Parte V

Architettura della CPU

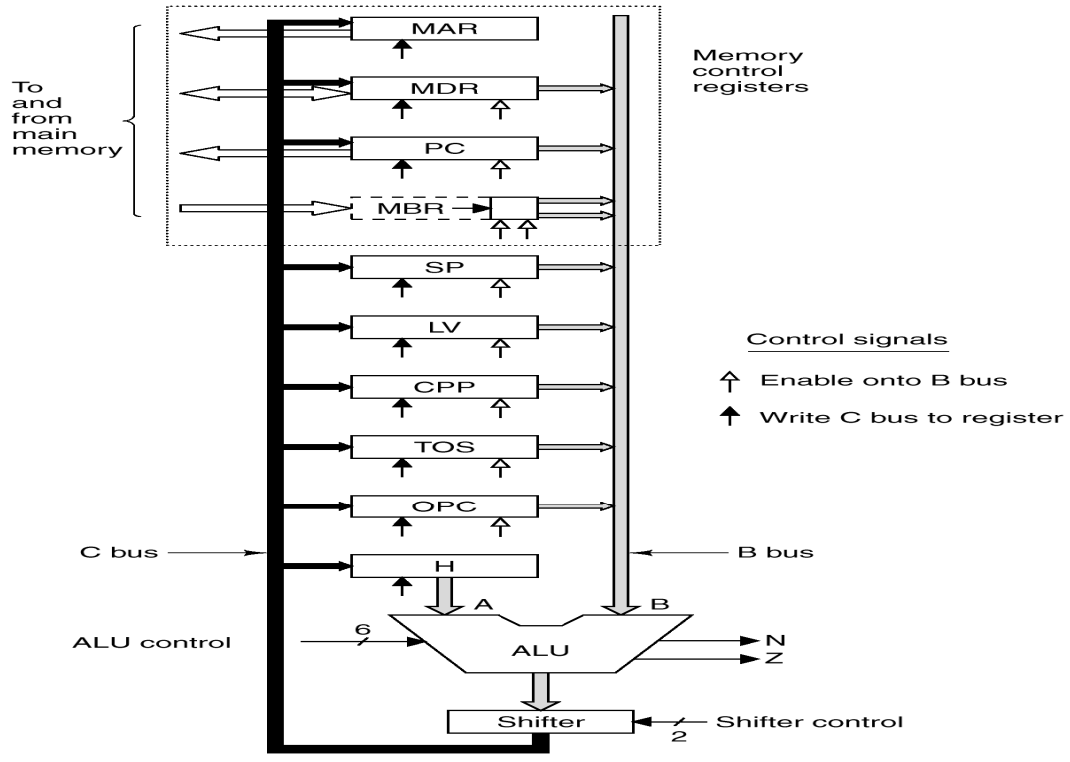
Interpretazione delle istruzioni

- In un'architettura *microprogrammata* le *istruzioni macchina* non sono eseguite direttamente dall'hardware
- L'hardware esegue istruzioni a livello più basso: *microistruzioni*
- All'esecuzione di ciascuna istruzione macchina corrisponde l'esecuzione di diverse microistruzioni
- Di fatto viene eseguito un programma, detto *microprogramma,* i cui dati sono le istruzioni macchina, e il cui risultato è l'interpretazione delle dette istruzioni
- **Vantaggi**: flessibilità, possibilità di gestire istruzioni macchina complesse
- **Svantaggi**: esecuzione relativamente lenta; ciascuna istruzione richiede più fasi elementari

Un esempio di μ-architettura

- Implementazione di una JVM (*Java Virtual Machine*) con sole istruzioni su interi
- In questo corso ci limitiamo a:
  - *La microarchitettura* (data path)
  - *La temporizzazione di esecuzione*
  - *L'accesso alla memoria (cache)*
  - *Il formato delle μ-istruzioni*
  - *La sezione di controllo*
- Sul libro l'esempio è sviluppato fino alla definizione di un μ-programma completo per una JVM (con aritmetica intera)
- *Questa ultima parte non fa parte del programma*

Cammino dei dati (*data path*)



ALU, registri bus

La microarchitettura è costituita dalla Unità Aritmetico Logica (ALU)*, da un insieme di registri e da un insieme di bus che consentono il trasferimento dei dati*

- **Registri:** contraddistinti da nomi simbolici ciascuno con una precisa funzione
- **Bus B**: presenta il contenuto di un registro all'ingresso B della ALU
- **ALU**: ha come ingressi il bus B e il registro H (*holding register*), effettua le manipolazioni dei dei dati
- **Shifter**: consente di effettuare vari tipi di *shift* sull'uscita della ALU
- **Bus C**: permette di caricare l'uscita dello *shifter* in uno o più registri

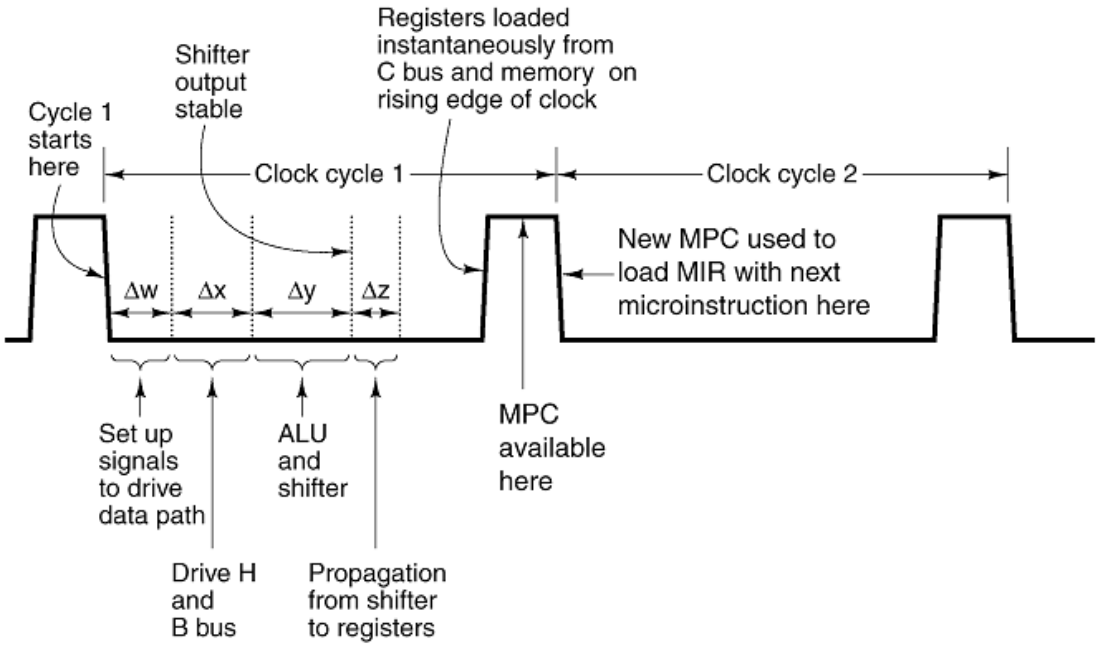
Segnali di controllo

I trasferimenti dei dati e le funzioni svolte dalla ALU e dallo shifter sono attivate da un insieme di segnali di controllo

