Il pipelining: tecniche di base

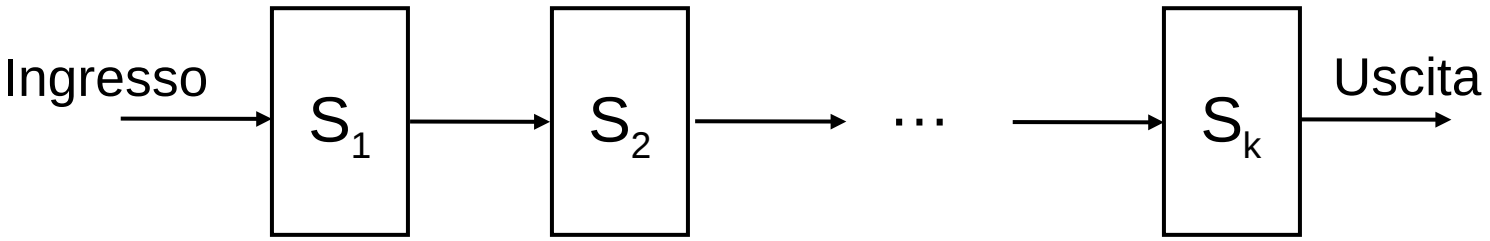
Il pipelining

- E’ una tecnica
  - per migliorare le prestazioni del processore
  - basata sulla sovrapposizione dell’esecuzione di più istruzioni appartenenti ad un flusso di esecuzione sequenziale

- Analogia con la catena di montaggio

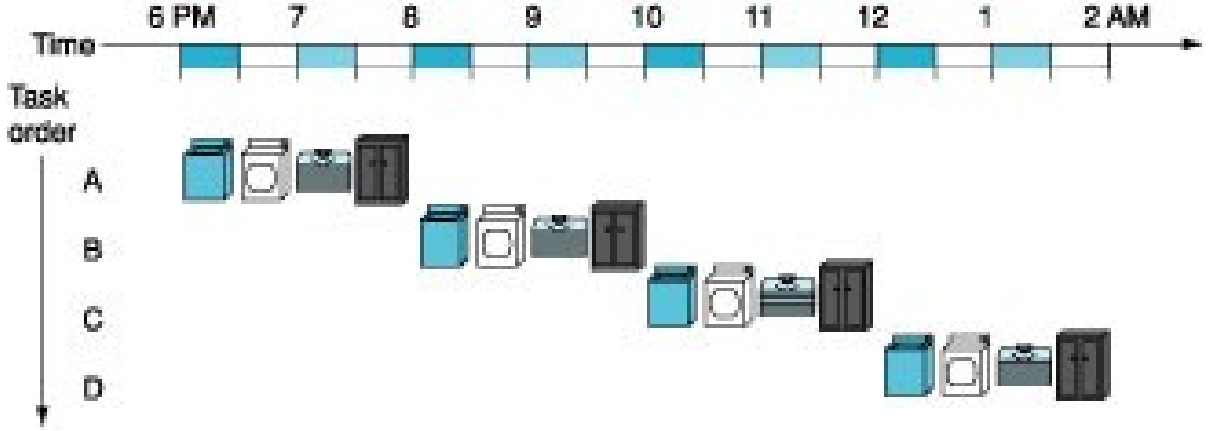
Idea base

- Il lavoro svolto da un processore con pipelining per eseguire un'istruzione è diviso in passi (*stadi della pipeline*), che richiedono una frazione del tempo necessario all'esecuzione dell'intera istruzione
- Gli stadi sono connessi in maniera seriale per formare la pipeline; le istruzioni:
  - entrano da un'estremità della pipeline
  - vengono elaborate dai vari stadi secondo l'ordine previsto
  - escono dall'altra estremità della pipeline

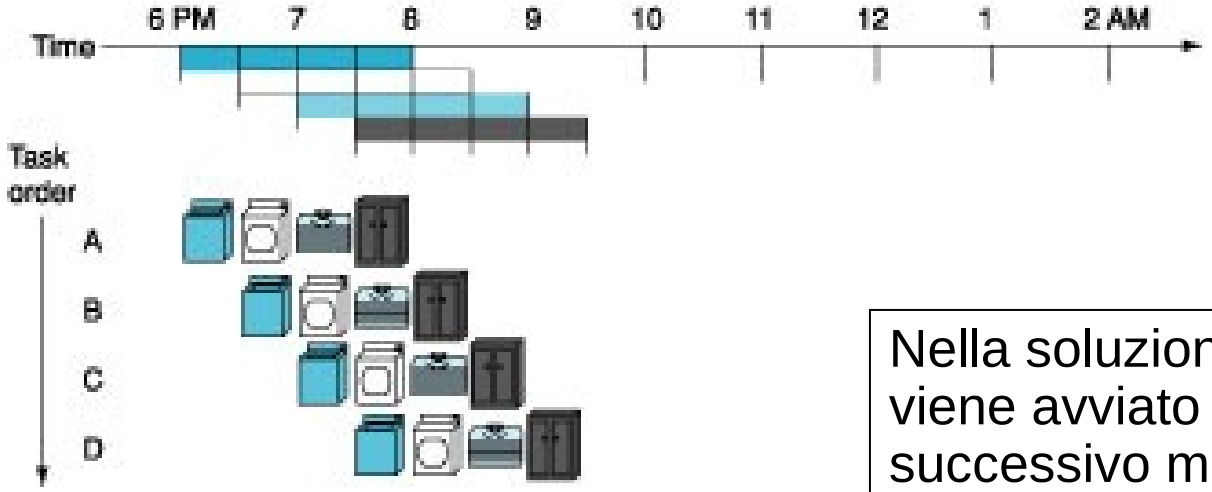


Un esempio pratico

Soluzione sequenziale/uniciclo



Soluzione con pipeline



Compiti

- Lavaggio 
- Asciugatura 
- Stiratura 
- Riordino 

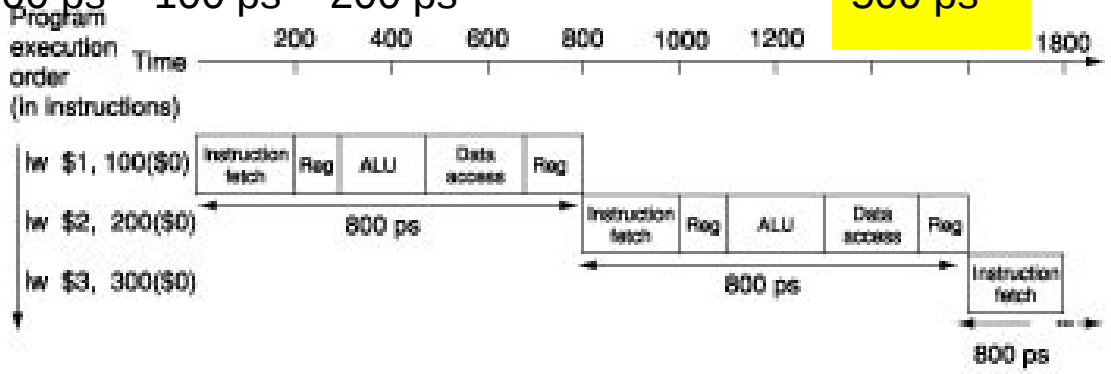
Nella soluzione con pipeline viene avviato il ciclo di lavaggio successivo mentre quello precedente è ancora in esecuzione in un’altra fase

Confronto tra ciclo singolo e pipeline

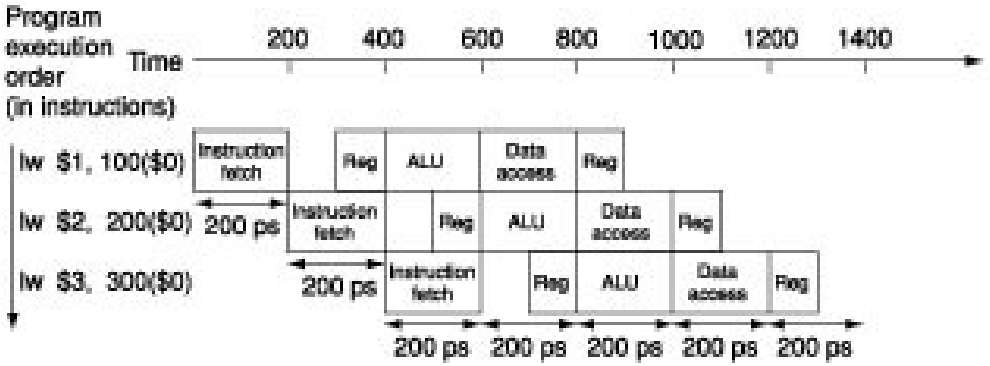
- Esempio di tempi di esecuzione delle diverse classi di istruzione

Istruzione	IF	ID	EX	MEM	WB	Totale
lw	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
sw	200 ps	100 ps	200 ps	200 ps		700 ps
formato R	200 ps	100 ps	200 ps		100 ps	600 ps
beq	200 ps	100 ps	200 ps			500 ps

Ciclo singolo



Pipeline



Alcuni commenti sul pipelining

- La presenza della pipeline aumenta il numero di istruzioni *contemporaneamente* in esecuzione

- Quindi, introducendo il pipelining nel processore, aumenta il ***throughput*** …
  - Throughput: numero di istruzioni eseguite nell'unità di tempo
- … ma non si riduce la ***latenza*** della singola istruzione
  - Latenza: tempo di esecuzione della singola istruzione, dal suo inizio fino al suo completamento
  - Un'istruzione che richiede 5 passi, continua a richiedere 5 cicli di clock per la sua esecuzione con pipelining, mentre una che ne richiederebbe 4 necessita di 5 cicli di clock

Stadi della pipeline

- Il tempo necessario per fare avanzare un'istruzione di uno stadio lungo la pipeline corrisponde ad un ciclo di clock di pipeline
- Poiché gli stadi della pipeline sono collegati in sequenza, devono operare in modo sincrono
  - Avanzamento nella pipeline sincronizzato dal clock
  - Durata del ciclo di clock del processore con pipeline determinata dalla durata dello stadio più lento della pipeline
    - Es.: 200 ps per l'esecuzione dell'operazione più lenta
  - Per alcune istruzioni, alcuni stadi sono cicli sprecati
- Obiettivo dei progettisti: bilanciare la durata degli stadi
- Se gli stadi sono *perfettamente bilanciati* e non ci sono istruz. con cicli sprecati, lo *speedup ideale* dovuto al pipelining è pari al numero di stadi della pipeline

$$\text{Speedup ideale}_{\text{pipeline}} = \frac{\text{tempo tra istruzioni}_{\text{no pipeline}}}{\text{tempo tra istruzioni}_{\text{pipeline}}} = \text{num. stadi pipeline}$$

Stadi della pipeline (2)

- Ma, in generale, gli stadi della pipeline non sono perfettamente bilanciati
- L'introduzione del pipelining comporta quindi costi aggiuntivi
  - L'intervallo di tempo per il completamento di un'istruzione è superiore al minimo valore possibile
  - Lo *speedup reale* sarà minore del numero di stadi di pipeline introdotto
    - In genere una pipeline a 5 stadi non riesce a quintuplicare le prestazioni

Miglioramento delle prestazioni

- Esempio: sequenza di 3 istruzioni lw (vedi lucido 4)
- Speedup ideale pari a 5, ma miglioramento più modesto
  - 3 istruzioni lw senza pipeline: 800×3 = 2400 ps
  - 3 istruzioni lw con pipeline: 1000 +200×2 = 1400 ps
  - Servono 2 stadi (400 ps) per svuotare la pipeline
  - Quindi 1400 ps invece di 2400 ps (2400/1400 = 1.7 circa)

- *In generale*: partendo dalla pipeline vuota con *k* stadi, per completare *n* istruzioni occorrono *k* + (*n*-1) cicli di clock
  - *k* cicli per riempire la pipeline e completare l'esecuzione della prima istruzione
  - *n*-1 cicli per completare le rimanenti *n*-1 istruzioni

Miglioramento delle prestazioni (2)

- All'aumentare del numero di istruzioni *n*, il rapporto tra i tempi totali di esecuzione su macchine senza e con pipeline si avvicina al limite ideale

  - Il tempo per riempire/svuotare la pipeline diventa trascurabile rispetto al tempo totale per completare le istruzioni

  - Esempio:
    - 1000 istruzioni lw senza pipeline: 800×1000 = 800000 ps
    - 1000 istruzioni lw con pipeline: 1000 + 200×(1000 -1)= 200800 ps
    - 200800 ps invece di 800000 ps (800000/200800 = 3.98 circa)

Miglioramento delle prestazioni (3)

- Nel caso asintotico ($n \rightarrow \infty$)
  - La latenza della singola istruzione lw peggiora
    - Passa da 800 ps (senza pipelining) a 1000 ps (con pipelining)
  - Il throughput migliora di 4 volte
    - Passa da 1 istruzione lw completata ogni 800 ps (senza pipelining) ad 1 istruzione lw completata ogni 200 ps (con pipelining)
- Se consideriamo un processore a singolo ciclo da 1000 ps (composto da 5 stadi ciascuno da 200 ps) ed un processore con pipelining (con 5 stadi da 200 ps ciascuno) nel caso asintotico
  - La latenza della singola istruzione rimane invariata e pari a 1000 ps
  - Il throughput migliora di 5 volte
    - Passa da 1 istruzione completata ogni 1000 ps (senza pipelining) ad 1 istruzione completata ogni 200 ps (con pipelining)

L’insieme di istruzioni MIPS ed il pipelining

- La progettazione dell’insieme di istruzioni del MIPS permette la realizzazione di una pipeline semplice ed efficiente
  - Tutte le istruzioni hanno la stessa lunghezza (32 bit)
    - Più semplice il caricamento dell’istruzione nel primo passo e la decodifica dell’istruzione nel secondo passo
  - Pochi formati di istruzioni con similitudine tra i formati
    - Possibile iniziare la lettura dei registri nel secondo passo, prima di sapere di che istruzione (e formato) si tratta
  - Le operazioni in memoria sono limitate alle istruzioni di load/store
    - Possibile usare il terzo passo per calcolare l’indirizzo
  - Allineamento degli operandi in memoria (un solo ciclo di lettura per leggere i 32 bit)
    - Possibile usare un solo stadio per trasferire dati tra processore e memoria
  - Ogni istruzione MIPS scrive al più un risultato e lo fa nell’ultimo ciclo della pipeline

Esecuzione delle istruzioni nel processore con pipeline

IF Instruction Fetch	ID Instruction Decode	EX EXecute	MEM MEMory access	WB Write-Back

- Istruzioni logico-aritmetiche

IF Prel. istr. e incr. PC	ID Lettura reg. sorgente	EX Op. ALU su dati letti		WB Scrittura reg. dest.

- Istruzioni di load

IF Prel. istr. e incr. PC	ID Lettura reg. base	EX Somma ALU	MEM Prelievo dato da M	WB Scrittura reg. dest.

