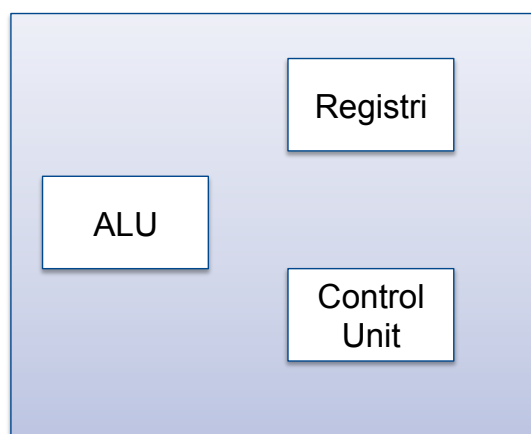


# Struttura e Funzione del Processore

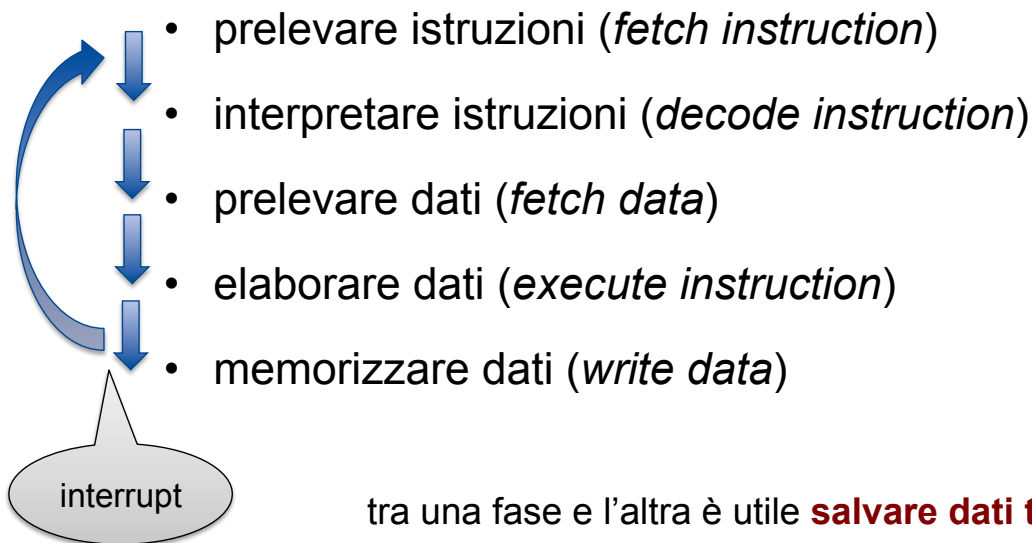
parte II

## Struttura CPU



- **ALU** (Arithmetic and Logic Unit) esegue le operazioni
- **Control Unit** controlla le operazioni di ALU e movimento di dati e istruzioni dentro/fuori CPU
- **Registri** memoria interna veloce
- **Internal processor Bus** per trasferire dati tra registri e ALU

# Compiti di CPU



tra una fase e l'altra è utile **salvare dati temporanei** nei *registri* (e.g. la prossima istruzione da eseguire)

## Ciclo esecutivo delle istruzioni

- l'esatta sequenza di eventi che avvengono durante un instruction cycle dipende
  - dall'architettura della CPU
  - dalla specifica istruzione che si esegue

- **mentre** si **esegue** un'istruzione, si può **prelevare** la successiva

in genere non deve accedere alla memoria principale

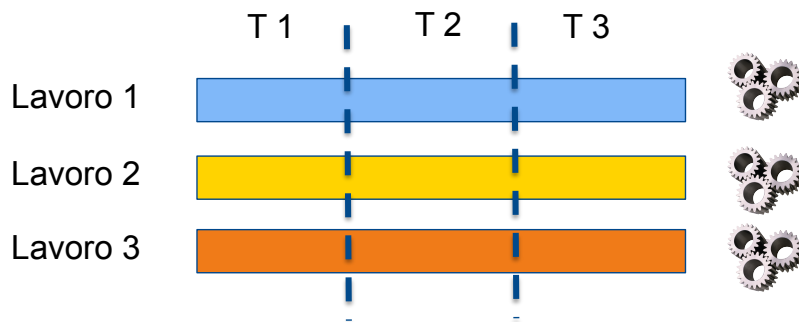
deve accedere alla memoria principale

### Instruction prefetch



# Parallelismo

- **scomporre** il lavoro in diverse attività
- eseguire più attività **contemporaneamente**
- il lavoro viene **completato in meno tempo**



