Architetture dei Calcolatori

Introduzione al Pipelining

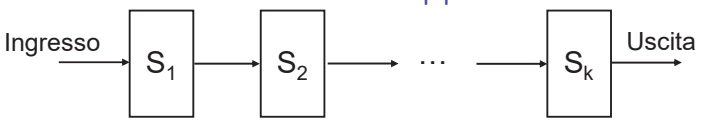
Prof. Francesco Lo Presti

Definizione di Pipelining

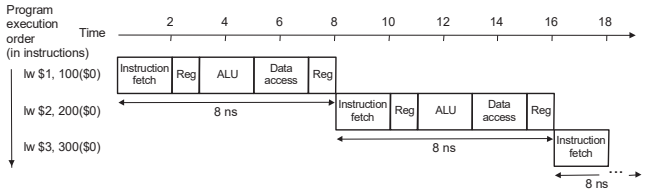
- E' una tecnica
  - per migliorare le prestazioni del processore
  - basata sulla sovrapposizione dell'esecuzione di più istruzioni
- Analogia con catena di montaggio

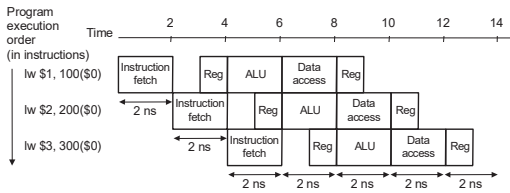
Idea base

- 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



Confronto fra ciclo singolo e pipeline



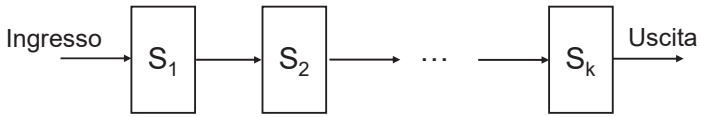


Osservazioni sul pipelining

- 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

Stadi della pipeline

- Gli stadi della pipeline sono collegati in sequenza
  - Gli stadi 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.: 2ns per l' esecuzione dell' operazione più lenta
  - Per alcune istruzioni, alcuni stadi sono cicli sprecati
- Obiettivo dei progettisti: bilanciare la lunghezza degli stadi

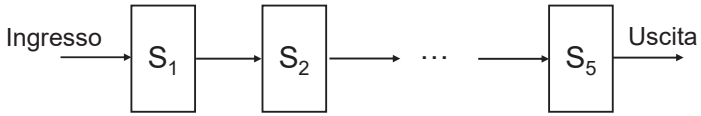


L' insieme di istruzioni MIPS ed il pipelining

- 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 (solo 3) con simmetria 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
    - ✓ Possibile usare un solo stadio per trasferire dati tra processore e memoria
  - Ogni istruzione MIPS scrive al più un risultato e lo fa verso la fine della pipeline

Unità di Elaborazione con Pipeline

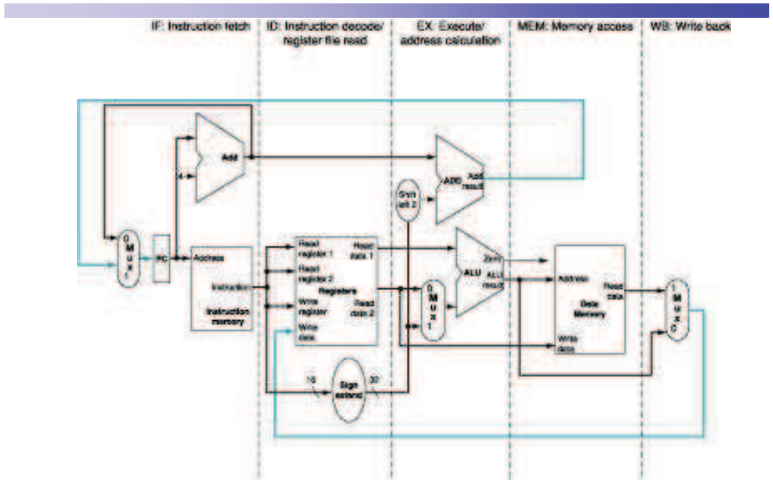
- Cinque passi/stadi di esecuzione di un istruzione
  - IF – Prelievo ed incremento PC
  - ID – Decodifica e lettura registri
  - EX – Esecuzione/Calcolo indirizzo
  - MEM – Accesso in Memoria
  - WB – Scrittura banco registri



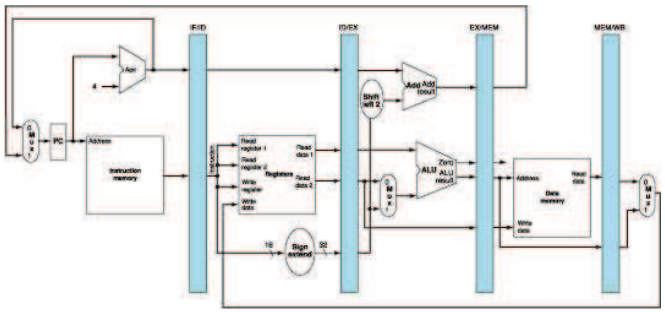
Come progettare l' unità di elaborazione?

- 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
- La suddivisione dell' istruzione in 5 stadi implica che in ogni ciclo di clock siano in esecuzione (al più) 5 istruzioni
- Diverse istruzioni in esecuzione nello stesso istante possono richiedono risorse hardware simili
  - Replicazione delle risorse hardware
- Struttura base del processore a singolo ciclo
  - ...a cui dobbiamo aggiungere dei registri per passare dati da uno stadio al successivo
- +Registri di pipeline

L' unità di elaborazione a ciclo singolo



L' unità di elaborazione con pipeline

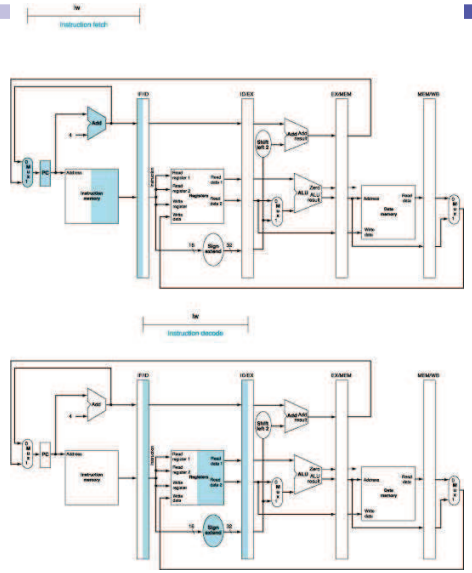


- Introduzione di *registri di pipeline* (registri interstadio)
  - Ad ogni ciclo di clock le informazioni procedono da un registro di pipeline a quello successivo
  - Il nome del registro è dato dal nome dei due stadi che separa
    - ✓ Registro IF/ID (Instruction Fetch / Instruction Decode)
    - ✓ Registro ID/EX (Instruction Decode / EXecute)
    - ✓ Registro EX/MEM (Execute / MEMory access)
    - ✓ Registro MEM/WB (MEMory access / Write Back)

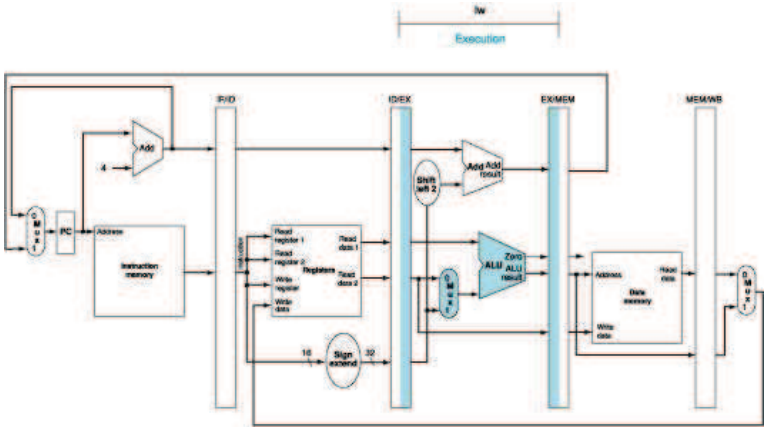
Uso dell' unità con pipeline

- 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

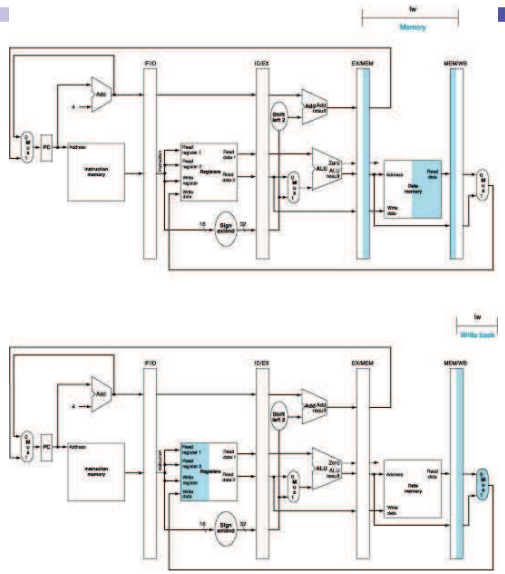
Esecuzione di lw: primo e secondo stadio