- **B bus enable**: trasferisce il contenuto di un dato registro sul bus B
- **Write C bus:** trasferisce il contenuto dello *shifter* in uno o più registri tramite il bus C
- **Controllo della ALU**: 6 segnali selezionano una delle funzioni calcolabili dalla ALU
- **Controllo dello shifter**: 2 segnali specificano se e come scalare l'uscita della ALU

Ulteriori connessioni permettono lo scambio di dati tra i registri MDR e MBR e due memorie cache (dati e istruzioni)

Temporizzazione del ciclo



Temporizzazione del ciclo (2)

- In *ciascun ciclo di clock* viene eseguita una microistruzione:
  1) Caricamento di un registro sul bus **B**
  2) Assestamento di ALU e shifter
  3) Caricamento di registri dal bus **C**
- Temporizzazione:
  - *Fronte di discesa*: inizio del ciclo
  - **Δw**: tempo assestamento segnali di controllo
  - **Δx:** tempo assestamento bus B
  - **Δy**: tempo assestamento ALU e shifter
  - **Δz**: tempo assestamento bus C
  - *Fronte di salita*: caricamento registri dal bus C
- I tempi **Δw**, **Δx**, **Δy**, **Δz** costituiscono *sottocicli* (impliciti)

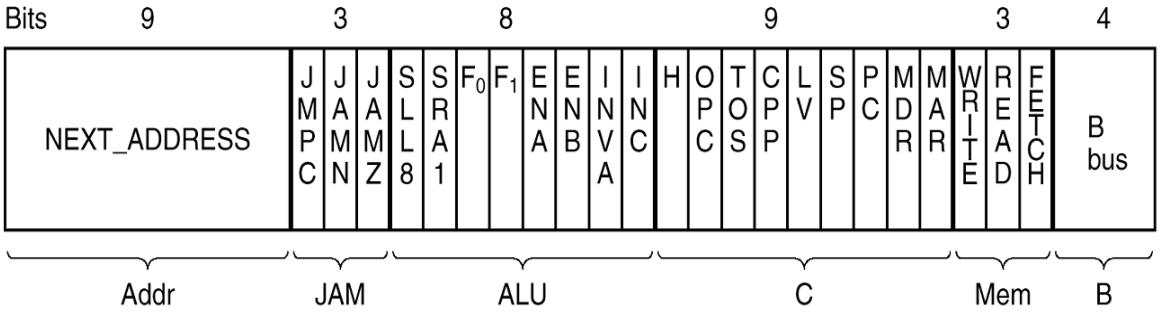
Accesso alla Memoria

- Accesso *parallelo* a due cache :
  - *Cache Dati*: 32 bit indirizzabili a *word* (lettura e scrittura)
  - *Cache Istruzioni*: 8 bit indirizzabili a *byte* (solo lettura)
- Registri coinvolti:
  - **MAR** (*Memory Address Register*): contiene l'indirizzo della word dati
  - **MDR** (*Memory Data Register*): contiene la word dati
  - **PC** (*Program Counter*): contiene l'indirizzo del byte di codice
  - **MBR** (*Memory Buffer Register*): riceve il byte di codice (sola lettura)
- Caricamento di **MBR**:
  - Estensione a 32 bit con tutti 0
  - Estensione del bit più significativo (*sign extension*)

Struttura delle μ-istruzioni

- Una μ-istruzione da **36 bit** contiene:
  - **A)** I segnali di controllo da inviare al *data path* durante il ciclo
  - **B)** Le informazioni per la scelta della μ-istruzione successiva
- **Segnali di controllo**:
  - **9** Selezione registri sul bus **C**
  - **9** Selezione registro sul bus **B**
  - **8** Funzioni ALU e *shifter*
  - **2** Lettura e scrittura dati (**MAR**/**MDR**)
  - **1** Lettura istruzioni (**PC**/**MBR**)
- **Selezione μ-istruzione successiva**:
  - **9** Indirizzo μ-istruzione (su 512)
  - **3** Modalità di scelta
- Dato che *si invia su* **B** *solo un registro per volta*, è possibile codificare i **9** segnali di selezione registro con **4** bit

Formato delle μ-istruzioni

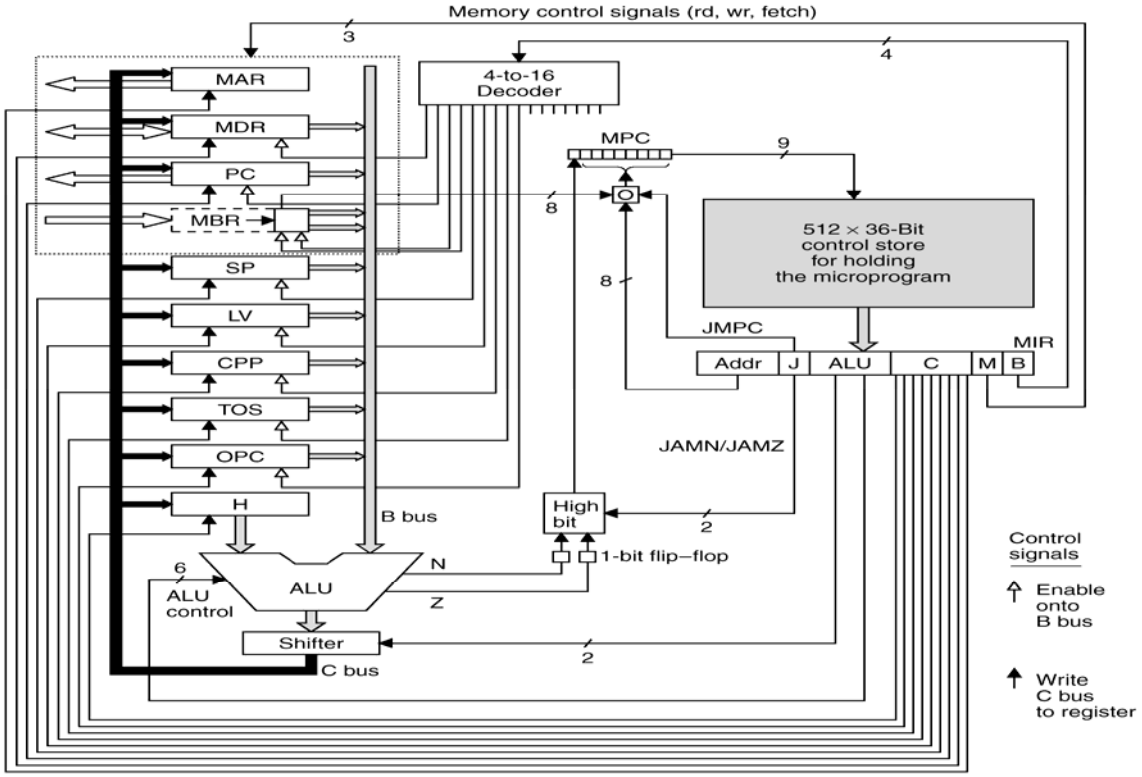


Addr - Indirizzo prossima μ-istruzione
JAM - Scelta prossima μ-istruzione
ALU - Comandi ALU e shifter
C - Registri da caricare da C
Mem - Controllo memoria
B - Registro da inviare su B

