

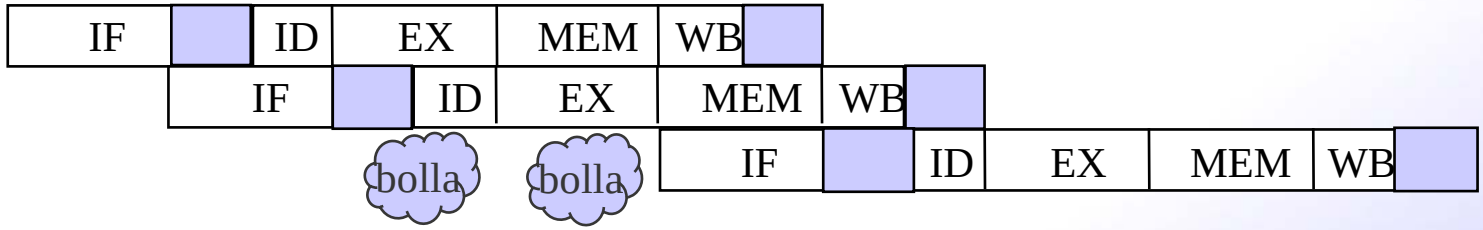
```
beq  $s1, $s2, 40
lw $s3, 300($s0)
```

- **Soluzioni:**
  - Stallo (hw)
  - Predizione (hw)
  - Decisione ritardata (sw)

Stallo *(stall, "bubble")*

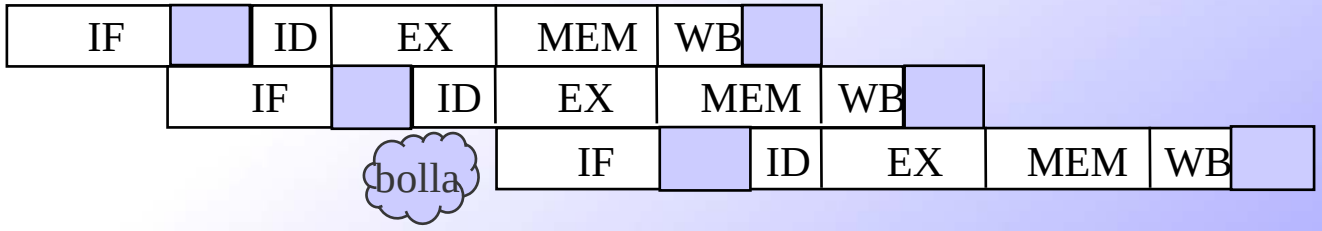
- Operare sequenzialmente fino al termine dell'istruzione che ha generato la criticità

```
add $s4, $s5, $s6
beq $s1, $s2, 40
lw $s3, 300($s0)
```



- Assumiamo ora che il confronto dei registri, il calcolo dell'indirizzo di salto e l'aggiornamento del PC possano essere eseguiti già nella fase ID. E' comunque necessaria una 'bolla' *(bubble):*

```
add $s4, $s5, $s6
beq $s1, $s2, 40
lw $s3, 300($s0)
```



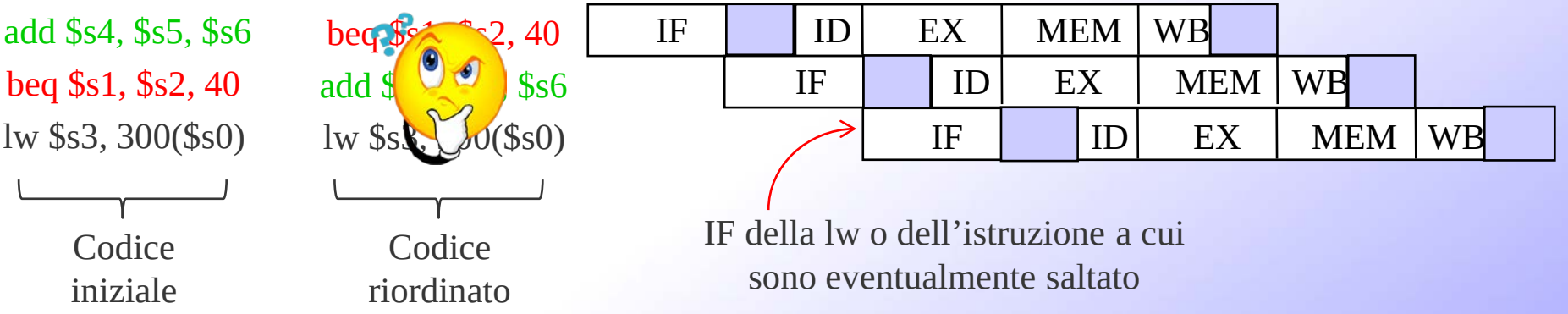
Predizione *(prediction)*

- Soluzione alternativa al problema delle criticità sul controllo
- Predire l'esito del salto. Opzioni:
  - predire sempre che non venga eseguito
  - distinguere per tipi di salto
    - ad esempio, salti all'indietro sempre eseguiti

