

# Le architetture a pipeline

M. Sonza Reorda

Politecnico di Torino  
Dip. Automatica e Informatica



# Introduzione

**Per migliorare le prestazioni di un processore si può agire in due direzioni:**

- **aumentando la velocità dei componenti**
- **modificando l'architettura.**

**L'uso di architetture a pipeline segue la seconda strada.**

# Processori RISC e superscalari

I processori tradizionali richiedono un certo numero di colpi di clock per l'esecuzione di ogni istruzione.

Grazie all'introduzione delle pipeline, in ogni colpo di clock

- i processori *RISC* possono completare un'istruzione
- i processori *superscalari* possono completare anche più di un'istruzione.

# CPI

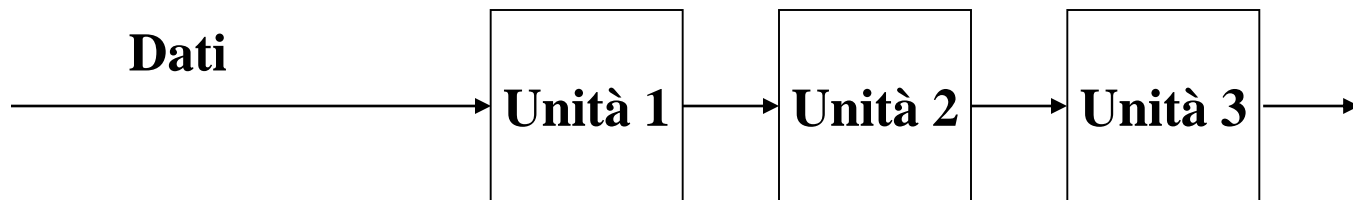
**Per misurare l'efficienza di un processore si usa a volte il parametro CPI (*Clocks per Instruction*):**

- **per processori CISC  $\Rightarrow$  CPI  $> 1$**
- **per processori RISC  $\Rightarrow$  CPI  $\sim 1$**
- **per processori superscalari  $\Rightarrow$  CPI  $< 1$ .**

# Pipeline

**La pipeline è l'equivalente elettronico della *catena di montaggio*.**

**Permette di eseguire in parallelo operazioni diverse sullo stesso flusso di dati.**



# Pipeline ideale

**Se**

- **tutti i dati devono passare attraverso gli stessi stadi**
- **tutti gli stadi richiedono sempre lo stesso tempo per eseguire la relativa elaborazione**

**allora l'uso di una pipeline di  $n$  stadi può portare a regime ad un miglioramento di  $n$  volte nelle prestazioni.**

# La pipeline nei processori

L'esecuzione di ciascuna istruzione può essere suddivisa in varie fasi, quali:

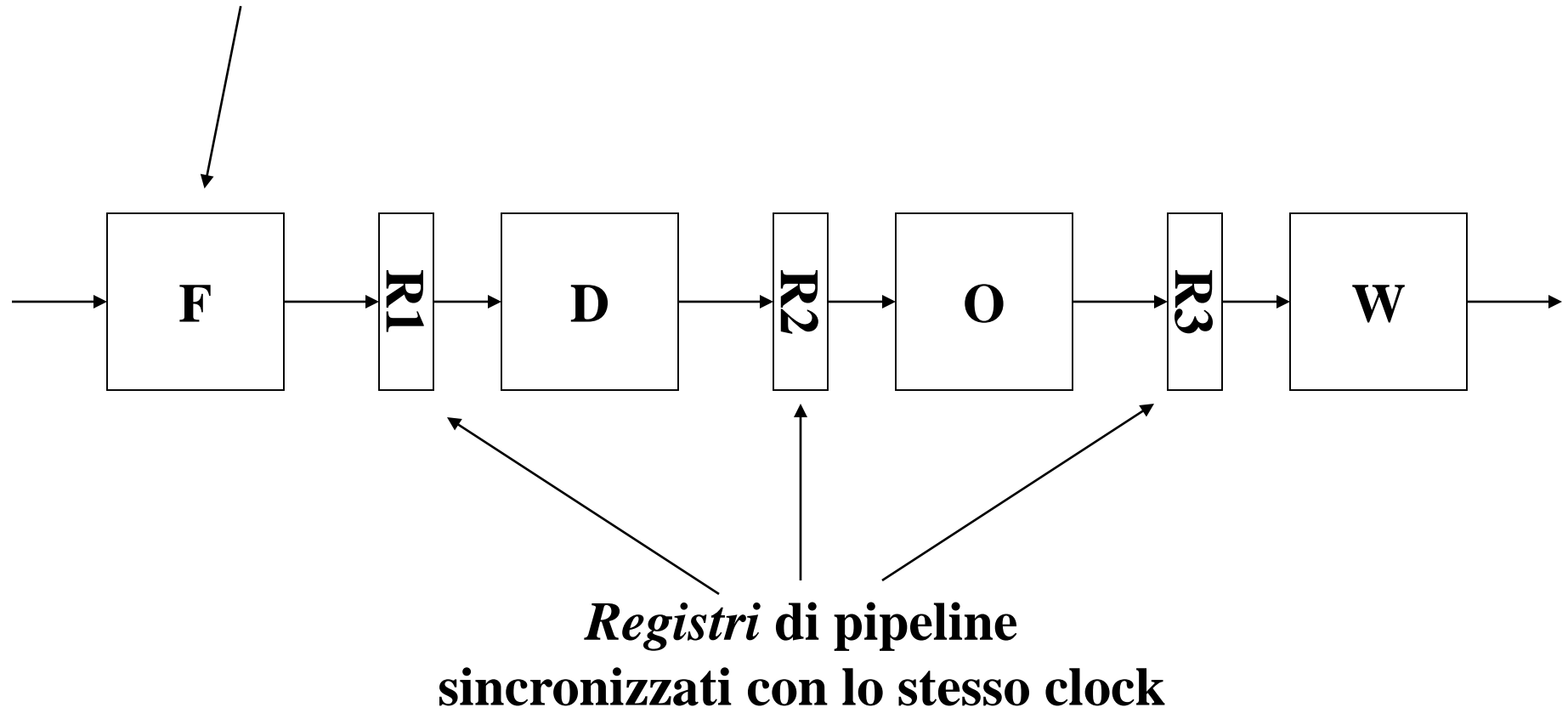
- *Fetch* (o prelievo): caricamento dell'istruzione dalla memoria
- *Decode* (o decodifica): decodifica dell'istruzione e lettura degli operandi sorgente
- *Operate* (o elaborazione): esecuzione dell'operazione
- *Write* (o scrittura): scrittura del risultato.

Ciascun processore può adottare una suddivisione in un numero di fasi diverso:

- Processori ARM: 3 fasi
- Processori Intel Netburst (Pentium 4): 31 fasi.

*Stadio* della pipeline  
in ogni periodo di clock  
esegue una fase  
su un'istruzione

# Architettura





# Esempio

Colpo di clock

Istruzione

**i   i+1   i+2   i+3   i+4   i+5   i+6   i+7**

**I<sub>1</sub>**

<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>
----------	----------	----------	----------

**I<sub>2</sub>**

<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>
----------	----------	----------	----------

**I<sub>3</sub>**

<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>
----------	----------	----------	----------

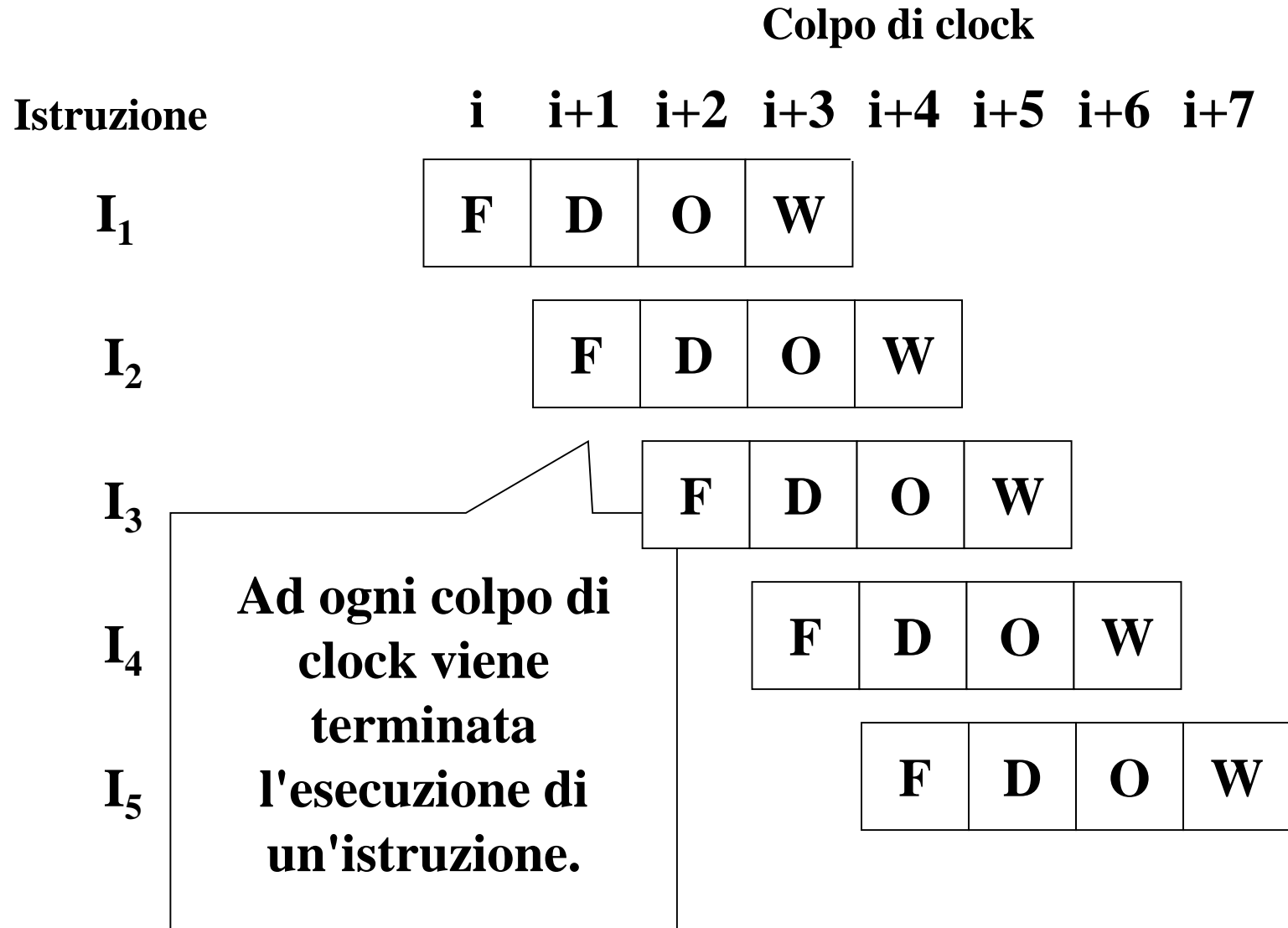
**I<sub>4</sub>**

<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>
----------	----------	----------	----------

**I<sub>5</sub>**

<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>
----------	----------	----------	----------

# Esempio



# Stallo

**In un processore reale, è possibile che a volte uno stadio non riesca a completare il proprio compito in un colpo di clock.**

