Architettura degli Elaboratori

L'architettura della CPU – Gestione dei control hazard

SAPIENZA
UNIVERSITÀ DI ROMA

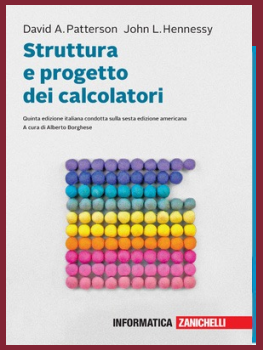
Alessandro Checco

checco@di.uniroma1.it

Special thanks and credits:

Andrea Sterbini, Iacopo Masi, Claudio di Ciccio

[S&PdC] 4.8



Anticipare i salti

Immagine: https://sonic.fandom.com/wiki/Time_Warp_Plate

Argomenti

Argomenti della lezione

- Spostare Jump nella fase IF
- Come gestire i control hazard
  - Eliminare le istruzioni non più necessarie
- Come ridurre l'impatto del control hazard
  - Anticipare la decisione di salto
  - Ritardare l'esecuzione del salto
  - «Prevedere» la destinazione del salto
- Soluzione esercizio

Jump: ogni istruzione viene riconosciuta nella fase ID mentre un'altra viene caricata
quindi il Jump implica almeno uno stallo (l'istruzione successiva)

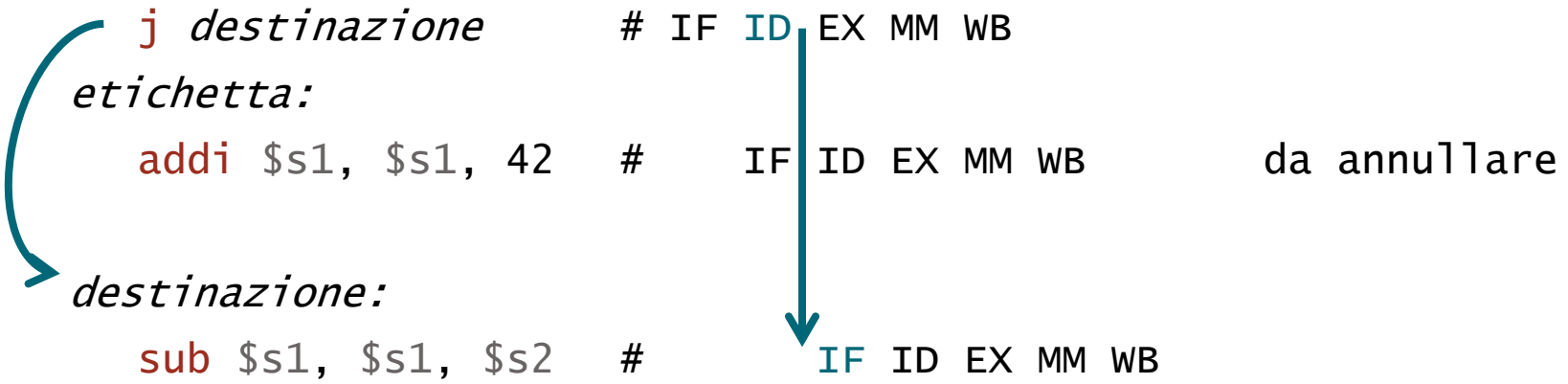
Control hazard:

quando deve essere fatto un salto la pipeline è mezza piena e
le istruzioni seguenti già caricate vanno eliminate dalla pipeline

Un inutile stallo per il Jump!

La decisione di eseguire il Jump viene presa dalla Control Unit nella fase ID

- nel frattempo, un'altra istruzione è stata già caricata (occorre che si annulli)



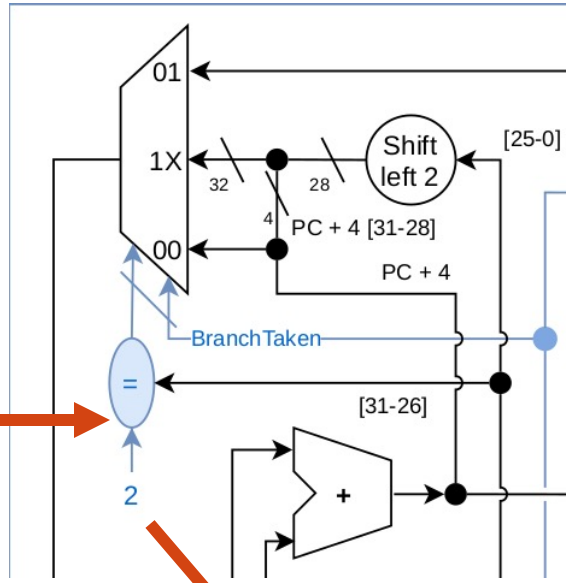
Per **eliminare l'istruzione** con uno stallo, occorre (nella fase ID):

