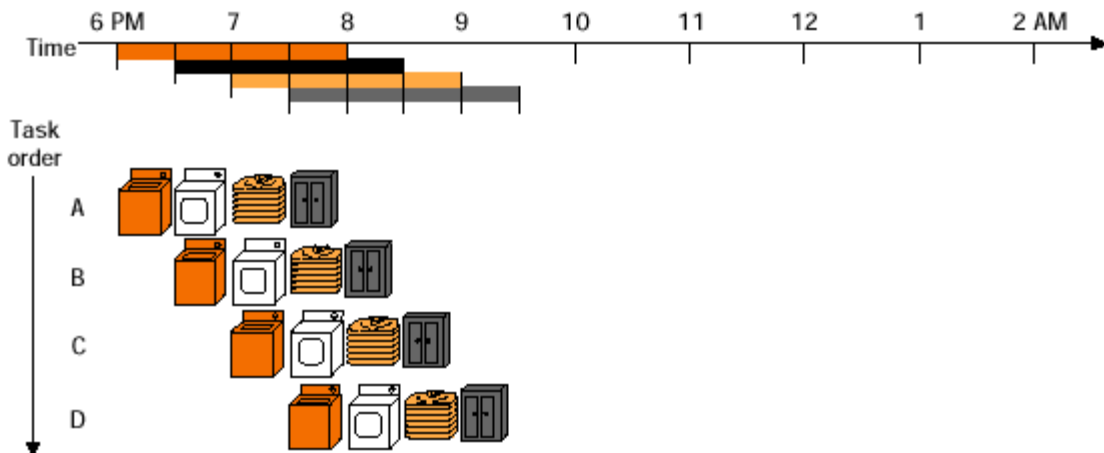
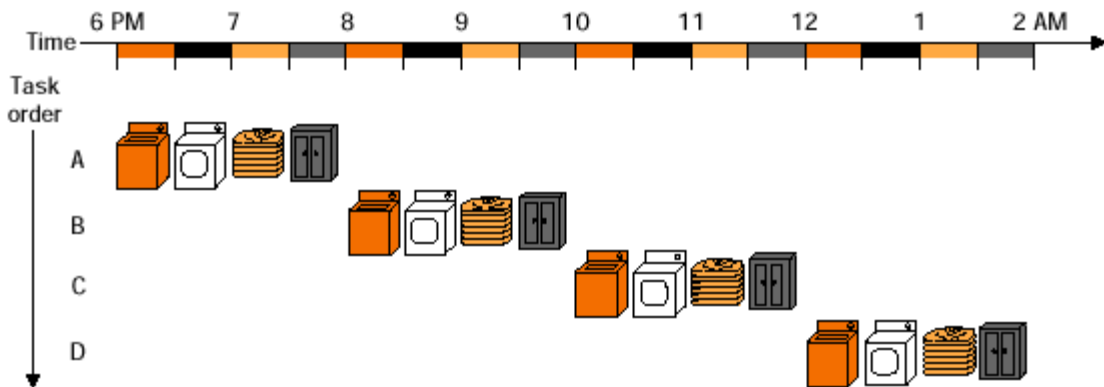

Instruction Level Parallelism & Parallel Multicore

Salvatore Orlando

Organizzazione parallela del processore

- I processori moderni hanno un'organizzazione interna che permette di eseguire più istruzioni in parallelo (ILP = instruction level parallelism)
- 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) ⇒ **von Neumann bottleneck**

Pipeline

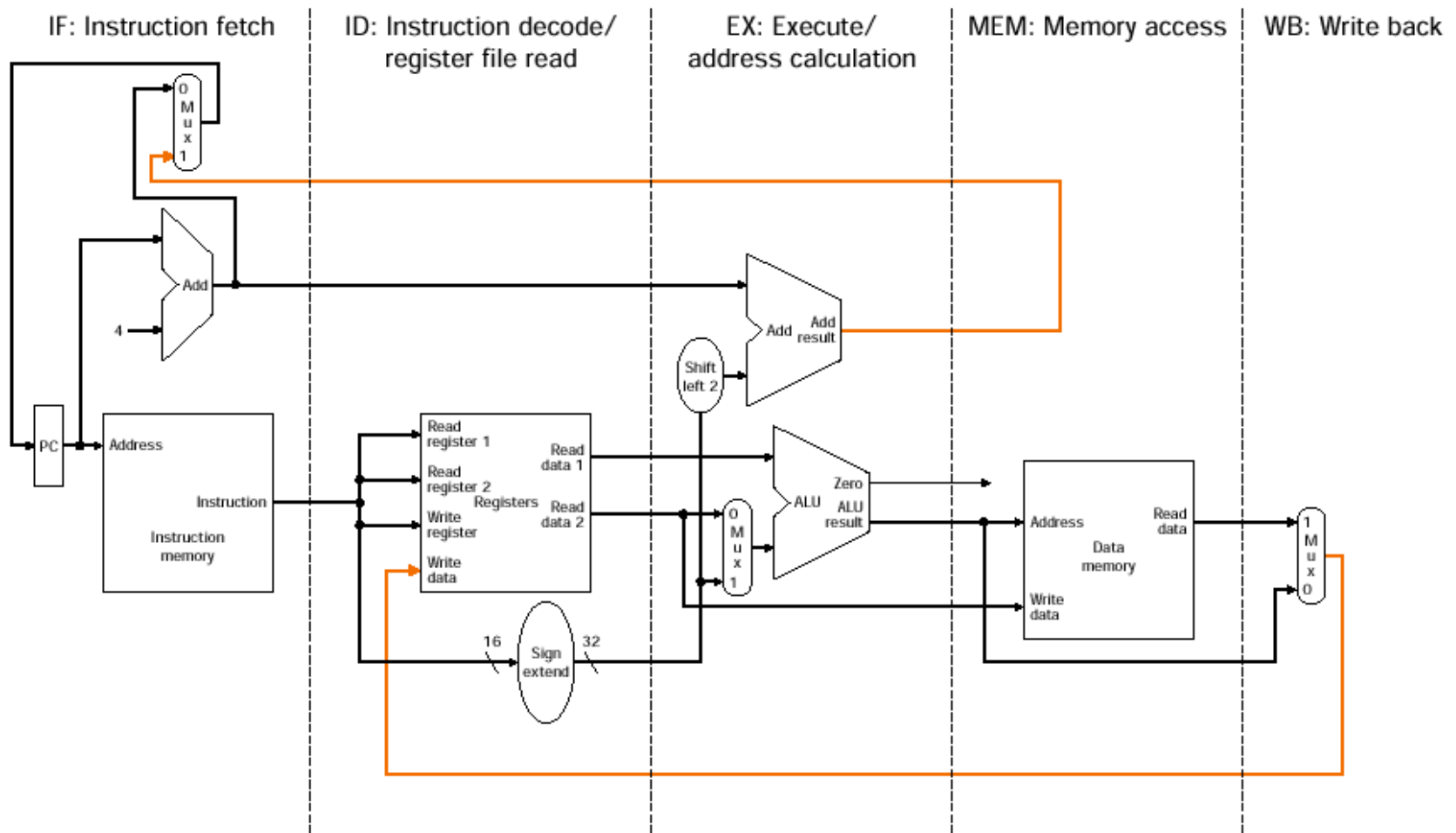


- Le **unità funzionali** (lavatrice, asciugatrice, stiratrice, armadio) sono usate **sequenzialmente** per eseguire i vari “job”
 - tra l’esecuzione di due job, ogni unità rimane inattiva per 1,5 ore
- In modalità **pipeline**, il job viene suddiviso in stadi, in modo da usare le **unità funzionali** 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)

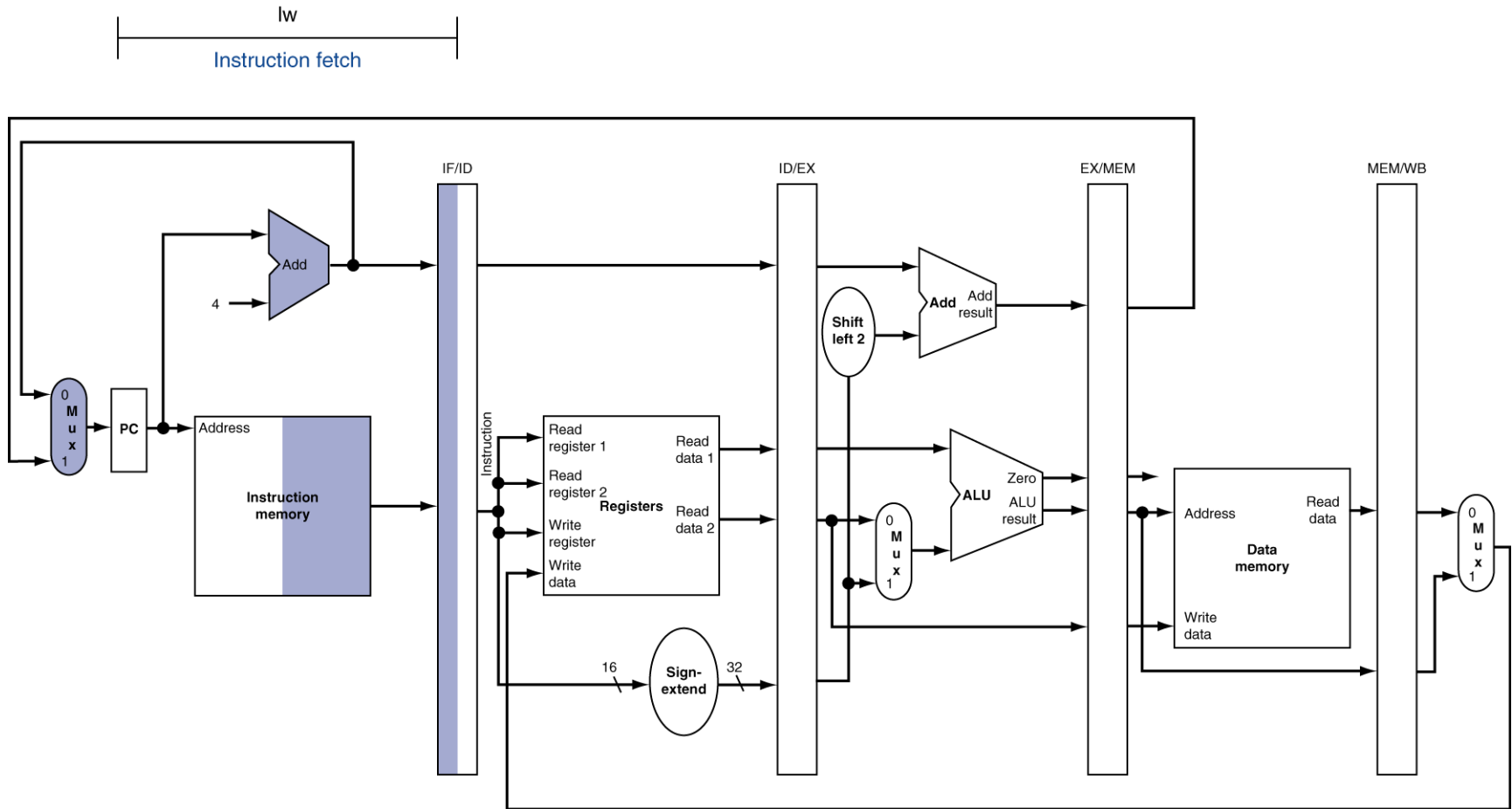


- Unità funzionali replicate (memoria, addizionatori) nei vari stadi
- Ogni stadio completa l'esecuzione in un ciclo di clock (2 ns)
- Necessari **registri addizionali**, per memorizzare i risultati intermedi degli stadi della pipeline

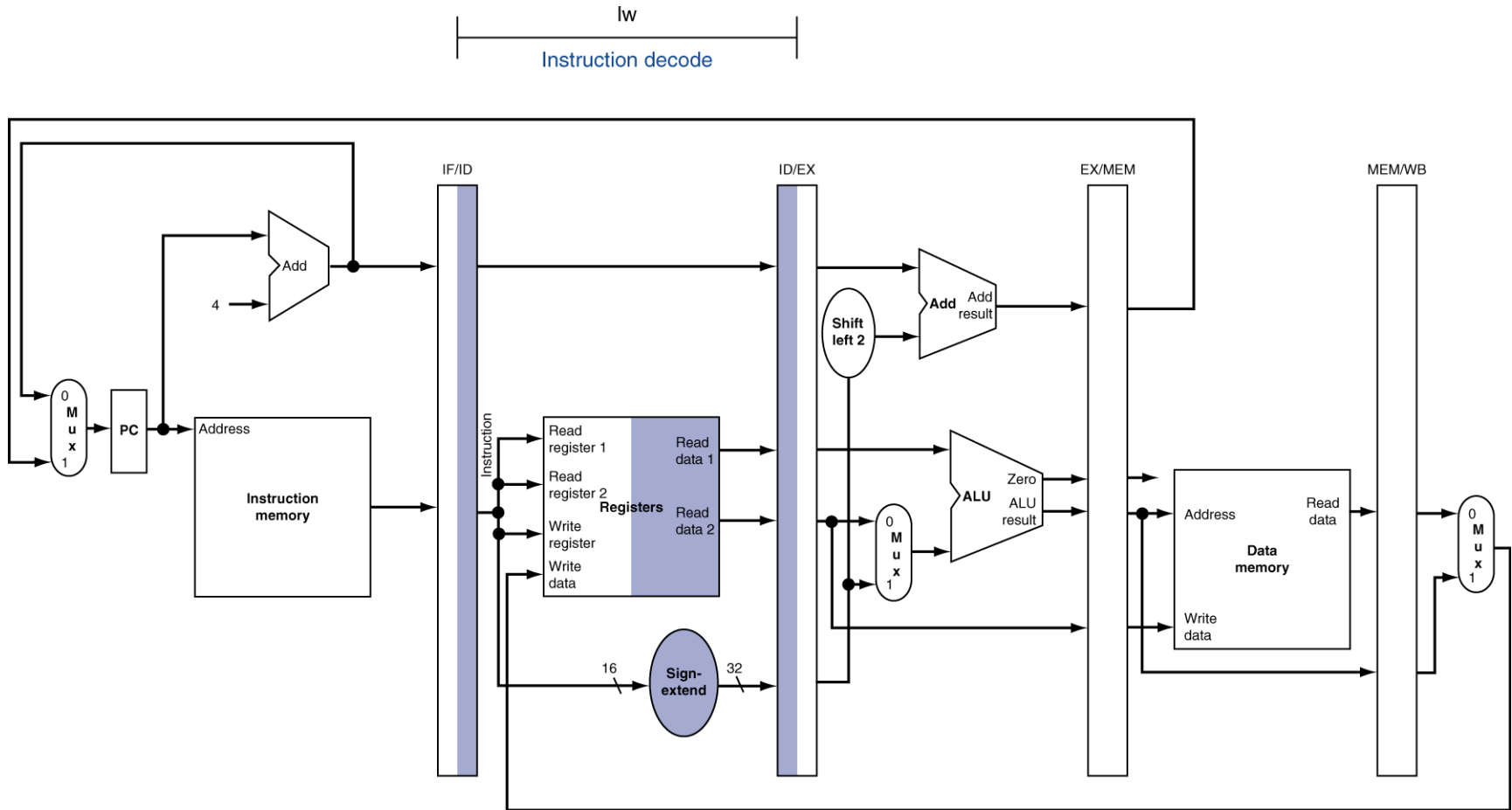
Pipeline Operation per l'istruzione l_w

- Nelle prossime slide illustriamo il **flusso “cycle-by-cycle”** delle istruzioni attraverso il **datapath pipeline**
 - **ESEMPIO:** istruzione l_w
- Diagramma “single-clock-cycle” pipeline
 - Mostra l'uso degli stadi della pipeline nei singoli cicli
 - Sono evidenziate le risorse usate