**In tal caso**

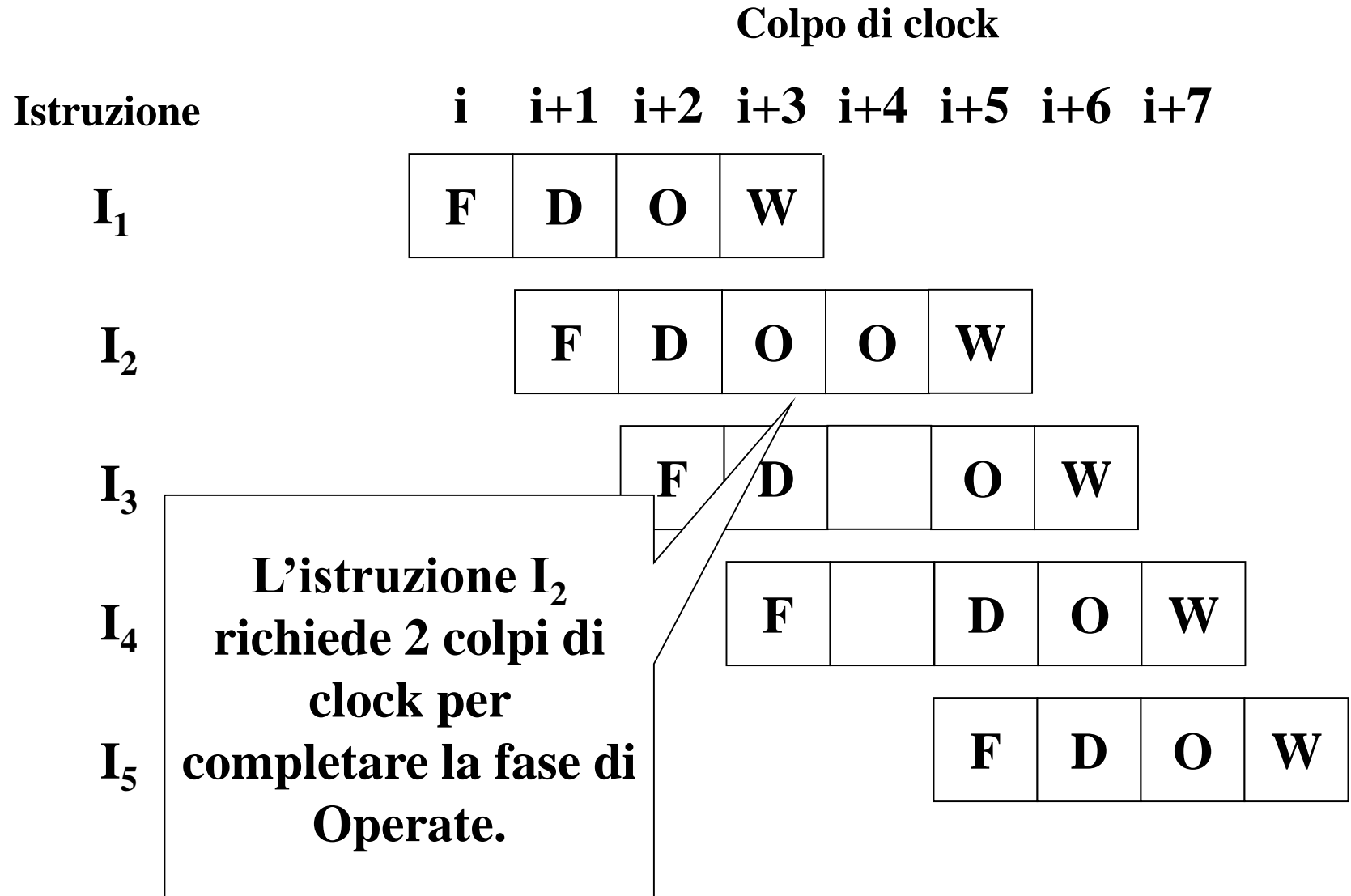
- gli altri stadi a monte devono arrestarsi, ossia andare *in stallo*, in quanto i registri in cui devono scrivere non sono liberi**
- gli stadi a valle possono invece continuare a lavorare, ma solo per smaltire le istruzioni già processate dallo stadio critico.**

**In tal modo le prestazioni del processore scendono rispetto a quelle ideali previste.**

# Cause di stallo

- **Lentezza negli accessi a memoria (fasi F, D e W)**
- **Lentezza nell'esecuzione di operazioni interne (fase O).**

# Esempio



# Cache

**Le cache sono fondamentali nei processori che usano una pipeline, in quanto permettono di accedere a dati e istruzioni in un solo colpo di clock.**

# Bolle

**Dopo che uno stadio è andato in stallo, si crea all'interno della pipeline una situazione di inattività (denominata *bolla*) che viene eliminata gradualmente.**

# Coda delle istruzioni

**Quando l'unità di fetch non trova un'istruzione in cache, manda in stallo l'intera pipeline.**

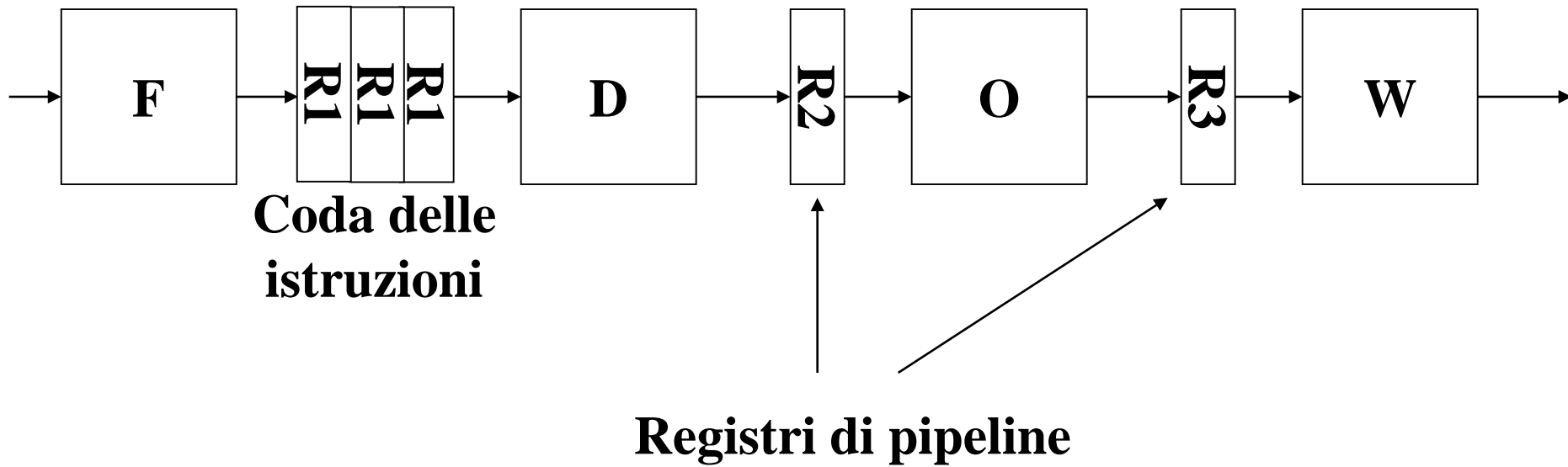
**Per evitare questa situazione, il buffer tra le unità di fetch e di decodifica è sostituito da una coda, detta *coda delle istruzioni*.**

**L'unità di fetch cerca di mantenere sempre piena tale coda, caricando anche più di un'istruzione per colpo di clock.**

**In tal modo si compensano eventuali ritardi dovuti a situazioni di miss nella cache.**



# Architettura



# Dipendenze di dato

**La presenza di una pipeline rende parallelo un processo (quello di esecuzione delle istruzioni) che è originariamente sequenziale.**

**Affinché il risultato sia lo stesso del processo originario, bisogna che non vi siano dipendenze tra dati.**

**Ad esempio bisogna che gli operandi sorgente di un'istruzione siano già stati prodotti dalle istruzioni precedenti.**

# Esempio

**Mul R2, R3, R4**

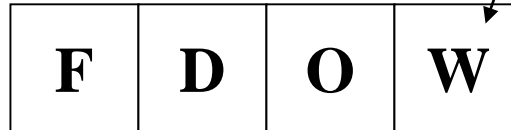
**Add R5, R4, R6**

**L'istruzione MUL  
carica in R4 il  
valore del  
risultato in questo  
colpo di clock**

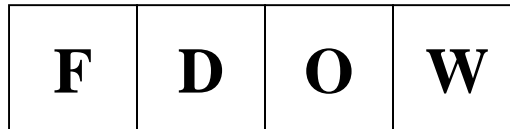
**Periodo di clock**    1    2    3    4    5    6    7    8

**Istruzione**

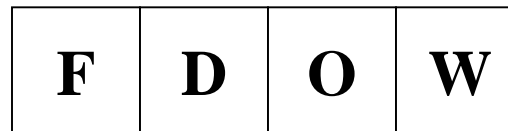
**Mul**



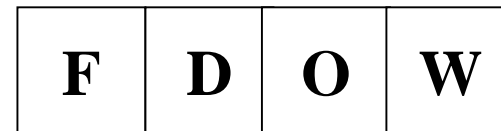
**Add**



**I<sub>3</sub>**



**I<sub>4</sub>**



# Esempio

**Mul R2, R3, R4**

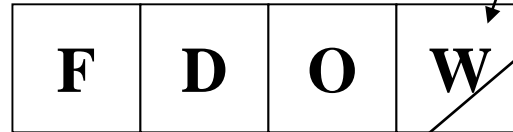
**Add R5, R4, R6**

**L'istruzione MUL  
carica in R4 il  
valore del  
risultato in questo  
colpo di clock**

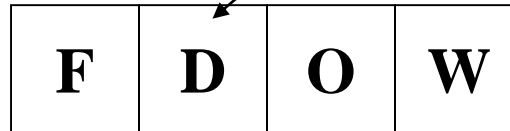
**Periodo di clock**    1    2    3    4    5    6    7    8

**Istruzione**

**Mul**

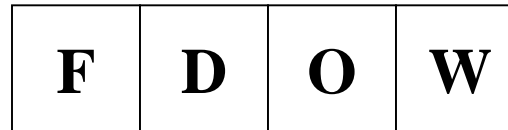


**Add**

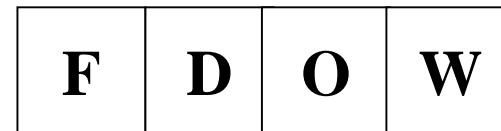


**L'istruzione ADD  
legge da R4 il valore  
dell'operando in  
questo colpo di  
clock**

**I<sub>3</sub>**



**I<sub>4</sub>**



# Esempio

L'istruzione MUL  
carica in R4 il  
valore del

questo  
clock

## Conclusione

Pe

Ist È necessario garantire la correttezza del risultato.

M Questo può essere fatto

- in HW
- in SW.

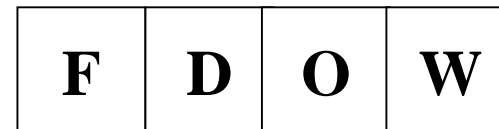
ADD  
valore  
do in  
o di

Ad

I<sub>3</sub>



I<sub>4</sub>



# Soluzione HW

**L'HW della pipeline è in grado di**

- riconoscere le situazioni di dipendenza dei dati**
- inibire il funzionamento di specifici moduli della pipeline (introducendo degli stalli) per garantire la correttezza del risultato.**

# Esempio

**Mul**    **R2, R3, R4**

**Add**    **R5, R4, R6**

**Periodo di clock**    **1      2      3      4      5      6      7      8**

**Istruzione**

**Mul**

<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>
----------	----------	----------	----------

**Add**

<b>F</b>			<b>D</b>	<b>O</b>	<b>W</b>
----------	--	--	----------	----------	----------

**I<sub>3</sub>**

		<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>
--	--	----------	----------	----------	----------

**I<sub>4</sub>**

<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>
----------	----------	----------	----------

# Esem

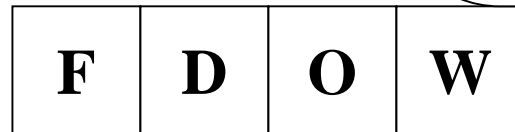
**Mul R2, R3, R4**

**Add R5, R4, R6**

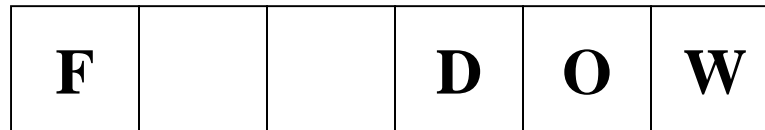
**Periodo di clock 1 2 3**

**Istruzione**

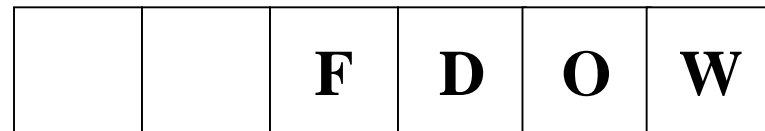
**Mul**



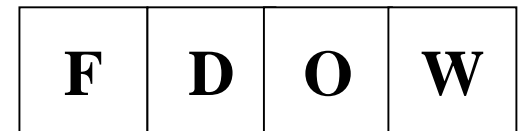
**Add**



**I<sub>3</sub>**



**I<sub>4</sub>**



**La fase di Decode non può essere eseguita sino a che non è stata eseguita la fase di Write dell'istruzione precedente, che produce il valore aggiornato di R4.**



