

Architettura degli Elaboratori e Laboratorio

Matteo Manzali

Università degli Studi di Ferrara

Anno Accademico 2016 - 2017

Criticità sui dati

- Questo è l'esempio utilizzato per spiegare il funzionamento della pipeline:


```
1000: lw    $8, 4($29)
1004: sub   $2, $4, $5
1008: and   $9, $10, $11
1012: or    $16, $17, $18
1016: add   $13, $14, $0
```

Dalla lezione
precedente


- Queste istruzioni sono **indipendenti**:
 - ogni istruzione legge e scrive registri differenti
 - il datapath riesce a gestirle senza problemi
- Purtroppo molto spesso le sequenze di istruzioni non sono indipendenti.



Esempio con dipendenze

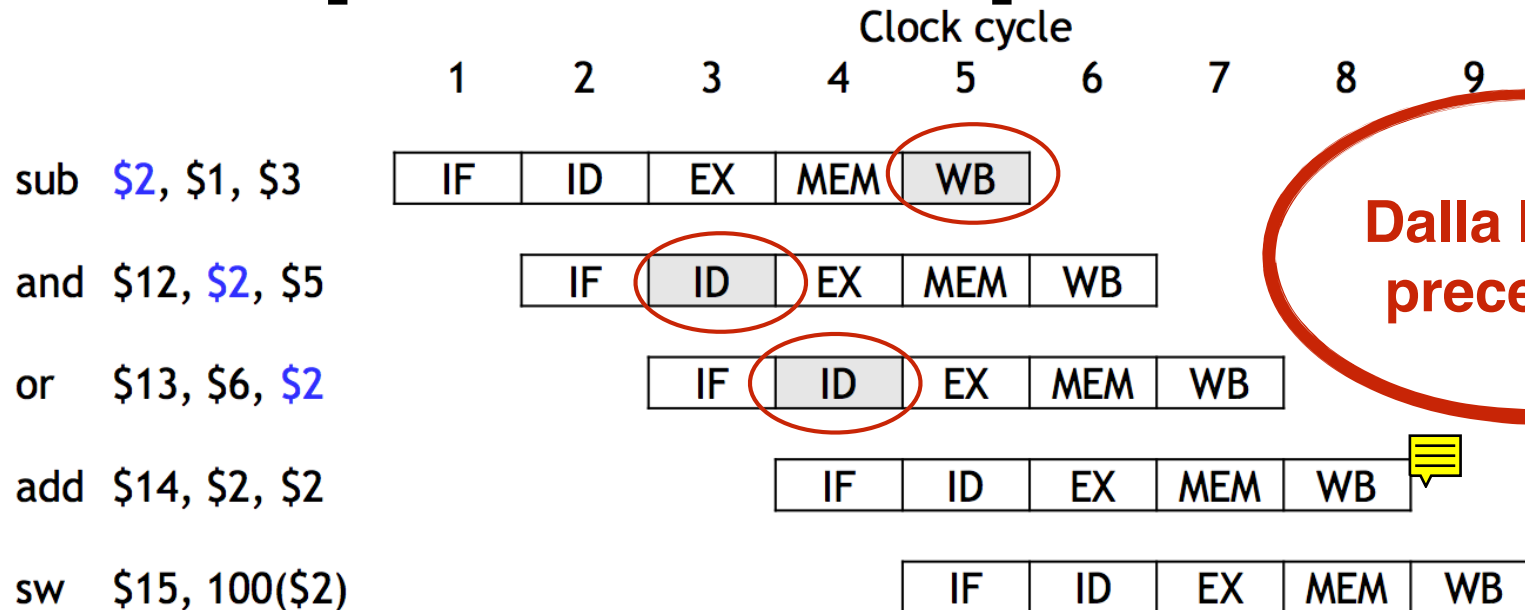
sub	\$2, \$1, \$3
and	\$12, \$2, \$5 
or	\$13, \$6, \$2
add	\$14, \$2, \$2
sw	\$15, 100(\$2)

**Dalla lezione
precedente**

- Questa sequenza di istruzioni non è un problema per il datapath a singolo ciclo di clock: 
 - ogni istruzione viene eseguita prima che la successiva cominci
 - questo assicura che non ci siano conflitti, nonostante tutte le istruzioni coinvolgano il registro \$2
- Con la pipeline invece che problemi potrebbero esserci?



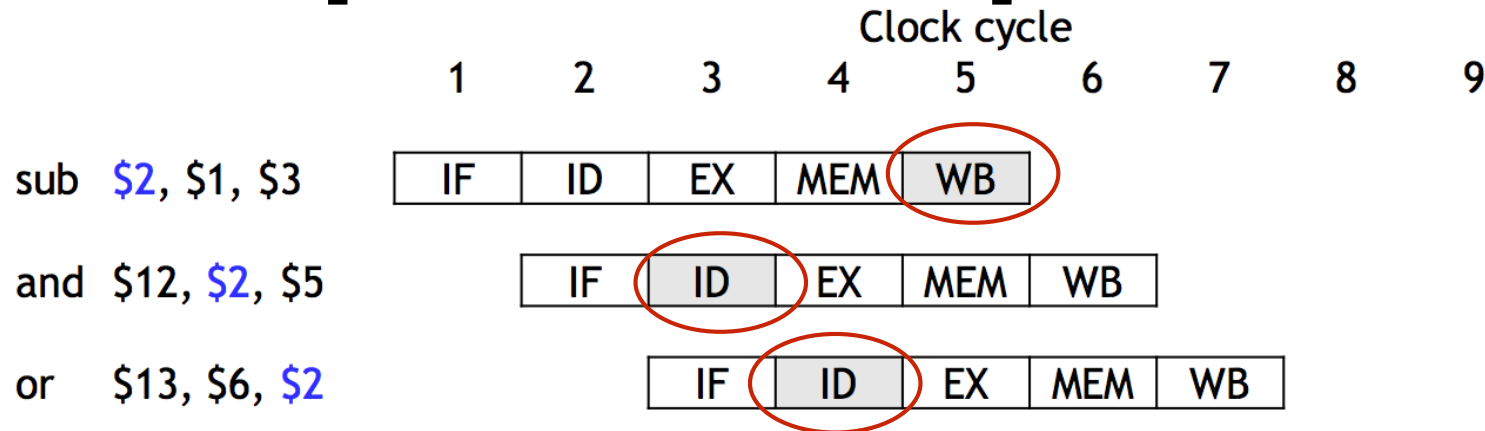
Esempio con dipendenze



- L'istruzione SUB scrive il risultato nel registro \$2 solo al ciclo 5, questo causa due criticità:
- L'AND legge il registro \$2 al ciclo 3, dal momento che SUB non ha ancora scritto il risultato, verrebbe letto il valore vecchio di \$2
- stesso discorso per l'OR, che usa il registro \$2 al ciclo 4



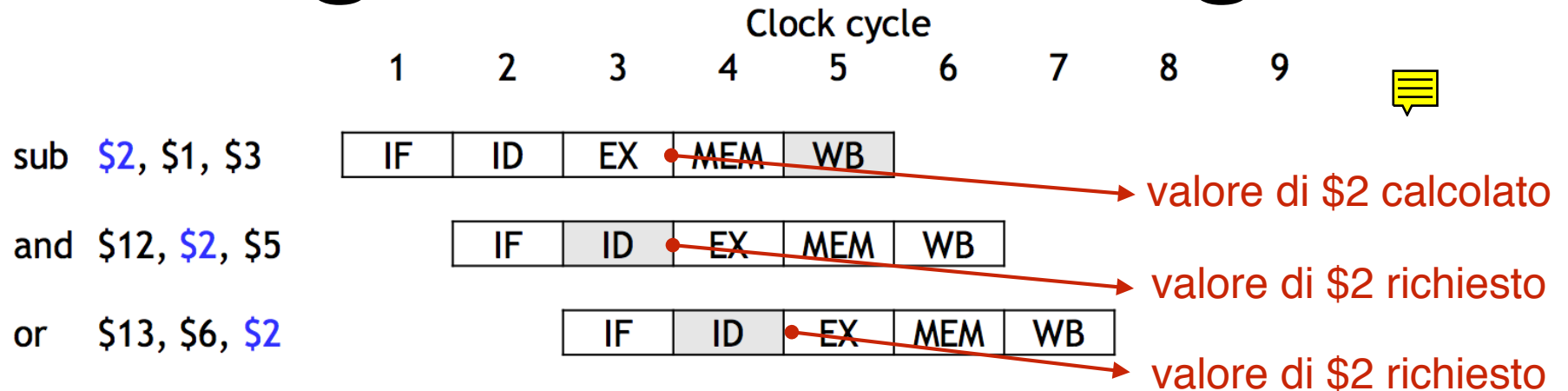
Esempio con dipendenze



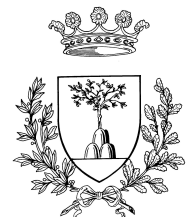
- Dobbiamo fare in modo che le istruzioni AND e OR utilizzino il valore aggiornato di \$2.
- **Come fare? Idee?**



Uno sguardo in dettaglio

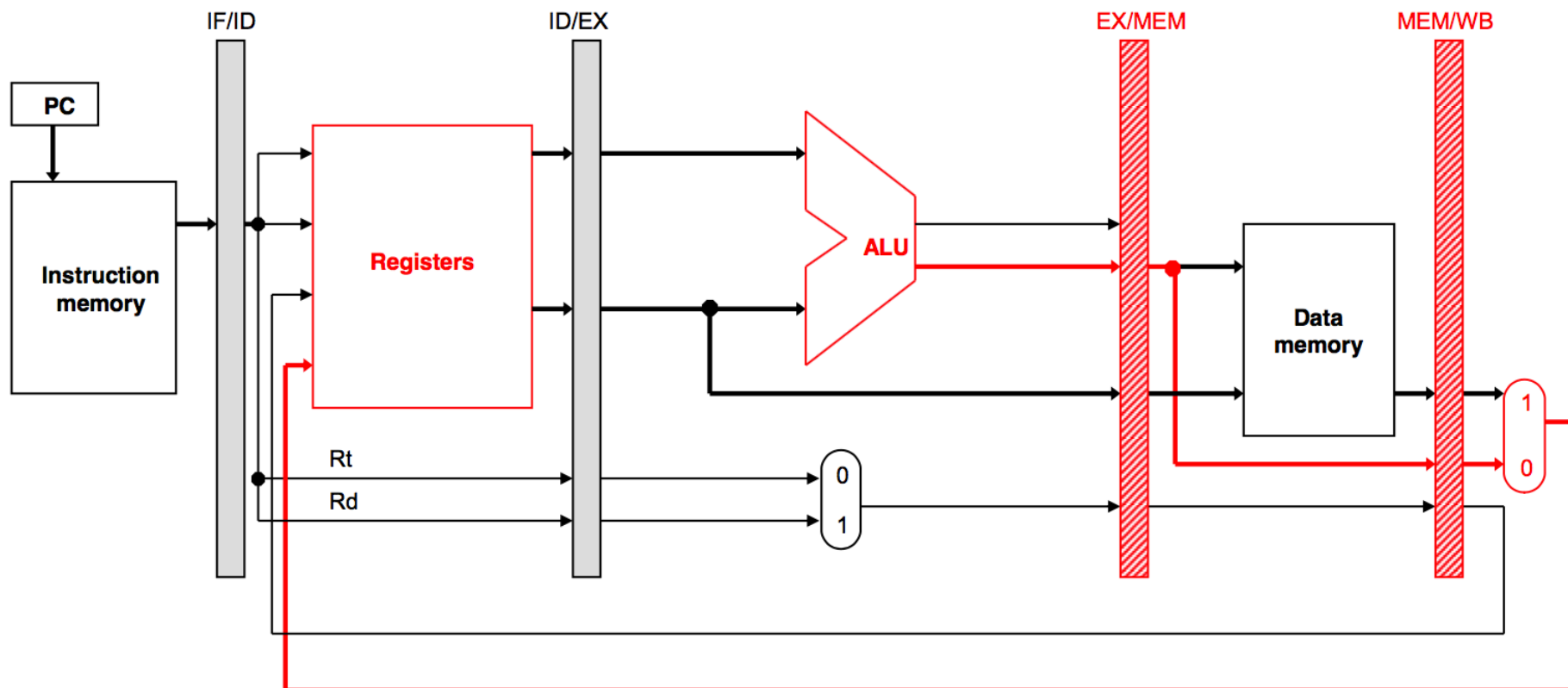


- La SUB calcola il risultato nella fase EX.
- La AND e la OR richiedono il valore corretto nella fase EX.
- Temporalmente il risultato è disponibile **prima** che venga richiesto:
 - si trova solo nel “posto sbagliato”
- Dobbiamo saltare la fase WB dell’istruzione SUB e portar il valore direttamente in input alla fase EX delle istruzioni AND e OR.