B bus registers

0 = MDR	5 = LV
1 = PC	6 = CPP
2 = MBR	7 = TOS
3 = MBRU	8 = OPC
4 = SP	9-15 none

La Sezione di Controllo

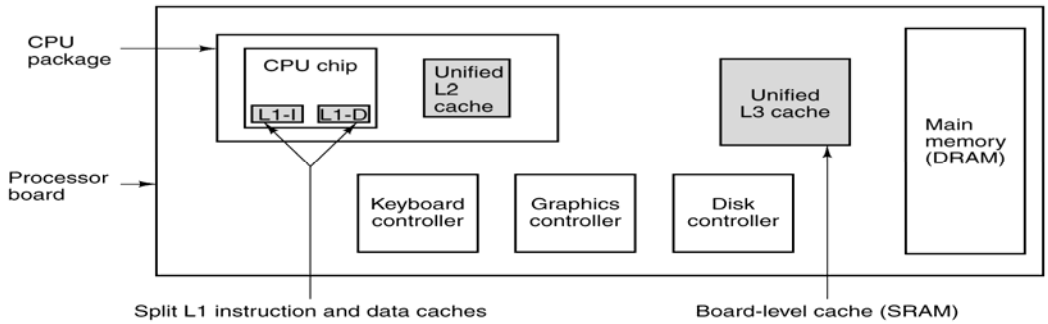


La Sezione di Controllo (2)

- **Control Store**: ROM **512**× **36** bit che contiene le μ-istruzioni
- **MPC** (*Micro Program Counter*): contiene l'indirizzo della prossima μ-istruzione
- **MIR** (*MicroInstruction Register*): contiene la μ-istruzione corrente
- Il contenuto di **MPC** diviene stabile sul livello alto del clock
- La μ-istruzione viene caricata in **MIR** sul fronte di discesa dell'impulso di clock

Temporizzazione della memoria:
- Inizio ciclo di memoria dopo il caricamento di **MAR** e di **PC**
- Ciclo di memoria durante il successivo ciclo di μ-istruzione
- Dati disponibili in **MDR** e **MBR** all'inizio del ciclo di μ-istruzione ancora successivo

Memorie cache



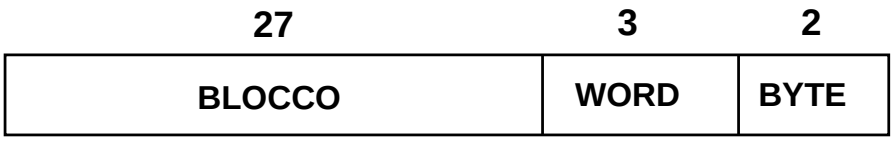
- *Scopo della cache*: disaccoppiare le velocità di CPU e RAM
- *Località spaziale*: alta probabilità di accedere in tempi successivi a indirizzi molto vicini
- *Località temporale*: alta probabilità di accedere più volte agli stessi indirizzi in tempi molto vicini
- *Gerarchie di cache*: a 2 o 3 livelli
- *Cache inclusive*: ciascuna contiene sempre quella del livello superiore

Organizzazione della memoria

- Lo spazio di memoria è organizzato in *blocchi* (da 4 a 64 byte), chiamati anche *linee di cache*
- Ciascuna *linea* contiene più *word*
- Ciascuna *word* contiene più *byte*
- Le *cache* sono organizzate in *righe* (o *slot*): ciascuna contiene una *linea di cache*, cioè un blocco di memoria
- Tutti i trasferimenti avvengono a livello di blocco

Quando una word non viene trovata in cache, si trasferisce l'intera linea dalla memoria, o dalla cache di livello più basso

Struttura dell'indirizzo (esempio)



- *Indirizzi* a **32** bit (spazio di indirizzamento di $\mathbf{2^{32}}$ byte)
- *Linee di cache* (blocchi) di **32** byte
- *Word* di **4** byte

STRUTTURA DELL'INDIRIZZO

- I 27 bit più significativi rappresentano il *numero di blocco*
- I successivi 3 bit il *numero della word* all'interno del blocco
- Gli ultimi due bit il *numero del byte* all'interno della word

Ricerca di un blocco in cache

- Una cache contiene un sottoinsieme di blocchi di memoria di indirizzo non contiguo
- Quando la CPU cerca una word, non sa in quale posizione essa si possa trovare nella cache (se effettivamente c'è)
- Non c'è modo di risalire dall'indirizzo di un blocco di memoria alla sua posizione in cache
- Non è possibile utilizzare il normale meccanismo di indirizzamento delle RAM:
  - Si fornisce un indirizzo
  - Viene letto il dato che si trova allo indirizzo specificato
- Si usano allora *Memorie Associative*

Memorie associative

- Elementi costituiti da due parti: *chiave* e *informazione*
- L'accesso ad un elemento viene effettuato non in base allo indirizzo ma a parte del suo contenuto (*chiave*)
- L'accesso *associativo* avviene in un unico ciclo

Nel caso di una cache:
- *La chiave è il numero del blocco*
- *L'informazione sono i byte del blocco*

Cache a mappa diretta

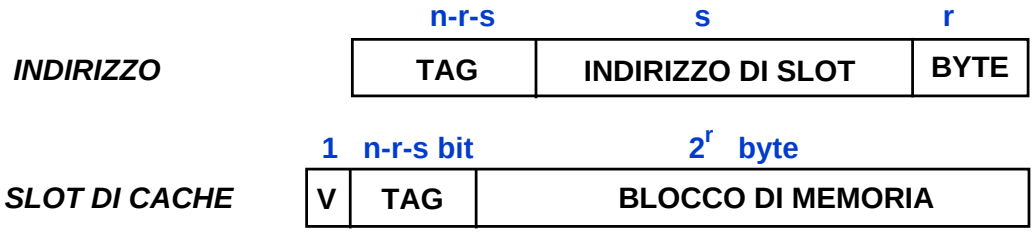
- La memoria è organizzata in blocchi da 2^r byte
- Gli **n-r** bit più alti sono l'*indirizzo di blocco*
- La cache è organizzata in 2^s slot, ciascuna di 2^r byte

