



UNIVERSITÀ DEGLI STUDI DI PARMA

Corso di Laurea in  
"Ingegneria Informatica, Elettronica e delle Telecomunicazioni"

## **Architettura dei Calcolatori Elettronici**

**Architetture avanzate**

**Andrea Prati**

## Prestazioni dei calcolatori

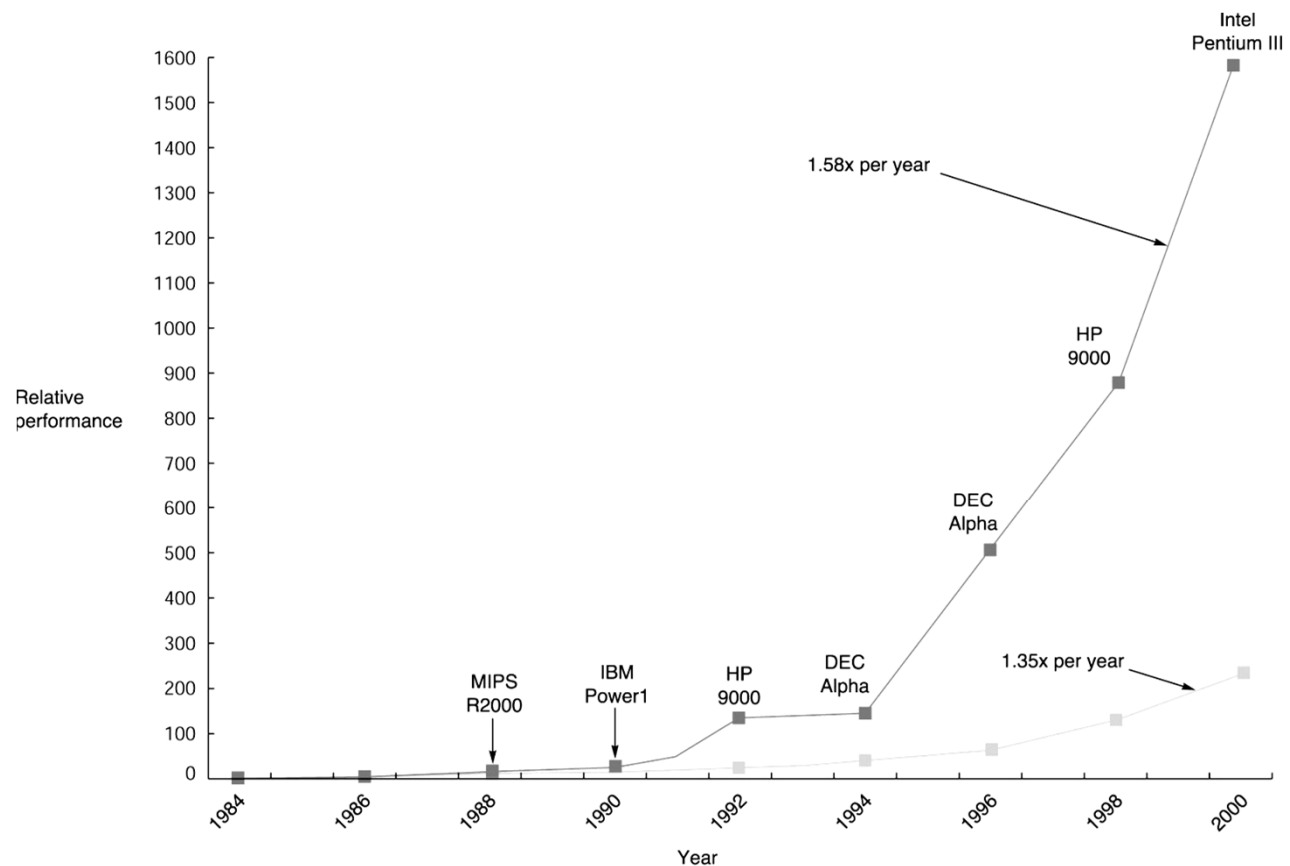
### ◆ Come calcolare le prestazioni?

In Termini di:

- velocità
- costo
- consumo

### ◆ Misura delle prestazioni:

- simulazione
- modelli analitici
- **Benchmark**



© 2003 Elsevier Science (USA). All rights reserved.

## Benchmark SPEC2000

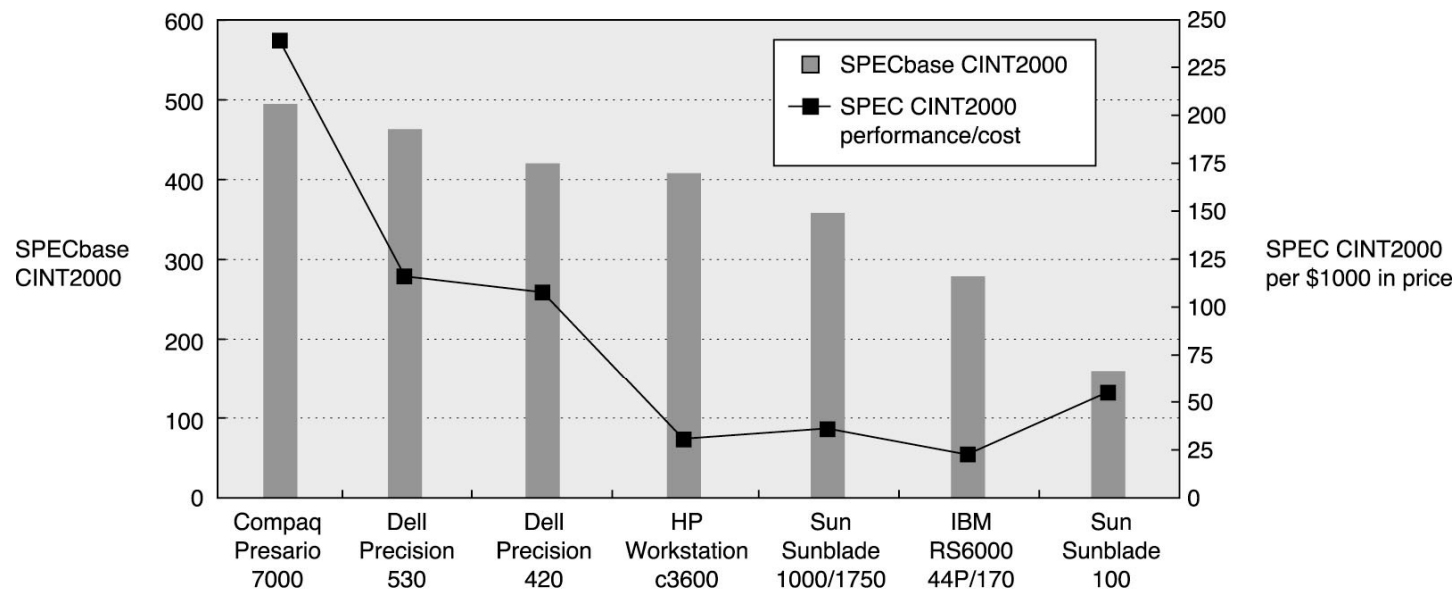
Table 2. CPU2000 integer and floating-point benchmark suite.

Benchmark	Language	KLOC	Resident size (Mbytes)	Virtual size (Mbytes)	Description
<b>SPECint2000</b>					
164.gzip	C	7.6	181	200	Compression
175.vpr	C	13.6	50	55.2	FPGA circuit placement and routing
176.gcc	C	193.0	155	158	C programming language compiler
181.mcf	C	1.9	190	192	Combinatorial optimization
186.crafty	C	20.7	2.1	4.2	Game playing: Chess
197.parser	C	10.3	37	62.5	Word processing
252.eon	C++	34.2	0.7	3.3	Computer visualization
253.perlbmk	C	79.2	146	159	Perl programming language
254.gap	C	62.5	193	196	Group theory, interpreter
255.vortex	C	54.3	72	81	Object-oriented database
256.bzip2	C	3.9	185	200	Compression
300.twolf	C	19.2	1.9	4.1	Place and route simulator
<b>SPECfp2000</b>					
168.wupwise	F77	1.8	176	177	Physics: Quantum chromodynamics
171.swim	F77	0.4	191	192	Shallow water modeling
172.mgrid	F77	0.5	56	56.7	Multigrid solver: 3D potential field
173.applu	F77	7.9	181	191	Partial differential equations
177.mesa	C	81.8	9.5	24.7	3D graphics library
178.galgel	F90	14.1	63	155	Computational fluid dynamics
179.art	C	1.2	3.7	5.9	Image recognition/neural networks
183.quake	C	1.2	49	51.1	Seismic wave propagation simulation
187.facerec	F90	2.4	16	18.5	Image processing: Face recognition
188.ammp	C	12.9	26	30	Computational chemistry
189.lucas	F90	2.8	142	143	Number theory/primality testing
191.fma3d	F90	59.8	103	105	Finite-element crash simulation
200.sixtrack	F77	47.1	26	59.8	Nuclear physics accelerator design
301.apsi	F77	6.4	191	192	Meteorology: Pollutant distribution

## Benchmark SPEC2000

### ◆ Da Hennessy Patterson

[http://www.mkp.com/books\\_catalog/catalog.asp?ISBN=1-55860-428-6](http://www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-428-6)



© 2003 Elsevier Science (USA). All rights reserved.

### ◆ [www.specbench.org](http://www.specbench.org)

Standard Performance Evaluation Corporation (SPEC) Homepage

## Tempo di esecuzione

- ◆ La CPU è una macchina sequenziale sincrona che esegue microoperazioni
- ◆ Normalmente necessita di almeno un ciclo di clock per eseguire ogni singola microoperazione.
- ◆ Alcune microoperazioni sono più lunghe delle altre. Ad es., la lettura e scrittura di memoria può impiegare più cicli di clock dipendentemente dal tipo di bus e dalla velocità delle memorie.
- ◆ Es. Si supponga di avere registri, bus e memoria a 32 bit. Ogni operazione in memoria richiede un ciclo di bus di 5 Tck (5 periodi di clock).  
Quanti clock sono necessari in assenza di pipeline per eseguire l'istruzione *add r1, alfa*?

Per calcolare il ciclo di istruzione bisogna conteggiare il tempo per prelevare l'istruzione (fetch), per decodificarla ed eseguirla

Fetch     $1+5+1=7$  Tck        per leggere l'istruzione dalla memoria

Decode   1 Tck                per decodificarla

Execute  $1+5+1+1+1+1= 10$  Tck        per eseguirla accedendo alla memoria

- ◆ La CPU deve fare due accessi in memoria → 18 TCK questo è il tempo di esecuzione dell'istruzione

## Prestazioni di CPU

- ◆ Misurando il tempo trascorso (*elapsed time*) bisogna considerare, il tempo di sistema (del s.o.) e il tempo di utente; all'interno di questo il Tempo di CPU è il tempo in cui è effettivamente impegnata la CPU nell'esecuzione del task
- ◆ Il modello più semplice utilizza il concetto di CPI:  
Ncc = numero di cicli di clock, NI = numero di istruzioni in un programma

$$T_{cpu} = Ncc \ Tck$$

- ◆ *Clock per instruction (medio)*  $\Longrightarrow$   **$CPI = Ncc / NI$**
- ◆ *per calcolare il CPI medio bisogna conoscere il  $CPI_i$  dell'istruzione  $i$ -esima e l'occorrenza media (in percentuale) dell'istruzione  $i$ -esima in un programma ( $FI_i$ )*

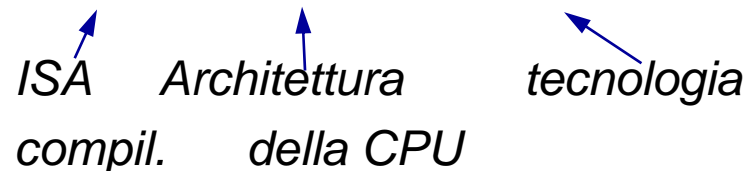
$$CPI = \sum_i FI_i CPI_i \Longrightarrow CPI = \sum_i \frac{NI_i}{NI} CPI_i$$

## Prestazioni di CPU

- ◆ L'ISA definisce le istruzioni eseguibili, mentre il CPI per ogni istruzione dipende dalla microarchitettura e dalla architettura di tutto il calcolatore (ad esempio per le operazioni di memoria)
- ◆ A posteriori si può valutare il CPI medio. Molto spesso il CPI è molto variabile (es. istruzioni tra registri o istruzioni con uso di memoria)  
E' molto difficile confrontare calcolatori diversi perchè il Tcpu per un dato programma si calcola in secondi a programma come

$$T_{cpu} = NI \cdot CPI \cdot T_{ck} = \text{Sec}/\text{Prog}$$

$$T_{cpu} = (NI/\text{Prog}) (N_{cc}/NI) (\text{Sec}/\text{Ciclo})$$



**Ciclo di clock dipende dalla tecnologia e dalla microarchitettura**

**CPI dipende dalla architettura (ISA sia microarchitettura)**

**NI dipende da ISA e dai compilatori**

## MIPS

*Prestazioni si misurano spesso (anche se in modo non affidabile) con MIPS (Mega Instruction Per Second): possono essere confrontati solo computer con la stessa ISA*