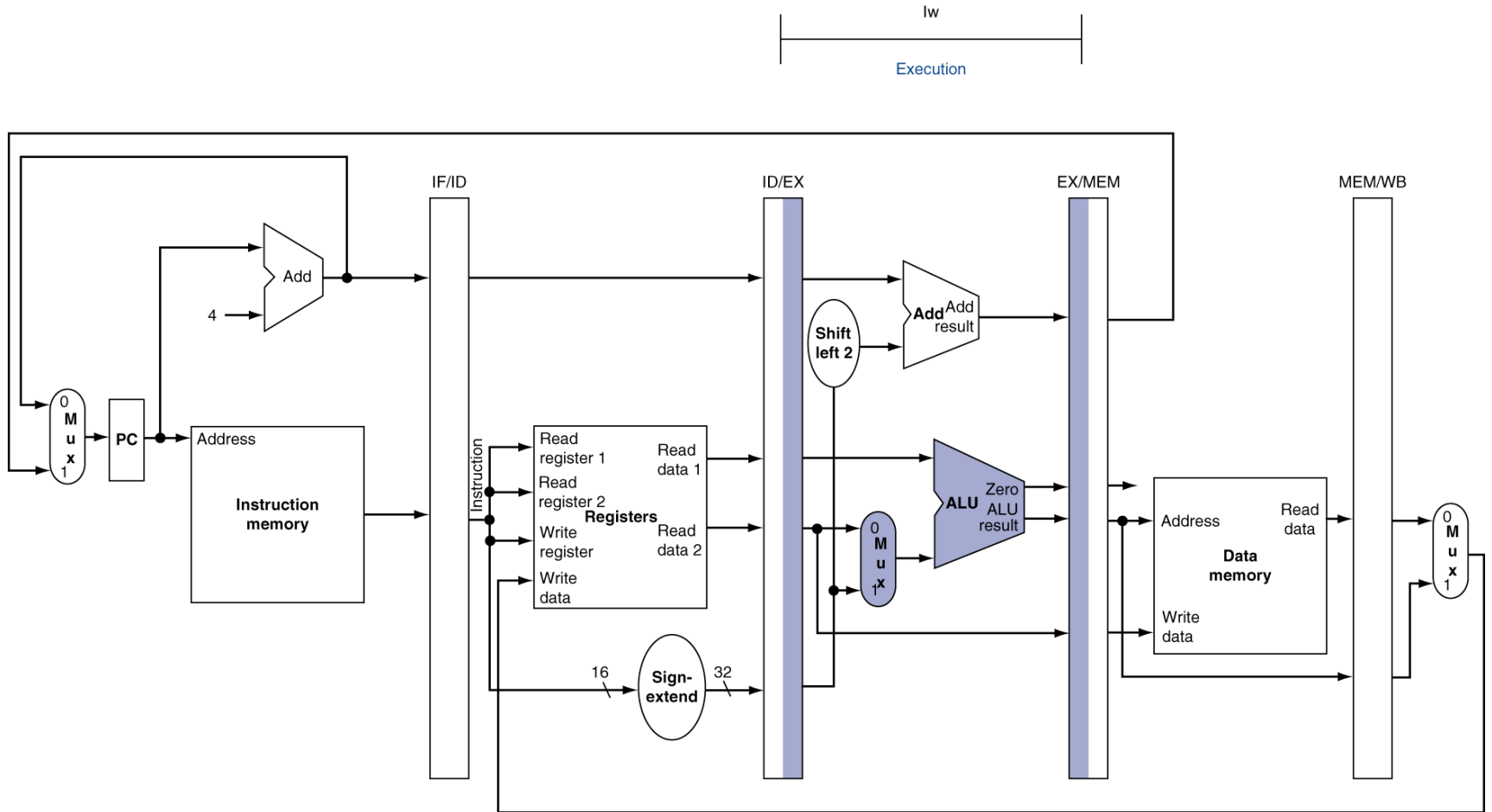
IF per lw



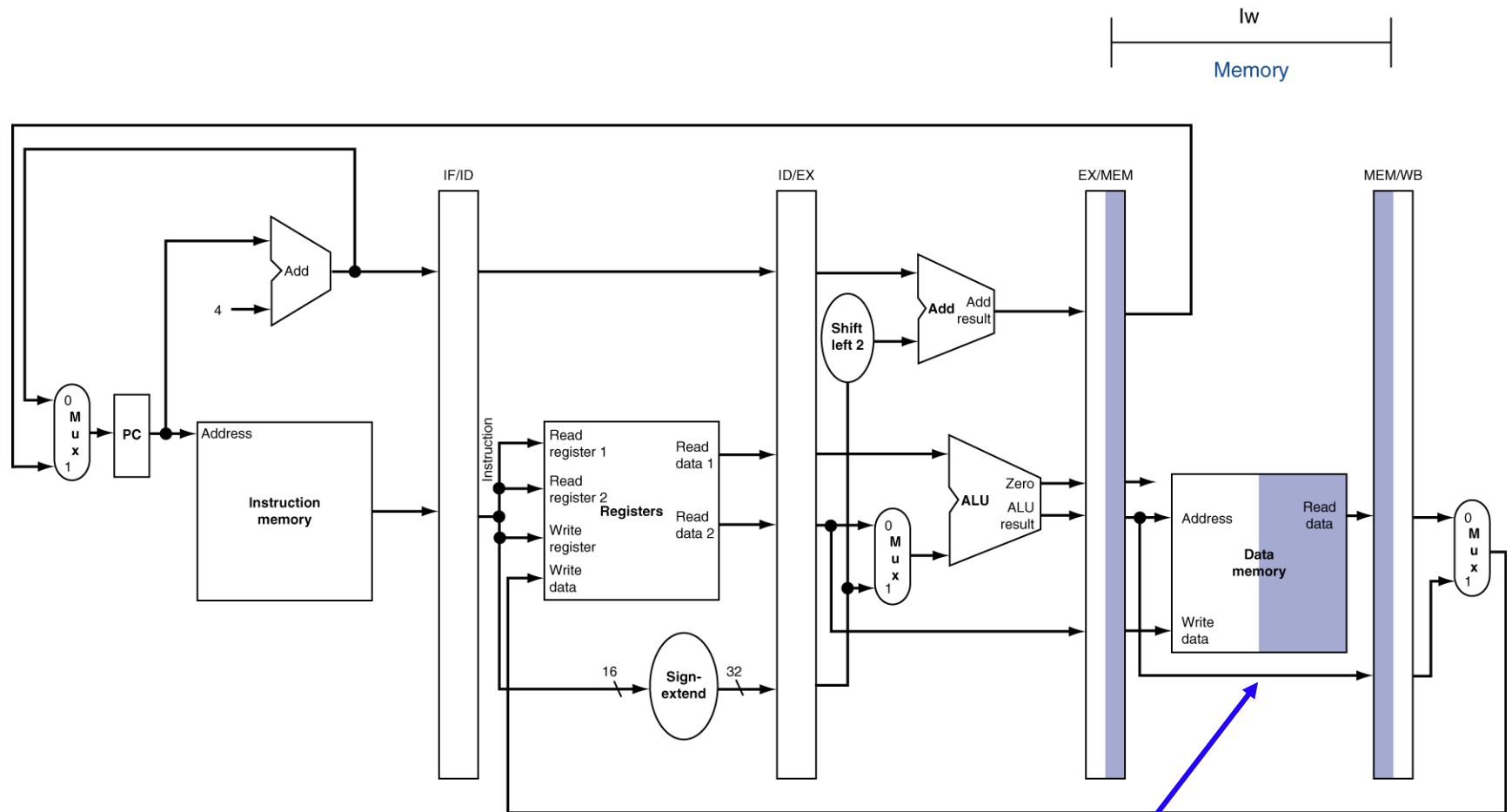
ID per lw



EX per lw

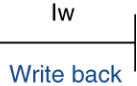


MEM per l_w

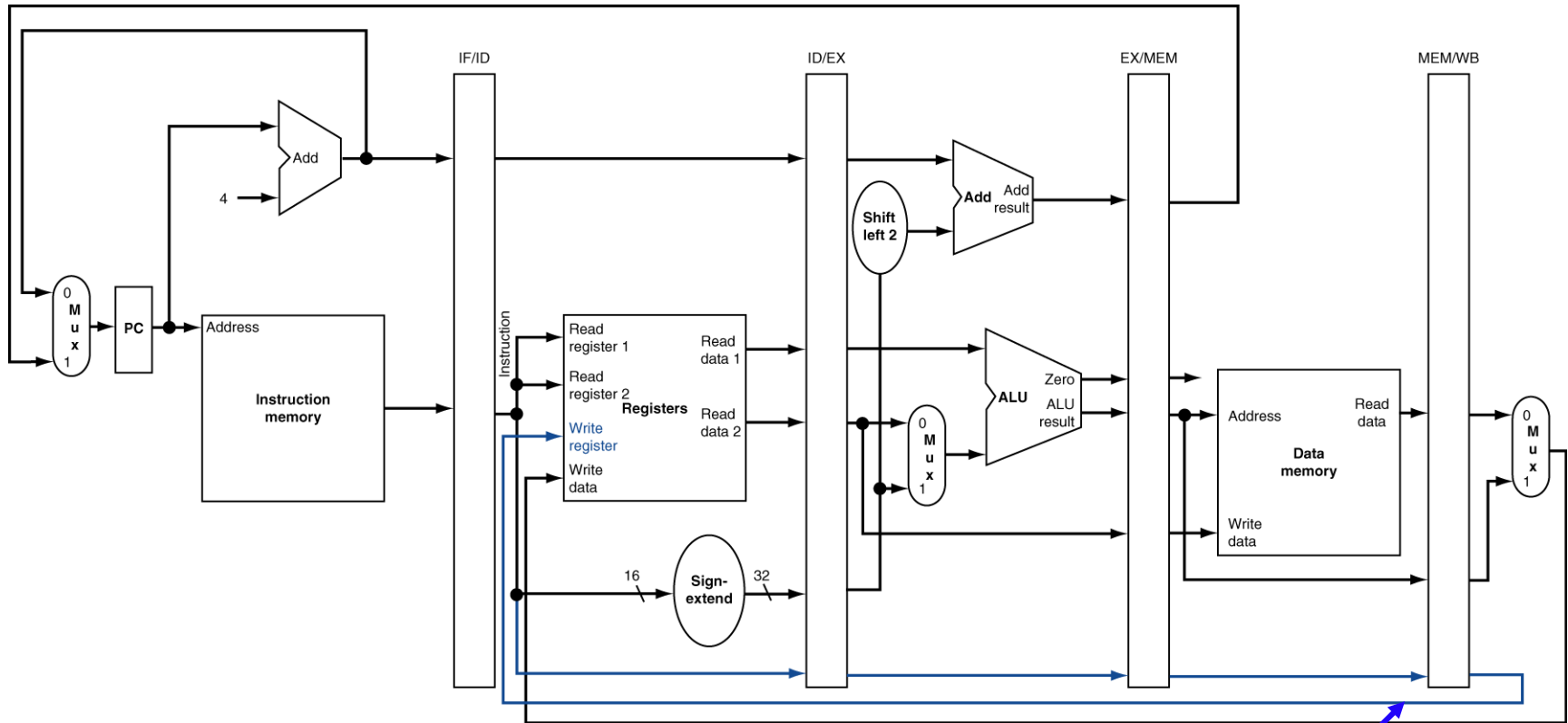


Bypass se l'istruzione non
usa la memoria ($\neq l_w, sw$)

**Numero
del
registro
errato**



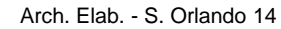
Datapath corretto per lw



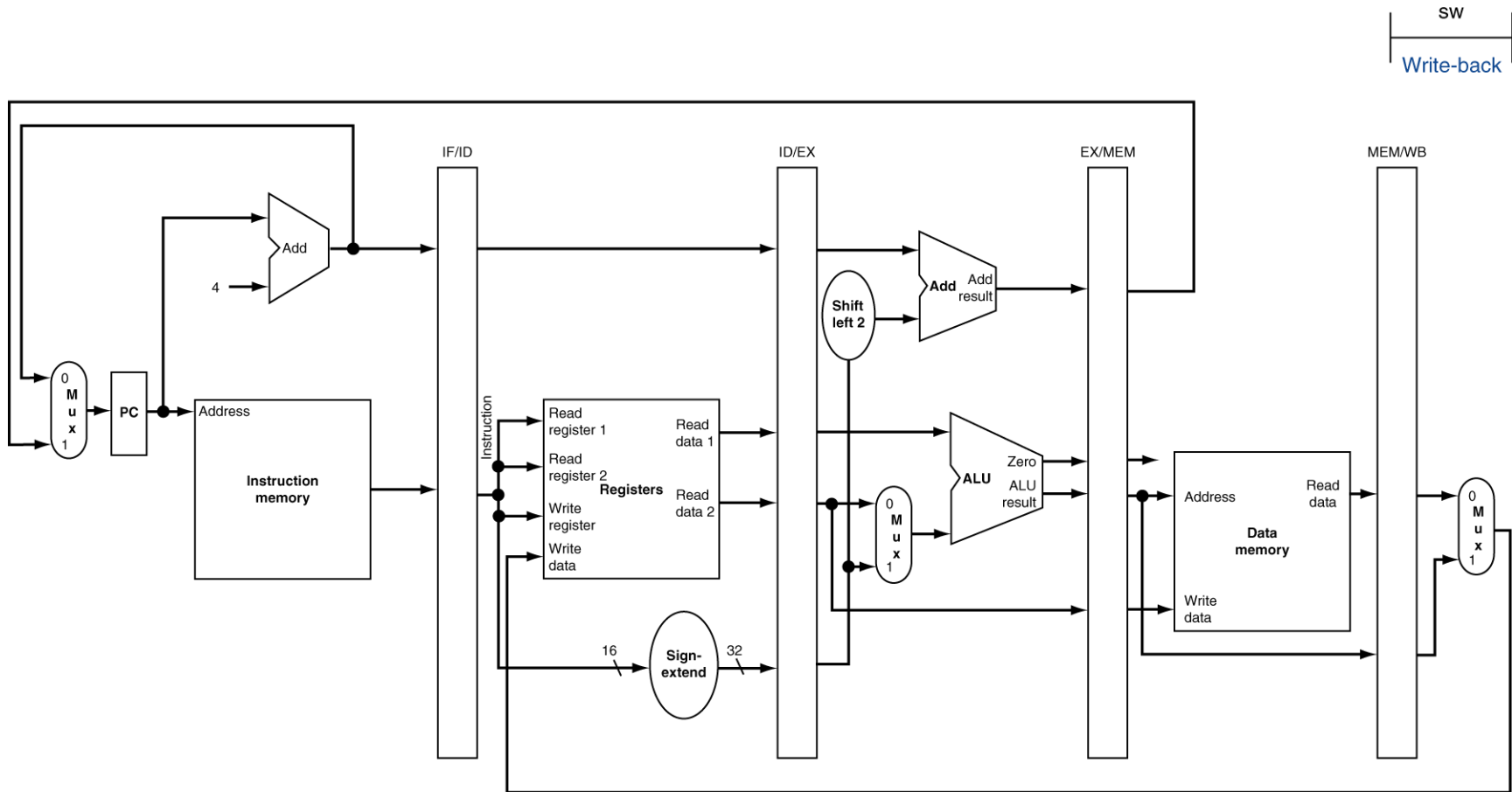
Nello stadio WB i dati calcolati in precedenza tornano indietro, assieme al numero del registro

Pipeline Operation per l'istruzione SW

- Il flusso “cycle-by-cycle” dell'istruzione SW attraverso il datapath pipeline è identica alla SW fino a EX
- Nelle prossime slide il diagramma “single-clock-cycle” pipeline per gli stadi MEM e WB della SW



WB per SW



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

Pipeline e prestazioni

- Data una **pipeline** composta da **n stadi**, 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
$$T_{\text{stadio}} > T_{\text{seq}}/n$$
 - delle dipendenze tra le istruzioni, che ritarda il fluire nella pipeline di qualche istruzione (**pipeline entra in stallo**)

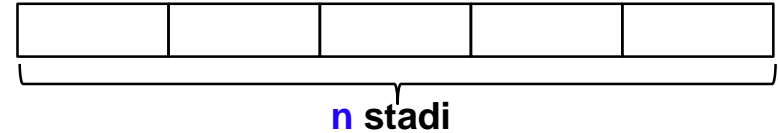
Pipeline e prestazioni

- 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**: $IC * T$

Pipeline e prestazioni

..... istr₈ istr₇ istr₆ istr₅ istr₄ istr₃ istr₂ istr₁

Flusso di IC istruzioni



- 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 = $\frac{IC * T}{(n-1) * T/n + IC * T/n} = \frac{IC}{(n-1)/n + IC/n} = \frac{n * IC}{n - 1 + IC}$

- quando IC è grande rispetto a n (ovvero, quando il flusso/stream di istr. in ingresso alla pipeline è molto lungo), allora lo **speedup tende proprio a n**

$$\lim_{IC \rightarrow \infty} \frac{n \cdot IC}{n - 1 + IC} = n$$

Pipeline e prestazioni

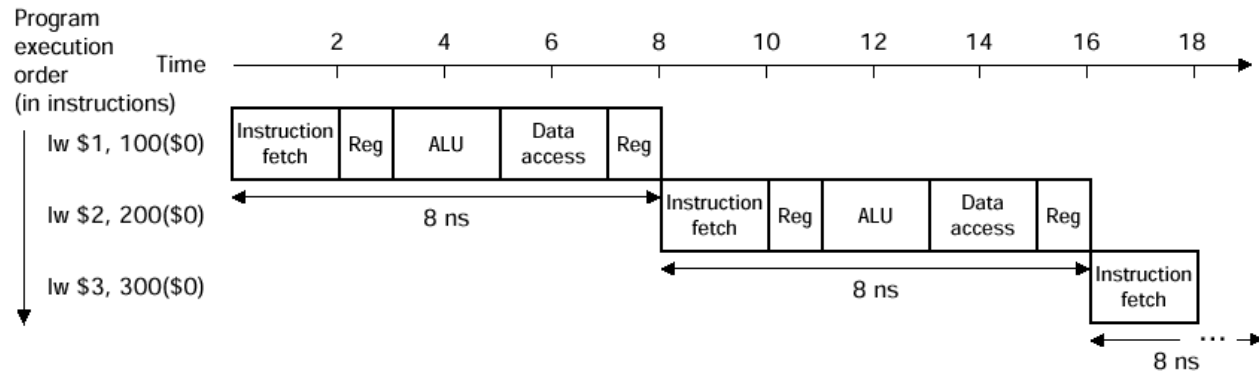
- Confrontiamo ora l'esecuzione sequenziale (a singolo ciclo) di IC istruzioni, con l'esecuzione di una pipeline a n stadi, dove il tempo di esecuzione di ogni stadio è maggiore di T/n : $T' > T/n$
- Tempo di esecuzione del processore a **singolo ciclo**: $IC * T$
- Tempo di esecuzione del processore **pipeline**: $(n-1) * T' + IC * T'$
- $$\text{Speedup} = \frac{IC * T}{(n-1) * T' + IC * T'}$$
- 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'**

$$\lim_{IC \rightarrow \infty} \frac{IC \cdot T}{(n-1) \cdot T' + IC \cdot T'} = \frac{T}{T'}$$

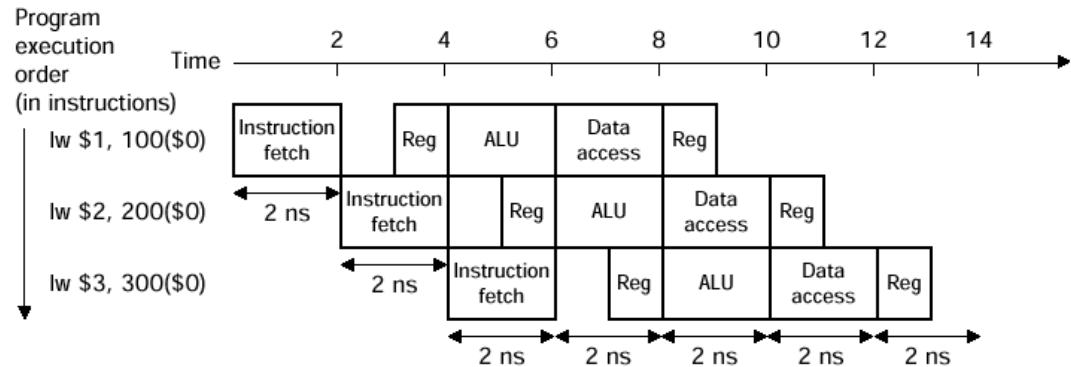
- Nota: se $T' = T/n$ allora $T/T' = n$ (abbiamo ritrovato il risultato precedente, ottenuto per $T' = T/n$)

Esempio con 3 istruzioni

- Pipeline a 5 stadi ($n=5$)
- $T = 8 \text{ ns}$
- $T' = 2 \text{ ns}$, dove
 $T' > T/n = T/5 = 1.6$