Dove trovare il risultato

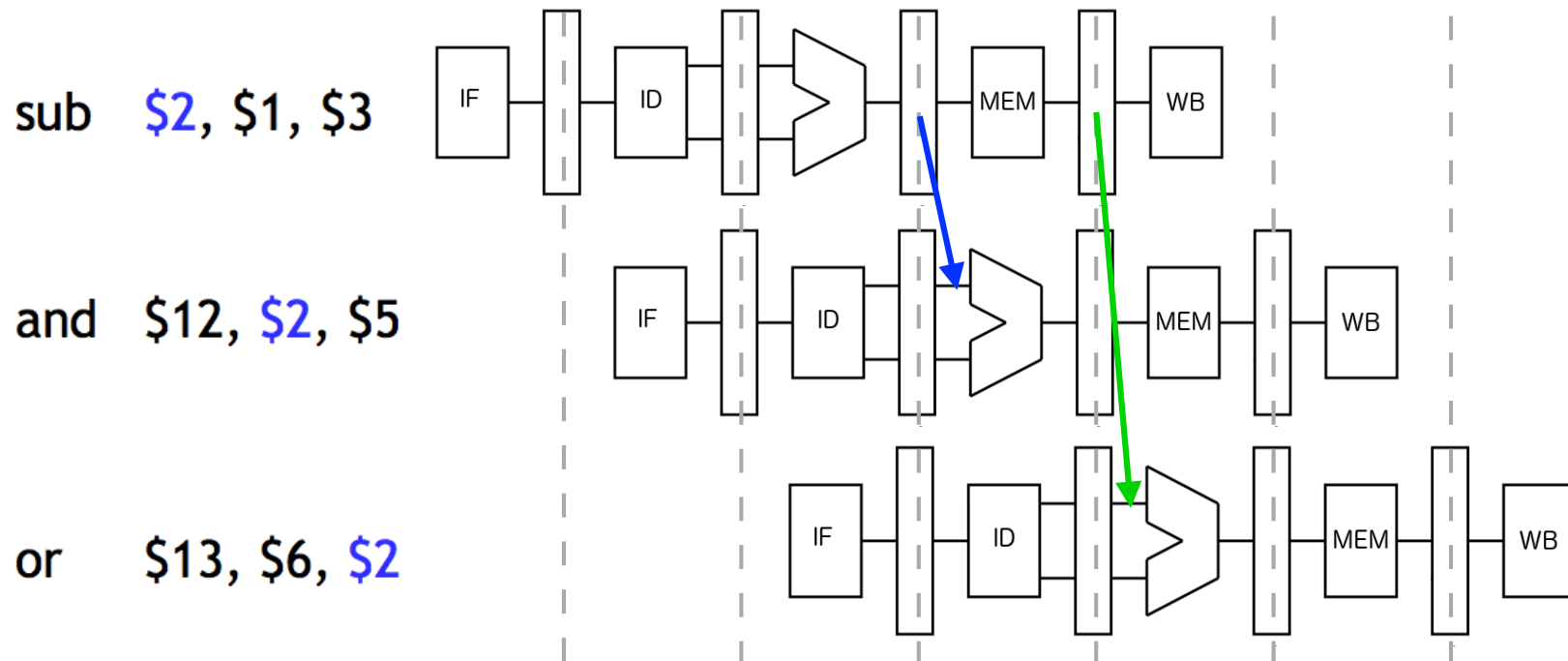
- Di norma il risultato della ALU viene passato alla fase MEM ed alla WB attraverso i registri della pipeline (quello mostrato è un datapath semplificato).



Forwarding

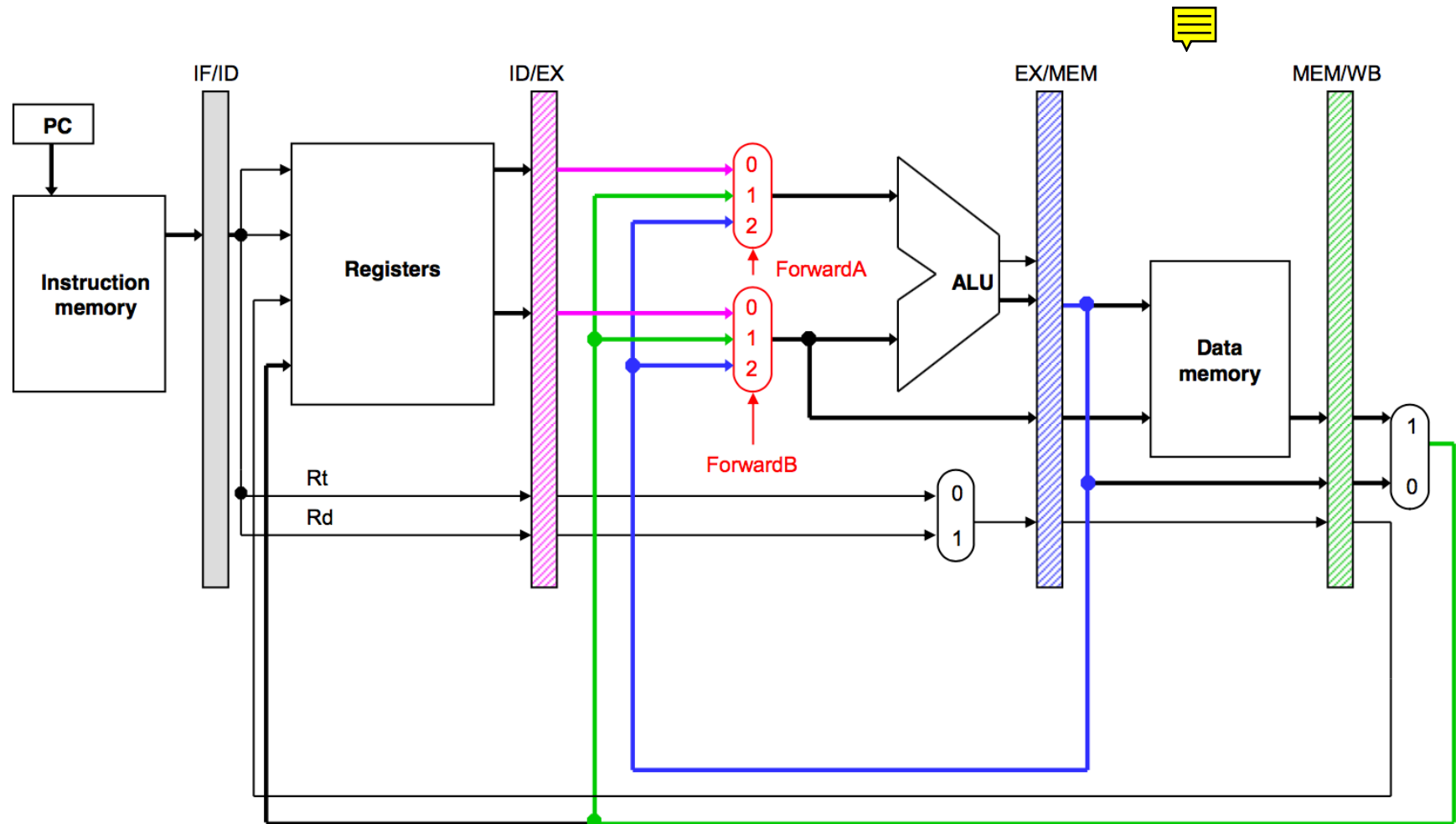


- Visto che il risultato è già presente nei registri della pipeline, possiamo “inoltrare” (**forward**) quel valore alle istruzioni successive:
- al ciclo 4 la AND può ottenere \$2 dai registri **EX/MEM**
- al ciclo 5 la OR può ottenere \$2 dai registri **MEM/WB**



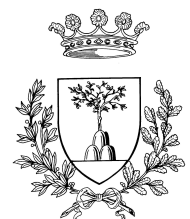
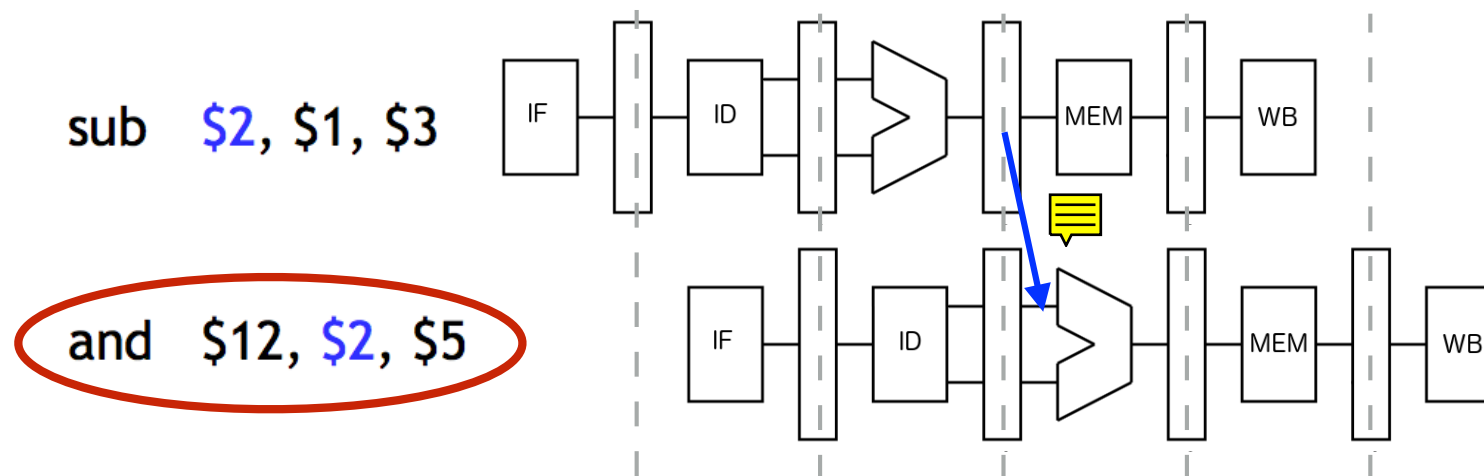
Forwarding

- Gli input della ALU vengono scelti da due nuovi multiplexer, con segnali di controllo ForwardA e ForwardB (quello mostrato è un datapath semplificato).



Criticità EX/MEM

- Come può l'hardware capire quando ci sono delle criticità che possono essere risolte con il forwarding?
- Abbiamo criticità EX/MEM tra due istruzioni quando:
 - l'istruzione in fase MEM deve scrivere su un registro
 - il registro è anche sorgente della ALU per l'istruzione correntemente in fase EX



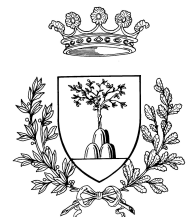
Criticità EX/MEM

- Il controllo della criticità EX/MEM può essere descritto con il seguente pseudo-codice:

```
if (  
    EX/MEM.RegWrite = 1 and  
    EX/MEM.RegisterRd = ID/EX.RegisterRs  
) then ForwardA = 2  
  
if (  
    EX/MEM.RegWrite = 1 and  
    EX/MEM.RegisterRd = ID/EX.RegisterRt  
) then ForwardB = 2
```

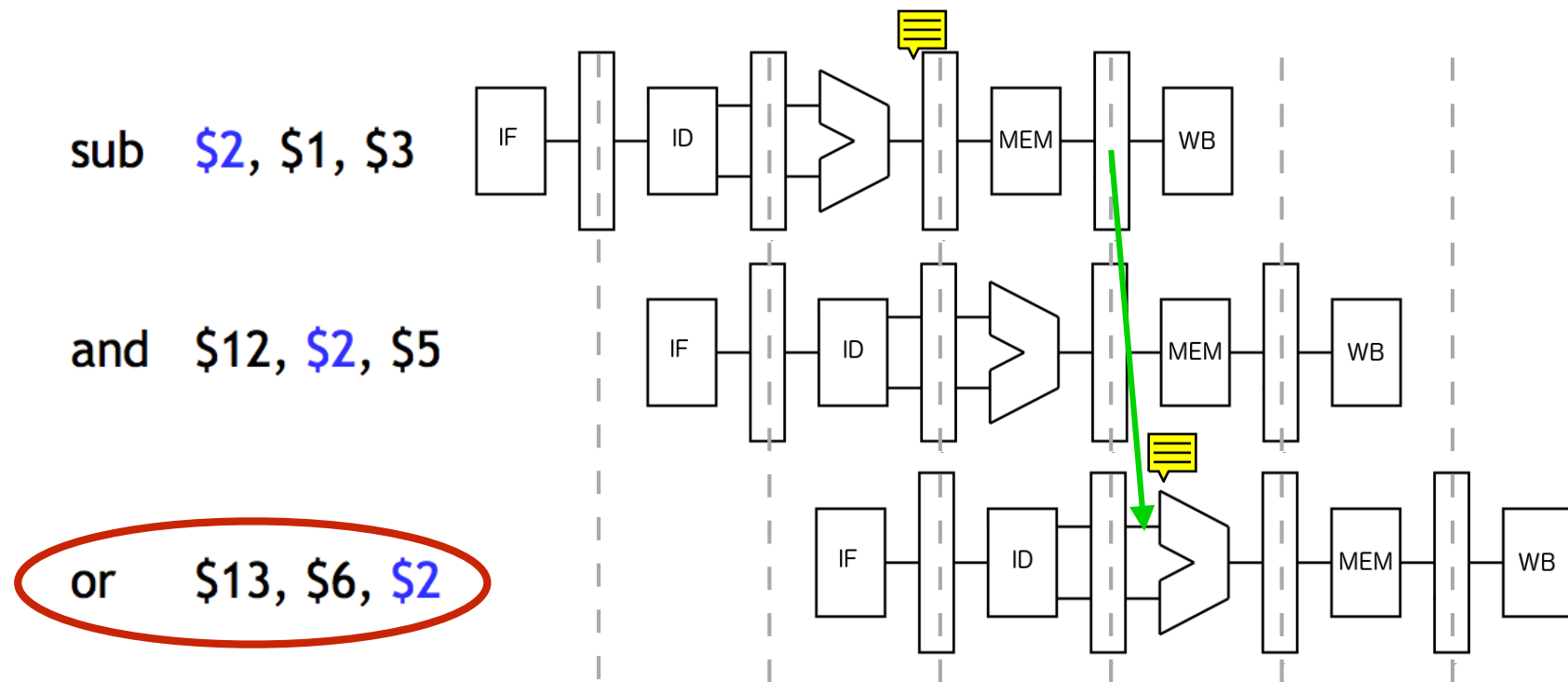
Diagram annotations:

- segnale di controllo per scrittura sul RF (green arrow pointing to `EX/MEM.RegWrite = 1`)
- 1° registro sorgente (red arrow pointing to `ID/EX.RegisterRs`)
- segnale di controllo 1° operando (purple arrow pointing to `ForwardA = 2`)
- 2° registro sorgente (red arrow pointing to `ID/EX.RegisterRt`)
- segnale di controllo 2° operando (purple arrow pointing to `ForwardB = 2`)