*MFLOPS (Mega FLoating point Operation Per Second)*

- ◆  $MIPS = NI / (CPU_{time} * (10^6))$   
inoltre:
- ◆  $MIPS = f_{ck} / CPI_{medio}$  ( $f_{ck}$  in MHz)
- ◆ dunque il numero di MIPS dipende da  $CPI_{medio}$  e quindi dal benchmark
- ◆ per di più il numero di MIPS non dipende da  $NI$  pertanto a parità di benchmark e di MIPS otteniamo valori diversi di  $CPU_{time}$  se  $NI$  cambia
- ◆ **Quindi il numero di MIPS, da solo, non è un buon indicatore delle prestazioni di una CPU**
- ◆ **Si possono confrontare tra loro due CPU rispetto a un determinato benchmark solamente se hanno lo stesso set di istruzioni**



## Migliorare le prestazioni

- ◆ Lo scopo di chi progetta la microarchitettura della CPU è di diminuire il  $T_{cpu}$ , rendendo più veloce possibile l'esecuzione delle istruzioni. Esistono diverse vie:

1) **ridurre il CPI di ogni tipo di istruzione.** Il ciclo di istruzione richiede più microoperazioni, che a loro volta richiedono uno o più cicli di clock. Di solito ogni passo elementare richiede un ciclo di clock (l'accesso alla memoria può richiederne molti di più)

2) **semplificare il sistema (e l'ISA)** in modo da rendere semplice ogni microoperazione e quindi diminuire il  $T_{ck}$  (ad es., la fase di decodifica dipende da quanti codici operativi diversi sono previsti)

3) **sovrapporre l'esecuzione di più istruzioni:** concetto di pipeline più istruzioni di seguito sono in fase di esecuzione, ma contemporaneamente sono in fasi diverse (es: mentre una istruzione è in fase di fetch, quella precedente è in fase di decodifica)

4) **parallelizzare l'esecuzione di microoperazioni** o di istruzioni duplicando (o moltiplicando) l'hardware (ad es: 2 ALU una per incrementare il PC ed una per eseguire l'operazione tra gli operandi, oppure inserire più ALU). Questa via è nota come Instruction level parallelism (ILP)

# ARCHITETTURE PIPELINE

## Presupposti

- L'architettura a pipeline è la più significativa soluzione architeturale che consente di aumentare la velocità di una CPU.
- Riprende concetti ampiamente applicati in altri settori. Ad esempio catena di montaggio industriale.
- La catena di montaggio consente di aumentare la frequenza con la quale vengono ultimati i prodotti (**Throughput**)
- Mantiene inalterato il tempo necessario a realizzare il prodotto (**Latenza**)



## Idee di base

- L'esecuzione di una istruzione consiste di più fasi.
- Le fasi sono svolte in sequenza da sezioni diverse.
- Si aumenta il throughput senza aumentare la velocità dell'unità centrale.

## Primi esempi

- Le prime architetture pipeline vennero introdotte dall'IBM:
  - IBM 3033 (1977). Pipeline a due stadi.
    - Istruzione e Calcolo
  - IBM 801 (1975-80). La prima architettura RISC con unità di esecuzione pipelined.



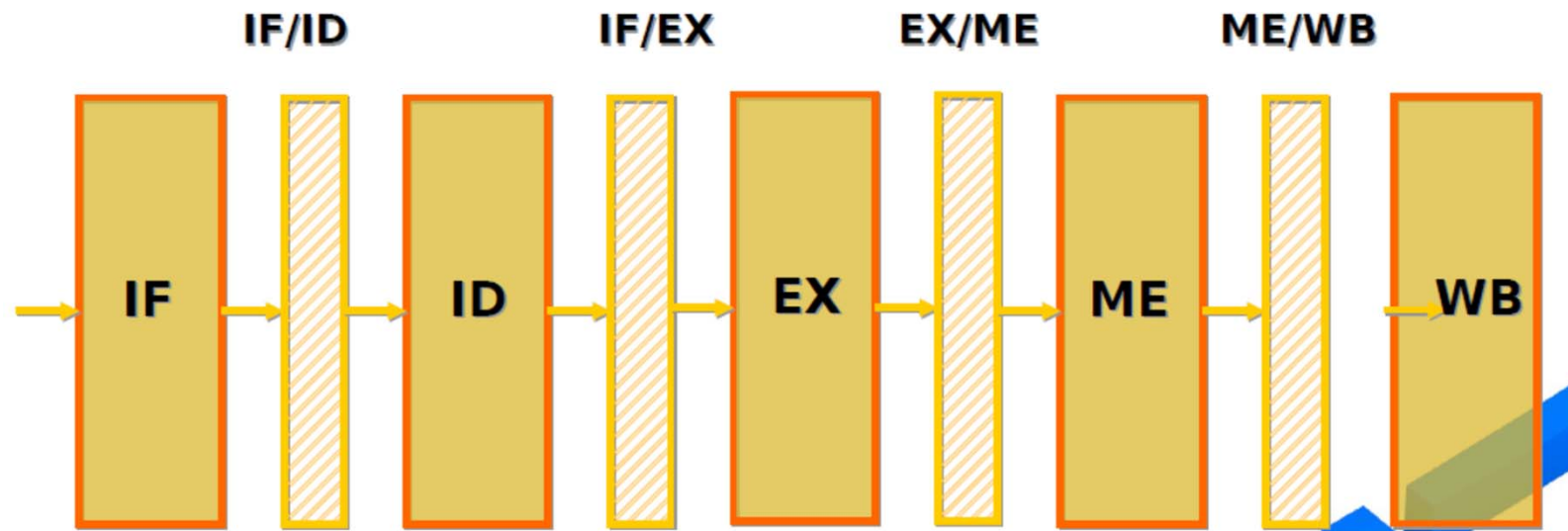
IBM 3033

## Modello

- 5 stadi (già visti)
  - IF: fetch o prelievo istruzione
  - ID: decodifica e lettura registri
  - EX: esecuzione
  - ME: accesso alla memoria
  - WB: scrittura registro destinazione

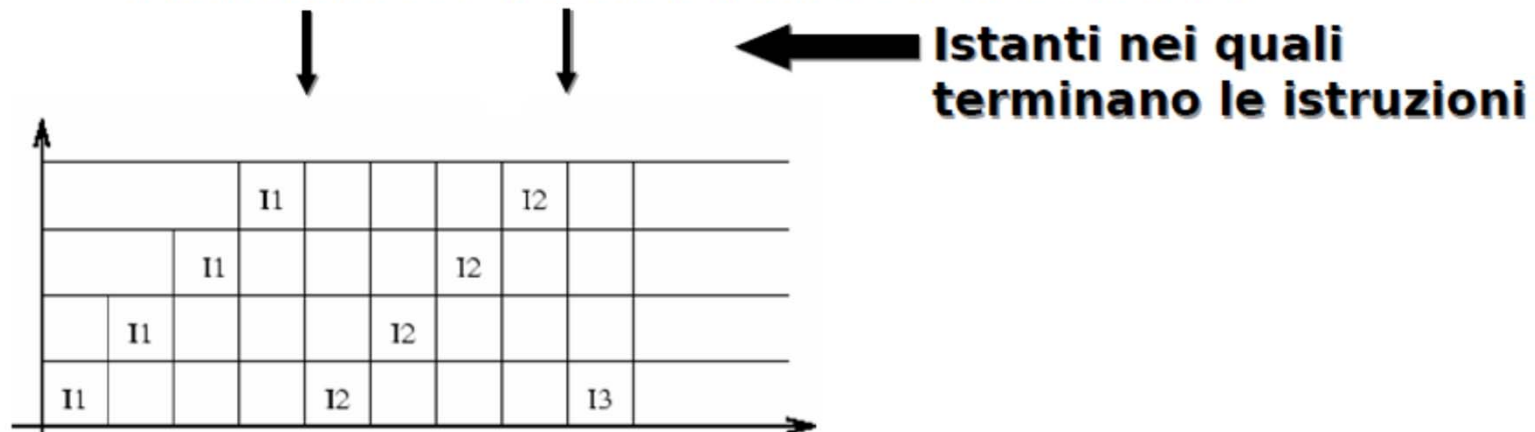
## Architettura pipeline: schema di massima

- Necessità di „disaccoppiare“ gli stadi
- Registri di *latch* che propagano i segnali



## Architettura pipeline: confronto

### ARCHITETTURA TRADIZIONALE



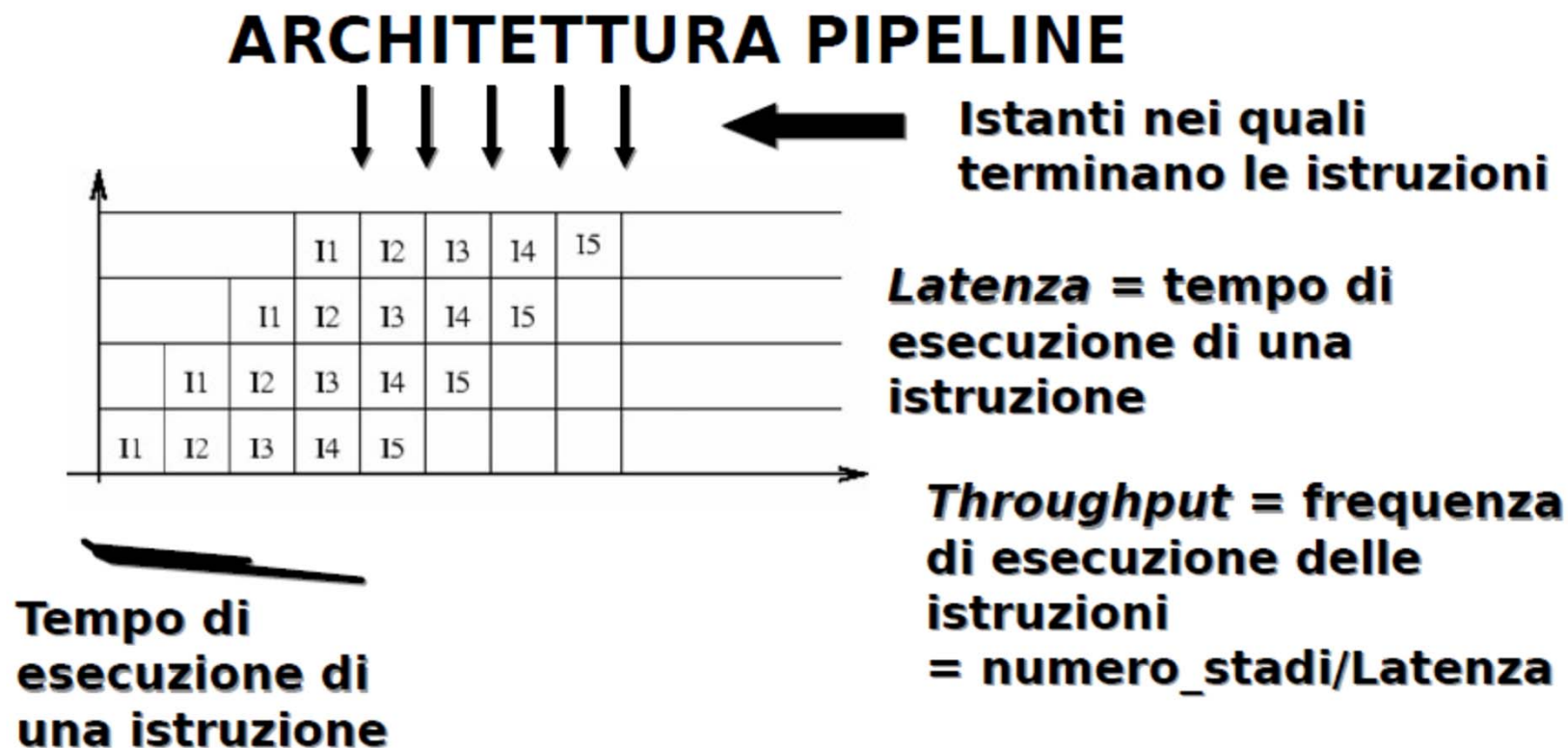
**Tempo di esecuzione di una istruzione in una architettura tradizionale**

***Latenza* = tempo di esecuzione di una istruzione**

***Throughput* = frequenza di esecuzione delle istruzioni  
= 1/Latenza**



## Architettura pipeline: confronto



A regime (pipeline piena) ogni istruzione richiede  $k$  periodi di clock ma ad ogni periodo viene completata un'istruzione