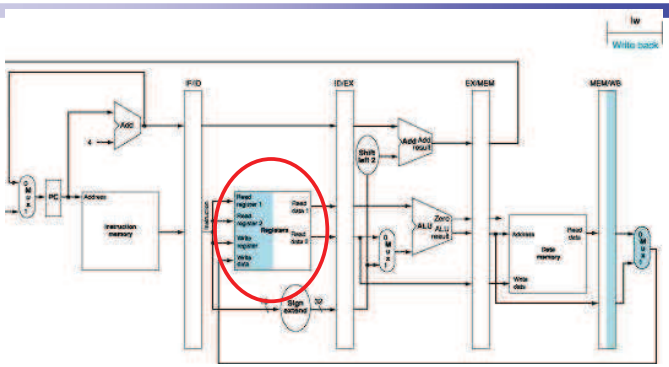
Esecuzione di lw: terzo stadio



Esecuzione di lw: quarto e quinto stadio

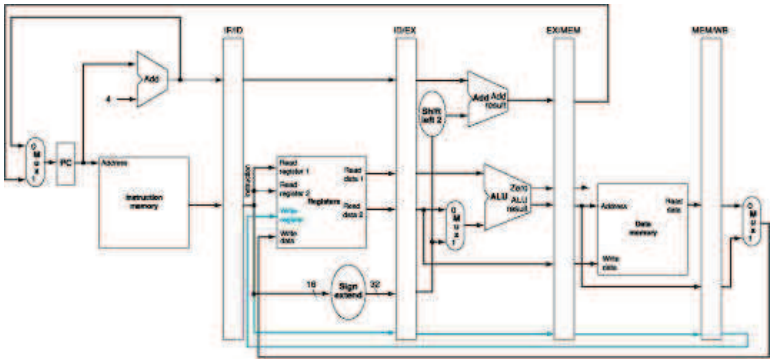


Caccia all' errore...



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

Soluzione



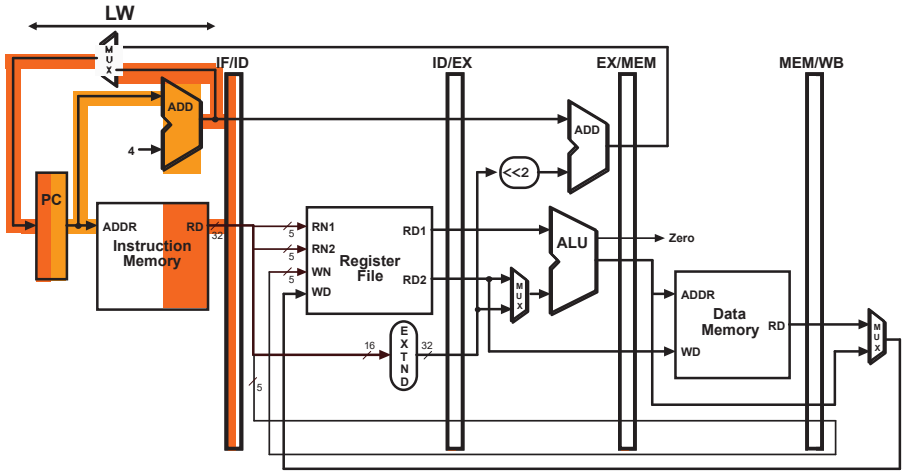
- Il numero del registro di destinazione viene scritto nei registri di pipeline:
  - Prima in ID/EX, poi in EX/MEM, infine in MEM/WB

Esempio con più istruzioni

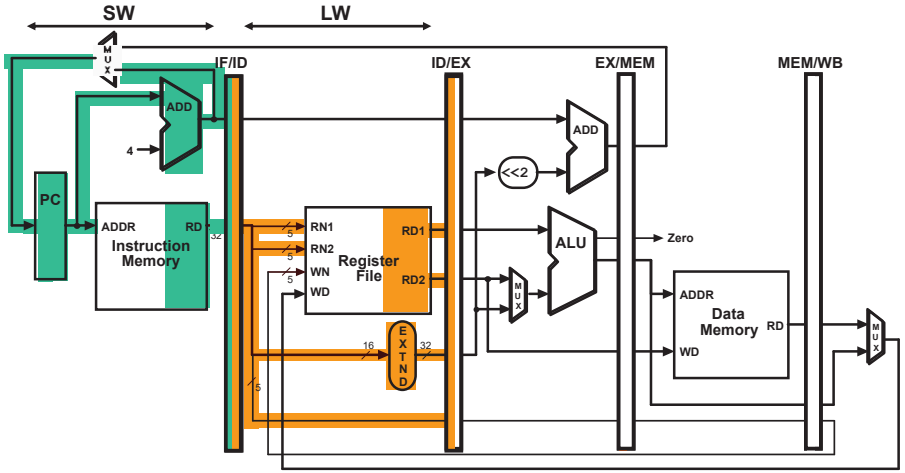
- Consideriamo la sequanza di istruzioni MIPS:

```
lw  $t0,  10($t1)
sw  $t3, 20($t4)
add $t5,  $t6,  $t7
sub $t8,  $t9,  $t10
```

