# Instruction Level Parallelism

**Salvatore Orlando**

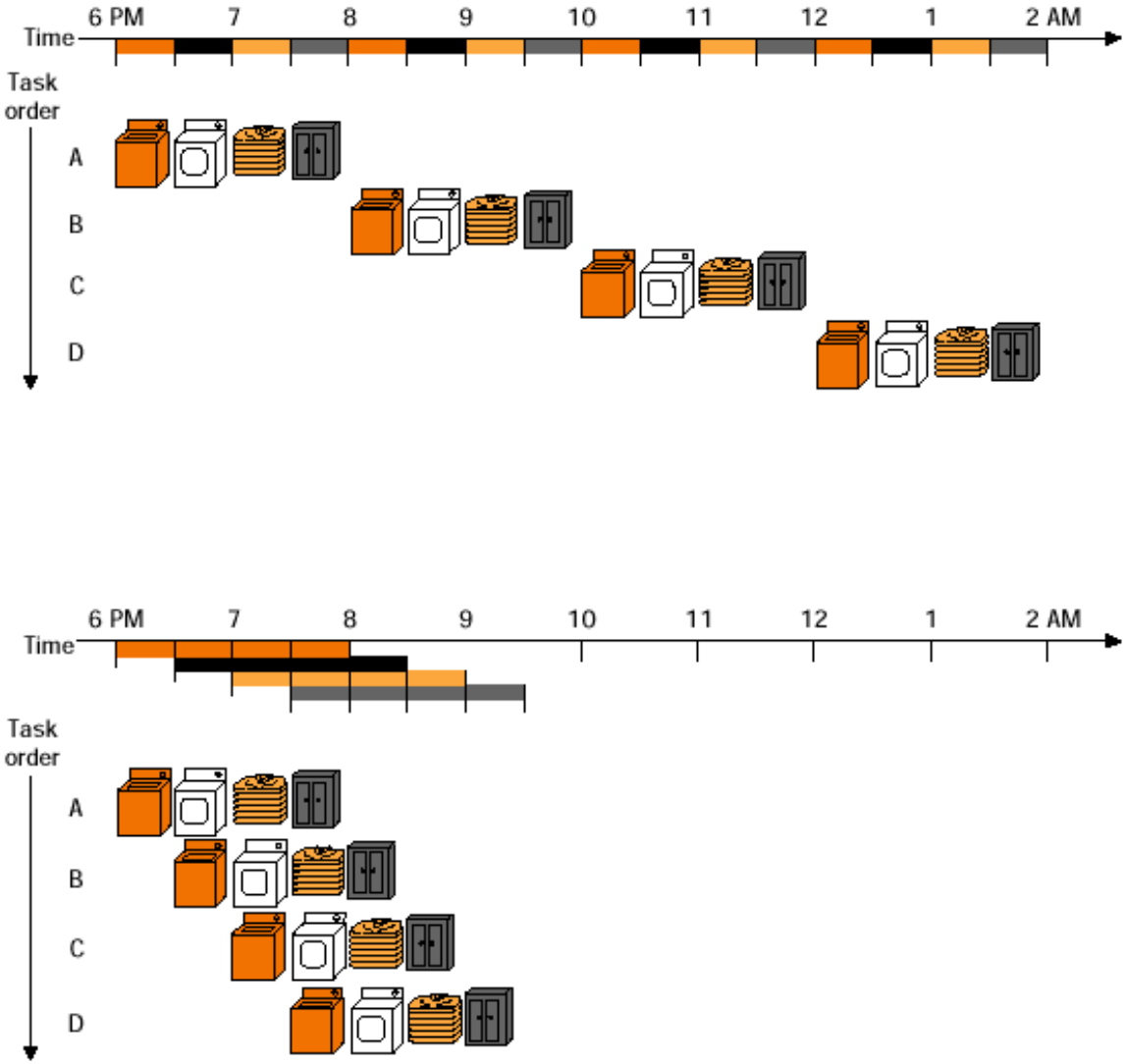
# Organizzazione parallela del processore

- I processori moderni hanno un'organizzazione interna che permette di eseguire più istruzioni in parallelo (ILP)
  - organizzazione pipeline

- Organizzazione pipeline
  - unità funzionali per l'esecuzione di un'istruzione organizzate come una catena di montaggio
  - ogni istruzione, per completare l'esecuzione, deve attraversare la sequenza di stadi della pipeline, dove ogni stadio contiene specifiche unità funzionali

- Grazie al parallelismo
  - abbassiamo il CPI
  - ma aumentiamo il rate di accesso alla memoria (per leggere istruzioni e leggere/scrivere dati) $\Rightarrow$ von Neumann bottleneck

# Pipeline



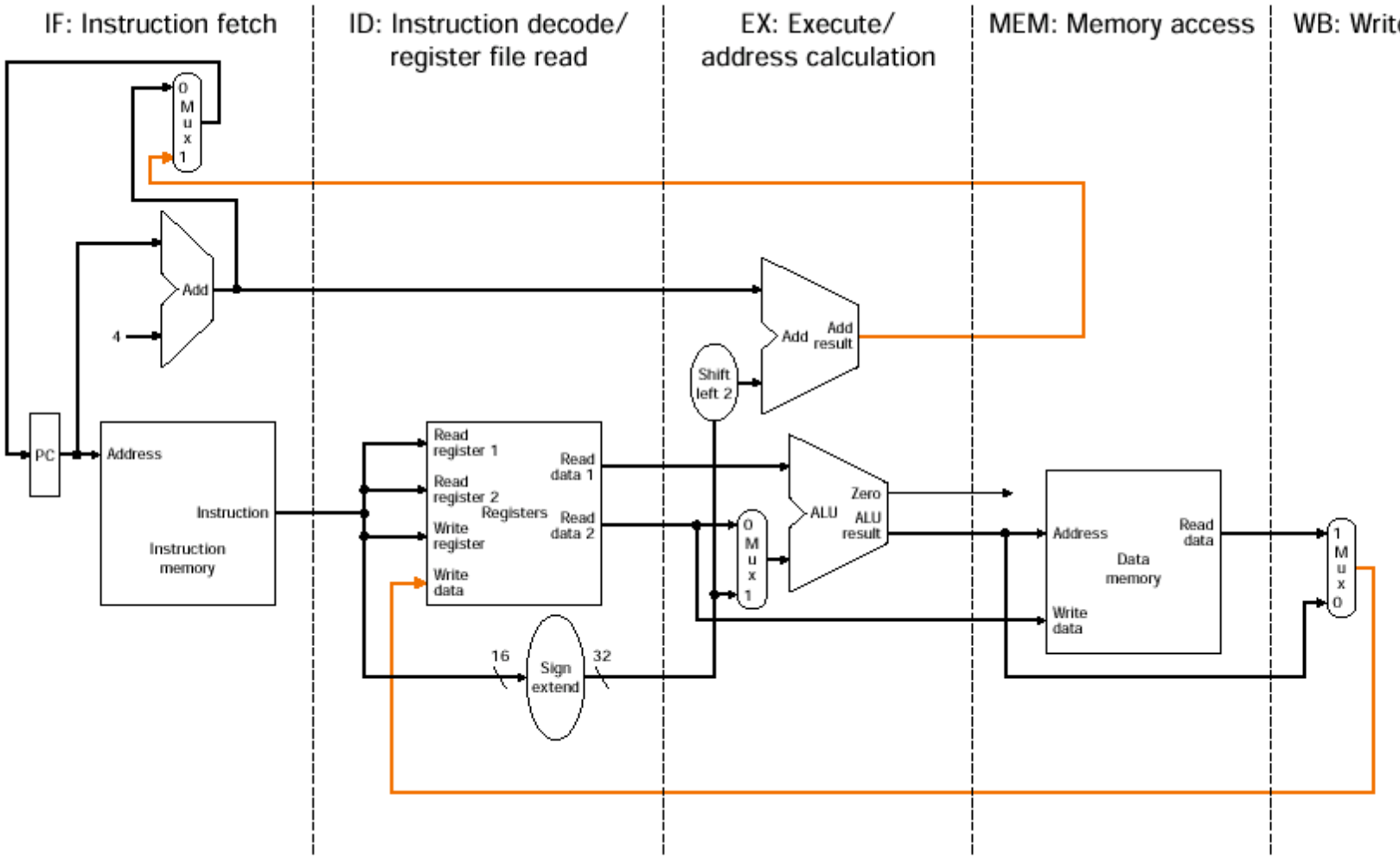

- Le unità funzionali (lavatrice, asciugatrice, stiratrice, armadio) sono usate sequenzialmente per eseguire i vari "job"
  - tra l'esecuzione di due job, ogni unità rimane inattiva per 1,5 ore
- In modalità pipeline, il job viene suddiviso in stadi, in modo da usare le unità funzionali in parallelo
  - unità funzionali usate in parallelo, ma per "eseguire" job diversi
  - nella fase iniziale/finale, non lavorano tutte parallelamente

# Pipeline MIPS

- La semplice pipeline usata per eseguire il set di istruzioni ristretto (lw,sw,add,or,beq,slt) del nostro processore MIPS è composta da 5 stadi

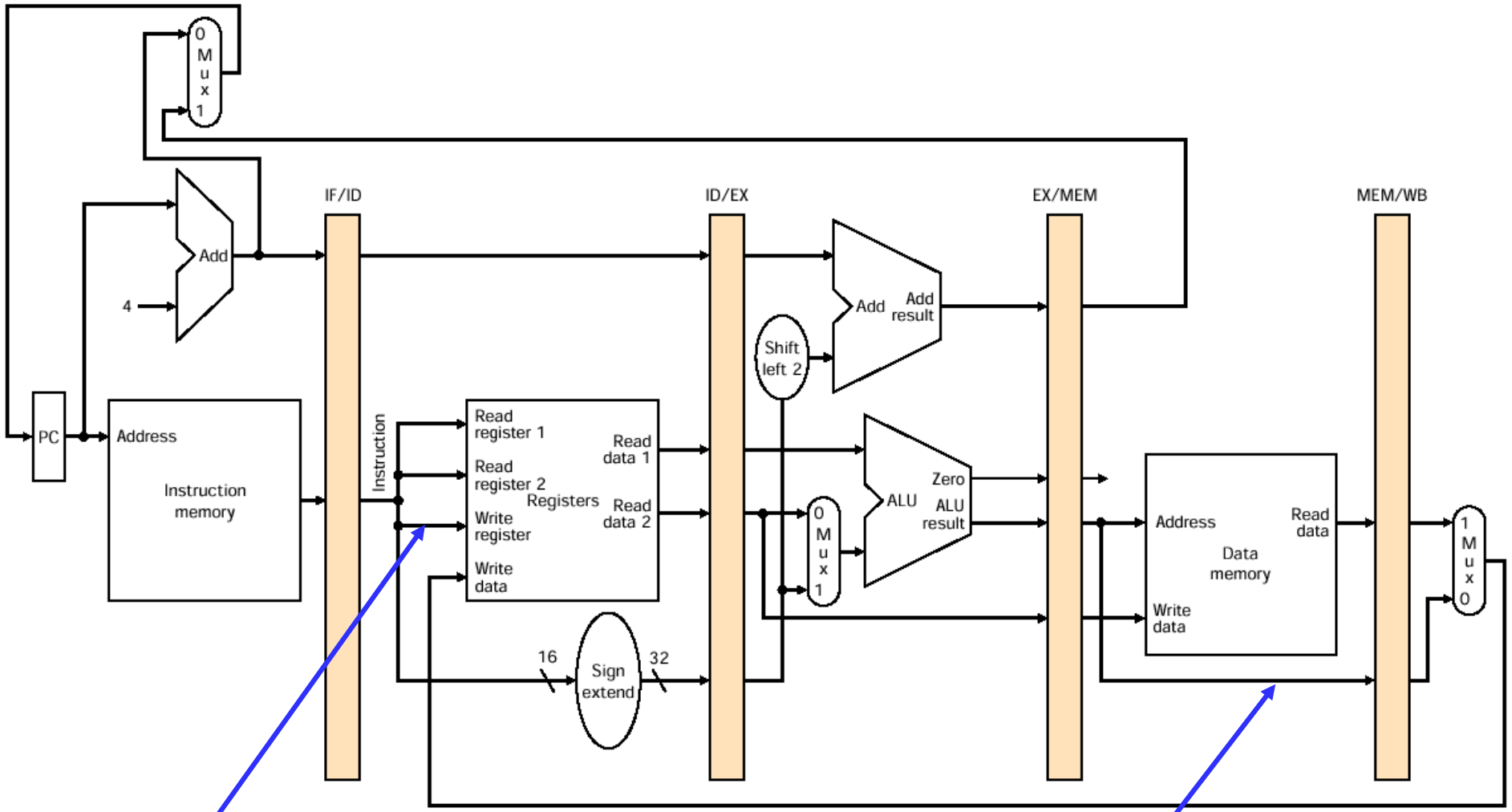
    1. IF : Instruction fetch (memoria istruzioni)
    2. ID : Instruction decode e lettura registri
    3. EXE : Esecuzione istruzioni e calcolo indirizzi
    4. MEM : Accesso alla memoria (memoria dati)
    5. WB : Write back (scrittura del registro risultato, calcolato in EXE o MEM)