## Prestazioni

- Supponendo  $k$  stadi,  $n$  istruzioni vengono elaborate in:

$$T_k = (k + (n - 1)) \cdot \tau$$

$\tau$  = tempo clock

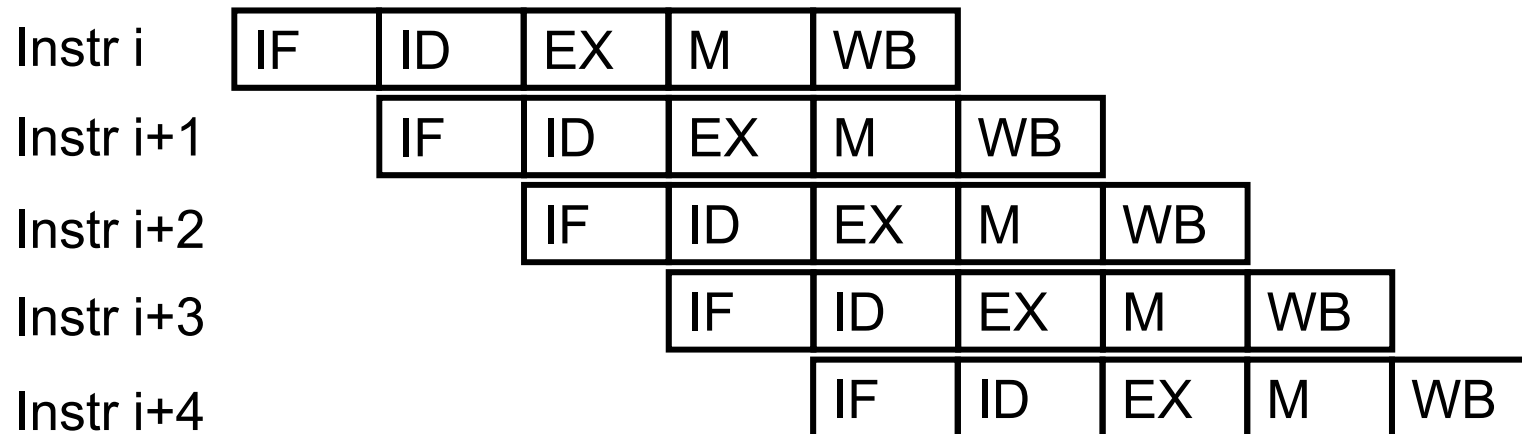
- Aumento velocità:

$$Sp = \frac{T_1}{T_k} = \frac{n \cdot k \cdot \tau}{(k + (n - 1)) \cdot \tau} = \frac{n \cdot k}{(k + n - 1)}$$

- Al crescere di  $k$ 
  - $S$  tende a  $n$
- Rispetto all'esecuzione in sequenza le prestazioni sono  $k$  volte superiori purchè gli stadi della pipeline siano sempre tutti attivi
- Non è, per ragioni differenti, possibile mantenere la pipeline sempre piena!

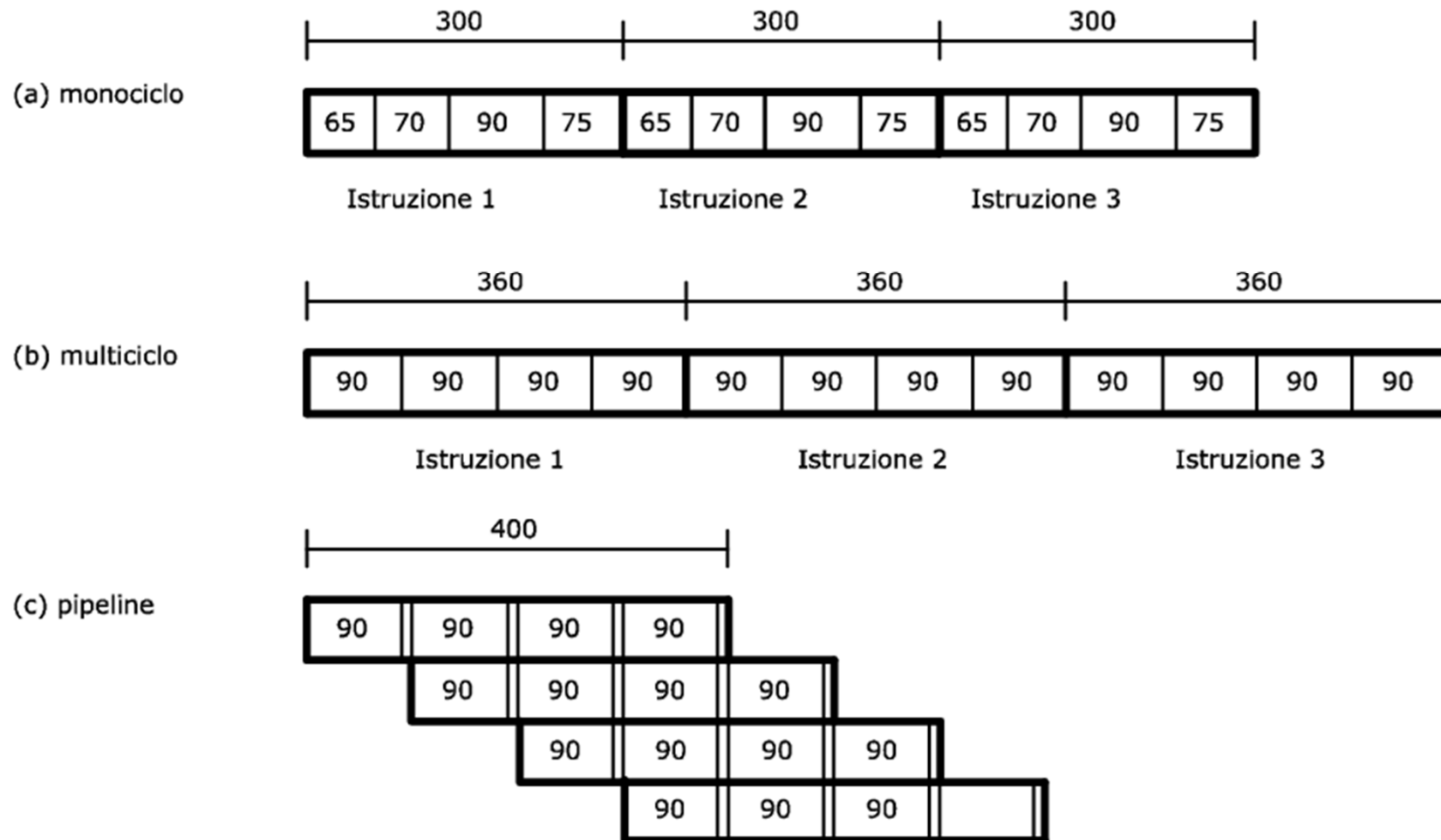
## Prestazioni

- ◆ Il problema fondamentale è come minimizzare le possibilità di stallo della pipeline e **cercare di mantenere la pipeline piena**



- ◆ Tutti i processori attuali hanno una pipeline più o meno profonda
  - Intel Pentium II 12 stadi
  - Alpha 21064 8 stadi
- Ragionevolmente  $2 < k < 15$

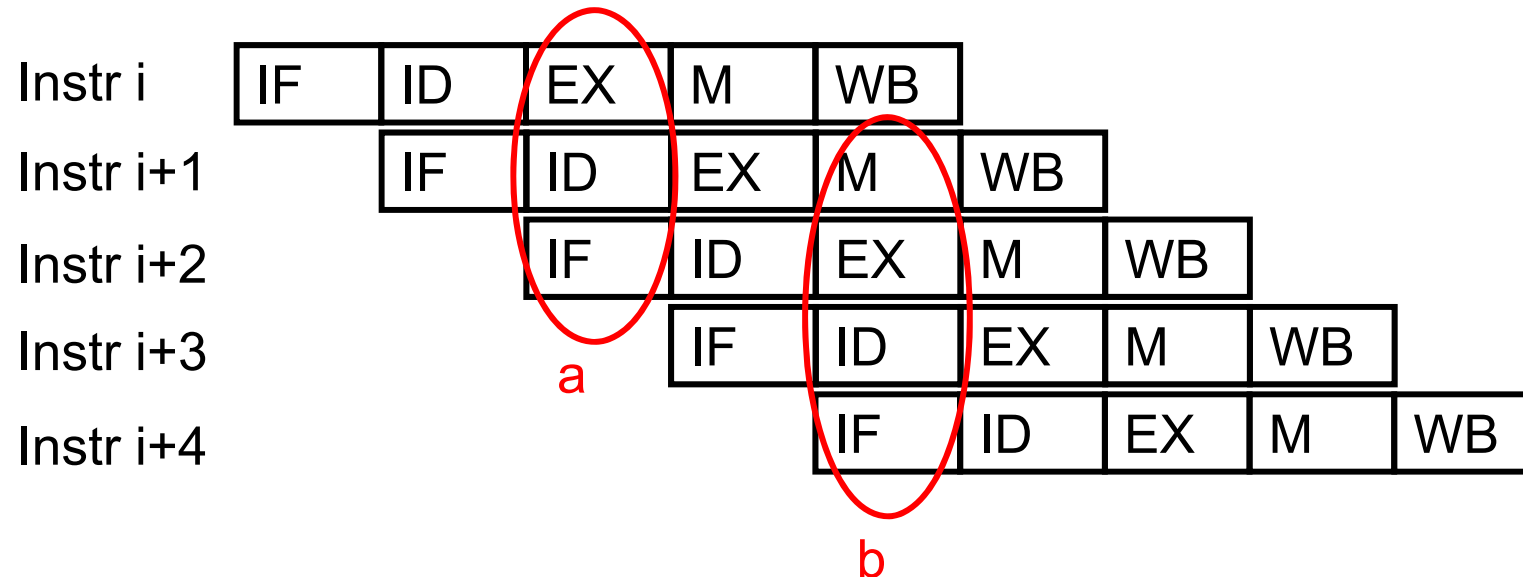
## Confronto



## Pipeline non ideale

- ◆ Verrebbe da dire che aumentare il numero degli stadi porta a migliorare le prestazioni.
- ◆ Esistono tre limiti all'aumento del numero degli stadi della pipeline (non idealità):
  - **Alee strutturali**: conflitto di risorse: a due blocchi occorre la stessa risorsa, esempio la ALU
  - **Alee di dato**: a un'istruzione serve un risultato non ancora pronto, in quanto l'istruzione che la precede nel programma è ancora in corso di esecuzione
  - **Alee di controllo**: il processore non sa dove trasferire il controllo (“saltare”) perché non è ancora stato determinato se e dove saltare

## Alee strutturali



### Alee strutturali

a - Conflitto sulla ALU fra EX (operazioni) e IF (incremento PC)

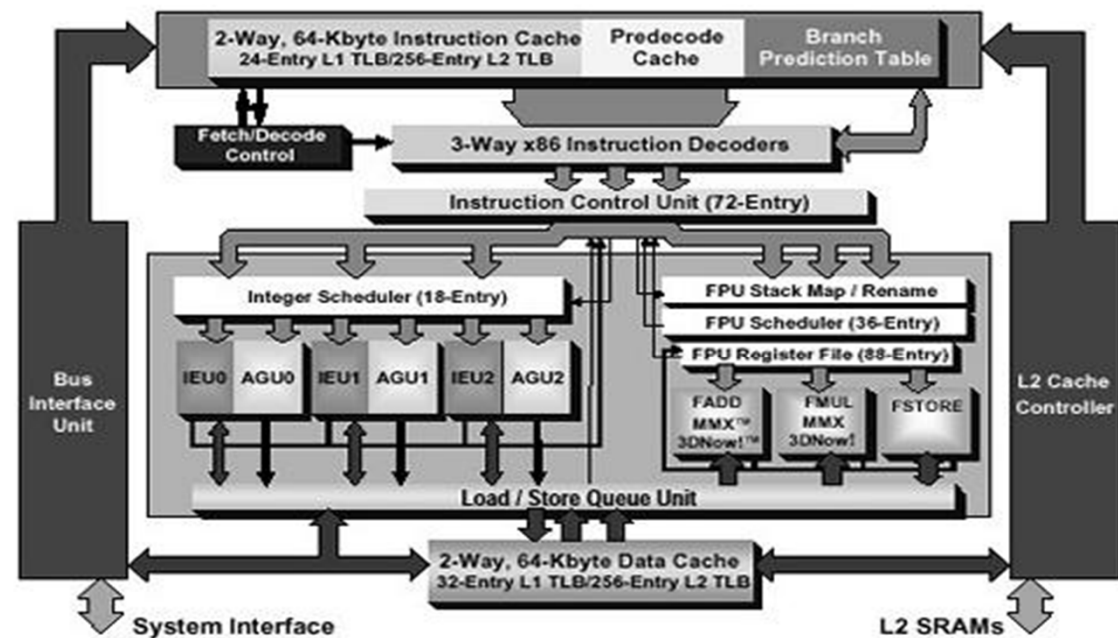
b - Conflitto per il bus fra MEM e IF

### Possibili soluzioni

- 1) Replicazione delle risorse (più comune) – es: 8086 oltre a ALU ha un sommatore per il PC per conflitto a; o arch. Harvard con separazione memoria istruzioni e memoria dati per conflitto b
- 2) Attendere, ritardando l'esecuzione dell'istruzione (da evitare, se possibile)