- Istruzioni di store

IF Prel. istr. e incr. PC	ID Lettura reg. base e sorgente	EX Somma ALU	MEM Scrittura dato in M	

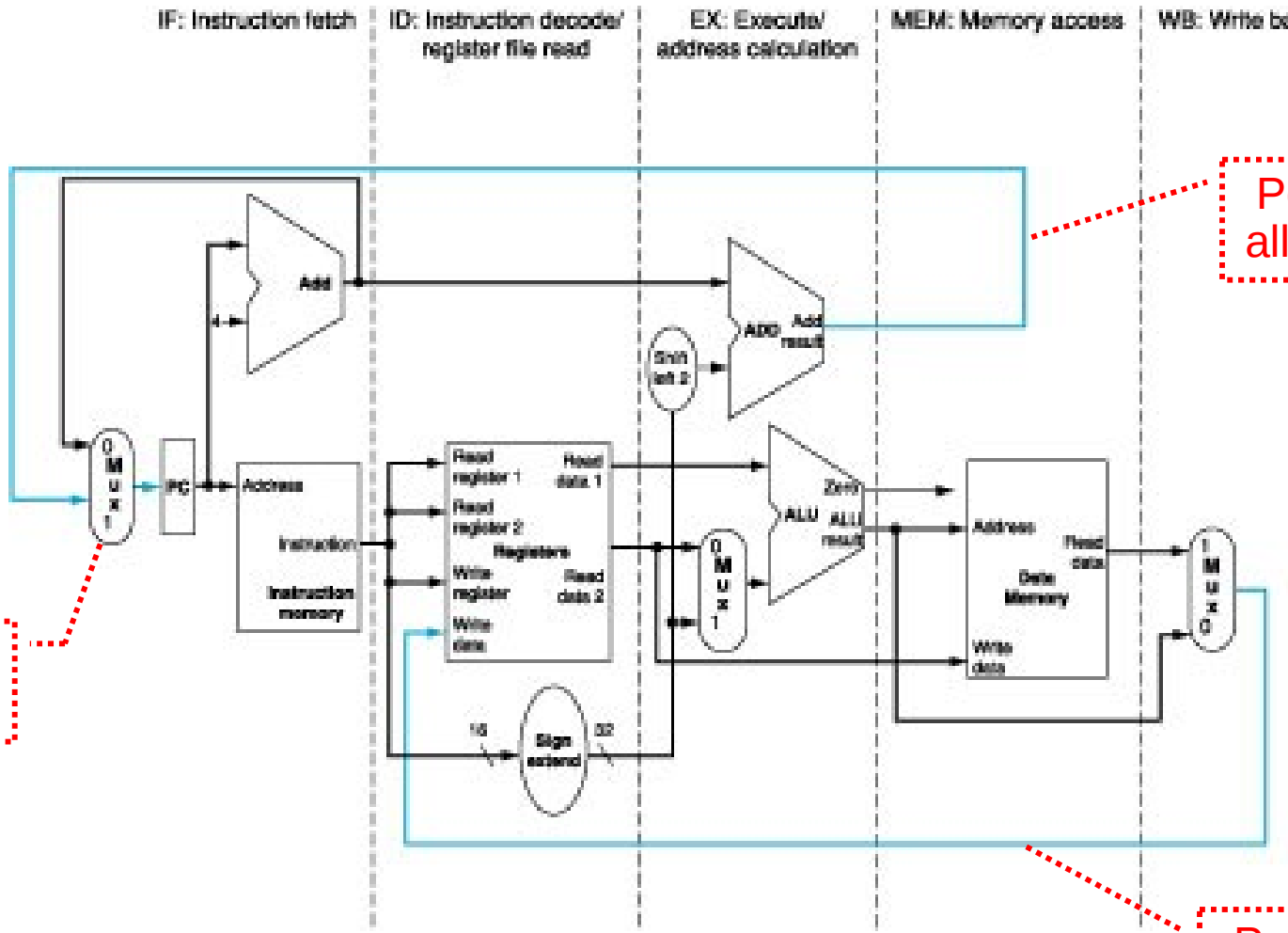
- Istruzioni di beq

IF Prel. istr. e incr. PC	ID Lettura reg. sorgente	EX calc. indirizzo salto	MEM Scrittura PC	

Come progettare l'unità di elaborazione?

- La suddivisione dell'istruzione in 5 stadi implica che in ogni ciclo di clock siano in esecuzione 5 istruzioni
  - La struttura di un processore con pipeline a 5 stadi deve essere scomposta in 5 parti (o *stadi di esecuzione*), ciascuna della quali corrispondente ad una delle fasi della pipeline
- Occorre introdurre una separazione tra i vari stadi
  - *Registri di pipeline*
- Inoltre, diverse istruzioni in esecuzione nello stesso istante possono richiedono risorse hardware simili
  - Replicazione delle risorse hardware
- Riprendiamo lo schema dell'unità di elaborazione a ciclo singolo ed identifichiamo i 5 stadi

L’unità di elaborazione a ciclo singolo



Percorso all’indietro

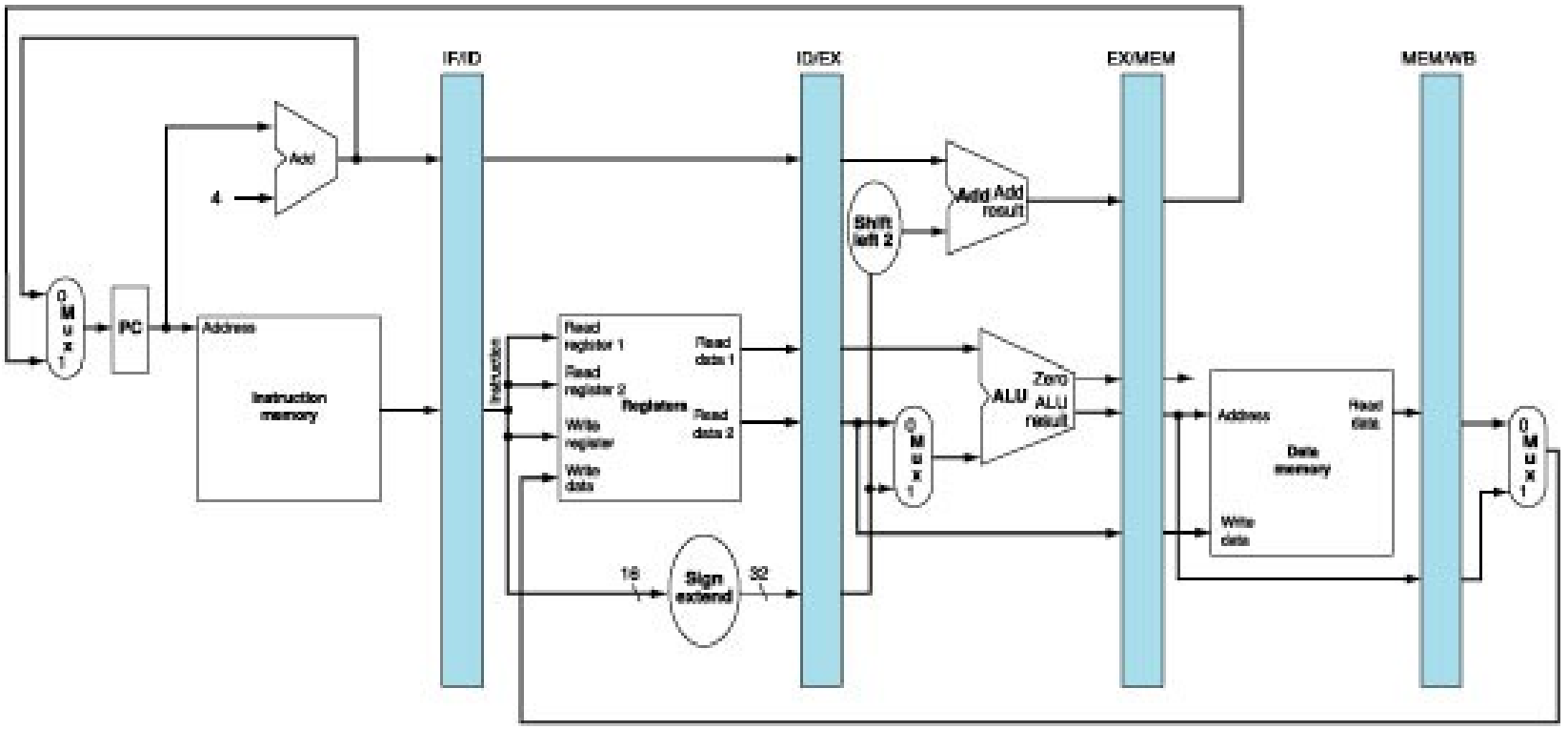
Spostato in IF

Percorso all’indietro

L’unità di elaborazione con pipeline

- Principio guida:
  - Permettere il riuso delle componenti per l’istruzione successiva
- Introduzione di *registri di pipeline* (registri interstadio)
  - Ad ogni ciclo di clock le informazioni procedono da un registro di pipeline a quello successivo
  - Il nome del registro è dato dal nome dei due stadi che separa
    - Registro IF/ID (Instruction Fetch / Instruction Decode)
    - Registro ID/EX (Instruction Decode / EXecute)
    - Registro EX/MEM (Execute / MEMory access)
    - Registro MEM/WB (MEMory access / Write Back)
  - Il PC può essere considerato come un registro di pipeline per lo stadio IF
- Rispetto all’unità a ciclo singolo, il multiplexer del PC è stato spostato nello stadio IF
  - Per evitare conflitti nella sua scrittura in caso di istruzione di salto

L’unità di elaborazione con pipeline (2)

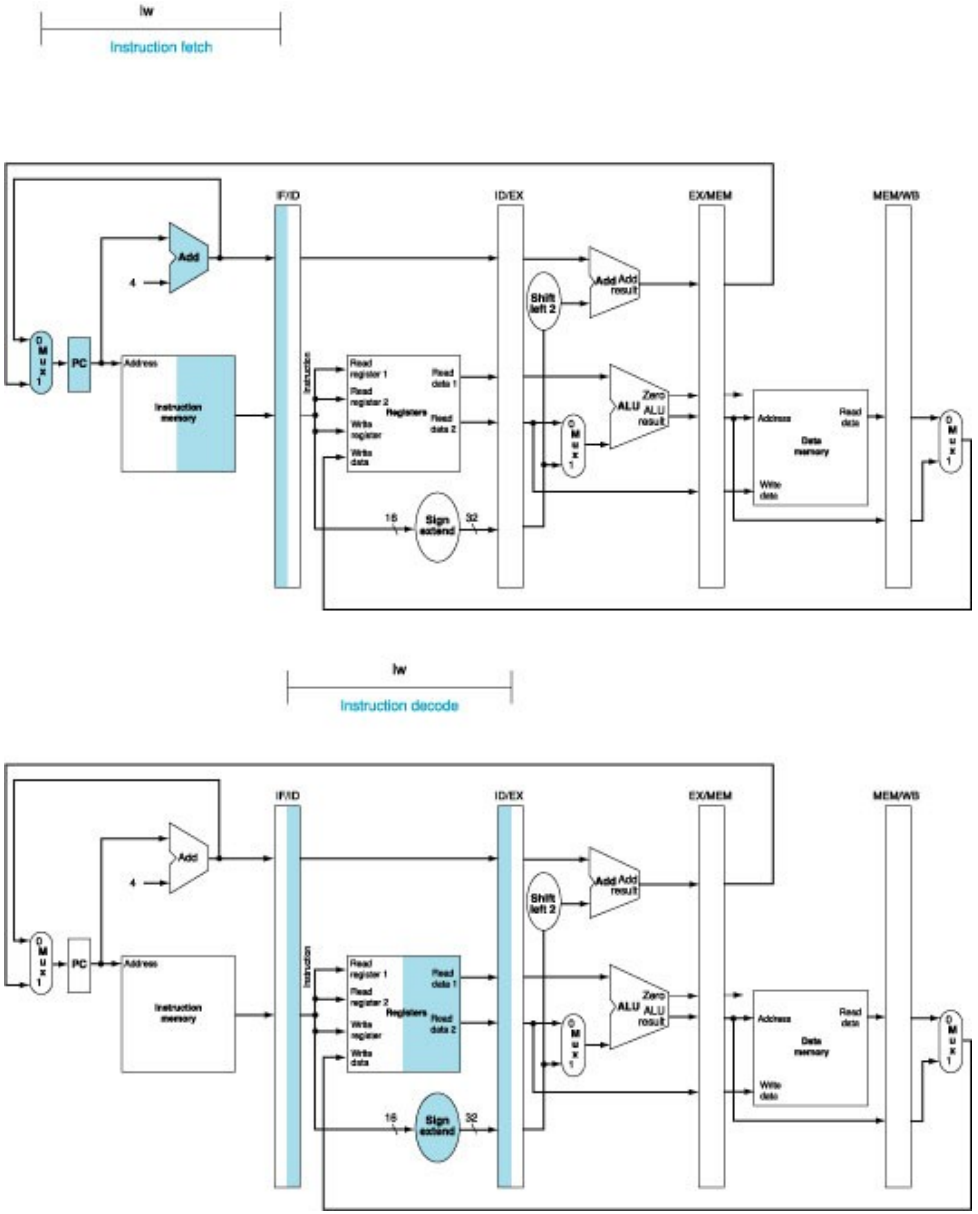


- Quale è la dimensione dei registri di pipeline ricavabile dallo schema?
  - IF/ID: 64 bit (32+32)
  - ID/EX: 128 bit (32+32+32+32)
  - EX/MEM: 97 bit (32+32+32+1)
  - MEM/WB: 64 bit (32+32)

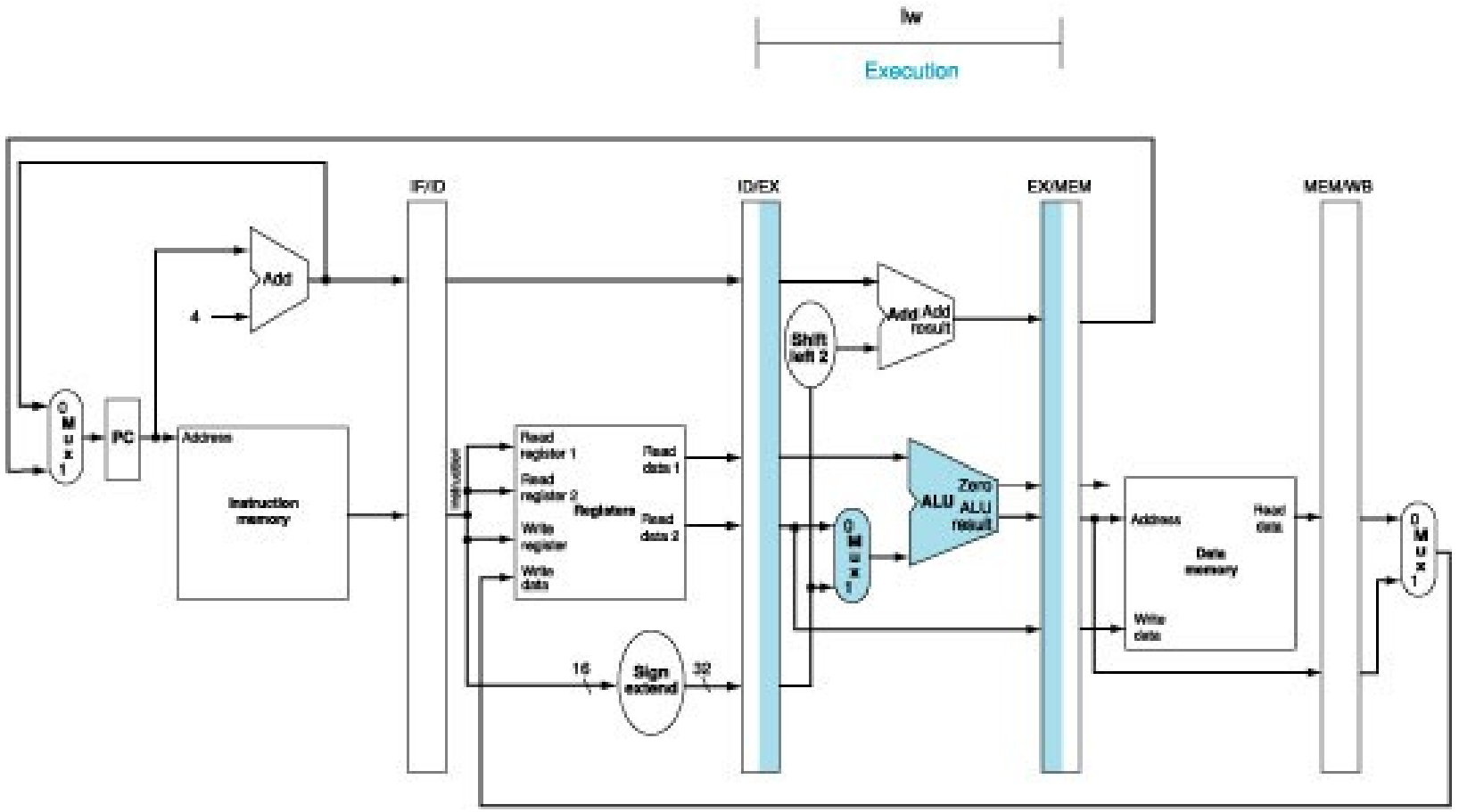
Uso dell’unità con pipeline

- Come viene eseguita un’istruzione nei vari stadi della pipeline?
- Consideriamo per prima l’istruzione lw
  - Prelievo dell’istruzione
  - Decodifica dell’istruzione e lettura dei registri
  - Esecuzione (uso dell’ALU per il calcolo dell’indirizzo)
  - Lettura dalla memoria
  - Scrittura nel registro
- Analizziamo poi l’esecuzione dell’istruzione sw
- Infine consideriamo l’esecuzione contemporanea di più istruzioni