# Soluzione SW

**Il compilatore introduce delle istruzioni NOP, che permettono alla pipeline di produrre un risultato corretto.**

## Esempio

**Codice originale**

**$I_i$**

**Mul    R2, R3, R4**

**Add   R5, R4, R6**

**$I_j$**

**$I_k$**

**...**

**codice modificato**

**$I_i$**

**Mul    R2, R3, R4**

**NOP**

**Nop**

**Add   R5, R4, R6**

**$I_j$**

# Esempio

**Mul**      **R2, R3, R4**

**Nop**

**Nop**

**Add**      **R5, R4, R6**

**Periodo di clock**   **1**      **2**      **3**      **4**      **5**      **6**      **7**      **8**

**Istruzione**

**Mul**

<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>
----------	----------	----------	----------

**Nop**

<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>
----------	----------	----------	----------

**Nop**

<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>
----------	----------	----------	----------

**Add**

<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>
----------	----------	----------	----------

# Istruzioni di salto

**Le istruzioni di salto sono potenzialmente molto dannose per le prestazioni della pipeline, in quanto interrompono il normale flusso di esecuzione sequenziale.**

# Esempio

L'istruzione JMP  
carica nel PC il  
nuovo valore in  
questo colpo di  
clock

$I_i$   
JMP lab  
 $I_h$   
 $I_j$   
lab:  $I_k$

Periodo di clock    1    2    3    4    5    6    7    8

Istruzione

$I_i$

F	D	O	W
---	---	---	---

JMP lab

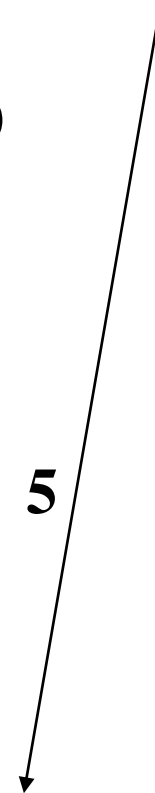
F	D	O	W
---	---	---	---

$I_h$

F	D	O	W
---	---	---	---

$I_j$

F	D	O	W
---	---	---	---



# Esempio

lab:  $I_i$   
 $I_h$   
 $I_j$   
 $I_k$

Periodo di clock 1 2 3 4 5 6

Istruzione

$I_i$

F	D	O	W
---	---	---	---

JMP lab

F	D	O	W
---	---	---	---

$I_h$

F	D	O	W
---	---	---	---

$I_j$

F	D	O	W
---	---	---	---

L'istruzione JMP  
 carica nel PC il  
 nuovo valore in  
 questo colpo di  
 clock

L'unità di fetch  
 carica dalla  
 memoria  
 l'istruzione  $I_h$   
 anziché  $I_k$

# Esempio

lab:  $I_i$   
 $I_h$   
 $I_j$   
 $I_k$   
 JMP lab

Periodo di clock 1 2 3 4 5 6

Istruzione

$I_i$

F	D	O	W
---	---	---	---

JMP lab

F	D	O	W
---	---	---	---

$I_h$

L'unità di fetch  
 carica l'istruzione  $I_j$   
 che non doveva  
 essere eseguita

$I_j$

F	D	O	W
---	---	---	---

F	D	O	W
---	---	---	---

L'istruzione JMP  
 carica nel PC il  
 nuovo valore in  
 questo colpo di  
 clock

L'unità di fetch  
 carica dalla  
 memoria  
 l'istruzione  $I_j$   
 anziché  $I_h$

# Intervallo di ritardo del salto

**Viene denominato intervallo di ritardo del salto (*branch delay slot*) il numero di colpi di clock successivi ad un'istruzione di salto, nei quali il processore esegue il fetch usando un valore del PC che non è necessariamente corretto.**

# Soluzioni

**Per garantire la correttezza dei risultati prodotti dal processore anche in presenza di salti si possono anche qui utilizzare**

- **tecniche HW**
- **tecniche SW.**



# Tecniche HW

**Possono**

- **rilevare le istruzioni di salto, e svuotare conseguentemente la pipeline dalle istruzioni erroneamente caricate**

**oppure**

- **implementare tecniche più sofisticate, che dipendono dal fatto che il salto sia incondizionato o condizionato.**

# Esempio

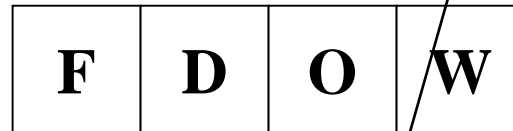
La CPU a questo istante capisce che l'istruzione corrente è di salto condizionato e interrompe i fetch, inserendo stalli

lab:  $I_i$   
 $I_h$   
 $I_j$   
 $I_k$

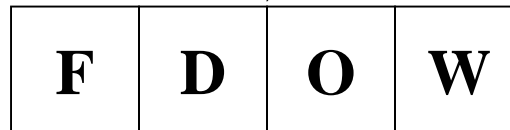
Periodo di clock 1 2 3 4 5 6 7 8

Istruzione

$I_i$



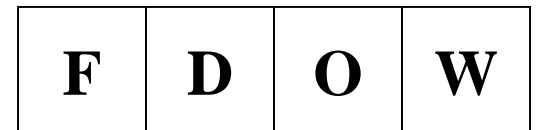
JMP lab



$I_h$



$I_k$



...



A questo istante viene caricato il nuovo valore di PC e riparte il fetch

# Tecniche SW

Sono basate su

- *Salto ritardato*

introduzione di istruzioni NOP da parte del compilatore (dopo ogni salto si inseriscono tante NOP quanto è il branch delay slot)

oppure

- un'ottimizzazione del codice.

# Soluzione SW

**Il compilatore introduce delle istruzioni NOP, che permettono alla pipeline di produrre un risultato corretto.**

## Esempio

**Codice originale**

**I<sub>a</sub>**

**I<sub>b</sub>**

**I<sub>c</sub>**

**I<sub>d</sub>**

**JMP lab\_1**

**I<sub>j</sub>**

**Lab\_1: I<sub>k</sub>**

**codice modificato**

**I<sub>a</sub>**

**I<sub>b</sub>**

**I<sub>c</sub>**

**I<sub>d</sub>**

**JMP lab\_1**

**NOP**

**NOP**

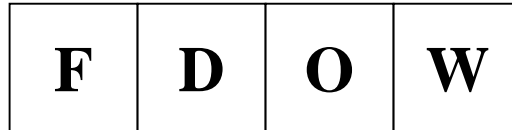
**NOP**

**I<sub>j</sub>**

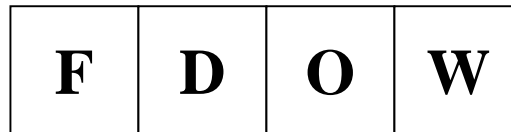
**Lab\_1: I<sub>k</sub>**

# Inserimento di Nop

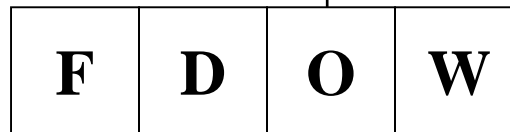
$I_d$



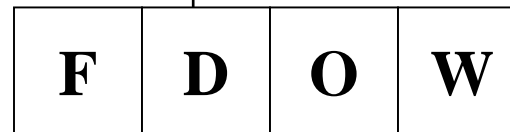
JMP



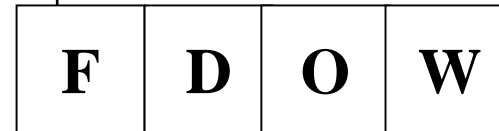
Nop



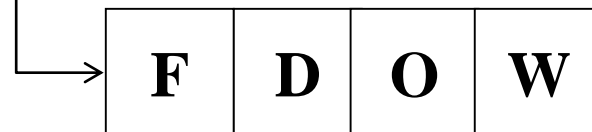
Nop



Nop



$I_k$



In questo colpo di  
clock  
l'istruzione JMP  
carica nel PC il  
nuovo valore,  
corrispondente  
all'indirizzo di  $I_k$

# Ottimizzazione

L'istruzione di salto viene anticipata ed eseguita prima di un numero istruzioni (se possibile) pari al *branch delay slot*, in modo da non far eseguire al processore le istruzioni NOP.

$I_a$

F	D	O	W
---	---	---	---

JMP

F	D	O	W
---	---	---	---

$I_b$

F	D	O	W
---	---	---	---

$I_c$

F	D	O	W
---	---	---	---

$I_d$

F	D	O	W
---	---	---	---

$I_k$

F	D	O	W
---	---	---	---

# **Salti condizionati**

**In questo caso è necessario attendere l'esecuzione dell'istruzione precedente a quella di salto per sapere se il salto deve essere eseguito o meno.**

**Nel frattempo però la coda delle istruzioni è stata riempita sequenzialmente.**

**Se il salto viene eseguito, vi saranno alcune istruzioni provenienti dalla coda che non dovranno essere eseguite (tante quante la dimensione del branch delay slot).**

# Predizione di salto

**Se si usa una tecnica HW, la predizione permette di ridurre la penalizzazione introdotta da un'istruzione di salto.**

**Consiste nel far sì che l'unità di fetch disponga di una previsione sul fatto che il salto venga effettuato o meno.**

**Se la previsione è affermativa, essa carica le istruzioni a partire da quella destinazione; se è negativa, continua a caricare le istruzioni successive a quella di salto.**

**Successivamente, si verificherà se la previsione era corretta; in caso negativo si deve svuotare la pipeline.**



# I processori RISC

A partire dai primi anni '80 furono progettati e poi realizzati alcuni processori caratterizzati da

- Elevata semplicità
- Numero ridotto di istruzioni (da cui la sigla RISC, che sta per *Reduced Instruction Set Computer*)
- Elevato numero di registri
- Architettura a pipeline.

Negli anni '90 esistevano sul mercato varie famiglie di processori RISC, tra cui

- ARM
- SPARC (Sun)
- MIPS
- PowerPC (IBM).

# **Caratteristiche dei RISC:**

## **1 istruzione = 1 colpo di clock**

**Ogni istruzione corrisponde a varie operazioni:**

- **fetch degli operandi dai registri**
- **attivazione della ALU**
- **memorizzazione del risultato in un registro.**

**Grazie all'adozione della pipeline, è possibile sovrapporre temporalmente l'esecuzione di più istruzioni, in modo che ad ogni colpo di clock si termini normalmente l'esecuzione di una istruzione.**

**Per poter utilizzare al meglio la pipeline è necessario che le istruzioni siano**

- **Semplici**
- **Regolari (formato fisso).**

# **Caratteristiche dei RISC: unità di controllo**

**Le istruzioni RISC hanno la complessità delle microistruzioni CISC; per questa ragione l'unità di controllo dei RISC può essere realizzata con la tecnica cablata, anziché microprogrammata.**

# **Caratteristiche dei RISC: LOAD & STORE**

**I processori RISC hanno normalmente due sole istruzioni che coinvolgono la memoria**

- LOAD (memoria  $\Rightarrow$  registro)**
- STORE (registro  $\Rightarrow$  memoria).**

**Questo permette di concentrare su 2 sole istruzioni gli sforzi per ridurre l'impatto negativo legato al calcolo dell'indirizzo e all'accesso in memoria.**

**Tali istruzioni sono le uniche a prevedere dei meccanismi per specificare l'indirizzo a cui accedere in memoria (modi di indirizzamento).**

# **Caratteristiche dei RISC: formato delle istruzioni**

**I RISC hanno un formato delle istruzioni fisso o con poche alternative: il codice operativo ha di solito una lunghezza fissa.**

**Ne conseguono alcuni vantaggi:**

- la decodifica del codice operativo può avvenire in parallelo con il caricamento degli operandi dai registri**
- l'unità di controllo è più semplice**
- la fase di fetch è più ottimizzata.**

# **Caratteristiche dei RISC: modi di indirizzamento**

**I RISC possiedono un numero limitato di modi di indirizzamento, che comunque vengono utilizzati solo nelle istruzioni LOAD e STORE.**

# **I registri**

**I registri sono la forma di memoria con minore tempo di accesso in quanto:**

- **risiedono sullo stesso chip della CPU**
- **sono costruiti con la tecnologia più veloce**
- **sono accessibili con un meccanismo di indirizzamento semplice.**

**Si può guadagnare in efficienza di esecuzione in 2 modi:**

- **aumentando il numero di registri**
- **ottimizzando il loro uso.**

**I registri sostituiscono lo stack (ad es. per la memorizzazione dei parametri per le procedure, delle variabili locali e dell'indirizzo di ritorno), riducendo il numero di accessi nello stack (ossia memoria).**

# Dimensione del codice eseguibile

- I compilatori per RISC sono più complessi di quelli per i CISC (pur producendo codice composto da istruzioni più semplici) in quanto devono ottimizzare il codice per evitare gli stalli
- Il codice generato per un RISC ha dimensioni comparabili con quelle per un CISC in quanto:
  - il numero di istruzioni RISC generate è maggiore, ma
  - ogni istruzione occupa un numero inferiore di byte
- Codici più corti sono più efficienti perché:
  - si riduce il numero di fetch
  - si riduce il numero di page fault.