Nelle cache a mappa diretta il numero di slot in cui un dato blocco di memoria entra è sempre lo stesso, e dipende funzionalmente dall'indirizzo del blocco

- All'atto della ricerca il numero di slot in cui cercare il blocco viene calcolato nello stesso modo a partire nell'indirizzo
- Pertanto si accede *a colpo sicuro* ad un'unica slot di cache

*Il numero di slot è dato dagli **s** bit meno significativi dell'indirizzo, tolti gli **r** che indirizzano nel blocco*

Struttura dell'indirizzo e della slot



- Più blocchi di memoria *condividono* la stessa slot di cache
- Per distinguerli si usano gli **n-s-r** bit più significativi dell'indirizzo, che vengono denominati **TAG**
- Ciascuna slot di cache in ogni istante contiene
  - il **TAG** del blocco in essa presente
  - i 2^r byte che costituiscono il blocco (*valore del blocco*)
  - un *bit valid* **V** che indica se il contenuto della slot è valido

Ricerca in cache

PROBLEMA

La CPU accede ad un byte di indirizzo dato di **n** bit

– deve prima controllare in cache

– se il tentativo non ha successo accede a memoria

PASSI DELLA RICERCA

– *si calcola l'indirizzo di slot (**s** bit dall'indirizzo di blocco)*

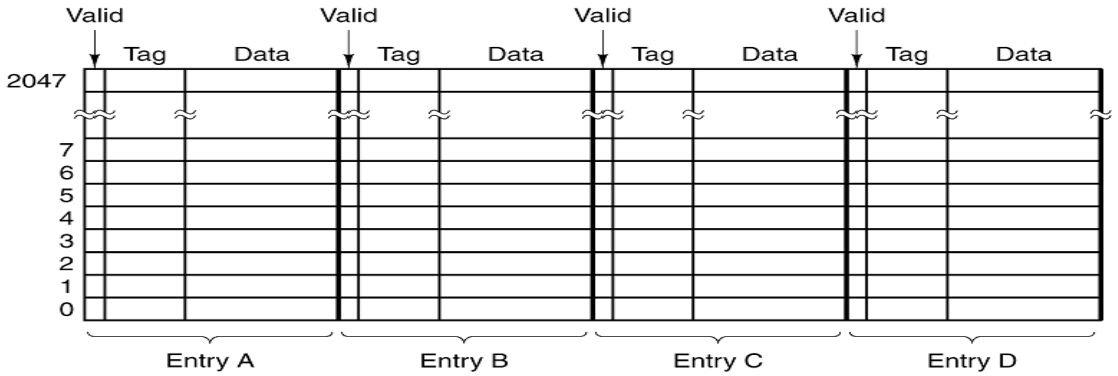
– *si accede alla slot corrispondente all'indirizzo di slot*

– *si confronta il **TAG** nella slot con quello nell'indirizzo*

– *se il confronto ha successo si estrae il byte individuato dagli **r** bit meno significativi dell'indirizzo*

– *altrimenti l'accesso a cache non ha avuto successo*

Cache associative ad insiemi

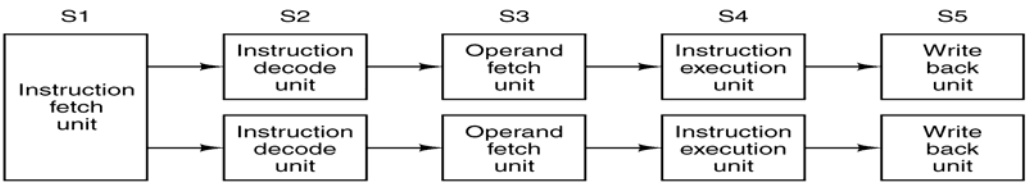


- Ogni slot è costituita da **k** elementi, ciascuno composto di bit valid, tag e blocco
- Un blocco entra in un elemento scelto a caso (e sempre diverso) della slot che gli corrisponde
- Il **TAG** del blocco ricercato viene confrontati in parallelo con i **TAG** *di tutti gli elementi della slot*
- Attenua il problema delle collisioni sistematiche

Politiche di gestione della cache

- La CPU accede alla cache ed effettua il confronto del TAG, poi a seconda dell'esito:
  - *Cache Hit in lettura*: tutto ok
  - *Cache Hit in scrittura*:
    - *politica write through*: scrive anche in memoria
    - *politica write back*: unica scrittura finale, quando il blocco è rimosso dalla cache
  - *Cache Miss in lettura*: il blocco viene trasferito in cache e sovrascrive quello presente (questo va copiato se modificato)
  - *Cache Miss in scrittura*:
    - *politica write allocation*: porta il blocco in cache (conviene per scritture ripetute)
    - *politica write to memory*: scrive direttamente in memoria

Pipeline e architetture superscalari



- Per aumentare la capacità di elaborazione della CPU:
  - *Pipeline a molti stadi* (anche 10 e più)
  - *Architetture superscalari*: parti di pipeline (o intere pipeline) multiple
- **Latenza**: tempo necessario a completare l'elaborazione di un istruzione
- **Banda della CPU**: numero di istruzioni elaborate nell'unità di tempo

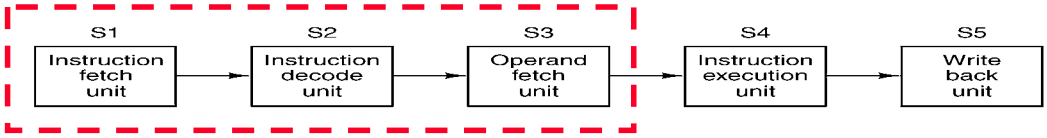
Pipeline e architetture superscalari aumentano la banda della CPU ma non riducono la latenza

Pipeline e salti condizionati

- La pipeline funziona bene se le istruzioni vengono *eseguite in sequenza*
- Il flusso delle istruzioni viene turbato dalla presenza di salti, condizionati e no

Non si sa quali istruzioni fare entrare nella pipeline fino al termine della esecuzione della istruzione di salto

- La pipeline si blocca fino a che l'istruzione di salto non viene elaborata, e si conosce il nome dell'istruzione a cui saltare
- Si svuotano tutti gli stadi della pipeline fino a quello di esecuzione



Pipeline e salti: esempio

```
        CMP i,0      ; compare i to 0
        BNE Else     ; branch to Else if not equal
Then:   MOV k,1      ; move 1 to k
        BR Next      ; unconditional branch to Next
Else:   MOV k,2      ; move 2 to k
Next:
```

- *Non si sa che istruzione prendere dopo BNE*
- *Occorre aspettare il risultato dell'esecuzione della CMP, che a sua volta dipende dal valore di i …*
- *Anche un salto incondizionato (come BR) da problemi, occorrono più stadi prima di capire...*