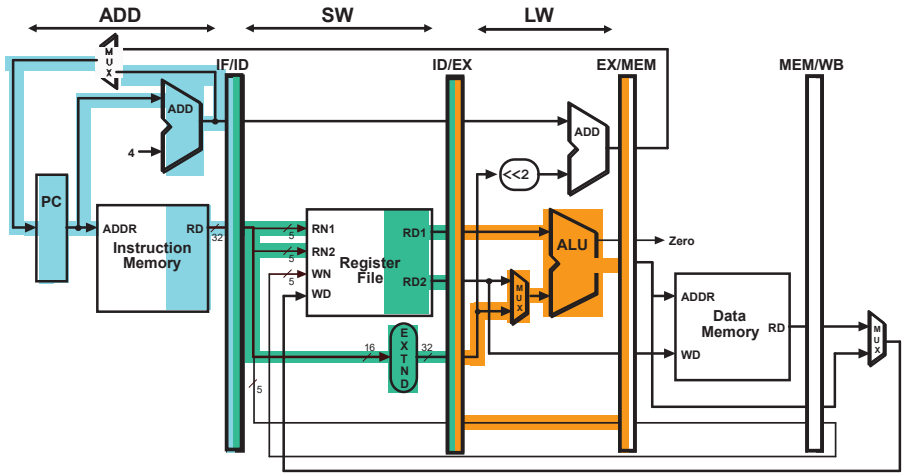
Primo ciclo di clock



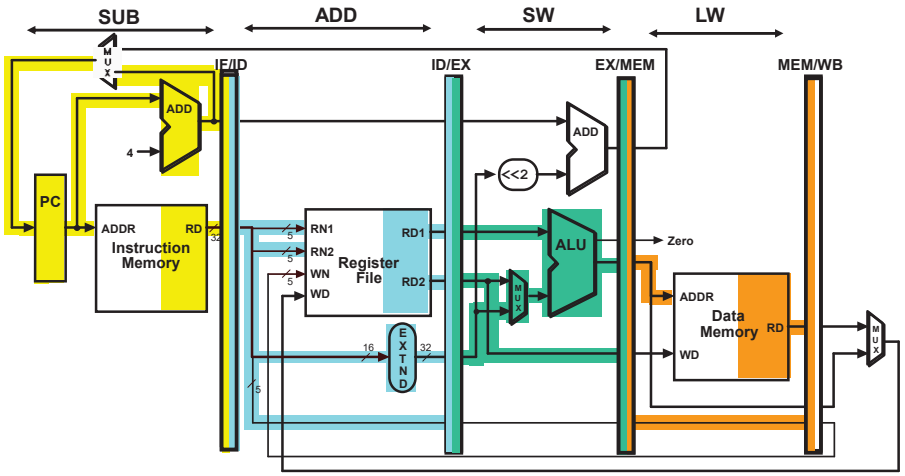
Secondo ciclo di clock



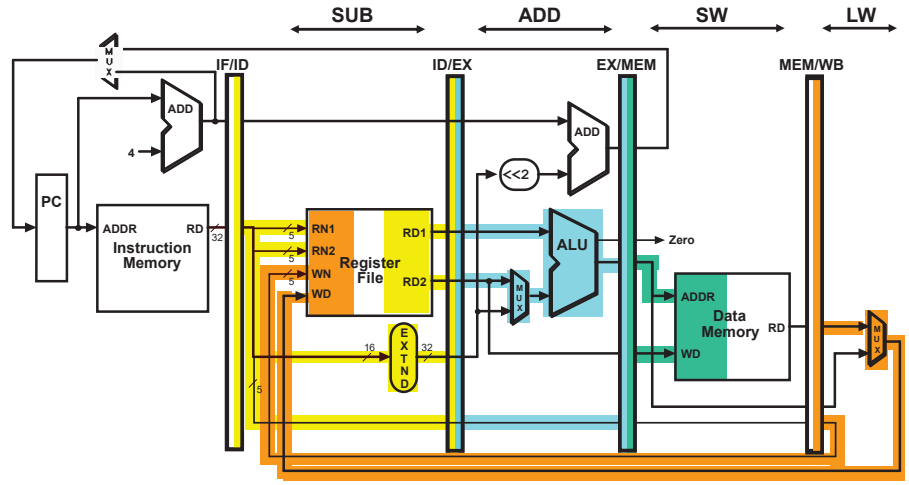
Terzo ciclo di clock



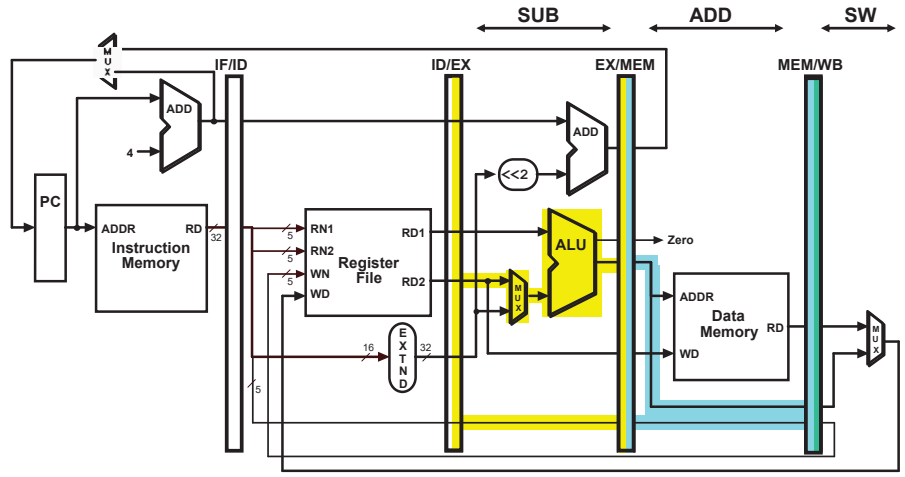
Quarto ciclo di clock



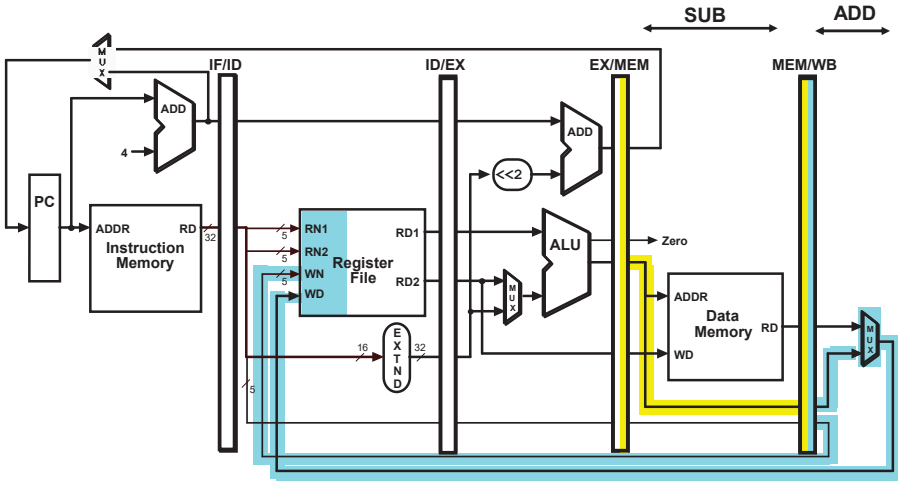
Quinto ciclo di clock



Sesto ciclo di clock



Settimo ciclo di clock



Ottavo ciclo di clock

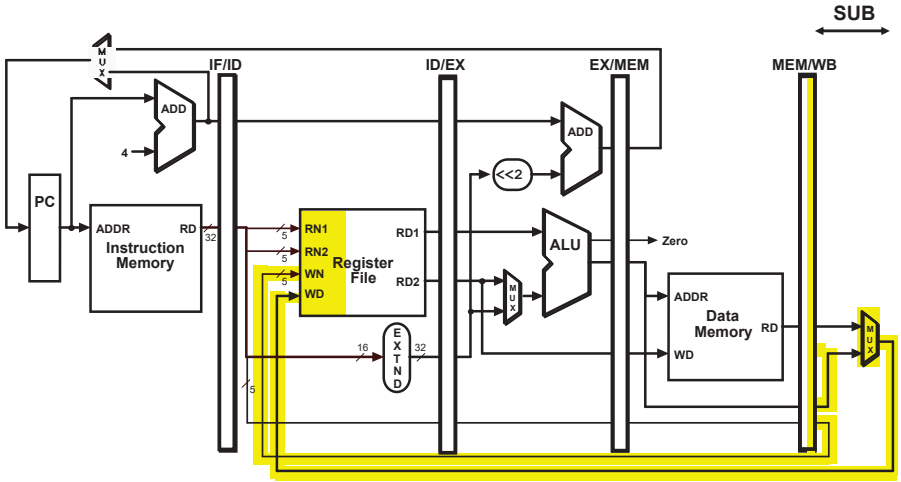
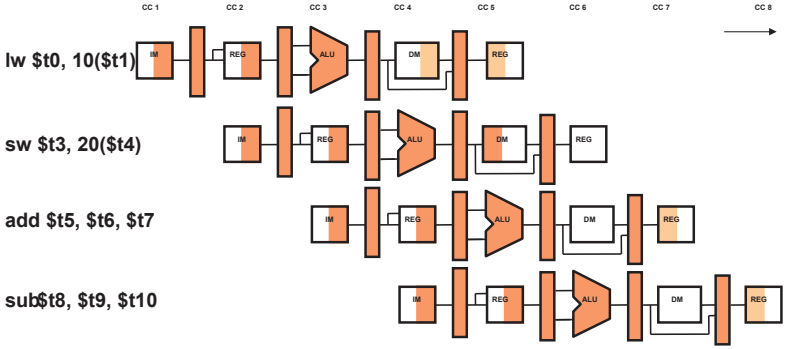


Diagramma Alternativo



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	MEM	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	

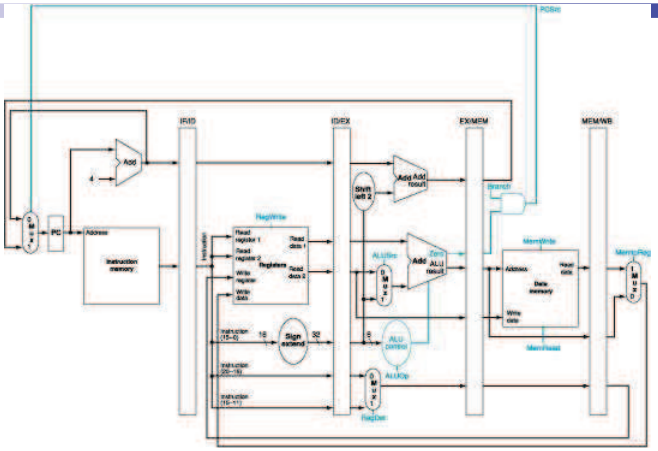
• Istruzioni di beq

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

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 devono viaggiare con l' istruzione

I segnali di controllo



- Non sono necessari segnali di controllo per la scrittura dei registri di pipeline

I segnali di controllo (2)