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



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

Multi-Cycle Pipeline Diagram

- Formato tradizionale del diagramma

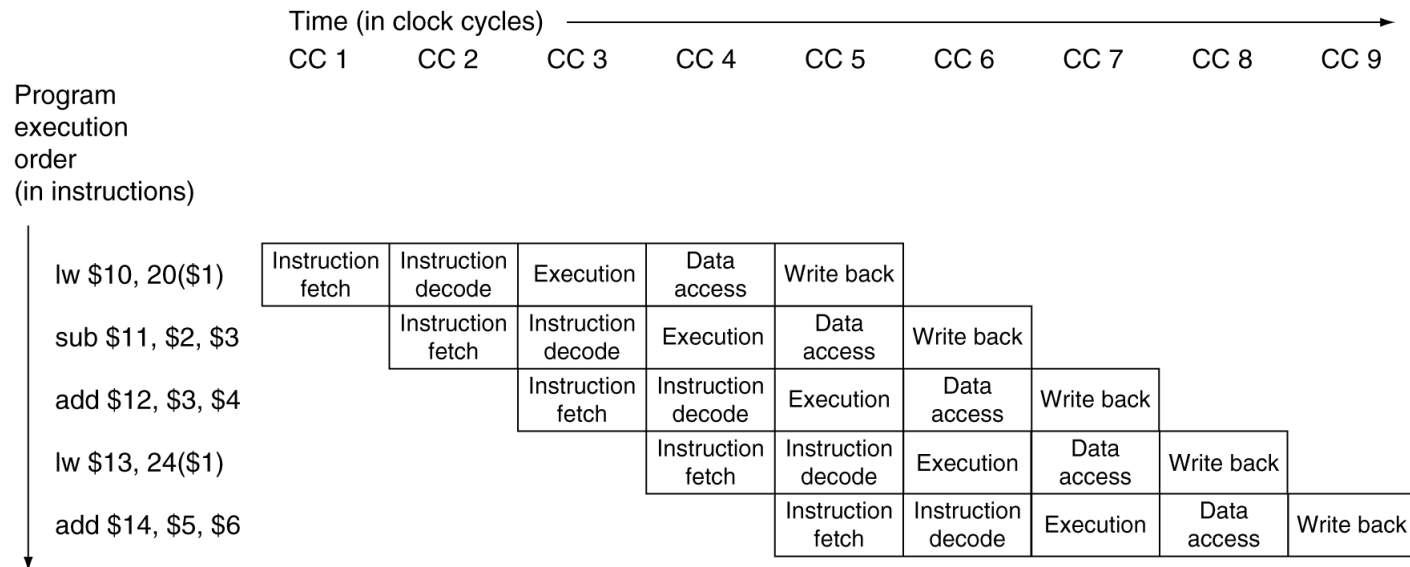


Diagramma temporale multiciclo

- In questo diagramma alternativo sono illustrate anche le risorse utilizzate

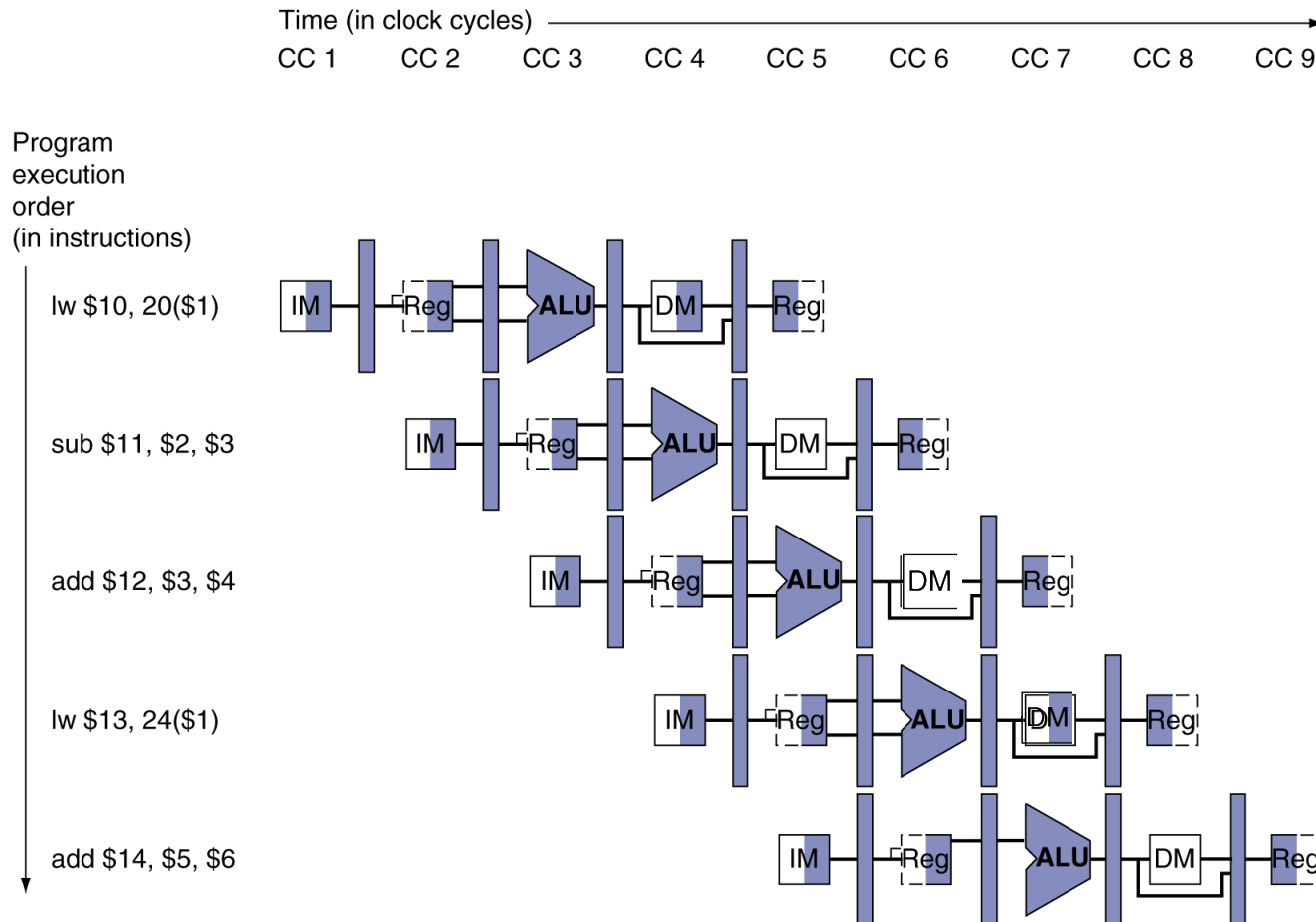
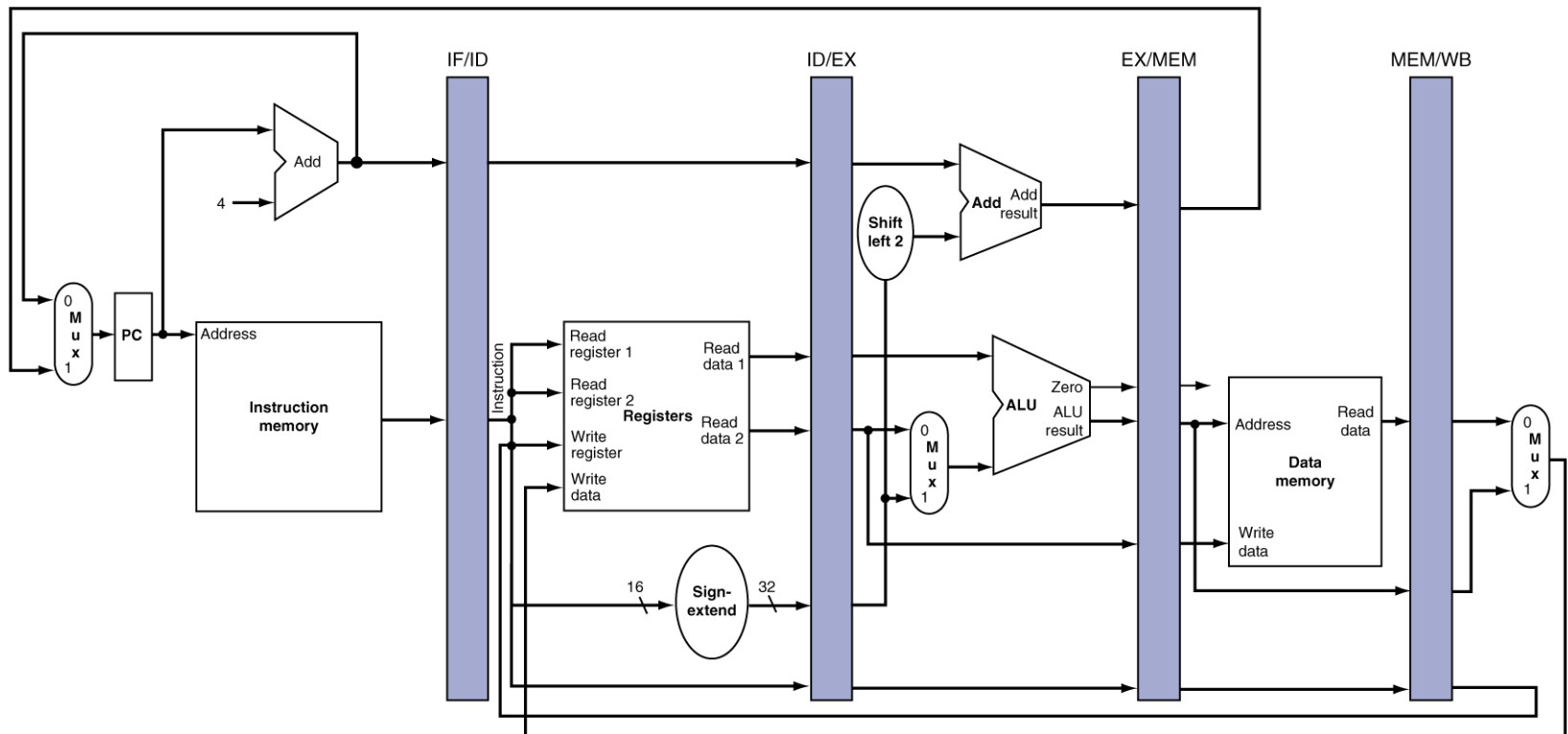


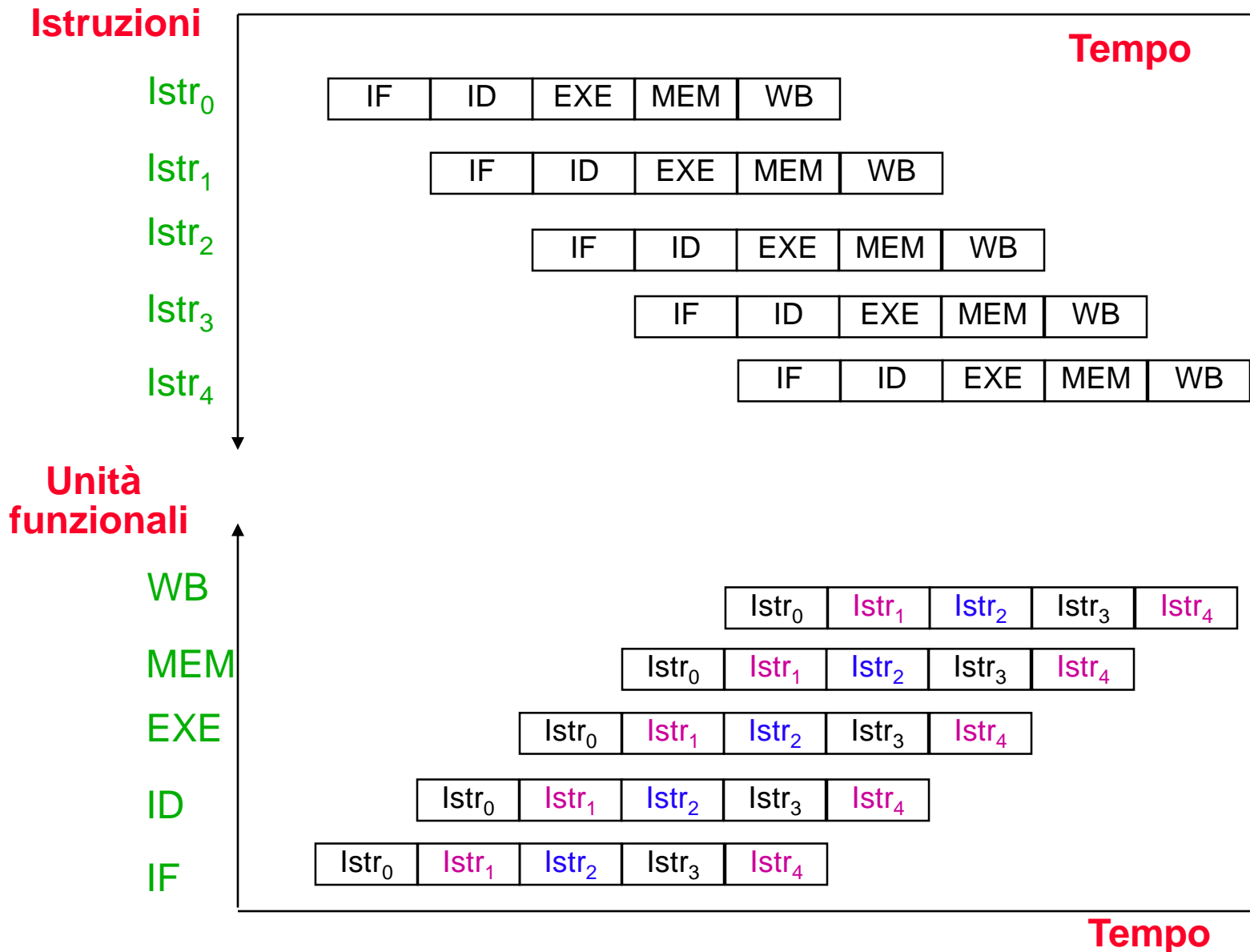
Diagramma temporale a Singolo-Ciclo

- Stato della pipeline ad un dato ciclo (CC=5 rispetto al diagramma multi-ciclo precedente)

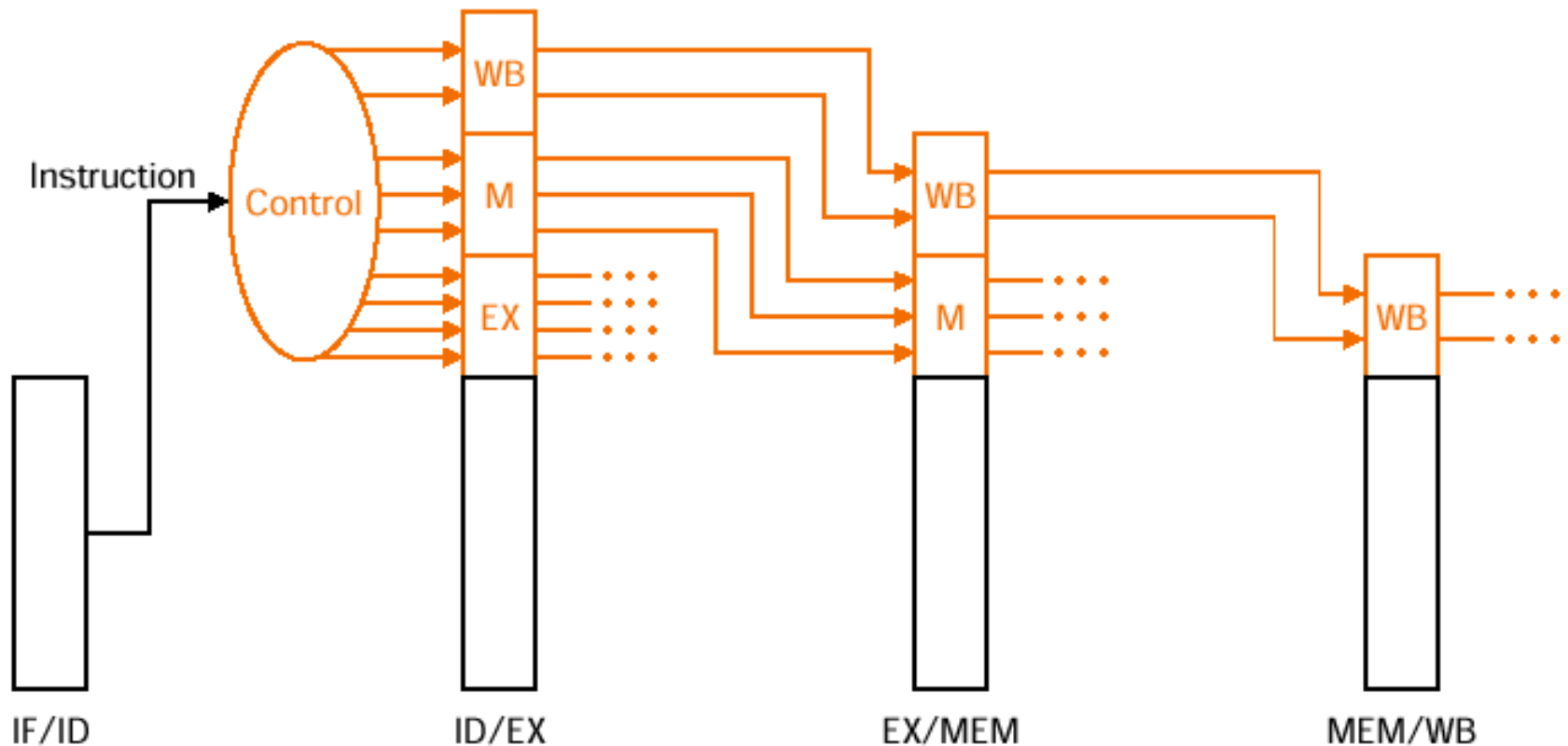
add \$14, \$5, \$6	lw \$13, 24 (\$1)	add \$12, \$3, \$4	sub \$11, \$2, \$3	lw \$10, 20(\$1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back



Diagrammi temporali multiciclo alternativi

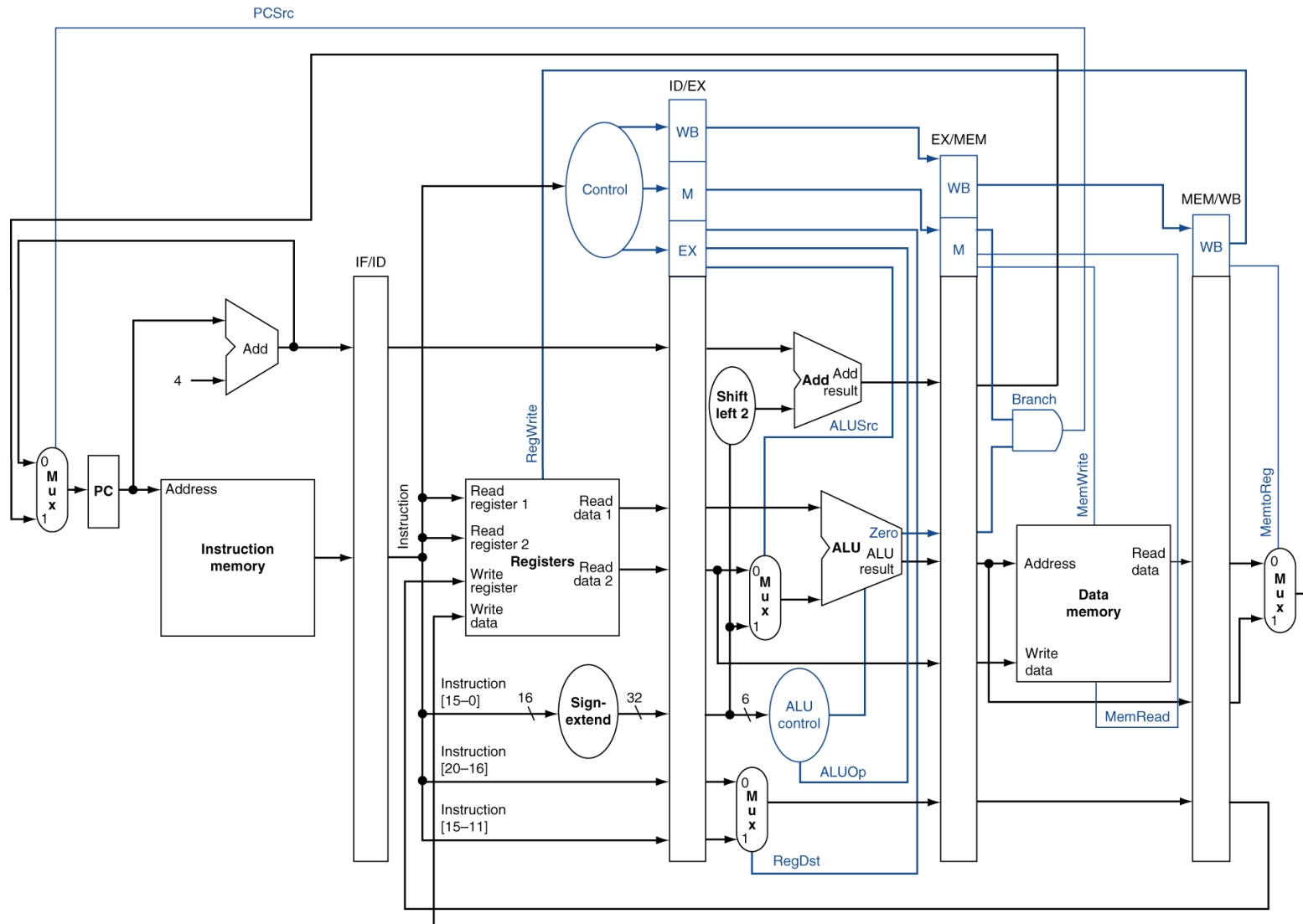


Controllo del processore pipeline



- **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.)

Controllo del processore pipeline



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 in qualche modo dai **risultati** di un'**istruzione precedente**.
 - l'istruzione precedente è già stata inviata (issued)
 - sta transitando nella pipeline
 - non ha completato l'esecuzione

Criticità (hazard)

- L'effetto delle criticità è lo **stallo della pipeline**
- lo stadio della pipeline, che ha scoperto la criticità, assieme agli stadi precedenti
 - rimangono in **stallo** (in pratica, rieseguiscono la stessa istruzione)
 - viene propagata una **nop (no operation)** alle unità seguenti nella pipeline (**bolla** d'aria nella pipeline = metafora nop)
- lo stallò 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**

- dipendenze causate dai dati letti/scritti dalle istruzioni
- es.: dipendenza RAW (Read After Write) : un'istruzione **legge** un registro **scritto** da un'istruzione precedente
 - l'esecuzione dell'istruzione che legge il registro 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/aggiornato 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 con cui i dati sono letti/scritti
- **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
```
- **WAW (Write After Write)** : un'istruzione **scrive** un registro **scritto** da un'istruzione precedente (*l'ordine di esecuzione non si può invertire*)


```
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 (*l'ordine di esecuzione non si può invertire*)

```
add $t0, $s1, $t1    # Read $s1
sub $s1, $s2, $s3    # Write $s1
```
- **RAR (Read After Read)**: ***non è una dipendenza.***
Possiamo anche invertire l'ordine di esecuzione della coppia di istruzioni.

Criticità sui dati

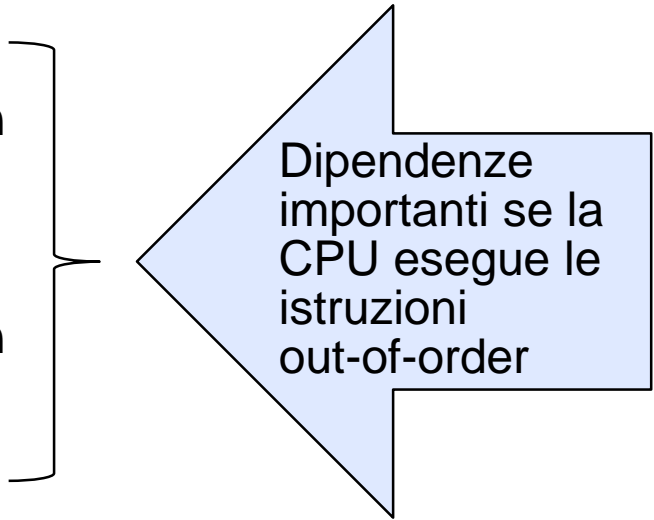
- **RAW (Read After Write)** : un'istruzione **legge** un registro **scritto** da un'istruzione precedente

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



L'unica dipendenza importante
se **non** si modifica
l'ordine di esecuzione

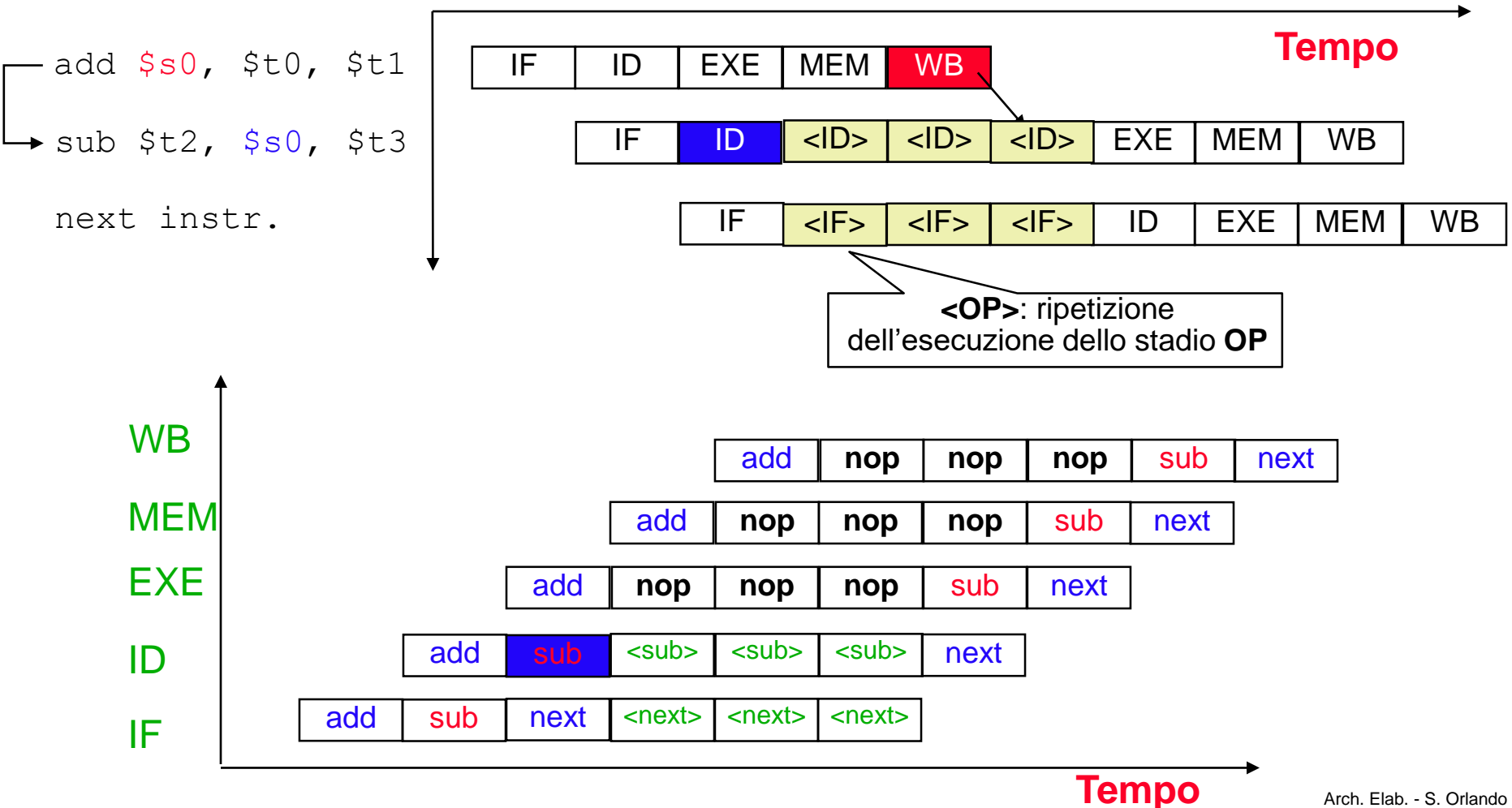
- **WAW (Write After Write)** : un'istruzione **scrive** un registro **scritto** da un'istruzione precedente
- **WAR (Write After Read)** : un'istruzione **scrive** un registro **letto** da un'istruzione precedente