## Problema memoria

- Memorie separate sono scomode
  - Duplicazione bus etc.
- Soluzione
  - Gerarchia di memorie (vedi dopo)
  - Cache separate



AMD Athlon™ Processor Block Diagram

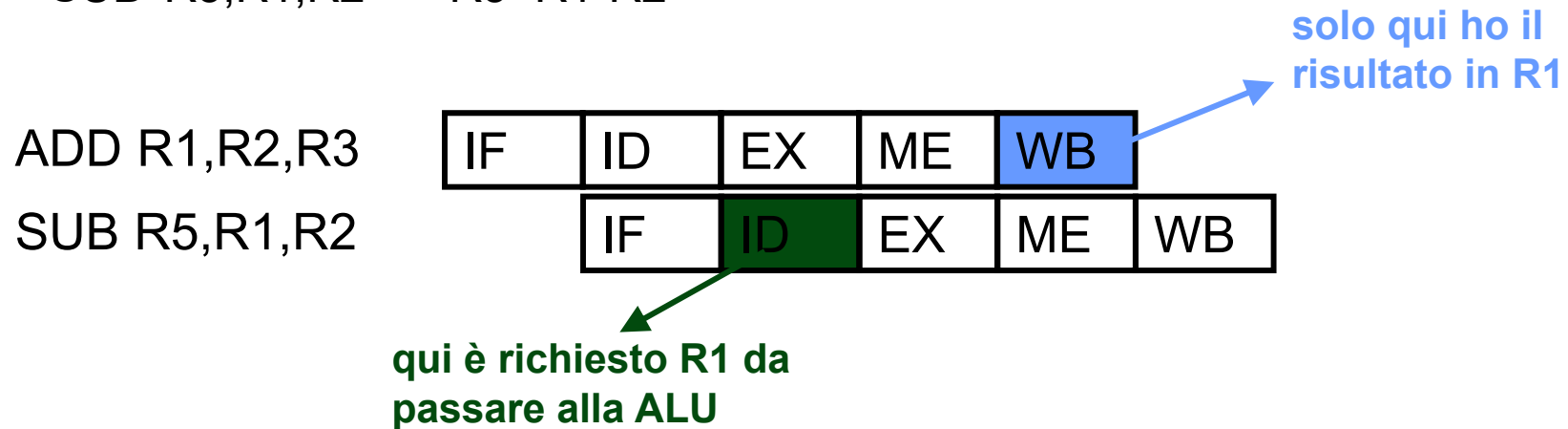
## Alee di dato

- Il conflitto dati si verifica quando vi è dipendenza dati tra istruzioni „vicine“
- Siano A e B due istruzioni con A che precede B
  - **RAW** (*read-after-write*): B legge un dato scritto da A prima che A abbia potuto scriverlo
  - **WAR** (*write-after-read*): B modifica un dato prima che sia letto da A.
  - **WAW** (*write-after-write*): B tenta di scrivere un dato prima che A lo abbia scritto



## Alee di dato: esempio RAW

- `ADD R1,R2,R3`  $\rightarrow R1=R2+R3$
- `SUB R5,R1,R2`  $\rightarrow R5=R1-R2$



- ADD modifica R1 (WB)
- SUB legge R1 (ID)

Quando SUB legge R1 questo non è ancora stato aggiornato da ADD

## Alee di dato: riconoscimento e possibili soluzioni

- Prima di provare a risolverlo va riconosciuto
  - ID vs ME, EX, WB
  - Campi specifici su registri di pipeline
- Differenti strategie:
  - **Stallo:** impedisco avanzamento istruzioni → molto penalizzante
  - **Anticipazione:** il dato prodotto viene reso immediatamente disponibile a chi lo deve utilizzare → circuiti aggiuntivi
  - **Sovrapposizione:** operazioni in conflitto vengono eseguite in successione nello stesso periodo di clock (*half clocking*)
  - **Riordinamento:** viene modificato l'ordine di esecuzione istruzioni

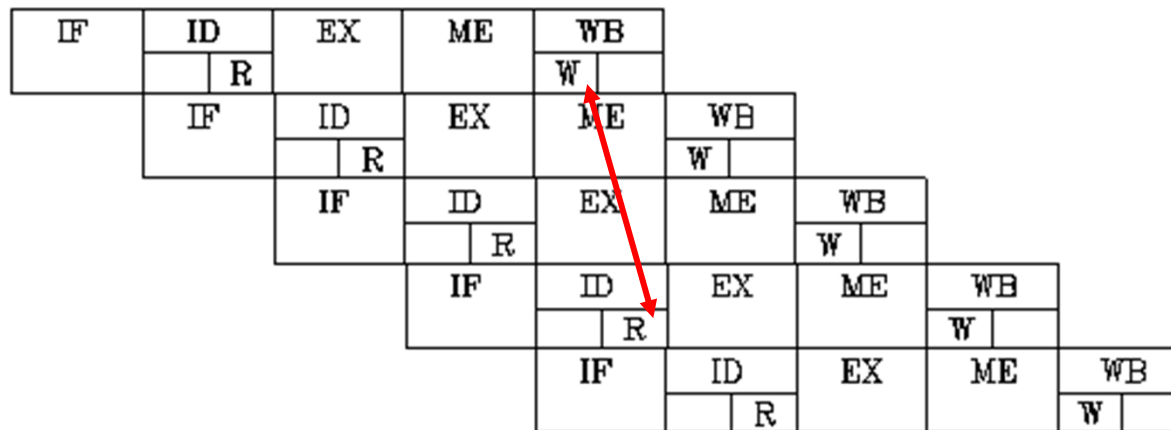
## Stallo

- È la soluzione più semplice



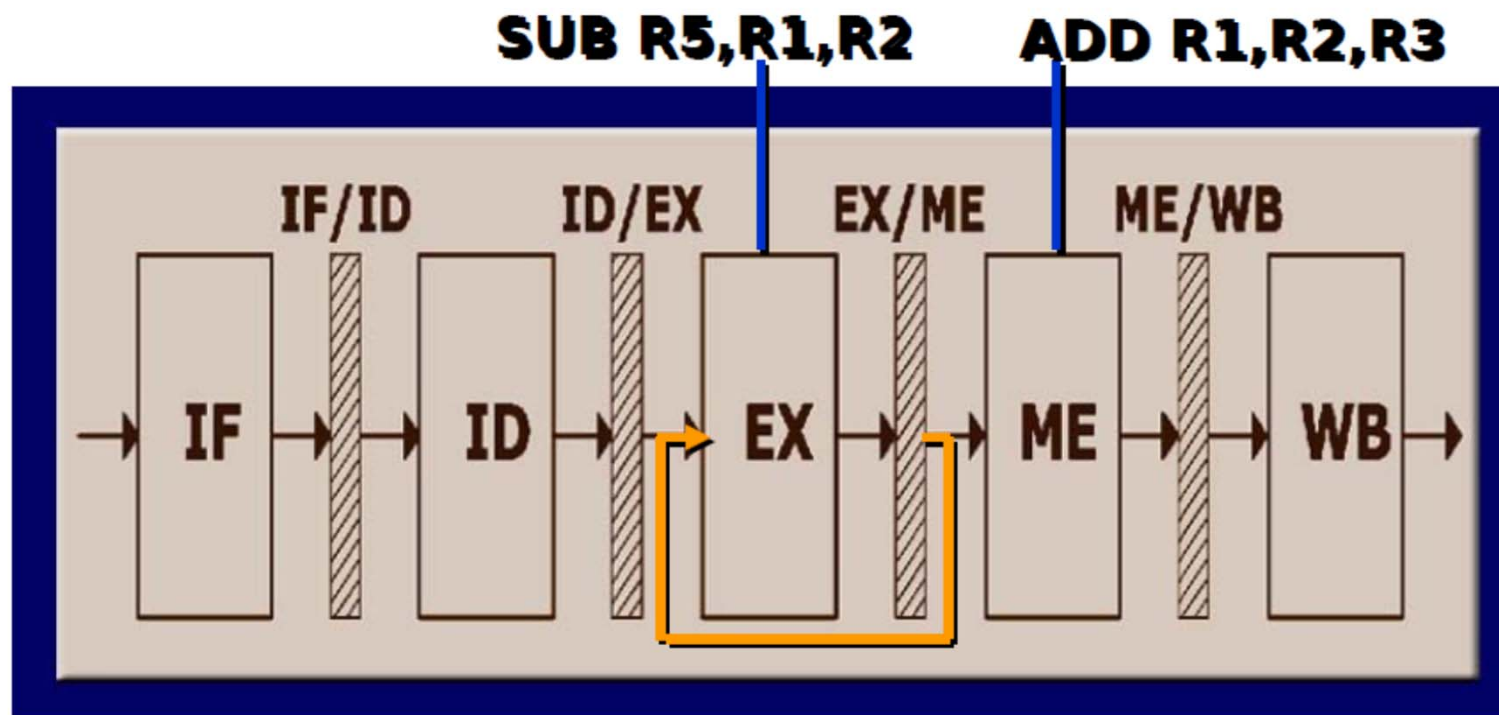
## Sovrapposizione

- Si gioca sul clock (di fatto lo si raddoppia)
  - ID legge dopo WB nello stesso clock



## Anticipazione (forwarding)

- Bypasso la pipeline prima del punto di prelievo
- Non banale (conflitti multipli)



## Riordinamento

- Tecniche precedenti non risolvono tutte le situazioni
  - LOAD/STORE
- Riordinamento eseguito a livello SW → compilatore

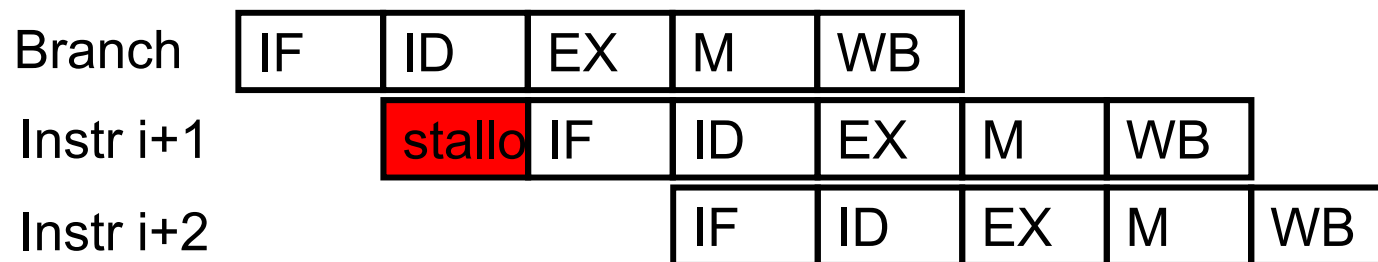
ADD	R1,R2,R3		LOAD	R7,M(D1)
SUB	R4,R5,R6		LOAD	R8,M(D2)
LOAD	R7,M(D1)		ADD	R1,R2,R3
LOAD	R8,M(D2)		SUB	R4,R5,R6
MUL	R9,R7,R8		MUL	R9,R7,R8

PRIMA

DOPO

## Alee di controllo

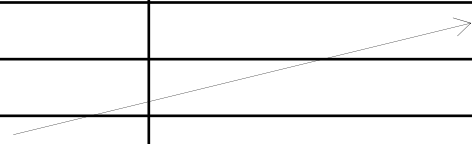
- Salti incondizionati
  - Statisticamente rappresentano il 2-8% delle istruzioni
- PC viene modificato in fase EX
  - Gli stadi precedenti vanno svuotati
- Un salto introduce uno stallo (1 clock sprecato) nella pipeline perché l'indirizzo a cui saltare è noto solo dopo la fase di decode



## Salti incondizionati: soluzione SW

- Riordinare le istruzioni permette di evitare di svuotare la pipeline
  - Si anticipa il salto di modo che la EX del salto preceda la IF opportuna

SUB	R1,R3,R9		SUB	R1,R3,R9
ADD	R2,R3,R8		JMP	ALTROVE
MUL	R5,R6,R1		ADD	R2,R3,R8
JMP	ALTROVE		MUL	R5,R6,R1

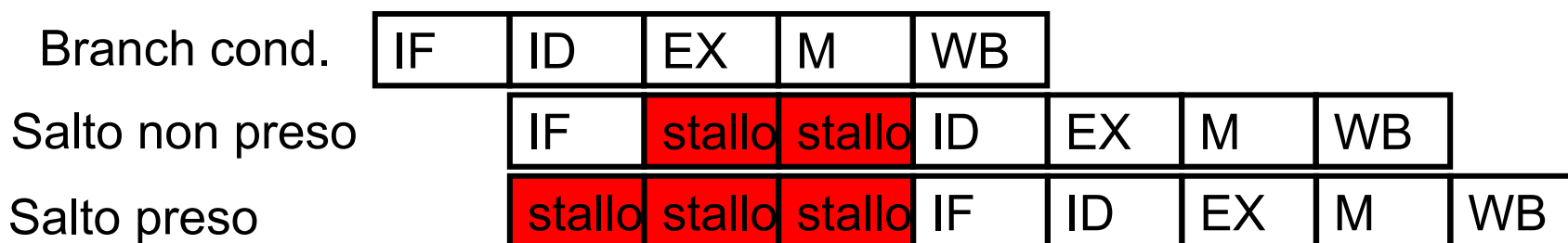


Non sempre possibile!