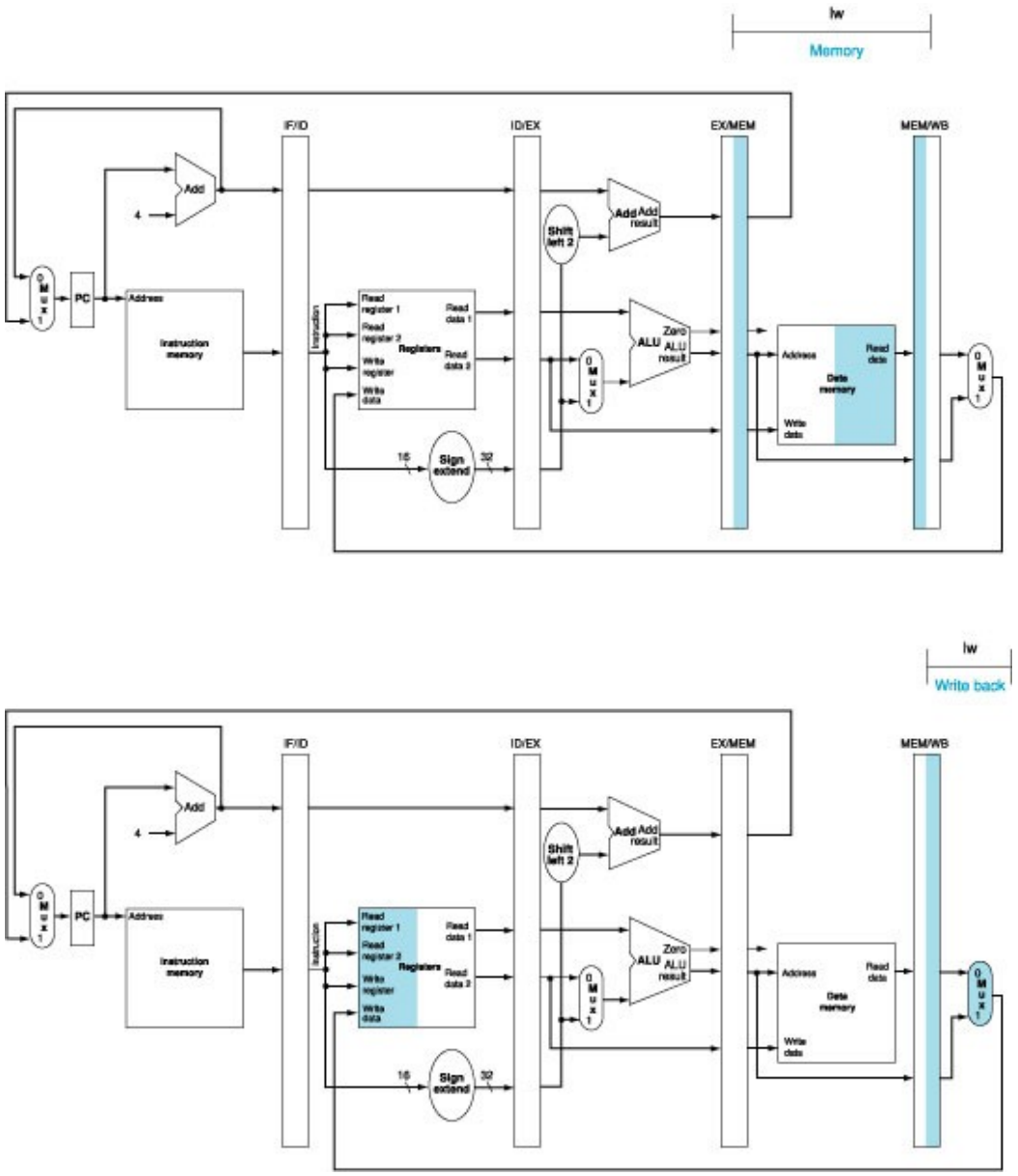
Esecuzione di lw: primo e secondo stadio



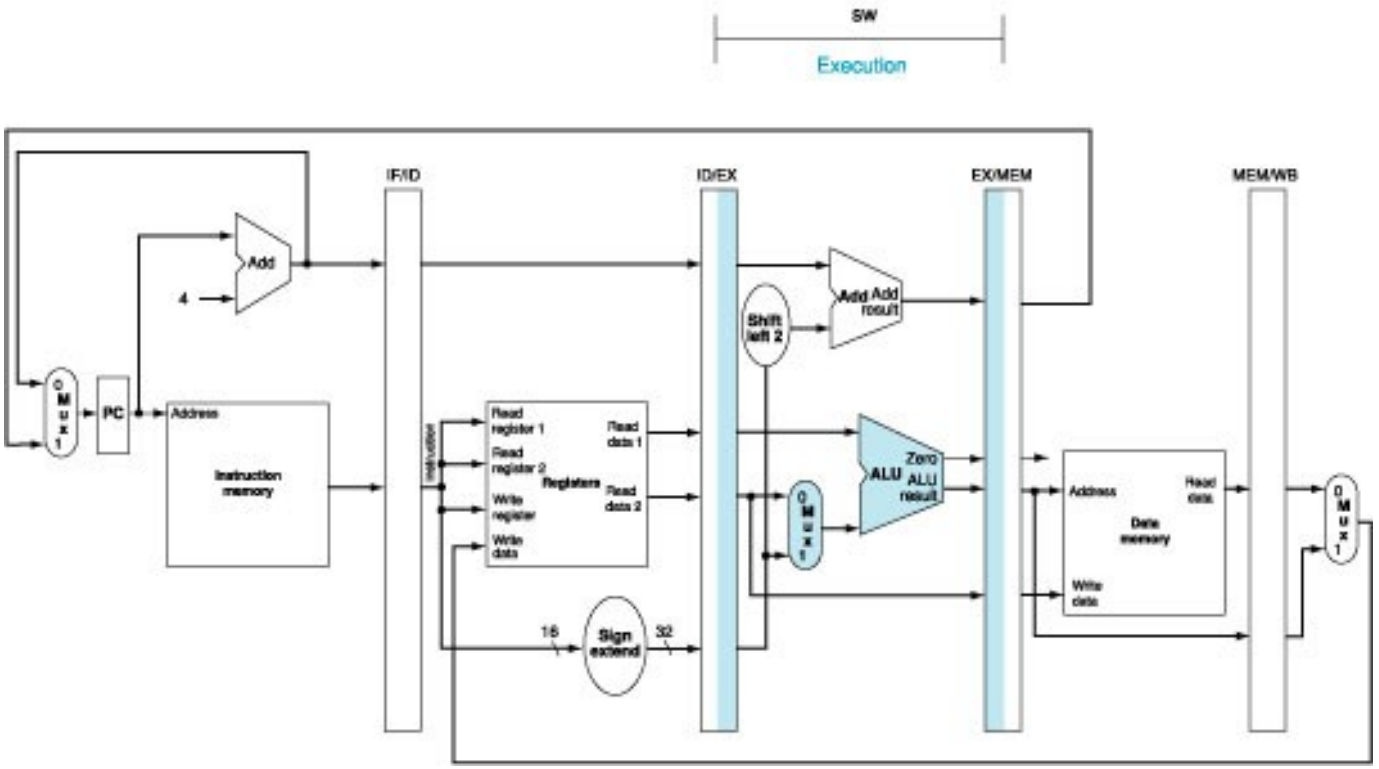
Esecuzione di lw: terzo stadio



Esecuzione di lw: quarto e quinto stadio

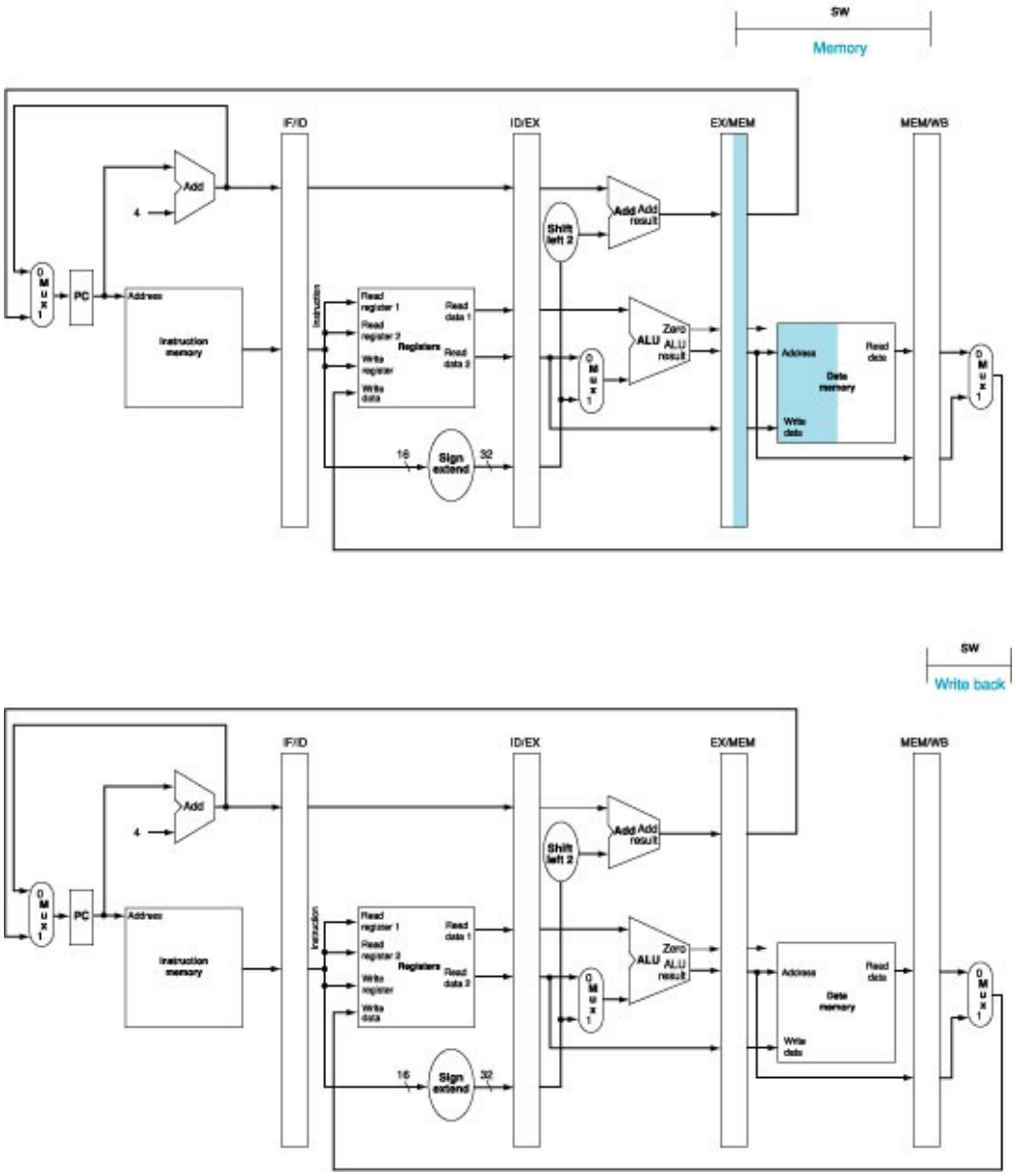


Esecuzione di sw: terzo stadio

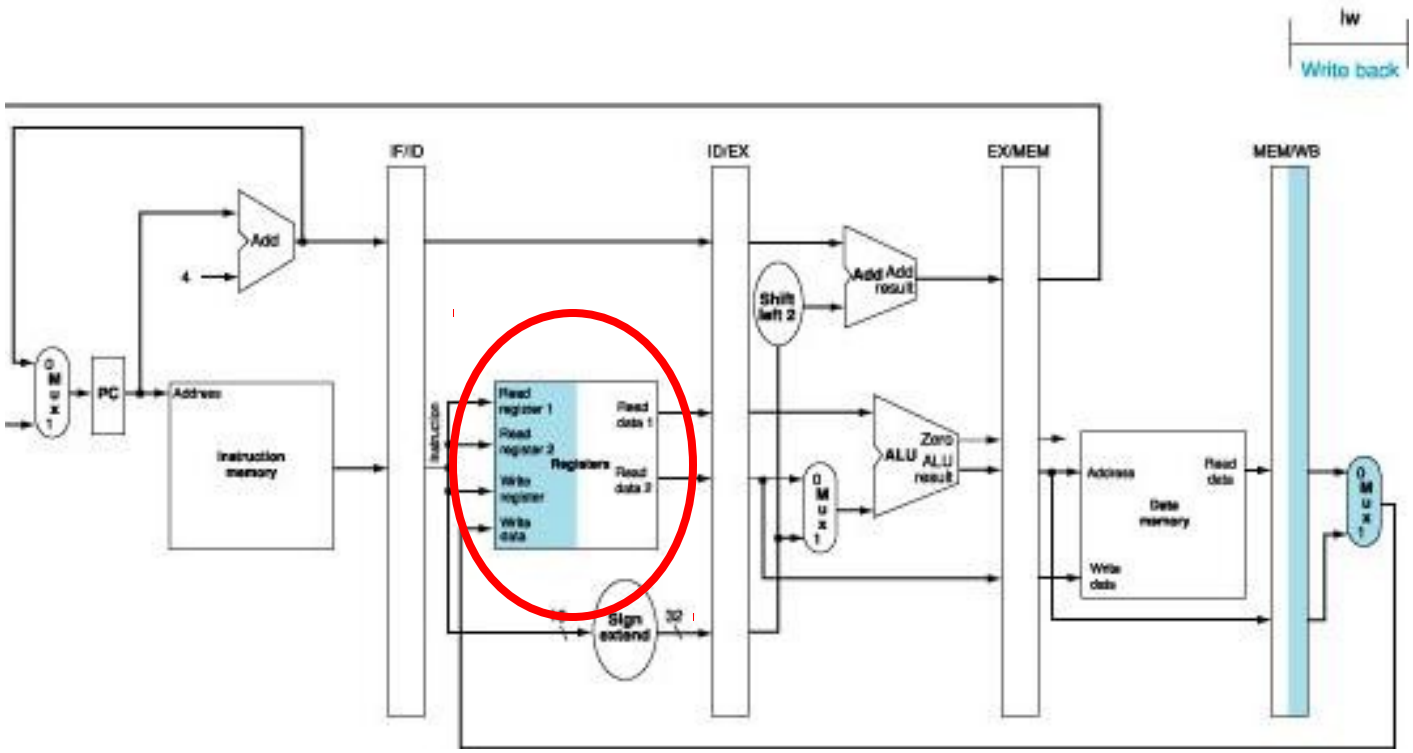


- Il valore del secondo registro viene scritto nel registro ID/EX per poterlo usare nello stadio MEM

Esecuzione di sw: quarto e quinto stadio

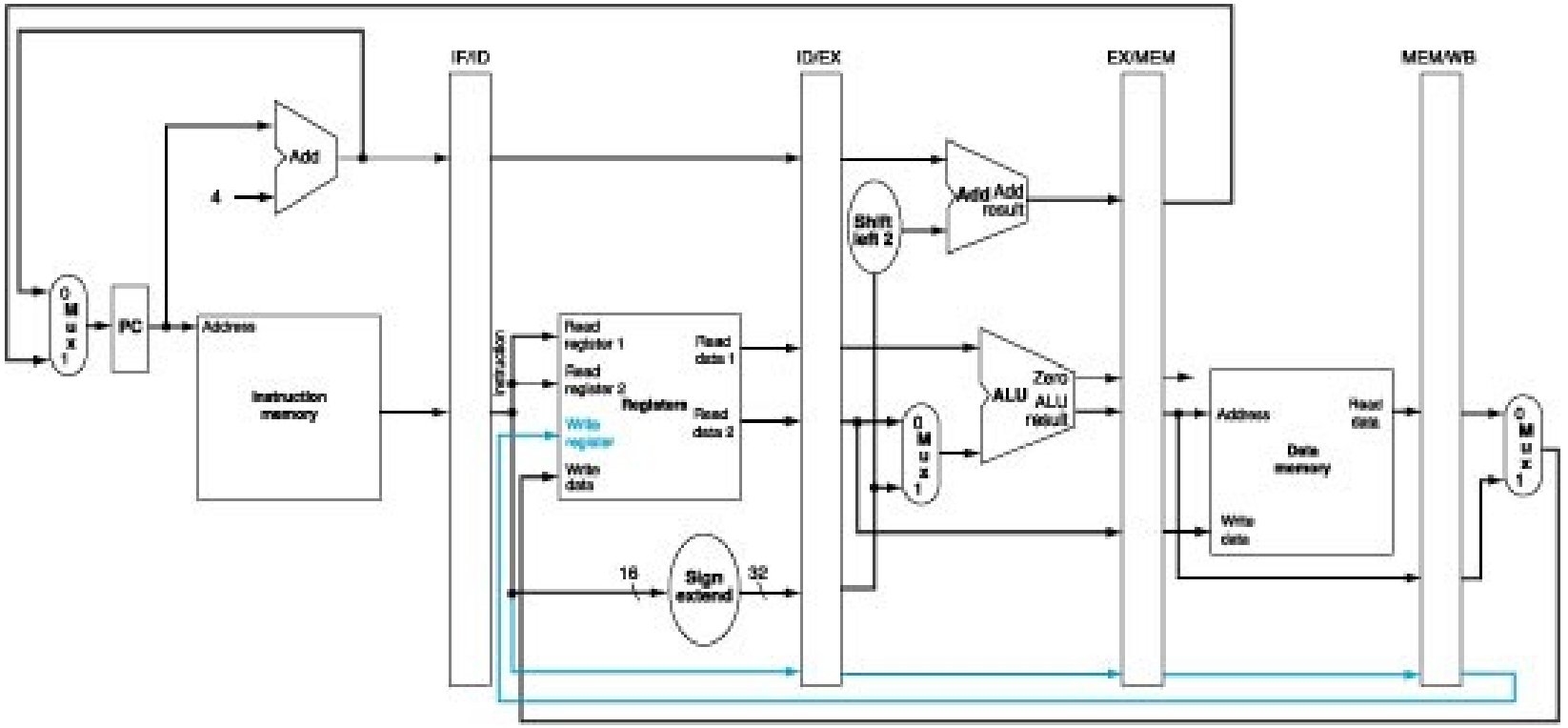


Caccia all'errore…



- Quale registro di destinazione viene scritto?
  - Il registro IF/ID contiene un'istruzione successiva a lw
- Soluzione
  - Occorre preservare l'identificativo del registro di destinazione