Dipendenze importanti se la
CPU esegue le
istruzioni
out-of-order

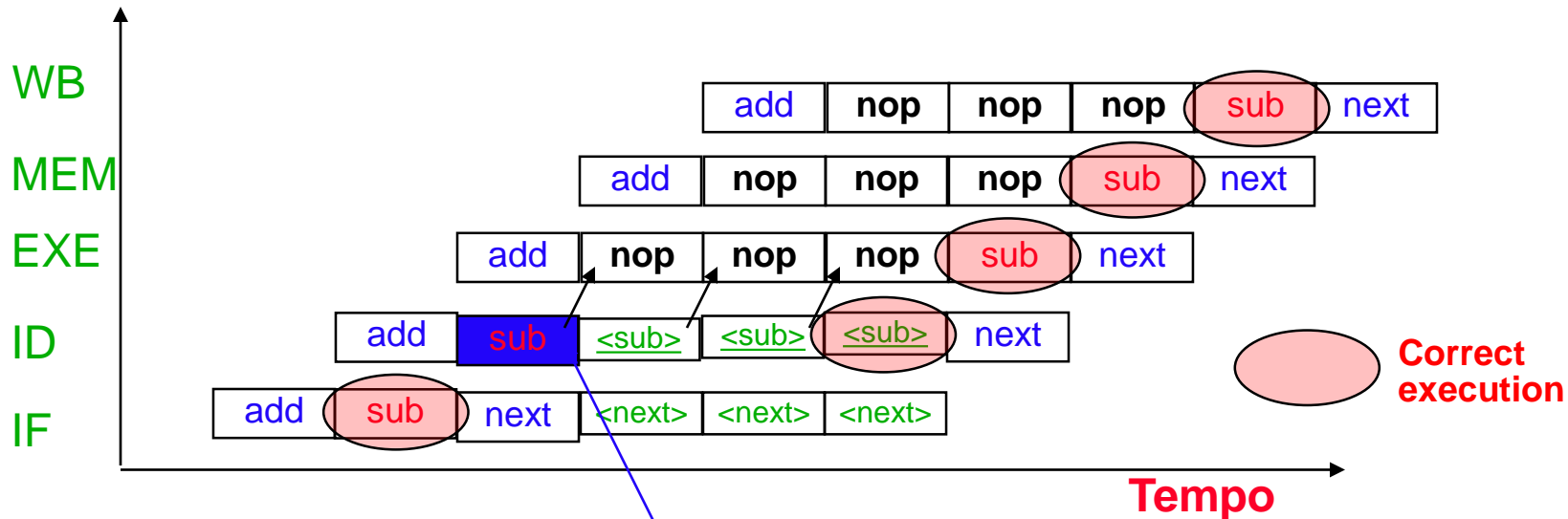
Criticità sui dati: stalli nella pipeline

- 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



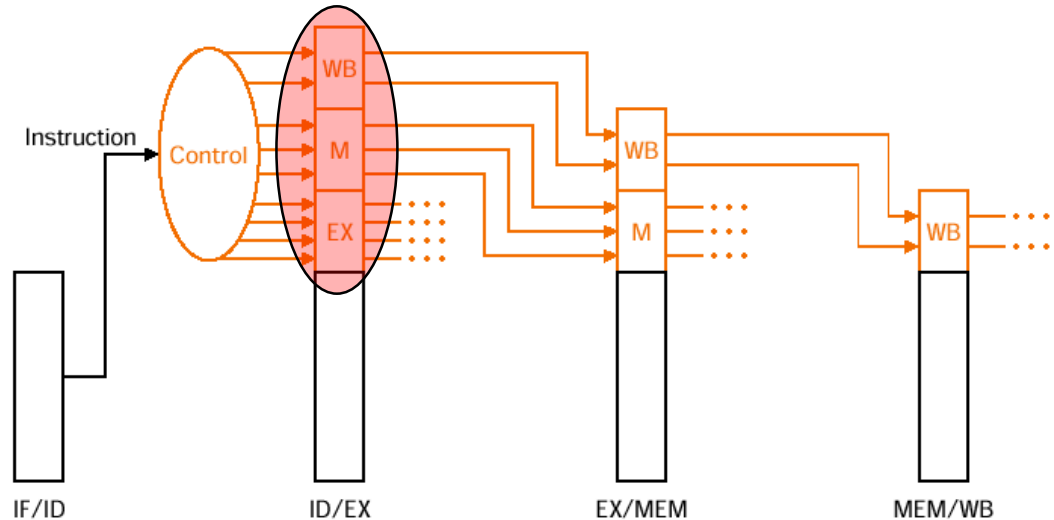
```
add $s0, $t0, $t1  
sub $t2, $s0, $t3  
next instr.
```

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**).

Come mettere in stallo la pipeline per un ciclo

- **Forza i valori di controllo nel registro intermedio ID/EX a 0**

- EX, MEM e WB forzati a eseguire nop (no-operation)



- **Previene l'aggiornamento di PC e del registro intermedio IF/ID**
 - L'istruzione corrente in ID è nuovamente decodificata
 - L'istruzione successiva, già entrata in IF, è nuovamente letta (fetched)

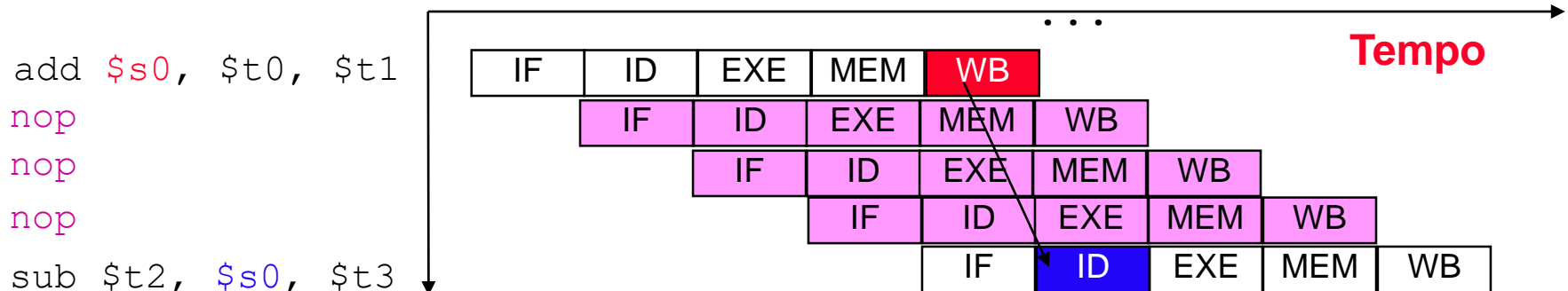
Soluzione software alle criticità sui dati

- Può il compilatore garantire la corretta esecuzione della pipeline anche in presenza di dipendenze sui dati?
 - sì, può esplicitamente inserire delle “**nop**” (speciali istruzioni di “no operation”) 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
...
```

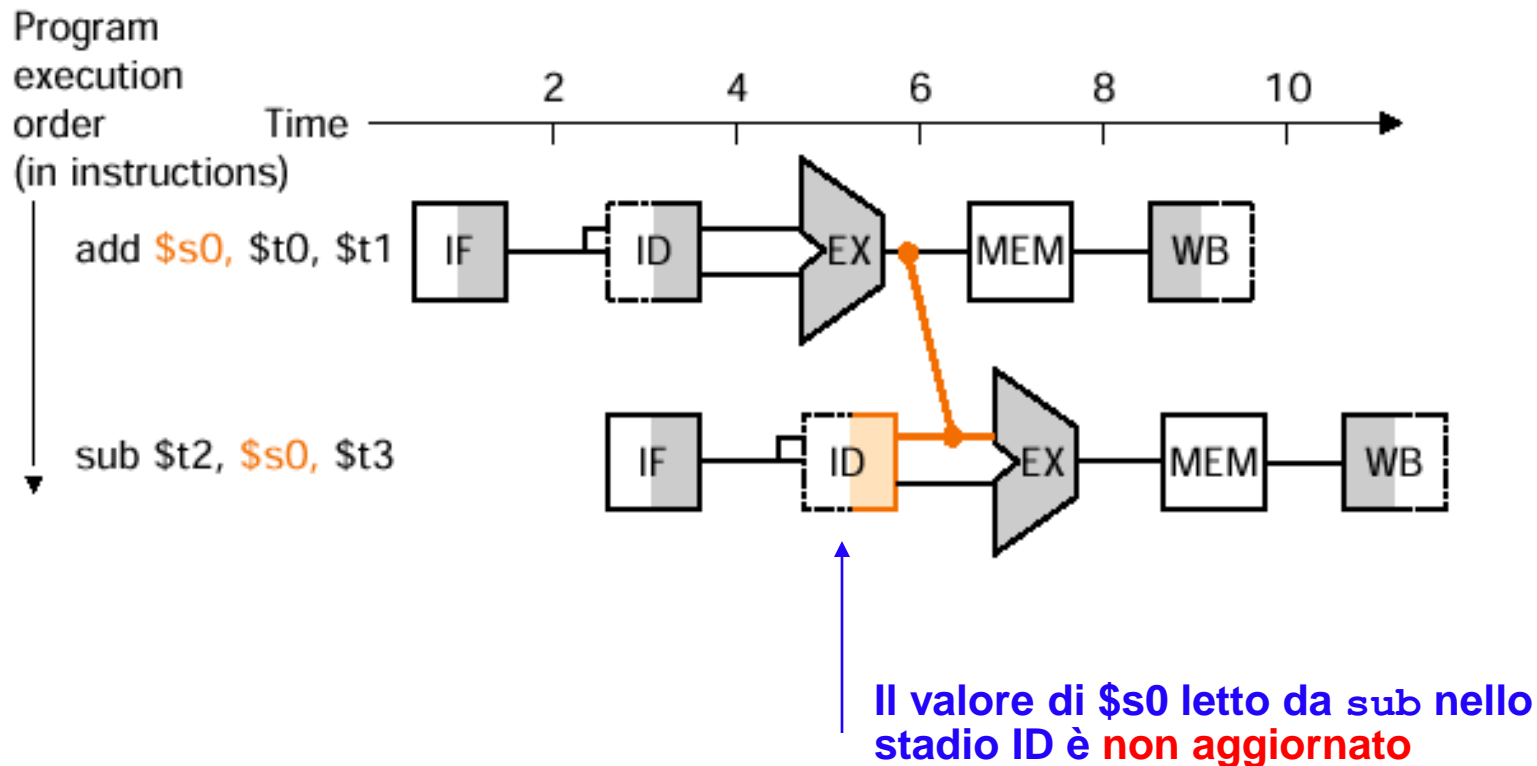