# Datapath MIPS (1)




- **Unità funzionali replicate (memoria, addizionatori) nei vari stadi**
- **Ogni stadio completa l'esecuzione in un ciclo di clock (2 ns)**
- **Necessari i registri addizionali, per memorizzare i risultati intermedi degli stadi della pipeline**

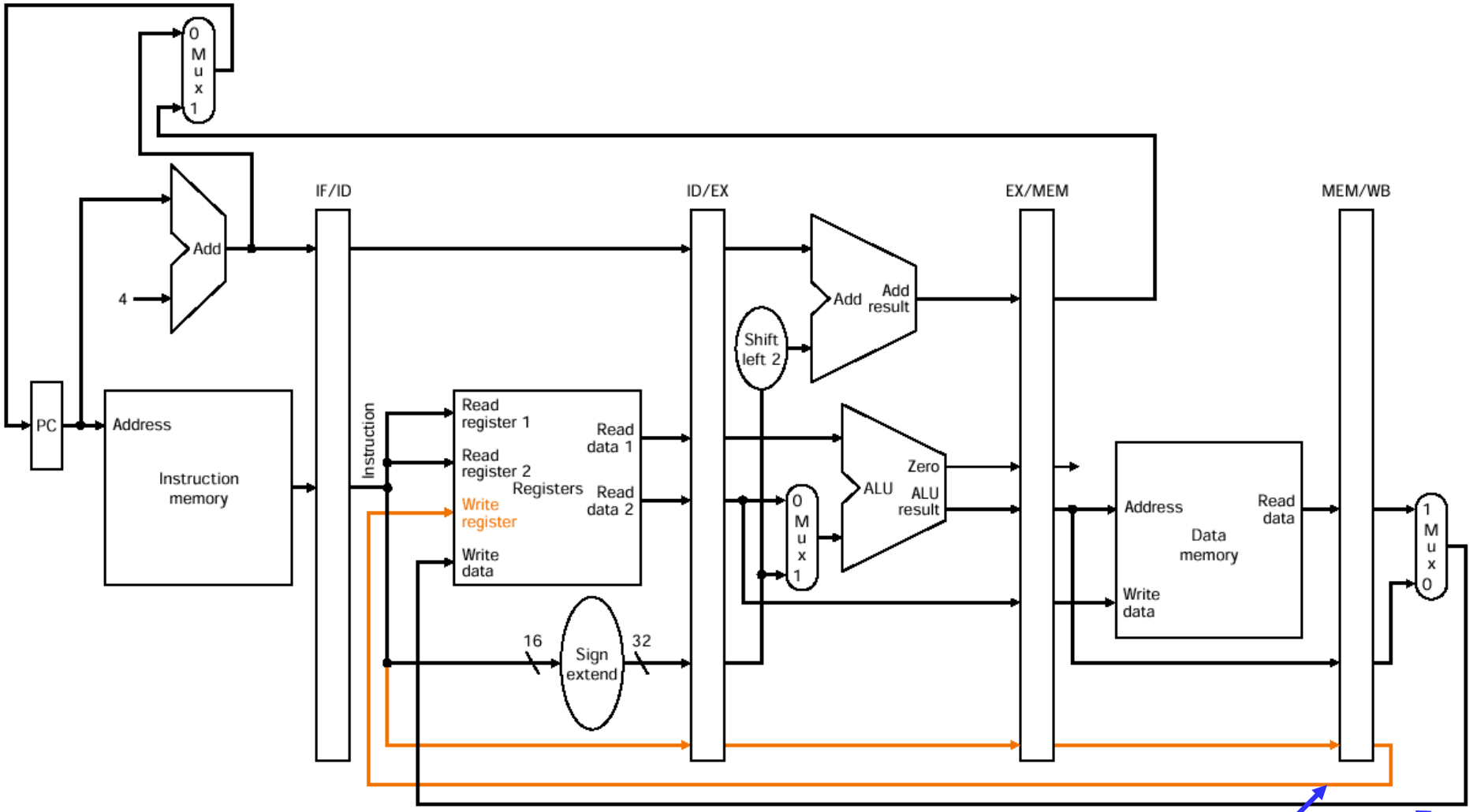
# Datapath MIPS (2)




**Questo collegamento può dar luogo a problemi ?**

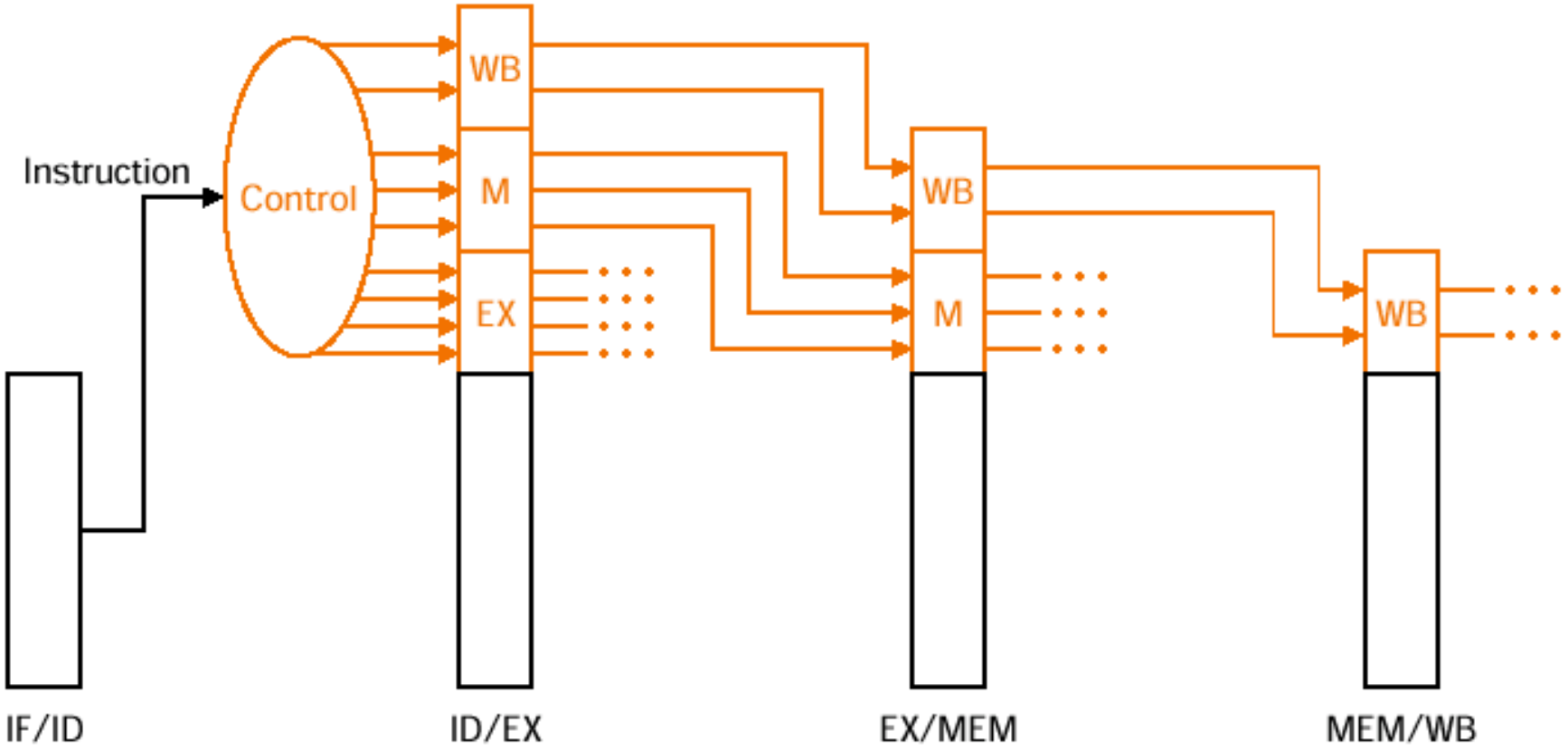
**Bypass**

# Datapath MIPS corretto




**Nello stadio WB i dati calcolati in precedenza tornano indietro, assieme al numero del registro**

# Controllo del processore pipeline




- **IF e ID devono essere eseguiti sempre, ad ogni ciclo di clock**
  - **i relativi segnali di controllo non dipendono quindi dal tipo di istruzione**
- **Il controllo, in corrispondenza di ID, calcola i segnali per tutte e 3 le fasi successive**
  - **i segnali vengono propagati attraverso i registri di interfaccia tra gli stadi (allo stesso modo dei registri letti/calcolati, valori letti dalla memoria, ecc.)**

# Pipeline e prestazioni

- Consideriamo una pipeline composta da *n stadi*
  - sia $T_{seq}$ il tempo di esecuzione sequenziale di ogni singola istruzione
  - sia $T_{stadio} = T_{seq}/n$ il tempo di esecuzione di ogni singolo stadio della pipeline
  - rispetto all'esecuzione sequenziale, lo speedup ottenibile dall'esecuzione pipeline su uno stream molto lungo di istruzioni
    - tende ad *n*

- In pratica, lo speedup non è mai uguale a *n* a causa:
  - del tempo di riempimento/svuotamento della pipeline, durante cui non tutti gli stadi sono in esecuzione
  - dello sbilanciamento degli stadi, che porta a scegliere un tempo di esecuzione di ogni singolo stadio della pipeline $T_{stadio}$, tale che
    $T_{stadio} > T_{seq}/n$
  - delle dipendenze tra le istruzioni, che ritarda il fluire nella pipeline di qualche istruzione (pipeline entra in stallo)

# Pipeline e prestazioni

- Confrontiamo l'esecuzione sequenziale (a singolo ciclo) di *IC istruzioni*, con l'esecuzione di una pipeline a *n* stadi

- Sia T è il periodo di clock del processore a singolo ciclo
- Sia T' = T/n il periodo di clock del processore pipeline
  - ogni stadio della pipeline completa quindi l'esecuzione in un tempo T/n

- Tempo di esecuzione del processore a singolo ciclo: IC * T

- Tempo di esecuzione del processore pipeline: (n-1) * T' + IC * T'
  - tempo per riempire la pipeline: (n-1) * T'
  - tempo per completare l'esecuzione dello *stream di IC istruzioni:* IC * T'
    (ad ogni ciclo, dalla pipeline fuoriesce il risultato di un'istruzione)

- Speedup = IC*T / ((n-1) * T/n + IC * T/n) = IC / ( (n-1)/n + IC/n ) = n * IC / (n –1 + IC)
  - quando IC è grande rispetto a *n* (ovvero, quando lo stream di istr. in ingresso alla pipeline è molto lungo), allora lo speedup tende proprio a *n*

# Pipeline e prestazioni

- **Confrontiamo ora l'esecuzione sequenziale (a singolo ciclo) di *IC istruzioni*, con l'esecuzione di una pipeline a *n* stadi, dove il tempo di esecuzione di ogni stadio è maggiore T', dove T' > T/n**