Soluzione



- L’identificativo del registro di destinazione viene scritto nei registri di pipeline:
  - Prima in ID/EX, poi in EX/MEM, infine in MEM/WB

Due istruzioni in esecuzione

- Consideriamo la sequenza di istruzioni MIPS

  lw $10, 20($1)

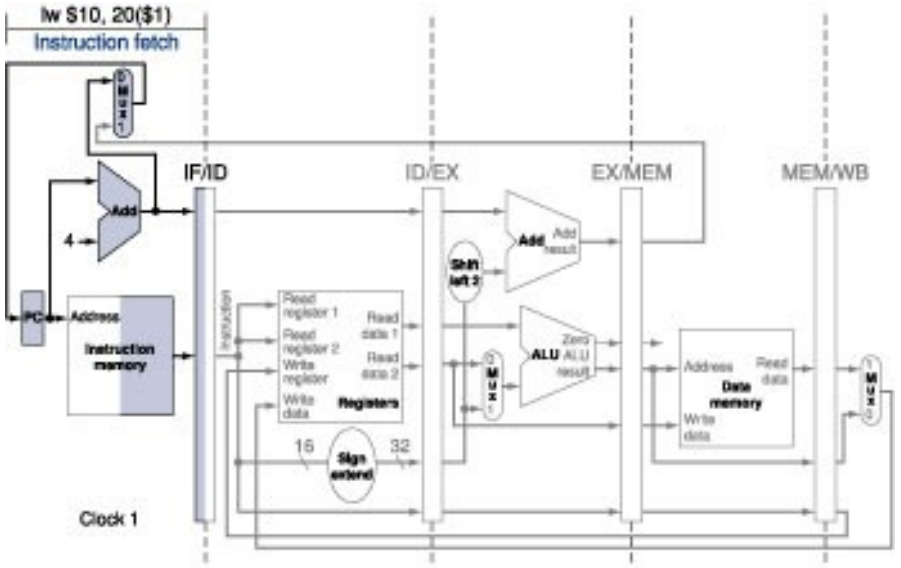
  sub $11, $2, $3

- Analizziamo l’esecuzione della sequenza nei 6 cicli di clock necessari

Il primo ed il secondo ciclo di clock

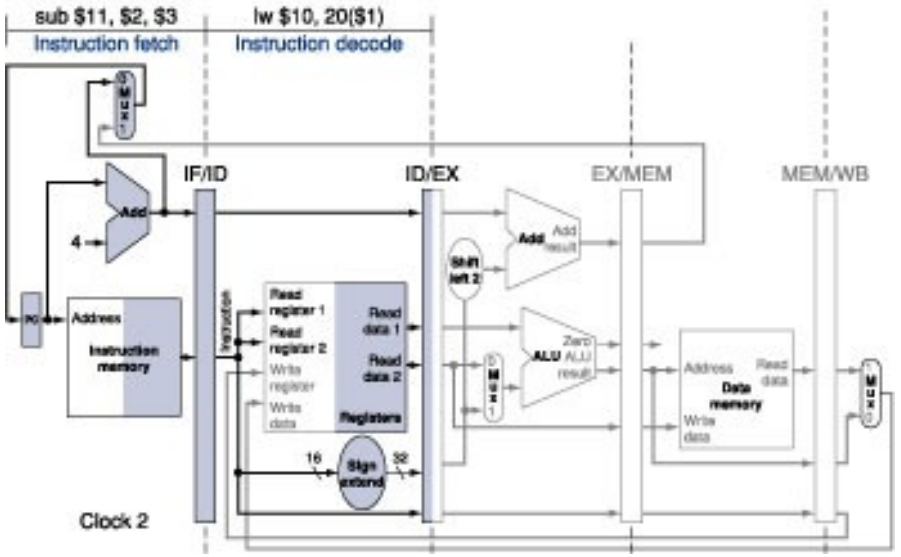
Ciclo 1

- lw: entra nella pipeline



Ciclo 2

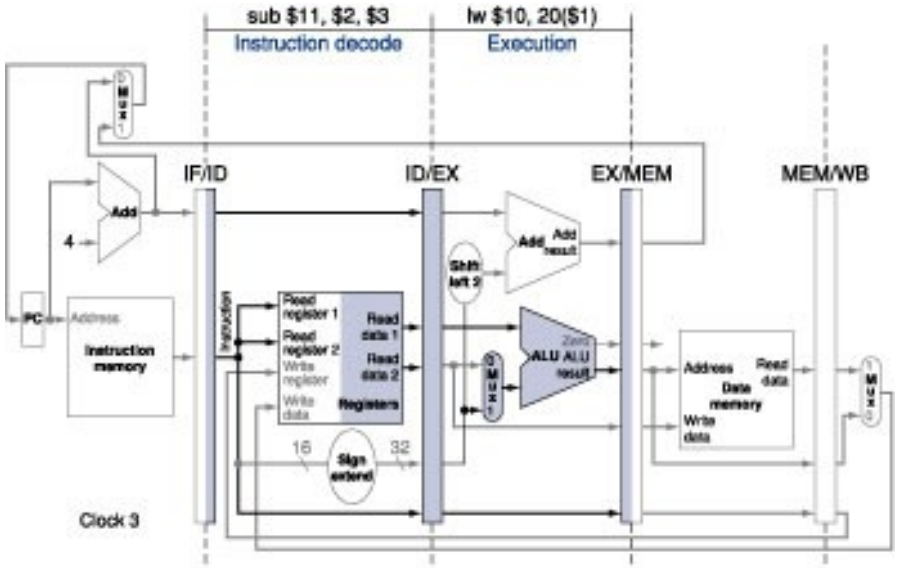
- sub: entra nella pipeline
- lw: entra nello stadio ID



Il terzo ed il quarto ciclo di clock

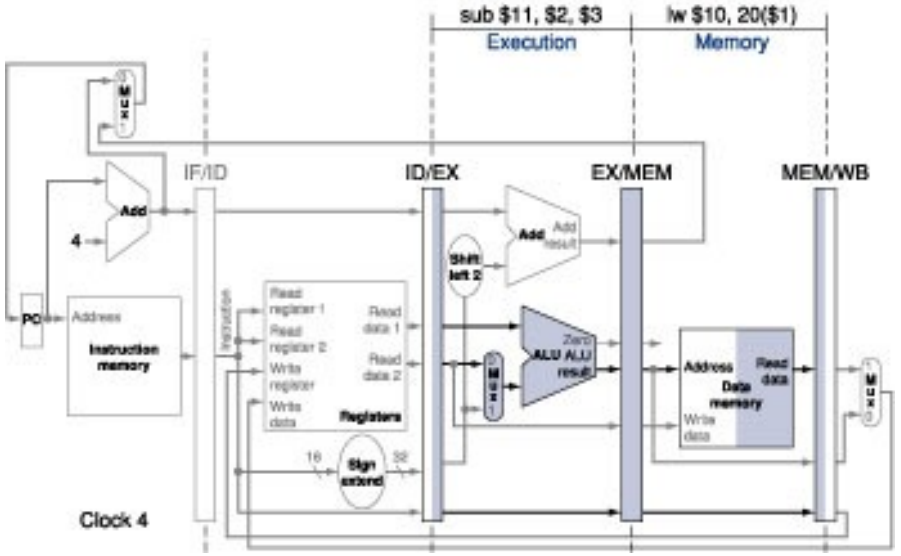
Ciclo 3

- lw: entra nello stadio EX
- sub: entra nello stadio ID



Ciclo 4

- lw: entra nello stadio MEM e legge la locazione di memoria con indirizzo salvato in EX/MEM
- sub: entra nello stadio EX; il risultato della sottrazione è scritto in EX/MEM alla fine del ciclo



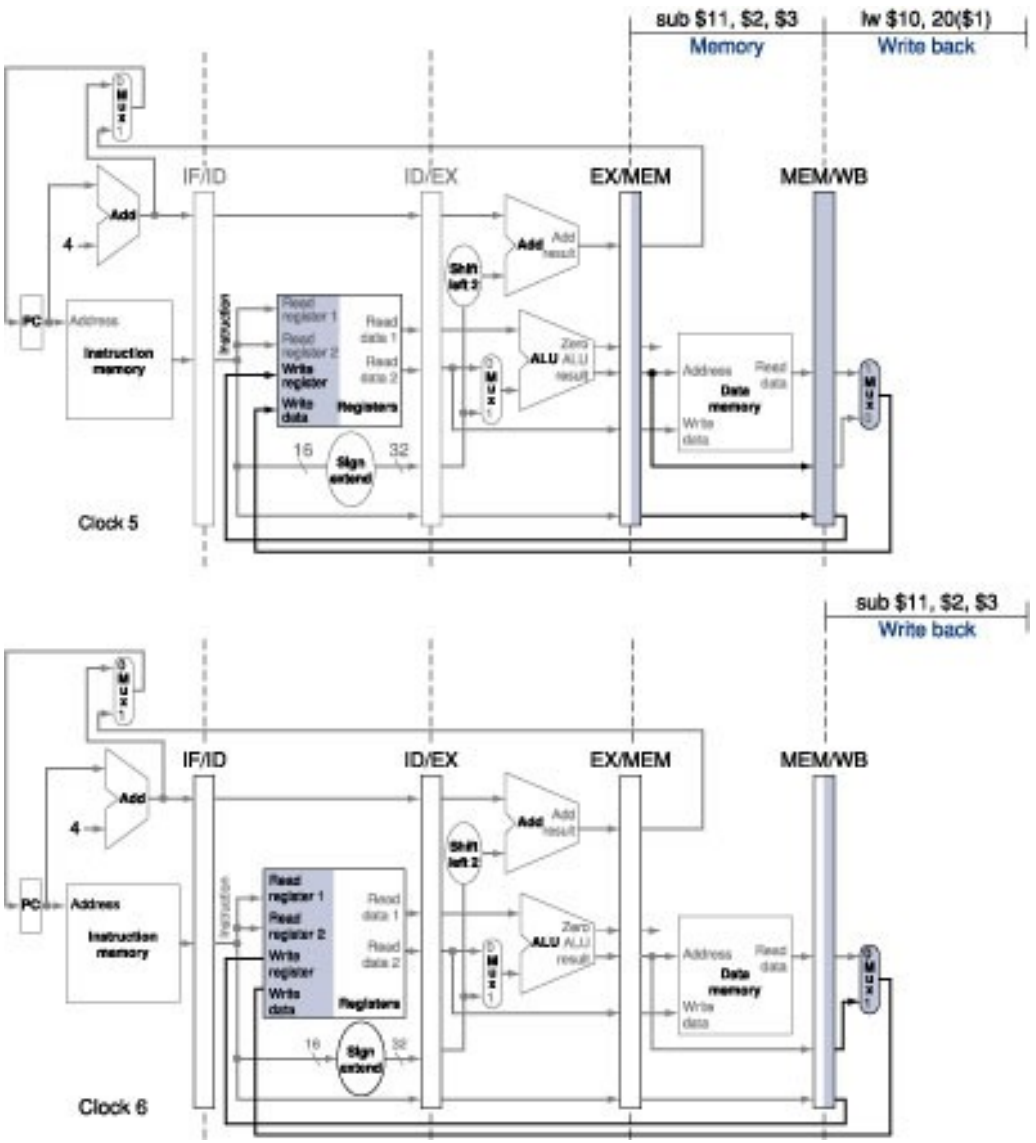
Il quinto ed il sesto ciclo di clock

Ciclo 5

- lw: termina con la scrittura del valore in MEM/WB nel registro $10 del banco
- sub: il risultato della sottrazione è scritto in MEM/WB