in 3 unità di Tempo

- 1 esecutore termina 1 solo lavoro
- 3 esecutori terminano 3 lavori

**Parallelismo totale**

**MA**

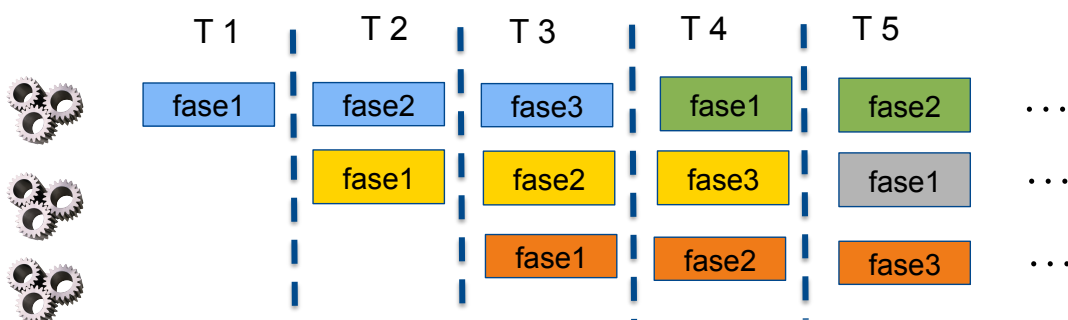
se c'è **dipendenza funzionale**  
tra un lavoro e il successivo?

# Parallelismo



*Dipendenza funzionale* tra lavori successivi

- ogni lavoro è diviso in 3 fasi successive
- la fase 1 di *Lavoro i* deve essere eseguita dopo la fase 1 del precedente *Lavoro i-1*



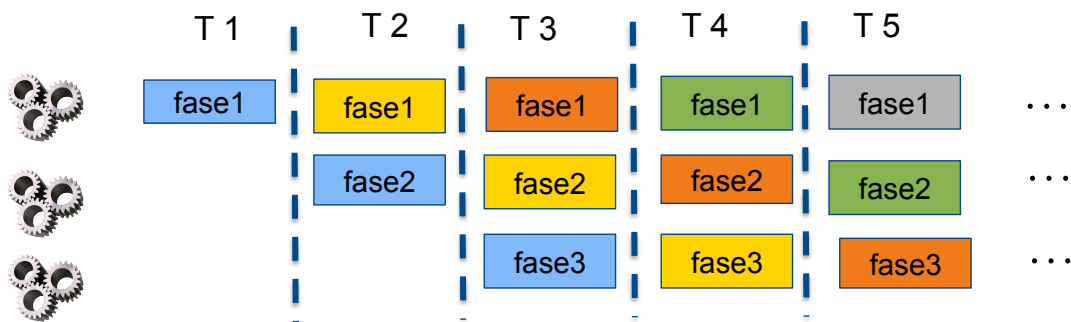
**Esecutori generici:**

- ognuno esegue un lavoro completo
- **a regime** ha lo stesso throughput del parallelismo totale
- ognuno ha le risorse necessarie per ogni fase: **sistema totalmente replicato**


# Parallelismo

## Esecutori specializzati

- ogni esecutore svolge sempre la stessa fase di ognuno dei lavori
- ogni esecutore **ha solo le risorse per eseguire quella fase**
- ogni lavoro passa da un esecutore all'altro
- **a regime** ha lo stesso throughput del parallelismo totale