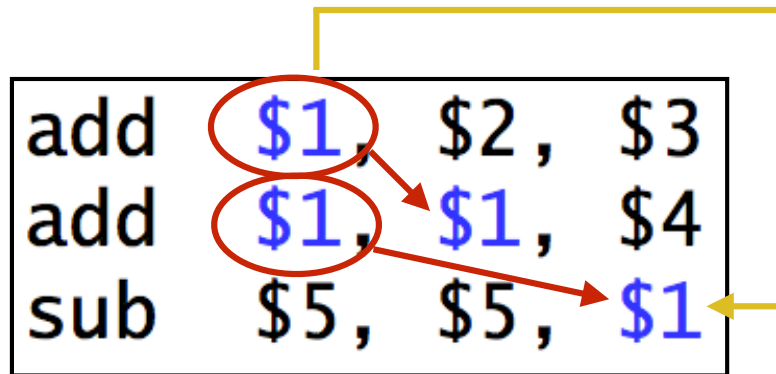
Criticità MEM/WB

- Abbiamo criticità MEM/WB tra due istruzioni quando:
 - l'istruzione in fase WB deve scrivere su un registro
 - il registro è anche sorgente della ALU per l'istruzione correntemente in fase EX



Criticità MEM/WB

- C'è però un ulteriore problema:
 - pensate ad istruzioni che leggono e scrivono lo stesso registro



- Il registro \$1 viene scritto da entrambe le istruzioni ADD, ma solo la solo il risultato più recente dovrebbe essere utilizzato dalla SUB.
- Non serve il forward da MEM/WB, anche se in teoria c'è una dipendenza!
- è necessario controllare che non ci sia criticità EX/MEM

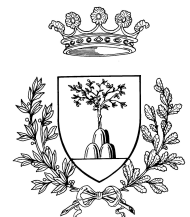
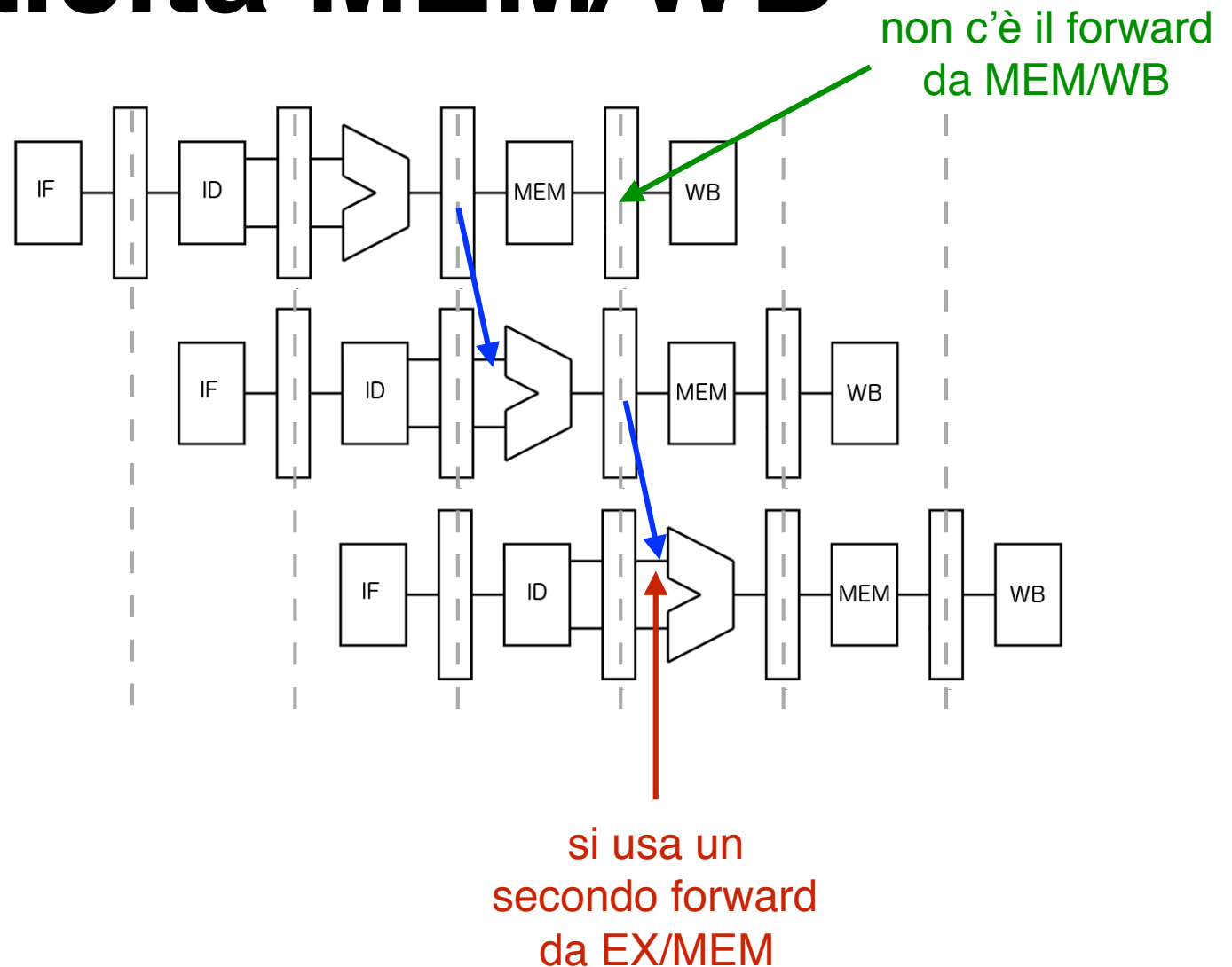


Criticità MEM/WB

add \$1, \$2, \$3

add \$1, \$1, \$4

sub \$5, \$5, \$1



Criticità MEM/WB

- Il controllo della criticità MEM/WB può essere descritto come:

if (
MEM/WB.RegWrite = 1 and
MEM/WB.RegisterRd = ID/EX.RegisterRs and
(EX/MEM.RegisterRd \neq ID/EX.RegisterRs or
EX/MEM.RegWrite = 0)
) then ForwardA = 1

if (
MEM/WB.RegWrite = 1 and
MEM/WB.RegisterRd = ID/EX.RegisterRt and
(EX/MEM.RegisterRd \neq ID/EX.RegisterRt or
EX/MEM.RegWrite = 0)
) then ForwardB = 1

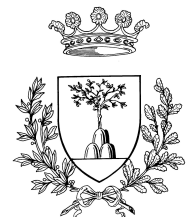
segnale di controllo per scrittura sul RF

controllo che non ci sia criticità EX/MEM

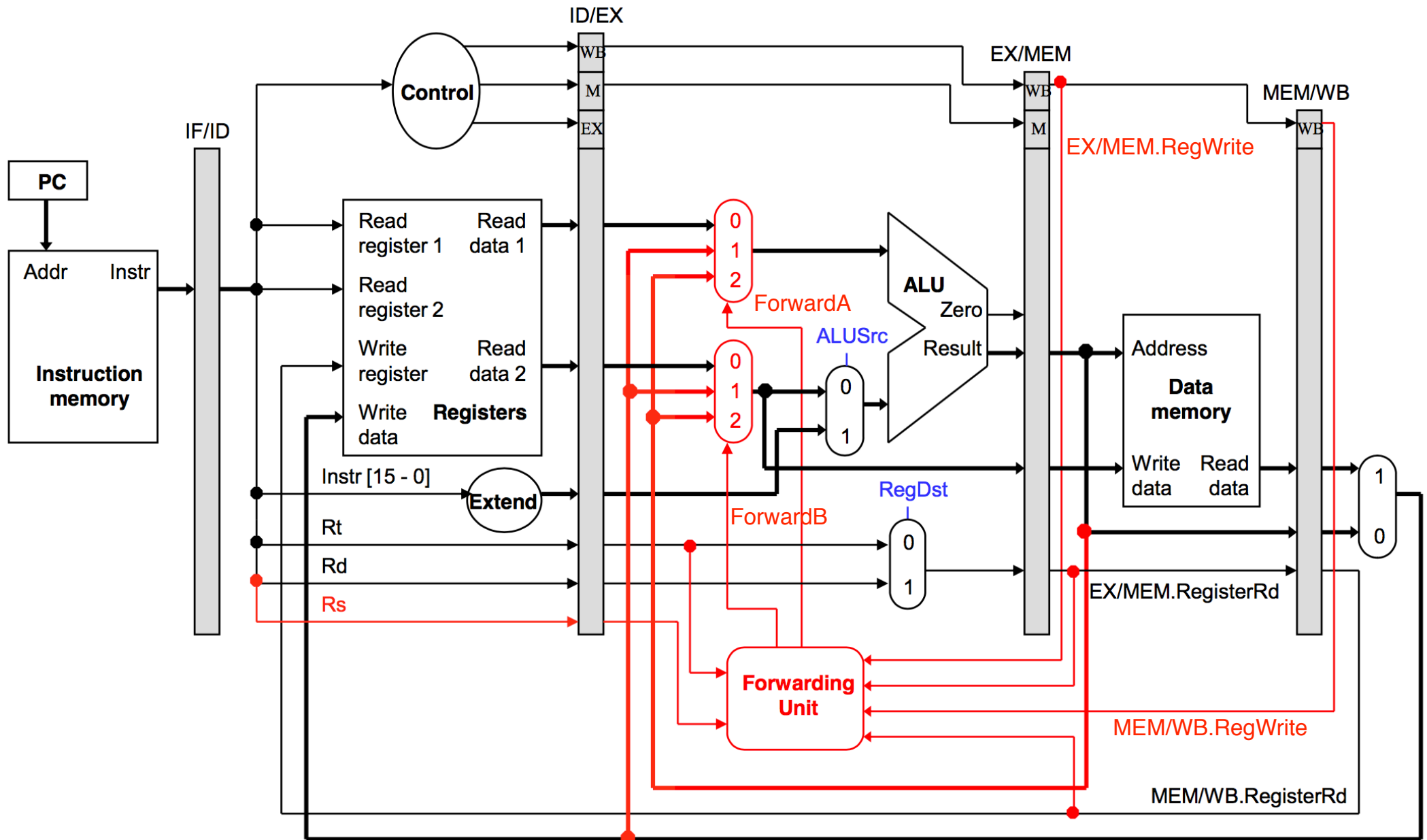
segnale di controllo 1° operando

controllo che non ci sia criticità EX/MEM

segnale di controllo 2° operando



Controllo del forwarding





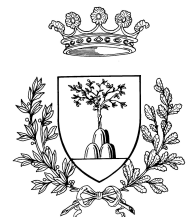
Controllo del forwarding



- L'Unità di Forwarding ha diversi segnali di controllo ed input:

ID/EX.RegisterRs	EX/MEM.RegisterRd	MEM/WB.RegisterRd
ID/EX.RegisterRt	EX/MEM.RegWrite	MEM/WB.RegWrite

- Gli output invece sono i due controlli dei multiplexer: **ForwardA** e **ForwardB**:
 - questi output sono generati a partire dagli input e seguendo le equazioni viste nelle precedenti slides



Esempio

sub	\$2, \$1, \$3
and	\$12, \$2, \$5
or	\$13, \$6, \$2
add	\$14, \$2, \$2
sw	\$15, 100(\$2)

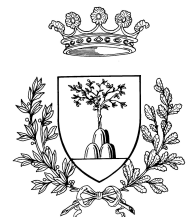
- Facciamo le stesse assunzioni della scorsa lezione:
 - ogni registro contiene come valore il suo ID + 100
 - dopo la prima istruzione, \$2 dovrebbe valere -2 (101 - 103)
 - le altre istruzioni dovrebbero usare -2 come valore per il registro \$2



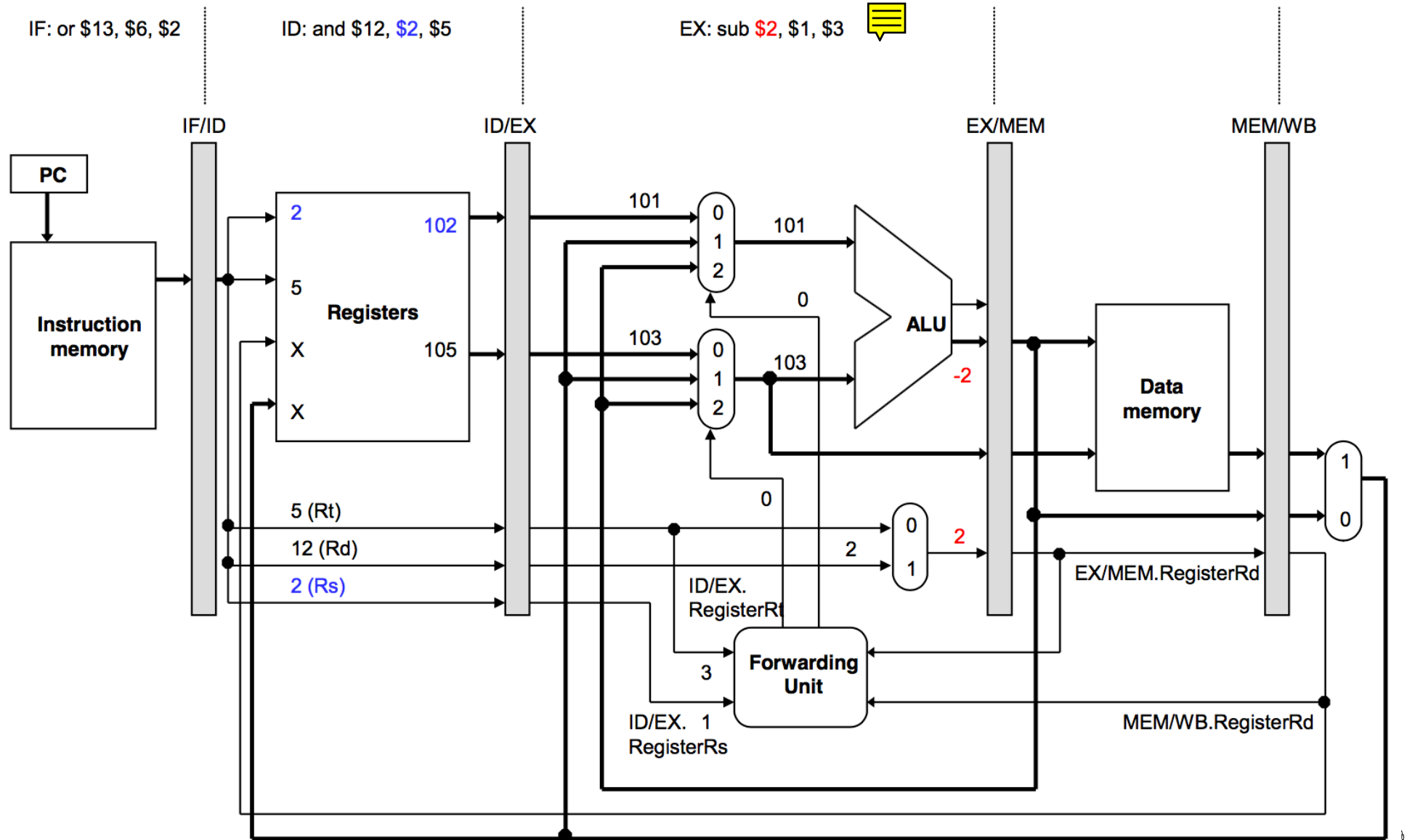
Esempio

sub	\$2, \$1, \$3
and	\$12, \$2, \$5
or	\$13, \$6, \$2
add	\$14, \$2, \$2
sw	\$15, 100(\$2)