- Raggruppiamo i segnali di controllo in base agli stadi della pipeline

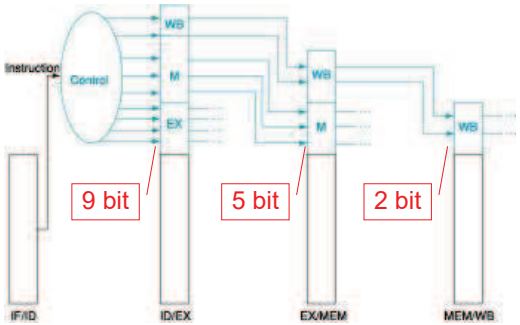
- Prelievo dell' istruzione
  - Identico per tutte le istruzioni
- Decodifica dell' istruzione/lettura del banco dei registri
  - Identico per tutte le istruzioni
- Esecuzione/calcolo dell' indirizzo
  - RegDst, ALUOp, ALUSrc
- Accesso alla memoria
  - Branch, MemRead,MemWrite
- Scrittura del risultato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

I segnali di controllo (3)

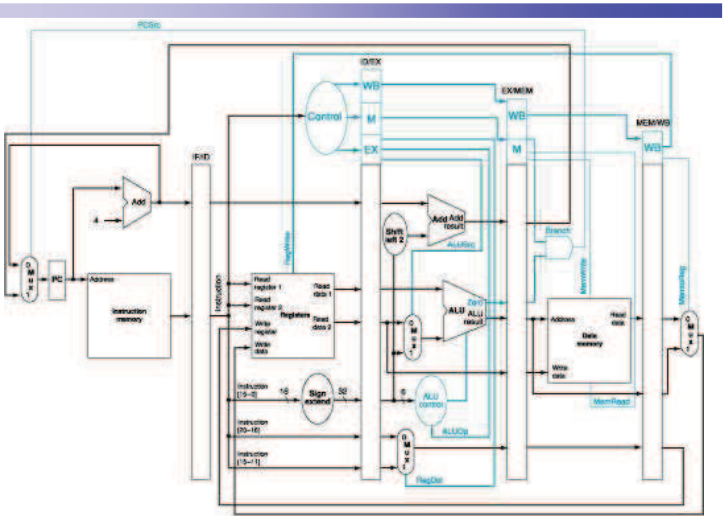
Istruzione	Segnali di controllo EX				Segnali di controllo MEM			Segnali di controllo WB	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Memto Reg
tipo-R	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Estensione con controllo

- I registri di pipeline contengono anche i valori dei segnali di controllo
  - Al massimo 8 segnali di controllo (9 bit)
- I valori necessari per lo stadio successivo vengono propagati dal registro di pipeline corrente al successivo



Estensione con controllo (2)



Le criticità

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

- Tre tipi di criticità
  - Criticità strutturali
  - Criticità sui dati
  - Criticità sul controllo

Le criticità (2)

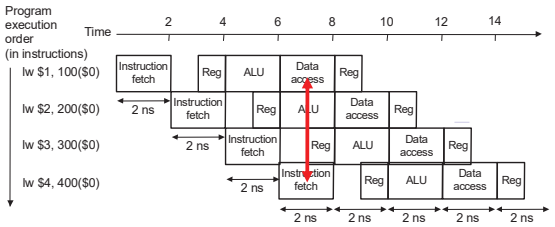
- Criticità **strutturale**
  - Istruzioni in fasi diverse della pipeline necessitano delle stesse risorse hardware
  - Es.: se nel MIPS avessimo un' unica memoria istruzioni e dati

- 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**
  - Tentativo di prendere una decisione sulla 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?

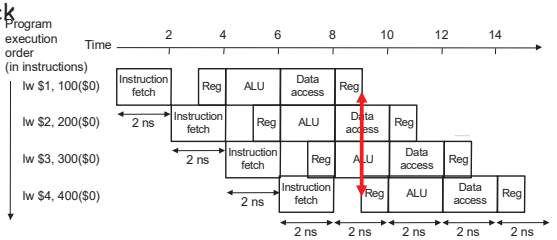
Criticità strutturali

- es: Memoria unica dati+istruzioni
  - criticità strutturale tra la prima e la quarta istruzione



Criticità strutturali

- Banco dei registri usato nello stesso ciclo di pipeline facendo un accesso in lettura da parte di un' istruzione ed un accesso in scrittura da parte di un' altra istruzione
  - Soluzione per evitare un' alea di tipo *Read After Write*
    - ✓ 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:

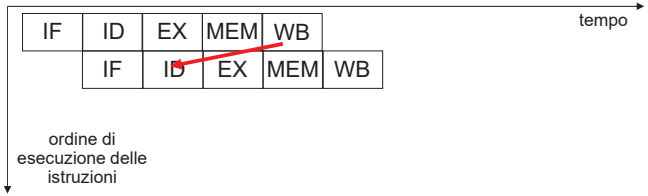
```
lw $s0, 20($t1)
sub $t2, $s0, $t3
```

  - 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)

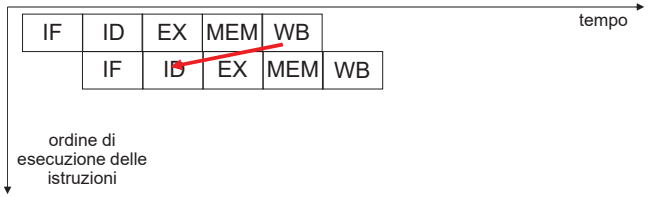
- Esempio 1:

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



- Esempio 2:

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

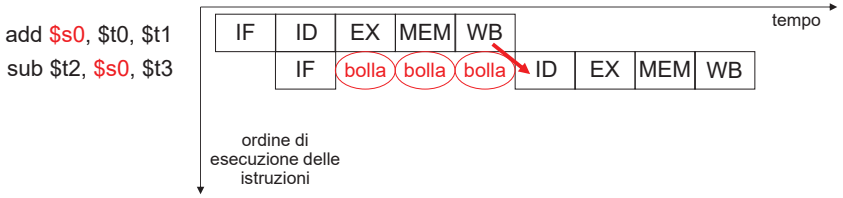


Soluzioni per criticità sui dati

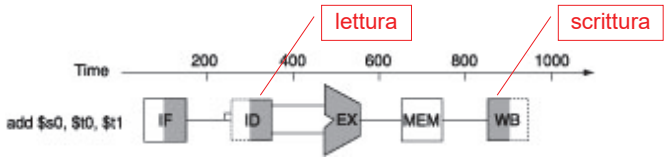
- Soluzioni di tipo hardware
  - Inserimento di bolle (*bubble*) o stalli nella pipeline
    - ✓ Si inseriscono dei tempi morti
    - ✓ Peggiora il throughput
  - Propagazione o scavalcamento (*forwarding* o bypassing)
    - ✓ Si propagano i dati in avanti appena sono disponibili verso le unità che li 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

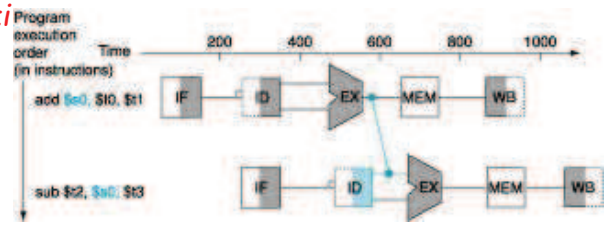
- Si inseriscono delle bolle nella pipeline, ovvero si blocca il flusso di istruzioni nella pipeline finché il conflitto non è risolto
  - *Stallo*: stato in cui si trova il processore quando le istruzioni sono bloccate
- Esempio 1: occorre inserire tre bolle per fermare l'istruzione sub affinché possano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



Propagazione (o forwarding)



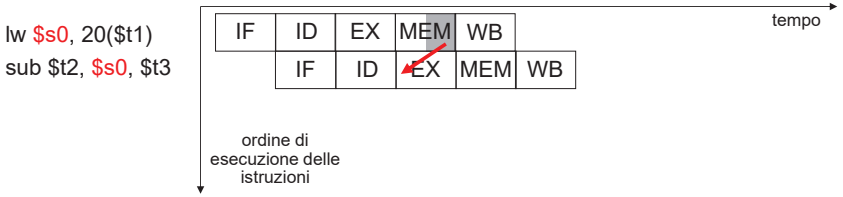
- Esempio 1: quando la ALU genera il risultato, questo viene *subito* messo a disposizione per il passo dell'istruzione che segue tramite una *propagazione in avanti*



Propagazione e stallo

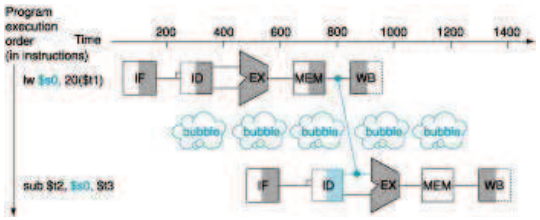
- Esempio 2:

  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



Propagazione e stallo (2)