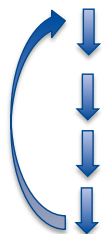
# Pipeline – catena di montaggio

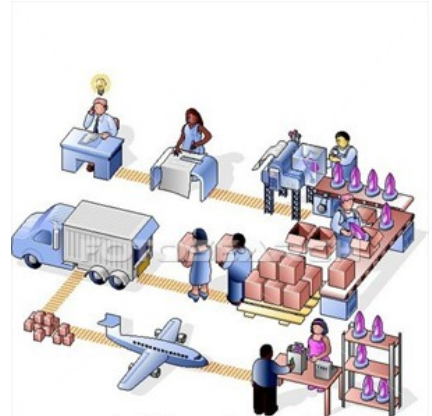
- si **decompone** un lavoro in **fasi successive**
    - un prodotto deve passare **una fase dopo l'altra**
    - ogni fase è realizzata da un **diverso** operatore
    - **nello stesso istante**
      - prodotti diversi sono in fasi diverse (**parallelismo**)
    - **l'istante successivo**
      - ogni fase ripete lo stesso lavoro sul prodotto successivo,
      - ogni lavoro avanza alla fase successiva
    - operatori/fasi diverse usano risorse diverse evitando conflitti (se possibile)
- 



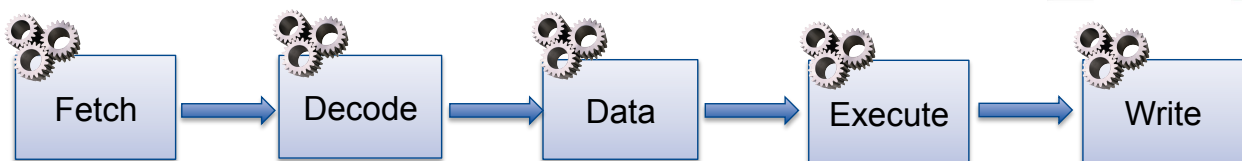
# Pipeline – catena di montaggio

diverse **fasi del ciclo esecutivo** di un' istruzione

- 
- prelevare istruzioni (*fetch instruction*)
  - interpretare istruzioni (*decode instruction*)
  - prelevare dati (*fetch data*)
  - elaborare dati (*execute instruction*)
  - memorizzare dati (*write data*)

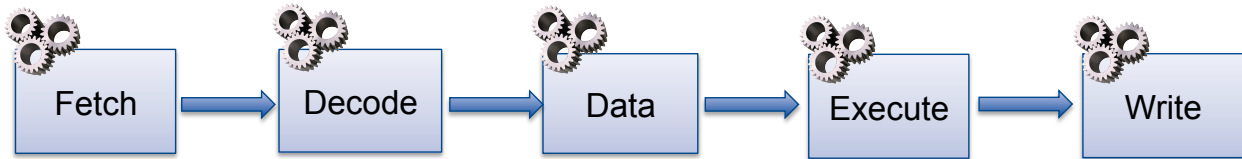
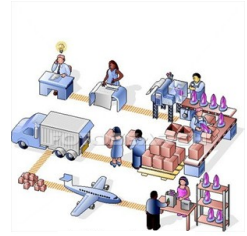


## Pipeline



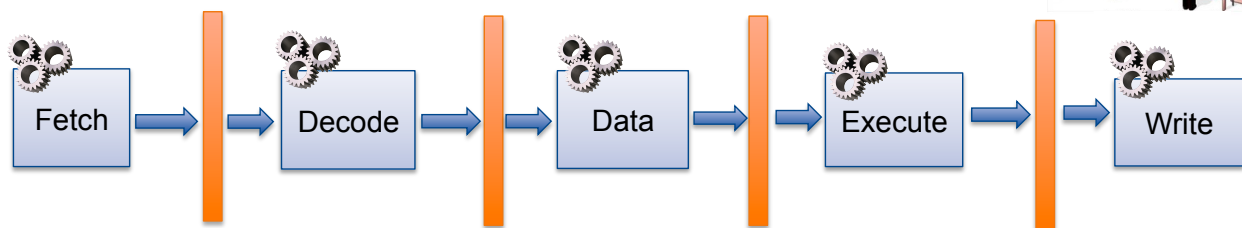
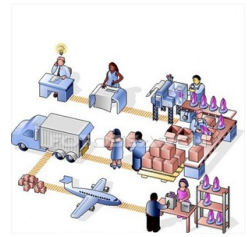
- **nello stesso istante:**
  - **istruzioni** diverse sono in fasi diverse
- **l'istante successivo**
  - ogni fase ripete lo stesso lavoro **sull'istruzione** successiva
  - ogni **istruzione** avanza alla fase successiva