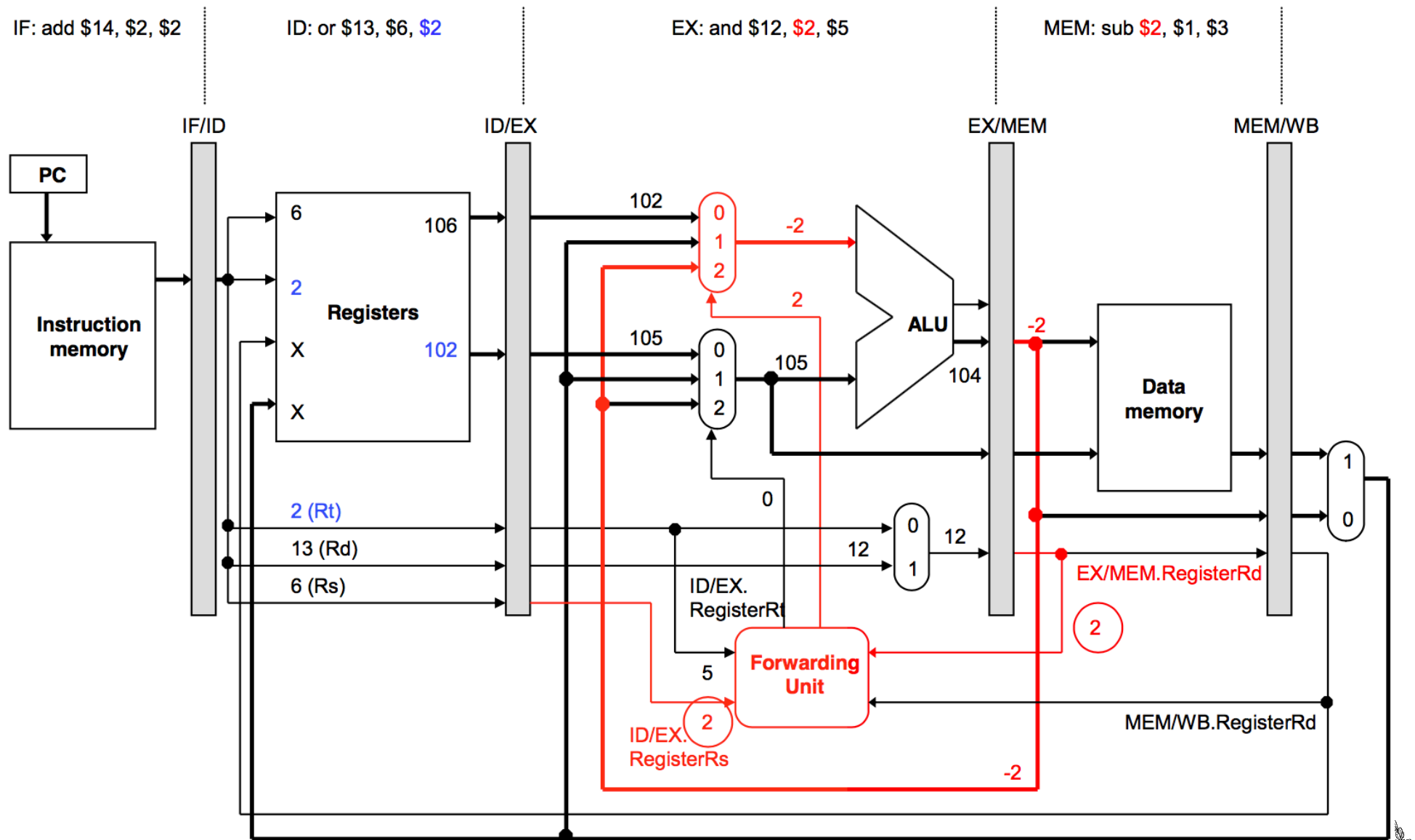
- Cercheremo di semplificare l'esempio:
- saltiamo i primi due cicli (non c'è criticità) e ci fermiamo al quinto
- non mostriamo nella Forwarding Unit gli input:
EX/MEM.RegWrite
MEM/WB.RegWrite
assumiamo che valgano sempre 1 in quanto tutte le istruzioni (eccezione per la sw) richiedono di scrivere sul Register File



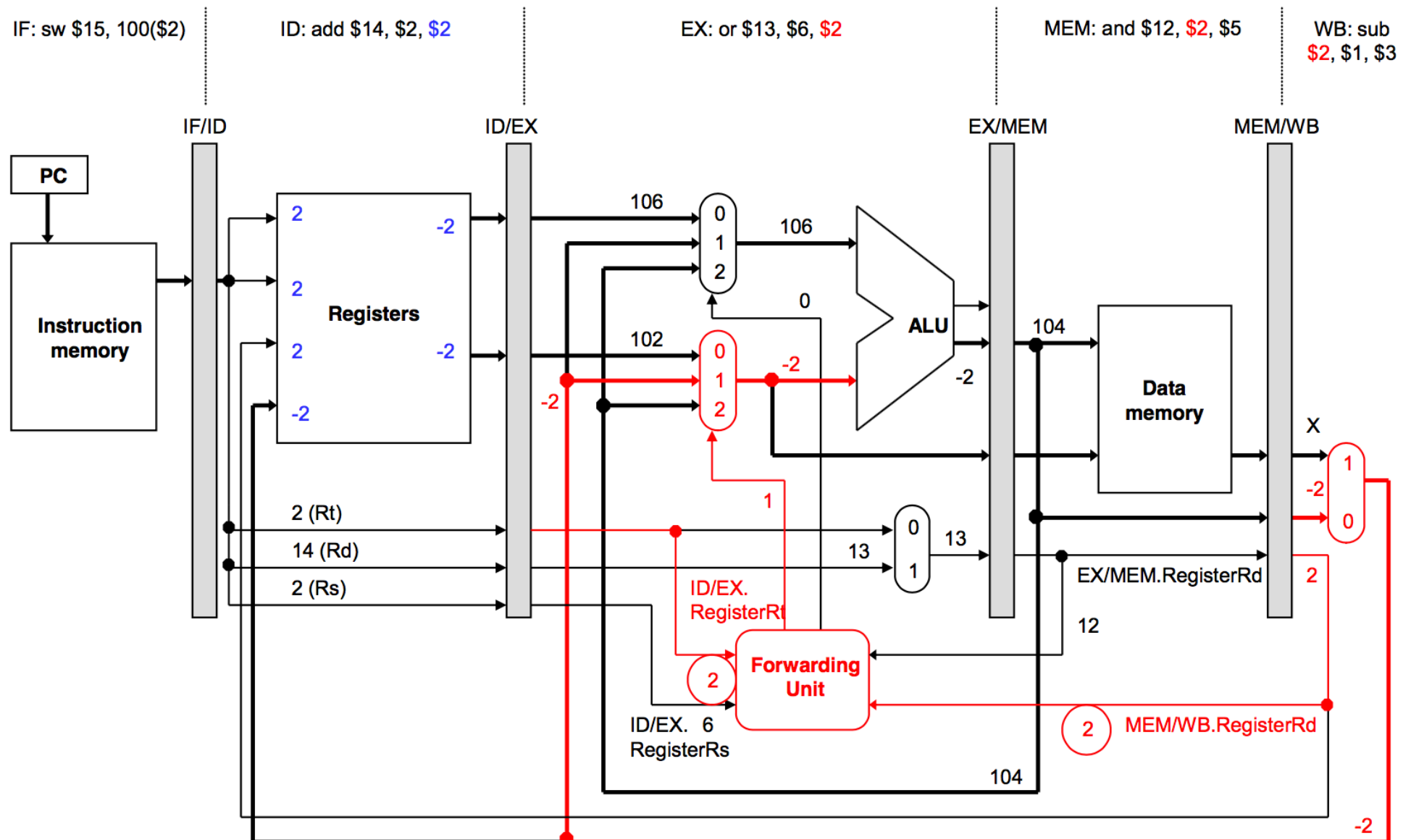
Ciclo di clock 3



Ciclo di clock 4



Ciclo di clock 5



Esempio soluzione

- La prima criticità accade durante il ciclo 4:
 - la Forwarding Unit rileva che il primo registro sorgente della ALU per la AND è anche il registro destinazione della SUB
 - il valore corretto viene inoltrato dal registro EX/MEM, sovrascrivendo il vecchio valore ancora nel registro
- Una seconda criticità accade durante il ciclo 5:
 - il secondo registro sorgente della ALU per la OR è anche il registro destinazione della SUB
 - questa volta viene inoltrato il valore corretto dal registro MEM/WB
- Non ci sono altre criticità in quanto la ADD legge il valore corretto dal Register File durante il ciclo 6.

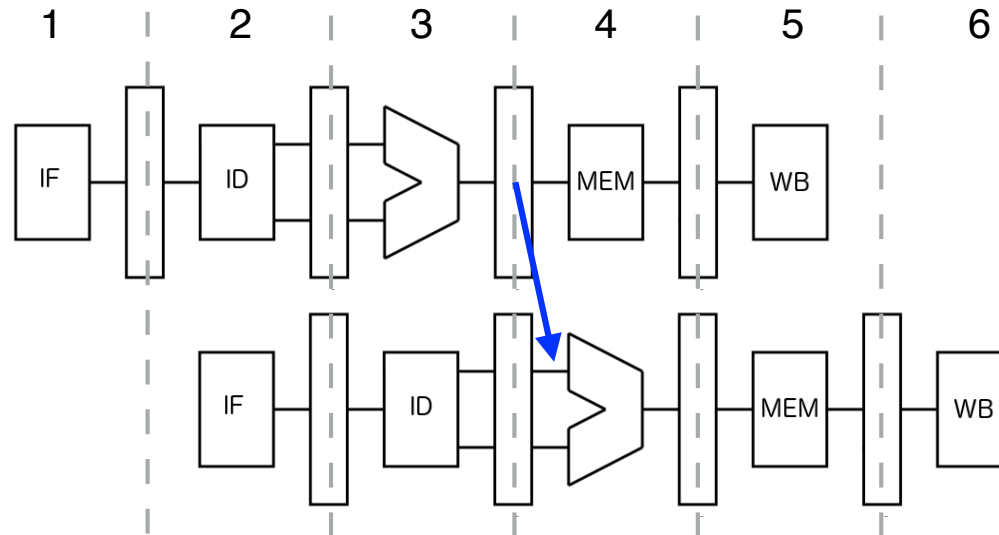


Criticità e store word

Esempio simile
a quelli già visti

add \$1, \$2, \$3

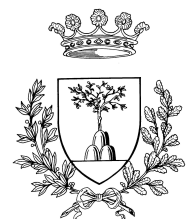
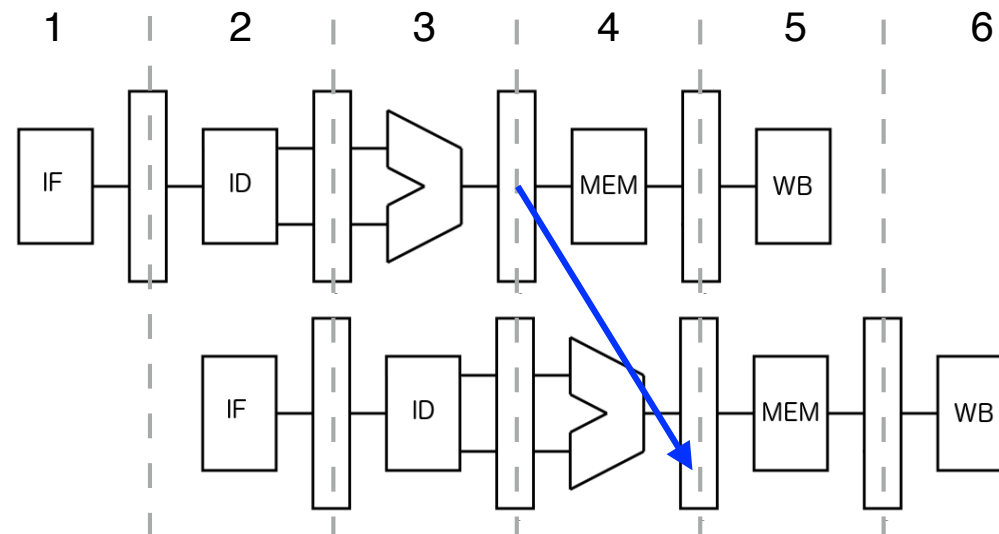
sw \$4, 0(\$1)



In questo caso il valore
calcolato deve essere
portato ai registri di pipeline

add \$1, \$2, \$3

sw \$1, 0(\$4)



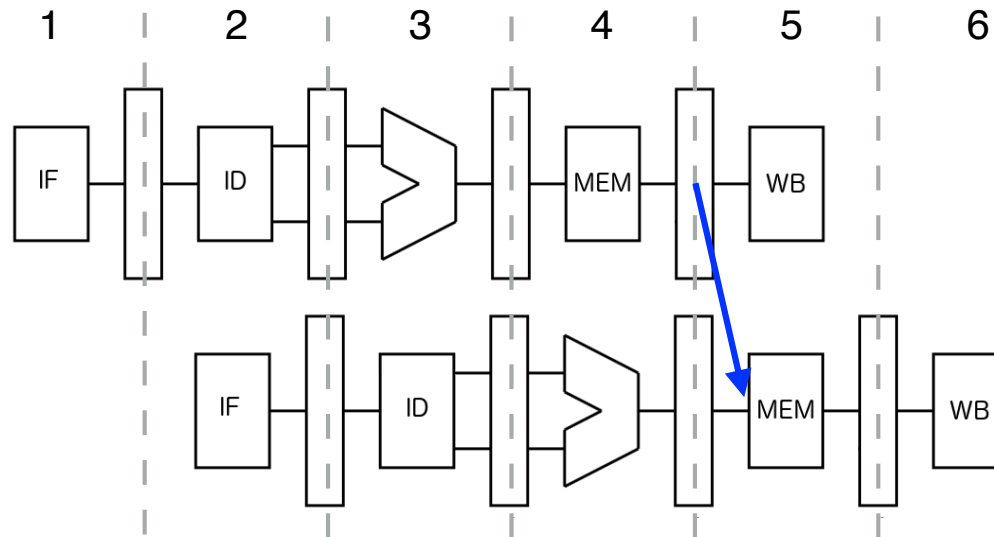
Criticità e store word

- Vediamo ora un caso più complesso:



lw \$1, 0(\$2)

sw \$1, 0(\$4)



- Sono necessarie ulteriori logiche di controllo per la Forwarding Unit (seguendo ragionamenti simili a quelli visti nelle precedenti slides).