# Conclusioni sull'approccio RISC

- Le istruzioni CISC, per quanto più complesse, non sono più veloci, in quanto richiedono un hardware più complicato
- La gestione ottimizzata della pipeline delle istruzioni è più semplice con i RISC
- La disponibilità di un elevato numero di registri permette di ridurre significativamente il numero di accessi in memoria
- Nei RISC il tempo medio di attesa per il servizio di un interrupt (*latenza*) è inferiore rispetto ai CISC.

# **I processori superscalari**

**I processori superscalari possono completare l'esecuzione di più di un'istruzione per ciascun colpo di clock.**

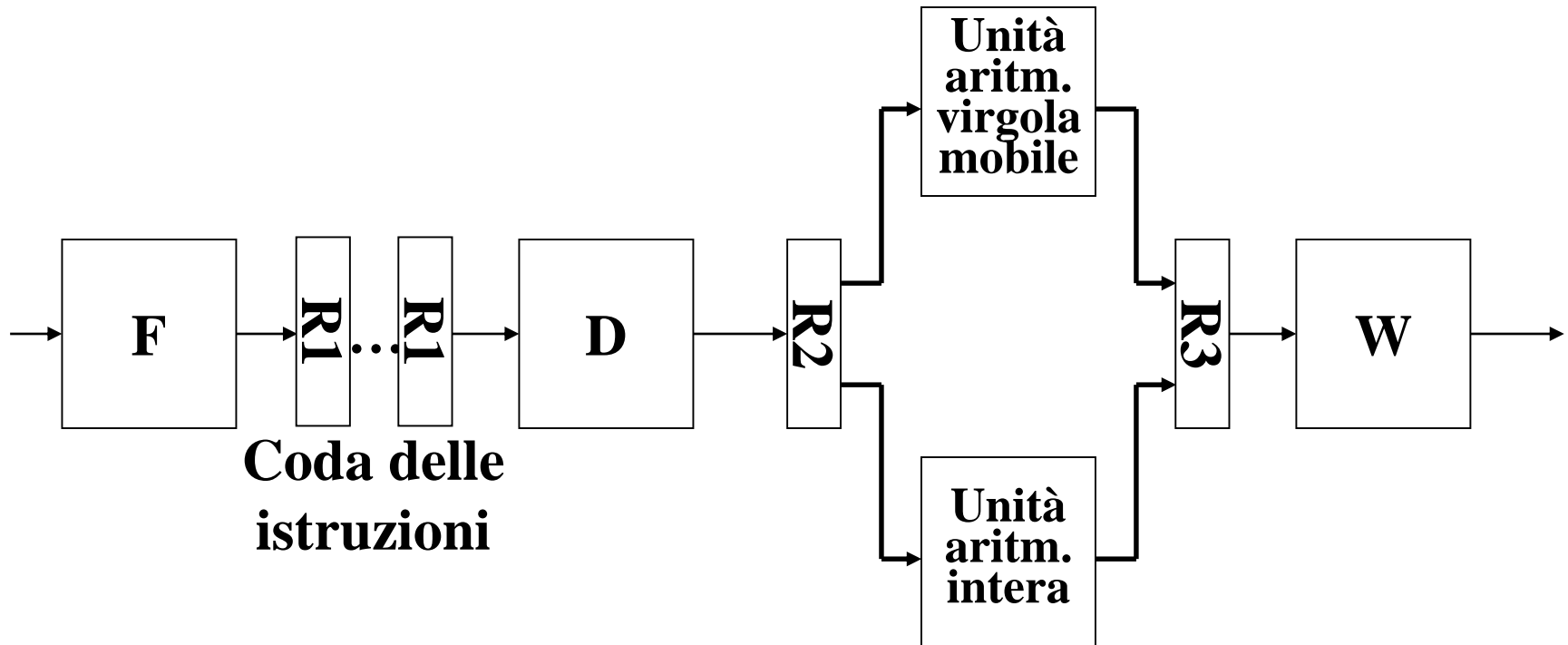
**Internamente, contengono una pipeline in cui a ciascuno stadio corrisponde più di una unità.**

# Comportamento ideale

Periodo di clock    1      2      3      4      5      6      7

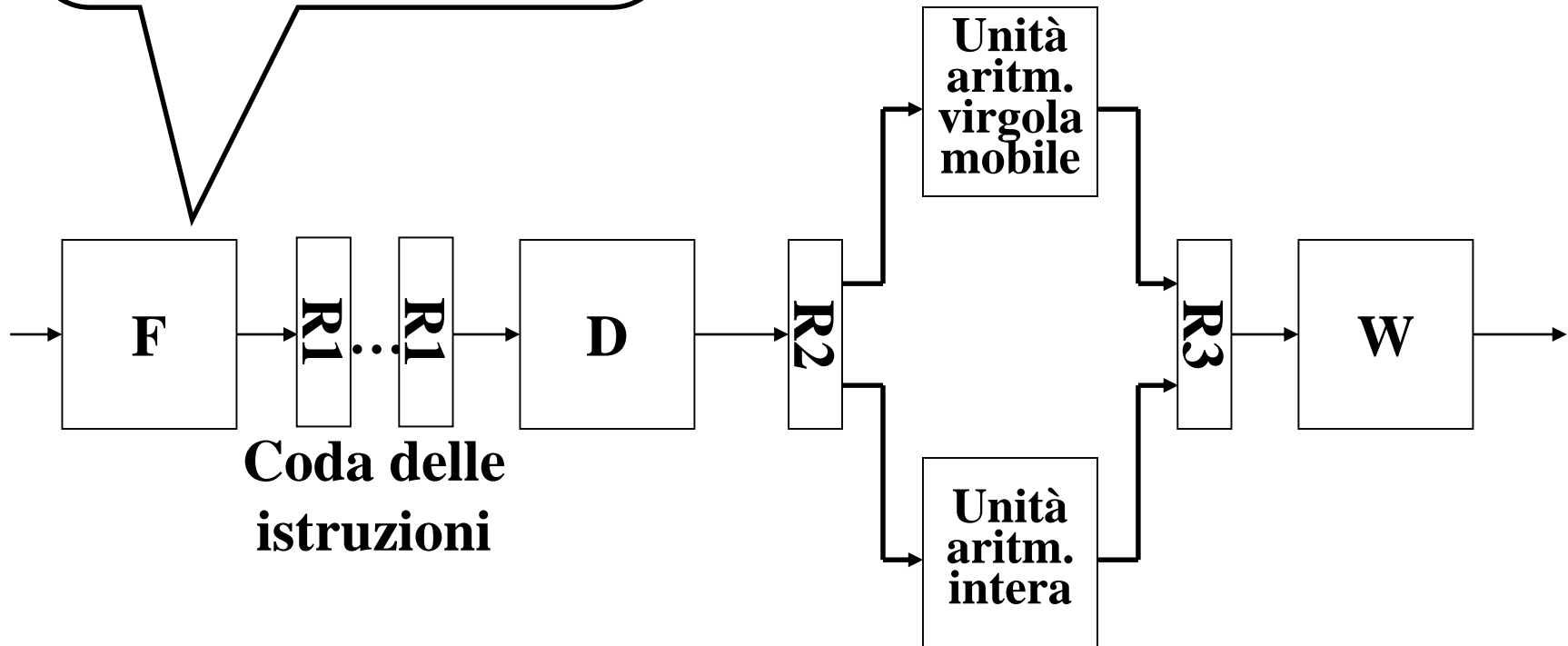
<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>			
<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>			
	<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>		
	<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>		
		<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>	
		<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>	
			<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>
			<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>

# Esempio



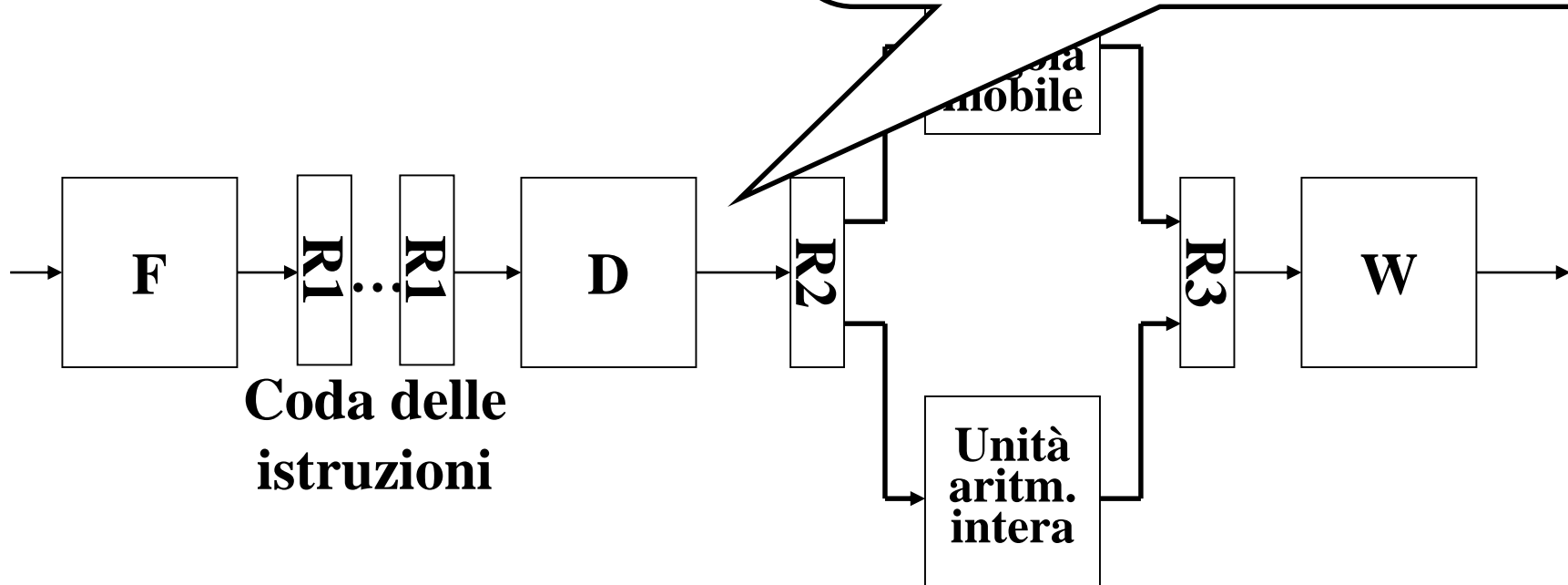
**L'unità di fetch può  
caricare 4 istruzioni  
per colpo di clock**