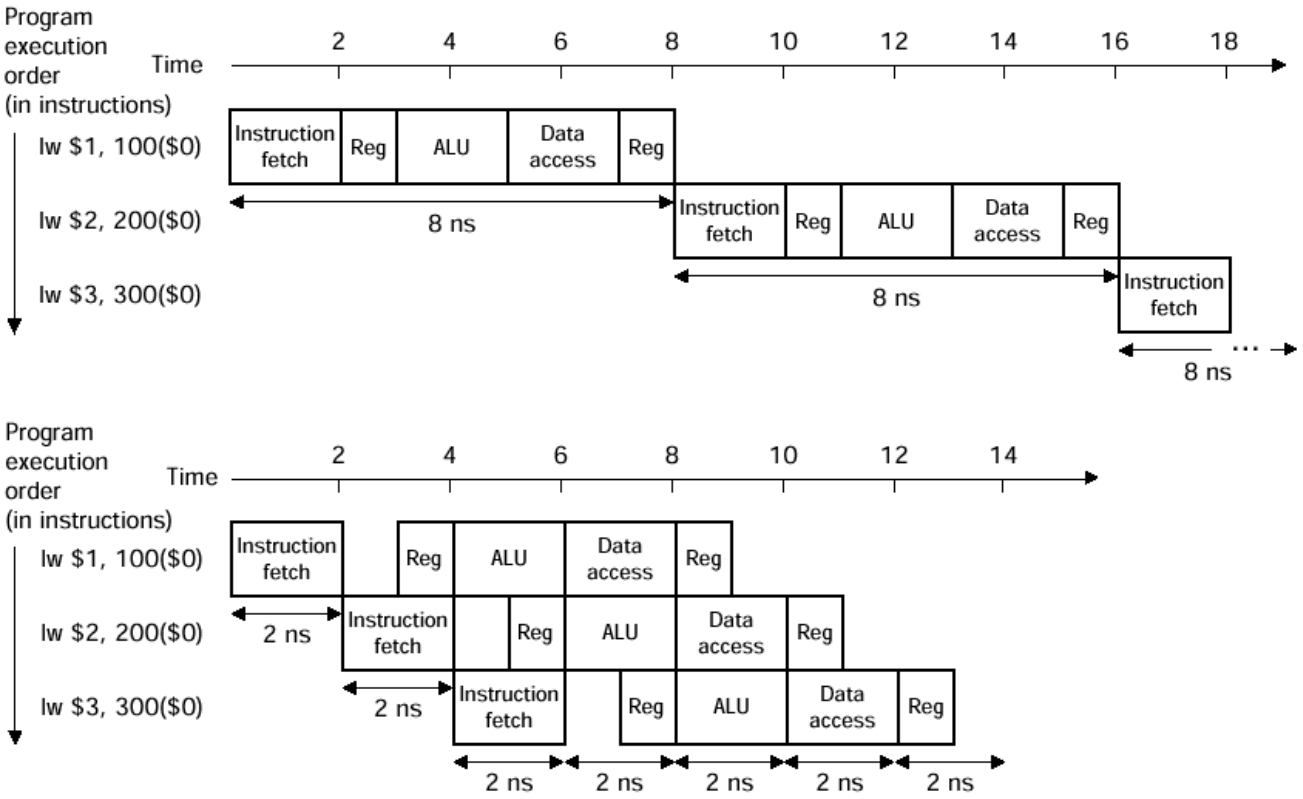
- **Tempo di esecuzione del processore a singolo ciclo: IC * T**
- **Tempo di esecuzione del processore pipeline: (n-1) * T' + IC * T'**

- **Speedup = IC * T / ((n-1) * T' + IC * T')**
  - **quando IC è grande rispetto a *n* (ovvero, quando lo stream di istr. in ingresso alla pipeline è molto lungo), allora lo speedup tende a T/T'**

# Esempio con 3 istruzioni

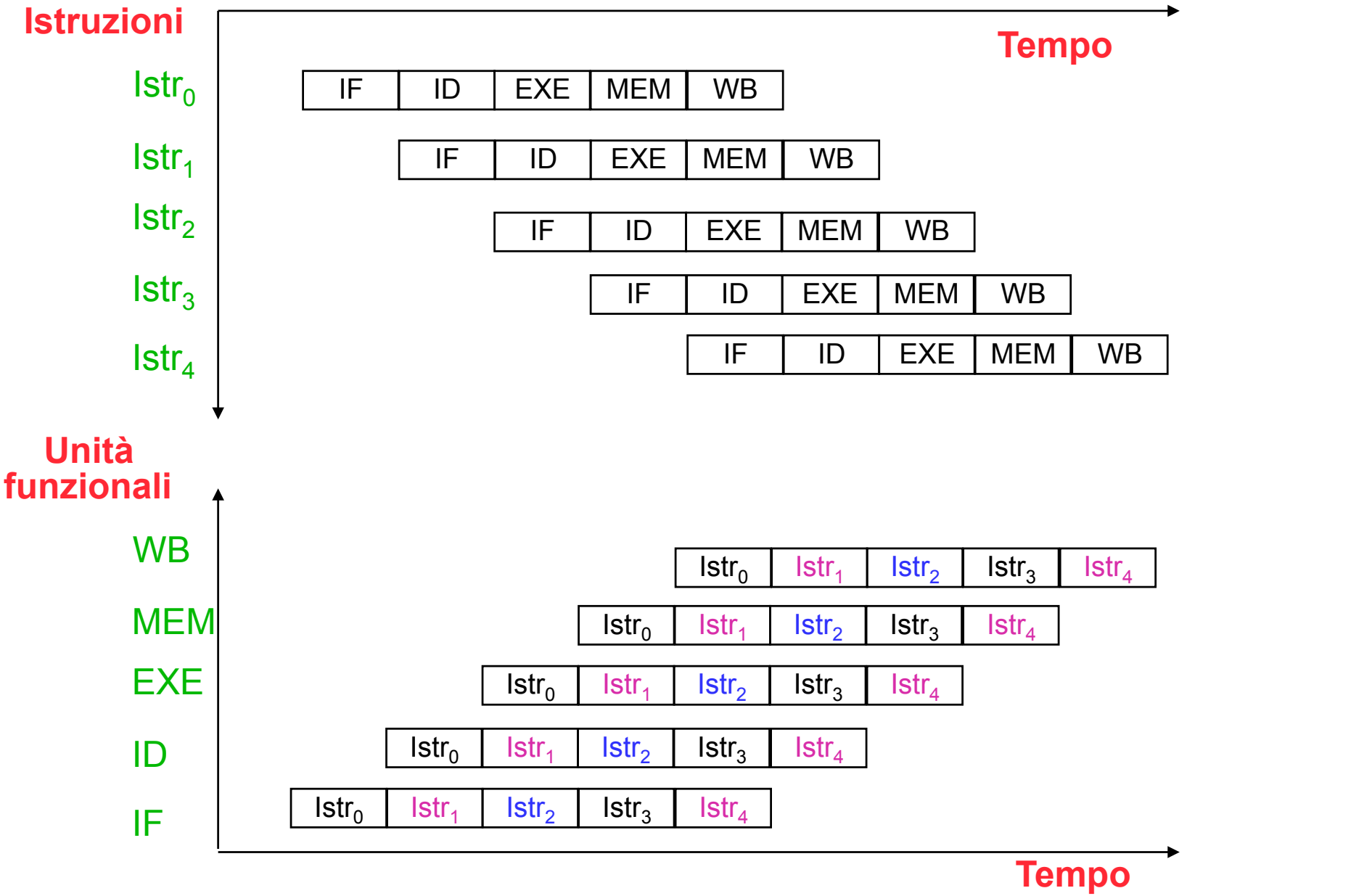
- **Pipeline a 5 stadi (n=5)**
- **T= 8 ns**
- **T' = 2 ns, dove**
  **T' > T/n = T/5 = 1.6**

- **Tempo di esecuzione singolo ciclo:**
  **IC * T = 3 * 8 = 24 ns.**
  **Tempo di esecuzione pipeline:**
  **(n-1) * T' + IC * T' =**
  **4*2 + 3*2 = 14 ns.**
- **Speedup = 24/14 = 1.7**




- **Ma se lo stream di istruzioni fosse più lungo, es. IC = 1003**
  - **Tempo di esecuzione singolo ciclo: IC * T = 1003 * 8 = 8024 ns.**
  - **Tempo di esecuzione pipeline: (n-1) * T' + IC * T' = 4*2 + 1003*2 = 2014 ns.**
  - **Speedup = 8024/2014 = 3.98 ≈ T / T' = 8/2 = 4**
- ***L'organizzazione pipeline aumenta il throughput dell'esecuzione delle istruzioni…. ma può aumentare la latenza di esecuzione delle singole istruzioni***

# Diagrammi temporali alternativi



# Criticità (hazard)

- **Negli esempi precedenti le istruzioni entrano nella pipeline (stadio IF) una dopo l'altra, senza interruzioni**

- **In realtà, a causa delle cosiddette criticità, alcune istruzioni non possono proseguire l'esecuzione (o entrare nella pipeline) finché le istruzioni precedenti non hanno prodotto il risultato corretto**
  - **criticità: l'esecuzione dell'istruzione corrente *dipende* dai risultati dell'istruzione precedente. Ma l'istruzione precedente è già stata inviata, si trova ancora nella pipeline e non ha completato l'esecuzione**

- **L'effetto delle criticità è lo stallo della pipeline**
  - **lo stadio che ha scoperto la criticità, assieme agli stadi precedenti**
    - **rimangono in stallo (in pratica, rieseguono la stessa istruzione)**
    - **viene propagata una nop (no operation) alle unità seguenti nella pipeline (bolla)**
  - **lo stallo può prorogarsi per diversi cicli di clock (e quindi più bolle dovranno essere propagate nella pipeline, svuotando gli stadi successivi della pipeline)**

# Tipi di criticità

- **Criticità strutturali**
  - **l'istruzione ha bisogno di una risorsa (unità funzionale) usata e non ancora liberata da un'istruzione precedente (ovvero, da un'istruzione che non è ancora uscita dalla pipeline)**
  - **es.: cosa succederebbe se usassimo una sola memoria per le istruzioni e i dati ?**

- **Criticità sui dati**
  - **dipendenza sui dati tra istruzioni**
  - **es.: dipendenza RAW (Read After Write) : un'istruzione legge un registro scritto da un'istruzione precedente**
    - **l'esecuzione dell'istruzione corrente deve entrare in stallo, finché l'istruzione precedente non ha completato la scrittura del registro**
  - **Esempio:**

```
add $s1, $t0, $t1    # Write $s1
sub $s2, $s1, $s3    # Read  $s1
```

- **Criticità sul controllo**
  - **finché le istruzioni di branch non hanno calcolato il nuovo PC, lo stadio IF non può effettuare il fetch corretto dell'istruzione**

# Criticità sui dati

- **Le dipendenze sui dati tra coppie di istruzioni implica un ordine di esecuzione relativo non modificabile**
  - **non possiamo invertire l'ordine di esecuzione**

- **WAW (Write After Write) : un'istruzione scrive un registro scritto da un'istruzione precedente**

```
add $s1, $t0, $t1    # Write $s1
. . .
sub $s1, $s2, $s3    # Write  $s1
```

- **WAR (Write After Read ) : un'istruzione scrive un registro letto da un'istruzione precedente**

```
add $t0, $s1, $t1    # Read $s1
sub $s1, $s2, $s3    # Write  $s1
```

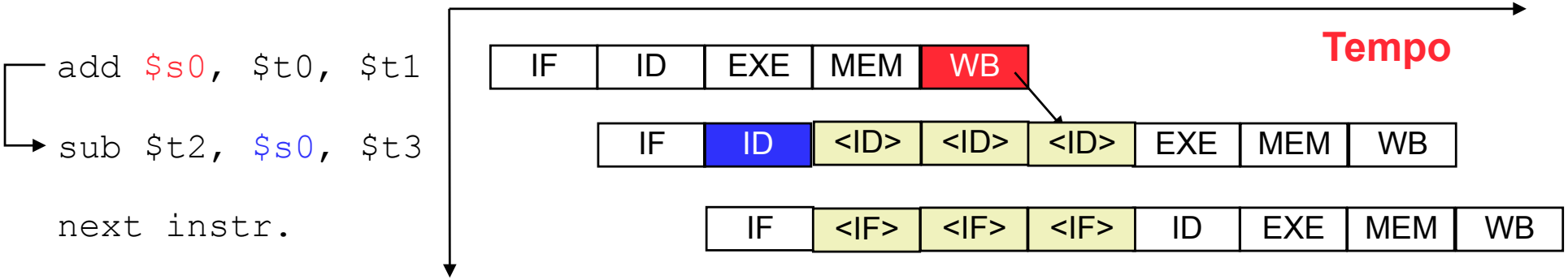
- **RAW (Read After Write) : un'istruzione legge un registro scritto da un'istruzione precedente**