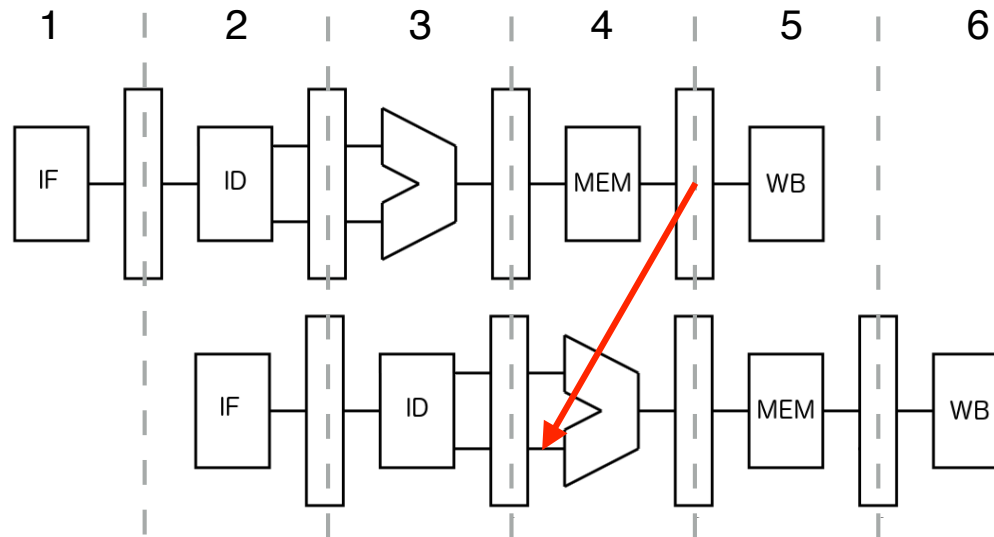
Criticità e load word

- La tecnica del forwarding è potente ma non risolve tutte le criticità sui dati:

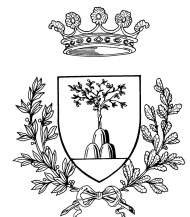


lw \$2, 20(\$3)

and \$12, \$2, \$5

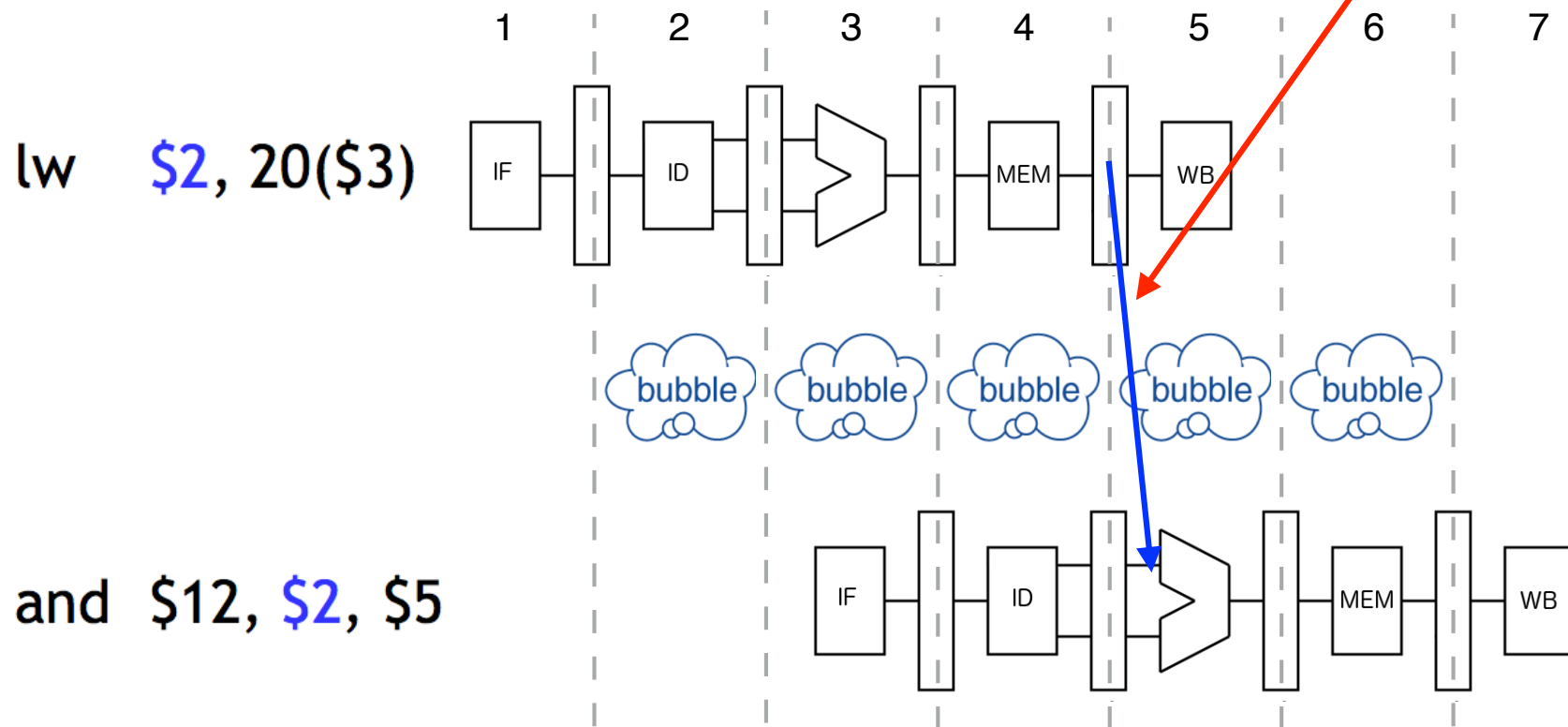


- Il valore di \$2 viene ottenuto al ciclo 4, ma nello stesso ciclo dovrebbe essere disponibile in ingresso alla ALU per la AND.
- La tecnica di forwarding permette di inoltrare dei valori, ma non permette di andare **indietro nel tempo**!



Stallo + forwarding

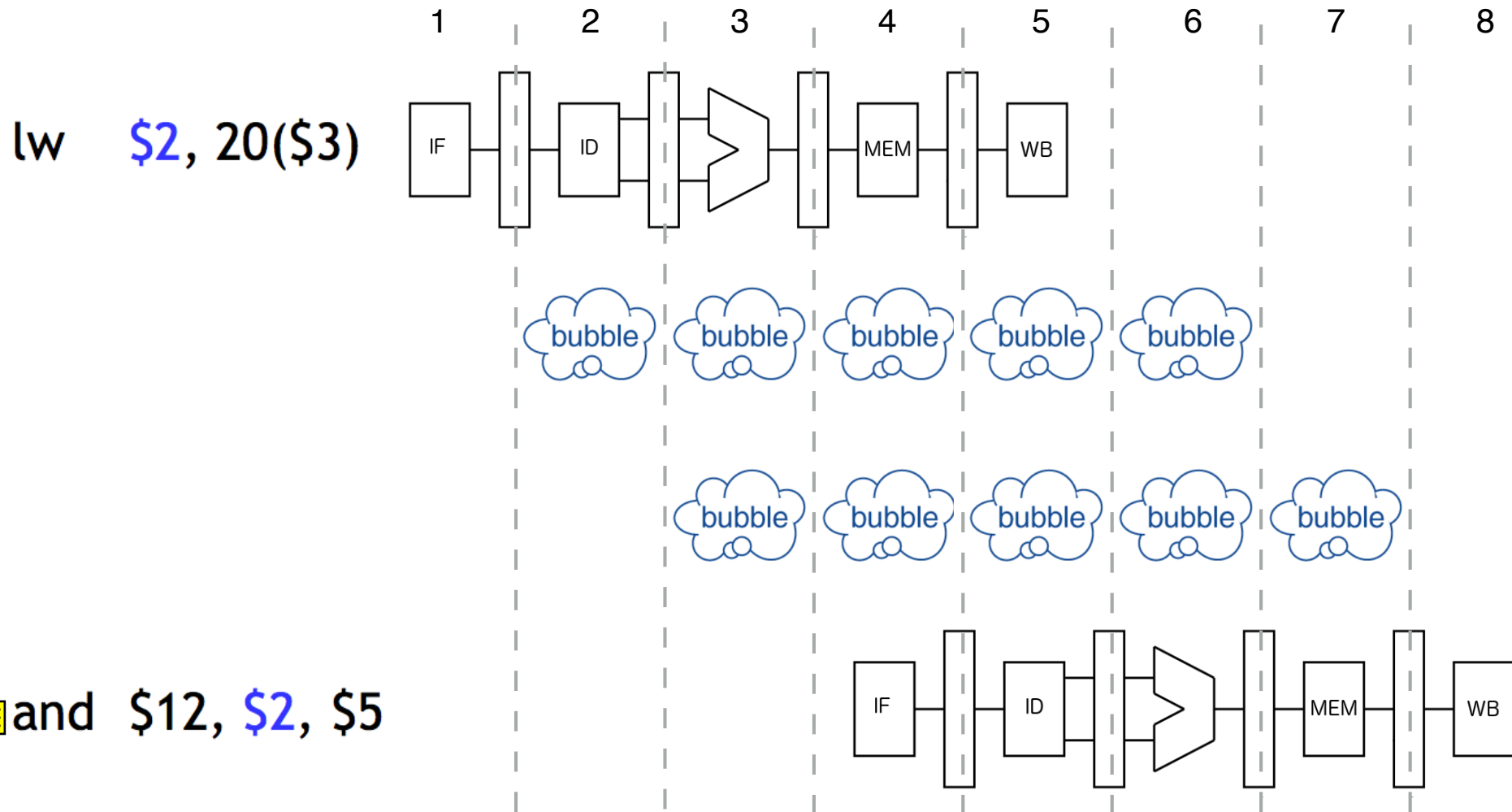
Mantengo l'uso
del forwarding



- Possiamo introdurre uno **stallo** (anche chiamato bolla) per ritardare l'esecuzione delle istruzioni successive di un ciclo.



Stallo senza forwarding

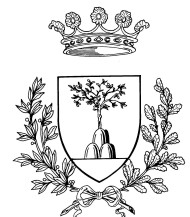


Con due stalli non c'è bisogno del forwarding (\$2 viene aggiornato nella WB della lw)

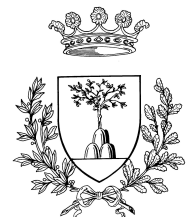
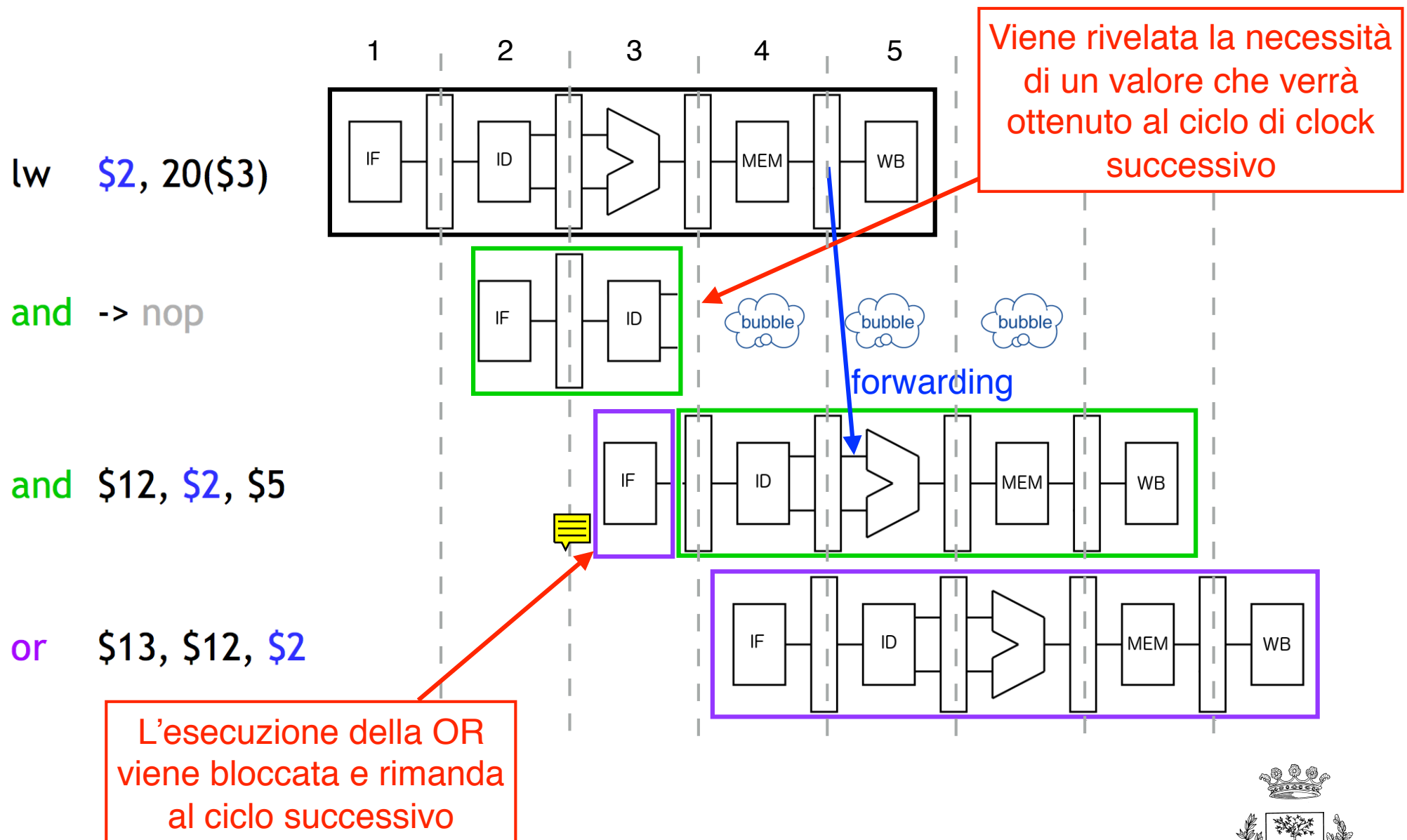