# Pipeline



- ogni fase è realizzata da una ***diversa unità funzionale della CPU***
- operatori/fasi diverse usano risorse diverse evitando conflitti (se possibile)

# Pipeline



- ogni fase è realizzata da una ***diversa unità funzionale della CPU***
- operatori/fasi diverse usano risorse diverse evitando conflitti (se possibile)
- tra due fasi successive si inseriscono dei **buffer (registri)** su cui si scrivono/leggono **dati temporanei** utili alla fase successiva

# Miglioramento delle prestazioni?

**il prefetch non raddoppia le prestazioni:**

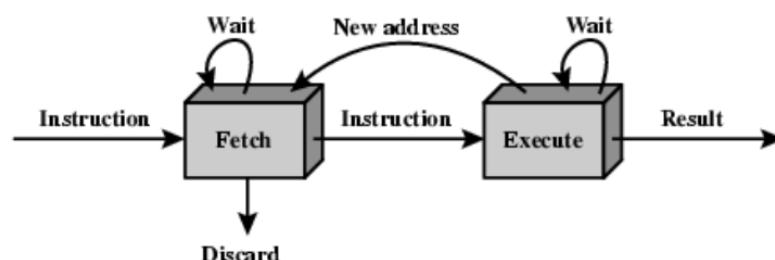
- la fase di **fetch è più breve**, ma prima di poter iniziare il fetch successivo **deve attendere** che termini anche la fase di esecuzione



# Miglioramento delle prestazioni?

**il prefetch non raddoppia le prestazioni:**

- la fase di **fetch è più breve**, ma prima di poter iniziare il fetch successivo **deve attendere** che termini anche la fase di esecuzione
- se viene eseguito un **jump o branch**, la prossima istruzione da eseguire **non è quella che è appena stata prelevata**:
  - la fase di **fetch deve attendere** che la **fase execute le fornisca l'indirizzo** a cui prelevare l'istruzione
  - la successiva fase di **execute deve attendere** che sia prelevata l'istruzione, perché quella pre-fetched non era valida



# Miglioramento delle prestazioni?

- la suddivisione in fasi **aggiunge overhead** per
  - spostare i dati nei buffer tra una fase e l'altra
  - per gestire il cambiamento di fase
- questo overhead potrebbe essere significativo quando:
  - istruzioni successive **dipendono logicamente** da quelle precedenti,
  - quando ci sono **salti**,
  - quando ci sono **conflitti negli accessi alla memoria/registri**
- la gestione logica e l'overhead aumentano con l'aumentare del numero di fasi della pipeline

**progettazione accurata** per ottenere  
**risultati** ottimali con una **complessità** ragionevole



## Pipeline – evoluzione ideale

Per aumentare le prestazioni bisogna

- decomporre il lavoro in un **maggior numero di fasi**
- cercare di rendere le fasi più **indipendenti** e con una **durata simile**

- **fetch** (FI) lettura dell'istruzione
- **decodifica** (DI) decodifica dell'istruzione

Fetch Data

- **esecuzione** (EI) esecuzione dell'istruzione
- **scrittura** (WO) scrittura del risultato in memoria



# Pipeline – evoluzione ideale

Per aumentare le prestazioni bisogna

- decomporre il lavoro in un **maggior numero di fasi**
- cercare di rendere le fasi più **indipendenti** e con una **durata simile**

Fetch Data	•	fetch	(FI)	lettura dell'istruzione
	•	decodifica	(DI)	decodifica dell'istruzione
	•	calcolo ind. op. (CO)		calcolo indirizzo effettivo operandi
	•	fetch operandi (FO)		lettura degli operandi in memoria
	•	esecuzione	(EI)	esecuzione dell'istruzione
	•	scrittura	(WO)	scrittura del risultato in memoria

# Pipeline – evoluzione ideale