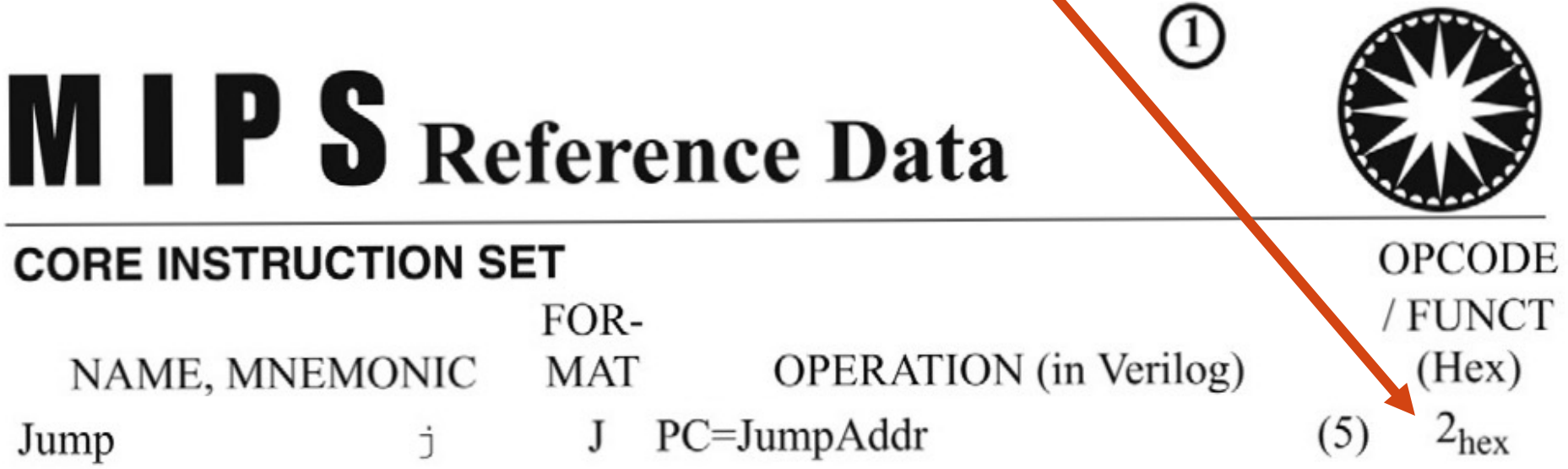
- annullare l'istruzione inutile (**bolla**)
  - ovvero **azzerare i segnali di controllo MemWrite e RegWrite e IF/ID**
- **aggiornare il PC** affinché possa leggere la destinazione al prossimo colpo di clock

Anticipare il Jump alla fase IF

Per saltare con la J in fase IF occorre:

- **anticipare il riconoscimento della istruzione**
  (che normalmente fa la CU in fase ID)
  - comparatore col valore dell'OpCode della J (`000010`)



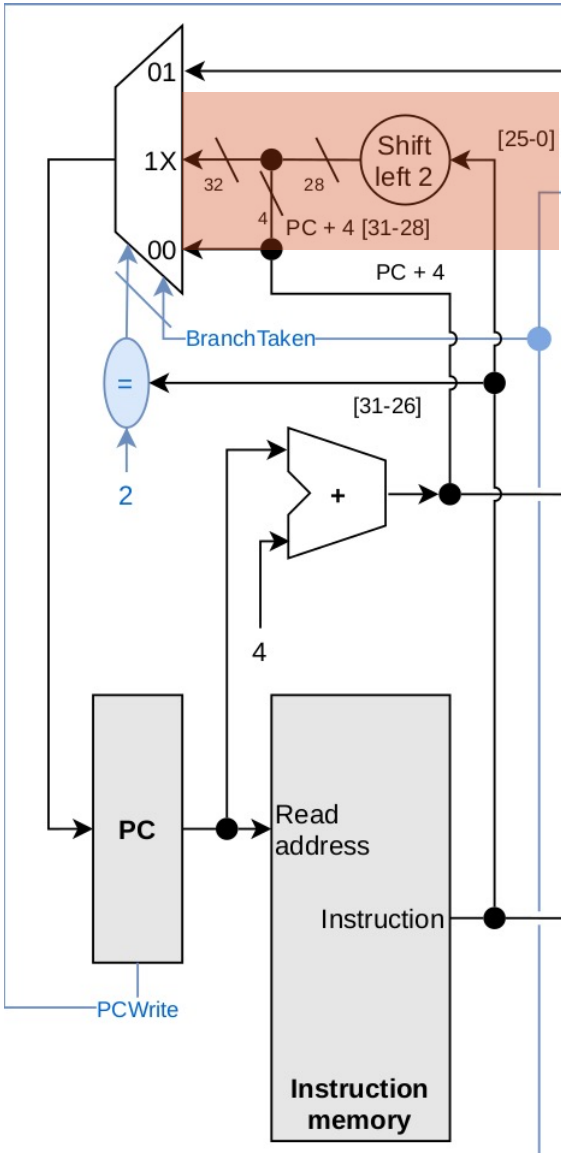


Anticipare il Jump alla fase IF

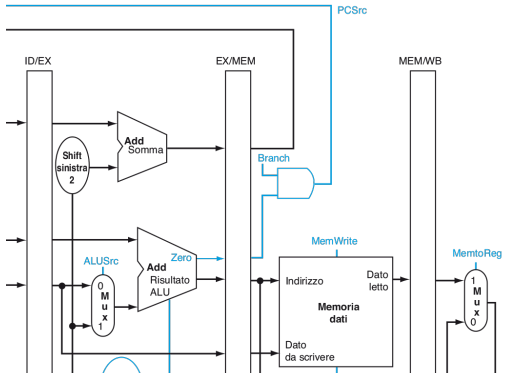
Per saltare con la J in fase IF occorre:

- **anticipare il riconoscimento della istruzione**
  (che normalmente fa la CU in fase ID)
  - comparatore col valore dell'OpCode della J (`000010`)
- **spostare la logica di aggiornamento del PC alla fase IF**
  - **shift logico a 2** dei 26 bit meno significativi dell'istruzione
  - aggiunta dei 4 bit più significativi di PC+4

Risultato: la Jump anticipata **non introduce più stalli**



Control hazards (beq)



L'istruzione **beq** normalmente **usa la ALU** per fare il confronto, per cui:

- il salto avviene normalmente **dopo** la fase **EXE** (nella fase MEM)
  - in caso di salto, le **2 istruzioni seguenti** (già caricate) vanno annullate
- necessita degli argomenti nella fase EXE
  - può aver bisogno di uno **stallo** solo se **preceduta da lw** (come ogni istruzione di tipo R)

Come **annullare le istruzioni** a monte in pipeline (istruzioni future)

- IF/ID.Istruzione viene azzerata → **NOP** (**No Operation**), 0x00…0
- ID/EXE.MemWrite e ID/EXE.RegWrite vengono azzerate
  - per non modificare memoria o registri (in realtà anche MemRead)

A cosa corrisponde un'istruzione che consta di soli bit pari a 0?

1) Una OpCode pari a 000000_{due} indica un'istruzione di tipo R

MIPS Reference Data ①

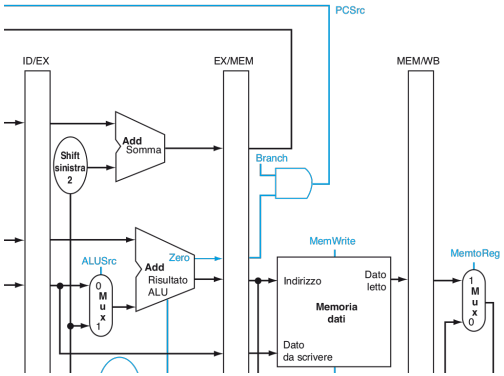
CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)	OPCODE / FUNCT (Hex)
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt	$0 / 00_{hex}$

2) A quale istruzione corrisponde una funct pari a 000000_{due}?

Bkpt	Address	Code	Basic	Source
☐	4194304	0x00000000	sll $0,$0,0	2: sll $zero,$zero,0

Control hazards (beq)



L'istruzione **beq** normalmente **usa la ALU** per fare il confronto, per cui:

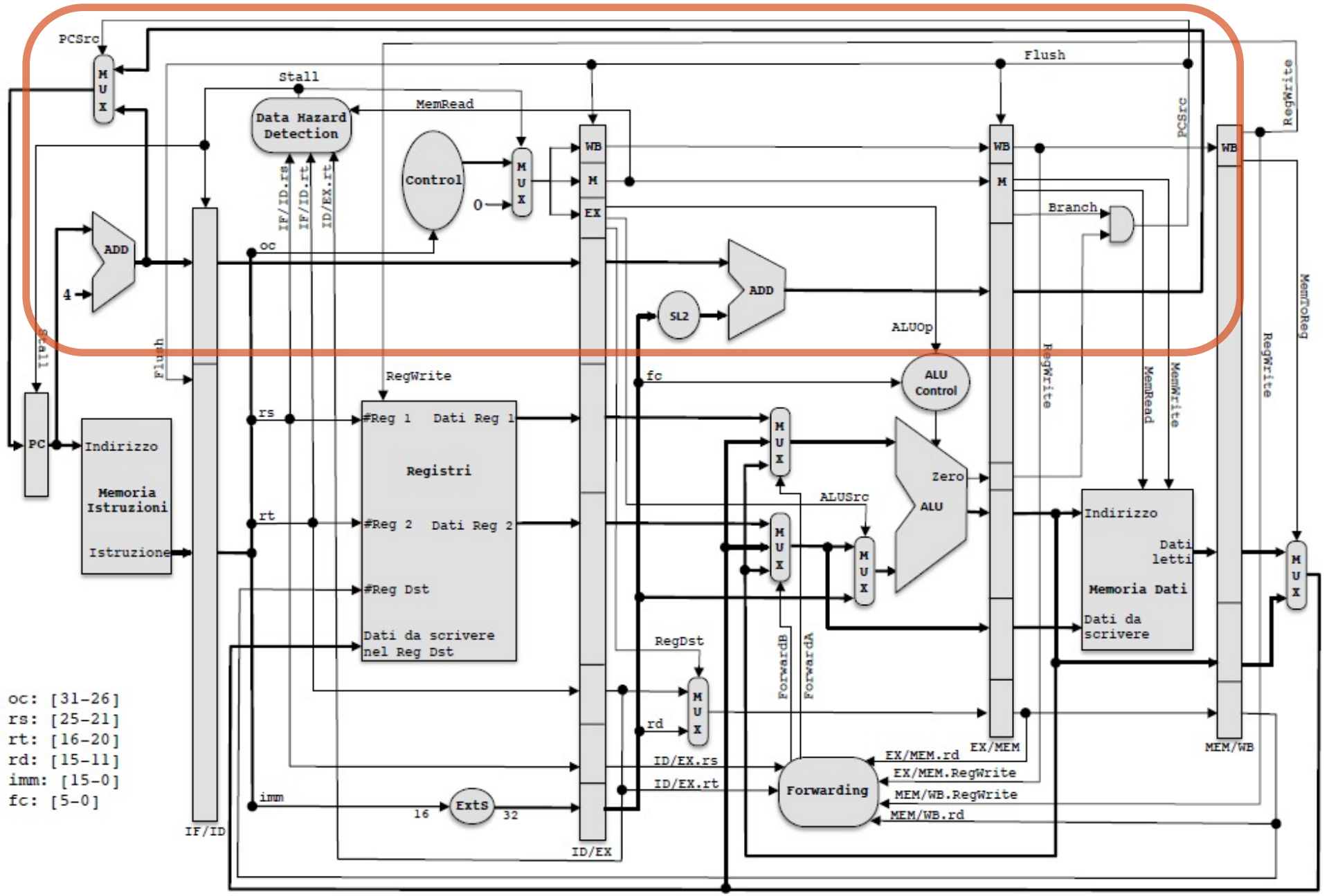
- il salto avviene normalmente **dopo** la fase **EXE** (nella fase MEM)
  - in caso di salto, le **2 istruzioni seguenti** (già caricate) vanno annullate
- necessita degli argomenti nella fase EXE
  - può aver bisogno di uno **stallo** solo se **preceduta da lw** (come ogni istruzione di tipo R)

Come **annullare le istruzioni** a monte in pipeline (istruzioni future)

- IF/ID.Istruzione viene azzerata → **NOP** (**No Operation**), 0x00…0
- ID/EXE.MemWrite e ID/EXE.RegWrite vengono azzerate
  - per non modificare memoria o registri (in realtà anche MemRead)

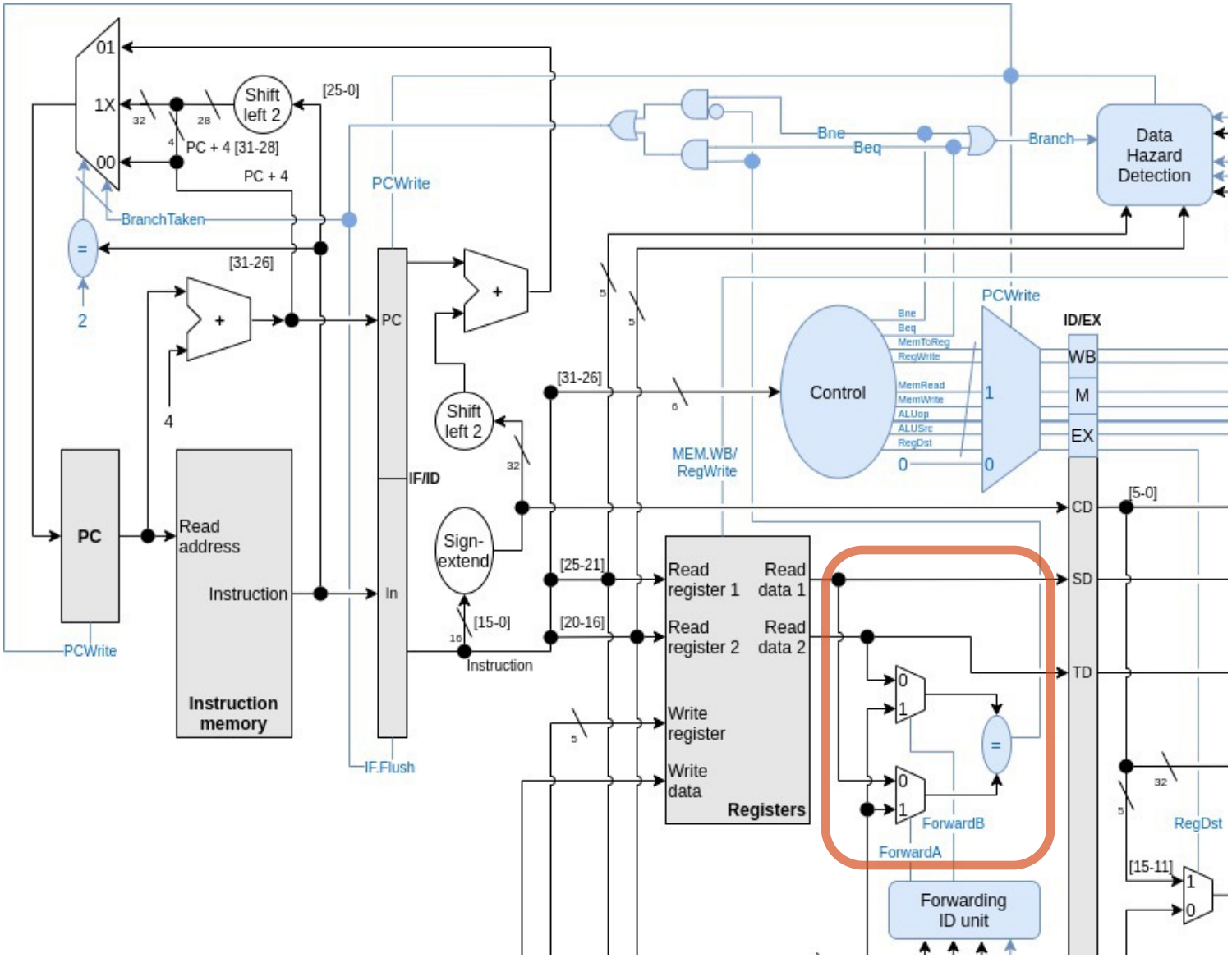
Per **anticipare la decisione di salto** alla fase **ID** occorre non usare la ALU:

- inserendo un comparatore tra i due argomenti letti dal blocco registri
- spostando la **logica di salto** ed il **calcolo del salto relativo** dalla fase EXE alla **fase ID**
- sarà necessaria una **unità di forwarding apposita** per la fase ID



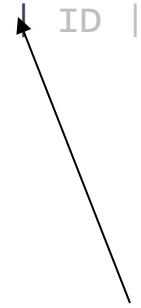
rappresentazione con branch dopo EXE (nota Flush)

Anticipiamo beq (e bne) alla fase ID



IF.Flush per branch in ID

```
                                CC       (1)   (2)   (3)   (4)   (5)
i1. beq (will be taken to i3)           # IF  | ID  | EX  | MM  | WB
12. instr (will be skipped)             #     XF  | ID  | EX  | MM  | WB
i3. instr (will be next)
```



at CC (2):
ID identifies branch taken and flushes the IF/ID pipeline

IF.Flush per branch in ID

```
                                CC       (1)   (2)   (3)   (4)   (5)
i1. beq (will be taken to i3)           # IF  | ID  | EX  | MM  | WB
12. instr (will be skipped)             #     | IF  | ID  | EX  | MM  | WB
i3. instr (will be next)                #           | IF  |
```

at CC (2):
ID identifies branch taken and flushes the IF/ID pipeline

at CC (3):
- IF of i3
- ID executes a **nop**
- Executes EX of i1 (do nothing)

Anticipando la decisione del branch in fase ID si passa da 2 stalli a uno (non eliminabile) quando il branch è taken

Anticipiamo beq (e bne) alla fase ID

L'abbassamento del numero di stalli nel caso di predizione sbagliata (da 2 a 1) non è gratuito.

La fase in cui beq e bne necessitano dei dati viene anticipata da EXE a ID.

Servirebbe dunque l'introduzione del forwarding da MEM a ID nel caso (cosa che ovviamente non abbiamo)

```
lw  $t0, ($t2)          # IF | ID | EX | MM | WB
beq $t0, $zero, dest    #      ->   -> | IF | ID | EX | MM | WB
```

Anche operazioni di tipo R seguite da **beq** o **bne** richiedono uno stallo per queste ultime (data hazard).