```
add $s0, $t0, $t1
nop
nop
nop
sub $t2, $s0, $t3
```

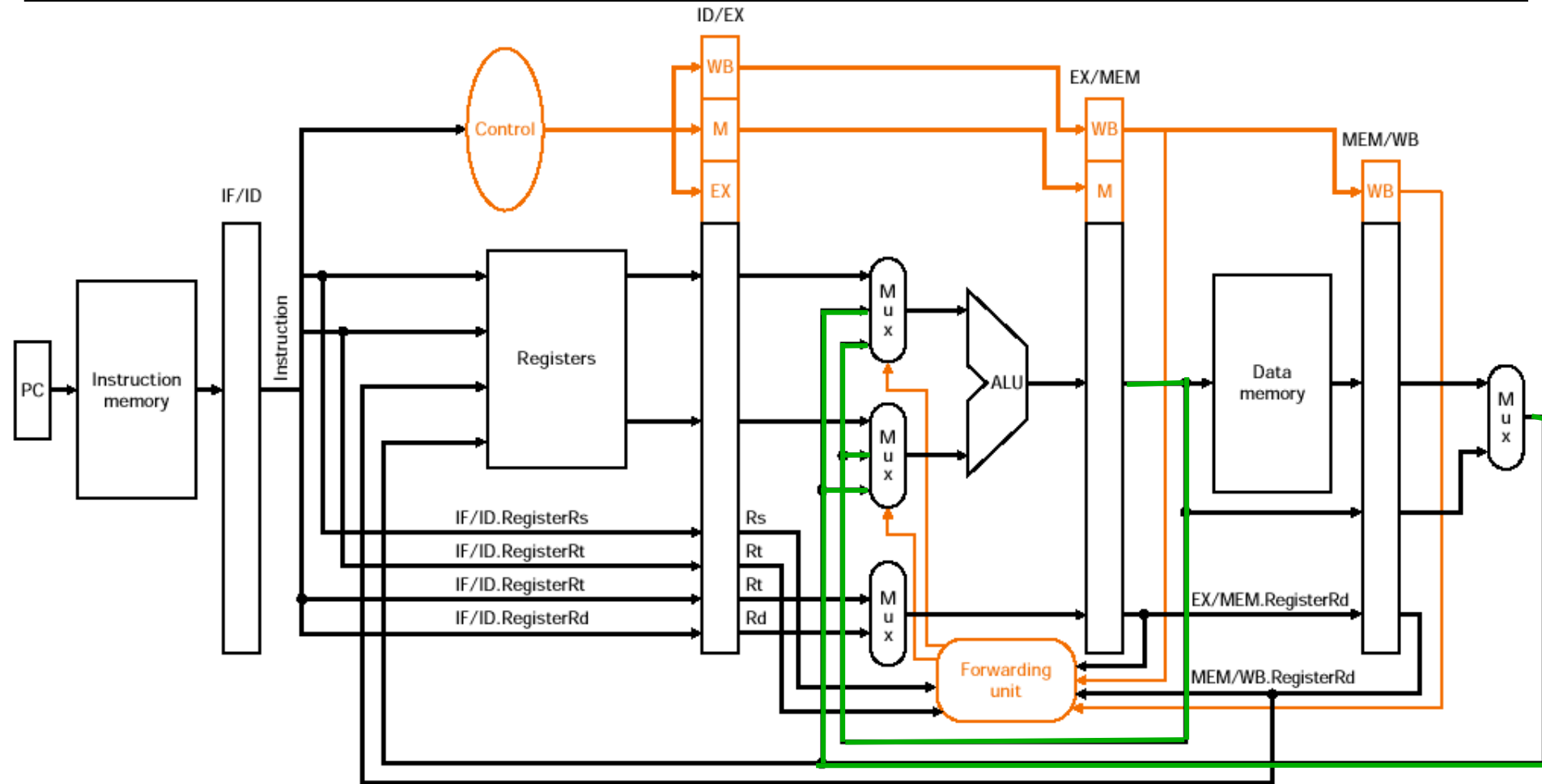


Forwarding

- 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`

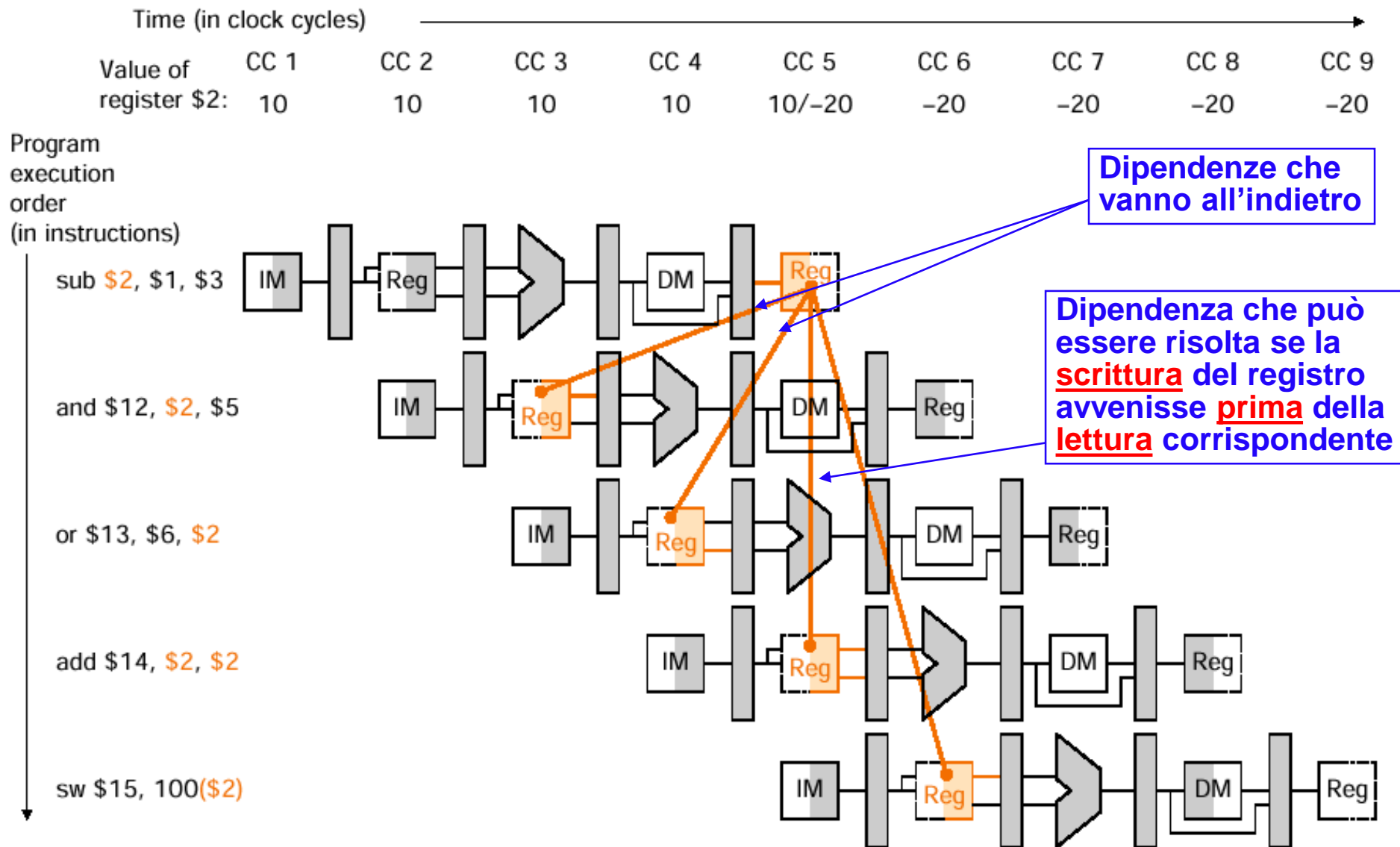


Forwarding e datapath



- I valori calcolati durante gli stadi successivi devono **tornare indietro** (essere *forwarded*) **verso lo stadio EXE** per sostituire i valori letti, ma scorretti e non aggiornati
 - vedi linee evidenziate in verde

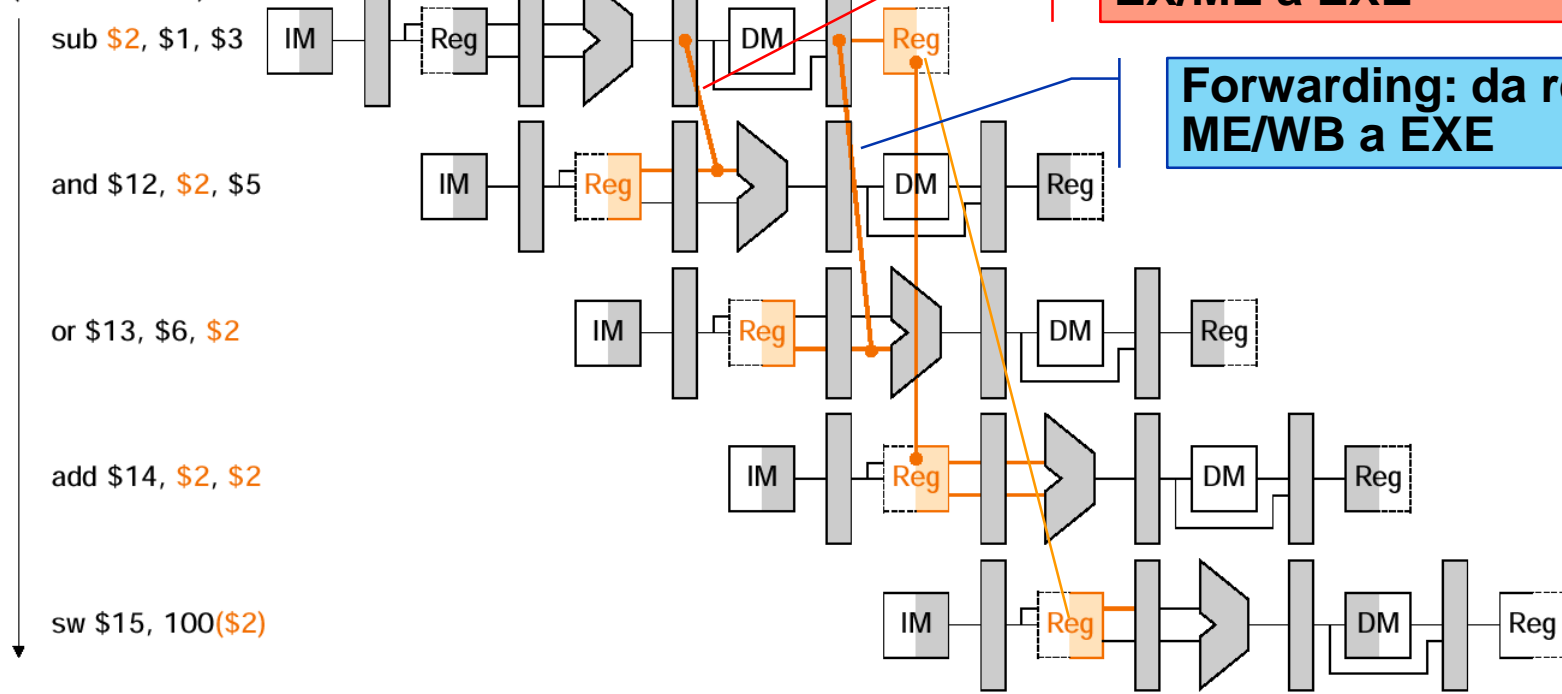
Dipendenze RAW in una sequenza di istruzioni



Risolvere le dipendenze tramite forwarding

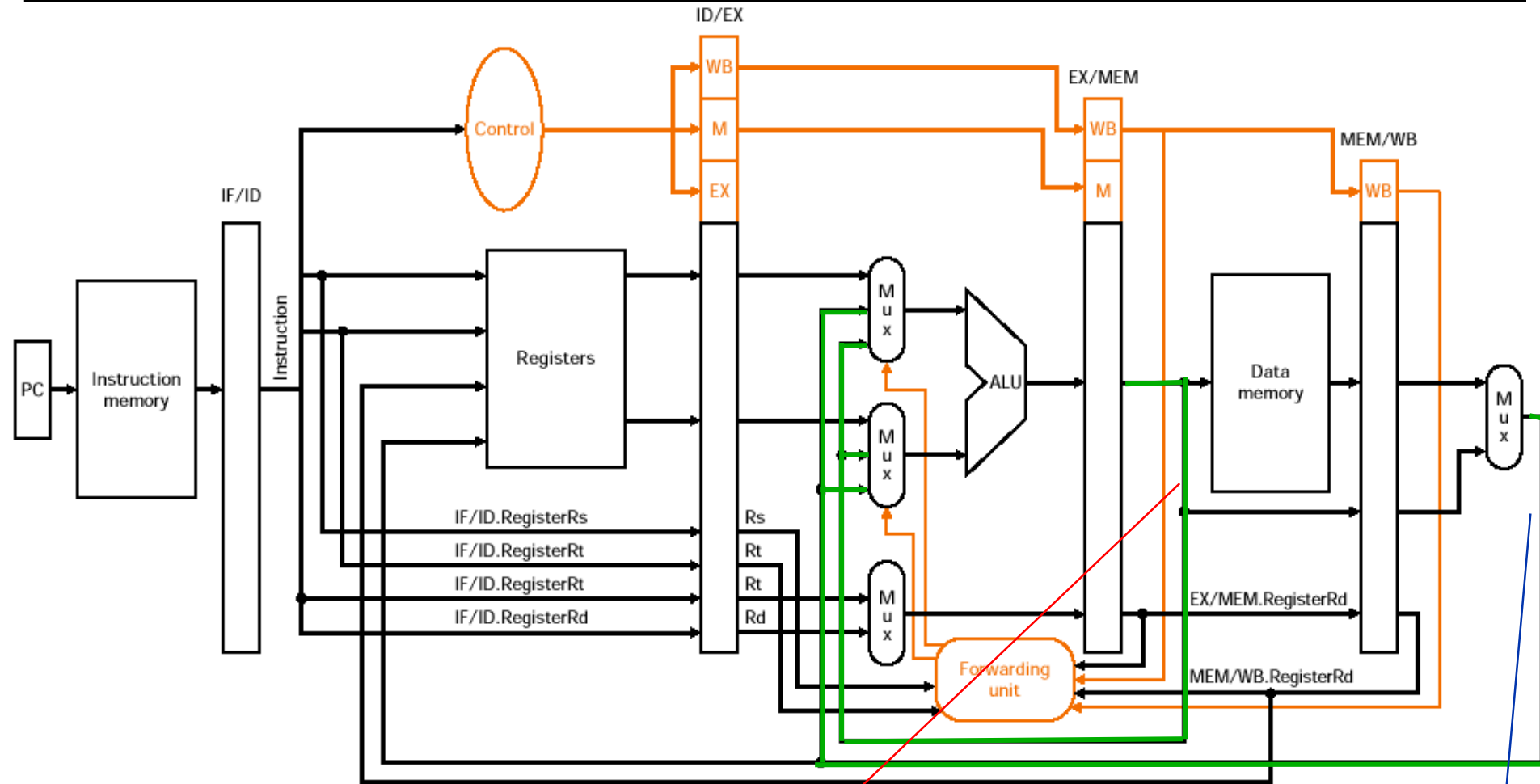
	Time (in clock cycles) →								
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register \$2 :	10	10	10	10	10/-20	-20	-20	-20	-20
Value of EX/MEM :	X	X	X	-20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	-20	X	X	X	X

Program
execution order
(in instructions)



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

Forwarding e datapath

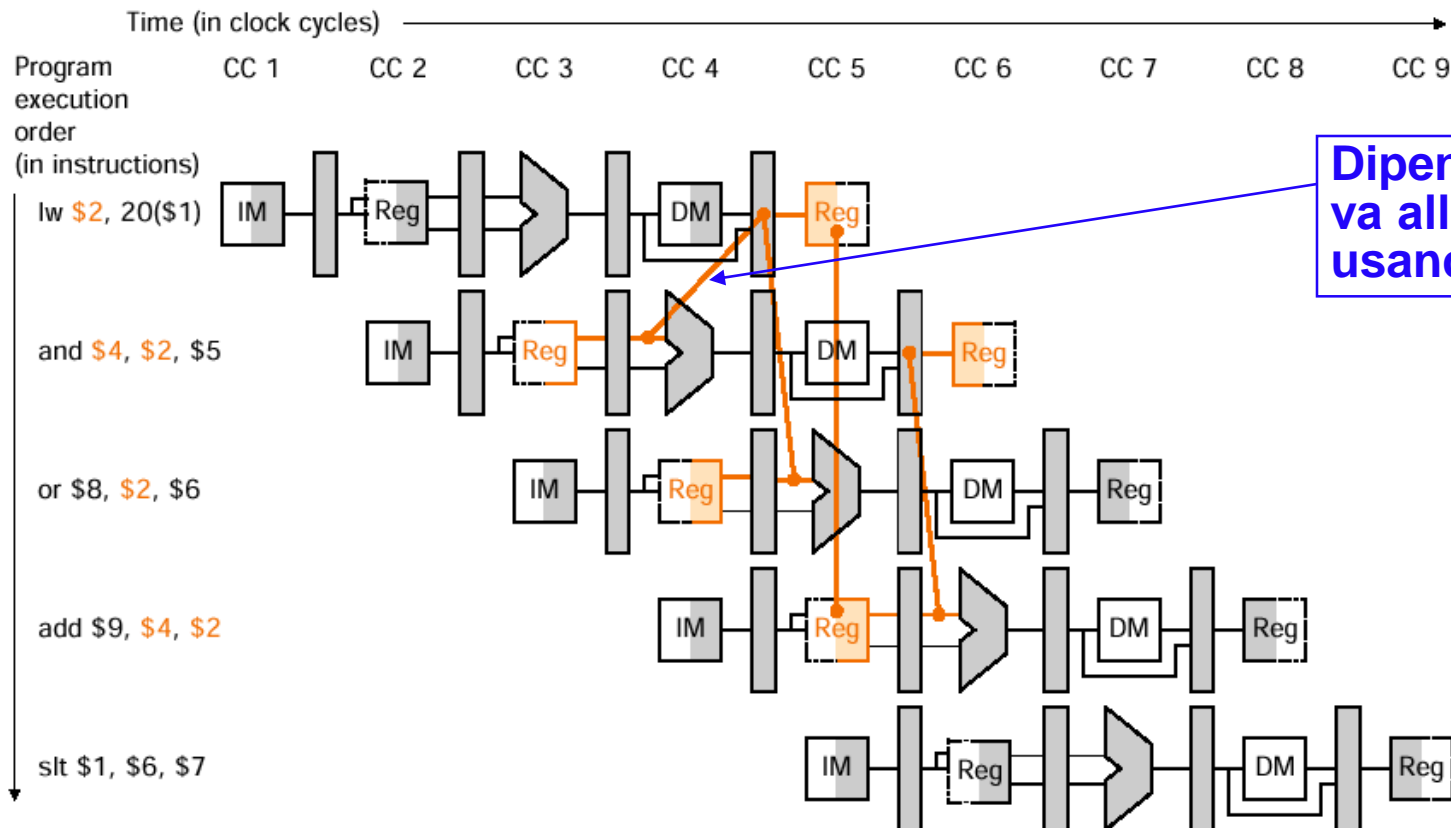


Forwarding: da registro EX/ME a EXE

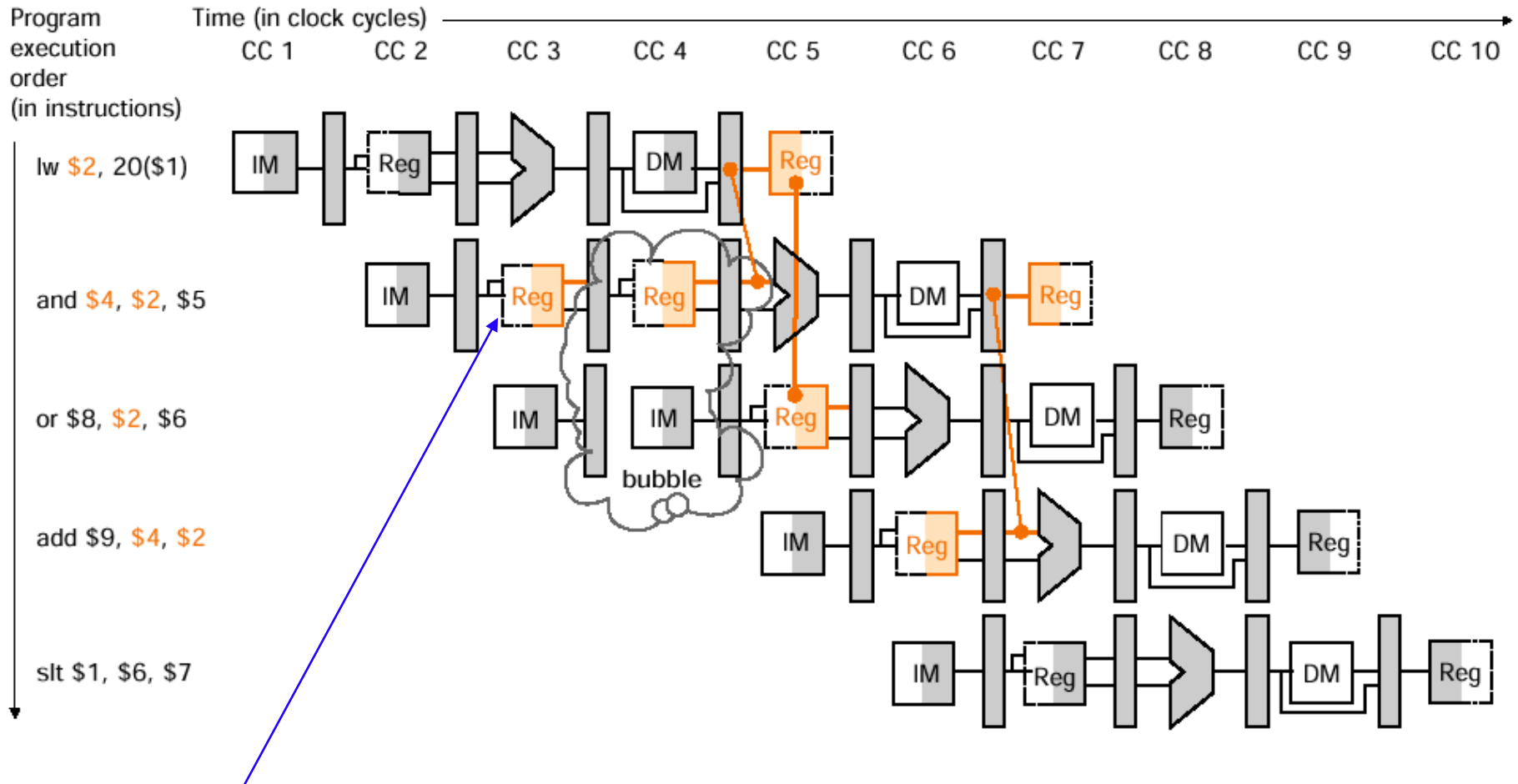
Forwarding: da registro ME/WB a EXE

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**



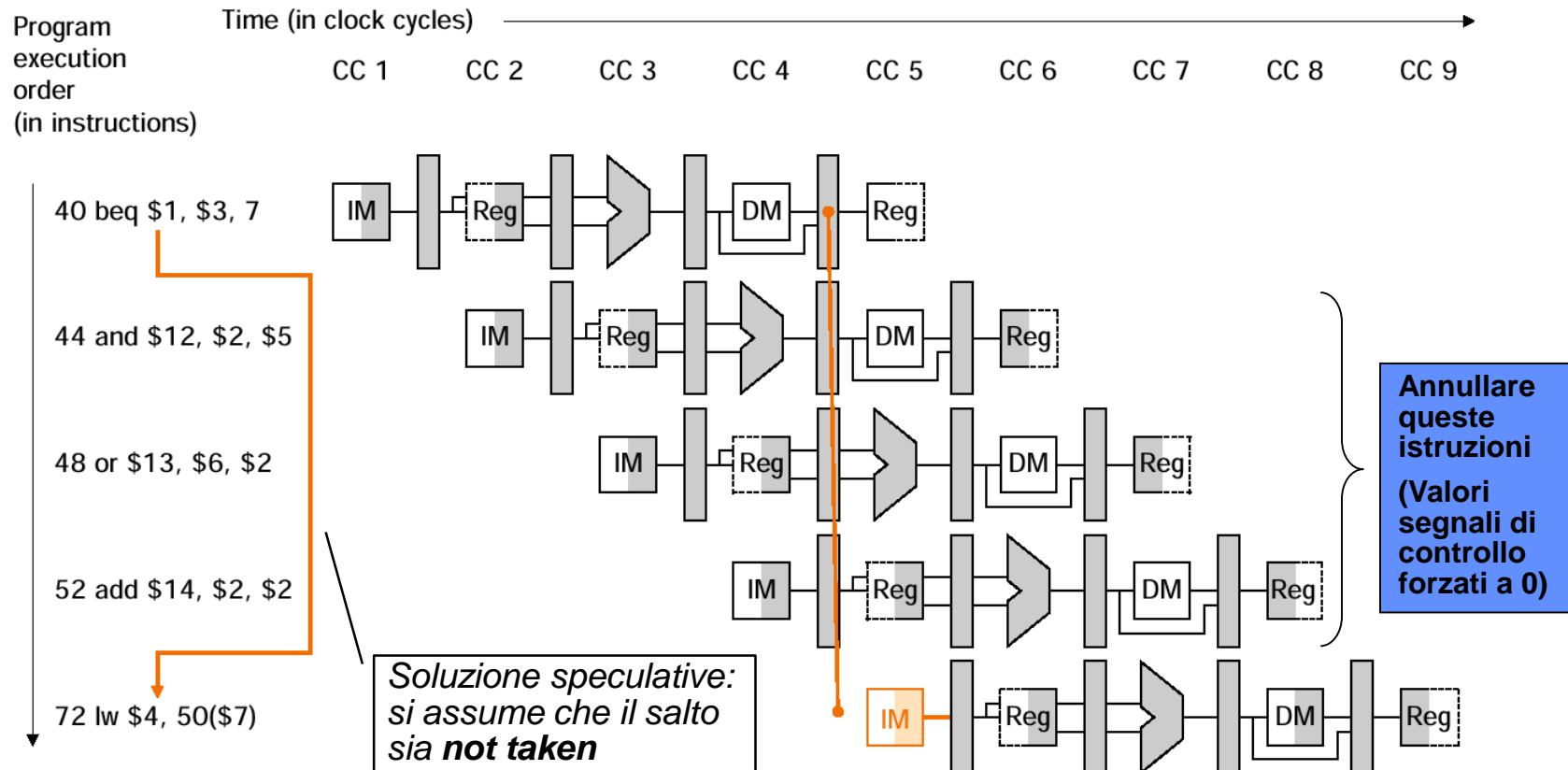
Load e hazard detection unit



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)

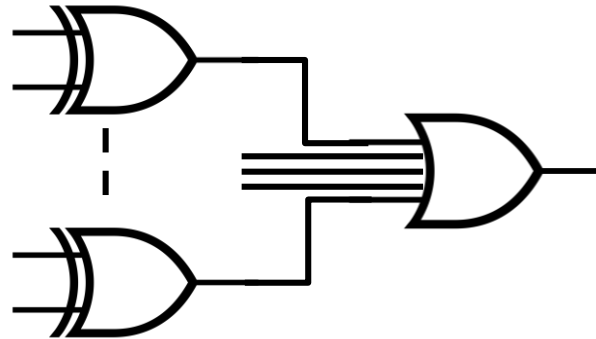
Criticità sul controllo

- 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

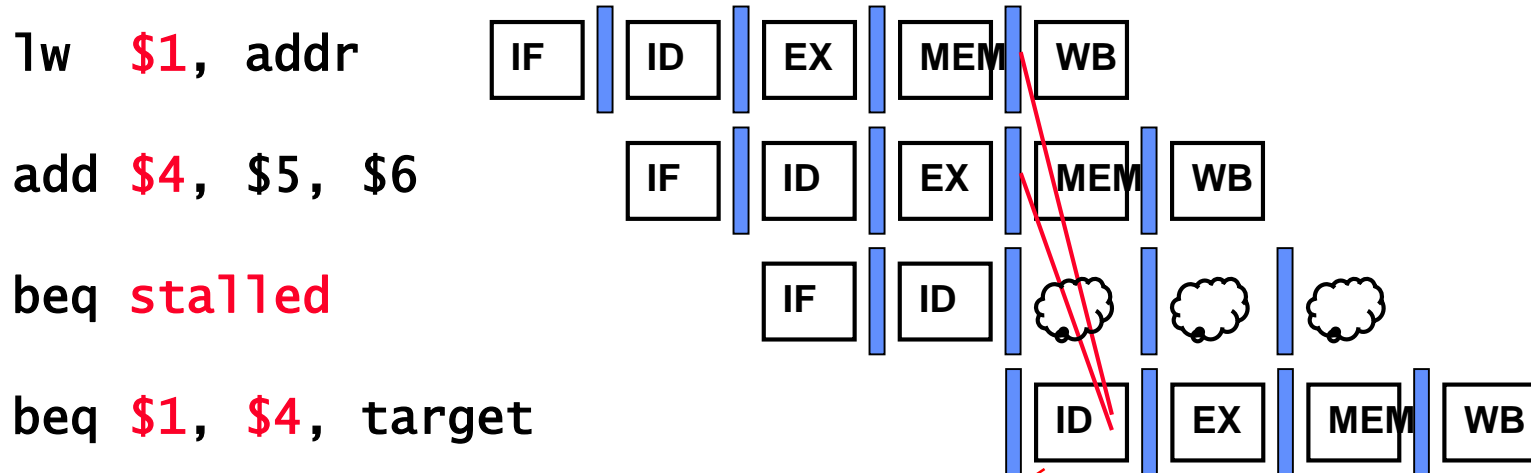
- Anticipiamo il **calcolo di PC** e il **confronto tra i registri** della **beq**
 - spostiamo in ID l'addizionatore che calcola l'indirizzo target del salto
 - invece di usare la ALU per il confronto tra registri, il confronto può essere effettuato in modo veloce da un'unità specializzata
 - tramite 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, a valle della lettura dei registri



- 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

Forwarding e Calcolo del branch anticipato

- Se uno dei registri da comparare nel branch è il registro di destinazione dell'istruzione aritmetica immediatamente precedente
- Se uno dei registri da comparare è il registro di destinazione di un'istruzione di load precedente (2^{nda} precedente)
 - È necessario 1 ciclo di stallo(*)

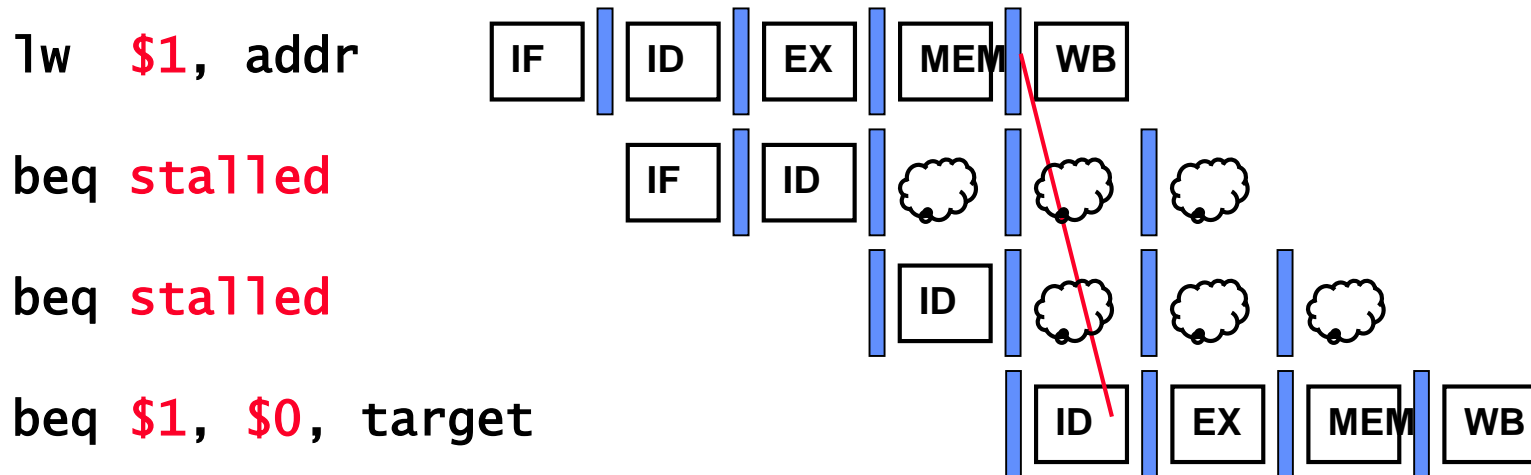


Comparazione tra \$1 e \$4 anticipata a ID.
Servono da subito i valori corretti dei registri,
non quelli letti dal Register File

(*) in molti esercizi svolti con il *delayed branch* questi cicli di stallo sono ignorati

Forwarding e Calcolo del branch anticipato

- Se uno dei registri da comparare è il registro di destinazione dell'istruzione di load immediatamente precedente
 - Sono necessari 2 cicli di stallo ☹



(*) in molti esercizi svolti con il *delayed branch* questi cicli di stallo sono ignorati

Dipendenze sui dati

- **Disallineamento** dei cicli in cui le varie istruzioni leggono e/o producono i dati da scrivere nei registri
- Istruzioni **aritmetiche**:
 - Producono il dato da scrivere nel *register file* al 3° ciclo, durante il quale hanno bisogno dei dati corretti in ingresso alla ALU
- Istruzioni di **load**:
 - Calcolano l'indirizzo al 3° ciclo, durante il quale hanno bisogno del dato corretto in ingresso alla ALU
 - Producono il dato da scrivere nel *register file* al 4° ciclo, durante il quale hanno bisogno del dato corretto in ingresso alla MEMORIA
- Istruzioni di **branch**:
 - Producono il dato da scrivere nel PC al 2° ciclo, durante il quale hanno bisogno dei dati corretti in ingresso al COMPARATORE
- Istruzioni di **store**:
 - Calcolano l'indirizzo al 3° ciclo, durante il quale hanno bisogno del dato corretto in ingresso alla ALU
 - Al 4° ciclo hanno bisogno del dato corretto in ingresso alla MEMORIA

Eliminare gli stalli dovuti alle criticità sul controllo

- Attendere sempre che l'indirizzo di salto sia stato calcolato correttamente porta a rallentare il funzionamento della pipeline
 - è una soluzione conservativa, che immette sempre bolle nella pipeline
 - i branch sono purtroppo abbastanza frequenti nel codice
- Per eliminare quanti più stalli possibile, solitamente si adotta una soluzione speculativa, basata sulla **previsione** del risultato del salto condizionato
 - lo stadio IF potrà quindi, da subito, effettuare il fetch “corretto” della **prossima istruzione da eseguire**

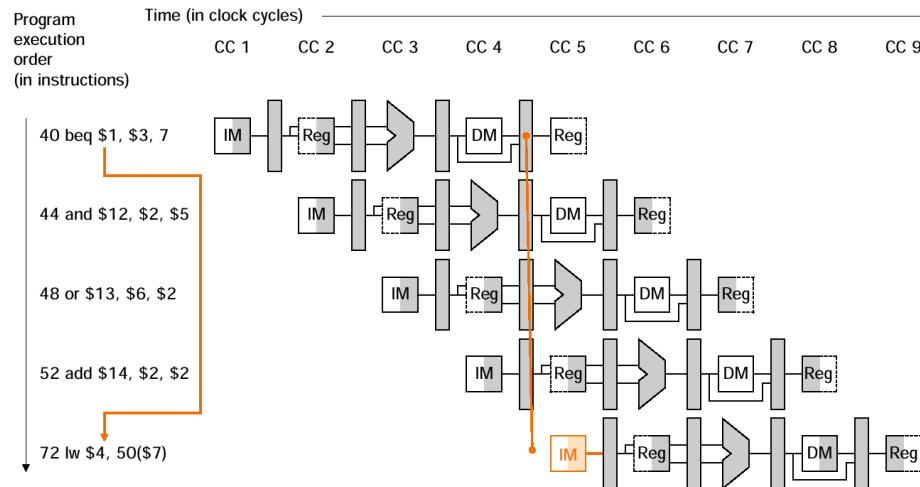


Eliminare gli stalli dovuti alle criticità sul controllo

- Problema con la soluzione speculativa:
 - cosa succede se la **previsione non risulterà corretta ?**
 - sarà necessario 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:
 - ovvero, farle proseguire come *nop operation* fino all'uscita dalla pipeline

Previsione semplice

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



- Se almeno nella metà dei casi il salto è *not-taken*, questa previsione dimezza i possibili stalli dell'**approccio conservativo**:
 - che prevede di bloccare il prossimo fetch fino al completamento del branch, con la scrittura del valore corretto nel PC

Previsione dinamica dei branch

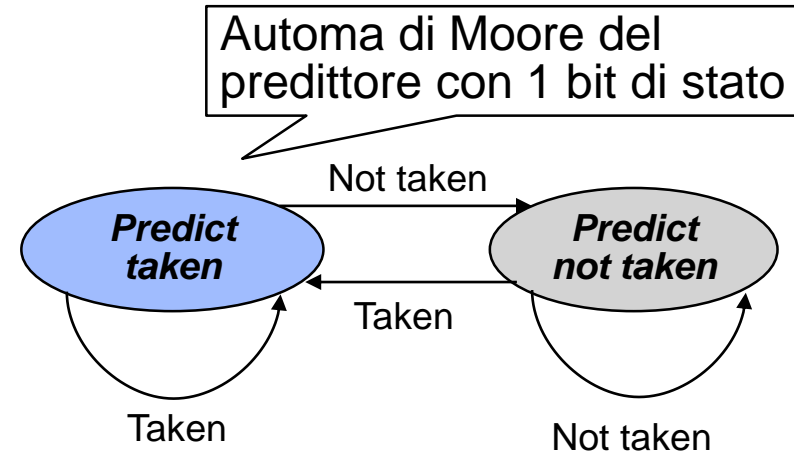
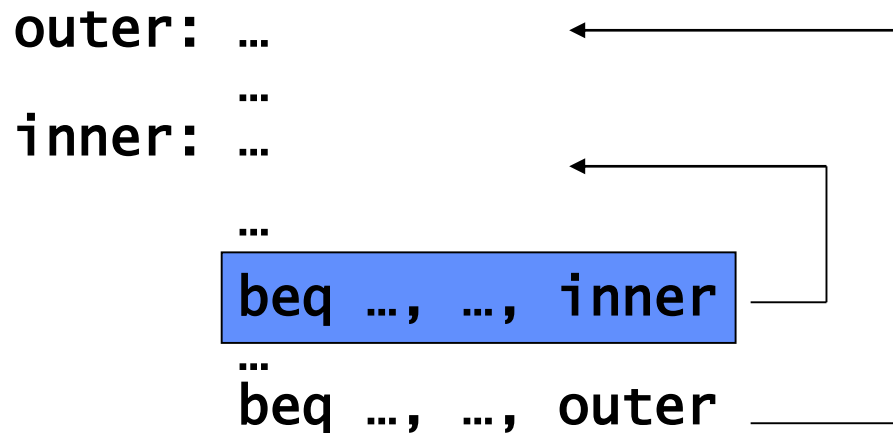
- Per pipeline profonde, il branch penalty (numero di istruzioni da nullificare) potrebbe essere molto più significativo
 - La previsione semplice, sempre *not-taken*, introdurrebbe ritardi dovuti alla numerose istruzioni da nullificare
- Usiamo la **previsione dinamica**, mantenendo una history table
 - indirizzata tramite gli **indirizzi delle istruzioni** di salto
 - nella tabella poniamo anche **l'indirizzo dell'istruzione successiva** al salto nel caso di **branch taken** (l'indirizzo è calcolato alla prima esecuzione)
 - nella tabella viene memorizzato 1 o più bit (**stato**) per memorizzare il risultato dell'esecuzione di ciascun salto (**taken** o **not-taken**)

Previsione dinamica dei branch

- Ogni volta che si esegue un branch
 - Controlliamo la tabella, leggiamo lo stato associato
 - Nota che la tabella può essere acceduta con l'indirizzo (PC) dell'istruzione nello stadio IF
 - Se esiste l'indirizzo in tabella, allora l'istruzione è un branch
 - Ci aspettiamo che lo stato corrisponda alla previsione corretta (se *l'istruzione di branch è già stata eseguita*)
 - **taken**: ind. calcolato e memorizzato
oppure
 - **not-taken**: PC+4
 - Effettuiamo il fetch della prossima istruzione da **fall-through** (PC+4) o da **target** (PC+4+displ)
 - Se previsione errata, flush della pipeline (**nullificazione** delle istruzioni in pipeline) e modifica della previsione in tabella (**cambio stato**)

Predittore con stato a un 1-Bit (2 stati): difetti

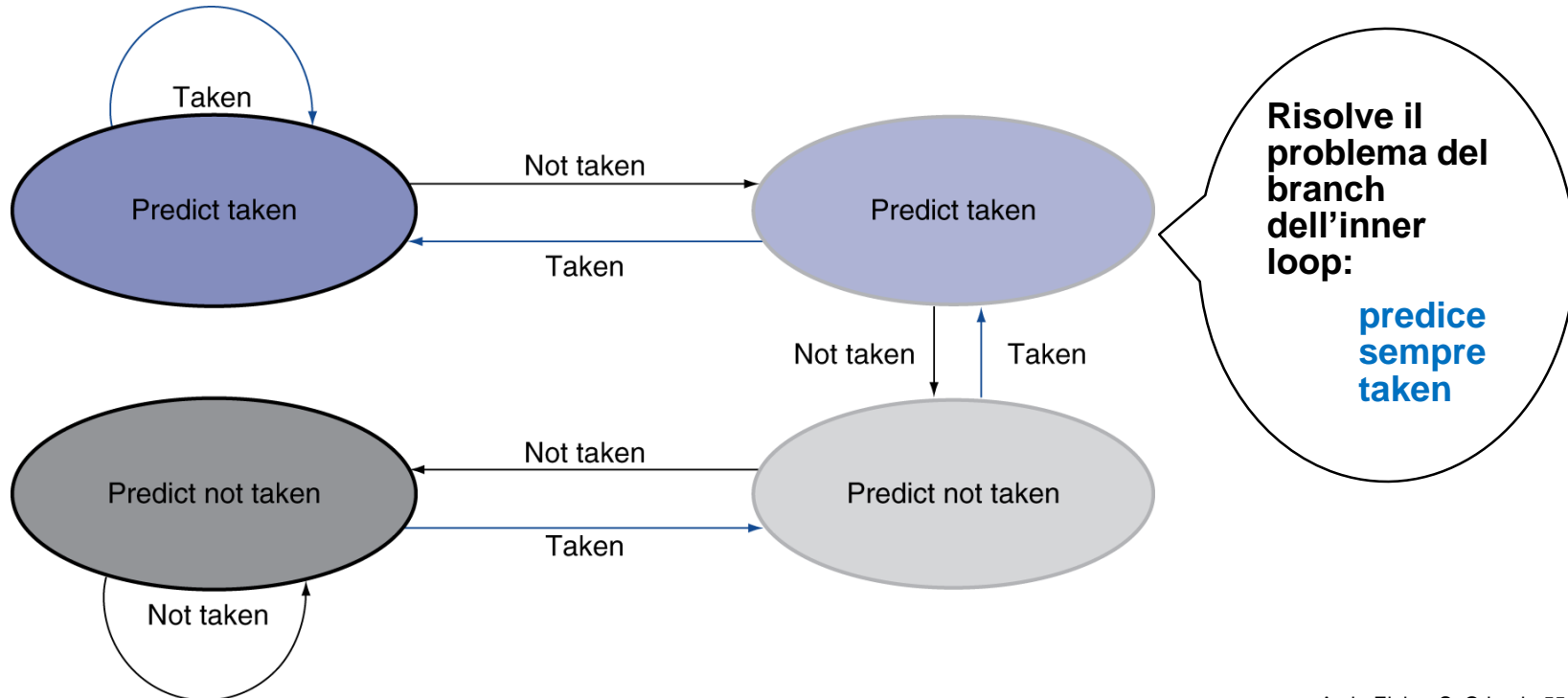
- Se la previsione fosse solo basata sull'**ultimo risultato**
 - I branch degli inner loop sono predetti male due volte di seguito!



- Il beq del loop interno è **not-taken** dopo una sequenza di **taken**
 - Si esce dal loop, ma la previsione (**taken**) è sbagliata (**1^a volta**)
- Si rientra nel loop interno e si riesegue il beq interno
 - La predizione (**not-taken**) è sbagliata (**2^a volta**)
- Il beq interno è *not-taken* solo raramente
 - Sarebbe meglio prevedere sempre taken!

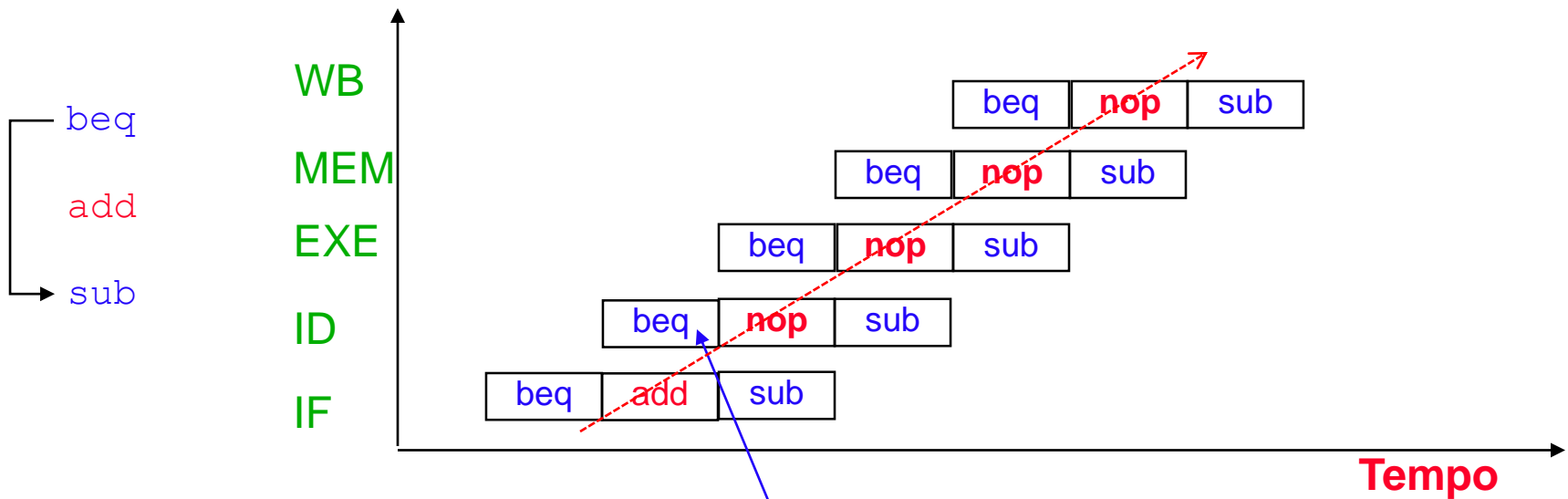
Predittore con stato da 2-Bit (4 stati)

- **IDEA: Cambiamo predizione solo dopo 2 mis-predizioni consecutive**
- 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 corretta, 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 letta dallo stadio IF (**add**),
in modo che questa prosegua come se fosse una **nop**

Delayed branch

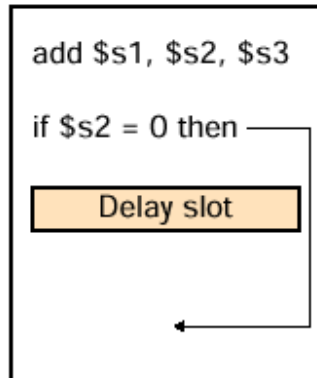
- Processori moderni fanno affidamento
 - sulla previsione dei salti, e
 - sull'annullamento delle istruzioni caricate in caso di previsione errata
- Il **vecchio processore MIPS** usava una tecnica molto più semplice, che non richiedeva hardware speciale, facendo affidamento solo sul software, per risolvere dipendenze su dati e controllo
 - 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

Delayed branch

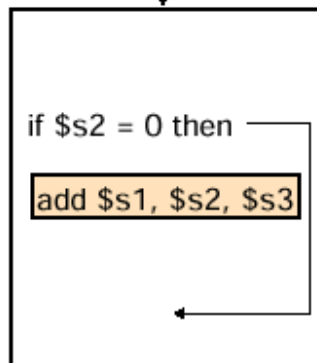
- La tecnica è nota come **salto ritardato**: il ritardo corrisponde ad un certo numero di *branch delay slot*
 - soluzione speculativa che prevede la modifica, a compile time, dell'ordine di invio delle istruzioni nella pipeline
 - gli slot dopo il branch devono essere riempiti con istruzioni che verranno comunque eseguite prima che l'indirizzo successivo al branch sia 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 ! \Rightarrow *difficile trovare tante istruzioni eseguibili nello slot*
 - Soluzione: *prediction*

Delayed branch

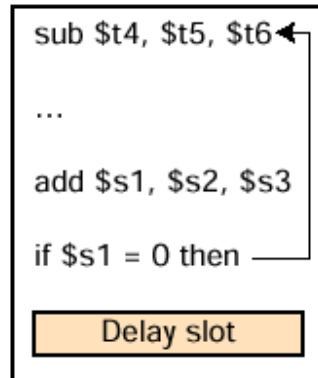
a. From before



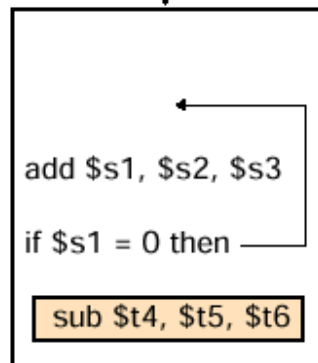
Becomes



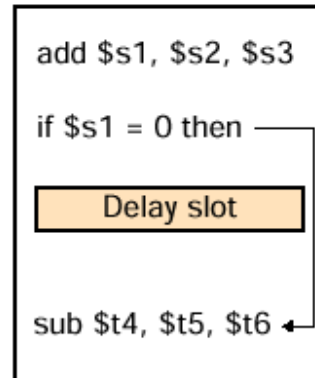
b. From target



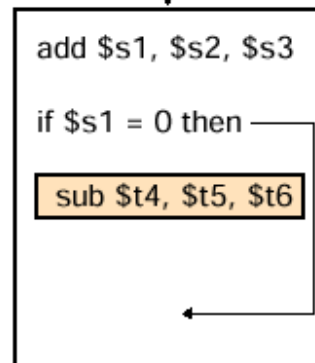
Becomes



c. From fall through



Becomes



- Nel caso **a)**, l'istruzione *precedente* può 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**) dall'istruzione `add` spostata nel delay slot 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
```