## Alee di controllo

- ◆ Addirittura, un salto condizionale introduce 3 stalli nella pipeline perché la condizione di salto è valutata solo dopo la fase di MEM



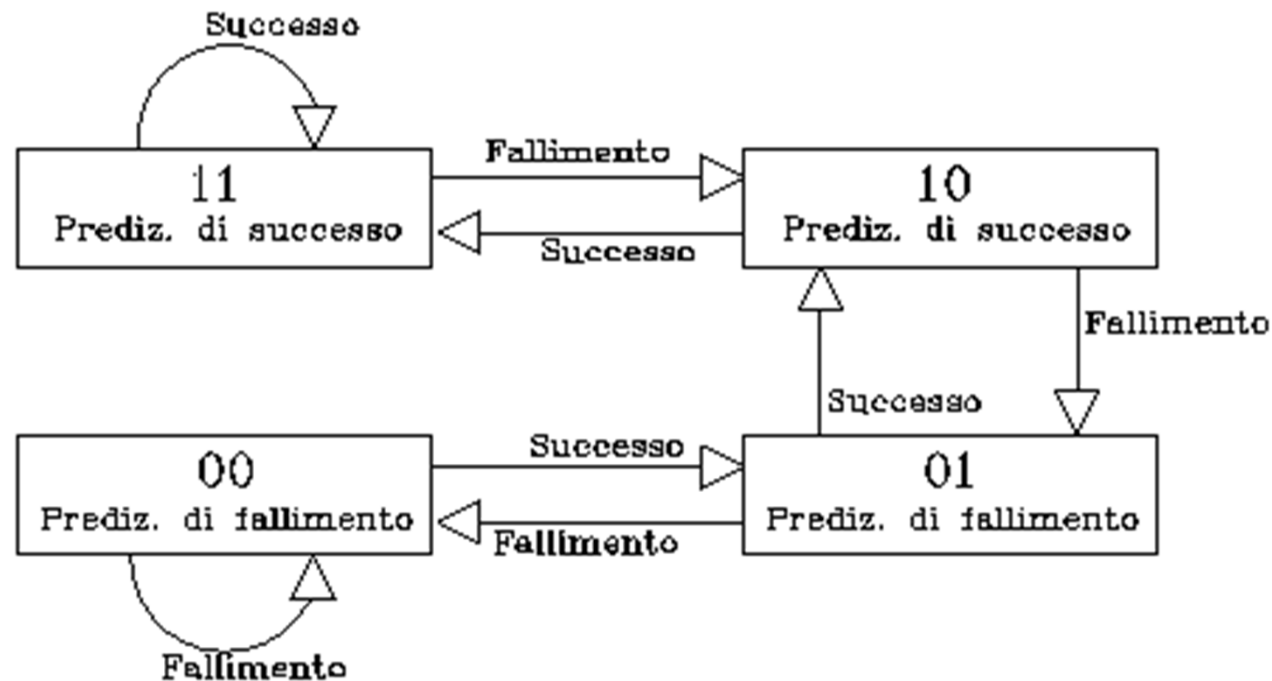
- Non risolvibile (completamente) via SW
  - Istruzioni che non modificano condizione
  - Unrolling cicli
  - Tecniche di predizione

## **Alee di controllo: tecniche di predizione**

- Predizione statica
  - Assumo che il salto avvenga sempre o mai
  - Assunzione anche basata sul tipo di istruzione
- Predizione dinamica
  - Predizione basata su storia precedente

## Alee di controllo: predizione dinamica

- Approccio semplificato
  - Tengo conto dell'ultimo comportamento determinato
  - 2 bit che memorizzano risultato



## Alee di controllo: predizione dinamica

- Non basta predizione
- Devo anche poterla usare subito
- Tabella di predizione (associativa)
  - *Branch Prediction Buffer*
  - Due colonne
    - Risultato predittore
    - Indirizzo (PC) salto condizionato

## Esempio: Intel 80486

- Pipeline a 5 stadi (non coincidono con quelli visti):
  - IF
  - ID1
  - ID2
  - EX
  - WB

## 80486: IF

- Prelevo istruzioni da cache o memoria
- Lunghezza istruzioni variabile (1-11 byte e oltre)
- 2 buffer di prefetch da 16 byte
  - Mediamente carico 5 istruzioni
- Funzionamento del tutto indipendente da stadi successivi
  - Ottimizzo riempimento buffer

## 80486: ID1

- Da IF riceve primi 3 byte istruzione
- Decodifico
  - Opcode
  - Modo indirizzamento

## 80486: ID2

- In base a risultato ID1 riceve da IF altri dati (opzionale)
  - Dati immediati o
  - Spiazzamento / offset
- Espando segnali controllo per ALU
- Gestione indirizzamento



## 80486: EX

- Fase esecutiva
  - Operazioni ALU
  - Genero segnali controllo per cache e/o registri

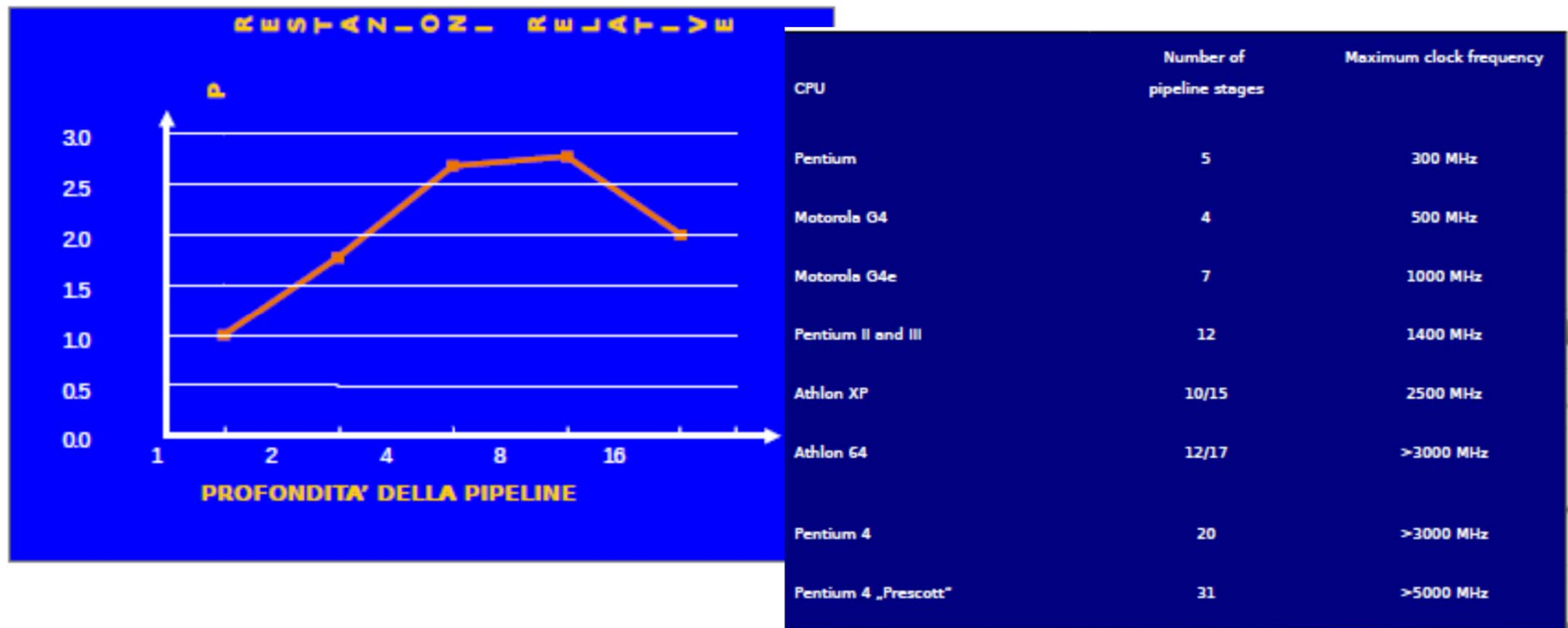
## 80486: WB

- Aggiorno registri (e flag)
- Risultati su buffer cache e/o I/O
- ME

# ARCHITETTURE SUPERSCALARI

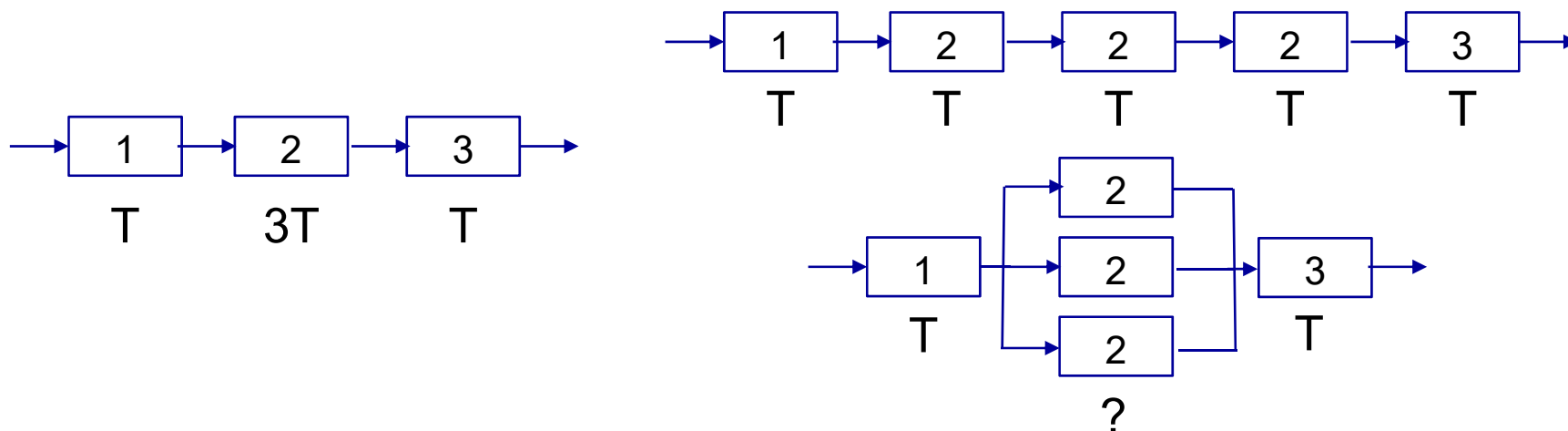
## Oltre la pipeline

- Modello visto sinora è chiamato **pipeline lineare**
- Non viene più utilizzato
- Speed up teorico  $\neq$  speed up reale