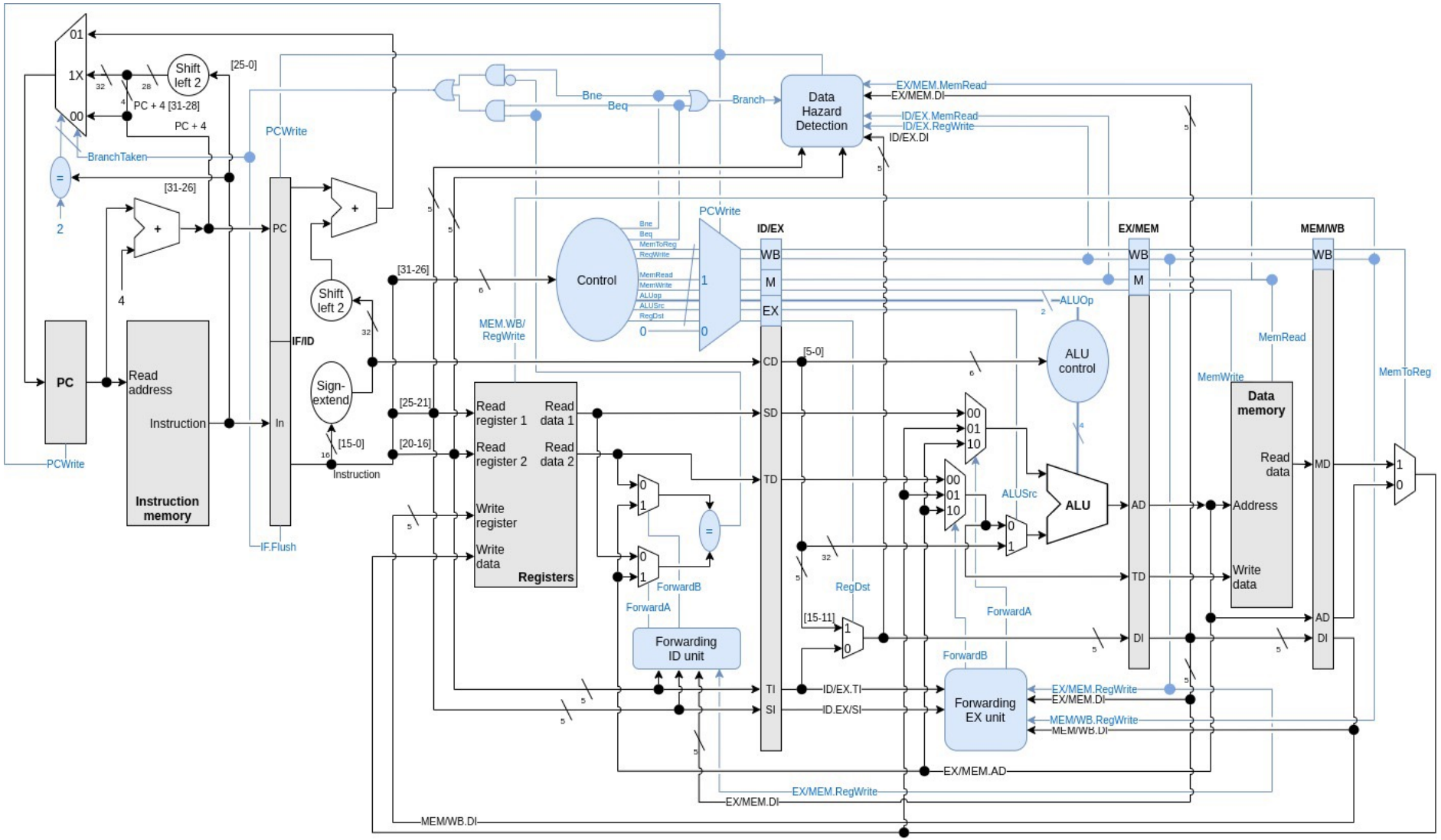
```
add $t0, $t1, $t2       # IF | ID | EX | MM | WB
beq $t0, $zero, dest    #      ->   IF | ID | EX | MM | WB
```

Questi stalli (come per lw->R) vengono gestiti dalla componente Data Hazard in fase di ID settando **PCWrite a 0** (ripete l'istruz al CC succ)

?

Perché non propaghiamo dati e segnali da MEM a ID?

Architettura della CPU con pipeline



Predire i salti

SAPIENZA
UNIVERSITÀ DI ROMA

Immagine: https://www.gossipetv.com/tag/otelma/page/2

salto ritardato (recap)

idea: inserire un'istruzione (o due) che verrà eseguita in entrambi i casi (salto o no) per evitare di dover inserire stalli dopo un branch

se l’istruzione che segue la beq viene SEMPRE eseguita anche se il salto viene fatto, si elimina di fatto lo stallo eseguendo quell’istruz al posto dello stallo (non sempre possibile)

Impatto della beq

Se il salto viene eseguito dopo **EXE** (adoperando la ALU) si hanno

- **2 stalli dopo**, se il salto viene eseguito (control hazard, eliminabili con **due salti ritardati**)
- **1 stallo prima** se la `beq` è preceduta da `lw` (data hazard)
  - riordinando il codice, si può ovviare allo stallo (non sempre)

Se la **`beq`** è spostata nella **fase ID**:

- **1 stallo** dopo, se il salto viene eseguito (eliminabile con il **salto ritardato**)
- **2 stalli prima** se preceduta da lw o 1 stallo se da istruzioni di tipo R (data hazard)
  - riordinando il codice, si può ovviare allo stallo (non sempre)

L'impatto dipende dal tipo di codice che usa la `beq`, soprattutto nei cicli:

- se il test è all'inizio, seguito dal corpo del ciclo, la `beq` fa **un solo salto alla fine del ciclo**
- se il test è alla fine, dopo il corpo, la `beq` effettua **molti salti** e in un solo caso continuerà

La CPU vista finora presume che il salto non venga effettuato (**branch not taken by default**)