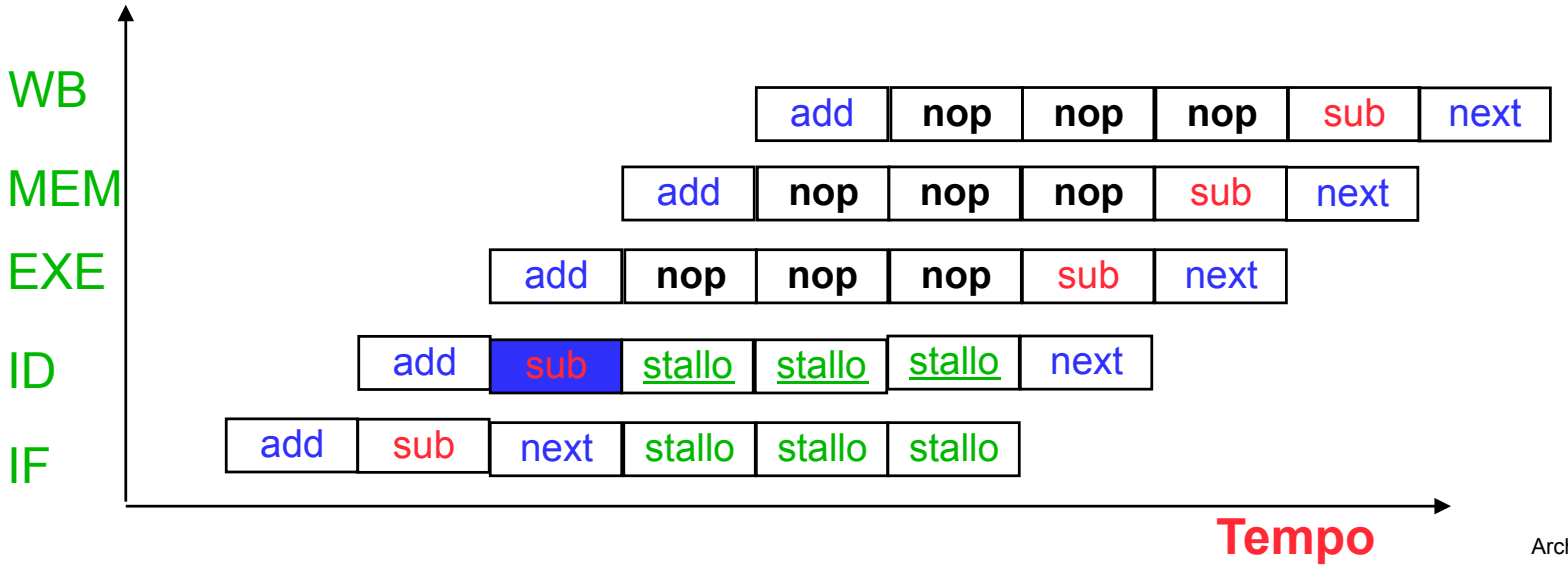
```
add $s0, $t0, $t1    # Write $s0
sub $t2, $s0, $t3    # Read  $s0
```

- **RAR (Read After Read): non è una dipendenza. Possiamo anche invertire l'ordine di esecuzione**

# Problemi con le pipeline

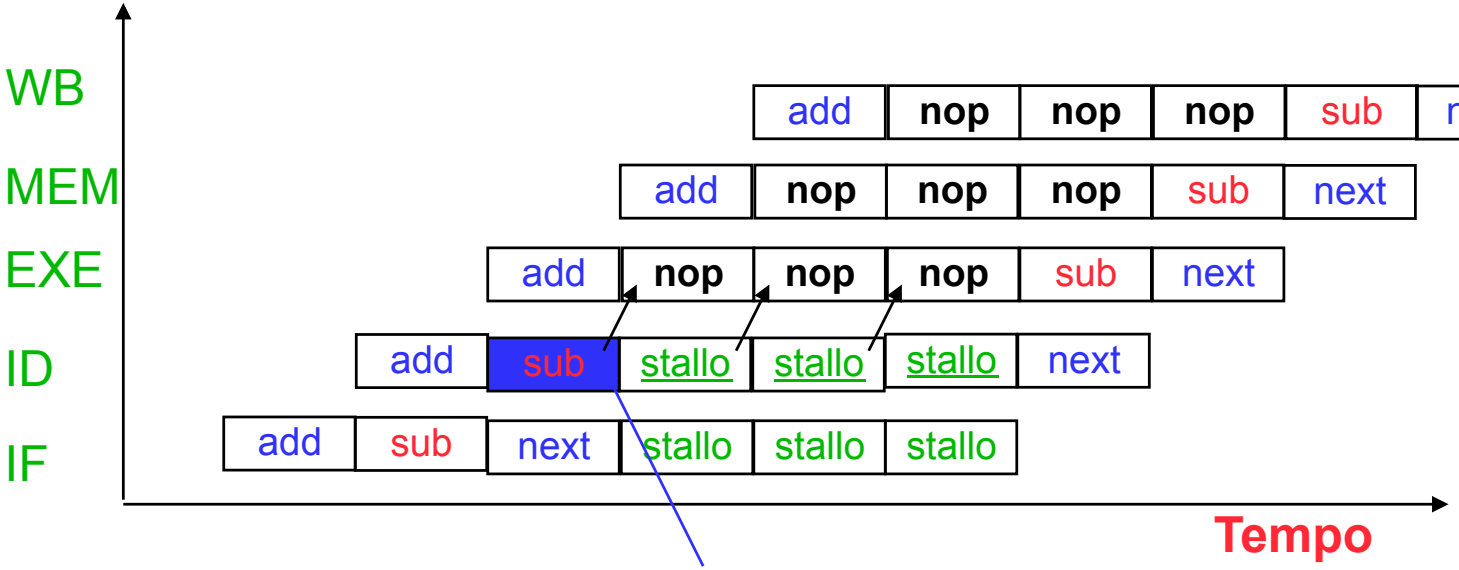
- **Anche se l'ordine di esecuzione delle istruzioni non viene modificato, l'esecuzione in pipeline comporta dei problemi a causa del parallelismo**
  - **problemi dovuti alle dipendenze RAW**



# Hazard detection unit

- **La necessità di mettere in stallo la pipeline viene individuata durante lo stadio ID della istruzione `sub`**
  - **lo stadio ID (impegnato nella `sub`) e lo stadio IF (impegnato nella fetch della `next instruction`) rimangono quindi in stallo per 3 cicli**
  - **lo stadio ID propaga 3 `nop` (bolle) lungo la pipeline**




```
add $s0, $t0, $t1
sub $t2, $s0, $t3
next instr.
```

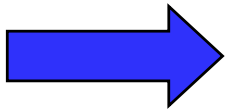
**L'*hazard detection unit* fa parte dello stadio ID. In questo caso, l'unità provoca lo stallo quando l'istruzione `sub` entra nello stadio ID.**

**L'unità confronta i numeri dei registri usati dalla `sub` e dall'istruzione precedente (`add`).**

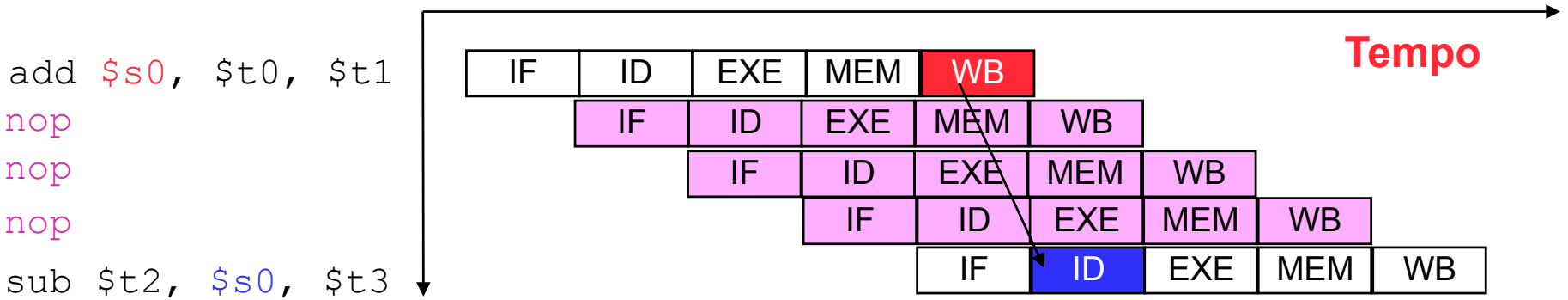
# Soluzione software alle criticità sui dati

- **Può il compilatore garantire la corretta esecuzione della pipeline anche in presenza di dipendenze sui dati?**
  - **sì, può esplicitamente inserire delle "nop" in modo da evitare esecuzioni scorrette**
  - **stalli espliciti**
  - **progetto del processore semplificato (non c'è bisogno dell'hazard detection unit)**

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
...
```

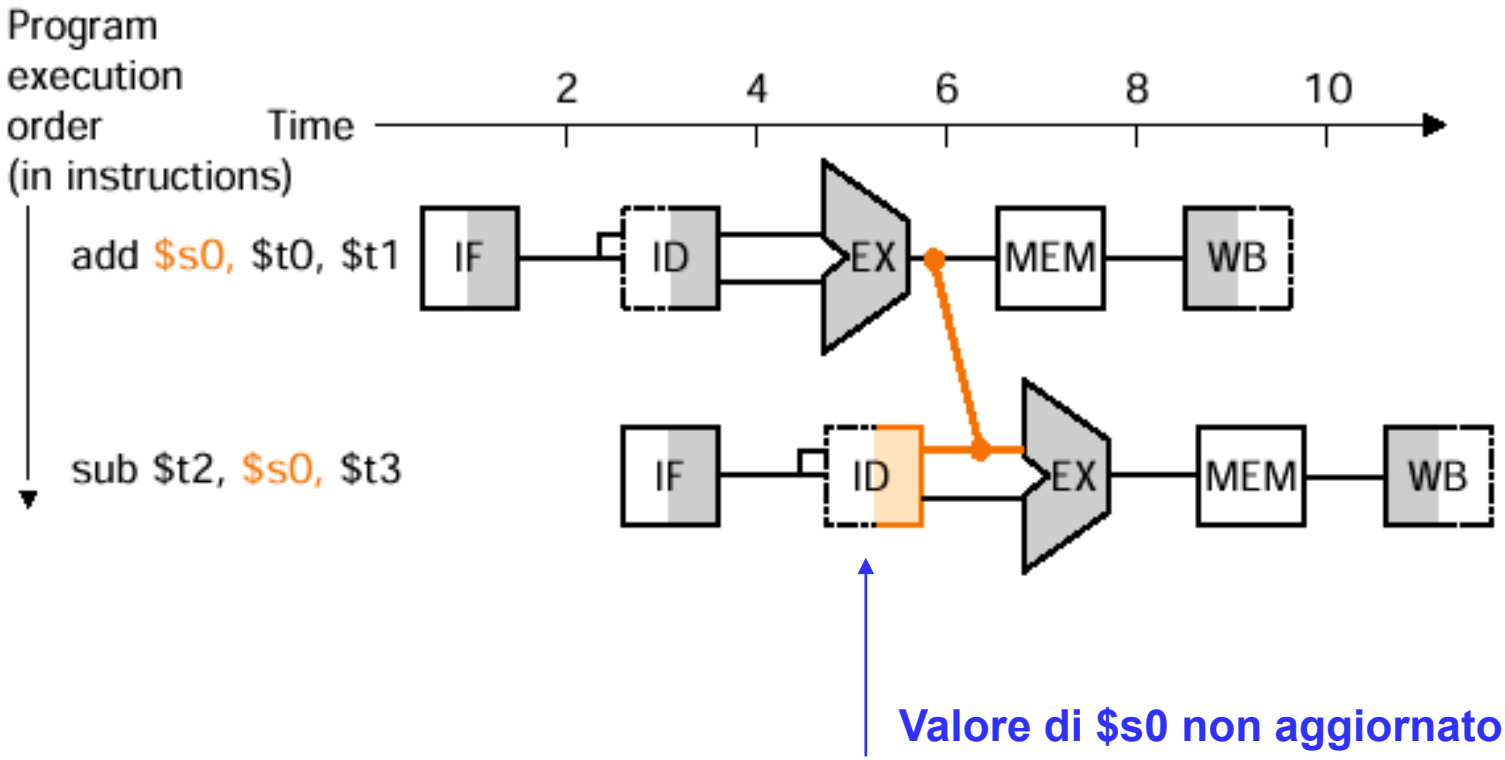