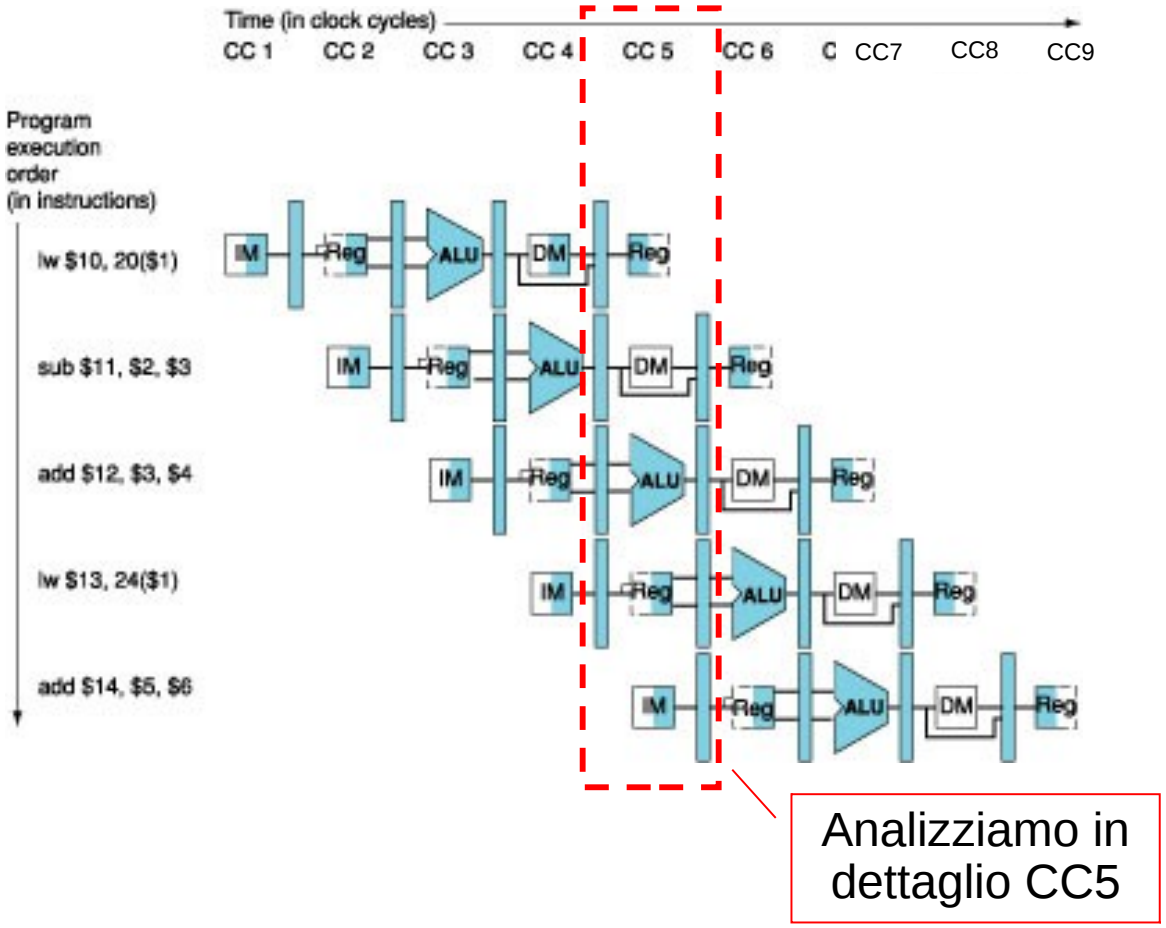
Ciclo 6

- sub: termina con la scrittura del valore in MEM/WB nel registro $11 del banco



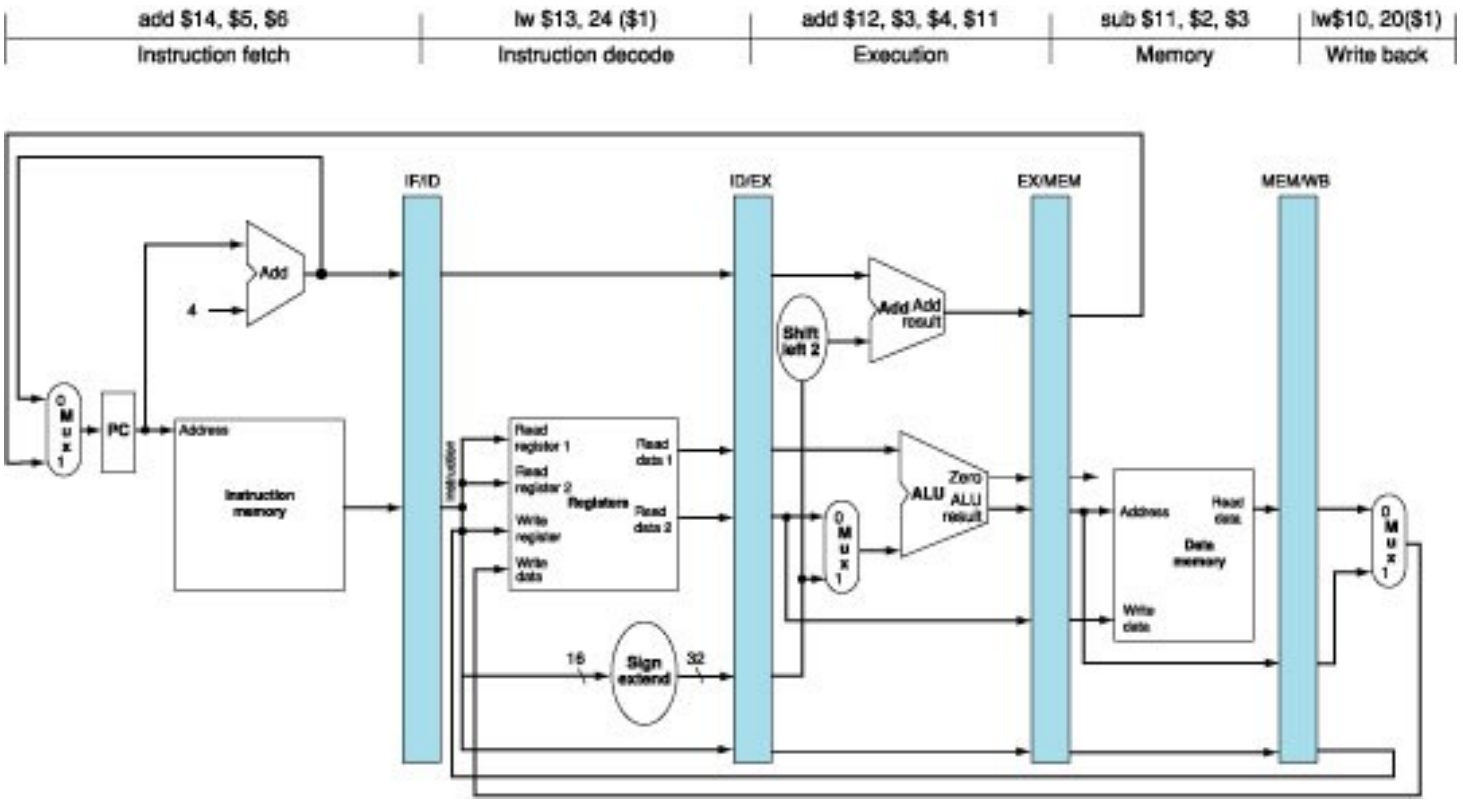
Cinque istruzioni in esecuzione

- Diagramma della pipeline con più cicli di clock
  - Fornisce una rappresentazione orientata alle risorse e semplificata



Il quinto ciclo di clock

- Diagramma della pipeline a singolo ciclo di clock
  - Fornisce una rappresentazione più dettagliata ed in verticale del diagramma con più cicli di clock



Esercizio

- Considerare la sequenza di istruzioni MIPS

  add $4, $2, $3

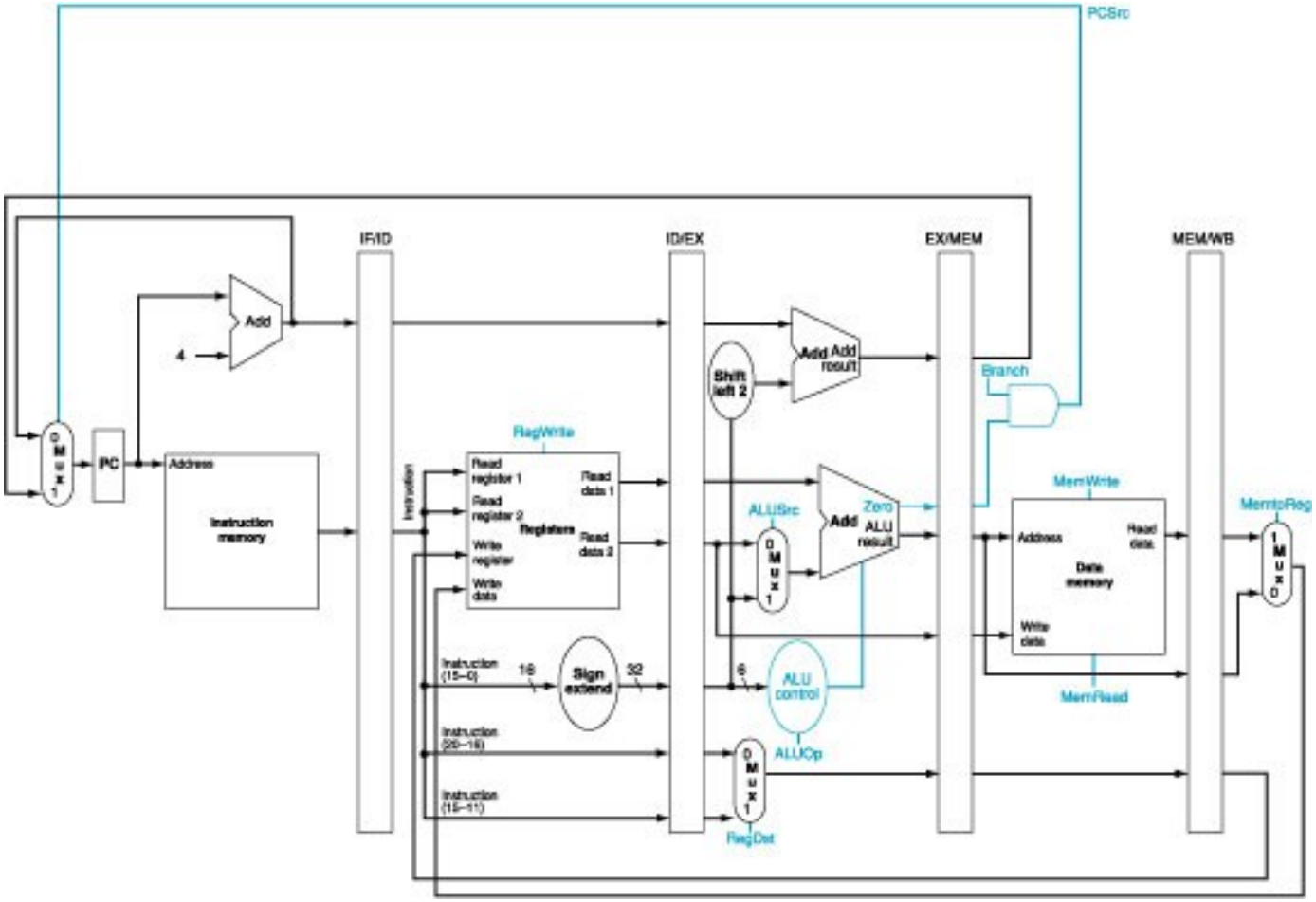
  sw $5, 4($2)

- Analizzare l’esecuzione della sequenza nei 6 cicli di clock necessari

Controllo dell’unità con pipeline

- I dati viaggiano attraverso gli stadi della pipeline
- Tutti i dati appartenenti ad un’istruzione devono essere mantenuti all’interno dello stadio
- Le informazioni si trasferiscono solo tramite i registri della pipeline
- Le informazioni di controllo di una istruzione devono “viaggiare” con gli operandi / i dati dell’istruzione stessa

I segnali di controllo



- Non sono necessari segnali di controllo per la scrittura dei registri di pipeline (si usa il clock)

I segnali di controllo (2)

Si raggruppano i segnali di controllo in base agli stadi della pipeline:

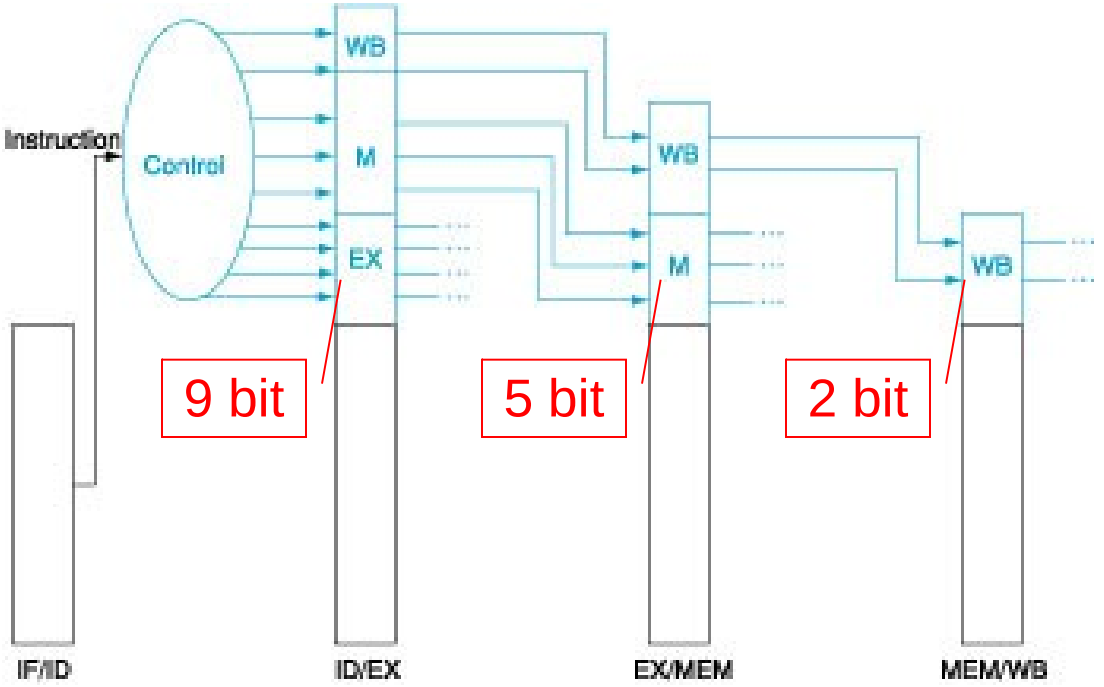
- Prelievo dell'istruzione
  - Identico per tutte le istruzioni
- Decodifica dell'istruzione/lettura del banco dei registri
  - Identico per tutte le istruzioni
- Esecuzione/calcolo dell'indirizzo
  - RegDst, ALUOp, ALUSrc
- Accesso alla memoria
  - Branch, MemRead,MemWrite
- Scrittura del risultato
  - MemtoReg, RegWrite

I segnali di controllo (3)

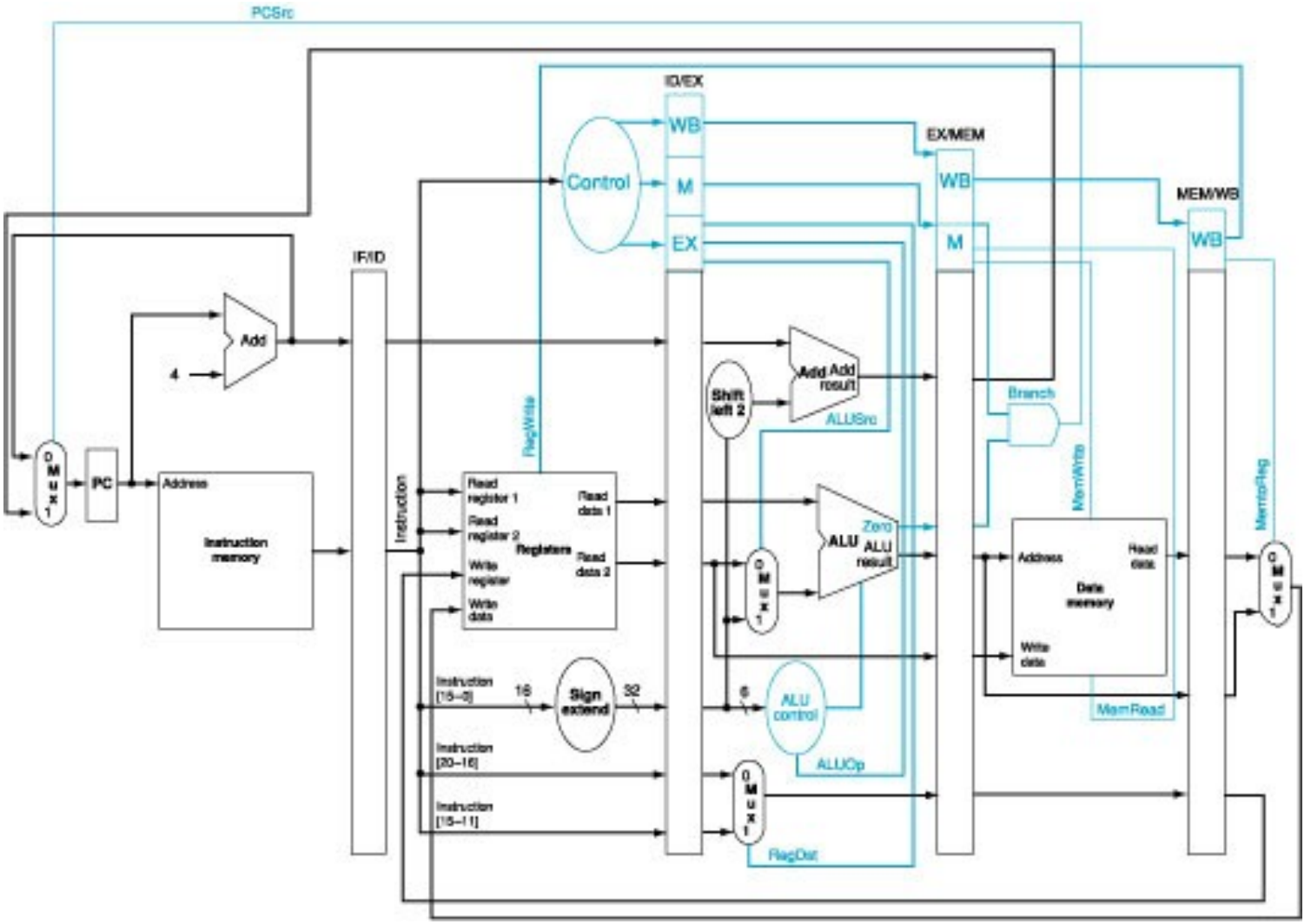
Istruzione	Segnali di controllo EX				Segnali di controllo MEM			Segnali di controllo WB	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem Read	Mem Write	Reg Write	Memto Reg
tipo-R	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

SCO

- I registri di pipeline contengono anche i valori dei segnali di controllo
  - Al massimo 8 variabili di controllo (9 bit nel disegno, di cui 2 per lo SCO ALU, ma noi gestiamo in modo diverso i segnali di controllo dell'ALU)
- I valori necessari per lo stadio successivo vengono propagati dal registro di pipeline corrente al successivo