  - predire dinamicamente in funzione del comportamento dell'istruzione di salto
    - ad esempio, ricordare la storia del salto
- Se la predizione è errata, "ripulire" e far ripartire dall'indirizzo esatto

Decisione ritardata *(delayed branch)*

- Soluzione alternativa al problema delle criticità sul controllo
  - effettivamente utilizzata dall'architettura MIPS
- Eseguire sempre l'istruzione dopo il salto:
  - l'assemblatore riordina il codice, inserendo dopo il salto un'istruzione che *comunque* va eseguita e che è *indipendente* da esso
  - salto eseguito dopo quell'istruzione
    - se non ne ha nessuna, inserisce una NOP (bolla)



Criticità sui Dati

- L'esecuzione di un'istruzione dipende dal risultato di un'istruzione precedente
- Es. criticità su $s0:

```
add $s0, $t0, $t1
add $t2, $s0
```

La 1ª add scrive $s0 so[illegible] 5° stadio;

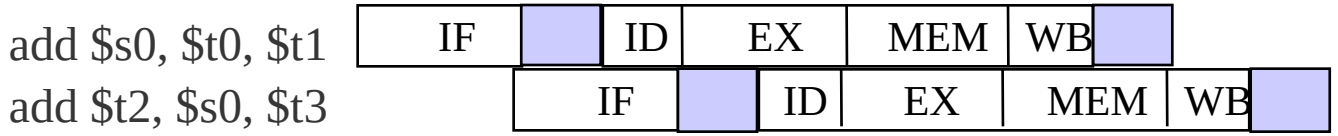
La 2ª add dovrebbe ave[illegible]lore di $s0 all'inizio del suo 2° stadio.

Soluzioni naive:

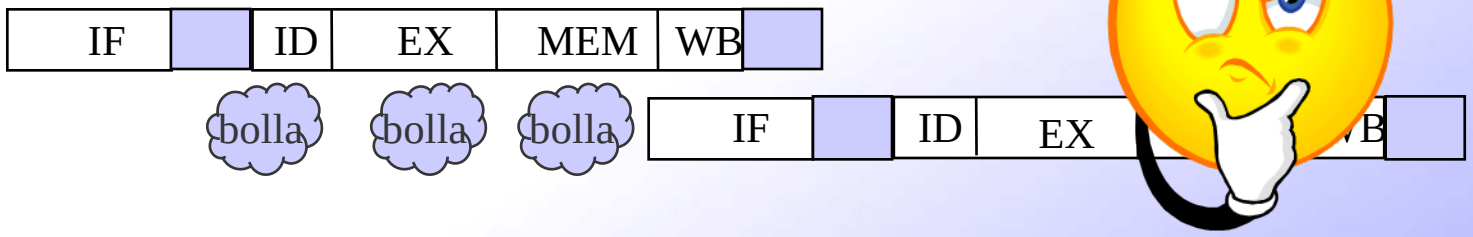
- Stallo: inserzione di bolle prima della 2ª add: troppo costoso!
- Delay (sw): l'assemblatore riordina il codice. In pratica non funziona, perché queste dipendenze capitano troppo spesso.

Soluzione perseguita: propagazione in avanti del risultato (*forwarding*, o *bypassing*)

Criticità sui dati (esempio)

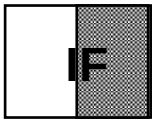


Utilizzando lo stallo, quante bolle dovrei inserire?



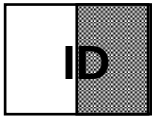
Rappresentazione grafica

Memoria istruzioni:

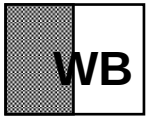


lettura

Banco registri:



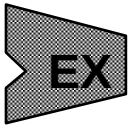
lettura



scrittura

non usato

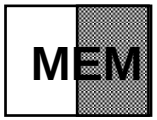
ALU:



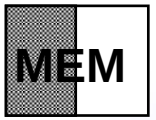
usata

non usata

Memoria dati:



lettura



scrittura

non usata

Esempio: add $s0,$t0,$t1