## Limiti della Pipeline Lineare

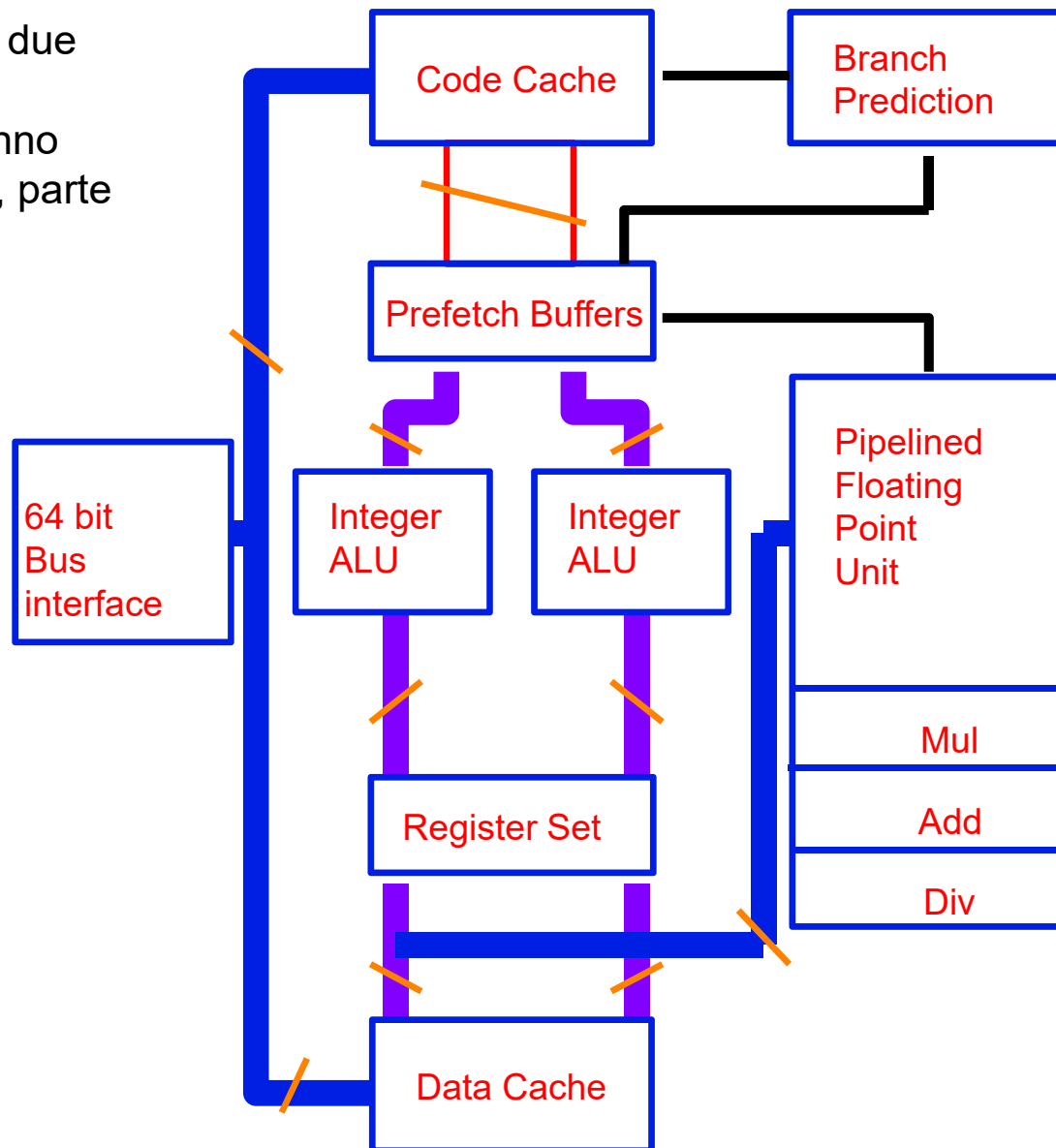
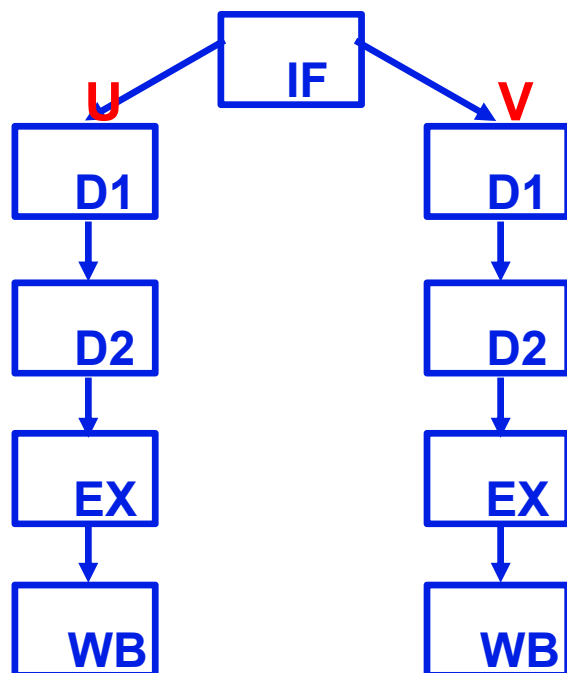
- Tutte le istruzioni attraversano tutti gli stadi
- Il periodo di clock è determinato dallo stadio più lento
  - Istruzioni diverse → tempi differenti
  - Esempio: istruzioni a virgola mobile
- Soluzione: parallelismo
  - Replico alcuni stadi della pipeline
  - Rammentiamo modello: IF, ID, EX, ME, WB



## Superscalarità del Pentium

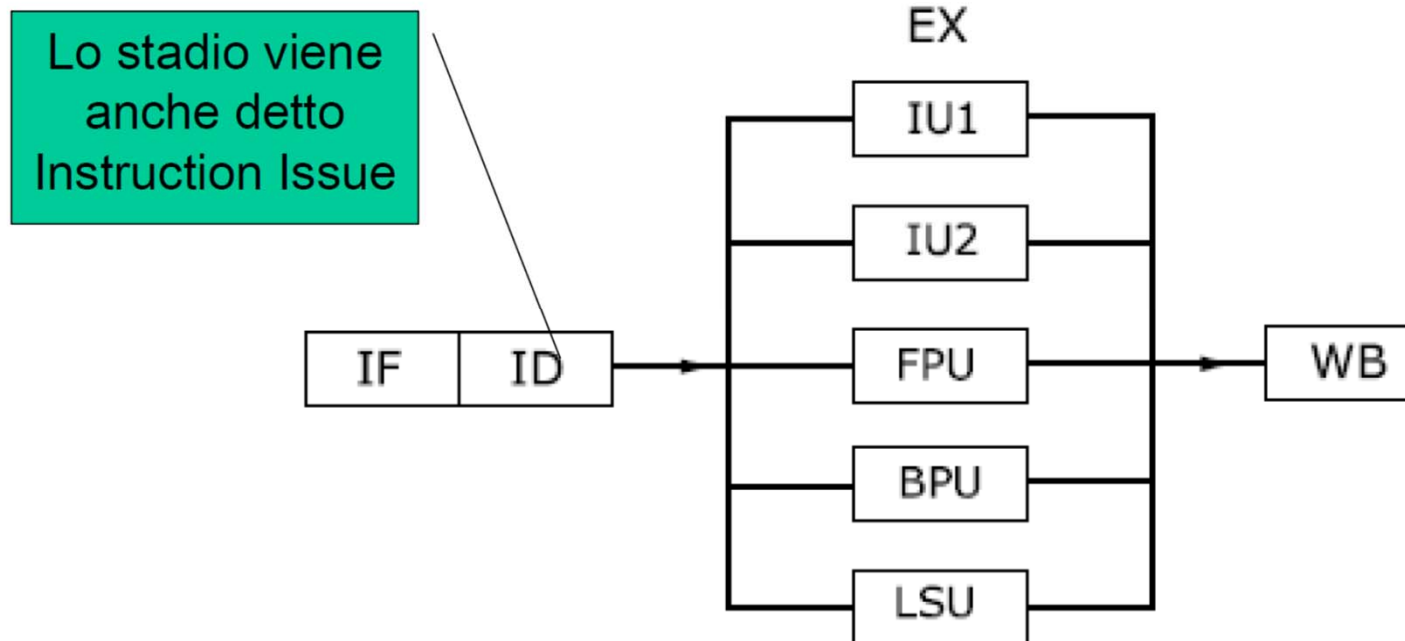
Superscalare a 2 vie: se possibile due istruzioni sono eseguite contemporaneamente (se non hanno riferimenti incrociati). Ovviamente, parte dell'hardware è duplicato

### PIPELINE DEL PENTIUM



## Parallelismo

- Tipicamente si replicano le unità funzionali



- IU1 = ALU principale in aritmetica intera (1 ciclo di clock)
- IU2 = ALU in aritmetica intera (2)
- FPU = ALU in aritmetica in virgola mobile (4)
- BPU = branch prediction unit – predizione dei salti (2)
- LSU = unità di load/store (2)

## Ipotesi sui CPI delle unità

- Riassumendo:
  - **Classe:** add/sub      **UF:** UI1      **CPI:** 1
  - **Classe:** mul      **UF:** UI2      **CPI:** 2
  - **Classe:** addf/mulf      **UF:** FPU      **CPI:** 4
- Ipotizzeremo che venga **emessa** verso lo stadio EX una istruzione alla volta:
  - Potrà **al massimo** essere ritirata una istruzione alla volta
  - Per ora di superscalare c'è solo il parallelismo tra le UF; non c'è l'esecuzione di più di un'istruzione alla volta!!!

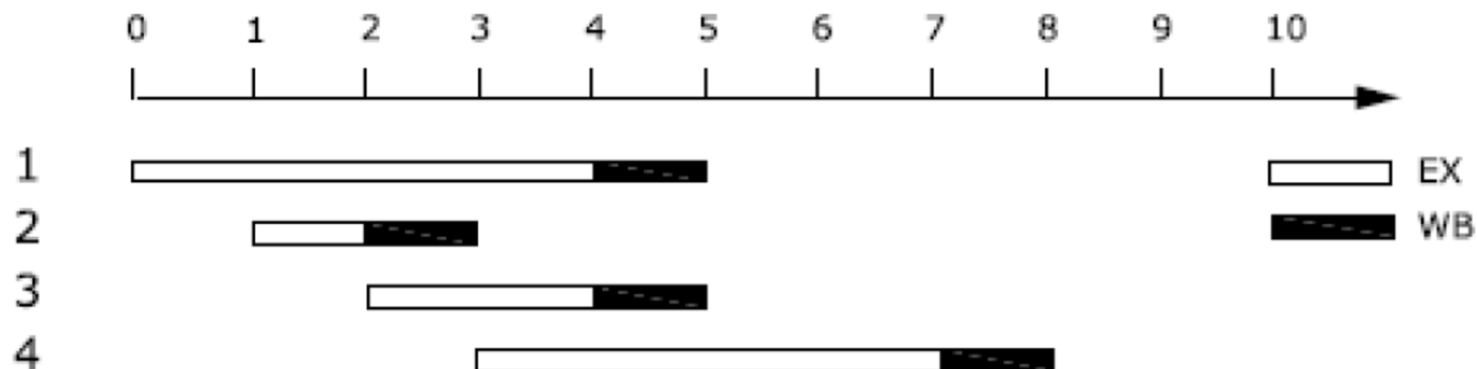


## Emissione in ordine

- Determina il completamento fuori ordine. Esempio:

1	(100) <b>mulf</b>	<b>f1,f7,f8</b>	4 cicli in FPU
2	(104) <b>add</b>	<b>r2,r4,r5</b>	1 ciclo in IU1
3	(108) <b>mul</b>	<b>r3,r10,r11</b>	2 cicli in IU2
4	(112) <b>addf</b>	<b>f4,f6,f6</b>	4 cicli in FPU

- ◆ Profilo di esecuzione



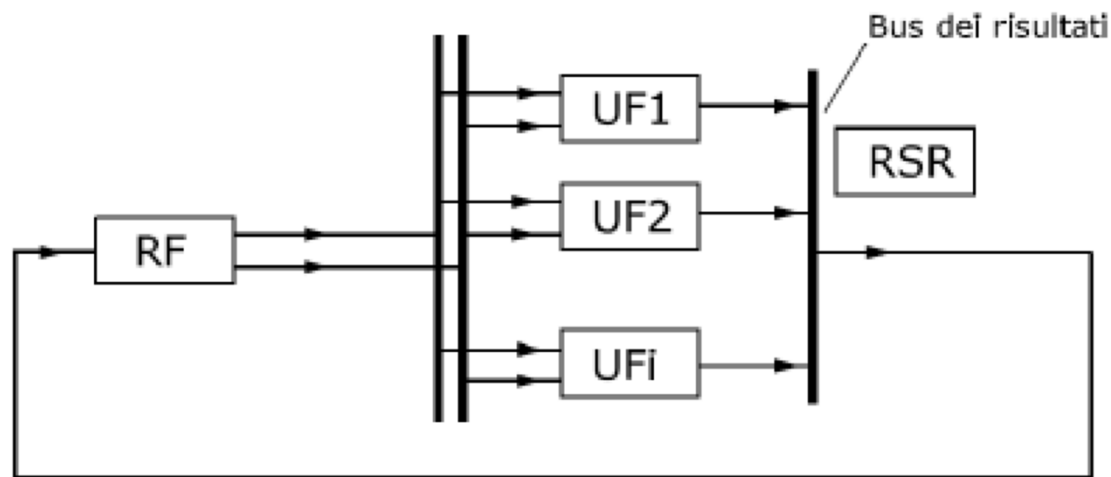
## Emissione in ordine

- In generale, date due istruzioni  $i$  e  $j$ , con  $i$  che precede  $j$ , se  $i$  impiega più cicli di clock di  $j$  per essere eseguita, l'ordine di completamento può essere **invertito**. Questo però provoca due problemi:
  - Nel caso precedente di 1 e 2, il WB avverrebbe in ordine diverso e questo pone il problema della **coerenza** della macchina!!
  - Nel caso di 1 e 3 si avrebbe una situazione di **contesa** per la fase di WB, quindi di **conflitto di risorse**