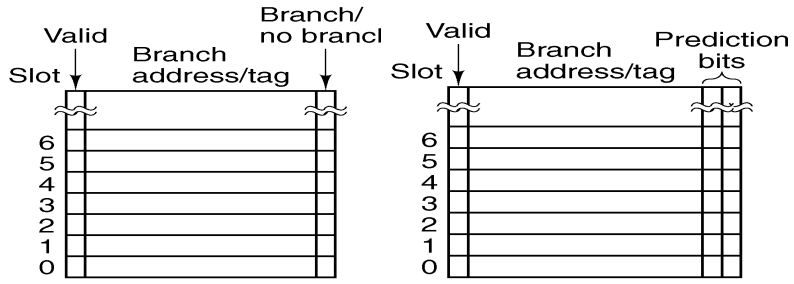
Soluzione ingenua

- Piuttosto che andare in stallo meglio fare una qualche scelta, e poi in caso rimediare
- Per esempio:
  - *Se il salto è in indietro si salta*
  - *Se il salto è in avanti non si salta*

> *Questo metodo funziona bene nel caso dei cicli*

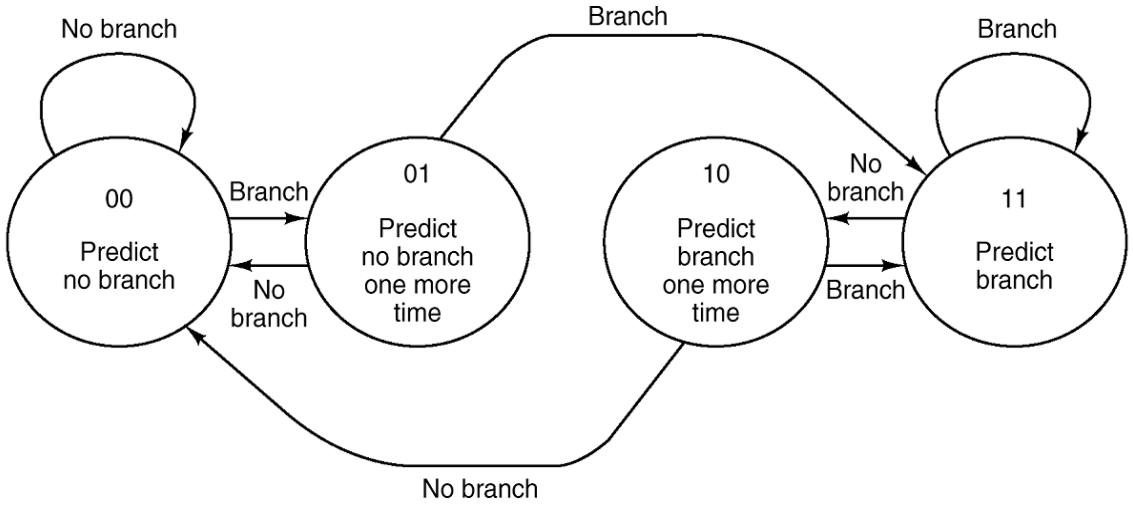
- Se la scelta era quella giusta tutto va bene
- Se era sbagliata, oltre al tempo perso, occorre disfare gli effetti delle istruzioni parzialmente eseguite:
  - *Squashing* (svuotamento) della pipeline
  - Ripristino del valore dei registri modificati dalle istruzioni eliminate

Predizione dinamica dei salti



- Si gestisce una tavola per memorizzare l'esperienza su ciascun salto
- Hardware speciale, gestito come una cache
- Ogni slot contiene:
  - Il *tag* di un indirizzo di salto
  - Un bit che indica se si salta o no
- Se la previsione risulta sbagliata si inverte il bit
- Funziona bene nel caso di cicli

Predizione dei salti a 2 bit



- Ciascuno stato rappresenta una predizione
- Si cambia stato a seconda di quello che in realtà avviene
- Non si cambia stato (cioè predizione) alla prima variazione
- Si comporta meglio della soluzione ingenua nel caso di cicli

Predizione statica dei salti

- Un approccio del tutto diverso è quello di disporre di due istruzioni diverse:
  - *Salto molto probabile*
  - *Salto poco probabile*
- Il compilatore, in base alla semantica del programma, decide a seconda dei casi quale conviene usare generando il codice
- La CPU sfrutta questa informazione

ESEMPIO

for (i=0; i < 1000000; i++) {…}

- Il salto di fine del ciclo conviene classificarlo come 'molto probabile'
- Fallisce solo una volta su un milione

Esecuzione in-order

- Nelle architetture superscalari il controllo cerca di saturare le pipeline, cioè di avviare l'esecuzione del maggior numero di istruzione possibile
- Occorre decidere quali vincoli mettere sull'ordine in cui le istruzioni vengono eseguite
- Nella generalità dei casi si pongono i vincoli:

> **In-order issue:** *le istruzioni devono iniziare in ordine*
> **In-order retire:** *le istruzioni devono terminare in ordine*

- La scelta più semplice e di imporre che *le istruzioni siano anche eseguite nello stesso ordine in cui sono state decodificate:*
- Scelta molto limitante sotto il profilo delle prestazioni

Vincoli di ordinamento

- Prima di iniziare un'istruzione *si considera quali registri usa* e il loro stato attuale
- Si *pospone l'inizio dell'istruzione* per garantire il rispetto dei seguenti vincoli di ordinamento:
  - ***RAW (Read After Write)****:* uno dei suoi operandi viene scritto da un'altra istruzione:
  - ***WAR (Write After Read)****:* il suo risultato viene letto da un'altra istruzione
  - ***WAW (Write After Write)****:* il suo risultato viene scritto da un'altra istruzione

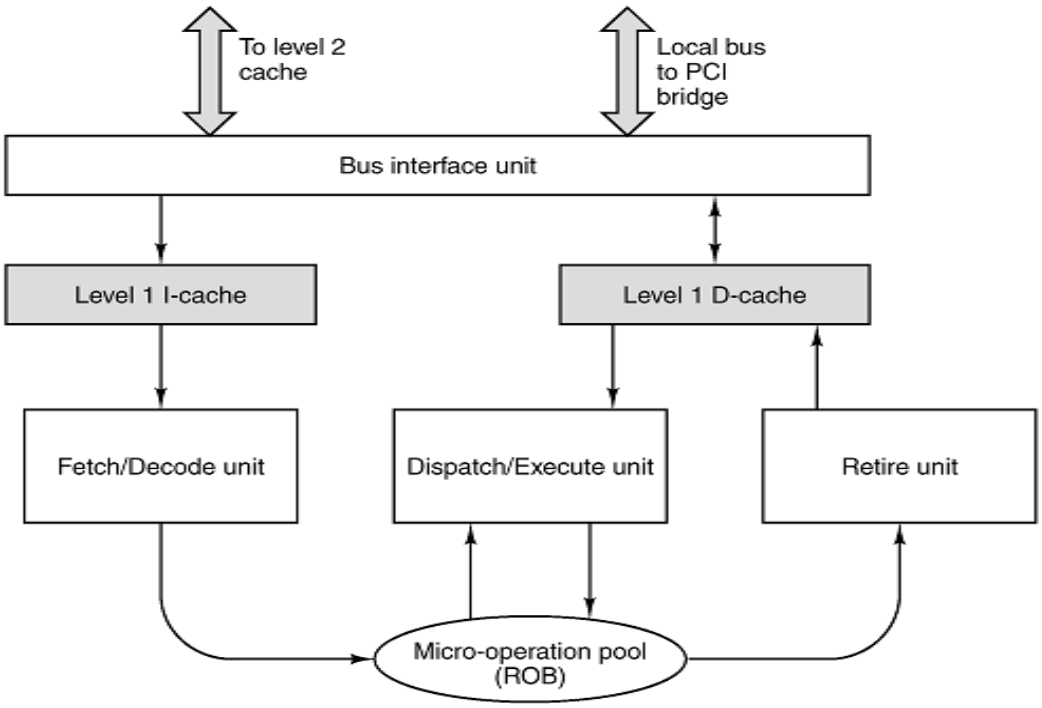
> Questi vincoli garantiscono la correttezza dell'esecuzione