SCO-SCA


PCSrc
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Control
IF/ID
Add
4
M u x
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALUSrc
Branch
MemWrite
MemtoReg
Zero
ALU
ALU result
Address
Data memory
Read data
Write data
MemRead
Instruction [15–0]
16
Sign extend
32
6
ALU control
ALUOp
Instruction [20–16]
Instruction [15–11]
RegDst

Esempio

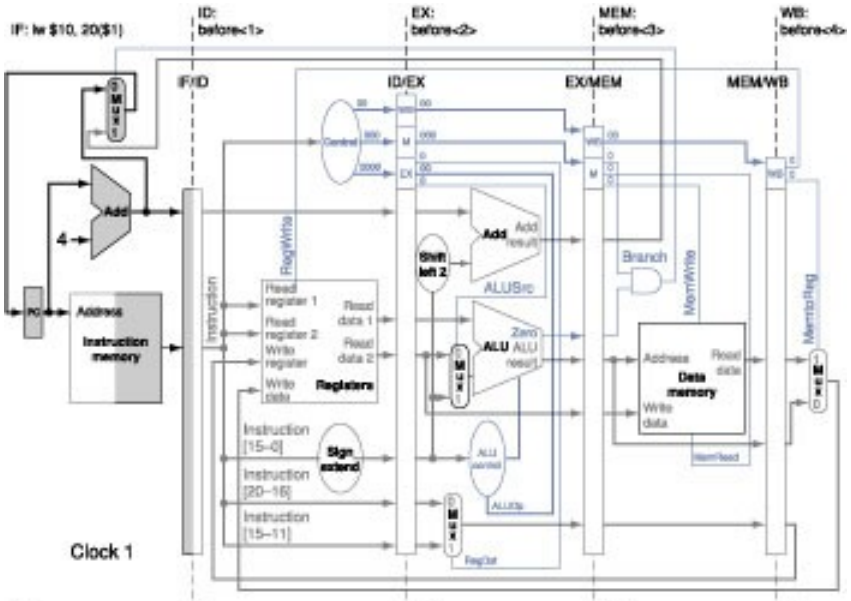
- Consideriamo la sequenza di istruzioni MIPS

```
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
```

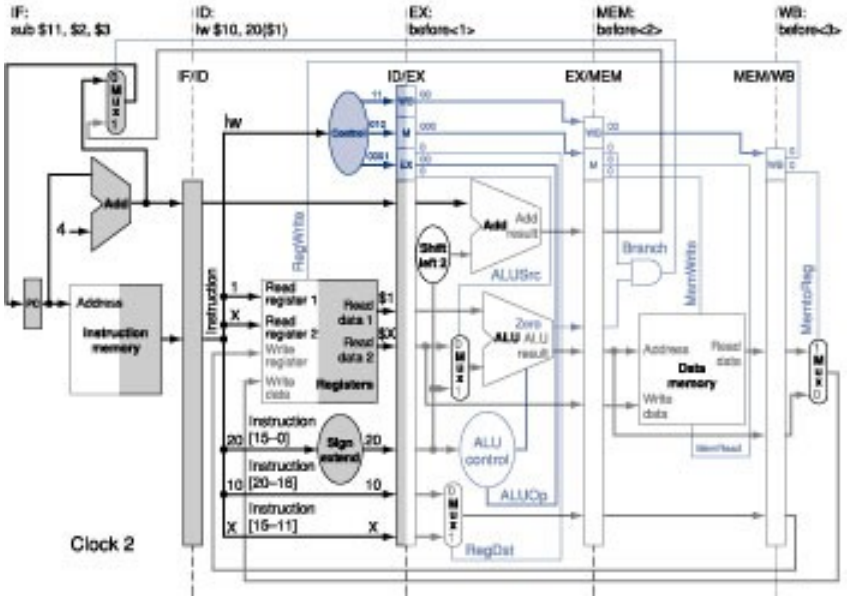
- Analizziamo l’esecuzione della sequenza nei 9 cicli di clock necessari

Esempio: cicli di clock 1 e 2

- lw: entra nella pipeline

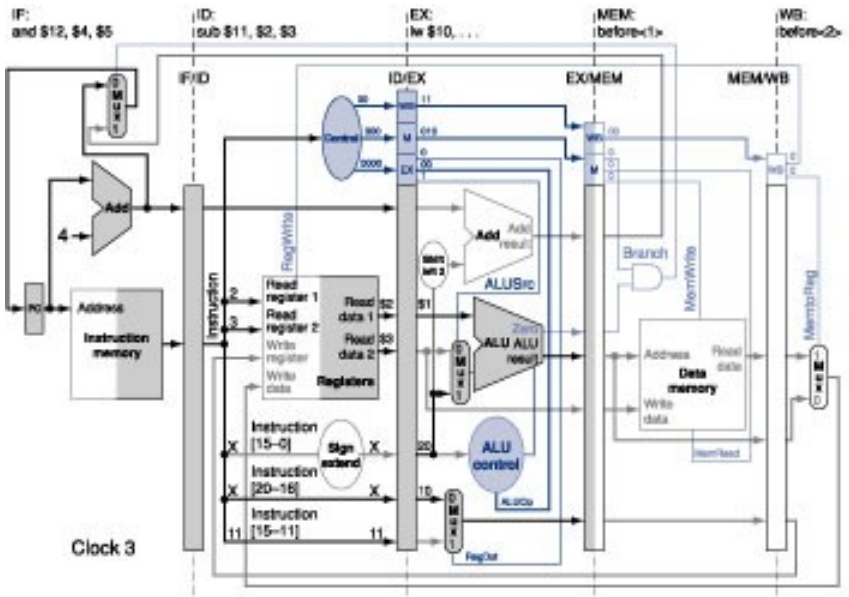


- sub: entra nella pipeline
- lw: in ID/EX scritti $1, 20 (offset) e 10 (numero del registro di destinazione)

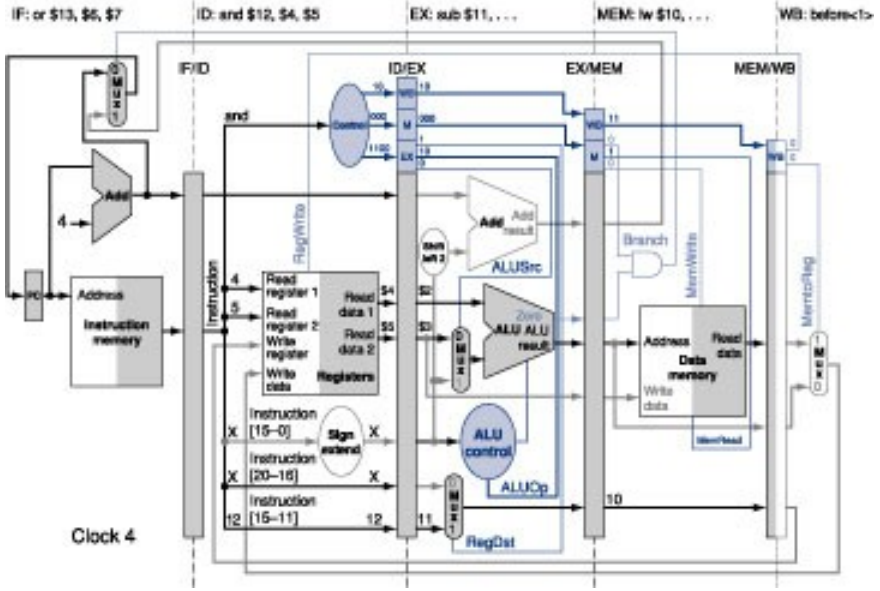


Esempio: cicli di clock 3 e 4

- and: entra nella pipeline
- sub: in ID/EX scritti $2, $3, e 11 (numero del registro di destinazione)
- lw: in EX/MEM scritti $1+20 e 10

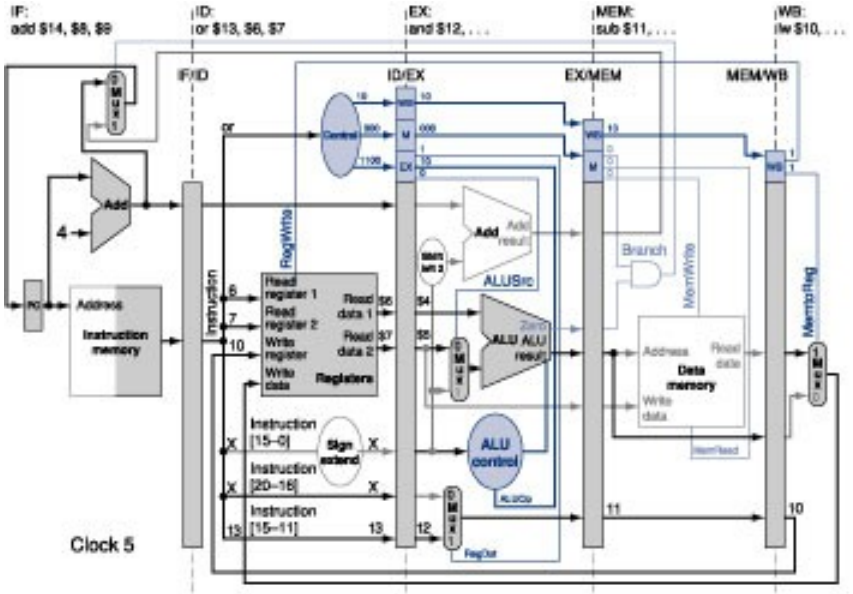


- or: entra nella pipeline
- and: in ID/EX scritti $4, $5, e 12 (numero del registro di destinazione)
- sub: in EX/MEM scritti $2-$3 e 11
- lw: in MEM/WB scritti il valore letto dalla memoria e 10

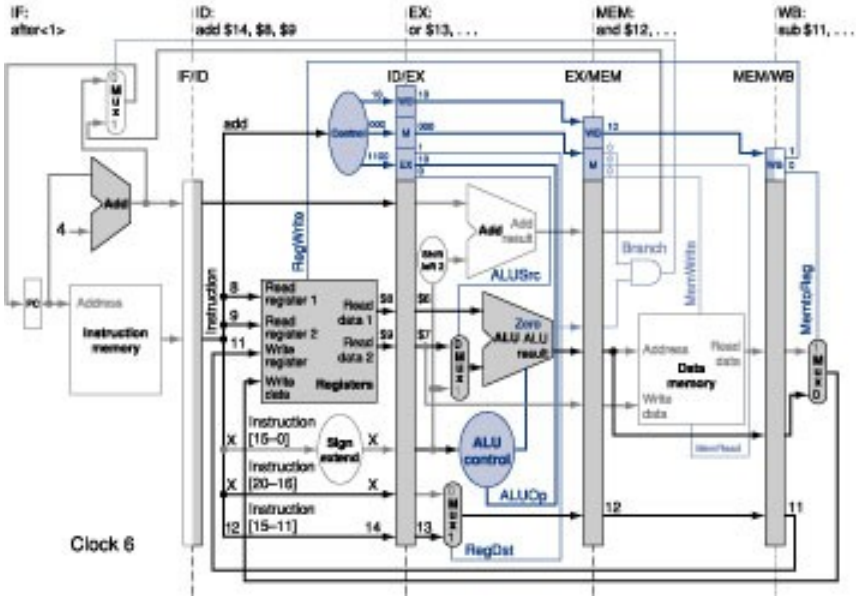


Esempio: cicli di clock 5 e 6

- add: entra nella pipeline
- or: in ID/EX scritti $6, $7, e 13 (numero del registro di destinazione)
- and: in EX/MEM scritti $4 AND $5 e 12
- sub: in MEM/WB scritti $2-$3 e 11
- lw: termina scrivendo $10



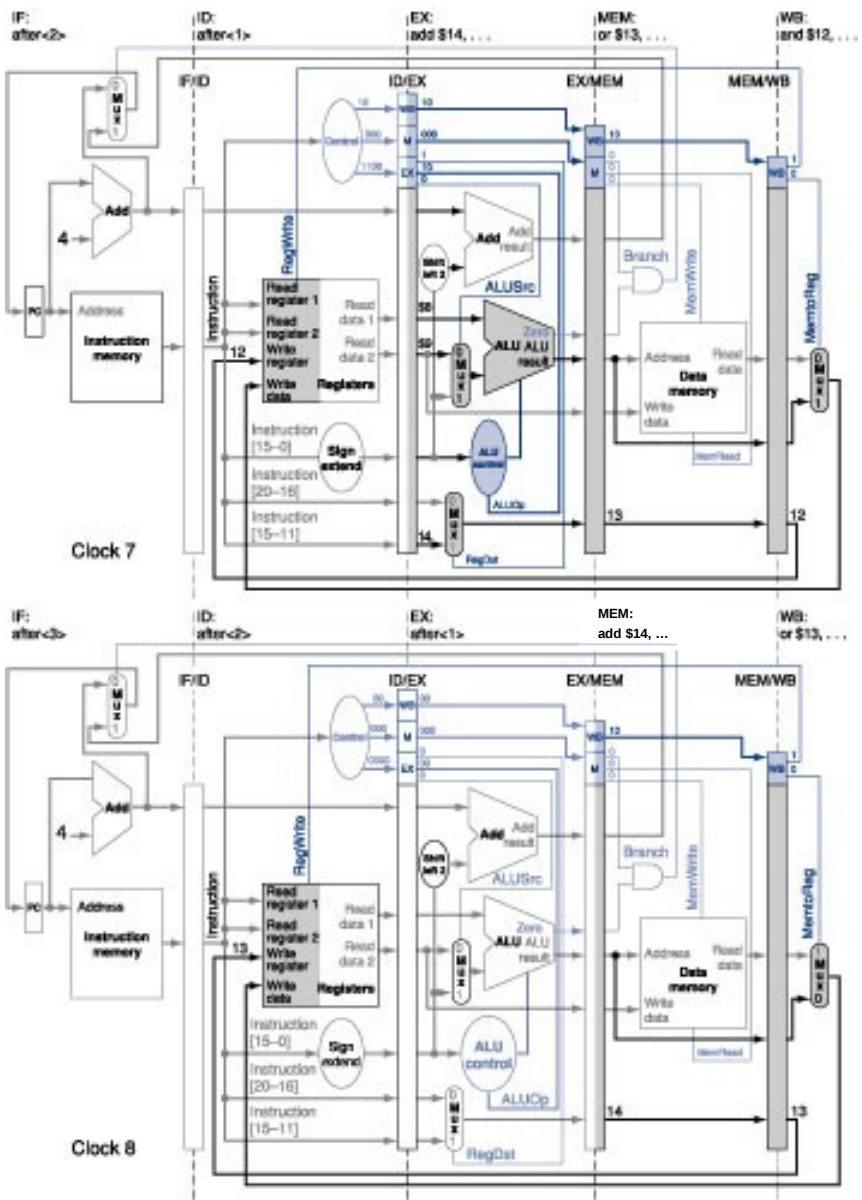
- add: in ID/EX scritti $8, $9, e 14 (numero del registro di destinazione)
- or: in EX/MEM scritti $6 OR $7 e 13
- and: in MEM/WB scritti $4 AND $5 e 12
- sub: termina scrivendo $11