## **Esempio**

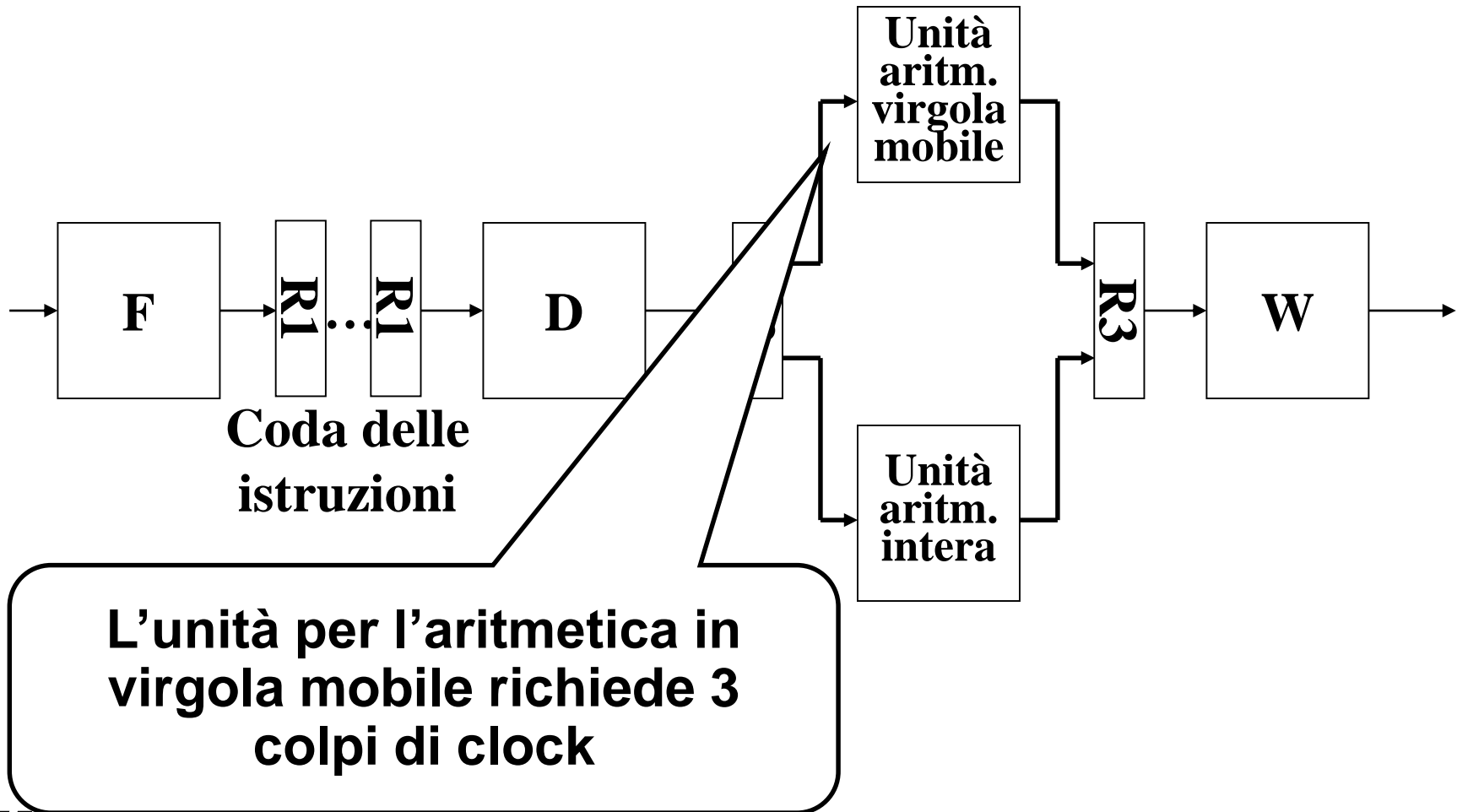


**Ese**

L'unità di decodifica può decodificare 2 istruzioni per colpo di clock e, se sono adatte, smistarle alle due unità aritmetiche dopo aver caricato i relativi operandi.

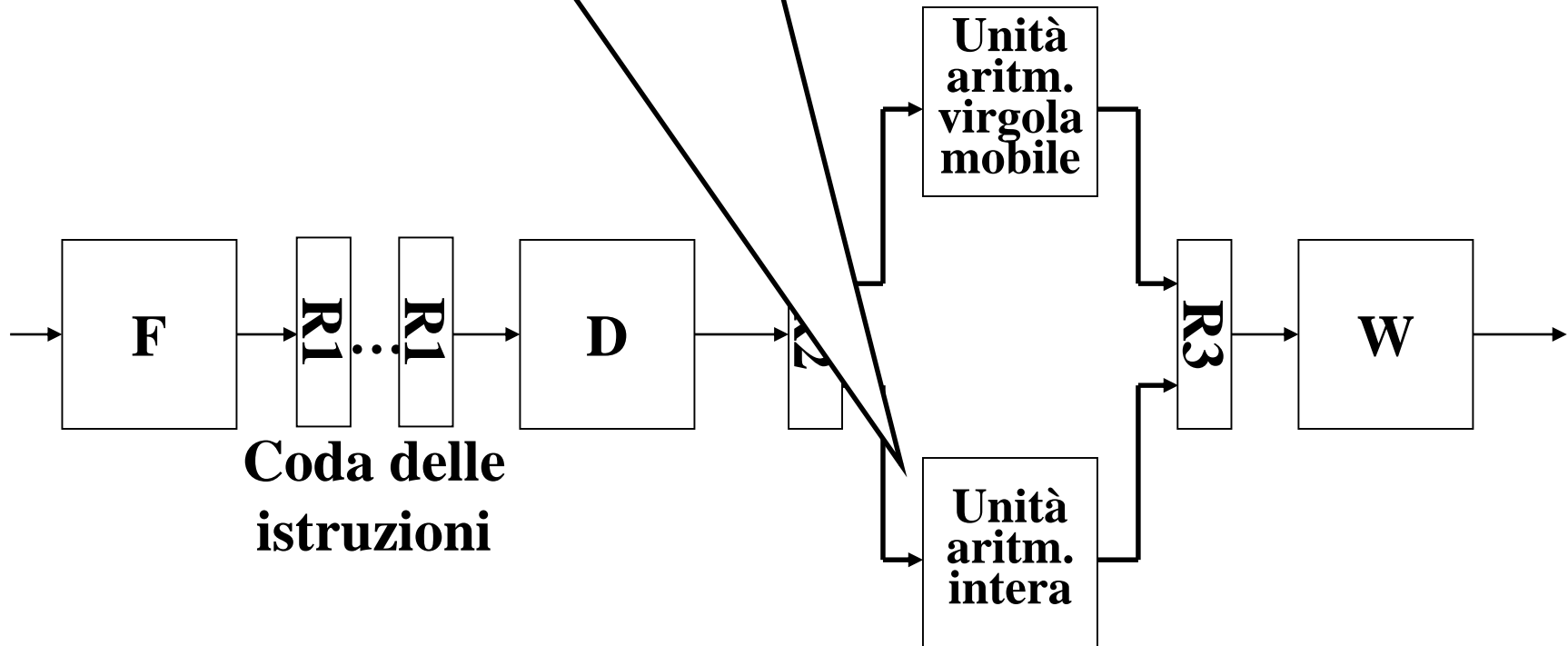


# Esempio



**L'unità per l'aritmetica in virgola mobile richiede 1 colpo di clock**

**pio**





# Flusso di esecuzione

Periodo di clock    1    2    3    4    5    6    7    8

Istruzione

**I<sub>1</sub> (Fadd)**

<b>F</b>	<b>D</b>	<b>O</b>	<b>O</b>	<b>O</b>	<b>W</b>
----------	----------	----------	----------	----------	----------

**I<sub>2</sub> (Add)**

<b>F</b>	<b>D</b>	<b>O</b>	<b>W</b>
----------	----------	----------	----------

**I<sub>3</sub> (Fsub)**

<b>F</b>		<b>D</b>	<b>O</b>	<b>O</b>	<b>O</b>	<b>W</b>
----------	--	----------	----------	----------	----------	----------

**I<sub>4</sub> (Sub)**

<b>F</b>		<b>D</b>	<b>O</b>	<b>W</b>
----------	--	----------	----------	----------

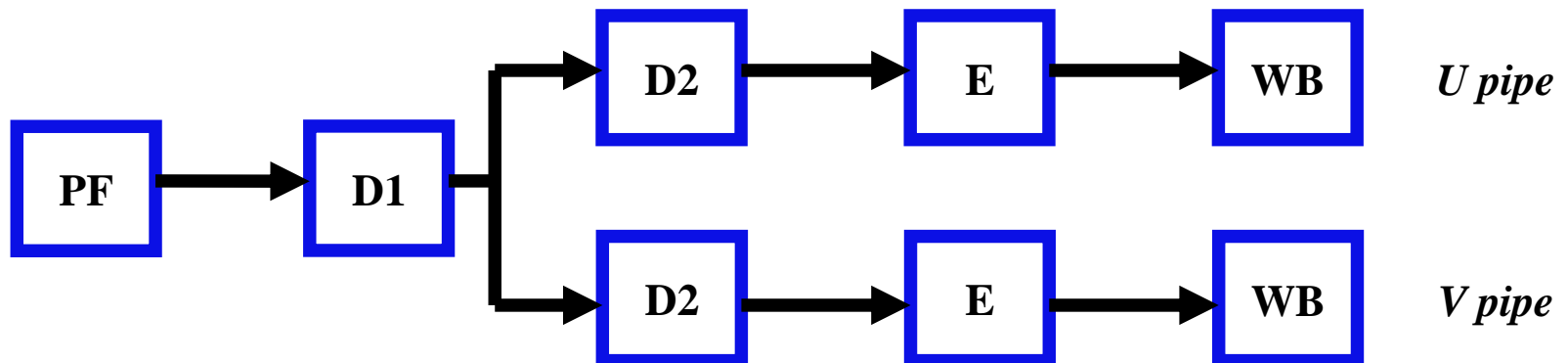
# Note

- Le unità appartenenti allo stesso stadio possono anche essere funzionalmente uguali
- La gestione della pipeline deve garantire la correttezza delle operazioni svolte, tenendo conto delle dipendenze tra dati
- L'esempio riporta un caso di *completamento non-in-ordine* delle istruzioni.

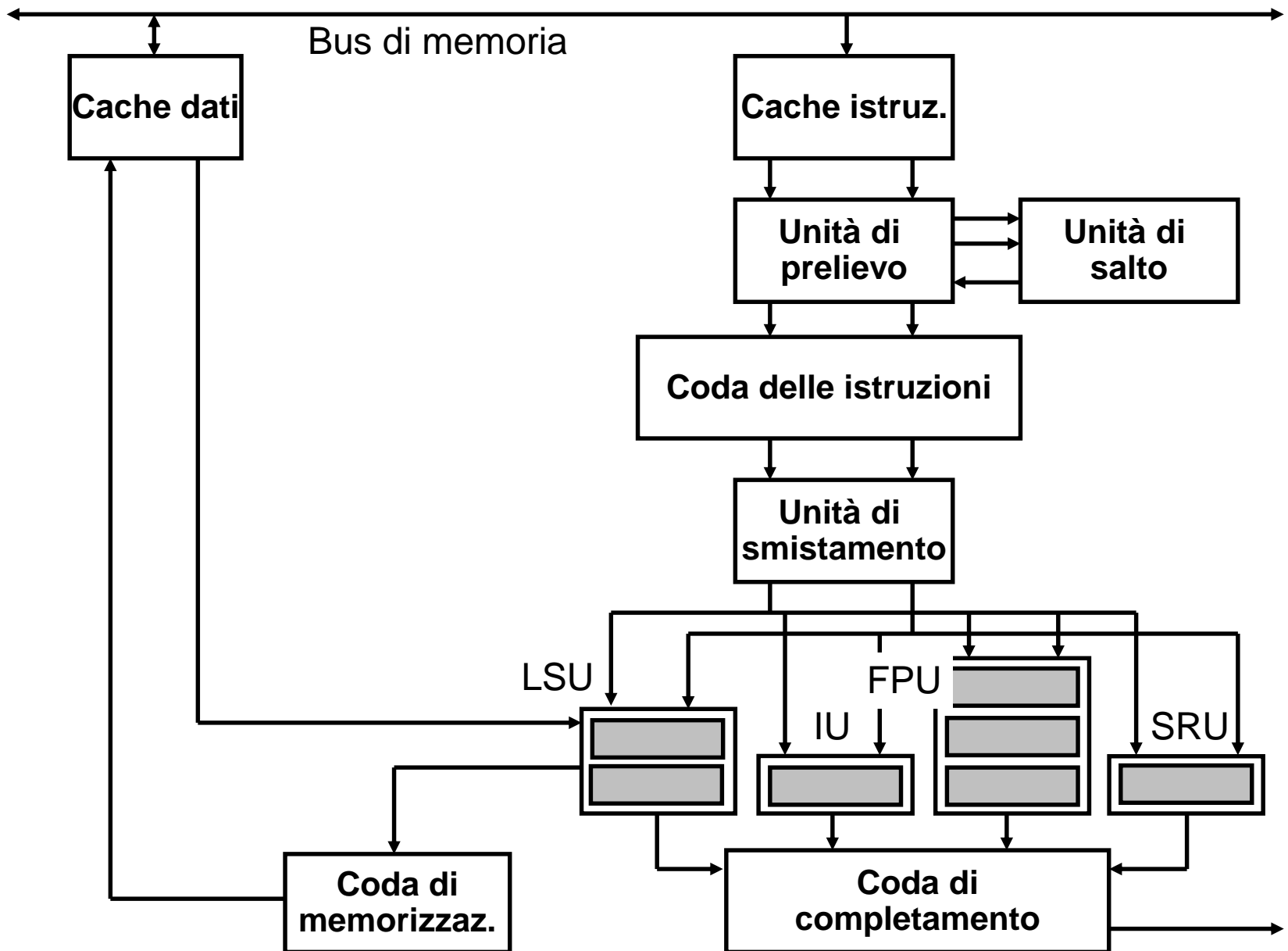
# Esempio: Intel Pentium

Processore Intel, della famiglia x86, prodotto dal 1993:

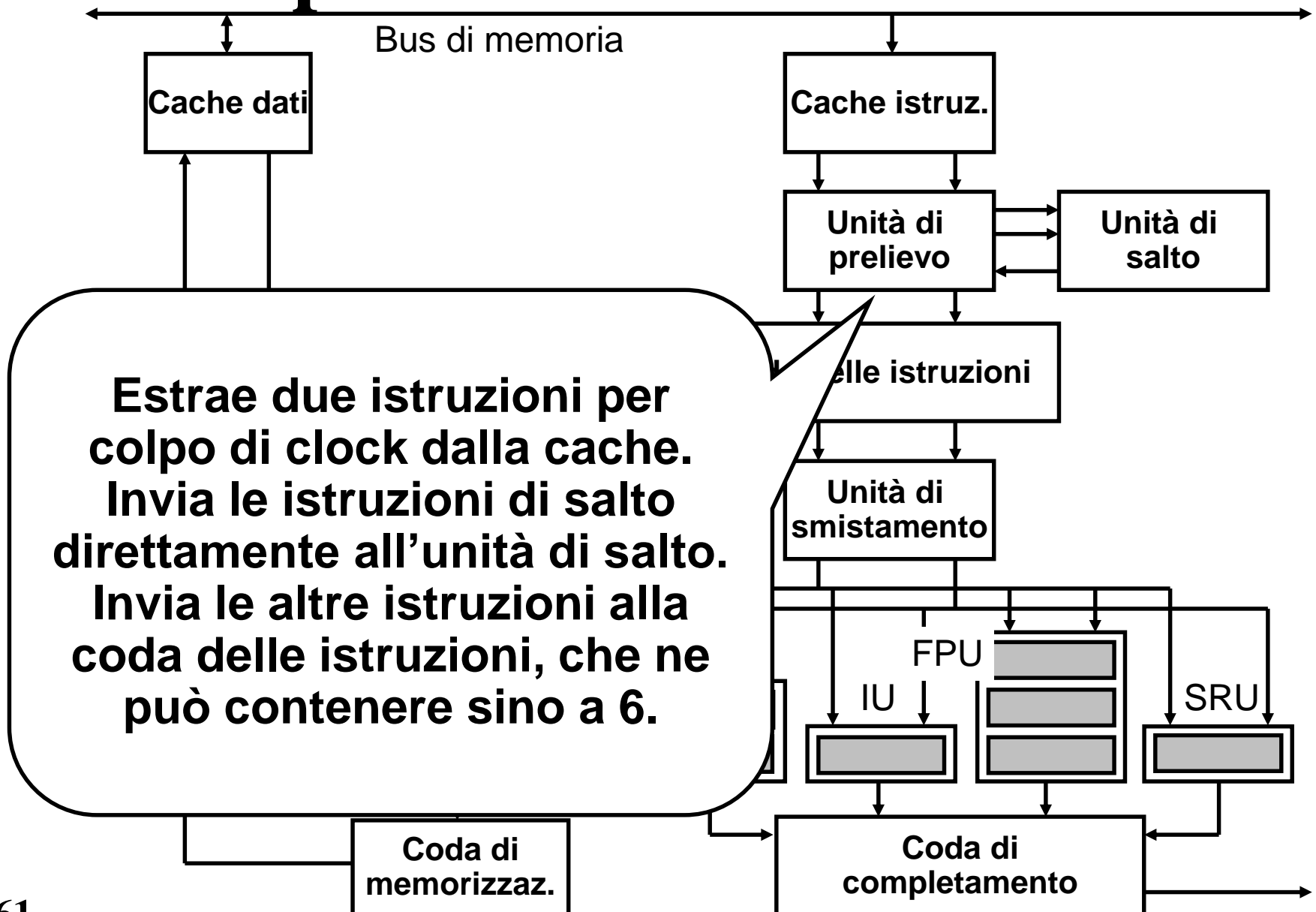
- possedeva 2 pipeline che gli permettevano di completare più di una operazione per ciclo di clock:
  - "*pipeline U*" poteva eseguire qualunque istruzione,
  - "*pipeline V*" era in grado di eseguire solo quelle più semplici e comuni (logica cablata).