Esecuzione out-of-order

- Rilassando i vincoli di ordine si hanno migliori prestazioni
- Si deve tuttavia garantire una esecuzione corretta
- Le motivazioni dei vincoli **RAW**, **WAR** e **WAW** devono continuare ad essere rispettate
- Si possono effettuare *azioni aggiuntive* per garantire la semantica dell'escuzioni
  - *Si deve tener conto delle scritture fatte da istruzioni in attesa*
  - Si eliminano conflitti sostituendo alcuni registri con *registri scratch (register renaming)*

Problemi nella gestione delle interruzioni: non si sa dove esattamente *l'esecuzione è stata interrotta*

Microarchitettura del Pentium II



Unità funzionali

Fetch-Decode Unit

Preleva le istruzioni dalla cache L1 e le scompone in *micro-operazioni che vengono messe nel* **ROB** (**R**e**O**rder **B**uffer)

Dispatch-Execute Unit

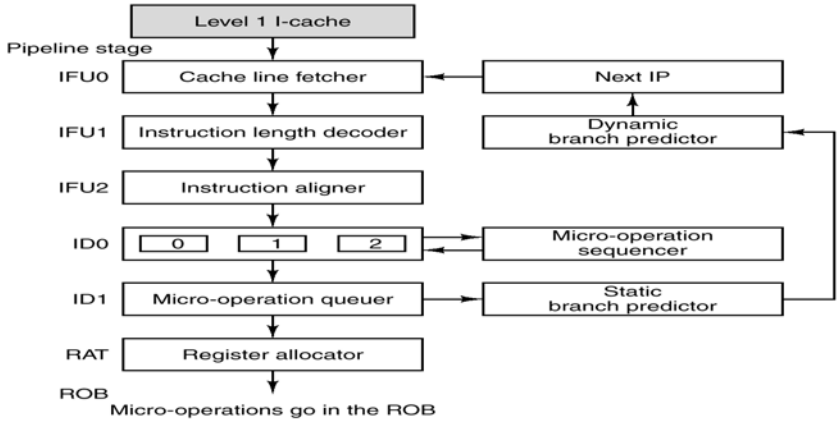
Preleva dal **ROB** le μ-operazioni che possono andare in esecuzione e le esegue su unità funzionali multiple

Retire Unit

Ritira dal **ROB** le μ-operazioni la cui esecuzione è terminata, aggiorna i registri 'ufficiali' aggiornati dalle μ-operazioni

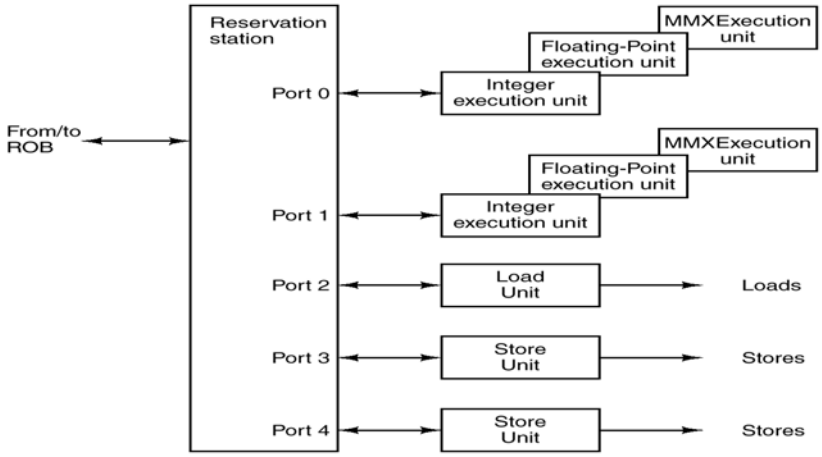
On-order issue, out of order execution, in-order retire

Fetch/decode unit



- Caricamento di *linee di cache* di 32 byte
- Scomposizione in μ-operazioni (*quaterne*): ⟨**OP, R_D ,R_{S1}, R_{S2}**⟩
- Predizione dei salti a 4 bit (pipeline a 12 stadi: *penalità forte*)
- *Ridenominazione dei registri* per evitare dipendenze
- Pool di 40 *registri scratch*
- Genera 3 μ-operazioni per ciclo

Dispatch/execute unit



- Esegue le μ-operazioni *risolvendo le dipendenze e i conflitti*
- Gestione di uno *scoreboard* (di μ-op)
- 5 porte e 9 unità funzionali: 3 unità duplicate, 1 di Load e 2 di Store
- Può eseguire fino a 5 μ-op per ciclo
- *Reservation Station*: coda di 20 μ-op pronte ad essere eseguite
- Complesso algoritmo di scheduling