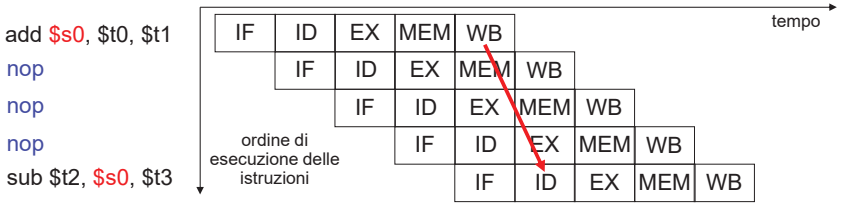
- Soluzione possibile: propagazione e uno stallo



- 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



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:

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

criticità

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

Tipi di criticità sui dati

- Consideriamo due istruzioni `I` e `J`, con `I` che precede `J` nell'esecuzione
- In generale, le criticità sui dati sono di tre tipi:
  - RAW (**Read After Write**)
  - WAR (**Write After Read**)
  - WAW (**Write After Write**)

- Criticità RAW
  - L'istruzione `J` prova a leggere un dato sorgente prima che l'istruzione `I` lo abbia scritto
    - ✓ L'istruzione `J` legge un dato sbagliato

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

  - Presente nella pipeline MIPS

Tipi di criticità sui dati (2)

- Criticità WAR
  - L'istruzione J prova a scrivere un dato destinazione prima che l'istruzione I lo abbia letto
    - ✓ L'istruzione I legge un dato sbagliato

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

  - Assente nella pipeline MIPS
    - ✓ Tutte le istruzioni richiedono 5 stadi
    - ✓ Le operazioni di lettura dei registri avvengono sempre nello stadio ID
    - ✓ Le operazioni di scrittura dei registri avvengono sempre nello stadio WB

Tipi di criticità sui dati (2)

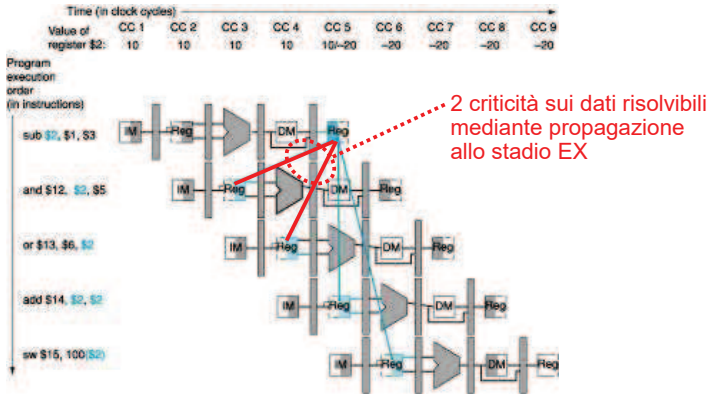
- Criticità WAW
  - L'istruzione J prova a scrivere un dato destinazione prima che l'istruzione I lo abbia scritto
    - ✓ Le operazioni di scrittura vengono eseguite nell'ordine sbagliato

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

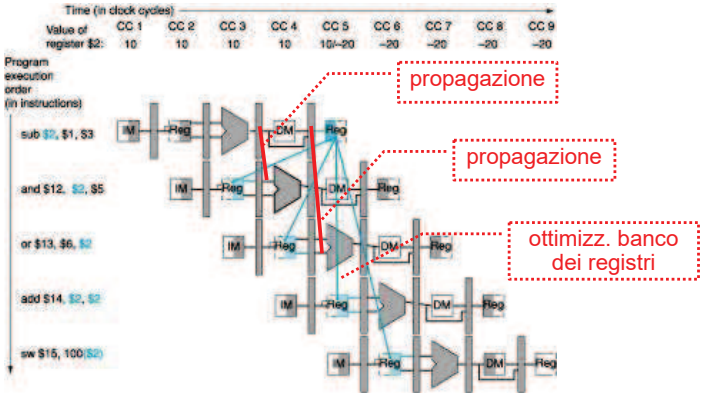
  - Assente nella pipeline MIPS
    - ✓ Tutte le istruzioni richiedono 5 stadi
    - ✓ Le operazioni di scrittura dei registri avvengono sempre nello stadio finale WB

Criticità sui dati

- Consideriamo una sequenza di 5 istruzioni



Soluzione con propagazione



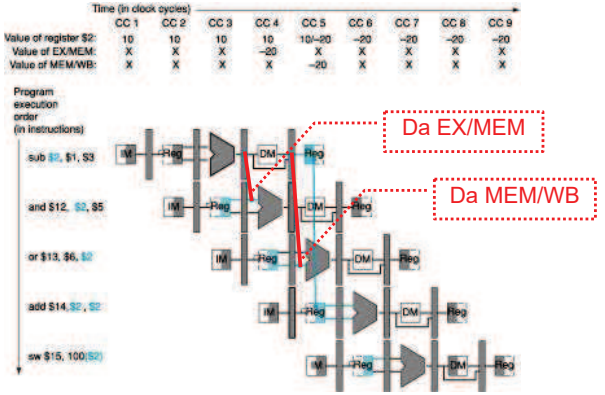
Soluzione con propagazione (2)

- Consideriamo la prima criticità (EX):
  sub $2, $1, $3 e and $12, $2, $5
  - Il dato prodotto dall'istruzione sub è disponibile alla fine dello stadio EX (*CC* 3)
  - Il dato è richiesto dall'istruzione and all'inizio dello stadio EX (*CC* 4)
  - La criticità può essere rilevata quando l'istruzione and si trova nello stadio EX e l'istruzione sub si trova nello stadio MEM

- Consideriamo la seconda criticità (MEM):
  sub $2, $1, $3 e or $13, $6, $2
  - Il dato prodotto dall'istruzione sub è disponibile alla fine dello stadio EX (*CC* 3)
  - Il dato è richiesto dall'istruzione or all'inizio dello stadio EX (*CC* 5)
  - La criticità può essere rilevata quando l'istruzione or si trova nello stadio EX e l'istruzione sub si trova nello stadio WB

Propagazione dai registri di pipeline

- Gli ingressi alla ALU sono forniti dai registri di pipeline anziché dal banco dei registri
  - In questo modo le dipendenze sono in avanti nel tempo



Riconoscimento della criticità sui dati

- Usiamo la notazione:
  - *NomeRegistroPipeline.CampoRegistro*
- Condizioni che generano la criticità sui dati
  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

- Consideriamo la prima criticità (EX):
  sub $2, $1, $3 e and $12, $2, $5
  - E' verificata la condizione 1a
    EX/MEM.RegisterRd = ID/EX.RegisterRs = $2
- Consideriamo la seconda criticità (MEM):
  sub $2, $1, $3 e or $13, $6, $2
  - E' verificata la condizione 2b
    MEM/WB.RegisterRd = ID/EX.RegisterRt = $2

Riconoscimento della criticità sui dati (2)

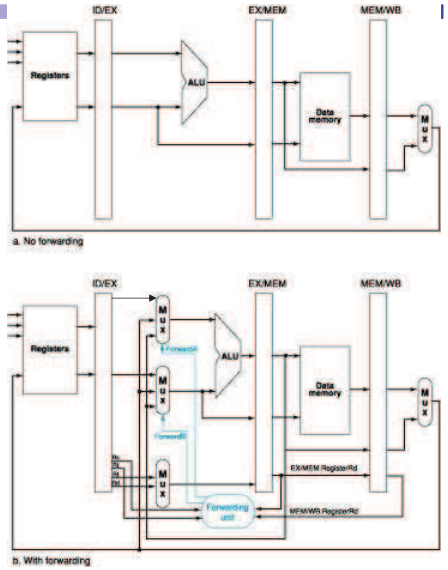
- Per evitare propagazioni inutili raffiniamo le condizioni
  - Non tutte le istruzioni scrivono un registro
    - Controlliamo se RegWrite è asserito
  - Il registro $0 come destinazione non richiede propagazione
    - Aggiungiamo EX/MEM.RegisterRd0 e MEM/WB.RegisterRd ≠ 0

- Quindi le condizioni divengono:
  1a. EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs)
  1b. EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)
  2a. MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs)

Hardware per la propagazione

- ALU e registri di pipeline senza propagazione

- ALU e registri di pipeline con propagazione
  - ***Unità di propagazione*** *(forwarding unit)*: assegna un valore ai segnali di controllo ForwardA e ForwardB per i due mux davanti alla ALU
  - In ID/EX salvato anche Instruction[25-21] (numero del registro sorgente rs)