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

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

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

Anche questa dipendenza provoca uno stallo: è come se ci fosse una **nop** tra **addi** e **bne**

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

Nell'esempio possiamo eliminare entrambi gli stalli spostando indietro, senza modificare l'ordine di esecuzione stabilito dalle dipendenze, l'istruzione:

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

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

Diagramma di esecuzione

(codice originale con **nop** nei delay slot)

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

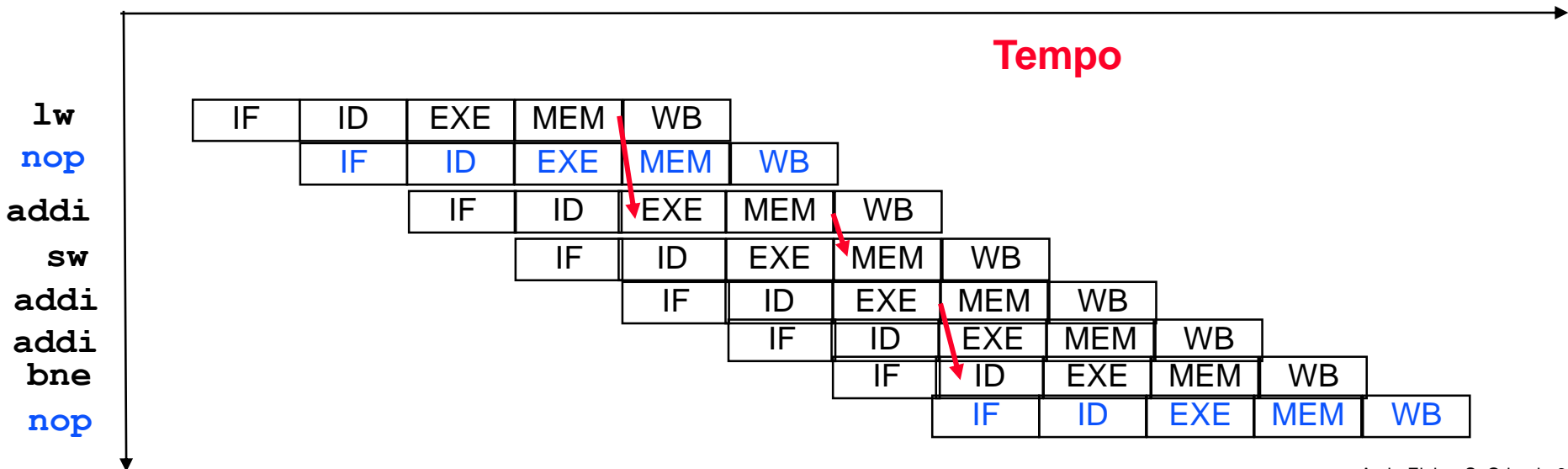
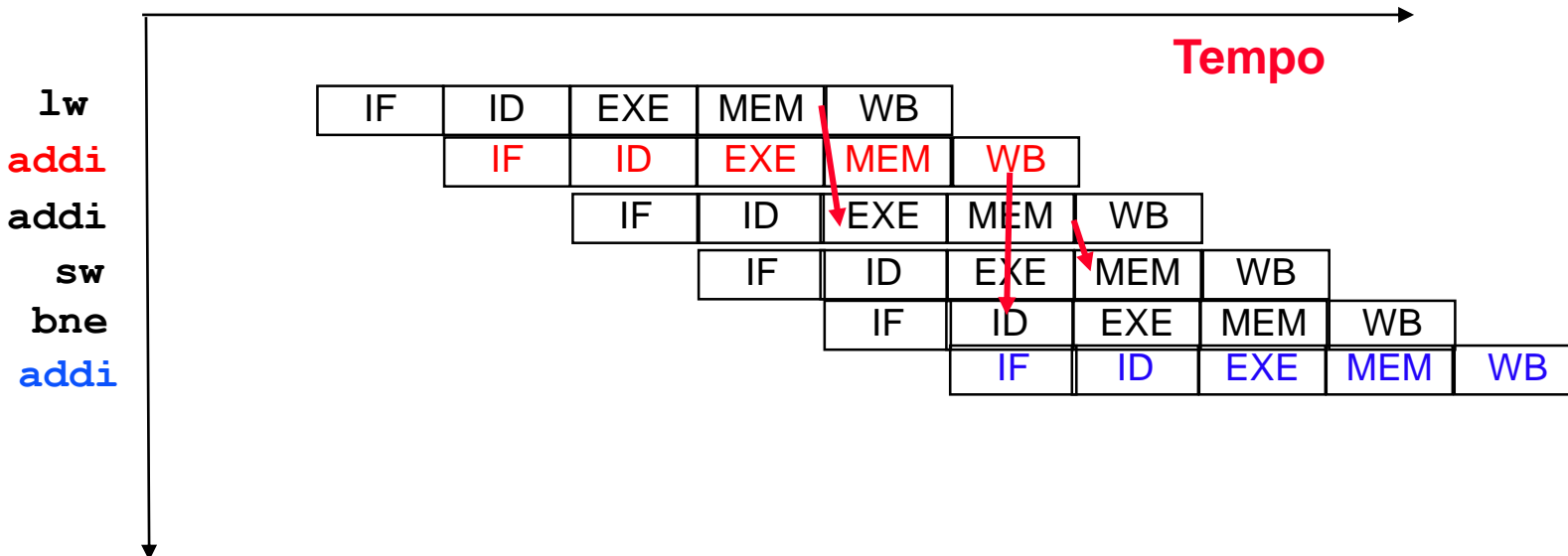


Diagramma di esecuzione (codice ottimizzato)

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



Confronto tra diversi schemi di progetto

- 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
 - $T_{\text{singolo}} = IC * CPI * \text{Periodo_clock} = IC * 8 \text{ ns}$
- Multiciclo
 - Ciclo di clock (periodo) = 2 ns
 - calcolato sulla base del passo più "costoso"
 - $$CPI_{\text{avg}} = 0.22 CPI_{lw} + 0.11 CPI_{sw} + 0.49 CPI_R + 0.16 CPI_{br} + 0.02 CPI_j =$$
$$0.22 * 5 + 0.11 * 4 + 0.49 * 4 + 0.16 * 3 + 0.02 * 3 = 4.04$$
 - $T_{\text{multi}} = IC * CPI_{\text{avg}} * \text{Periodo_clock} = IC * 4.04 * 2 \text{ ns} = IC * 8.08 \text{ ns}$

Confronto tra diversi schemi di progetto

- **Pipeline**

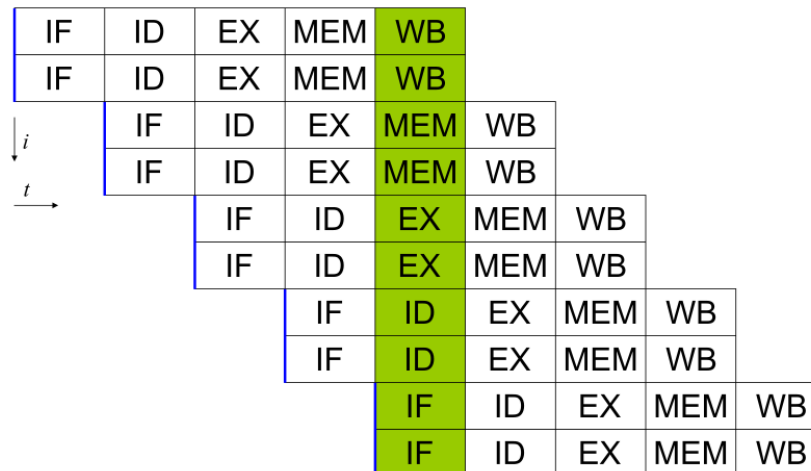
- Ciclo di clock (periodo) = **2 ns**
 - calcolato sulla base dello stadio più “costoso”
- Nel calcolo di CPI, non consideriamo il tempo di riempimento della pipeline (ininfluente)
 - CPI = 1: un’istruzione è completata ad ogni ciclo di clock
 - $CPI_{sw} = 1$ $CPI_R = 1$ $CPI_j = 2$
 - CPI_{lw} : per il 50% dei casi lw seguita da un’istruzione che legge il registro scritto (stallo di 1 ciclo)
 - $CPI_{lw} = 1.5$
 - CPI_{br} : per il 25% dei casi, la previsione dell’indirizzo del salto è 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**

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

Processori superscalari e prestazioni

- I processori che inviano dinamicamente più istruzioni (*multi issue*) sono chiamati **superscalari**
- Versione con invio **in-order**
 - istruzioni inviate **in-order**: il controllo decide se zero, una o più **istruzioni indipendenti** (lette dal flusso sequenziale) possono essere inviate ad ogni ciclo di clock.
 - Il compilatore è importante per rischedulare istruzioni ed eliminare dipendenze, in modo da facilitare gli invii multipli di istruzioni
- Versione con invio **out-of-order**
 - Il processore schedula **dinamicamente** quali istruzioni eseguire (**out-of-order**), cambiando l'ordine per mantenere le unità funzionali occupate

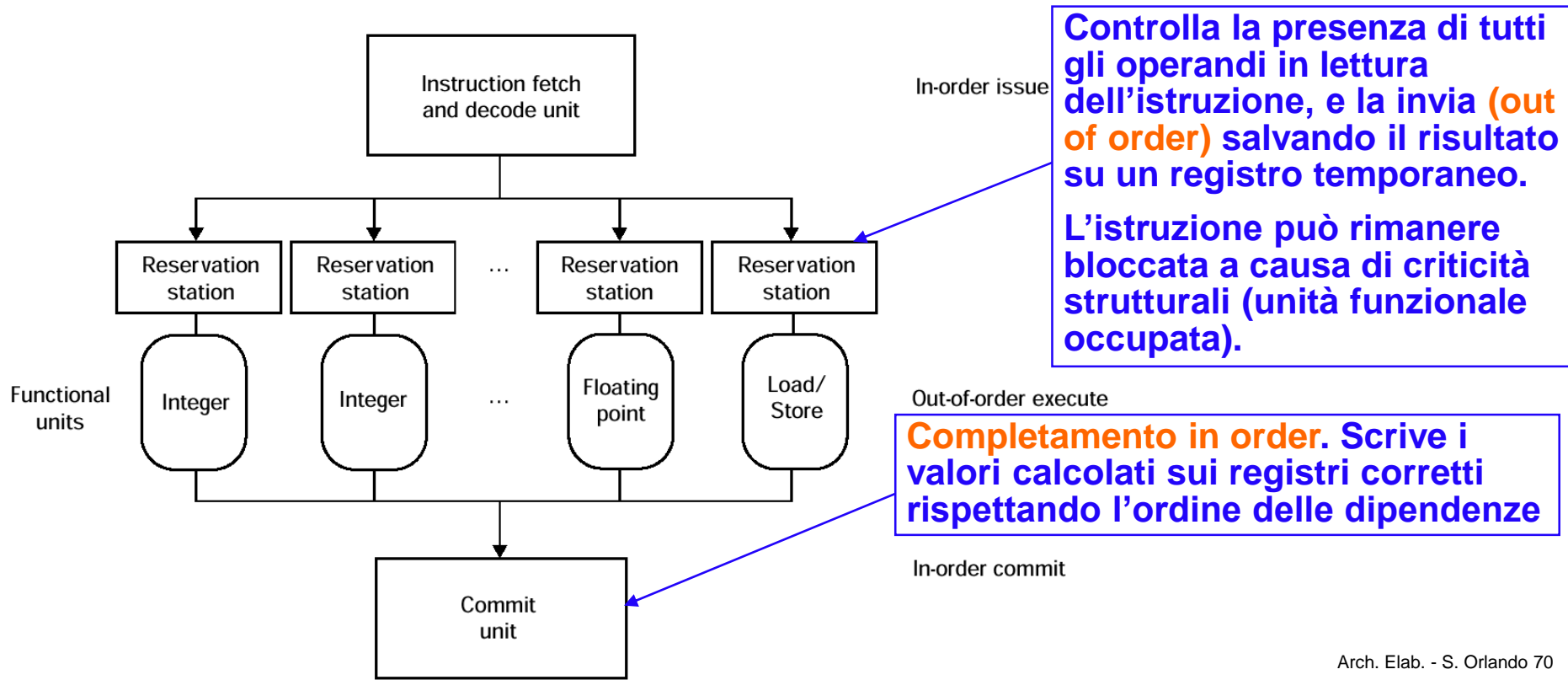


Processori superscalari & performance

- Se **più istruzioni sono inviate** ad ogni ciclo, il **CPI medio diventa < 1**
- Dato un microprocessore a **4GHz**, con invio multiplo a **quattro-vie**
 - Ad ogni ciclo ($\frac{1}{4} = 0.25 \text{ nsec}$), sono completate 4 istruzioni
 - **CPI = 0.25**
 - Ogni $\text{nsec} = 10^{-9} \text{ sec}$ (4 cicli), sono completate **16 istruzioni**
 - Performance di picco: **16 miliardi di Istr. per sec.**
 - **16 GFLOPs se le istruzioni sono floating point**
- I processori di alta fascia sono attualmente in grado di inviare fino a 6 istruzioni per ciclo
 - ma ci sono diversi vincoli che impediscono di sfruttare questo parallelismo, che richiederebbe di individuare fino a 6 istruzioni da inviare per ciclo

Processori superscalari e dinamici

- I processori moderni **supescalari** sono in grado di
 - inviare più istruzioni contemporaneamente
 - le istruzioni inviate contemporaneamente devono essere “indipendenti”
 - l’ordine di esecuzione delle istruzioni rispetto a quello fissato dal flusso di controllo del programma può essere modificato (**scheduling dinamico**)
 - per evitare stalli dovuti a dipendenze o cache miss



Dipendenze sui dati e scheduling dinamico

- Le criticità dovute 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 (**out-of-order execution**), e le scritture avvengono scrivendo in **registri temporanei interni**
 - l'unità di commit si farà poi carico di ordinare (**in-order commit**) tutte le scritture (dai registri temporanei a quelli del register file)

Criticità sui dati e processori moderni

- I processori moderni sfruttano il fatto che le dipendenze WAW e WAR sono «**deboli**»
 - WAW (Write After Write) : un'istruzione **scrive** un registro **scritto** da un'istruzione precedente
 - 1) add \$s1, \$t0, \$t1 # Write \$s1
 - 2) sub \$s1, \$s2, \$s3 # Write \$s1
 - WAR (Write After Read) : un'istruzione **scrive** un registro **letto** da un'istruzione precedente
 - 1) add \$t0, \$s1, \$t1 # Read \$s1
 - 2) sub \$s1, \$s2, \$s3 # Write \$s1
- In verità possiamo «eseguire» le due istruzioni scambiandone l'ordine o eseguendole in parallelo, ma non possiamo completarne l'esecuzione
- E' importante che le scritture avvengano in-order, rispettando l'ordine fissato dal programma. Ad esempio, se consideriamo la dipendenza WAW dell'esempio, l'ultima scrittura in \$s1 deve essere quella effettuata dalla sub

Dal calcolo sequenziale al calcolo parallelo

- **Il power wall**

- Limite all'aumento dell'energia utilizzata è dato dalla capacità di raffreddamento
- Comunque, nell'era del PostPC: **energia = risorsa critica**
 - Carica della batteria nei mobile device, Data center dei cloud (dare energia e raffreddare migliaia di server)

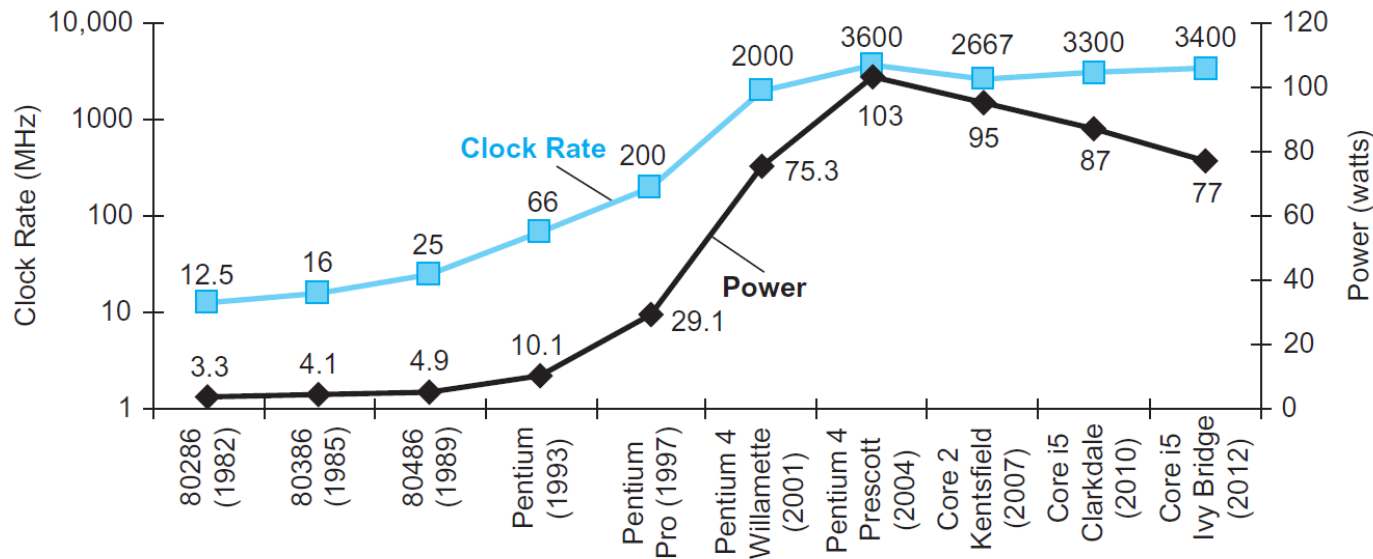


FIGURE 1.16 Clock rate and Power for Intel x86 microprocessors over eight generations and 25 years. The Pentium 4 made a dramatic jump in clock rate and power but less so in performance. The Prescott thermal problems led to the abandonment of the Pentium 4 line. The Core 2 line reverts to a simpler pipeline with lower clock rates and multiple processors per chip. The Core i5 pipelines follow in its footsteps.

Dal calcolo sequenziale al calcolo parallelo

- **Riduzione delle prestazioni**

- SPECint sequenziale rispetto al VAX11/780
- Dal 2002, il power wall, la ridotta disponibilità di parallelismo ILP (instruction-level parallelism), la lunga latenza della memoria rispetto alle prestazioni potenziali dei processori, hanno causato la riduzione dell'incremento annuo delle prestazioni

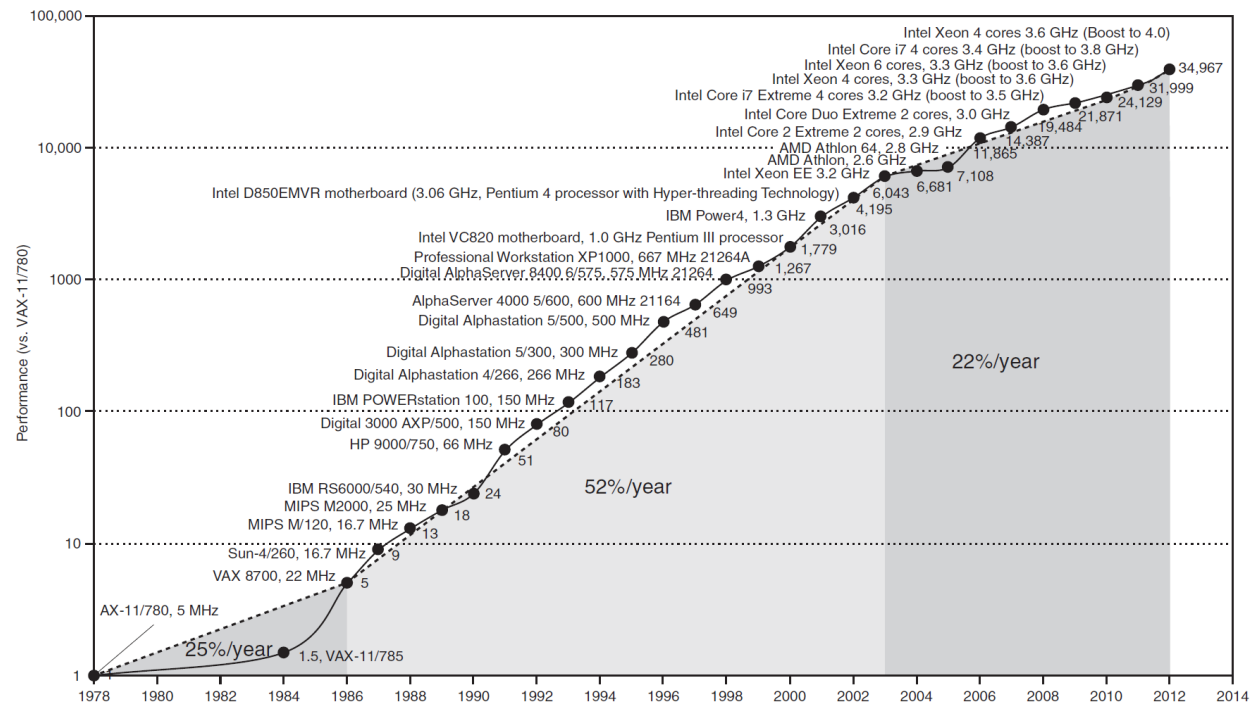
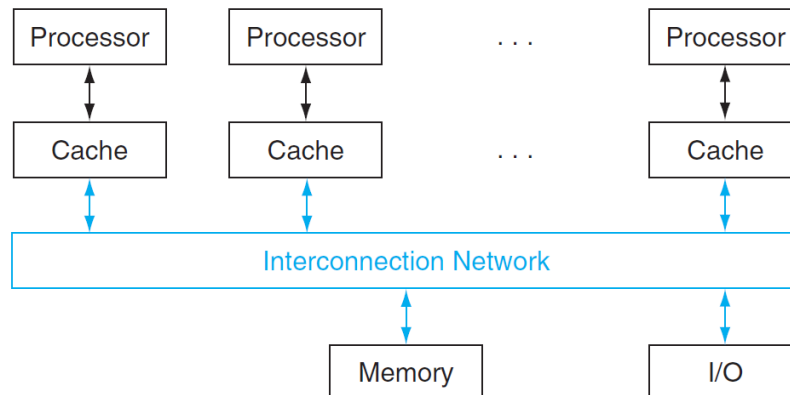


FIGURE 1.17 Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.10). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. The higher annual performance improvement of 52% since the mid-1980s meant performance was about a factor of seven higher in 2002 than it would have been had it stayed at 25%. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 22% per year.