Stalli nella pratica

- Fino ad ora abbiamo rappresentato gli stalli come se venisse eseguita un'operazione “nulla” (nop) che ritarda l'esecuzione delle istruzioni successive.
- Nella realtà la logica di controllo del datapath individua una condizione di stallo in tempo reale:
 - le istruzioni successive potrebbero già aver iniziato la loro esecuzione
- Gli stalli vengono individuati nella fase ID:
 - ovvero quando si devono leggere i registri dal RF
 - per introdurre le “bolle” si interviene sui segnali di controllo nei registri ID/EX



Stalli nella pratica



Ordine delle istruzioni

- Gli stalli possono a volte essere evitati riordinando le istruzioni:
- invece di non fare nulla per un ciclo posso eseguire una istruzione che non ha criticità

Si utilizza il forwarding
(quindi ad esempio la
prima lw non ha criticità)

```
a = b + e;  
c = b + f;
```

stallo →

stallo →

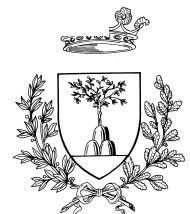
```
lw $t1, 0($t0)  
lw $t2, 4($t0)  
add $t3, $t1, $t2  
sw $t3, 12($t0)  
lw $t4, 8($t0)  
add $t5, $t1, $t4  
sw $t5, 16($t0)
```

13 cicli



```
lw $t1, 0($t0)  
lw $t2, 4($t0)  
lw $t4, 8($t0)  
add $t3, $t1, $t2  
sw $t3, 12($t0)  
add $t5, $t1, $t4  
sw $t5, 16($t0)
```

11 cicli

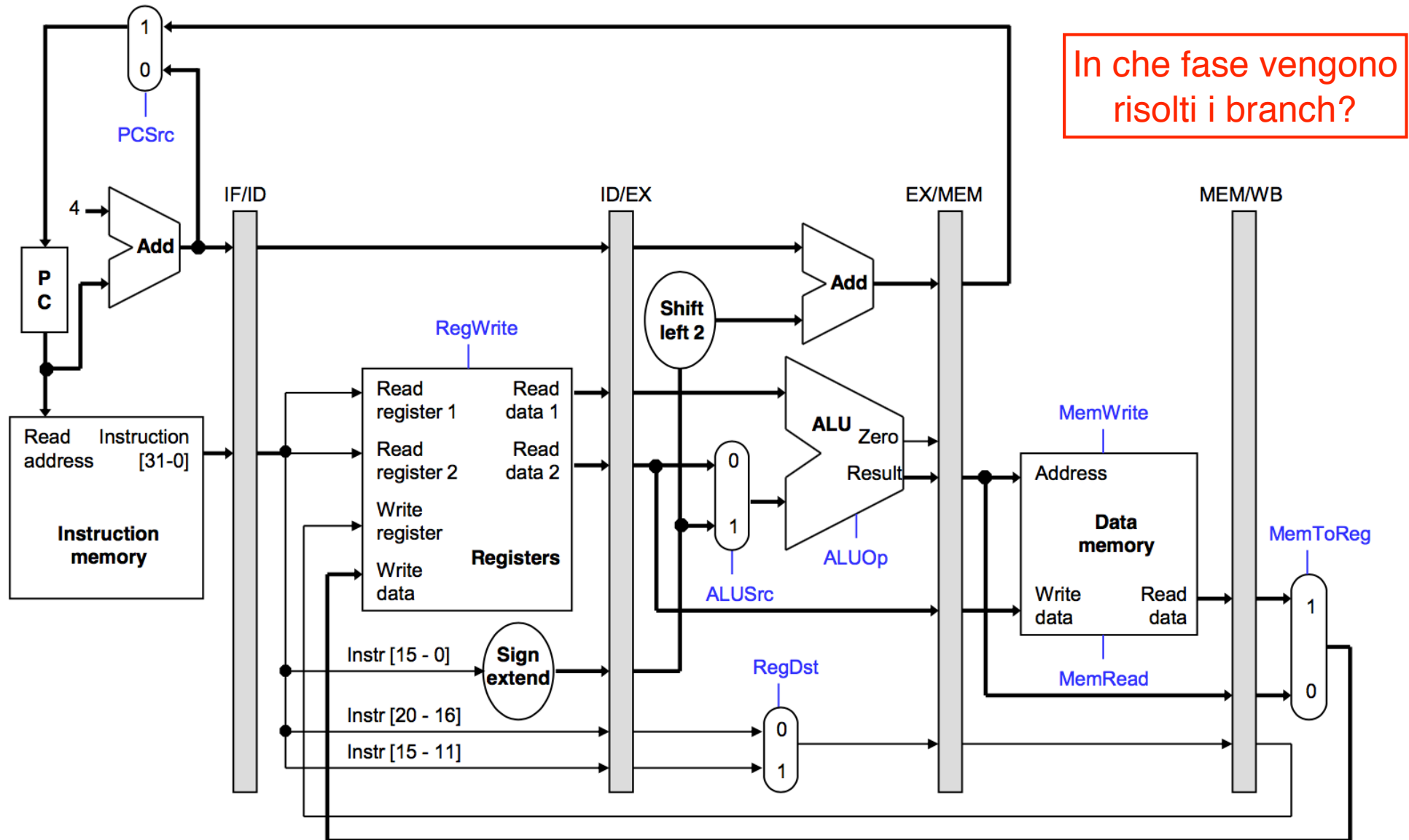


Ordine delle istruzioni

- Il riordinamento può essere fatto:
 - dal compilatore → deve conoscere l'architettura per cui genera codice
 - dal processore → in genere associato a tecniche predittive-speculative (vedi oltre)
- Il riordinamento (ovviamente) non deve modificare il significato di un programma.



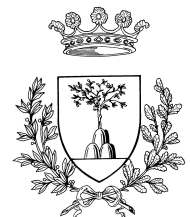
Branch nella pipeline



Criticità sul controllo

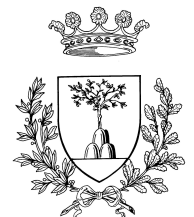
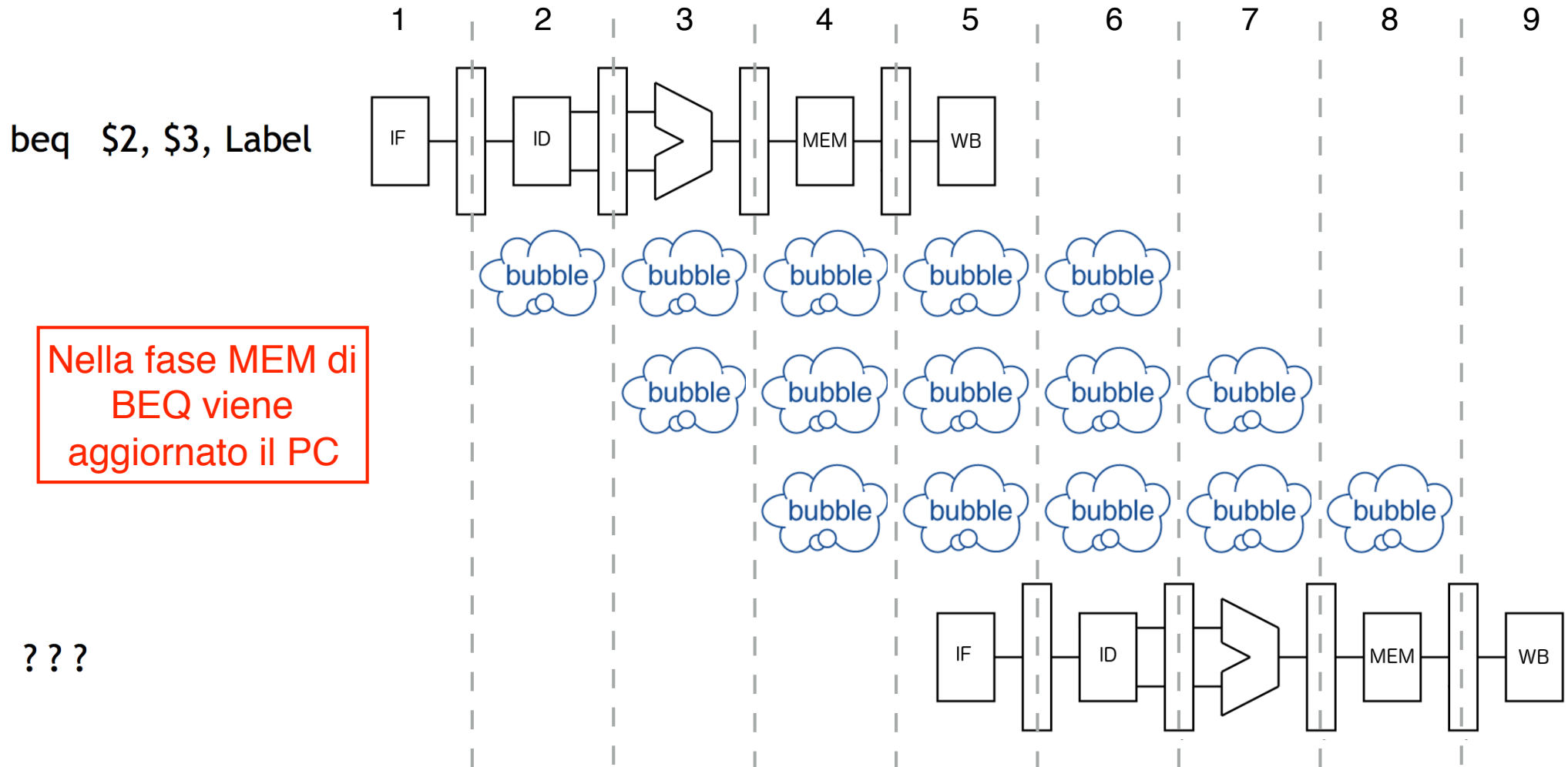


- La gran parte del lavoro per un'istruzione branch viene fatta nella fase EX:
 - viene calcolato il target address
 - vengono confrontati i valori dei due registri sorgente e viene settato di conseguenza il segnale zero
- Quindi la decisione del branch non può essere presa fino alla fase MEM:
 - ma ad ogni ciclo di clock inizia l'esecuzione di una nuova istruzione
 - questo porta alla terza tipologia di criticità: **criticità sul controllo.**