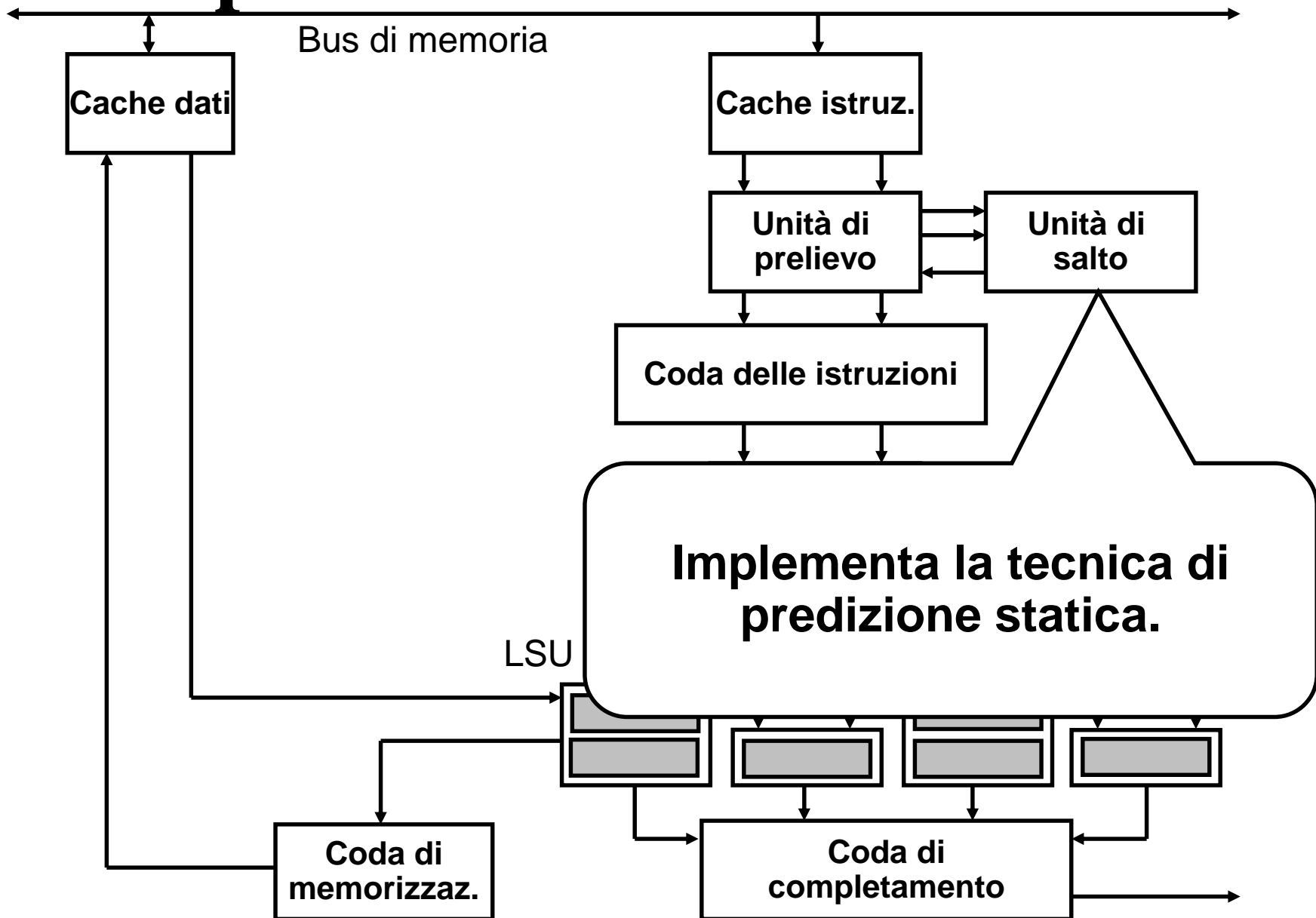
# Architettura del PowerPC 603



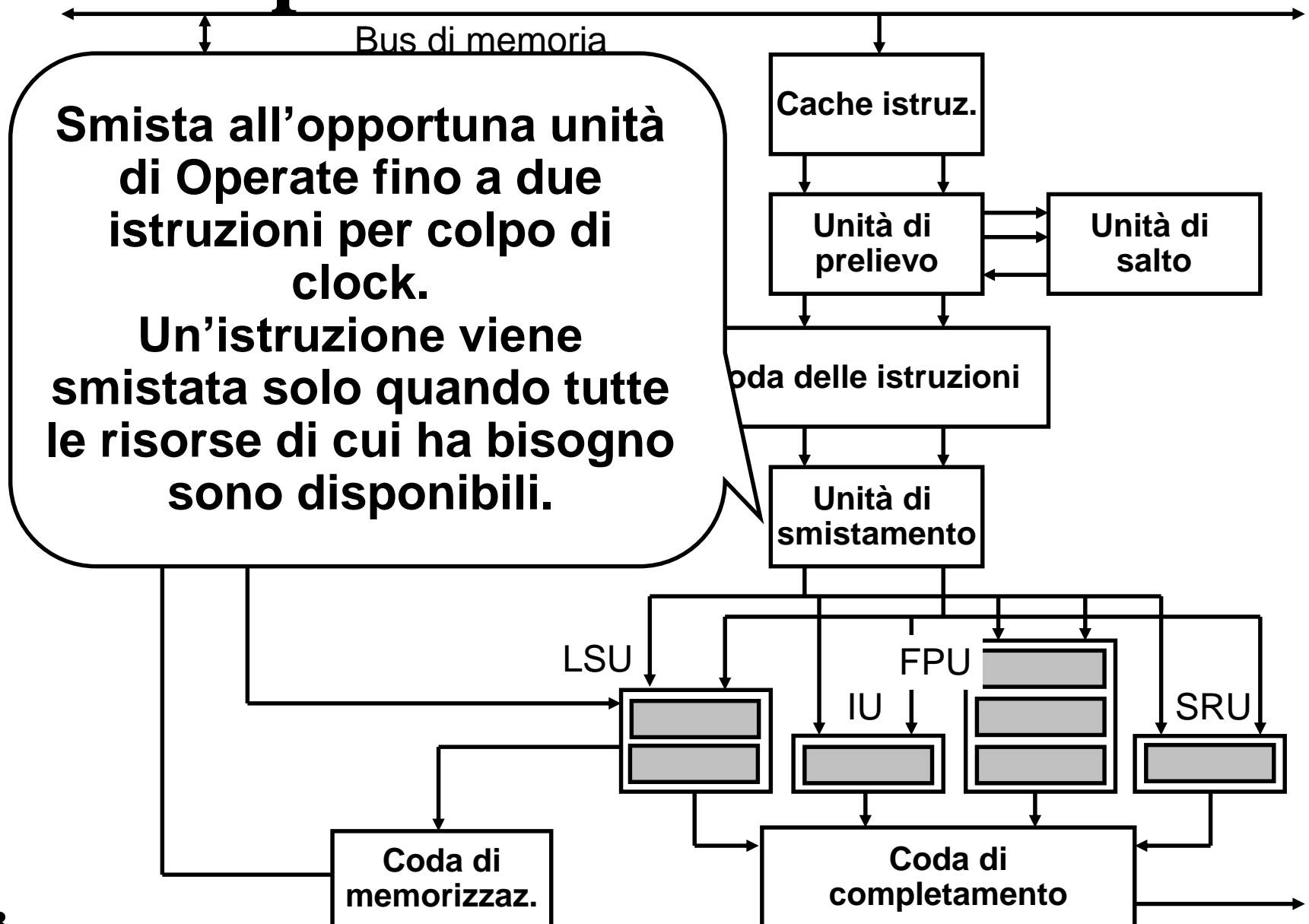
# Pipeline del PowerPC 603



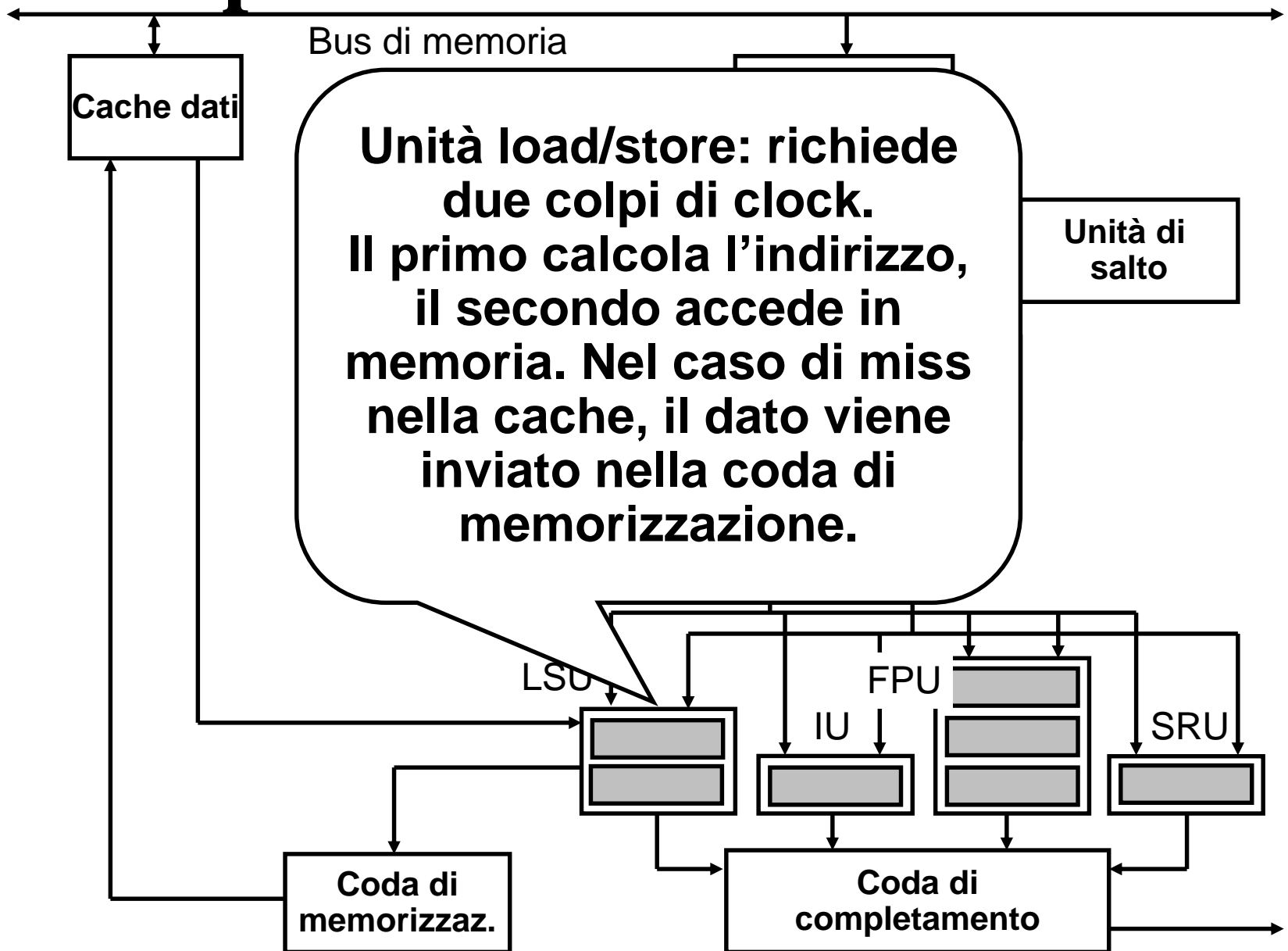
# Pipeline del PowerPC 603



# Pipeline del PowerPC 603

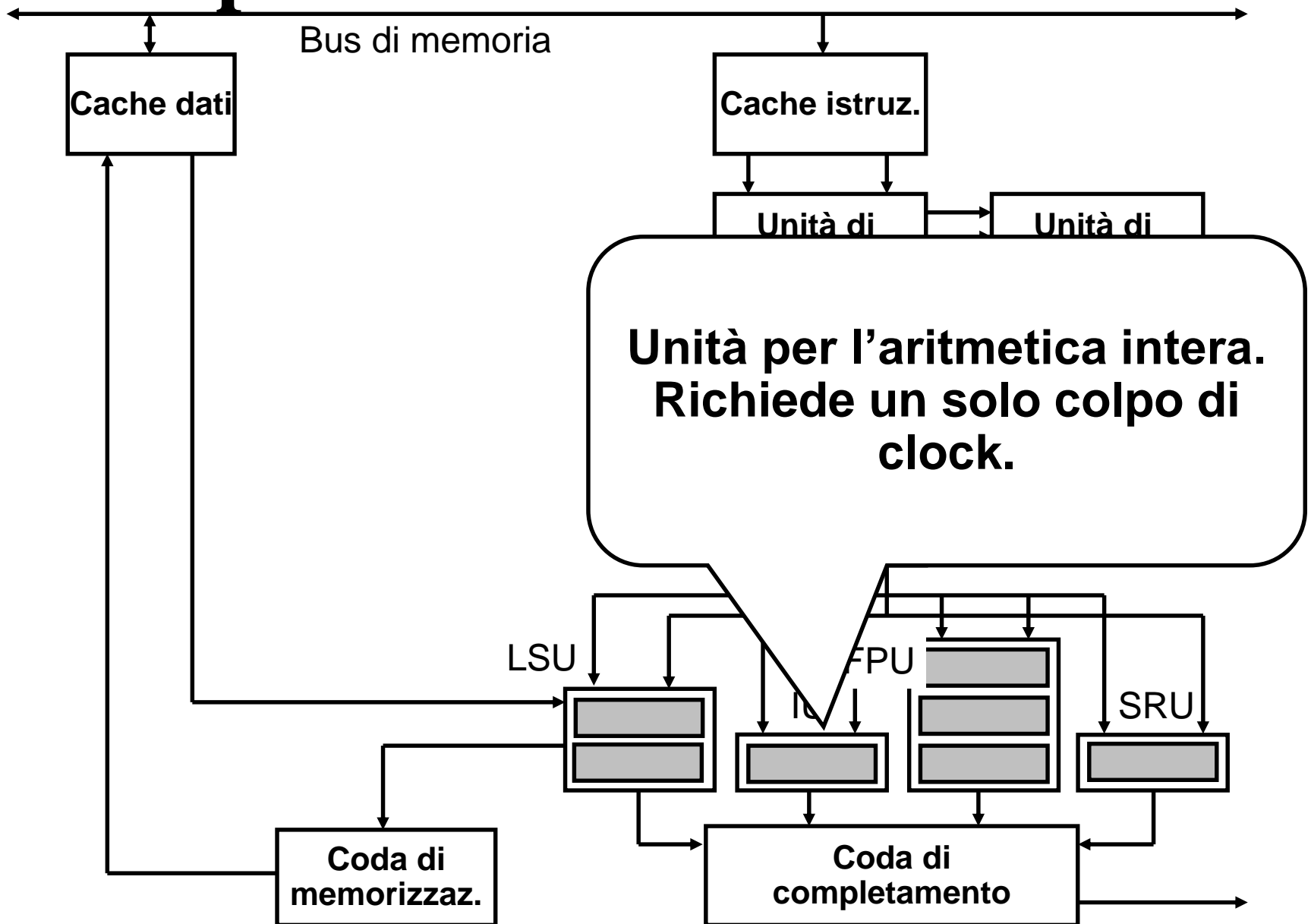


# Pipeline del PowerPC 603

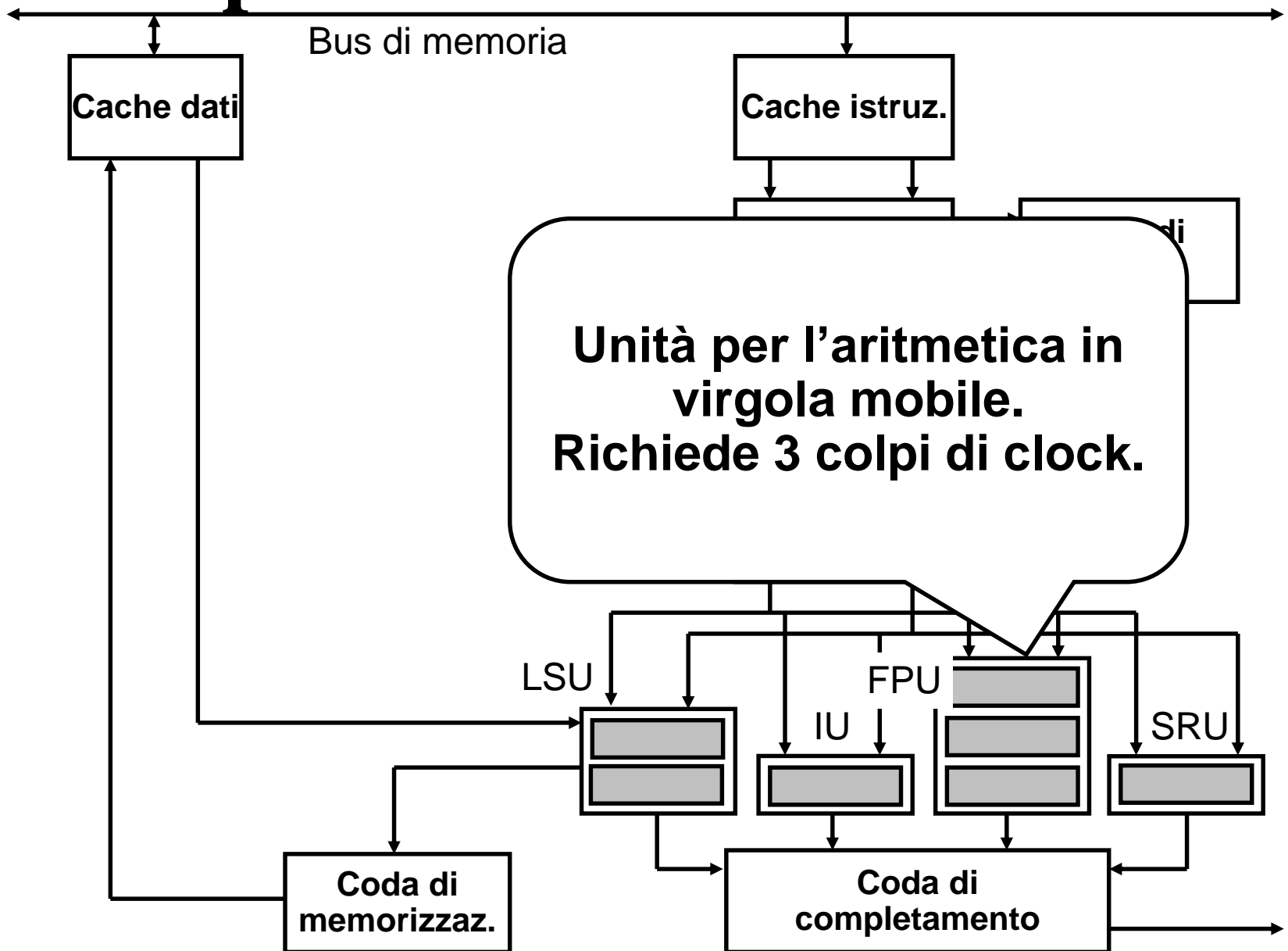




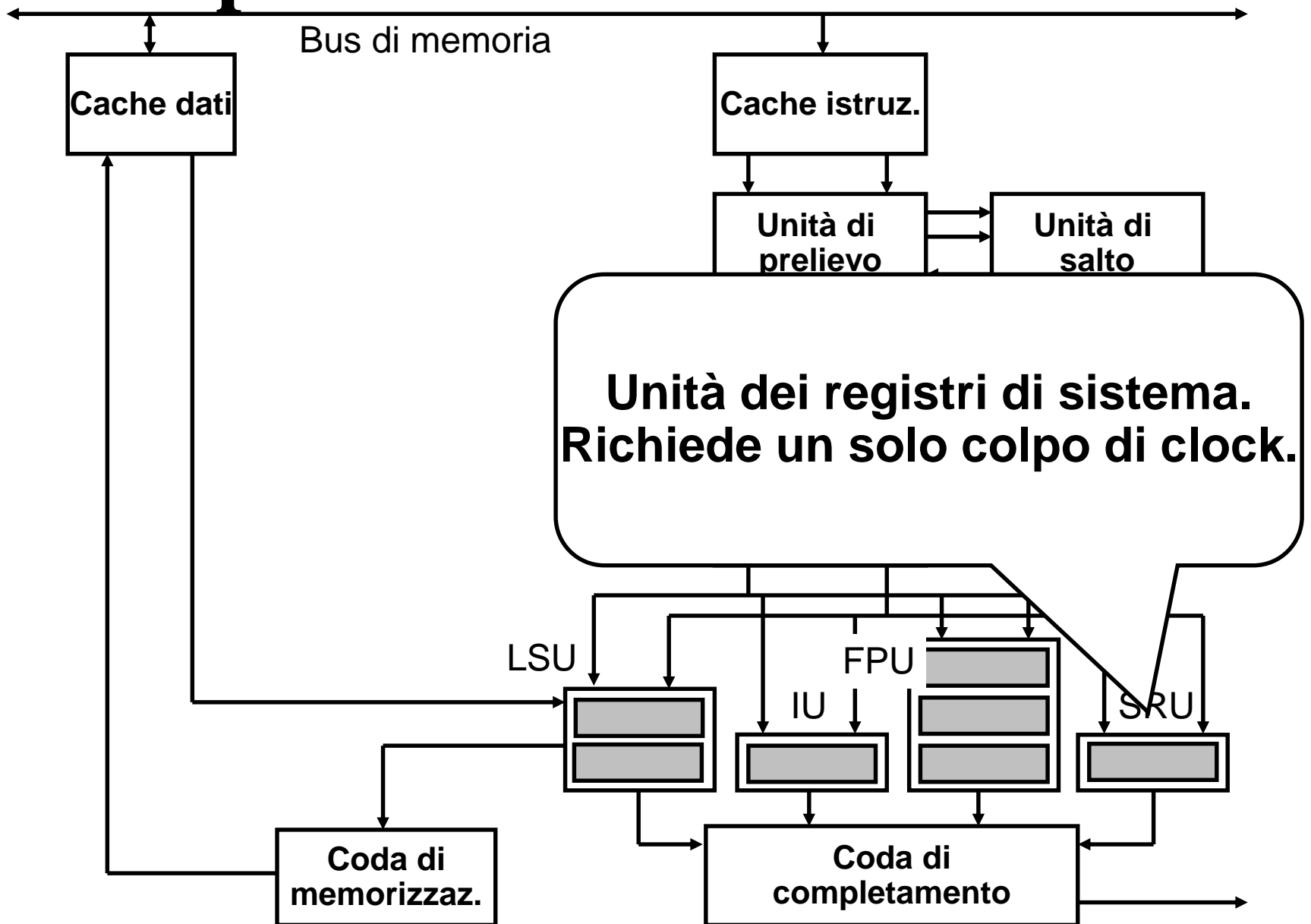
# Pipeline del PowerPC 603



# Pipeline del PowerPC 603



# Pipeline del PowerPC 603



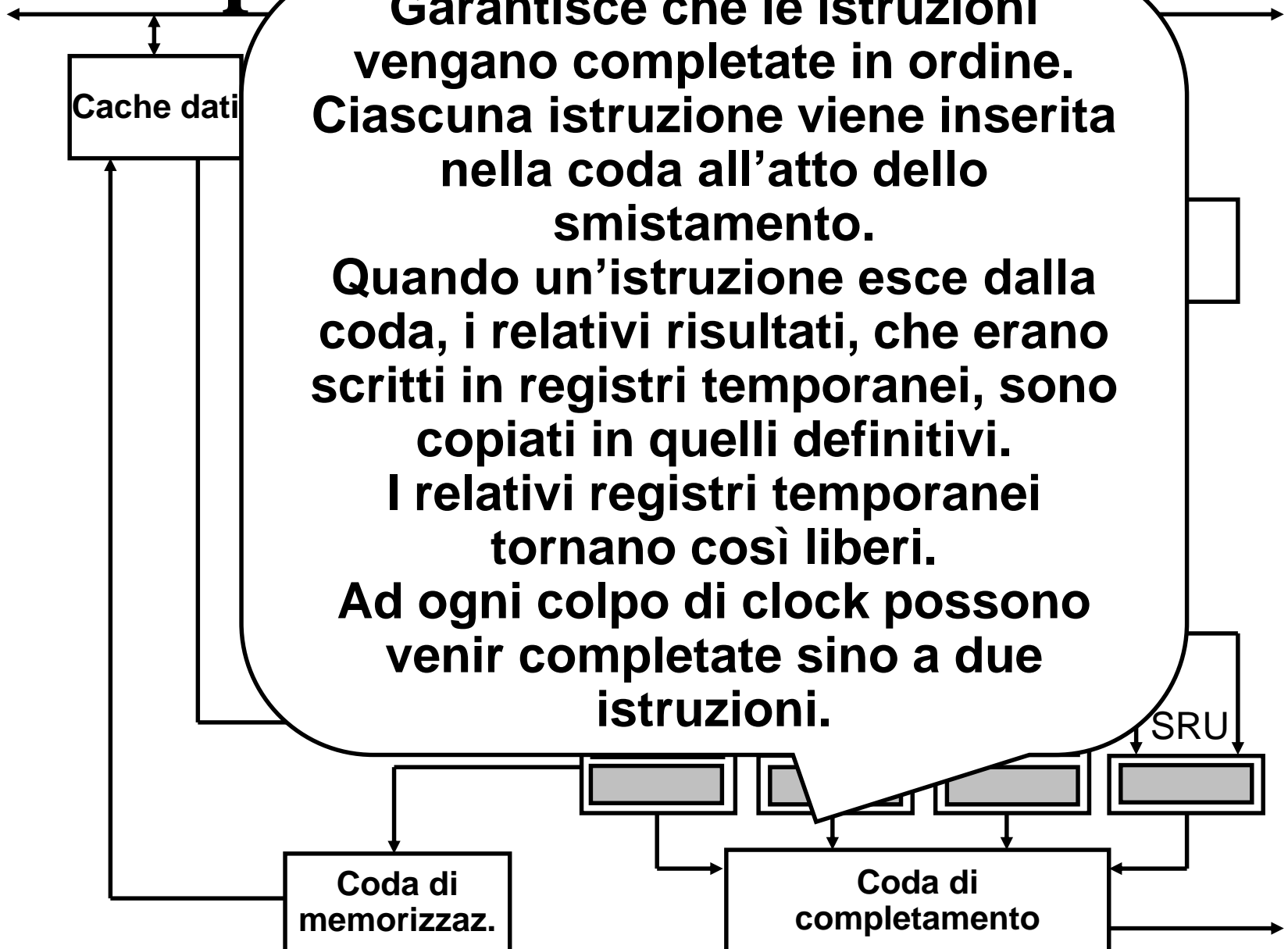
# Pipeline del PowerPC 603

**Garantisce che le istruzioni vengano completate in ordine. Ciascuna istruzione viene inserita nella coda all'atto dello smistamento.**

**Quando un'istruzione esce dalla coda, i relativi risultati, che erano scritti in registri temporanei, sono copiati in quelli definitivi.**

**I relativi registri temporanei tornano così liberi.**

**Ad ogni colpo di clock possono venir completate sino a due istruzioni.**



# **Numero di stadi della pipeline**

**Se cresce, permette teoricamente di aumentare le prestazioni del processore.**

**Ma crescendo il numero di stadi, cresce la probabilità di dipendenze, e quindi che la pipeline debba andare in stallo.**

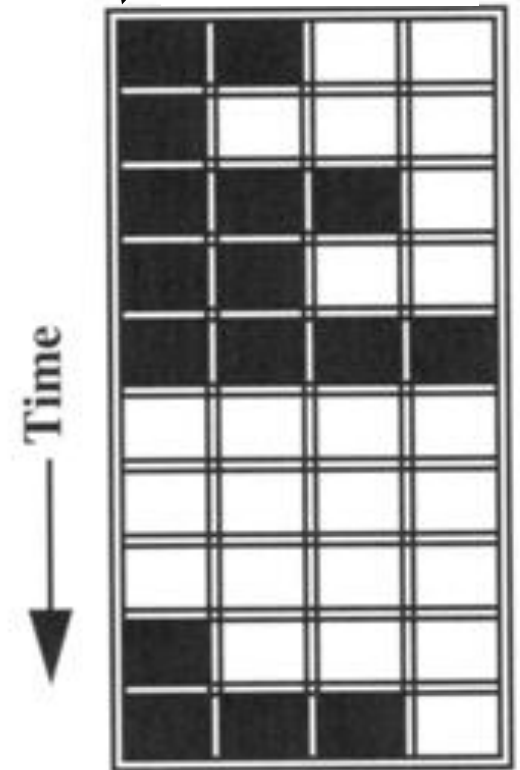
# Instruction Level Parallelism

I processori con pipeline sfruttano il fatto che le istruzioni di un programma possono in molti casi essere eseguite in parziale sovrapposizione (*Instruction Level Parallelism*, o ILP).

Le architetture dei processori superscalari sono in grado di sfruttare al meglio tale forma di parallelismo.

Ogni rettangolo nero rappresenta un'istruzione terminata nell'unità di tempo

ILP



# Processori multithread

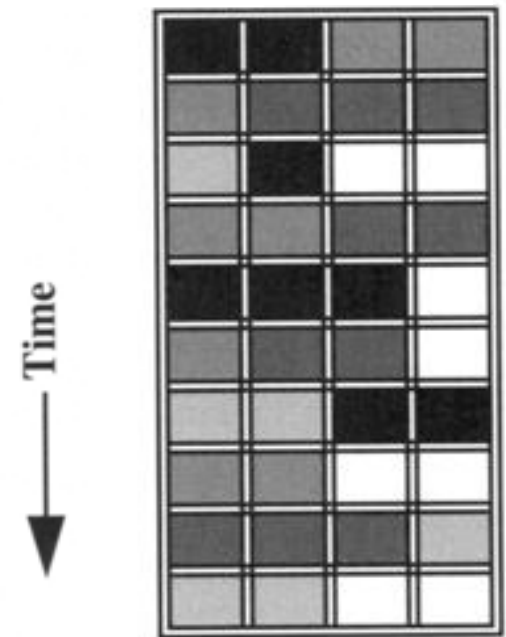