Retire unit

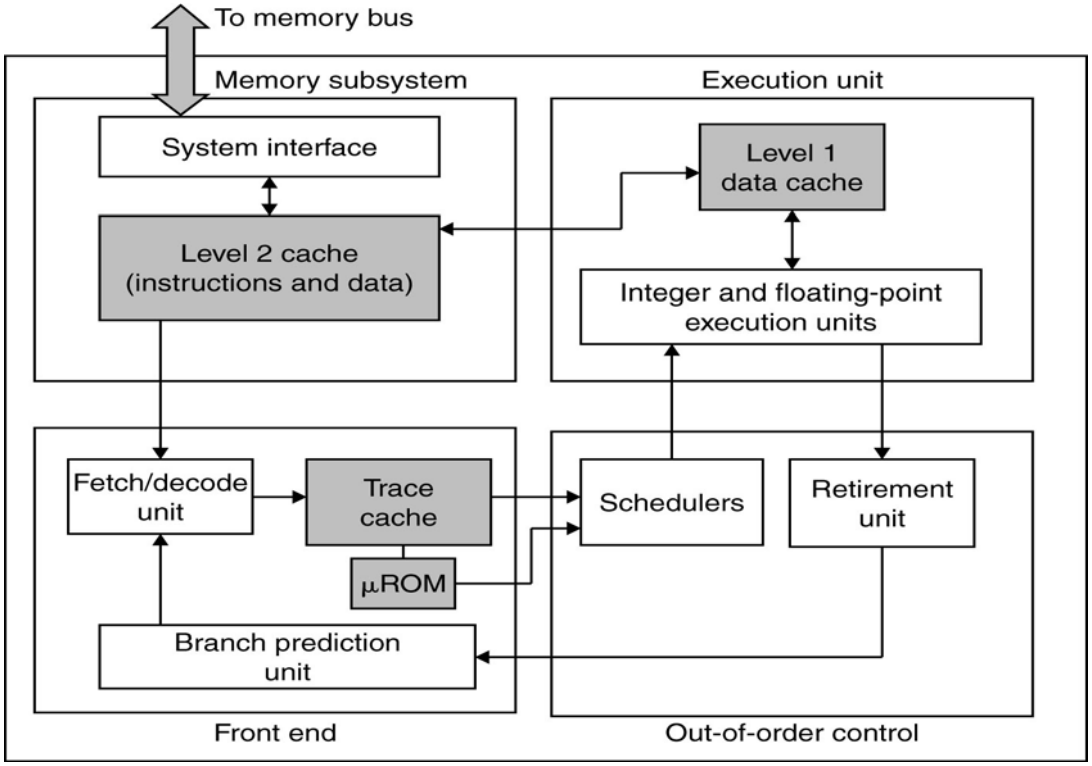
- Le μ-operazioni dopo l'esecuzione tornano nella *Reservation Station*
- Poi vengono rimesse nel ROB in attesa di essere ritirate
- Il ritiro avviene *in ordine* a differenza dell'esecuzione
- Solo all'atto del ritiro i valori dei registri 'ufficiali' sono cambiati
- Nello spirito dell'*esecuzione speculativa* alcune istruzioni vengono *eseguite e mai ritirate*
- Solo le μ-operazioni di istruzioni che era corretto eseguire vengono ritirate, e in ordine

Il ritiro in ordine assicura la capacità di rollback

Microarchitettura del Pentium 4

- Architettura innovativa rispetto ai suoi predecessori
- Pentium II e Pentium III erano basati sull'architettura P6
- Il Pentium 4 è basato sulla μ-architettura NetBurst
- A livello delle istruzioni macchina il Pentium 4 appare come un processore CISC:
  - Repertorio di istruzioni molto vasto
  - Solo pochi registri visibili al programmatore
- L'architettura interna è di tipo RISC:
  - Esecuzione veloce di istruzioni semplici
  - Centinaia di registri a disposizione (renaming)

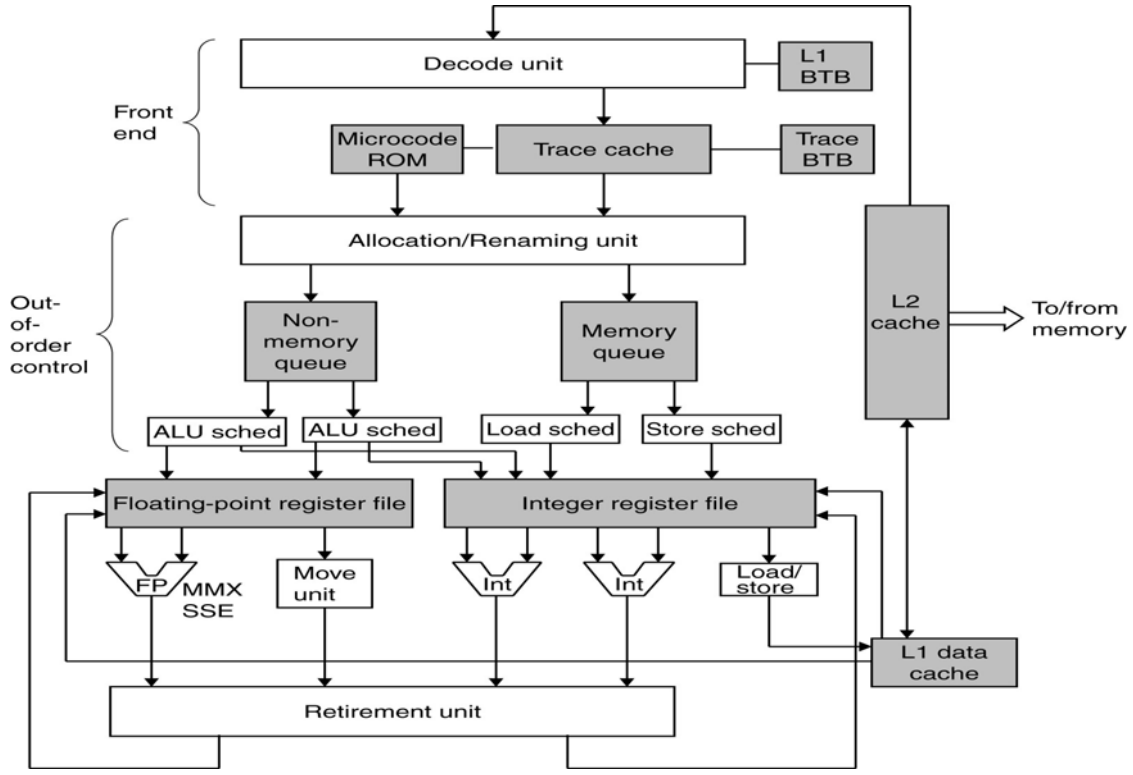
Microarchitettura NetBurst



Microarchitettura NetBurst (2)

- *Memory subsystem*
  - Cache L2 da 256 KB cresciuti poi fino ad 1 MB
  - Cache associativa ad 8 vie, *write-back,* con line da 128 Byte
- *Front end*
  - Preleva *in order* le istruzioni dalla cache L2 e le decodifica
  - Ogni istruzione è suddivisa in una sequenza di μ-operazioni
  - Le μ-op sono messe nella *Trace cache* (cache L1 istruzioni)
- *Execution unit*
  - Unità di esecuzione multiple che lavorano in parallelo
  - Dati conservati nella cache L1 dati
- *Out-of-order control*
  - Esecuzione *out-of-order* delle μ-operazioni
  - *Renaming* dei registri per rilassare vincoli WAR e WAW
  - Ritiro *in-order* delle istruzioni: interruzioni 'precise'

Pipeline NetBurst



Pipeline NetBurst: sezione fetch-decode

- La *Trace cache* prende il posto della cache L1 istruzioni
- Nel Pentium II e III la cache L1 lavora sulle istruzioni macchina
- Nel Pentium 4 la Trace cache contiene μ-operazioni
- La Trace cache contiene fino a 12 K μ-operazioni
- Ciascuna μ-operazioni eseguita più volte in caso di cicli
- La Trace cache viene riempita al ritmo di tre μ-op per ciclo
- BTB (*Branch Target Buffer*) usati nella previsione dei salti