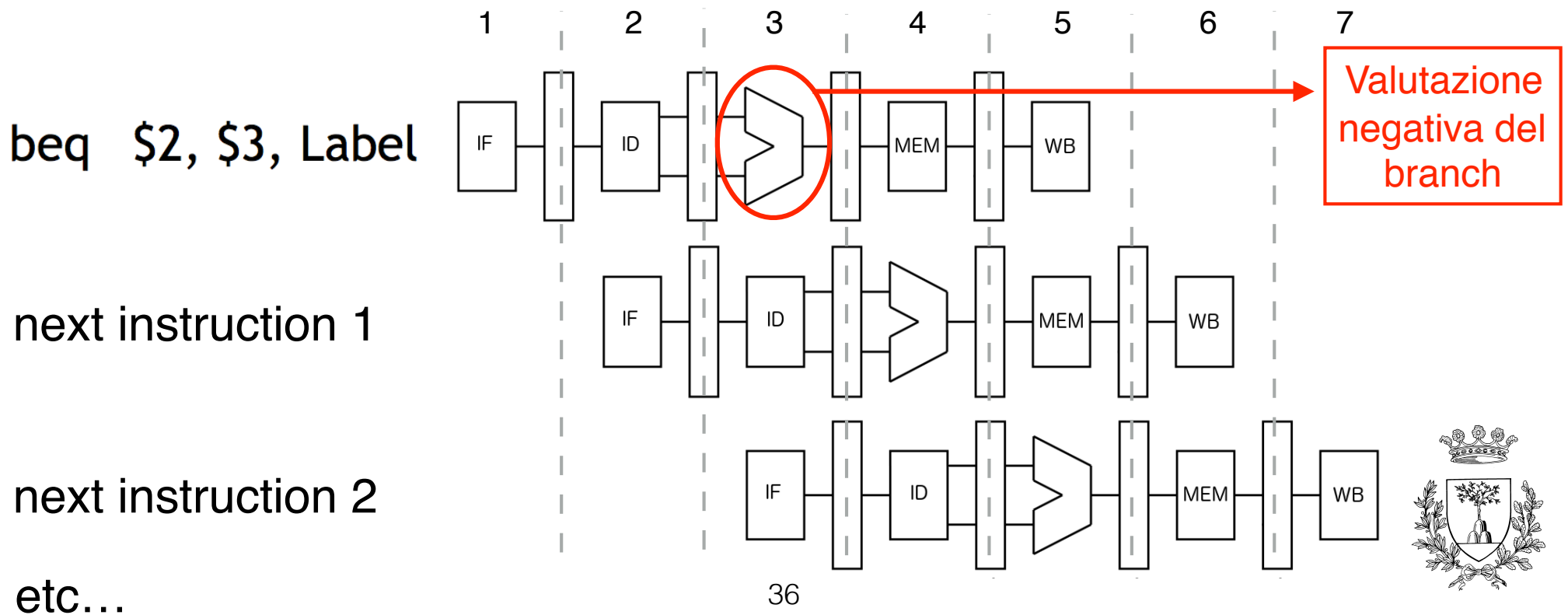


Soluzione semplice: stalli

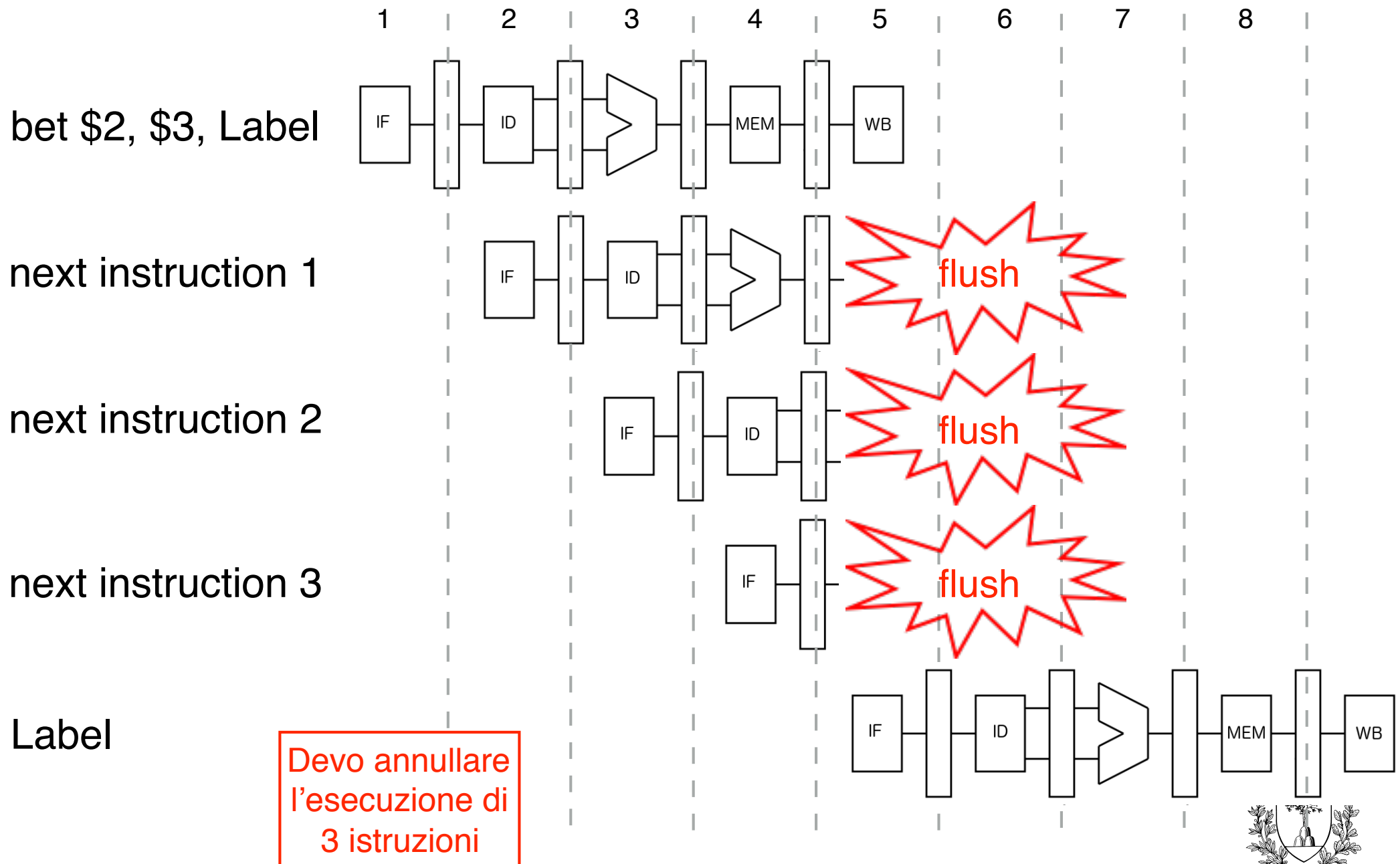


Altra soluzione: predizione

- Un altro approccio è quello di provare a prevedere l'esecuzione del programma.
- A livello hardware è più semplice prevedere che il branch fallisca:
 - incremento il PC di 4 ed eseguo l'istruzione successiva
- Se abbiamo indovinato possiamo evitare gli stalli.

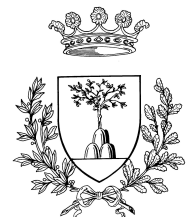


Predizione sbagliata



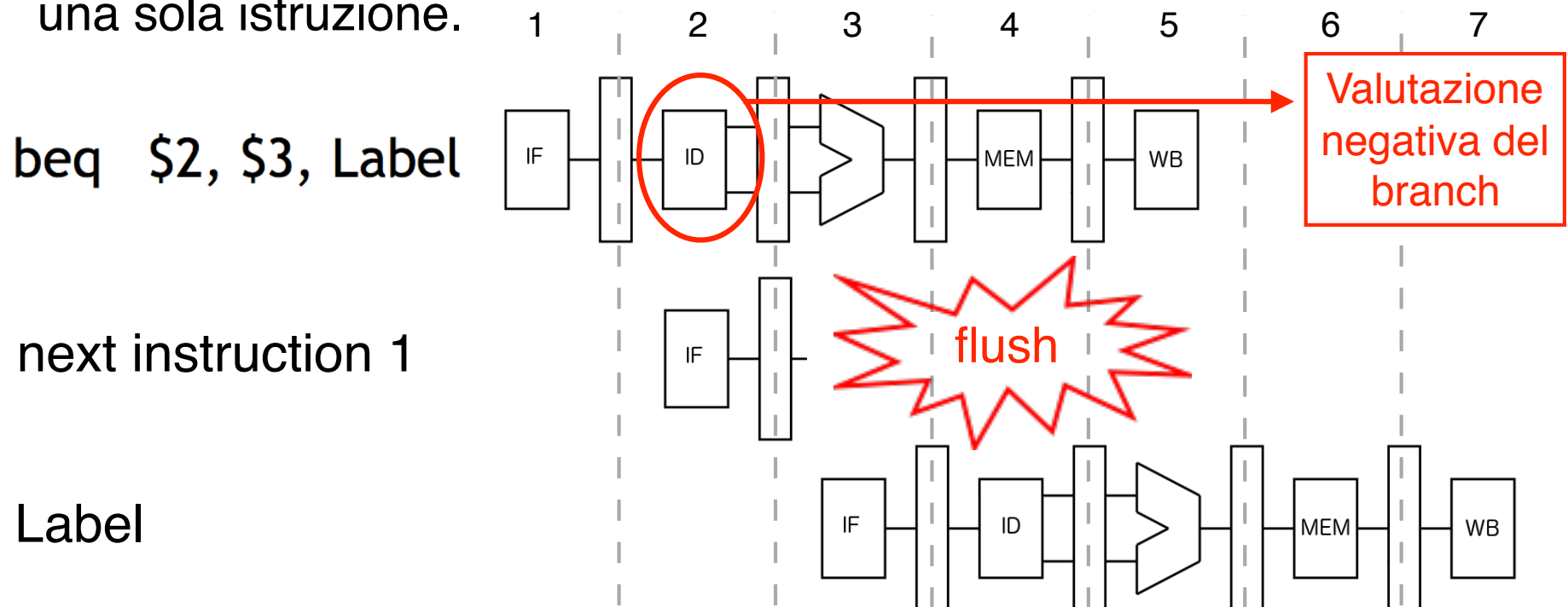
Predizione: performance

- Nel complesso, la predizione (branch prediction) è vantaggiosa:
 - sbagliare la previsione significa perdere tre cicli di clock
 - ma senza predizione avremo **sempre** la perdita di tre cicli di clock
- Tutte le moderne CPU usano la branch prediction:
 - predizioni accurate sono fondamentali per le performance
 - la maggior parte delle CPU fanno previsioni dinamiche mantenendo statistiche a runtime sull'esecuzione o meno dei branch



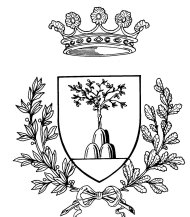
Predizione nella pratica

- Prima di tutto possiamo introdurre una ottimizzazione:
- aggiungiamo un piccolo componente hw nella fase ID
- questo ci permette di valutare l'uguaglianza di due registri letti dal Register File
- Così facendo in caso di salto dobbiamo annullare l'esecuzione di una sola istruzione.

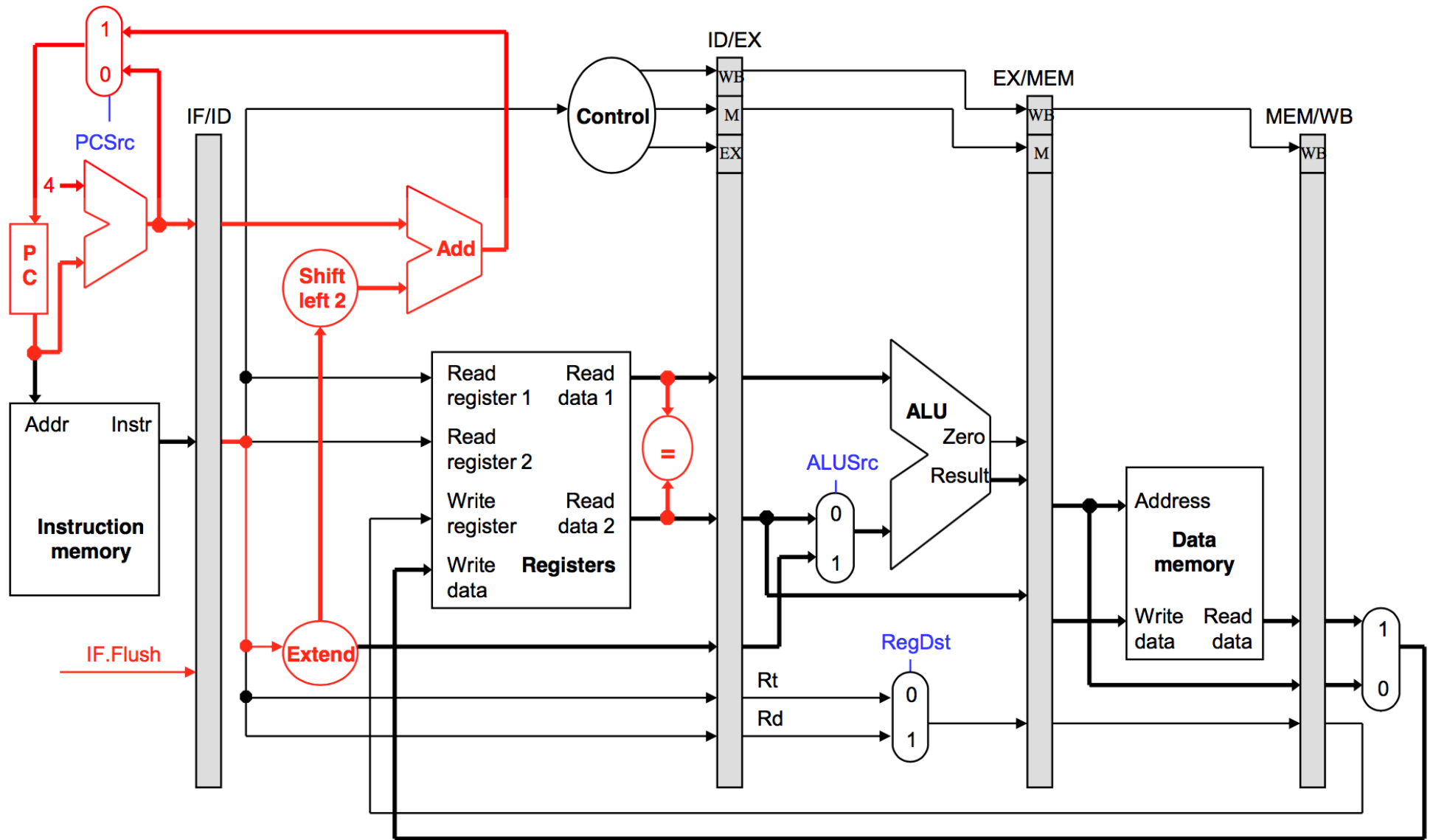


Predizione nella pratica

- Dobbiamo annullare (flush) una istruzione (nella fase IF) se l'istruzione precedente è un BEQ e i due suoi registri sorgente sono uguali.
- Possiamo annullare una istruzione nella fase IF sostituendola con una istruzione “nulla” (**nop**) nei registri IF/ID:
 - MIPS usa l'istruzione `sl $0, $0, 0` come istruzione nop
 - a rappresentazione binaria di questa istruzione è composta da 32 bit a zero
- Fare il flush equivale ad introdurre uno stallo nella pipeline.
- Il segnale di controllo IF.Flush mostrato nella prossima slide implementa questo concetto (ma non mostro tutti i dettagli).



Predizione nella pratica





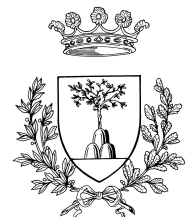
Instruction Level Parallelism

- Abbiamo visto che il pipelining sovrappone parzialmente l'esecuzione delle istruzioni:
 - sfrutta il potenziale parallelismo insito nelle istruzioni stesse
- Questa forma di parallelismo prende il nome di “Parallelismo a livello di istruzione” o “Instruction Level Parallelism” (solitamente abbreviato nell'acronimo ILP).
- L'ILP è limitato dalle possibili criticità strutturali, sui dati e sul controllo.
- Al netto delle varie criticità ci sono due modi di incrementare l'ILP:
 - **aumentare la frequenza della CPU**
 - **eseguire più istruzioni in parallelo**



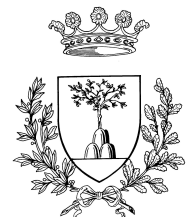
Aumento della frequenza

- Aumentare la frequenza della CPU significa ridurre la durata di un ciclo di clock:
- un ciclo di clock deve comunque essere in grado di eseguire la fase più lunga della pipeline 
- si potrebbe frammentare la pipeline in un numero maggiore di fasi (ma di durata minore)
- Frequenza della CPU e profondità della pipeline sono due concetti strettamente legati tra loro.
- Non è possibile sfruttare queste tecniche indefinitamente: 
 - maggior complessità della pipeline, maggior complessità della logica di controllo
 - componenti dei circuiti, consumo, calore, etc...