Per ottenere ulteriori incrementi di prestazioni è necessario individuare e sfruttare forme diverse di parallelismo, in particolare quello a livello di thread (*Thread Level Parallelism*, o TLP).

Un thread corrisponde ad una porzione di programma con i suoi dati e un programma può essere partizionato su più thread.

I processori multithread (*Simultaneous Multi-Threading*, SMT) eseguono in parallelo più thread.

Ogni colore (tranne il bianco) rappresenta un thread

SMT



# Processori multicore

**Un'altra tendenza degli ultimi anni è quella ad integrare nello stesso dispositivo più processori.**

**Ciascun processore ha una complessità non troppo elevata, ed è in grado di eseguire più thread.**

**In tal modo (se si riescono a sfruttare tutti i core) si ottengono prestazioni elevate con bassi consumi.**



# Esempio: Intel Sandy Bridge

**L'architettura Intel Sandy Bridge fu sviluppata a partire dal 2005. Il primo prodotto basato su quest'architettura fu commercializzato a partire dal 2011.**

## L'architettura permette di combinare fino a 8 core.

## Altre caratteristiche dell'architettura:

- **Cache L1: 32 kB data + 32 kB instruction per core**
- **Cache L2: 256 kB per core**
- **Cache L3 cache condivisa (fino a 20MB).**



### Intel "Sandy Bridge" (SNB) / Mainstream Quad-Core

32 mm Process / ~225 mm<sup>2</sup> Die Size / BSW TOP / AO Stamping / Tape Out: WW23709

Expected : 0f11 @ 3.0 - 3.875 GHz

# Esempio: ARM Cortex-A9

ARM produce IP core di processori per applicazioni embedded.

Il Cortex-A9 può essere utilizzato da solo o in configurazione multicore.

Rappresenta una soluzione per applicazioni in cui sono richieste alte prestazioni e basso consumo.