## Tre metodi

- Lo stato di macchina deve corrispondere all'ordine del programma (*coerenza*). Per mantenerla ci sono tre metodi:
  - **Completamento in ordine**: si forzano le istruzioni ad essere completate secondo l'ordine prestabilito
  - **Buffer di riordinamento**: istruzioni sono completate fuori ordine ma sono forzate a modificare lo stato della macchina in ordine, mediante l'interposizione di un **buffer** tra l'unità di emissione e quella di scrittura dei registri (o della memoria).
  - **History buffer**: le istruzioni sono aggiornate in qualsiasi ordine, ma lo stato coerente può essere **ripristinato in presenza di conflitti**

## Modello generale



- **RSR** = Reservation Shift Register
- Per evitare i conflitti sul bus dei risultati serve un **meccanismo di prenotazione del bus**, mediante un RSR, con la seguente struttura

UF	Rd	V	PC
----	----	---	----

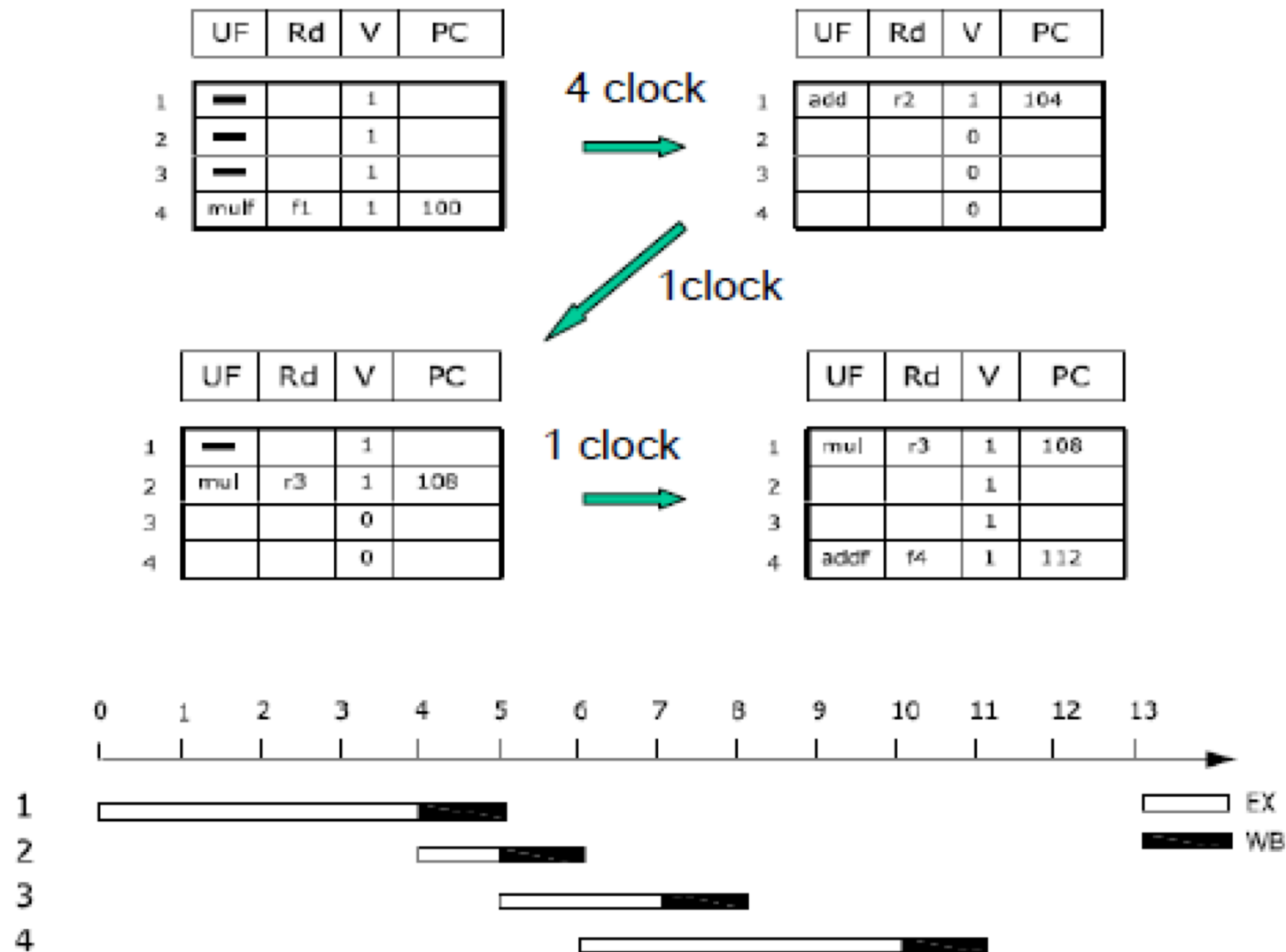
1			
2	mul	r3	1
3			
4			

## Campi di RSR

- ◆ **V** = bit di validità; dice se la posizione contiene informazioni o è vuota
- ◆ **PC** = PC dell'istruzione; necessario per ripristinare uno stato coerente in caso di predizione di salto errata
- ◆ **UF** = al completamento di una istruzione serve ad individuare da quale UF deve essere preso il risultato da trasmettere a Rd
- ◆ **Rd** = individua il registro destinazione del risultato

	UF	Rd	V	PC
1				
2	mul	r3	1	108
3				
4				

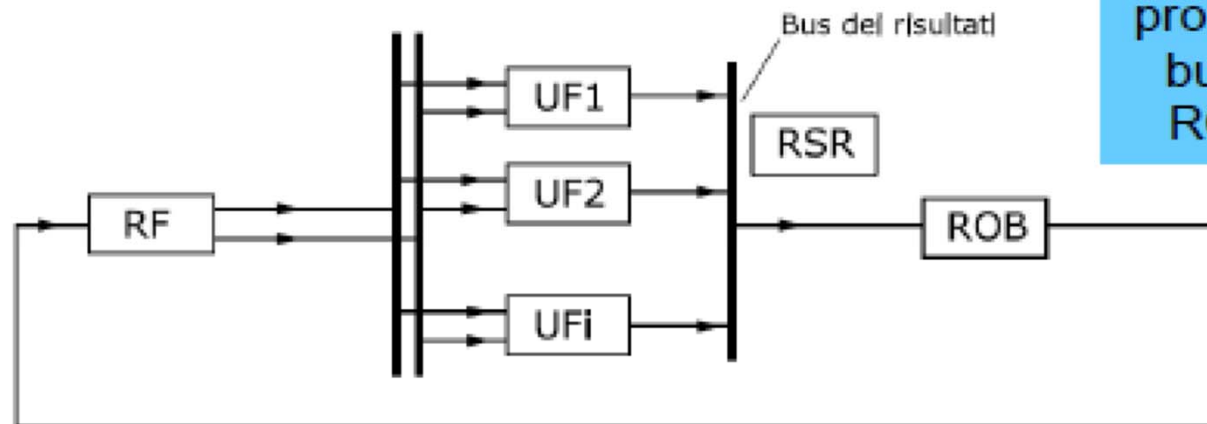
## Completamento in ordine



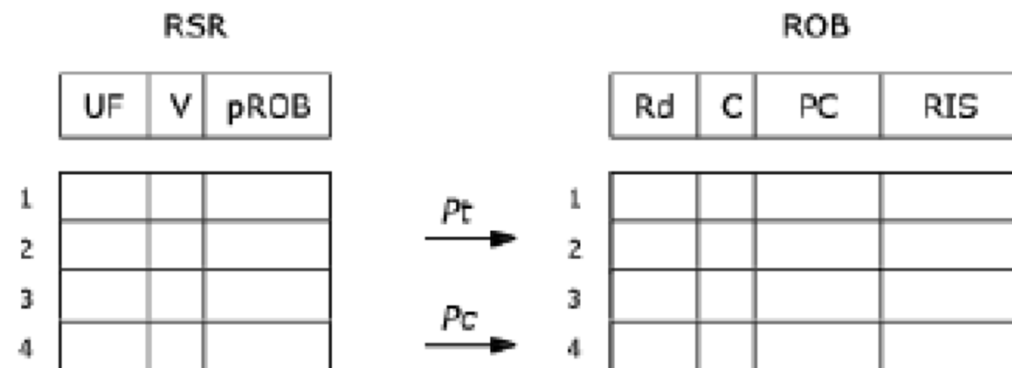
## Completamento in ordine

- ◆ **Completamento in ordine rispetto alla memoria:**
  - Due metodi (equivalenti):
    - ◆ Non viene emesso alcun comando di memorizzazione (store) prima che le istruzioni emesse precedentemente siano state completate (finchè quindi RSR non è vuoto)
    - ◆ La memoria viene considerata come un'unità funzionale, quindi store occupa una posizione in RSR in modo che questa raggiunga la cima quando tutte le istruzioni precedenti sono state completate.

## Buffer di riordinamento (ROB)



Se non ci fosse il problema dell'accesso al bus basterebbe solo il ROB a tenere l'ordine



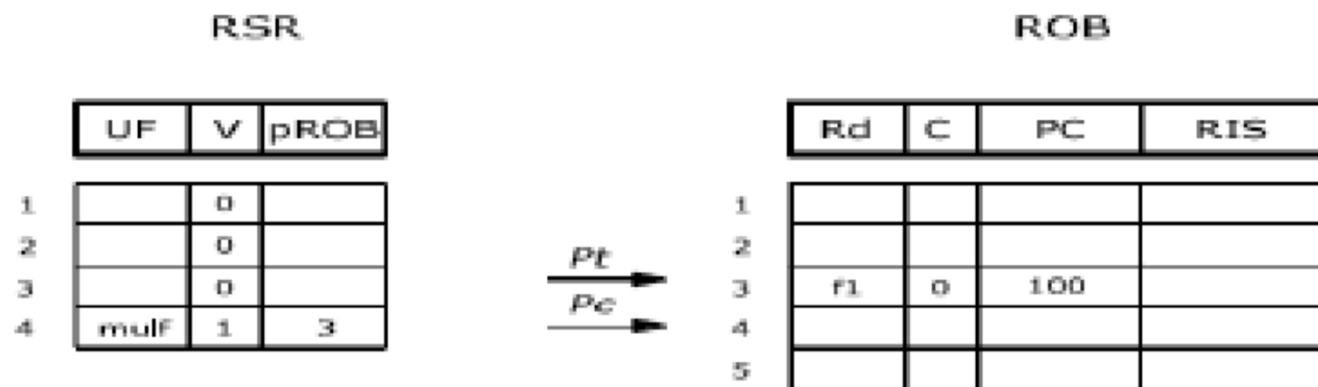


## Buffer di riordinamento (ROB)

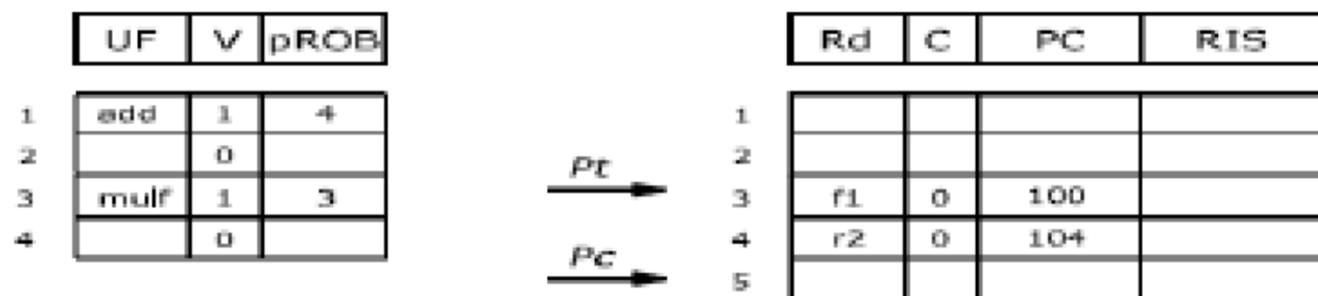
- ◆ Il ROB è un buffer interposto tra unità funzionali e registri che ha una duplice funzione:
  - Tenere traccia, ordinata come da programma, delle istruzioni che sono in corso sulle UF o che hanno concluso l'elaborazione ma non hanno ancora depositato il risultato
  - Servire da appoggio per i risultati delle istruzioni completate
- ◆ E' gestito come un **buffer circolare** con due puntatori, Pt per la testa e Pc per la coda. Un elemento è costituito da 4 componenti: **C** (bit di completamento), **PC**, **Rd**, **RIS** (risultato prodotto dall'istruzione)