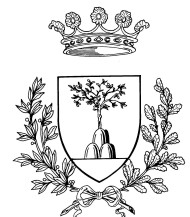
Istruzioni parallele

- I moderni processori consentono di eseguire più istruzioni in parallelo (tecnica “**multiple issue**”).
- Questo tipo di parallelismo è detto “**intrinseco**”:
 - non è noto al programmatore
 - non è il multithreading!
- Il processore deve sapere quando le istruzioni sono indipendenti, ovvero quando nessuna istruzione necessita del risultato di un'altra istruzione eseguita in parallelo.
- Il multiple issue richiede un datapath più ampio:
 - replicazione di unità funzionali per eseguire più istruzioni in parallelo




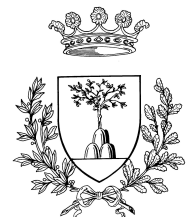
Istruzioni parallele

- Deve essere possibile:
 - prelevare dalla memoria più istruzioni in parallelo
 - leggere e scrivere in memoria più registri per ciclo di clock
 - leggere e scrivere dal Register File più operandi per ciclo di clock
- In un datapath con pipeline senza multiple issue in condizioni ottimali è possibile eseguire una istruzione per ciclo di clock (CPI = 1).
- Con il multiple issue è possibile (**in teoria**) avere $CPI < 1$, in realtà a causa di criticità e latenze è molto difficile ottenere questo risultato.
- Questo tipo di processori sono chiamati **superscalari**.



Multiple issue statico

- Nel multiple issue statico è il compilatore (software) che si occupa di raggruppare le istruzioni in “issue packets”.
- “Issue packet”:
 - gruppo di istruzioni che possono essere eseguite in parallelo
 - definito dall'hardware
- Il compilatore deve riconoscere eventuali criticità durante il raggruppamento delle istruzioni:
 - può eventualmente riordinare le istruzioni
- Potete pensare ad un issue packet come ad una istruzione molto lunga che specifica differenti operazioni concorrenti (Very Long Instruction Word - VLIW). 



MIPS: 2-issue packets

- Un esempio di multiple issue statico in MIPS è la possibilità di eseguire parallelamente due istruzioni:
 - 1 ALU o branch
 - 1 load o store
- Allineamento a 64 bit:
 - 32 higher bit → ALU/branch
 - 32 lower bit → load/store
 - in caso di istruzione mancante si mette un **nop**





MIPS: 2-issue packets

Clock

1

2

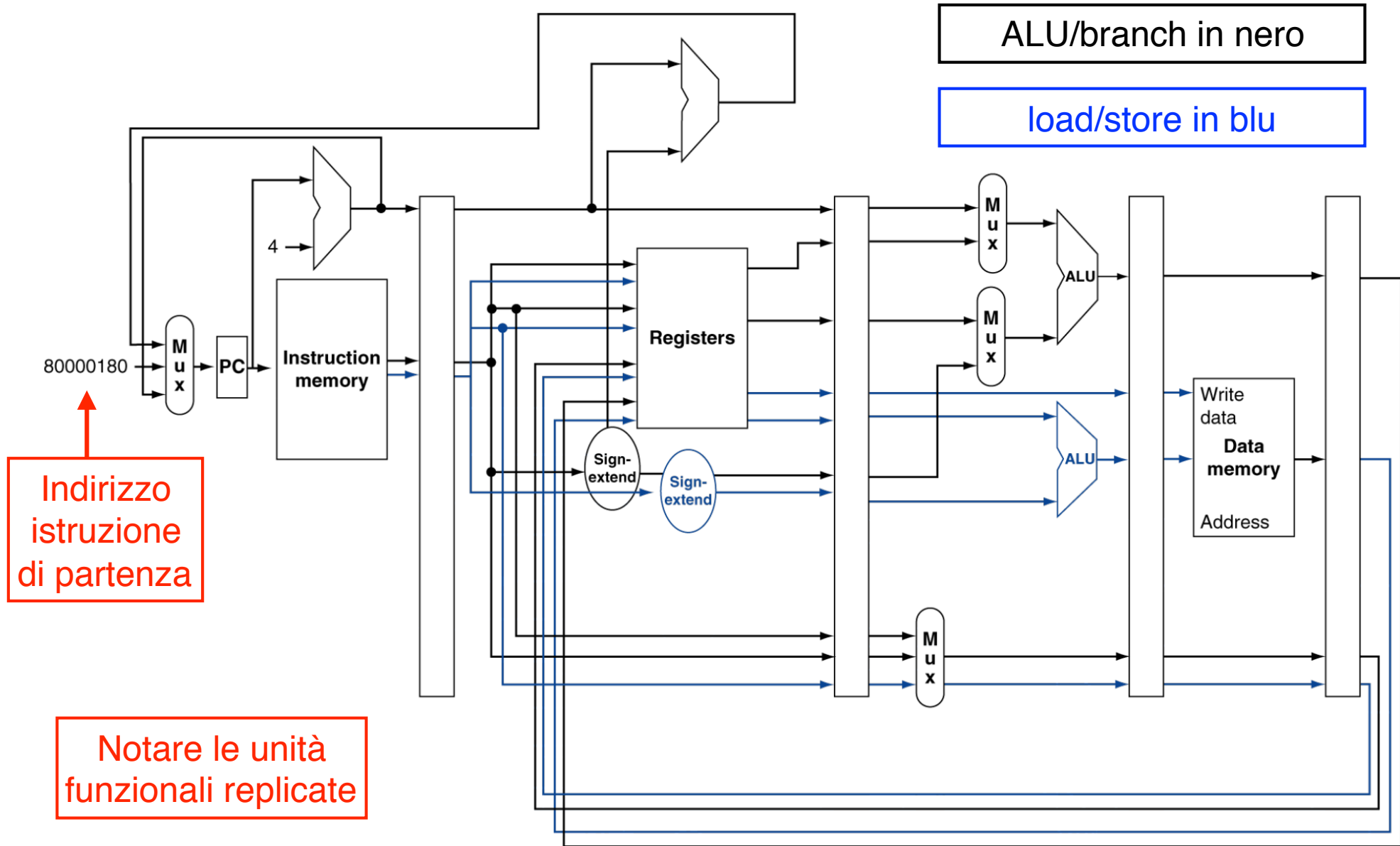
3

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

- Ovviamente non è detto che sia possibile suddividere un programma in questo modo in maniera efficiente:
- se una load/store richiede un valore ottenuto da una operazione aritmetica, non possono essere nello stesso packet



MIPS: 2-issue packets



MIPS: 2-issue packets

- Esempio di scheduling delle istruzioni:

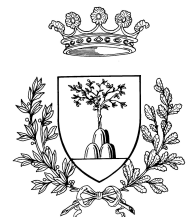
```
Loop: lw    $t0, 0($s1)      # $t0=array element
      addu  $t0, $t0, $s2    # add scalar in $s2
      sw    $t0, 0($s1)      # store result
      addi  $s1, $s1, -4     # decrement pointer
      bne   $s1, $zero, Loop # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	<code>nop</code>	<code>lw \$t0, 0(\$s1)</code>	1
	<code>addi \$s1, \$s1, -4</code>	<code>nop</code>	2
	<code>addu \$t0, \$t0, \$s2</code>	<code>nop</code>	3
	<code>bne \$s1, \$zero, Loop</code>	<code>sw \$t0, 4(\$s1)</code>	4



Multiple issue dinamico

- Nel multiple issue dinamico è il processore (hardware) che si occupa di scegliere quante istruzioni (0, 1, 2, 3, ...) iniziano ad ogni ciclo.
- Non c'è necessità di un compilatore che conosca l'hardware (come nel caso del multiple issue statico).
- Il processore permette di eseguire le istruzioni in ordine sparso (**out-of-order**):
 - permette di evitare gli stalli
 - i risultati delle istruzioni devono essere comunque scritti seguendo l'ordine corretto

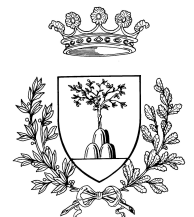


Multiple issue dinamico

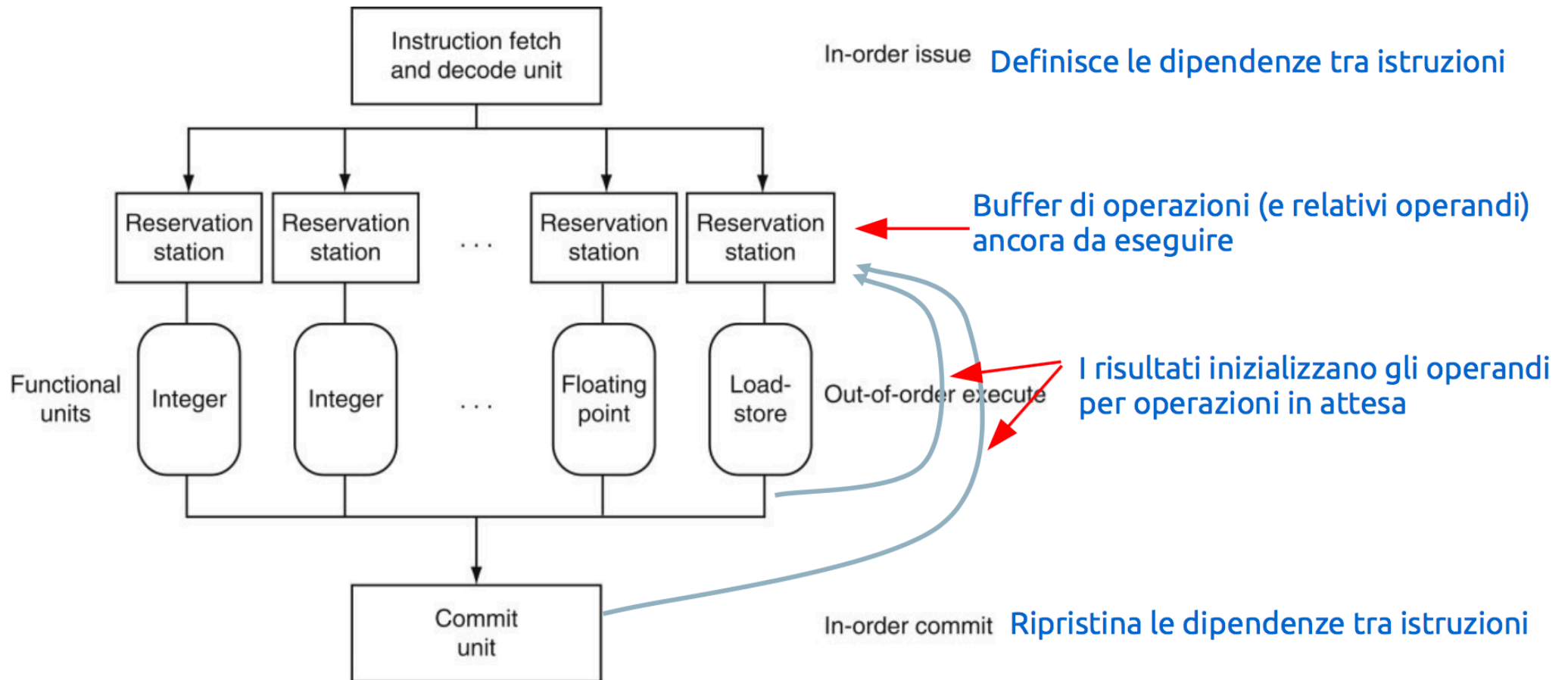
- Ad esempio:

```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub      $s4, $s4, $t3
slti     $t5, $s4, 20
```


- Il processore può eseguire la SUB mentre la ADDU è in attesa della LW.

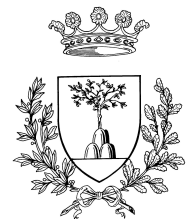


Multiple issue dinamico



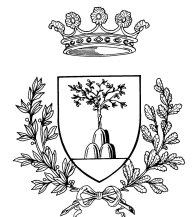
Multiple issue dinamico

- Perchè non accontentarci del multiple issue statico? 
 - gestione delle cache miss (prossime lezioni)
 - migliore gestione della predizione dei branch
 - ogni ISA ha latenza e criticità specifiche (il compilatore potrebbe non essere ottimizzato)



Speculazione

- Si cerca di indovinare cosa fare con una istruzione:
 - si inizia l'operazione appena possibile
 - i risultati parziali vengono tenuti fino a quando non si è in grado di verificare la loro utilità
- Si controlla se si è indovinato:
 - se sì, si completa l'operazione (commit)
 - se no, si fa roll-back e si esegue l'operazione giusta
- Tecnica comune sia a multiple issue statico che dinamico.




Speculazione

- Branch prediction:
 - prevedo il risultato di un branch e continuo → non faccio il commit fin tanto che non ho determinato il risultato del branch
- Load speculation:
 - cerco di evitare le latenza dovute ai cache miss (in dettaglio nelle prossime lezioni)
 - se ho risorse libere faccio le load future:
 - c'è il rischio che non servano
 - o che una store modifichi nel frattempo il valore che deve essere caricato
 - non faccio il commit fin tanto che non sono sicuro



Il multiple issue funziona?

- Sì, ma non quanto speriamo!
- Generalmente i programmi hanno così tante dipendenze che limitano molto le potenzialità dell'ILP.
- La “finestra” di istruzioni che si possono riordinare è limitata. 
- Gli accessi in memoria sono spesso costosi in termini di tempo:
 - è difficile occupare efficientemente tutta la pipeline
- La speculazione può aiutare molto se fatta correttamente.