```
add $s0, $t0, $t1
nop
nop
nop
sub $t2, $s0, $t3
...
```

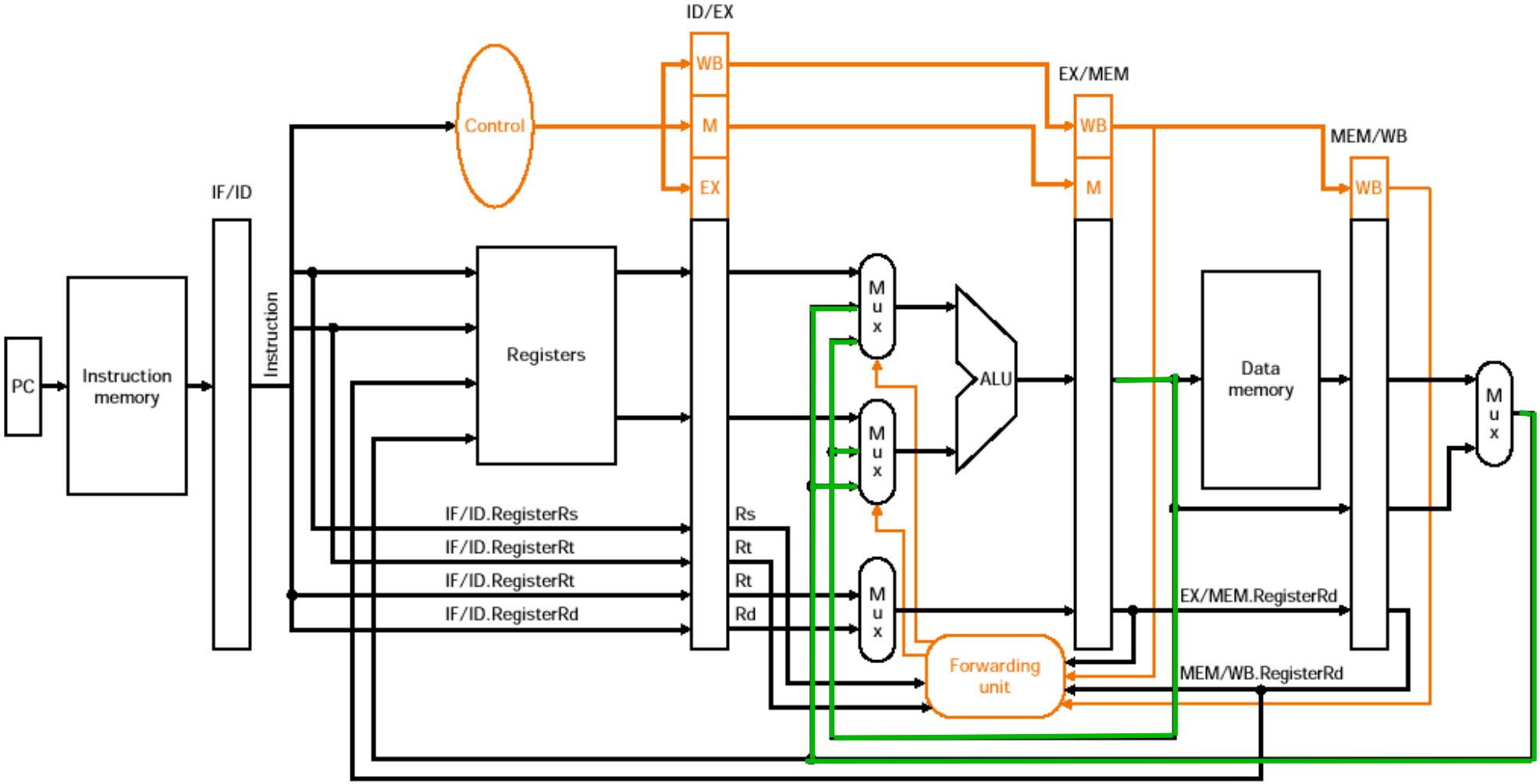


# Forwarding

- **Tramite il forwarding possiamo ridurre i cicli di stallo della pipeline**
- **Nuovo valore del registro** `$s0`
  - **prodotto nello stadio EXE della** `add`
  - **usato nello stadio EXE della** `sub`

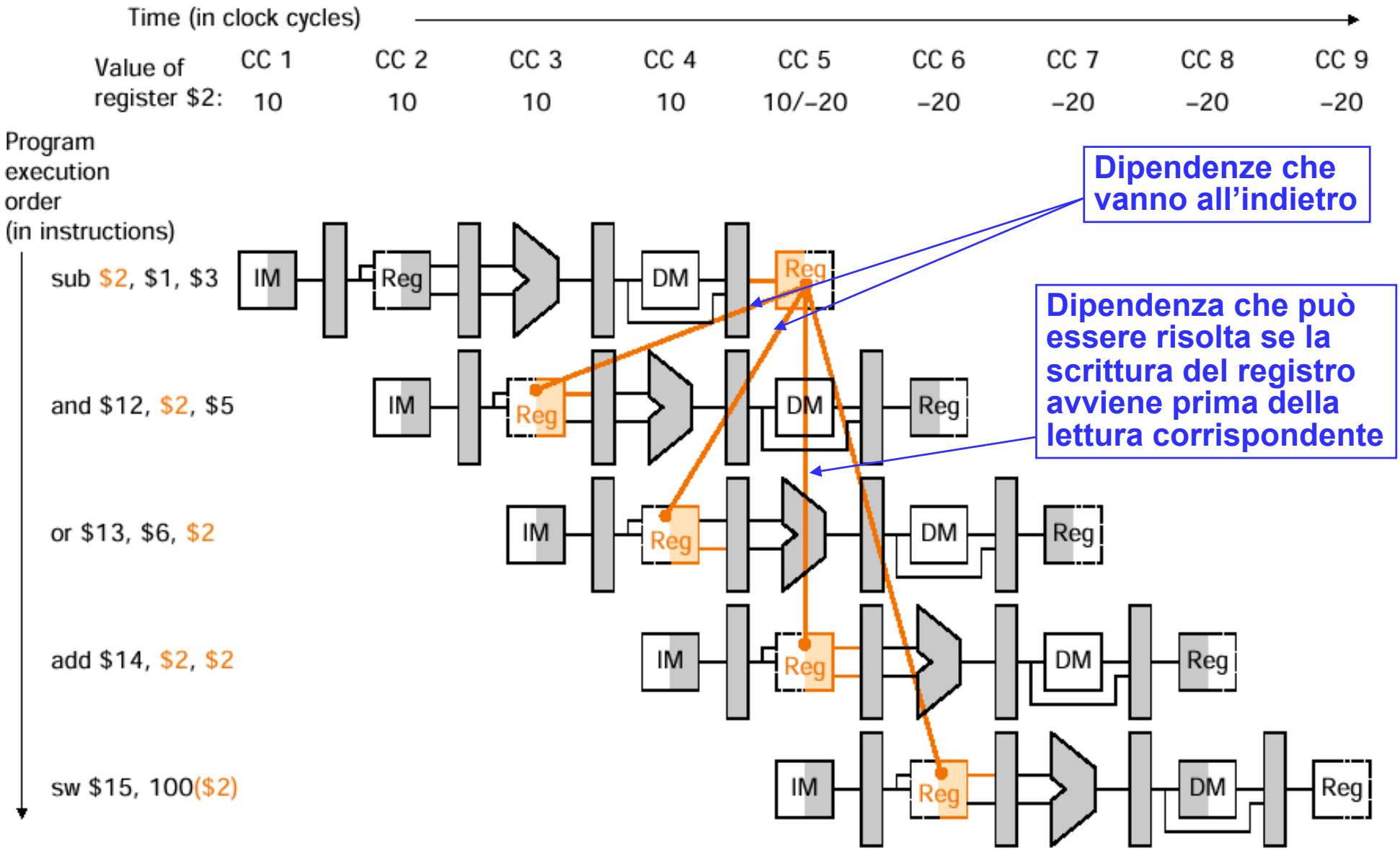


# Forwarding e datapth

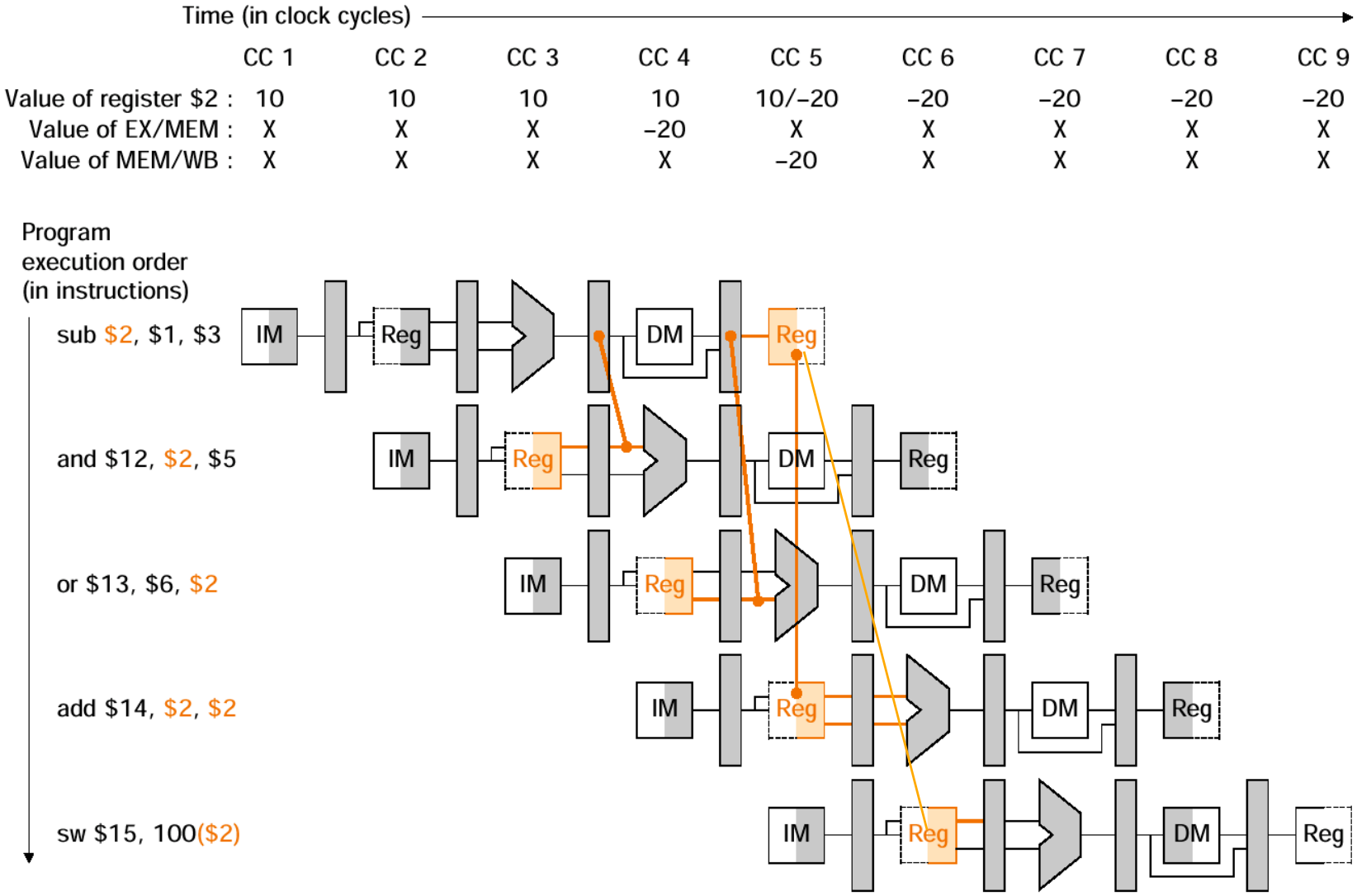



- **Per permettere il forwarding, i valori calcolati durante gli stadi successivi devono tornare indietro verso lo stadio EXE**
  - **vedi linee evidenziate in verde**

# Dipendenze RAW in una sequenza di istruzioni



# Risolvere le dipendenze tramite forwarding

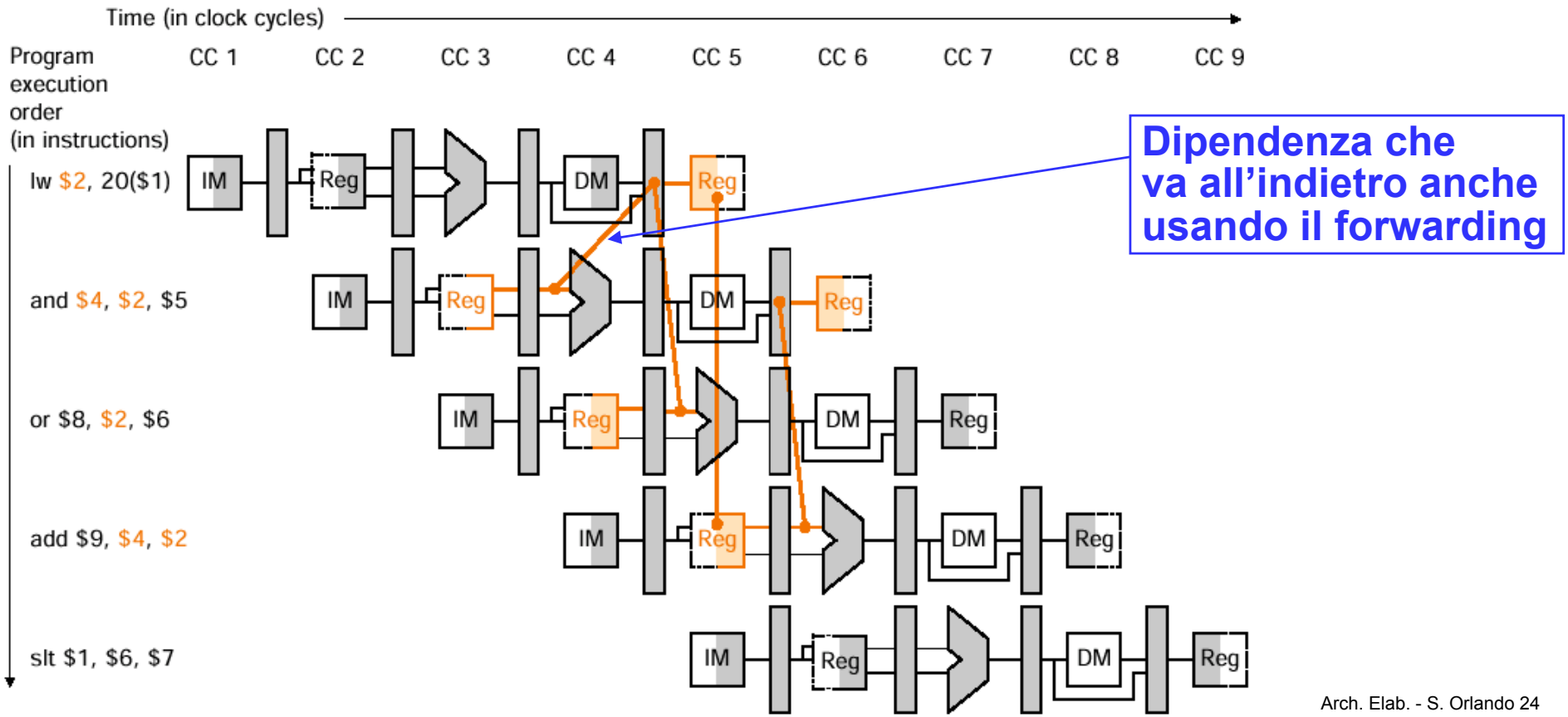



- **Il Register file permette il forwarding: scrive un registro nella prima parte del ciclo, e legge una coppia di registri nella seconda parte del ciclo**

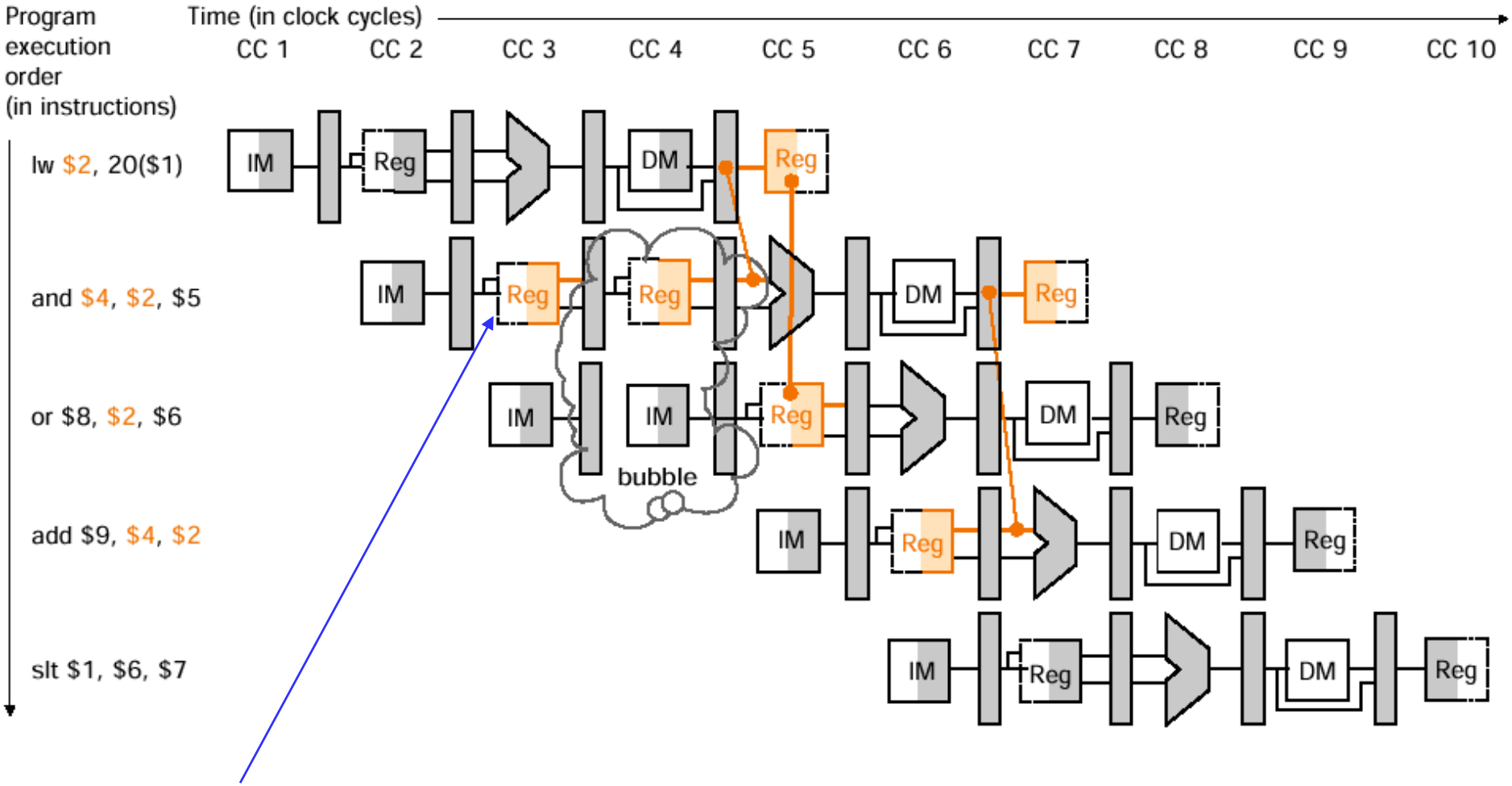
# Problema con le `lw`

- Le **load** producono il **valore** da memorizzare nel **registro target** durante lo **stadio MEM**
- Le istruzioni aritmetiche e di branch che seguono, e che leggono lo **stesso registro,** hanno bisogno del valore corretto del registro durante lo **stadio EXE**

  ⇒ **stallo purtroppo inevitabile, anche usando il forwarding**



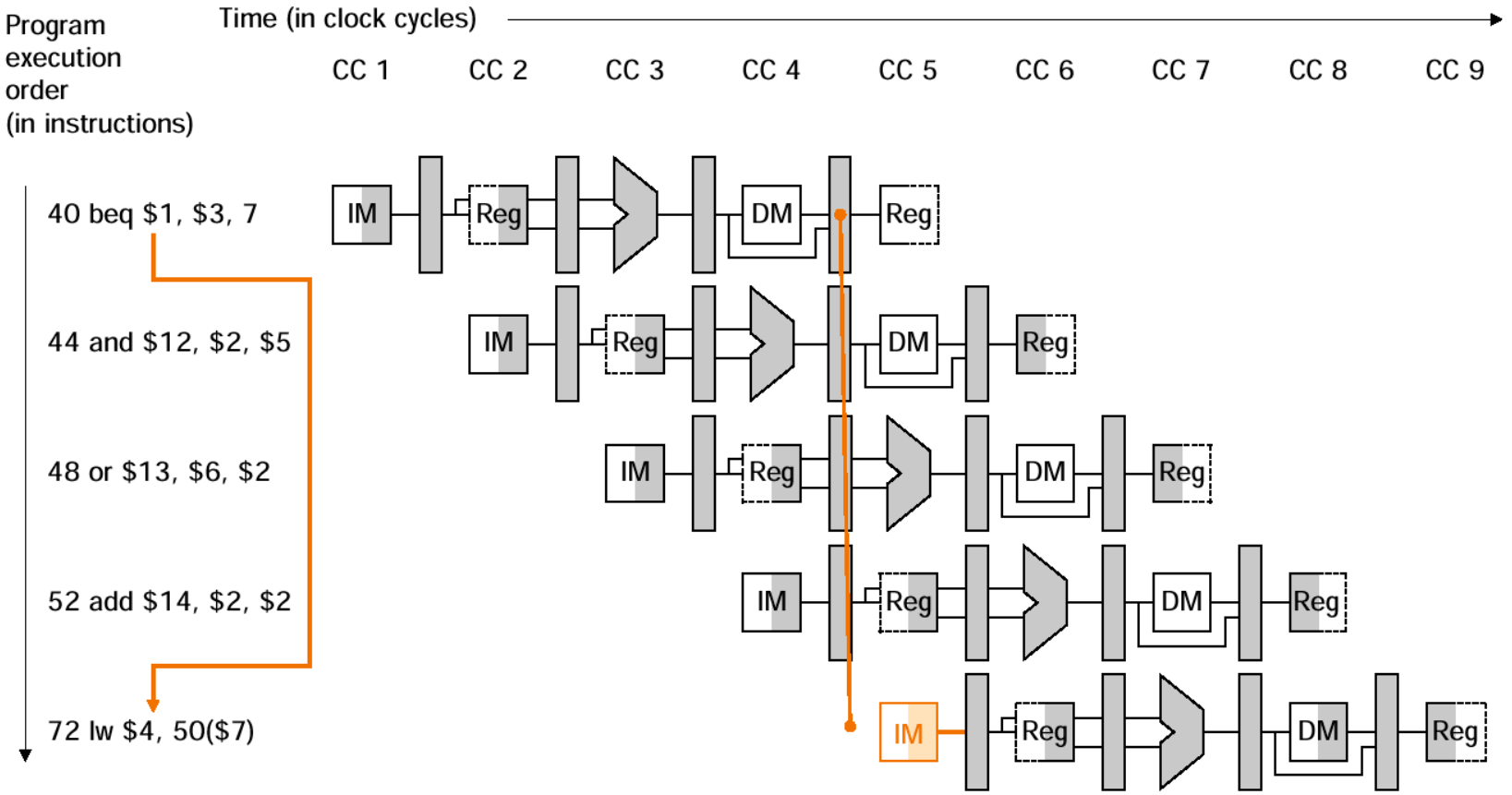
# Load e hazard detection unit




**Criticità scoperta nello stadio ID dell'istruzione `and`**
**Le istruzioni `and` e `or` rimangono per un ciclo nello stesso stadio (rispettivamente IF e ID), e viene propagata una `nop` (bubble)**

# Criticità sul controllo

- **Nuovo valore del PC calcolato dal *branch* viene *memorizzato* durante MEM**
  - **se il branch è taken, in questo caso abbiamo che le 3 istruzioni successive sono già entrate nella pipeline, ma fortunatamente non hanno ancora modificato registri**
  - **dobbiamo annullare le 3 istruzioni: l'effetto è simile a quello che avremmo ottenuto se avessimo messo in stallo la pipeline fino al calcolo dell'indirizzo del salto**



# Riduciamo gli stalli dovuti alla criticità su controllo

- **Anticipiamo il calcolo di PC e il confronto tra i registri della `beq`**
  - **spostiamo in ID l'addizionatore che calcola l'indirizzo target del salto**
  - **invece di usare la ALU per il confronto tra registri, il confronto può essere effettuato in modo veloce da un'unità specializzata**
    - tramite lo XOR bit a bit dei due registri, e un OR finale dei bit ottenuti (se risultato è 1, allora i registri sono diversi)
    - quest'unità semplificata può essere aggiunta allo stadio ID

- **In questo caso, se il branch è taken, e l'istruzione successiva è già entrata nella pipeline**
  - **solo questa istruzione deve essere eliminata dalla pipeline**

# Eliminare gli stalli dovuti alle criticità sul controllo

- Attendere sempre che l'indirizzo di salto sia stato calcolato correttamente porta comunque a rallentare il funzionamento della pipeline
  - è una soluzione conservativa, che immette sempre bolle nella pipeline
  - i branch sono purtroppo abbastanza frequenti nel codice

- Un modo per eliminare gli stalli, è quella di prevedere il risultato del salto condizionato
  - lo stadio IF potrà quindi, da subito, effettuare il fetch "corretto" della prossima istruzione da eseguire

- Problema:
  - cosa succede se la previsione non risulterà corretta ?
  - sarà necessario ancora una volta eliminare le istruzioni che nel frattempo sono entrate nella pipeline
  - sarà necessaria un'unità che si accorga dell'hazard, e che si occupi di eliminare dalla pipeline le istruzioni che vi sono entrate erroneamente

# Previsione semplice

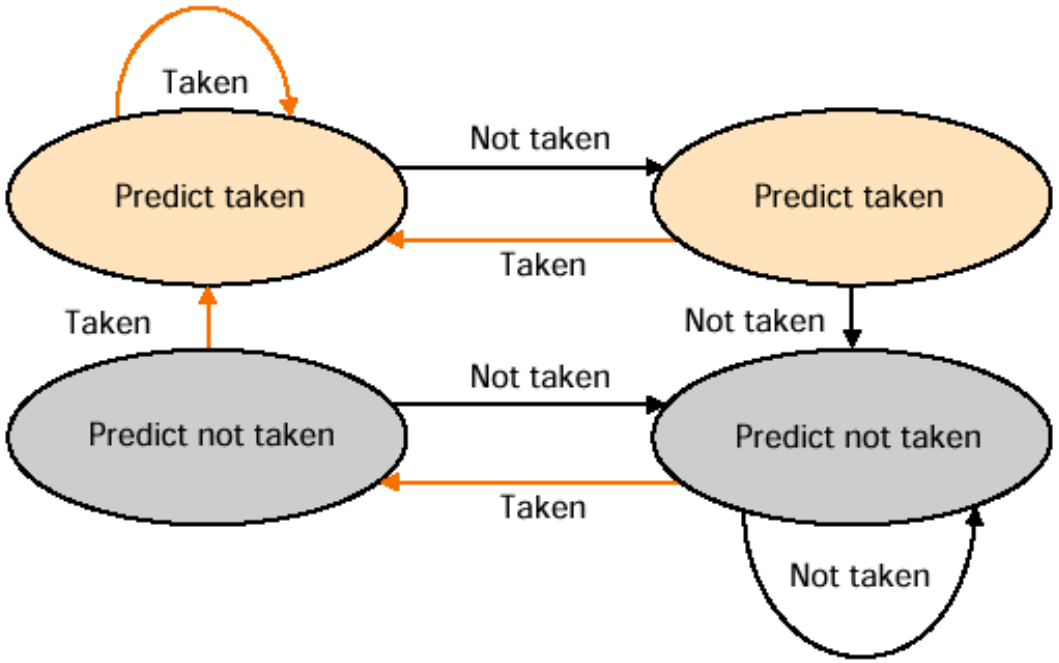