Segnali di controllo per la propagazione

Controllo MUX	Sorgente	Significato
ForwardA = 00	ID/EX	Primo operando della ALU dal banco dei registri
ForwardA = 10	EX/MEM	Primo operando della ALU propagato dal precedente risultato della ALU
ForwardA = 01	MEM/WB	Primo operando della ALU propagato dalla memoria dati o da un precedente risultato della ALU
ForwardB = 00	ID/EX	Secondo operando della ALU dal banco dei registri
ForwardB = 10	EX/MEM	Secondo operando della ALU propagato dal precedente risultato della ALU
ForwardB = 01	MEM/WB	Secondo operando della ALU propagato dalla memoria dati o da un precedente risultato della ALU

Condizioni e segnali di controllo

- Criticità EX

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
  ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 10

Condizioni e segnali di controllo (2)

- Criticità MEM

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
  ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 01

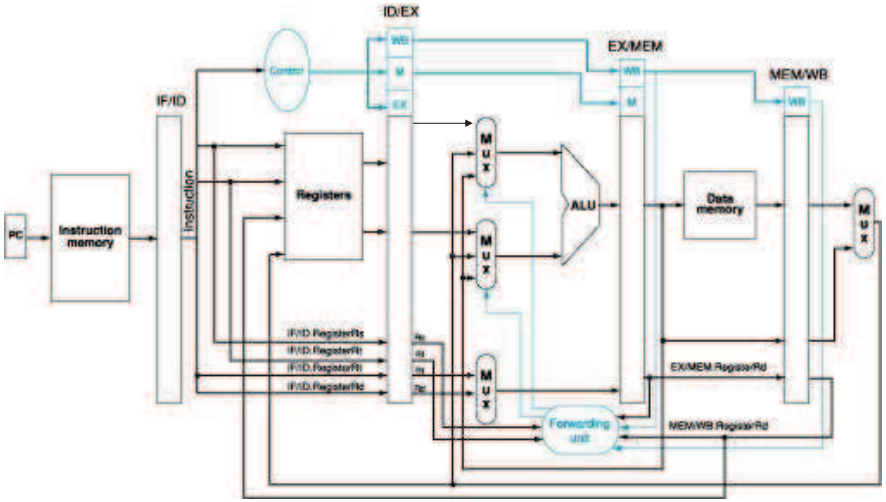
Condizioni e segnali di controllo (3)

- Potenziale criticità tra risultato dell'istruzione nello stadio WB, risultato dell'istruzione nello stadio MEM e operando sorgente dell'istruzione nello stadio EX
  - Esempio
    add $1, $1, $2
    add $1, $1, $3
    add $1, $1, $4
- Quindi, le condizioni per la criticità MEM diventano:

if (MEM/WB.RegWrite
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

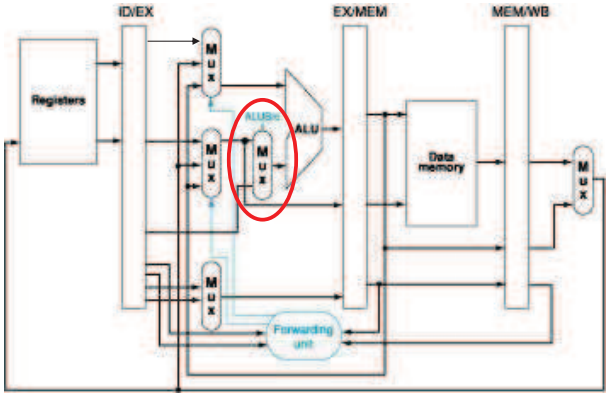
if (MEM/WB.RegWrite

Unità di elaborazione dati con forwarding



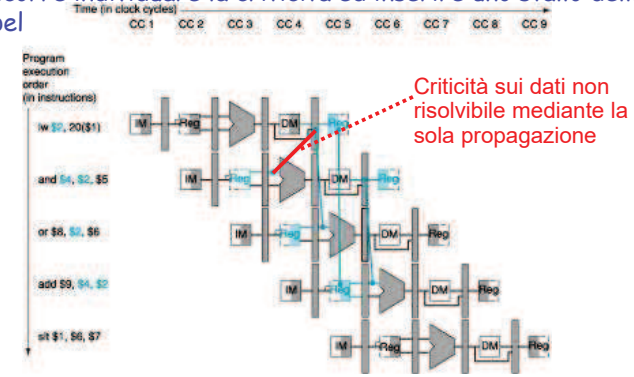
ALU e registri di pipeline con forwarding

- Aggiungiamo un MUX per scegliere come secondo operando sorgente della ALU anche il valore immediato (esteso in segno a 32 bit)



Criticità sui dati e stalli

- La propagazione non basta per risolvere una criticità sui dati determinata da un'istruzione che legge il registro scritto dalla precedente istruzione di lw (criticità *load/use*)
  - Occorre individuare la criticità ed inserire uno stallo della pipel



Condizione per la criticità load/use

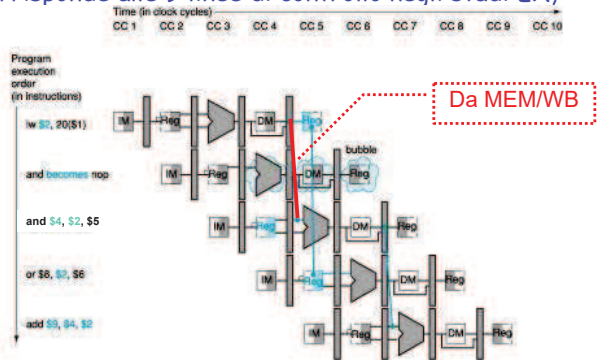
- Condizione per individuare la criticità sui dati di tipo load/use
  - Controllare se istruzione lw nello stadio EX
  - Controllare se il registro da caricare con lw è usato come operando dall'istruzione corrente nello stadio ID
  - In caso affermativo, bloccare la pipeline per un ciclo di clock

```
if (ID/EX.MemRead and
   and ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
          (ID/EX.RegisterRt = IF/ID.RegisterRt)))
stall the pipeline
```

- Condizione implementata dall'*unità di rilevamento di criticità (hazard detection unit)*

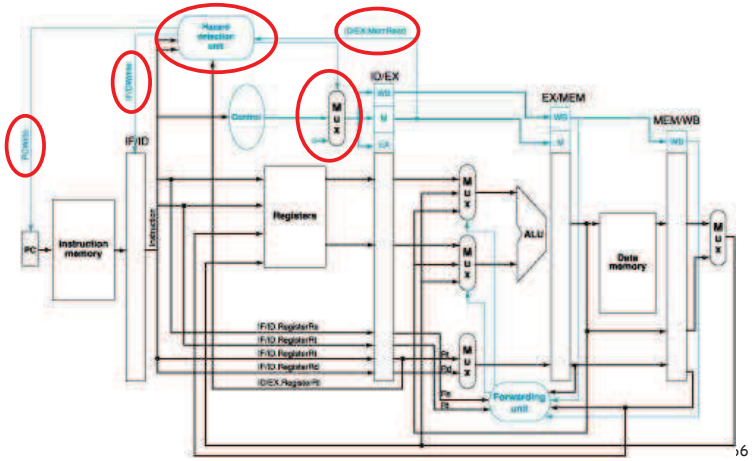
Implementazione di uno stallo

- Per un ciclo di clock
  - Non aggiornare il PC (PCWrite = 0)
  - Mantenere il contenuto del registro IF/ID (IF/IDWrite = 0)
  - nop corrisponde alle 9 linee di controllo negli stadi EX,



Unità di elaborazione

- Pipelining
- Propagazione
- Rilevamento di criticità e stallo

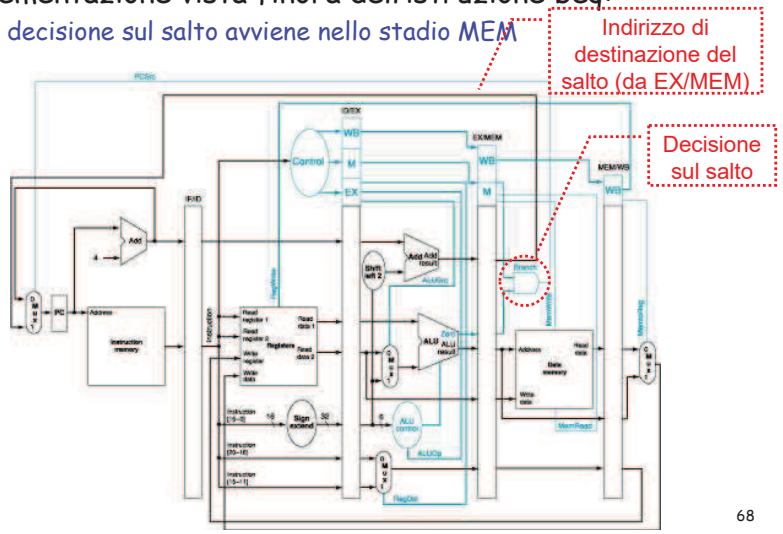


Criticità sul controllo

- 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

Salto condizionato nel MIPS

- Implementazione vista finora dell'istruzione beq:
  - La decisione sul salto avviene nello stadio MEM