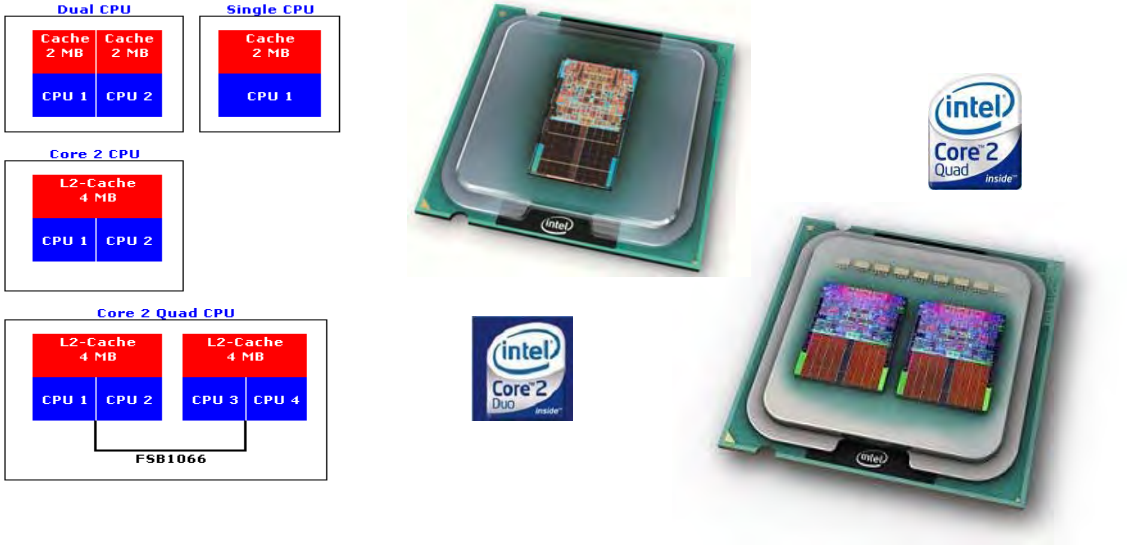
Frazionare le istruzioni ha due scopi:
- Istruzioni a livello CISC $\Rightarrow$ μ-operazioni a livello RISC
- Sfruttare il parallelismo all'interno delle istruzioni

Pipeline NetBurst: out-of-order control

- La *Allocation/Renaming unit* verifica quali μ-op possono partire
- 120 registri scratch sono usati per il *register renaming*
- Code separate per operazioni su registri e memoria
- Fino a 126 μ-op contemporaneamente 'in volo'
- Un ROB (*ReOrder Buffer*) traccia il loro stato di esecuzione
- Cache L1 da 8K associativa a 4 vie, *write-through,* line da 64 B
- 4 scheduler smistano le μ-op alle unità di esecuzione
- Scheduler e ALU lavorano a velocità doppia rispetto al clock
- A 3 GHz vengono eseguite 12 miliardi di μ-op al secondo
- Ritiro delle istruzioni in ordine: se arriva un interrupt si perdono le μ-op delle istruzioni incomplete

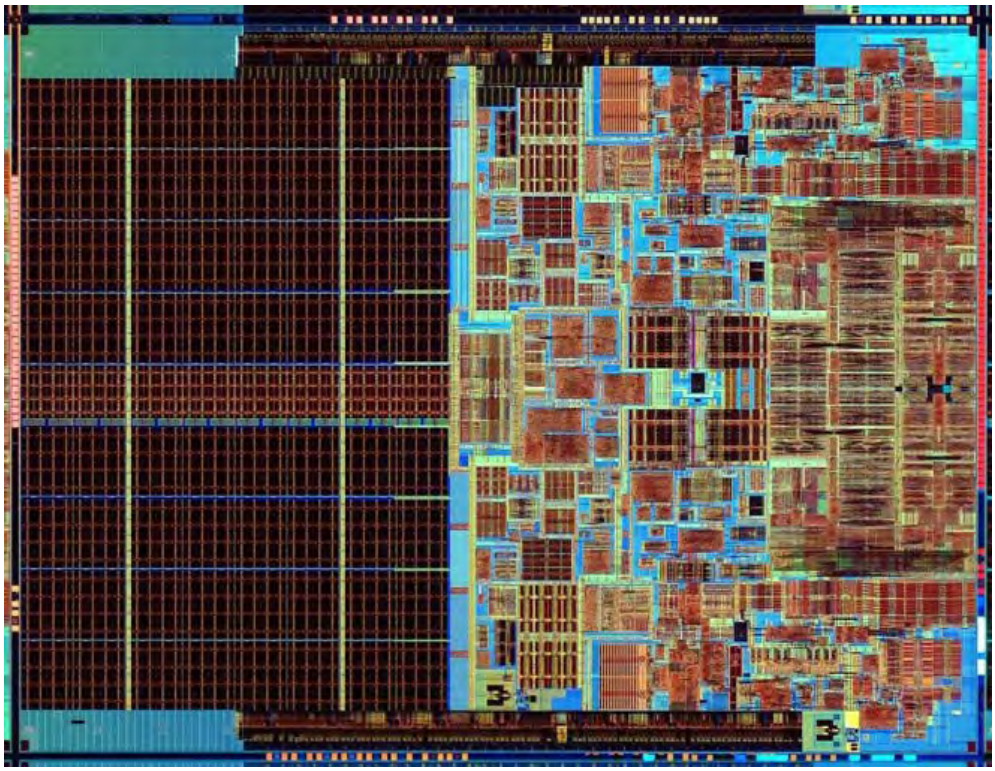
In-oder issue, out-of-order execute, in-order retire

Architetture multi core



- Sullo stesso chip vengono messi più CPU (*core*='nucleo')
- *Die* (piastrina) monolitico: unica architettura integrata (Itanium 2)
- *Die* singolo: più CPU indipendenti sulla stessa piastrina
- *Die* multiplo: più piastrine nello stesso contenitore

Architetture multi core: il chip



Architetture multi core: pro e contro

Vantaggi

- Maggiore potenza di elaborazione dovuta al parallelismo
- Frequenze più basse: si riduce la dissipazione
- Si sfruttano moduli di CPU esistenti e collaudati
- Semplifica la gestione della *coerenza di cache*

Svantaggi

- Occorre che il carico abbia il necessario parallelismo
- Le connessioni con l'esterno diventano collo di bottiglia
- Banda con la memoria condivisa da più CPU

Gestione della cache

- Cache L2 separate: necessari trasferimenti interni
- Cache L2 unica condivisa: soluzione più efficiente, possibile solo nel caso di die singolo