- **Ipotizziamo che il salto condizionato sia sempre *not-taken***
  - **abbiamo già visto questo caso**
  - **l'istruzione da eseguire successivamente al salto è quella seguente (PC+4)**

- **Se almeno nella metà dei casi il salto è *not-taken*, questa ottimizzazione dimezza i possibili stalli dovuti alla criticità del controllo**

# Previsione dinamica

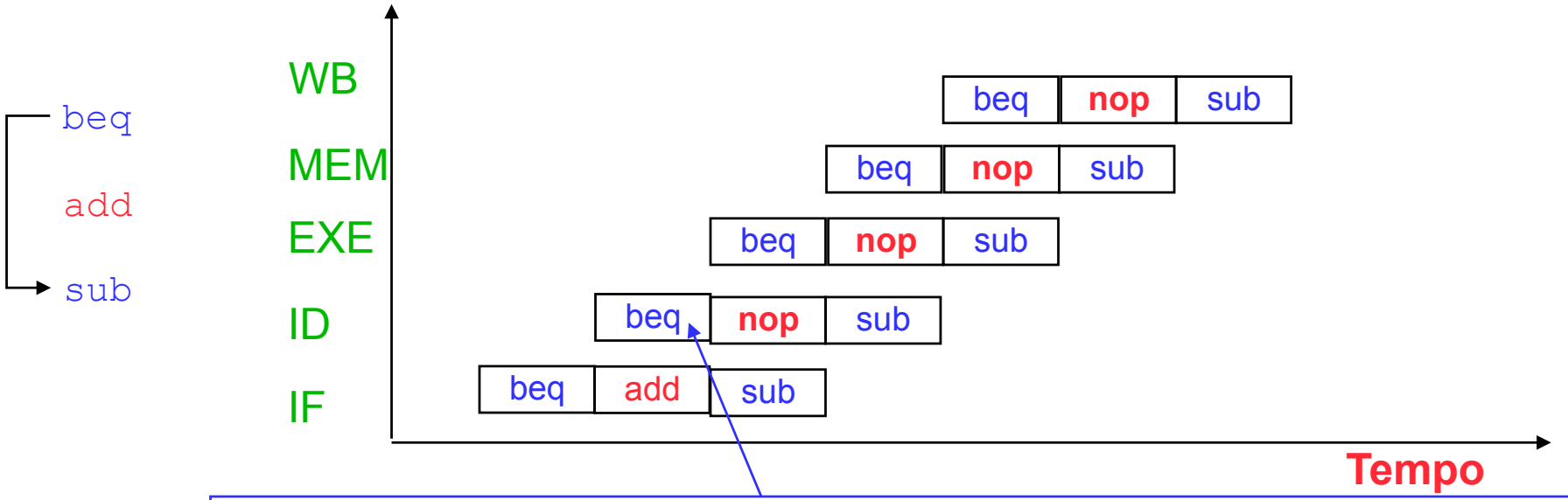
- Manteniamo una *history table* (tabella della storia dei salti)
  - indirizzata tramite gli indirizzi  delle istruzioni di salto
  - nella tabella poniamo anche l'indirizzo dell'istruzione successiva al salto nel caso di *brach taken*
- Nella tabella viene memorizzato
  - 1 o più bit per mantenere la storia riguardo al passato dell'esecuzione di ciascun salto (taken o not-taken)

- Ogni entry della *history table* è associato con 4 possibili stati
- Automa a stati finiti per modellare le transizioni di stato
  - *2 bit* per codificare i 4 stati
  - una sequenza di previsioni corrette (es. *taken*) non viene influenzata da sporadiche previsioni errate



# Hazard detection unit

- **Ancora, come nel caso delle dipendenze sui dati**
  - **unità di controllo per individuare possibili criticità sul controllo**
  - **nella semplice soluzione prospettata, l'unità può essere posizionata nello stadio ID**
  - **se l'istruzione caricata nello stadio IF non è quella giusta, bisogna annullarla, ovvero forzarne il proseguimento nella pipeline come se fosse una `nop` (bubble)**



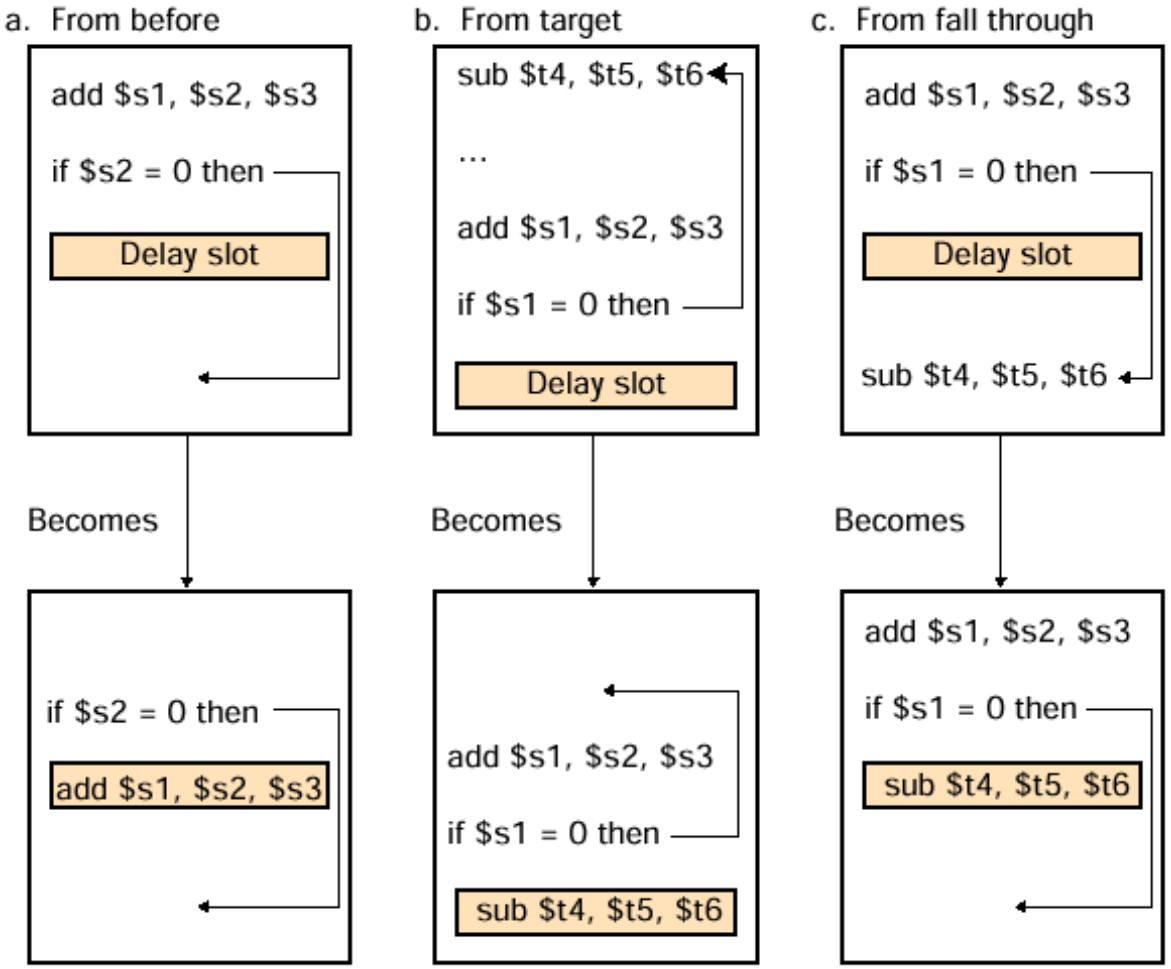

**Il calcolo dell'indirizzo corretto del PC avviene qui (stadio ID di `beq`)**

**Sempre in ID si sovra-scrive l'istruzione appena letta dallo stadio IF precedente, in modo che questa prosegua come se fosse una `nop`**

# Delayed branch

- Processori moderni fanno affidamento
  - sulla previsione dei salti, e
  - sull'annullamento delle istruzioni caricate in caso di previsione errata
- Il vecchio processore MIPS usava una tecnica molto più semplice, che non richiede hardware speciale, facendo affidamento solo sul software
  - l'indirizzo del salto viene calcolato nello stadio ID dell'istruzione branch
  - l'istruzione posta successivamente al salto entra comunque nella pipeline e viene completata
  - è compito del compilatore/assemblatore porre successivamente al salto
    - una `nop` esplicita, oppure
    - un'istruzione del programma che, anche se completata, non modifica la semantica del programma (es. viene rispettato l'ordinamento tra le istruzioni determinato dalle dipendenze sui dati)
- La tecnica è nota come salto ritardato: il ritardo corrisponde ad un certo numero di *branch delay slot*
  - slot da riempire con istruzioni da eseguire comunque dopo il branch, prima che l'indirizzo del salto venga calcolato (nel MIPS, delay slot = 1)
  - i processori moderni, che inviano più istruzioni contemporaneamente e hanno pipeline più lunghe, avrebbero bisogno di un grande numero di delay slot ! ⇒ *difficile trovare tante istruzioni eseguibili nello slot*

# Delayed branch




- **Nel caso a), bisogna che l'istruzione possa essere spostata in accordo alle dipendenze sui dati**
  - **$s1 non è letto dalla beq**
  - **Non esistono dipendenze RAW, WAR, WAW con beq**
- **Nei casi b) e c), il registro assegnato ($t4) potrebbe essere stato modificato erroneamente**
  - **se il branch non segue il flusso previsto, è necessario che il codice relativo non abbia necessità di leggere, come prima cosa, il registro $t4**
  - **ad esempio, prima assegna $t4 e poi lo usa**

# Esempio di delay branch e ottimizzazione relativa

- **Individua in questo programma le dipendenze tra le istruzioni, e trova un'istruzione prima del branch da spostare in avanti, nel branch delay slot**