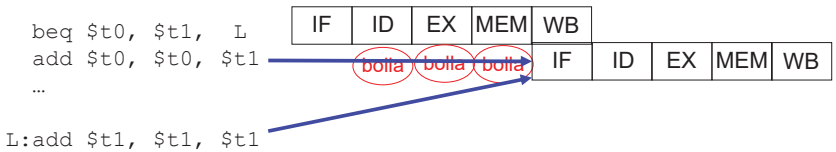


Soluzioni possibili

- Stallo della pipeline
- Ipotizzare che il salto condizionato non sia eseguito (branch not taken)
- Ridurre i ritardi associati ai salti
- Predizione dei salti
- Salto Ritardato

Stallo delle pipeline

- 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 (**stalling until resolution**)
    - ✓ Nel MIPS il risultato del confronto è noto al quarto passo: occorre inserire tre stalli



Salto non eseguito (**branch not taken**)

- Si assume che il salto non sia eseguito (**branch not taken**)
  - Si continuano a caricare nella pipeline le istruzioni successive a quella di salto condizionato
- Se il salto non è effettivamente eseguito
  - Non c'è nessuna penalizzazione
- Se invece il salto è eseguito (la predizione è errata)
  - Si scartano le istruzioni che sono state nel frattempo caricate nella pipeline
  - Si puliscono gli stadi IF, ID, EX
    - ✓ *Flushing* (annullamento) delle istruzioni
  - La pipeline viene caricata a partire dall'istruzione di destinazione del salto
  - Non è stato modificato nessun registro perché nessuna istruzione successiva al salto ha raggiunto lo stadio WB
  - Riduzione del throughput

Riduzione del costo

- Terza soluzione: si anticipa la decisione sul salto ad uno stadio precedente a MEM
- Occorre anticipare tre azioni
  - Calcolare l'indirizzo di salto
  - Valutare la decisione del salto
    - ✓ Per beq e bne occorre confrontare i registri
  - Aggiornare il PC
- Occorre aggiungere delle risorse hardware

1. Per il calcolo dell'indirizzo di salto
- Se l'indirizzo di salto è calcolato
  - Alla fine dello stadio EX: due stalli
  - Alla fine dello stadio ID: uno stallo
- Si sposta l'addizionatore per l'indirizzo di salto nello stadio ID

Riduzione del costo del salto (2)

2. Per confrontare i registri
   - Nel caso di beq: l'**unità di confronto** posta nello stadio ID esegue lo XOR bit a bit dei due registri e poi l'OR del risultato dello XOR
- Occorre gestire la propagazione all'ingresso dell'unità di confronto
  - Gli operandi sorgente possono provenire dai registri di pipeline EX/MEM o MEM/WB
- Può essere necessario uno stallo per risolvere una criticità sui dati
  - Es.: l'istruzione immediatamente precedente beq produce uno dei due operandi confrontati

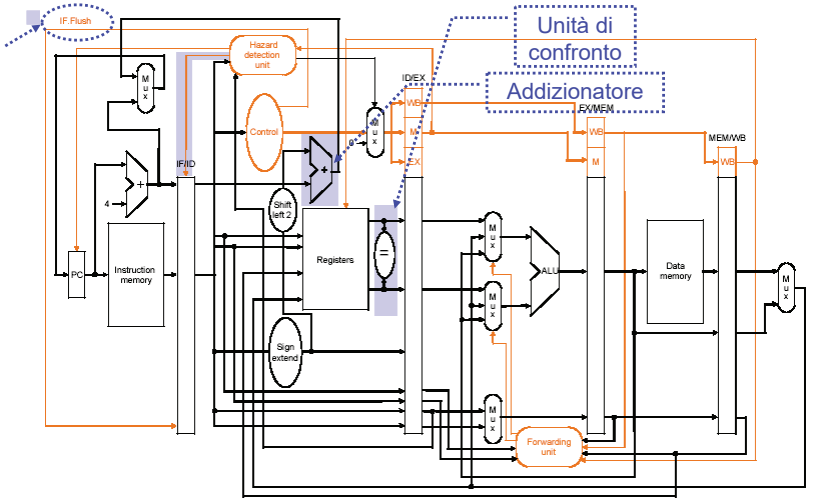
```
addi $6, $6, 4
beq $6, $7, Loop
```


Riduzione del costo del salto (3)

3. Per aggiornare il PC
   - Si sposta nello stadio IF la porta AND con ingressi il segnale di controllo Branch e l'uscita dell'unità di confronto
   - Se il salto è eseguito, il PC è scritto con l'indirizzo di destinazione del salto al termine del ciclo di clock dello stadio ID di beq

- Anticipando la decisione sul salto allo stadio ID si riducono i ritardi associati ai salti (*branch penalty*) delle prime due soluzioni
  - Occorre inserire un solo stallo *dopo* ogni salto
  - Oppure svuotare la pipeline di una sola istruzione

Modifica dell'unità di elaborazione



IF.Flush: segnale per azzerare i campi dell'istruzione nel registro di pipeline IF/ID

Tecniche per la predizione dei salti

- Obiettivo delle tecniche di predizione: predire quanto più presto possibile il risultato del salto
- Le prestazioni di una tecnica di predizione dipendono da:
  - *Accuratezza della predizione*: misurata in termini di percentuale di predizioni errate
  - *Costo di una predizione errata*: misurato in termini di tempo sprecato per eseguire istruzioni inutili
- Occorre considerare anche la frequenza dei salti
  - L'importanza di una tecnica di predizione accurata è maggiore in programmi che hanno un'elevata frequenza di salti

Tecniche per la predizione dei salti (2)

- Tecniche ***statiche*** per la predizione dei salti
  - Predizione fissata per ogni salto *a tempo di compilazione* e per tutta l'esecuzione del programma
  - Esempio: salto non eseguito (soluzione già considerata)
  - Soluzioni semplici, adatte per pipeline con pochi stadi
  - Con pipeline di profondità maggiore, il ritardo associato al salto diventa considerevole
- Tecniche ***dinamiche*** per la predizione dei salti
  - Predizione effettuata *a tempo di esecuzione*, usando informazioni sull'esecuzione di salti passati
  - Implementate in hardware

Tecniche di predizione statiche

- Usate prevalentemente in processori in cui il comportamento del salto può essere predetto correttamente con buona probabilità *a tempo di compilazione*
- Utilizzate anche come supporto a tecniche di predizione dinamiche

- Tipologie di tecniche statiche
  - **Salto non eseguito** (Branch Always Not Taken)
    - ✓ Già considerata
  - **Salto eseguito** (Branch Always Taken)
  - **Salto all'indietro eseguito in avanti non eseguito** (Backward Taken Forward Not Taken)
  - **Predizione guidata da profili** (profile-driven prediction)

Salto eseguito (Branch Always Taken)

- Approccio alternativo al Branch Always Not Taken
- Non appena l'istruzione di salto viene decodificata e viene calcolato l'indirizzo di salto, si assume che il salto sia sempre eseguito
  - Si carica nella pipeline l'istruzione corrispondente alla destinazione del salto
- Schema applicabile in pipeline in cui l'indirizzo di salto è noto prima del risultato del confronto
- Nel MIPS l'indirizzo di salto non è noto prima del risultato del confronto
  - Non c'è vantaggio nell'adottare questo approccio nel MIPS

Backward Taken Forward Not Taken Predizione guidata da profili

- Backward Taken Forward Not Taken
  - La predizione dipende dalla direzione del salto
  - Se il salto è all'indietro (*backward*) la tecnica predice che il salto verrà eseguito
    - ✓ Es.: il salto al termine di un ciclo per tornare indietro alla successiva iterazione del ciclo
  - Se il salto è in avanti (*forward*) la tecnica predice che il salto non verrà eseguito
- Predizione guidata da profili (profile-driven prediction)
  - La predizione è basata su informazioni relative all'esito dei salti acquisite in esecuzioni precedenti del programma

Tecniche di predizione dinamiche

- Idea: usare informazioni sull'esito di *salti passati* per predire il futuro
- Uso di hardware per effettuare la predizione del salto
  - La predizione dipende dall'esito del salto *a tempo d'esecuzione* e cambia se il salto modifica il proprio comportamento durante l'esecuzione
- Esaminiamo inizialmente uno schema semplice, per poi considerare approcci che incrementano l'accuratezza della predizione

Meccanismi per la predizione dinamica

La predizione dinamica è basata su due meccanismi:

1. Predizione dell'esito del salto
   - Si predice la direzione del salto: *taken* o *not taken*
   - Si utilizza la tabella di storia del salto (***Branch Prediction Buffer*** o BPB, anche detto ***Branch History Table*** o BHT)
2. Predizione della destinazione del salto
   - Si predice l'indirizzo di destinazione in caso di salto eseguito
   - Si utilizza il buffer di destinazione del salto (***Branch Target Buffer*** o BTB)

Predizione dell'esito del salto

- Analizziamo le seguenti tecniche:
  - Predizione ad 1 bit *(1-bit BHT)*
  - Predizione a 2 bit *(2-bit BHT)*
  - Predizione ad n bit *(n-bit BHT)*
  - ...