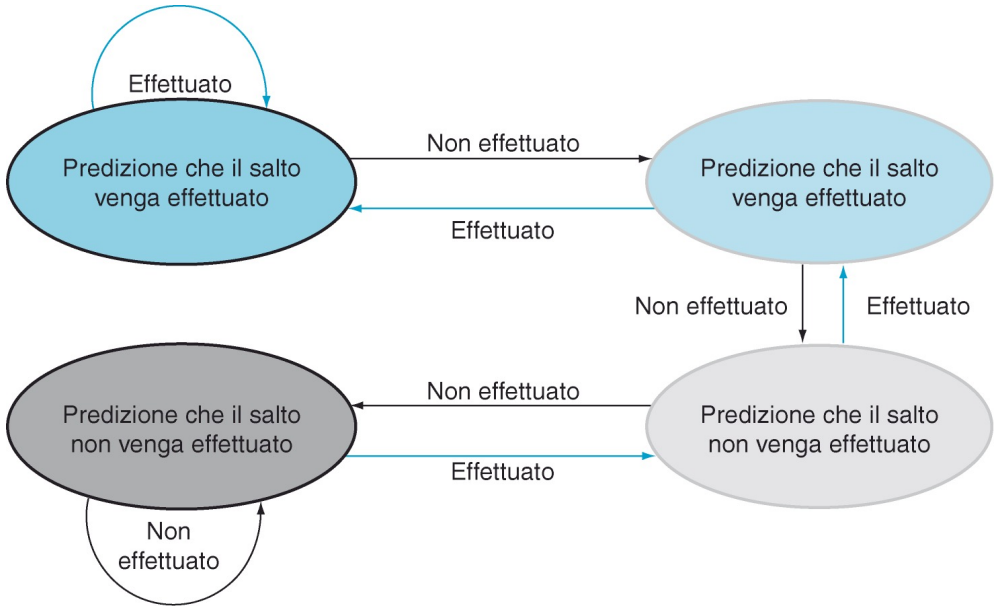
- se si potesse «prevedere» il salto si potrebbero caricare le istruzioni giuste ed evitare stalli

Default: decisione in ID senza salto ritardato

Predizione dei salti (livello HW): default branch (not?) taken

Ad ogni istruzione di salto possiamo associare **bit che «contano»** i salti già effettuati per decidere se sia più probabile che il salto venga effettuato o meno

- indice in micro-memoria indicizzata dagli LSB dell'istruzione
  - problema di «hash collision» (istruzioni con stessi LSB ma diversi comportamenti)
- con **un bit** si può rappresentare l'informazione «l'ultima volta il salto è stato effettuato»
  - nel realizzare un ciclo la previsione sarà sbagliata 2 volte: **entrando e uscendo**
- con **due bit** si può realizzare una macchina a stati finiti
  - necessita di due errori consecutivi per cambiare previsione
  - quindi **incorre in meno errori** nei cicli a forte prevalenza di uno specifico tipo di scelta (una sola predizione errata)



Predizione dei salti come problema di classificazione binaria

La predizione dei salti può essere concepita come un problema **di classificazione binaria.**

La favola di Pierino ed il lupo (in inglese)

We can summarize our "wolf-prediction" model using a 2x2 confusion matrix that depicts all four possible outcomes:

Stato effettivo (branch)

Assunzione

True Positive (TP): • Reality: A wolf threatened. • Shepherd said: "Wolf." • Outcome: Shepherd is a hero.	**False Positive (FP):** • Reality: No wolf threatened. • Shepherd said: "Wolf." • Outcome: Villagers are angry at shepherd for waking them up.
False Negative (FN): • Reality: A wolf threatened. • Shepherd said: "No wolf." • Outcome: The wolf ate all the sheep.	**True Negative (TN):** • Reality: No wolf threatened. • Shepherd said: "No wolf." • Outcome: Everyone is fine.

Branch not taken

Fonte: https://developers.google.com/machine-learning/crash-course/classification/true-false-positive-negative

Ritardare il salto

Oppure per recuperare il tempo perso dallo stallo si può **ritardare il salto**

- **eseguire** in ogni caso l'istruzione che segue la beq (**delay slot**)

Questa strategia richiede

- una scrittura del codice assembly diversa, oppure
- l'uso di un compilatore capace di automatizzare le necessarie modifiche al codice macchina prodotto

Nel delay slot è possibile copiare una delle istruzioni che vanno sempre eseguite:

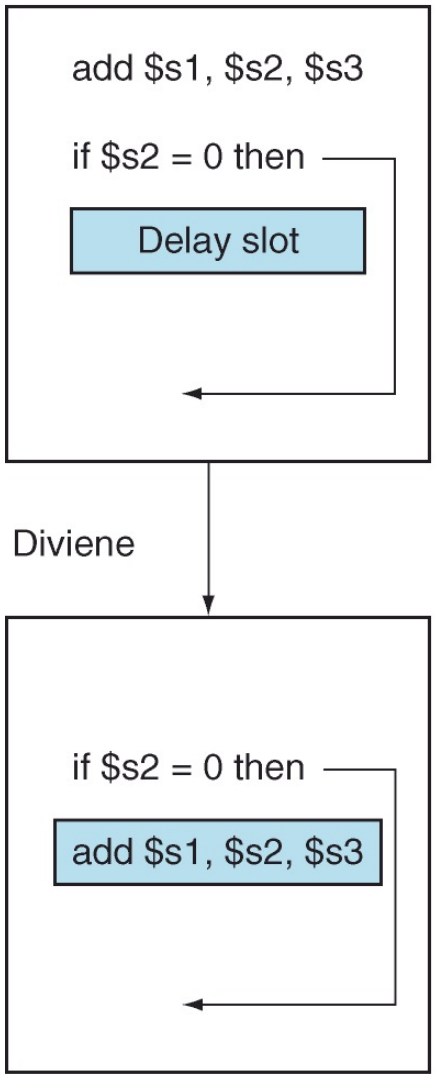
- Caso 1: **un'istruzione precedente** che non abbia dipendenze (anche indirette) con la `beq`