Esempio: cicli di clock 7 e 8

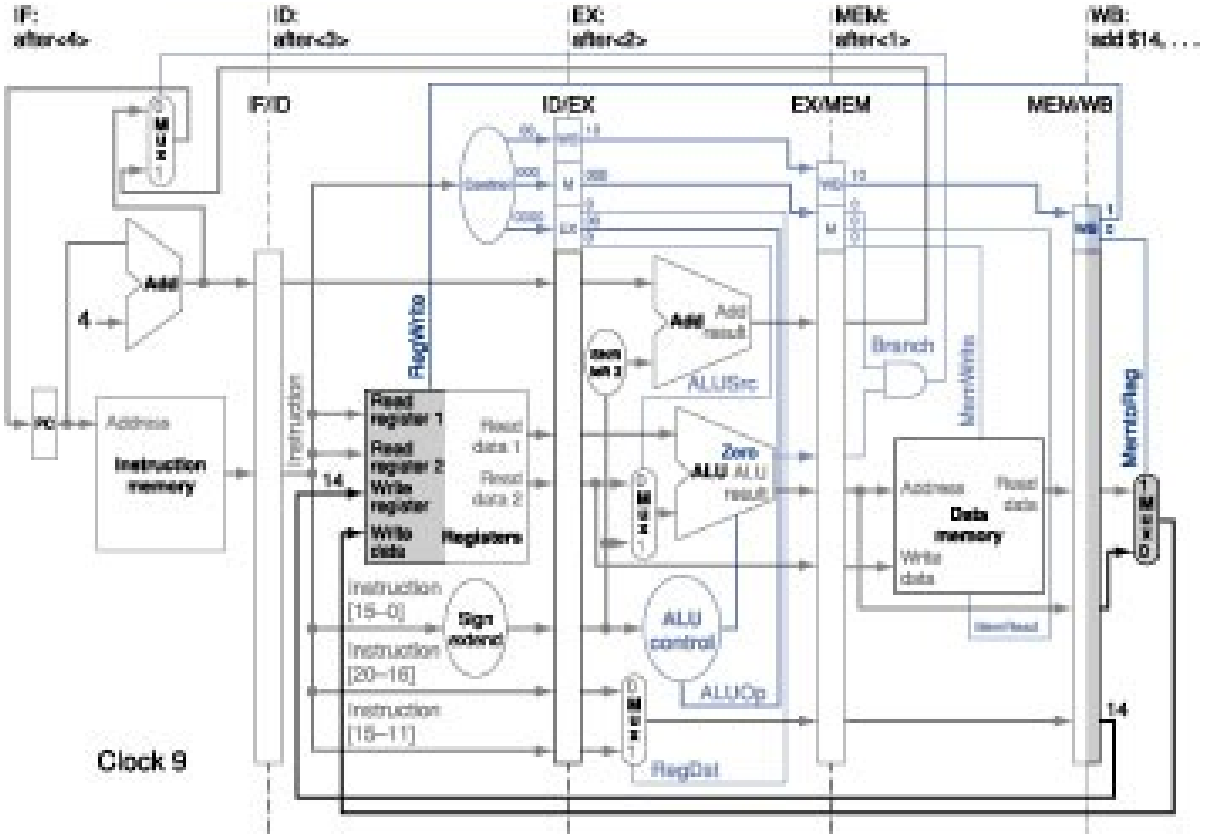
- add: in EX/MEM scritti $8+$9 e 14
- or: in MEM/WB scritti $6 OR $7 e 13
- and: termina scrivendo $12

- add: in MEM/WB scritti $8+$9 e 14
- or: termina scrivendo $13



Esempio: ciclo di clock 9

- add: termina scrivendo $14



Prestazioni del pipelining

- Il pipelining incrementa il throughput del processore (numero di istruzioni completate nell'unità di tempo), ma **non** riduce il tempo di esecuzione (latenza) della singola istruzione
- Anzi, in generale il pipelining aumenta il tempo di esecuzione della singola istruzione, a causa di sbilanciamenti tra gli stadi della pipeline e overhead di controllo della pipeline
  - Lo sbilanciamento tra gli stadi della pipeline riduce le prestazioni
    - Il clock non può essere minore del tempo necessario per lo stadio più lento della pipeline
  - L'overhead della pipeline è causato
    - dai ritardi dei registri di pipeline e dal clock skew (ritardo di propagazione del segnale di clock sui fili)
    - dalla presenza di criticità

Le criticità

- Le *criticità* (o *conflitti* o *alee*) sorgono nelle architetture con pipelining quando non è possibile eseguire un'istruzione nel ciclo immediatamente successivo

- Tre tipi di criticità:

    - Criticità strutturali
    - Criticità sui dati
    - Criticità sul controllo

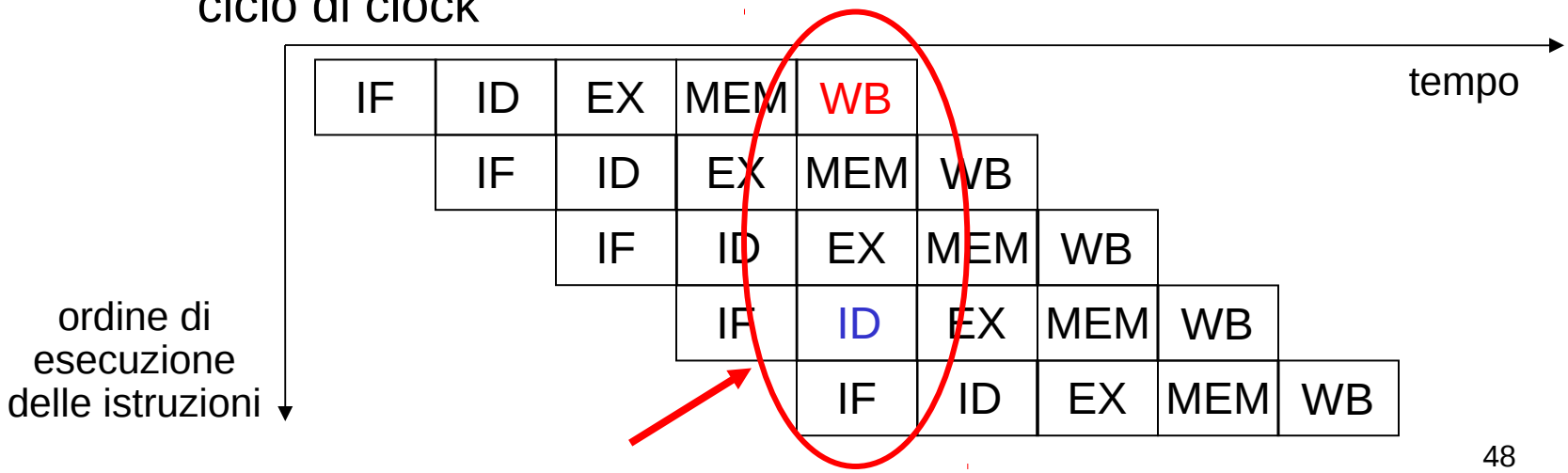
Le criticità (2)

- Criticità **strutturale**
  - Tentativo di usare la stessa risorsa hardware da parte di diverse istruzioni in modi diversi nello stesso ciclo di clock
  - Es.: se nel MIPS avessimo un'unica memoria istruzioni e dati

    oppure un banco dei registri non progettato accuratamente (vedi lucido successivo)
- Criticità **sui dati**
  - Tentativo di usare un risultato prima che sia disponibile
  - Es.: istruzione che dipende dal risultato di un'istruzione precedente che è ancora nella pipeline
- Criticità **sul controllo**
  - Nel caso di salti, decidere quale prossima istruzione da eseguire prima che la condizione sia valutata
  - Es.: istruzioni di salti condizionato: se si sta eseguendo beq, come si fa a sapere (in anticipo) quale è la successiva istruzione da iniziare ad eseguire?

Criticità strutturali

- Nell'architettura MIPS pipeline non abbiamo conflitti strutturali
  - Memoria dati separata dalla memoria istruzioni
  - Banco dei registri progettato per evitare conflitti tra la lettura e la scrittura nello stesso ciclo
    - Soluzione
      - Scrittura del banco dei registri nella prima metà del ciclo di clock
      - Lettura del banco dei registri nella seconda metà del ciclo di clock



Criticità sui dati

- Un'istruzione dipende dal risultato di un'istruzione precedente che è ancora nella pipeline

- Esempio 1:

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

  - Uno degli operandi sorgente di sub ($s0) è prodotto da add, che è ancora nella pipeline
  - Criticità sui dati di tipo *define-use*
- Esempio 2:

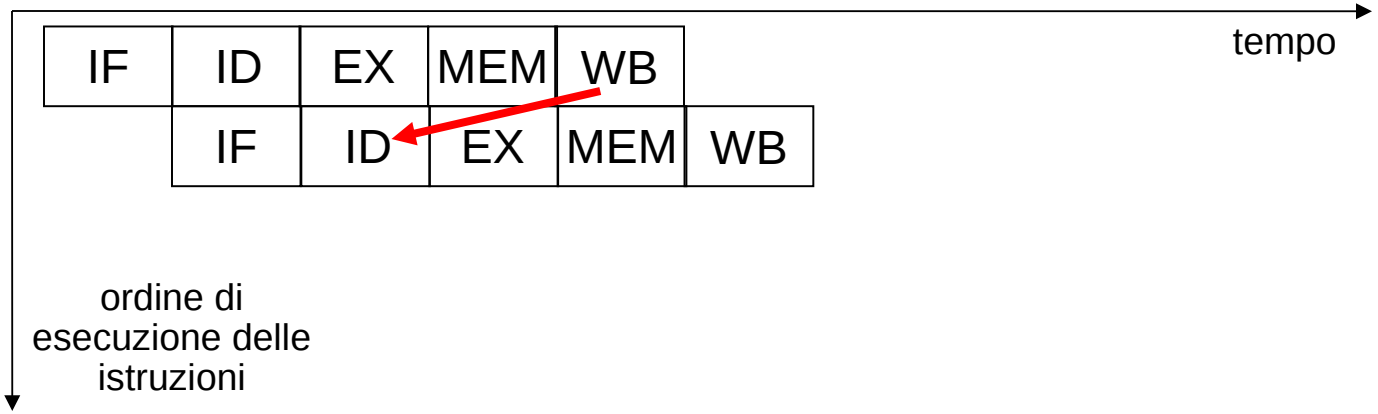
```
lw $s0, 20($t1)
sub $t2, $s0, $t3
```

  - Uno degli operandi sorgente di sub ($s0) è prodotto da lw, che è ancora nella pipeline
  - Criticità sui dati di tipo *load-use*

Criticità sui dati (2)

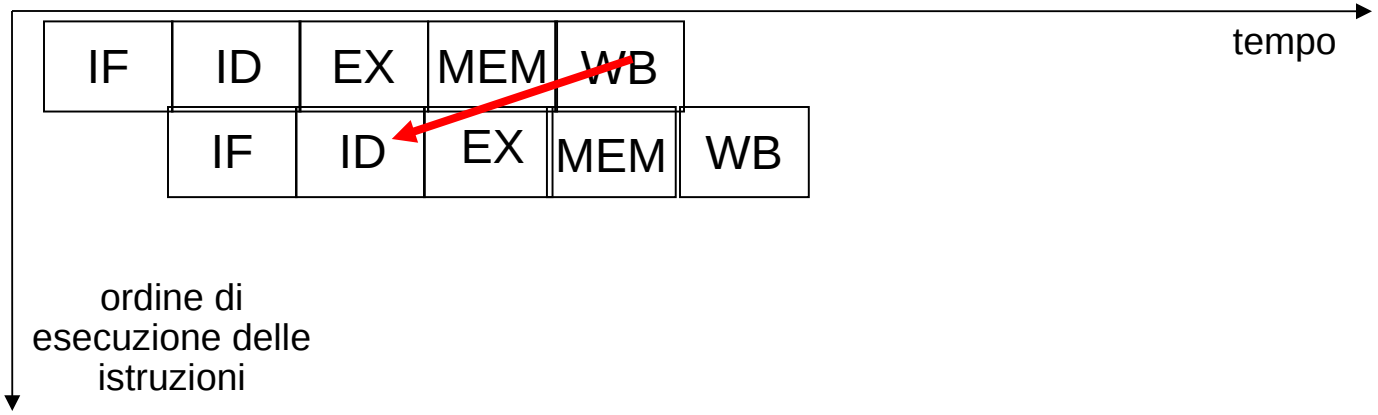
- Esempio 1:

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



- Esempio 2:

lw $s0, 20($t1)
sub $t2, $s0, $t3

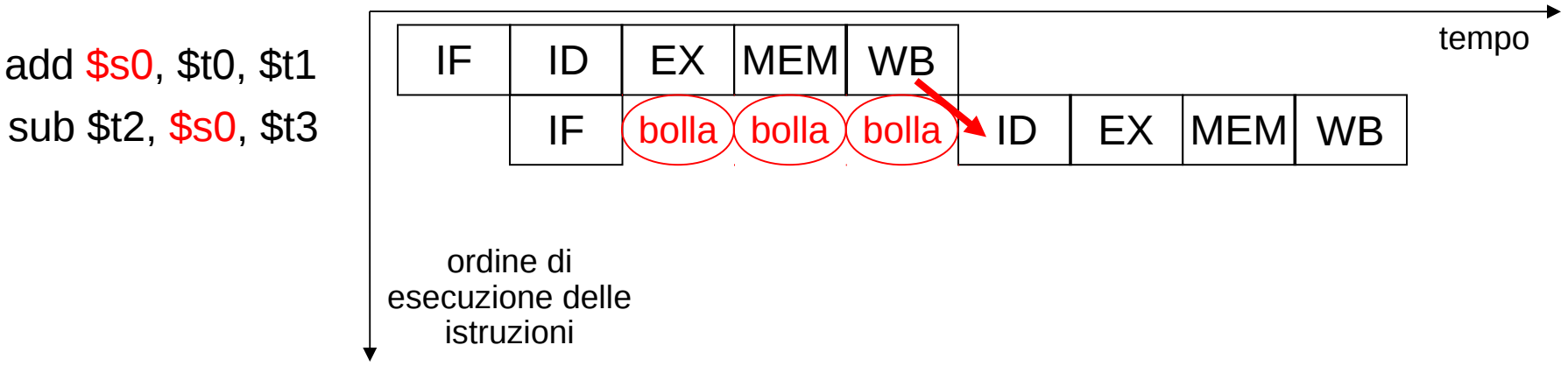


Soluzioni per criticità sui dati

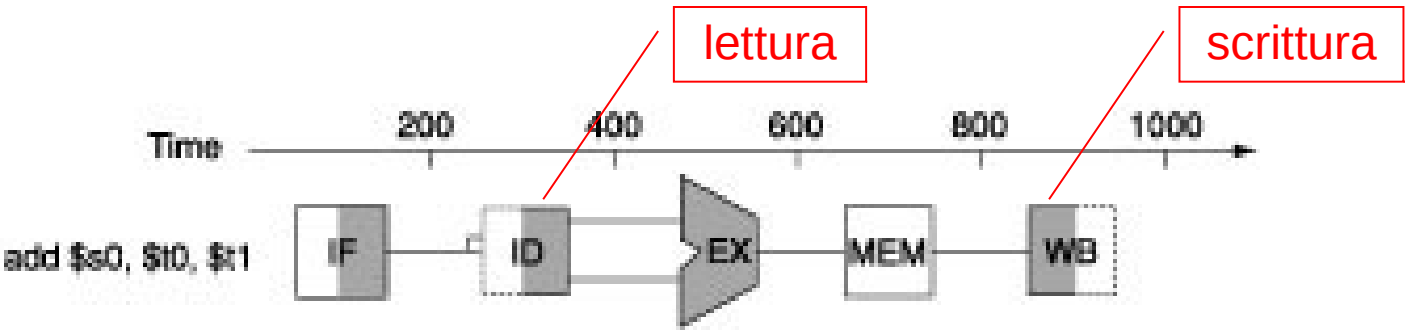
- Soluzioni di tipo hardware
  - Inserimento di bolle (*bubble*) o stalli nella pipeline
    - Si inseriscono dei tempi morti
    - Peggiora il throughput
  - Propagazione o scavalcamento (*forwarding* o bypassing)
    - Si propagano i dati in avanti appena sono disponibili verso le unità che li richiedono
- Soluzioni di tipo software
  - Inserimento di istruzioni *nop* (no operation)
    - Peggiora il throughput
  - Riordino delle istruzioni
    - *Spostare* istruzioni “innocue” in modo che esse eliminino la criticità

Inserimento di bolle

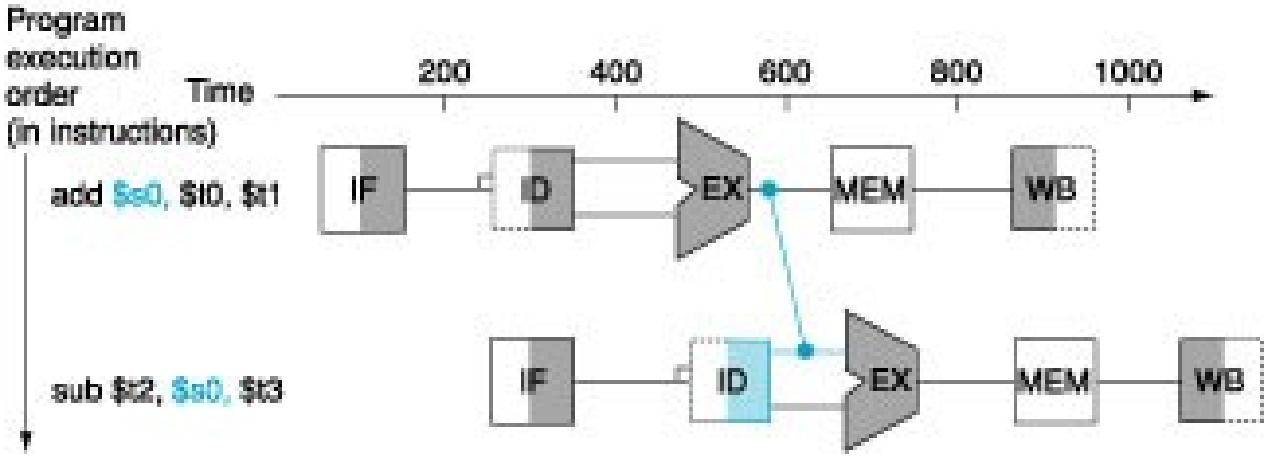
- Si inseriscono delle bolle nella pipeline, ovvero si blocca il flusso di istruzioni nella pipeline finché il conflitto non è risolto
  - *Stallo*: stato in cui si trova il processore quando le istruzioni sono bloccate
- Esempio 1: occorre inserire tre bolle per fermare l'istruzione sub affinché possano essere letti i dati corretti
  - Due bolle se ottimizzazione del banco dei registri