## Buffer di riordinamento (ROB)

a)



b)



## Buffer di riordinamento (ROB)

c)

	UF	V	pROB
1		0	
2	mult	1	3
3		0	
4		0	

$P_t$  →

$P_C$  →

	Rd	C	PC	RIS
1				
2				
3	r1	0	100	
4	r2	1	104	2222
5				

d)

	UF	V	pROB
1	mult	1	3
2	mul	1	5
3		0	
4		0	

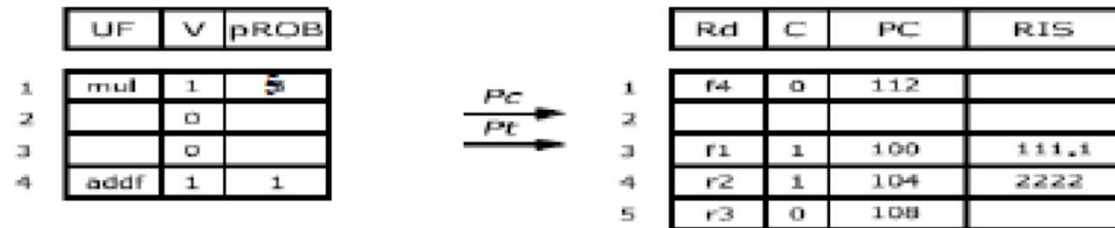
$P_C$  →

$P_t$  →

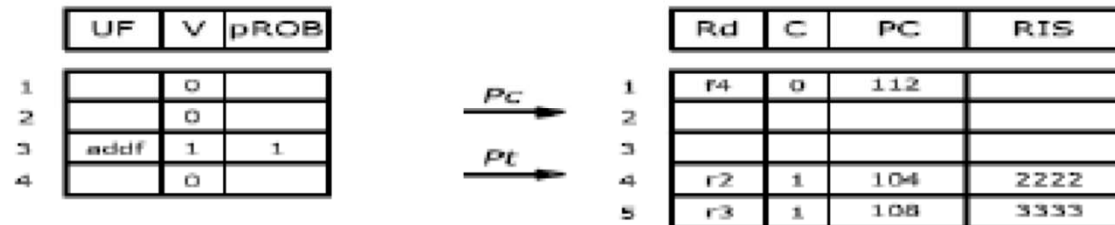
	Rd	C	PC	RIS
1				
2				
3	r1	0	100	
4	r2	1	104	2222
5	r3	0	108	

## Buffer di riordinamento (ROB)

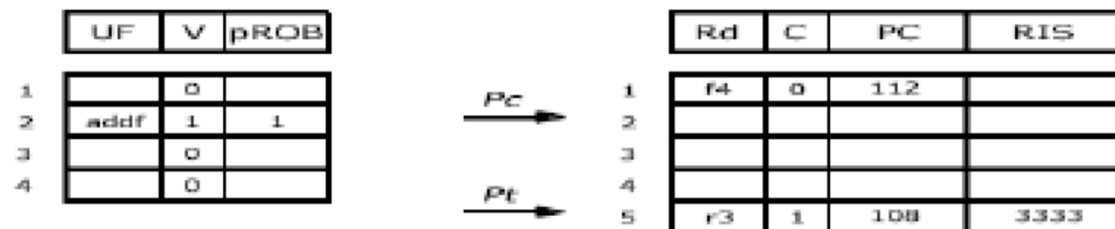
e)



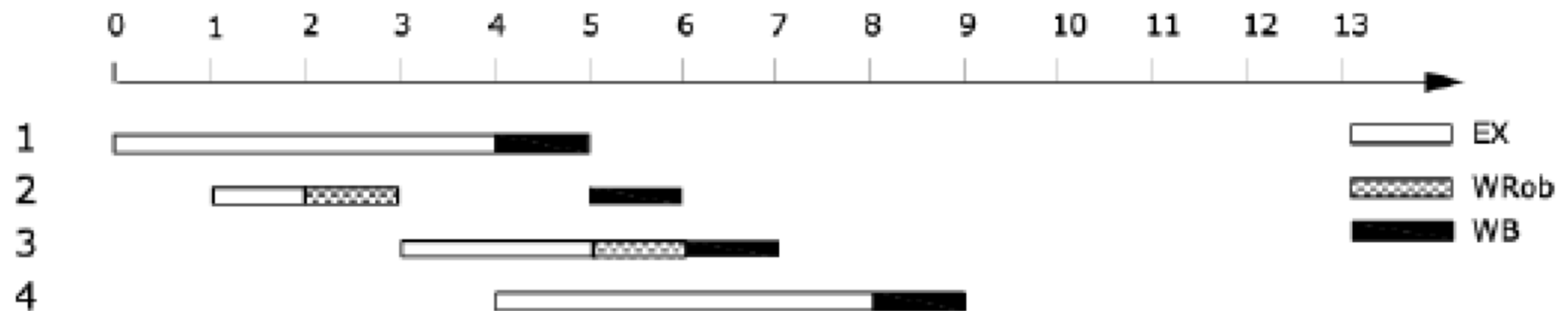
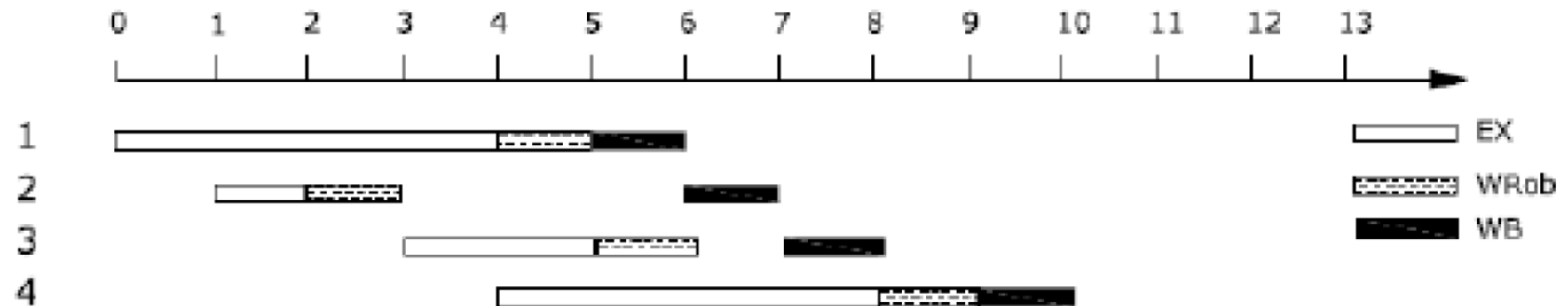
f)



g)



## Buffer di riordinamento (ROB)

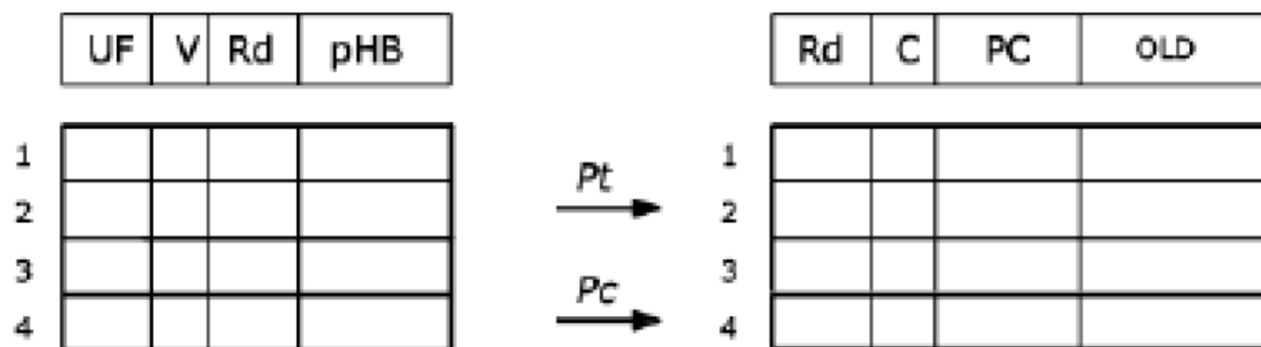
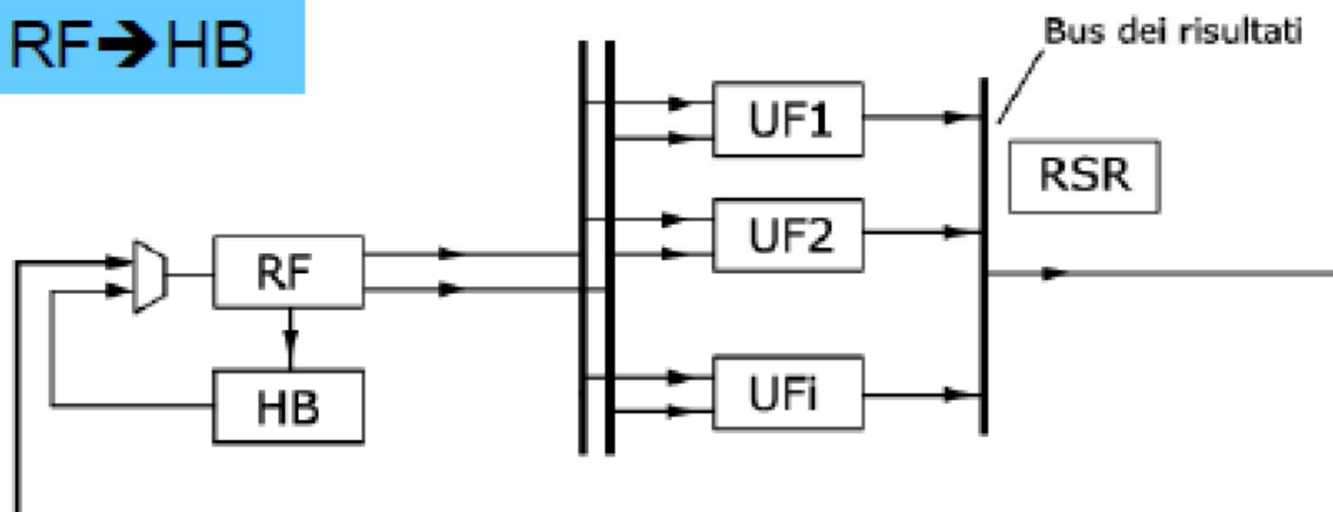


## History Buffer

- ◆ L'istruzione completata dalla sua UF scrive immediatamente i risultati in RF, ma il vecchio valore contenuto in RF resta in HB fino a che l'istruzione non emerge:
  - L'ordine di emissione deve garantire coerenza rispetto alle dipendenze dati
  - I motivi di ripristinare lo stato corrente sono legati alle interruzioni e alle previsioni di salto errate
- ◆ E' gestito come nel caso di ROB con due puntatori Pt e Pc e come un buffer circolare. Un elemento del HB è costituito da 4 componenti: **C**, **PC**, **Rd** e **OLD** (per mantenere il valore di Rd al momento dell'emissione del risultato e ripristinarlo nel caso di errore per mantenere la coerenza).

## History Buffer

Ci vuole un  
percorso  $RF \rightarrow HB$



## Gestione conflitti di controllo

- ◆ Predizione errata con esecuzione fuori ordine:
  - Istruzioni provenienti dal percorso errato possono essere state completate prima della soluzione della salto
  - In tal caso, lo stato della macchina è modificato ed occorre ripristinare lo stato che si aveva prima dell'istruzione di salto
  
- ◆ **Nota:** nel caso di pipeline lineare bastava svuotarla delle istruzioni prese dal percorso errato



## Ripristino stato coerente con HB

- ◆ Si aggiunge alle componenti di HB un ulteriore campo EPR (errata previsione). Quando un'istruzione di salto arriva in testa ad HB, se EPR è a 1:
  - Viene bloccato HB e viene bloccata l'emissione di nuove istruzioni
  - Si attende che le operazioni ancora attive vengano completate (cioè si svuota la pipeline)
  - Partendo dalla coda verso la testa, vengono cancellati gli elementi in HB e, contemporaneamente, vengono **riscritti nei registri destinazione i vecchi valori** contenuti in HB fino ad arrivare alla prima istruzione sul percorso sbagliato
  - L'esecuzione riprende prelevando l'istruzione proveniente dal percorso corretto

## Superscalari?

- ◆ Abbiamo assunto l'emissione e il ritiro di una istruzione alla volta
- ◆ Se viene emessa una sola istruzione alla volta non accadrà mai che vengano eseguite più di una istruzione per clock:
  - Decodificare ed emettere più istruzioni in parallelo
  - Ritirare più istruzioni in parallelo
- ◆ Come nel caso del Pentium visto precedentemente.
- ◆ Il resto rimane come descritto finora.