```
Loop:  lw $t0, 0($s0)
       addi $t0, $t0, 20
       sw $t0, 0($s1)
       addi $s0, $s0, 4
       addi $s1, $s1, 4
       bne $s0, $a0, Loop
       < delay slot >
```

**Dipendenze RAW**

**Dipendenze WAR**

# Esempio di delay branch e ottimizzazione relativa

- **Individua in questo programma le dipendenze tra le istruzioni, e trova un'istruzione prima del branch da spostare in avanti, nel branch delay slot**

```
Loop:  lw $t0, 0($s0)
       addi $t0, $t0, 20
       sw $t0, 0($s1)
       addi $s0, $s0, 4
       addi $s1, $s1, 4
       bne $s0, $a0, Loop
       < delay slot >
```

**Le dipendenze in rosso sono di tipo RAW**

**Le dipendenze in verde sono di tipo WAR**

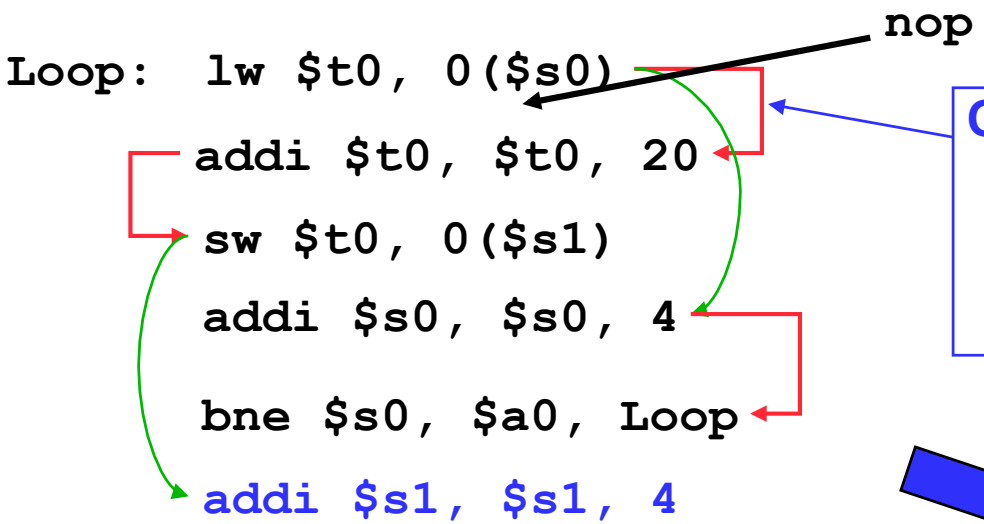
**L'unica istruzione che possiamo spostare in avanti, senza modificare l'ordine di esecuzione stabilito dalle dipendenze, è:**