Propagazione (o forwarding)



- Esempio 1: quando la ALU genera il risultato, questo viene *subito* messo a disposizione per il passo dell'istruzione che segue tramite una *propagazione in avanti*



Propagazione e stallo

- Esempio 2:

  lw $s0, 20($t1)

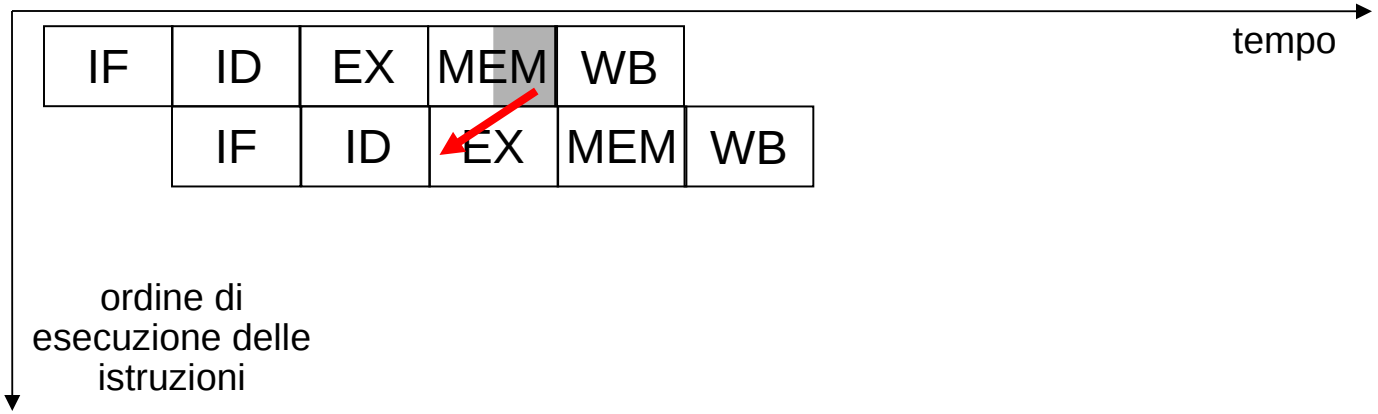
  sub $t2, $s0, $t3

  – E' una criticità sui dati di tipo *load-use*
    - Il dato caricato dall'istruzione di load non è ancora disponibile quando viene richiesto da un'istruzione successiva

- La sola propagazione è insufficiente per risolvere questo tipo di criticità – necessità di almeno una bolla per far completare la lettura del dato

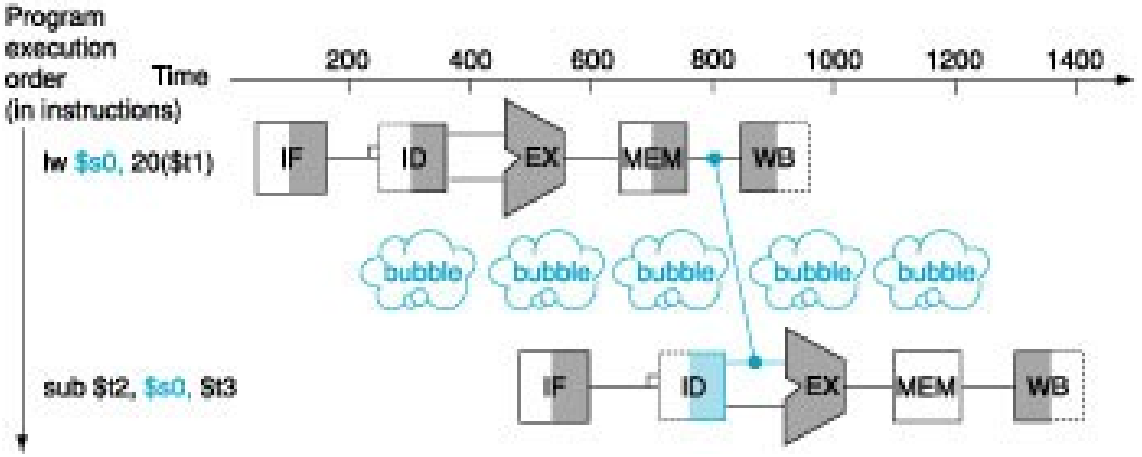
lw $s0, 20($t1)

sub $t2, $s0, $t3



Propagazione e stallo (2)

- Soluzione possibile: propagazione e uno stallo

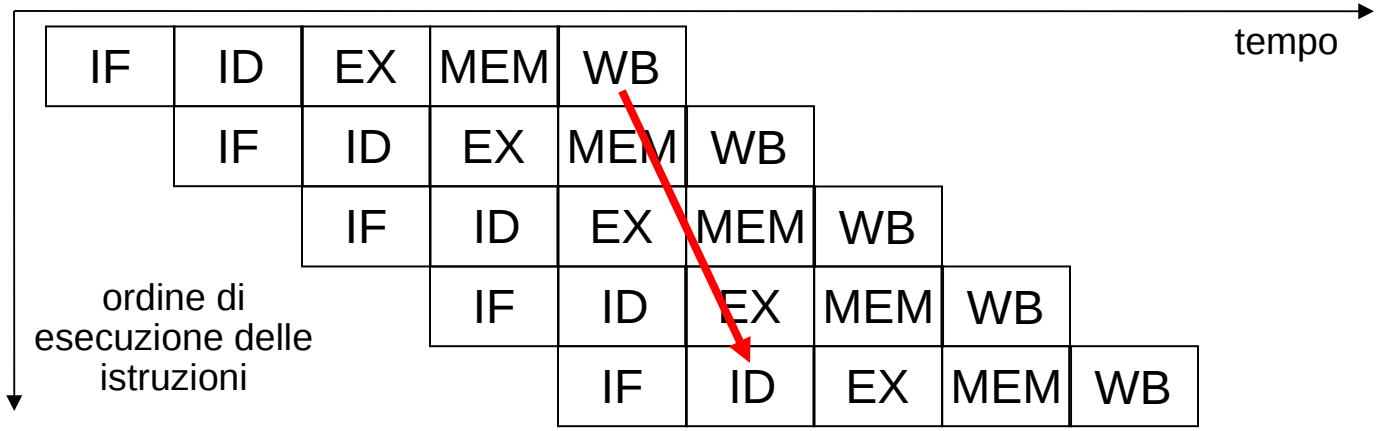


- Senza propagazione e ottimizzazione del banco dei registri, sarebbero stati necessari tre stalli

Inserimento di nop

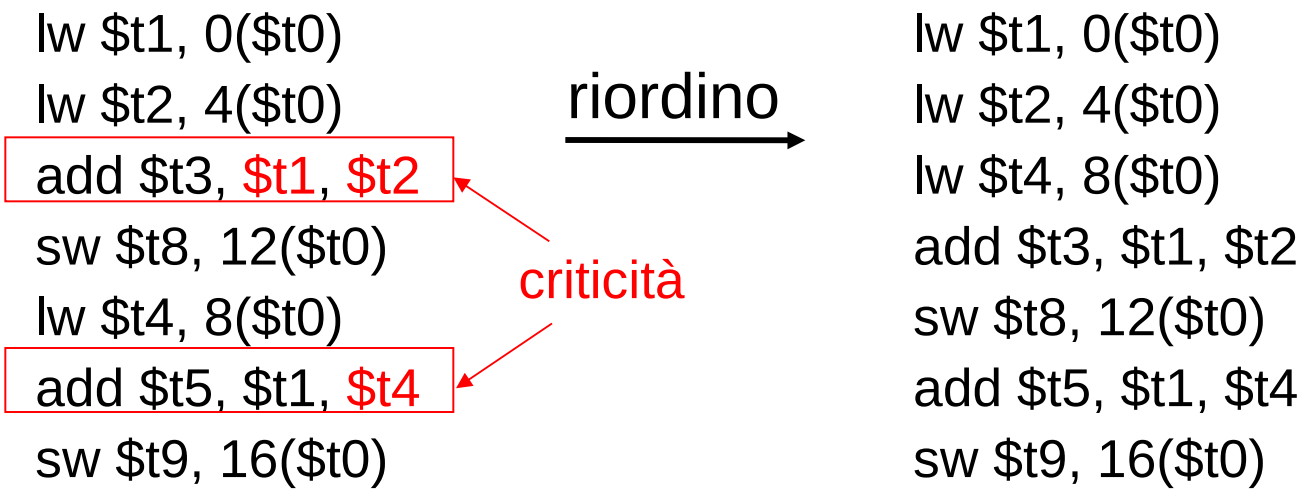
- Esempio 1: l'assemblatore deve inserire tra le istruzioni add e sub tre istruzioni nop, facendo così scomparire il conflitto
  - L'istruzione nop è l'equivalente software dello stallo

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



Riordino delle istruzioni

- L'assemblatore riordina le istruzioni in modo da impedire che istruzioni correlate siano troppo vicine
  - L'assemblatore cerca di inserire tra le istruzioni correlate (che presentano dei conflitti) delle istruzioni *indipendenti* dal risultato delle istruzioni precedenti
  - Quando l'assemblatore non riesce a trovare istruzioni indipendenti deve inserire istruzioni nop
- Esempio:



  - La propagazione permette di risolvere i conflitti rimanenti dopo il riordino

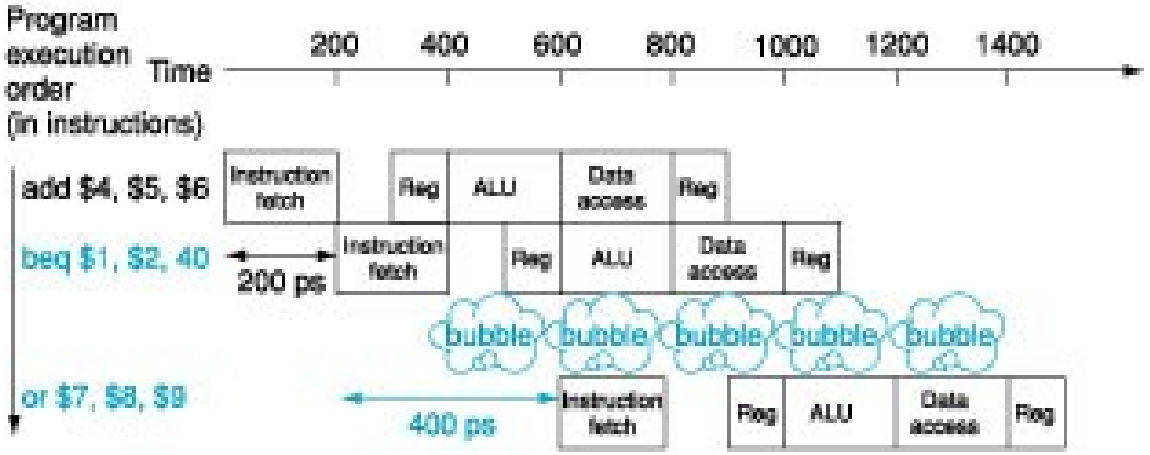
Criticità sul controllo

- Per alimentare la pipeline occorre inserire un'istruzione ad ogni ciclo di clock
- Tuttavia, nel processore MIPS la decisione sul salto condizionato non viene presa fino al quarto passo (MEM) dell'istruzione beq

- Comportamento desiderato del salto
  - Se il confronto fallisce, continuare l'esecuzione con l'istruzione successiva a beq
  - Se il confronto è verificato,non eseguire le istruzioni successive alla beq e saltare all'indirizzo specificato

Soluzioni per criticità sul controllo

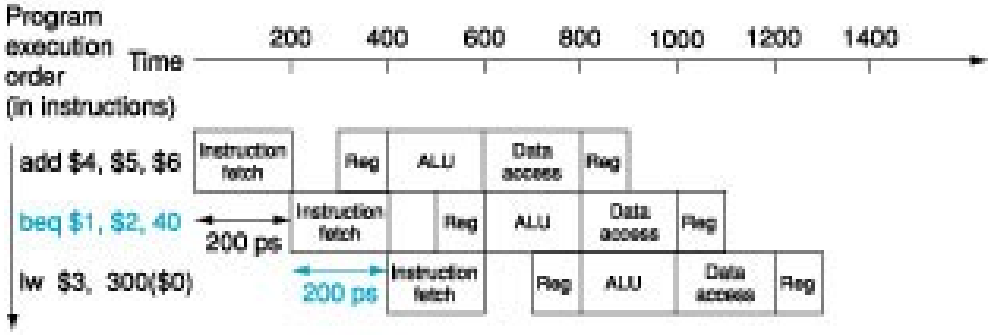
- Inserimento di bolle
  - Si blocca la pipeline finché non è noto il risultato del confronto della beq e si sa quale è la prossima istruzione da eseguire
    - Nel MIPS il risultato del confronto è noto al quarto passo: occorre inserire tre stalli
- Anticipazione del confronto al secondo passo (ID)
  - Si aggiunge dell'hardware extra: dopo aver decodificato l'istruzione, si può decidere e modificare il PC se necessario
  - Occorre comunque aggiungere uno stallo prima dell'istruzione successiva alla beq



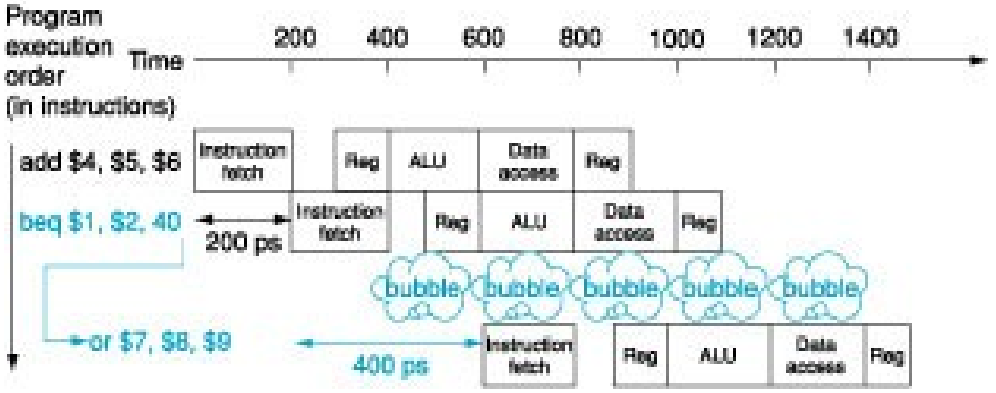
Soluzioni per criticità sul controllo (2)

- Predizione del salto
  - Tecniche di predizione statica
    - Es.: si predice che il salto non sia eseguito (*untaken branch*)
  - Tecniche di predizione dinamica

Salto non eseguito



Salto eseguito



Soluzioni per criticità sul controllo (3)

- Salto ritardato (delayed branch)
  - In ogni caso (indipendentemente dal risultato del confronto) viene eseguita l'istruzione che segue immediatamente il salto (definita *branch-delay slot*), istruzione scelta in modo che non cambi la semantica del programma
  - Caso peggiore
    - Inserimento di nop
  - Caso migliore
    - E' possibile trovare un'istruzione precedente al salto che possa essere posticipata al salto *senza* alterare il flusso di controllo (e dei dati)
  - Esempio

```
        or $t0, $t1, $t2                          add $s0, $s1, $s2
        add $s0, $s1, $s2     delayed branch      sub $s3, $s4, $s5
        sub $s3, $s4, $s5     ------------>       beq $s0, $s3, Exit
        beq $s0, $s3, Exit                        or $t0, $t1, $t2
        xor $t2, $s0, $t3                         xor $t2, $s0, $t3
        …                                         …
Exit:                                     Exit:
```