Se non ci sono istruzioni precedenti senza dipendenze:

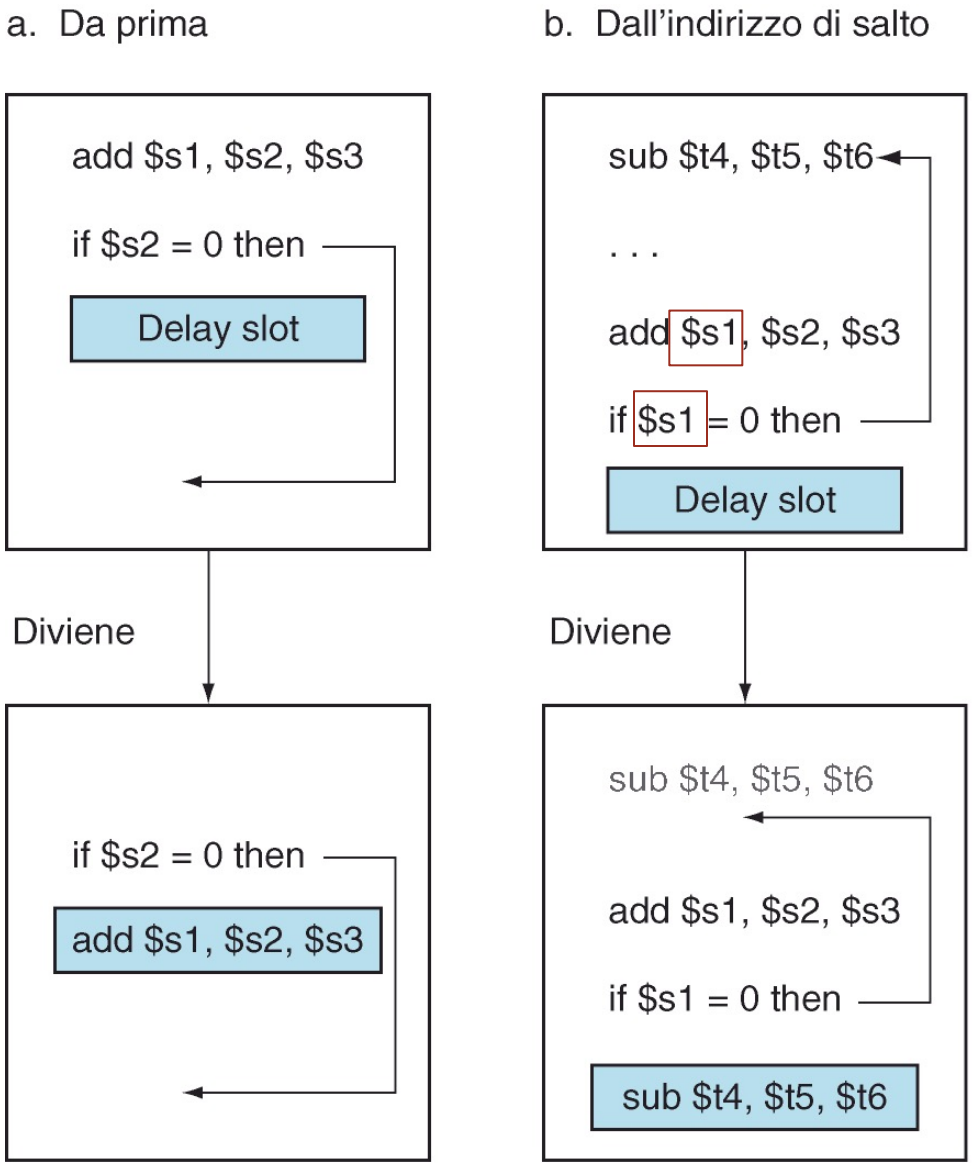
- Caso 2: **l'istruzione alla destinazione del salto (non sempre possibile)**
  - NOTA: l'istruzione viene **copiata** perché potrebbe far parte di altri flussi di controllo
  - NOTA: se il salto NON viene effettuato, l'istruzione scelta (sempre eseguita) non deve creare problemi:
    - ad esempio, può calcolare un valore non più necessario nel codice seguente
    - l'importante è che non sia dannosa per l'esecuzione successiva
- Caso 3: inserisco una nop se non ci sono alternative

Riordinamento con salto ritardato (livello assemblatore)

a. Da prima



Riordinamento con salto ritardato



Riordinamento con salto ritardato