```
addi $s1, $s1, 4
```

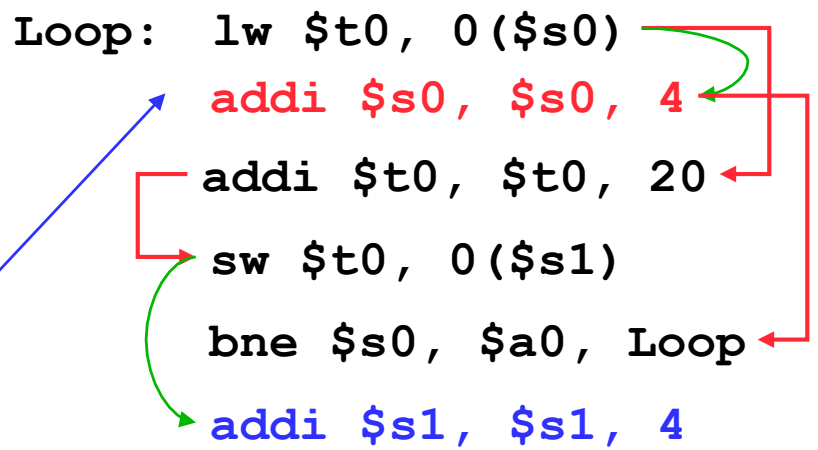
```
Loop:  lw $t0, 0($s0)
       addi $t0, $t0, 20
       sw $t0, 0($s1)
       addi $s0, $s0, 4
       bne $s0, $a0, Loop
       addi $s1, $s1, 4
```

# Rimozione statica degli stalli dovuti alle load

- **Il processore con forwarding non è in grado di eliminare lo stallo dopo la `lw` se è presente una dipendenza RAW verso l'istruzione successiva**




**Questa dipendenza provoca uno stallo: rispetto al comportamento della pipeline, è come se ci fosse una nop tra la lw e la addi**




**Per eliminare lo stallo, possiamo trovare un'istruzione dopo (o prima della lw) da spostare nel load delay slot**

**Nell'esempio, possiamo spostare indietro, senza modificare l'ordine di esecuzione stabilito dalle dipendenze, l'istruzione:**

```
addi $s0, $s0, 4
```

# Confronto tra diversi schemi di controllo

- Sappiamo che
  - lw: 22% IC  sw: 11% IC  R-type: 49% IC  branch: 16% IC  jump: 2% IC
- Singolo ciclo
  - Ciclo di clock (periodo) = 8 ns
    - calcolato sulla base dell'istruzione più "costosa": `lw`
  - CPI =1
  - $T_{singolo}$= IC * CPI * Periodo_clock = IC * 8 ns
- Multiciclo
  - Ciclo di clock (periodo) = 2 ns
    - calcolato sulla base del passo più "costoso"
  - $CPI_{avg}$ = 0.22 $CPI_{lw}$ + 0.11 $CPI_{sw}$ + 0.49 $CPI_{R}$ + 0.16 $CPI_{br}$ + 0.02 $CPI_{j}$ = 0.22 * 5 + 0.11 * 4+ 0.49 * 4 + 0.16 * 3 + 0.02 * 3 = 4.04
  - $T_{multi}$= IC * $CPI_{avg}$ * Periodo_clock = IC * 4.04 * 2 ns = IC * 8.08 ns

# Confronto tra diversi schemi di progetto

- **Pipeline**
  - Ciclo di clock (periodo) = 2 ns, calcolato sulla base dello stadio più "costoso"
  - Nella determinazione del CPI non considerare il tempo di riempimento della pipeline (piccolo)
    - CPI = 1: significa SOLO che un'istruzione è completata per ogni ciclo di clock
    - $CPI_{sw} = 1$ $CPI_R = 1$ $CPI_j = 2$
    - per il 50% dei casi
      - lw seguita da un'istruzione che legge il registro scritto (stallo di 1 ciclo)
      - $CPI_{lw} = 1.5$
  - Per il 25% dei casi
    - previsione dell'indirizzo del salto errata (eliminazione dell'istruzione entrata erroneamente nella pipeline, e quindi un ciclo in più dopo il branch)
    - $CPI_{br} = 1.25$
  - $CPI_{avg} = 0.22\ CPI_{lw} + 0.11\ CPI_{sw} + 0.49\ CPI_R + 0.16\ CPI_{br} + 0.02\ CPI_j =$ $0.22 * 1.5 + 0.11 * 1 + 0.49 * 1 + 0.16 * 1.25 + 0.02 * 2 = 1.17$
  - $T_{pipe} = IC * CPI_{avg} * Periodo\_clock = IC * 1.17 * 2\ ns = IC * 2.34\ ns$

- **Speedup**
  - $T_{singolo} / T_{pipe} = 8 / 2.34 = 3.42$ $T_{multi} / T_{pipe} = 8.08 / 2.34 = 3.45$

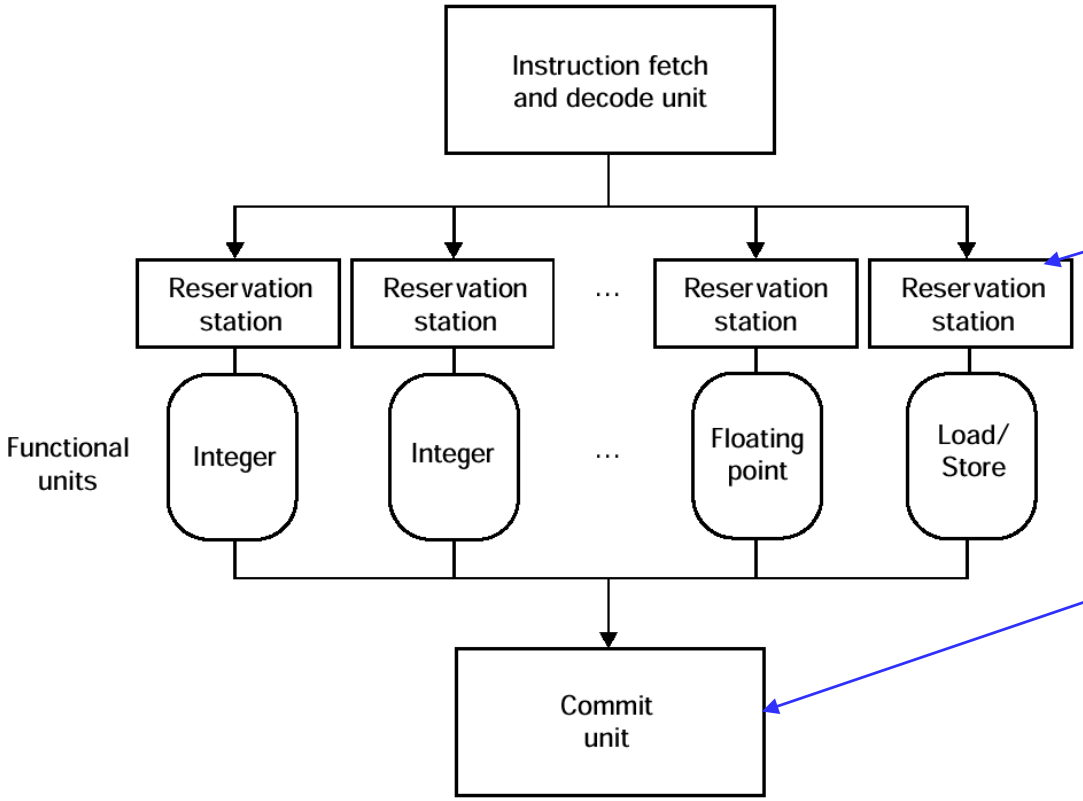
# Pipeline e eccezioni/interruzioni

- **Il verificarsi di un'eccezione è legata all'esecuzione di una certa istruzione**
  - **le istruzioni precedenti devono essere completate**
  - **l'istruzione che ha provocato l'eccezione e quelle successive devono essere eliminate dalla pipeline (trasformate in `nop`)**
  - **deve essere fetched la prima routine dell'*exception handler***

- **Le interruzioni sono asincrone, ovvero non sono legate ad una particolare istruzione**
  - **siamo più liberi nello scegliere quale istruzione interrompere per trattare l'interruzione**

# Processori superscalari e dinamici

- I processori moderni sono in grado di
  - inviare più istruzioni contemporaneamente
    - processori con questa caratteristica sono detti superscalari
    - le istruzioni inviate contemporaneamente devono essere “indipendenti”
  - modificare l’ordine di invio delle istruzioni rispetto a quello fissato nel flusso di controllo del programma (scheduling dinamico)
    - per evitare stalli dovuti a dipendenze o cache miss




Controlla la presenza di tutti gli operandi in lettura dell’istruzione, e la invia

L’istruzione può anche essere bloccata a causa di criticità strutturali (unità funzionale occupata)

Completamento in ordine. Scrive i registri rispettando l’ordine dato dalle dipendenze

# Dipendenze sui dati e scheduling dinamico

- **Le criticità dovuti alle dipendenze che portano al blocco dell'invio di un'istruzione riguardano essenzialmente**
  - **le dipendenze RAW (dipendenze data-flow vere)**

- **Le dipendenze WAW (dipendenze di output) e dipendenze WAR (anti-dipendenze) possono essere risolte dal processore senza bloccare l'esecuzione**
  - **l'istruzione viene comunque eseguita, e le scritture avvengono scrivendo in registri temporanei interni**
  - **l'unità di commit si farà poi carico di ordinare tutte le scritture (dai registri temporanei a quelli del register file)**