1-bit Branch History Table

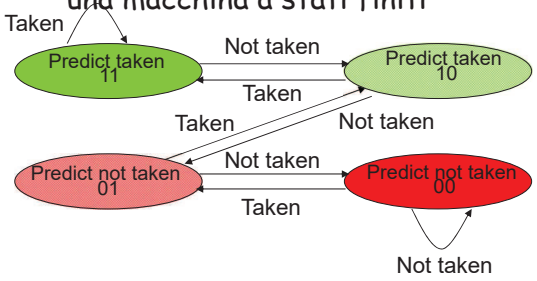
- Nella forma più semplice di predizione dinamica, si tiene conto dell'esito dell'ultima esecuzione del branch
- La tabella sulla storia dei salti (**1-bit BHT**):
  - Contiene righe da 1 bit, ciascuna della quali indica se recentemente il salto corrispondente è stato effettuato o meno
  - Tabella indicizzata tramite i bit meno significativi dell'indirizzo dell'istruzione di salto
- Predizione: è un suggerimento che si assume sia corretto
  - Se si scopre che la predizione è sbagliata, il bit di predizione è invertito e riscritto nella BHT; si svuota la pipeline e si esegue la sequenza di istruzioni corretta
- Il bit di predizione può essere stato scritto da un'altra istruzione di salto, il cui indirizzo ha gli stessi bit usati per indicizzare la tabella rispetto al salto che si sta considerando

Accuratezza della 1-bit Branch History Table

- Si ha una predizione errata quando
  - Per quel salto la predizione è sbagliata
  - La stessa riga della tabella è acceduta da due salti diversi e la storia precedente (il bit) si riferisce all'altro salto
    - ✓ Soluzioni possibili: ampliare il numero di righe della BHT per ridurre il numero di interferenze o usare una funzione di hashing per l'indicizzazione)
- Svantaggio della BHT ad 1 bit
  - In un loop, anche se il salto viene quasi sempre effettuato e solo una volta non viene effettuato (alla fine del loop), si ha un doppio errore
    - ✓ Alla fine dell'iterazione del loop e all'inizio dell'iterazione successiva dello stesso loop

2-bit Branch History Table

- Per aumentare l'accuratezza si usa una tabella a 2 bit
- La predizione deve essere errata per due volte consecutive prima che venga invertita
- I due bit sono usati per codificare i quattro stati di una macchina a stati finiti



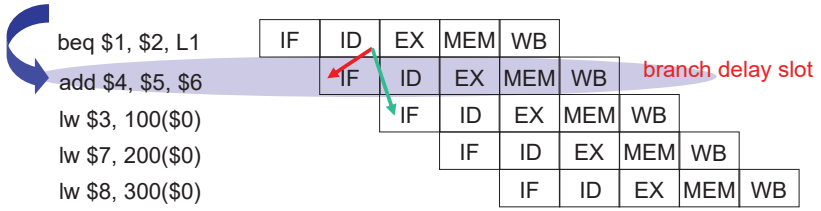
- Contatore a 2 bit in saturazione (valore min 00, valore max 11)
- Contatore incrementato se branch taken
- Contatore decrementato se branch not taken

Salto ritardato

- Il compilatore seleziona un'istruzione da porre nel *branch delay slot*, riorganizzando il codice
  - Branch delay slot: la posizione dopo l'istruzione di salto
- L'istruzione nel branch delay slot viene eseguita sempre, indipendentemente dall'esito del salto
  - Non si devono scaricare dalla pipeline le istruzioni che seguono il salto!
- Se assumiamo di avere un *branch delay* di un ciclo, abbiamo solo un branch delay slot
  - E' il caso del processore MIPS
  - E' possibile avere un branch delay maggiore di un ciclo, ma tutti i processori che adottano questa soluzione hanno un solo delay slot

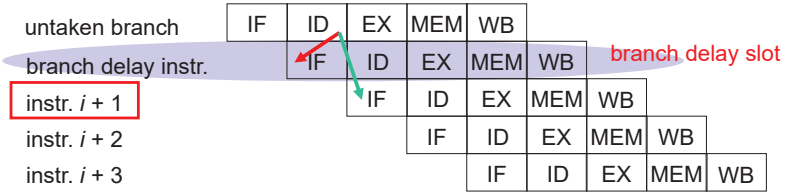
Salto ritardato (2)

- Il compilatore MIPS cerca di posizionare dopo il salto un'istruzione indipendente dall'esito del salto
- Esempio
  - Per il branch delay slot viene selezionata un'istruzione add precedente a beq (senza effetti su beq)

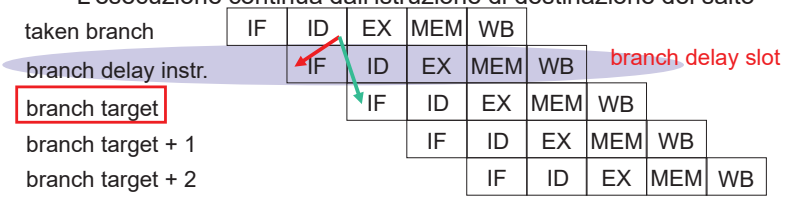


Salto ritardato (3)

- Se il salto non è eseguito
  - L'esecuzione continua con l'istruzione successiva al salto

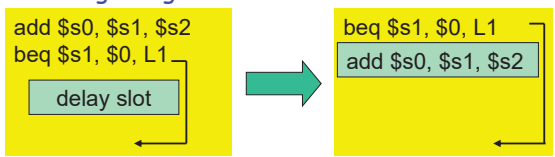


- Se il salto è eseguito
  - L'esecuzione continua dall'istruzione di destinazione del salto



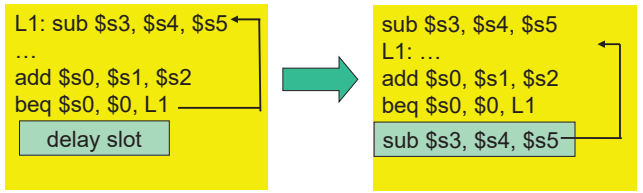
Selezione del delay slot

- Compito del compilatore è selezionare un'istruzione da porre nel delay slot che sia valida ed utile
- Tre strategie per selezionare l'istruzione da spostare
  - Da prima del salto
  - Dalla destinazione del salto
  - Tra salto e destinazione del salto
- Strategia 1: da prima del salto
  - Nel delay slot si posiziona un'istruzione indipendente proveniente dalla parte di codice prima del salto
  - L'istruzione nel delay slot viene *sempre* eseguita
  - E' la strategia migliore



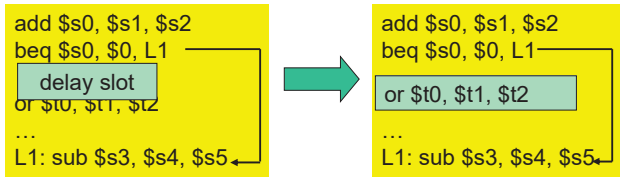
Selezione del delay slot (2)

- Strategia 2: dalla destinazione del salto
  - Non è possibile scegliere un'istruzione prima del salto
  - Nel delay slot si posiziona l'istruzione di destinazione del salto
  - Questa istruzione viene generalmente copiata perché può essere raggiunta anche tramite un altro percorso
  - Strategia utile per salti all'indietro (elevata probabilità che il salto sia eseguito)



Selezione del delay slot (3)

- Tra salto e destinazione del salto
  - Non è possibile scegliere un'istruzione prima del salto
  - Nel delay slot si posiziona un'istruzione proveniente dalla sequenza tra salto e destinazione del salto
  - Strategia utile per salti in avanti



- Nelle ultime due strategie deve essere possibile eseguire l'istruzione spostata quando il salto va nella direzione inattesa