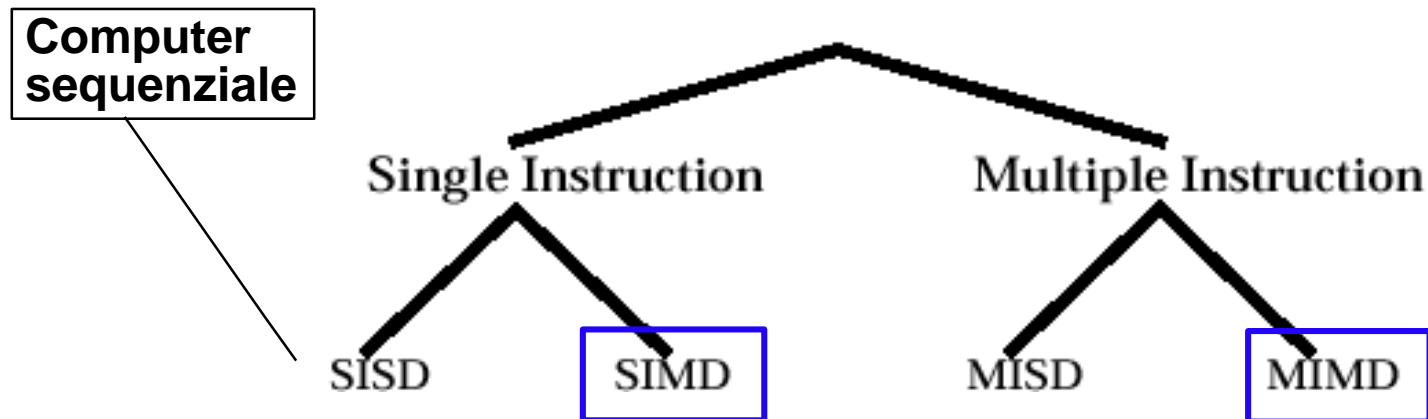
Dal calcolo sequenziale al calcolo parallelo

- Organizzazione **multiprocessore** diventata mainstream negli attuali **microprocessori**
- Sono stati chiamati **multicore microprocessors** invece di **multiprocessor microprocessors** forse per evitare ridondanze nel nome
 - I processori sono anche chiamati **core** nei chip multicore chip
 - Si prevede un aumento del numero di cores se continua il trend di crescita dei transistor per chip (legge di Moore)
- I multicore sono quasi sempre Shared Memory Processors (SMPs)
 - I core accedono allo **stesso spazio fisico di memoria**

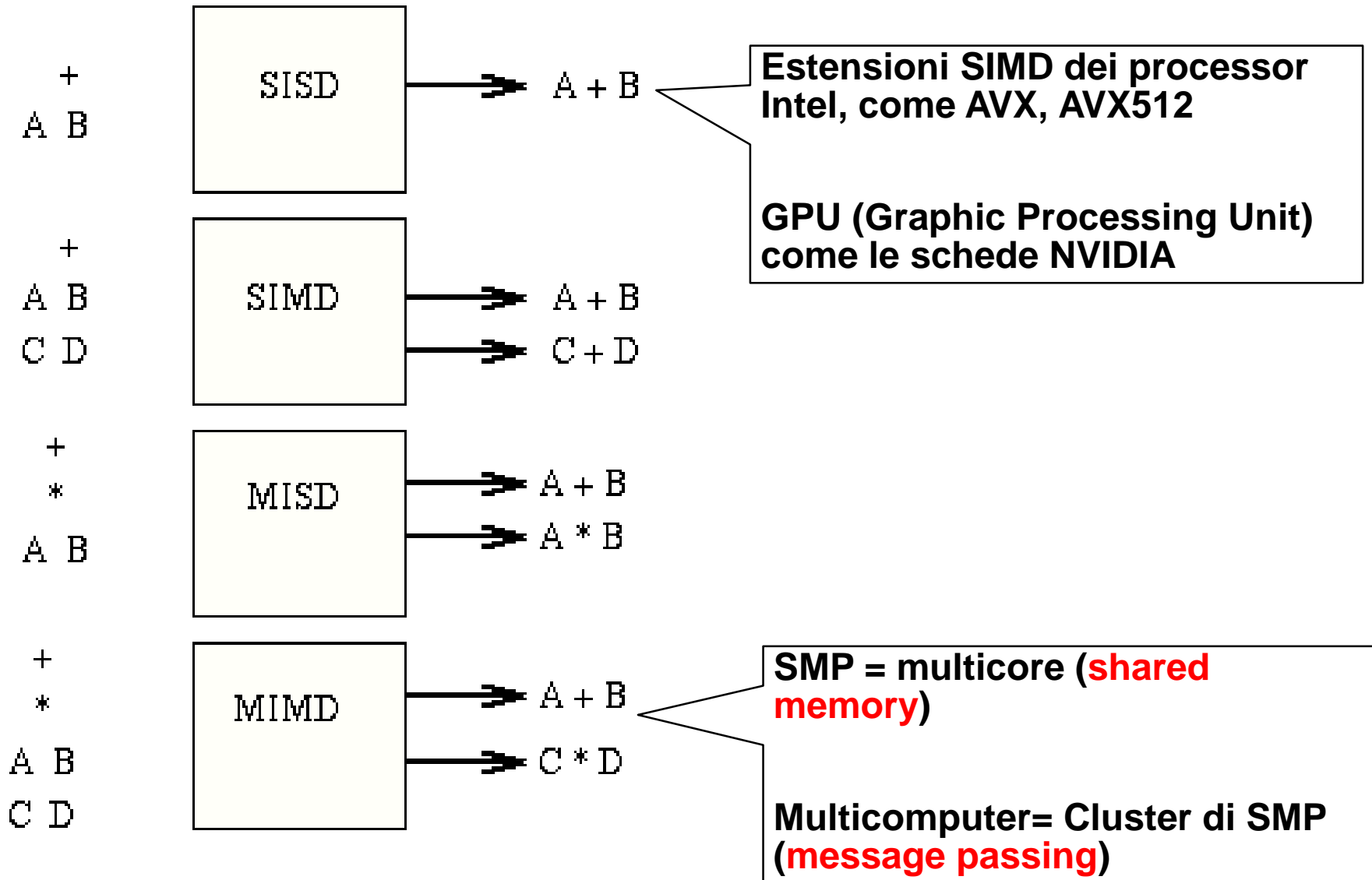


Tassonomia dei computer (sequenziali/paralleli)

- **Flynn** [1972] ha introdotto la seguente **tassonomia** dei computer, dove le classi interessanti di **computer paralleli** sono **MIMD** (Multiple Instruction stream-Multiple Data stream) e **SIMD** (Single Instruction stream - Multiple Data stream)
 - **SIMD**: controllo unico e centralizzato che distribuisce la stessa istruzione ai vari processori (che operano su dati distinti)
 - **MIMD**: ogni processore ha il suo proprio controllo e il proprio flusso di istruzioni, e quindi può eseguire istruzioni differenti su dati distinti.
I **multicore** sono MIMD.



Potenzialità dei 4 modelli di computer



Dal calcolo sequenziale al calcolo parallelo

- **Passato**

- I programmatori **raddoppiavano la performance dei loro programmi ogni anno** senza modificare il codice, facendo affidamento sulle innovazioni:
 - delle tecnologie costruttive dell'hardware
 - delle architecture (ILP e Gerarchie di memoria)
 - dei compilatori



- **Oggi**

- Se i programmatori necessitano di migliorare i tempi di risposta dei loro programmi, devono **riscrivere i programmi con thread paralleli espliciti** per sfruttare in modo vantaggioso i **core multipli** (multiprocessori/multicore)
- Inoltre, i programmatori devono continuare a migliorare/ottimizzare i programmi all'aumentare del numero di core

Dal calcolo sequenziale al calcolo parallelo



Numero di core (repliche di processori) per il cui sfruttamento è necessario usare **programmi paralleli**

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/Speculation	Cores/Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	1	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

FIGURE 4.73 Record of Intel Microprocessors in terms of pipeline complexity, number of cores, and power. The Pentium 4 pipeline stages do not include the commit stages. If we included them, the Pentium 4 pipelines would be even deeper.

Processori superscalari, con invio multiplo di istruzioni

Dal calcolo sequenziale al calcolo parallelo

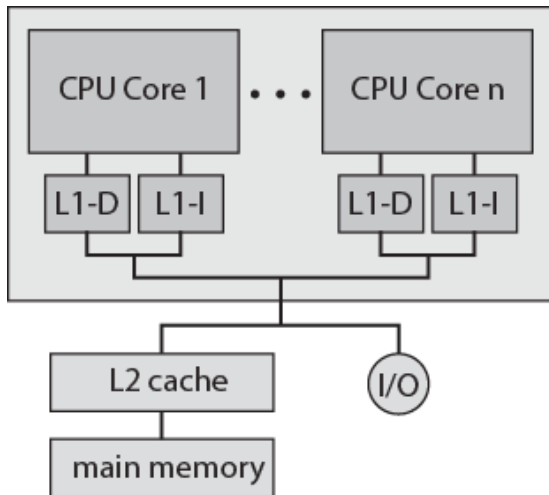


Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	14	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	2-level	2-level
1st level caches / core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level cache / core	128 - 1024 KiB	256 KiB
3rd level cache (shared)	–	2 - 8 MiB

FIGURE 4.74 Specification of the ARM Cortex-A8 and the Intel Core i7 920.

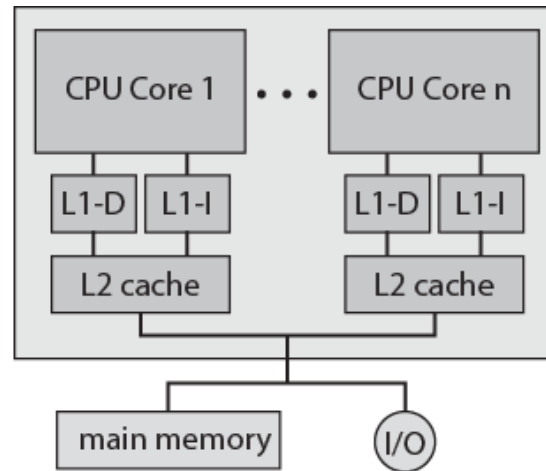
Multicore UMA – Uniform Memory Access (Multiprocessori MIMD a memoria condivisa)

ARM11 MPCore



(a) Dedicated L1 cache

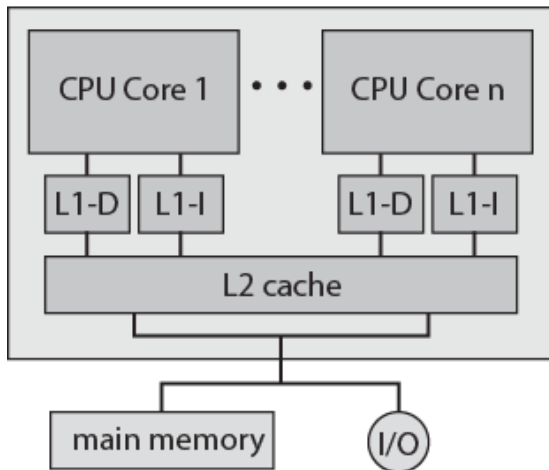
AMD Opteron



(b) Dedicated L2 cache

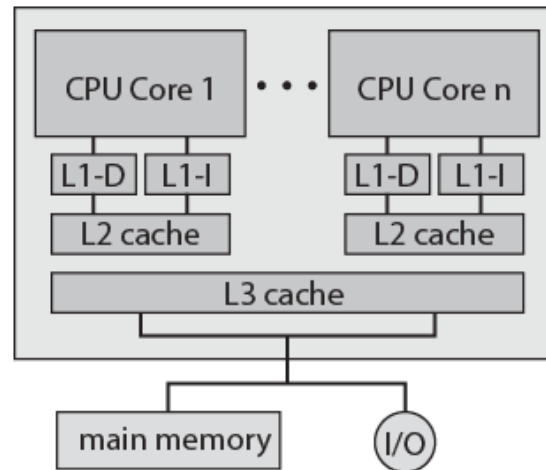
Cache
dedidata
on-chip

Intel Core Duo



(c) Shared L2 cache

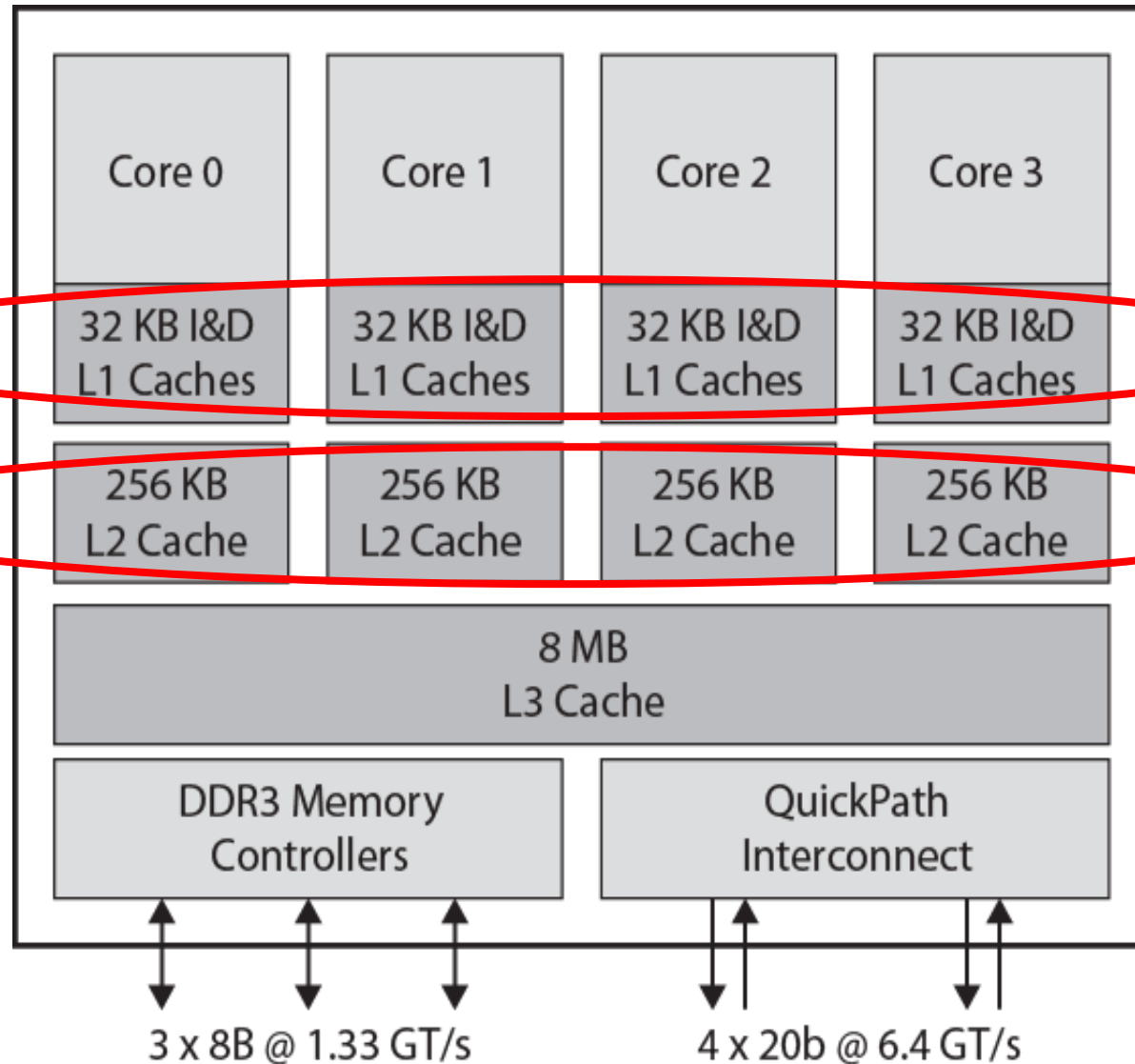
Intel Core i7



(d) Shared L3 cache

Cache
condivise
on-chip

Intel Core i7 Block Diagram

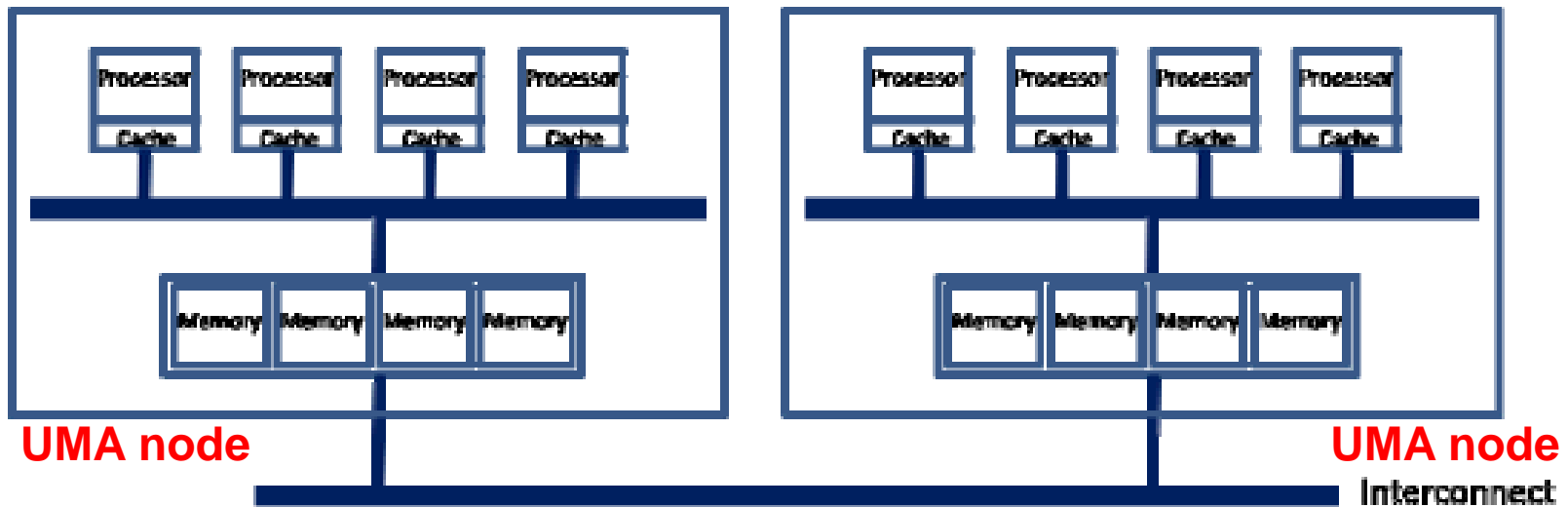


Cache coherence:

Repliche dello stesso blocco di memoria in cache diverse devono essere mantenute aggiornate

UMA/NUMA Intel Core i7 (Xeon)

- Possiamo mettere assieme diversi multicore UMA per creare un'architettura UMA/NUMA (NUMA = Non-Uniform Memory Address)
- Processori/core in un nodo UMA
 - Accesso condiviso ai moduli di memoria
 - Possono **accedere la memoria in nodi remoti** usando la **comunicazioni tra i nodi**, con penalizzazioni nella performance
- Il multicore Intel Core i7 usa l'interconnessione veloce nota come **QuickPath Interconnect** (QPI) che mitiga il problema degli accessi lenti alle memorie remote



Multicomputer = Cluster di multiprocessori (Multiprocessori MIMD message-passing)

- Memoria e Cache private per ogni nodo
- Reti di interconnessione tra i nodi
- I processi scambiano dati privati tramite *message passing* (MP)
 - Send + Matching Recv per copiare tra le memorie private per sincronizzare le attività dei processi

Nodo (qui rappresentato come singolo core, ma in realtà multicore o SMP NUMA)

