Evoluzione delle architetture
Evoluzione strutturale

- L'evoluzione tecnologica corrente ha **limiti fisici** di progresso. Miglioramenti importanti necessitano di cambiamenti architetturali radicali
  - Quantum computing?
- L'aumento di prestazioni dipende sempre più da **organizzazioni** architetturali più **efficienti**
  - Individuazione e soluzione di colli di bottiglia (*bottleneck*)
  - Aumento del parallelismo interno

Evoluzione delle architetture
Evoluzione strutturale

- **Parallelismo**
  - Se un lavoro non può essere svolto più velocemente da una sola persona (unità), allora conviene **decomporlo** in parti che possano essere eseguite da più persone (unità) **contemporaneamente**
    - **Catena di montaggio**

Pipeline
Generalità 1

- Per svolgere un lavoro si devono eseguire tre fasi distinte e sequenziali

  L $\Rightarrow$ [*fase1*] [*fase2*] [*fase3*]
- Se ogni fase richiede un tempo **T**, un unico esecutore svolge un lavoro **L** ogni **3T**
- Per ridurre i tempi di produzione si possono utilizzare **più esecutori**

Pipeline
Generalità 2

- Soluzione (ideale) a parallelismo totale

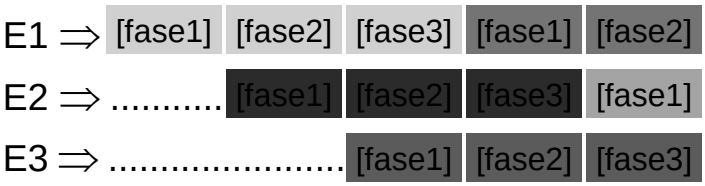
  E1 $\Rightarrow$ [fase1.**A**] [fase2.**A**] [fase3.**A**] | [fase1.**D**] …

  E2 $\Rightarrow$ [fase1.**B**] [fase2.**B**] [fase3.**B**] | [fase1.**E**] …

  E3 $\Rightarrow$ [fase1.**C**] [fase2.**C**] [fase3.**C**] | [fase1.**F**] …
- **N** esecutori svolgono un lavoro ogni **3T/N**
- Il problema è **come** preservare la **dipendenza funzionale** nell'esecuzione (di fasi) dei 'lavori' **A**, **B**, **C**, **D**, **E**, **F**, …

Pipeline
Generalità 3

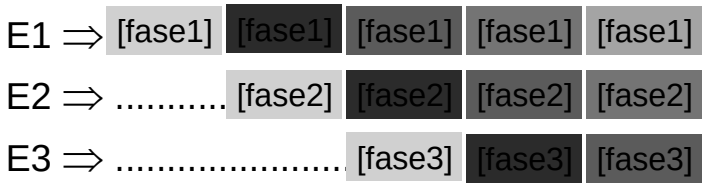
- Soluzione **pipeline** ad **esecutori generici**



- Ogni esecutore esegue un ciclo di lavoro **completo** (*sistema totalmente replicato*)
- A regime, **N** esecutori svolgono un lavoro ogni **3T/N** rispettandone la sequenza

Pipeline
Generalità 4

- Soluzione **pipeline** ad **esecutori specializzati**

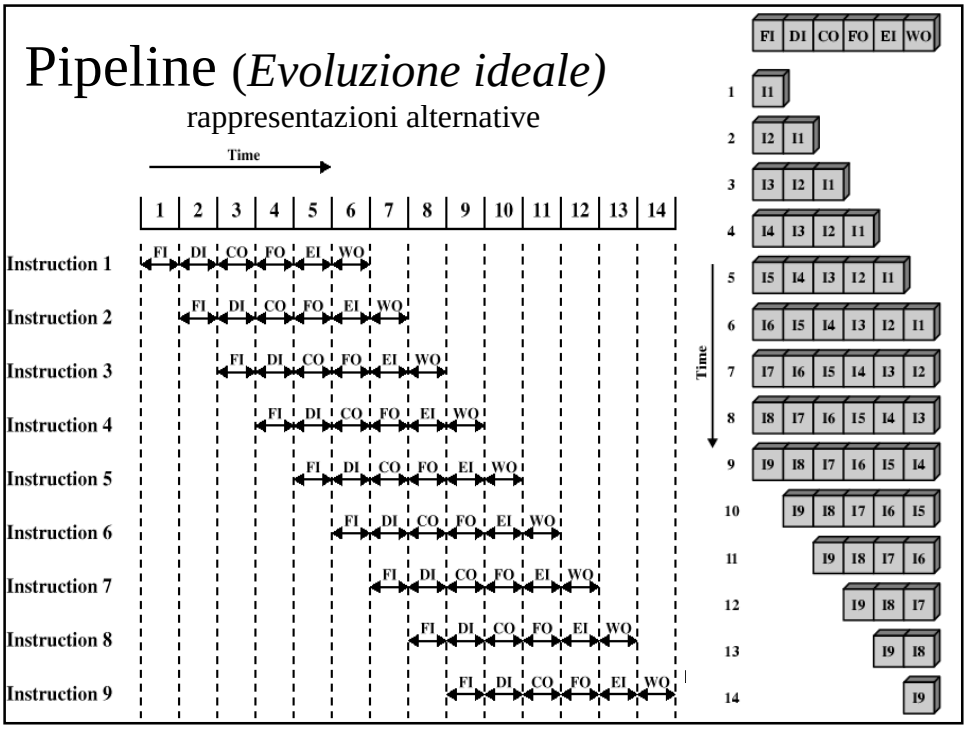


- Ogni esecutore svolge sempre e solo la **stessa** fase di lavoro
- Soluzione più efficace in termini di **uso di risorse** (**3T/N** lavori con **N/3** risorse)

Pipeline
Decomposizione in fasi

- L'esecuzione di una generica istruzione può essere suddivisa nelle seguenti fasi:
  - **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)*
rappresentazioni alternative



Pipeline
Problemi 1

- Vari fenomeni pregiudicano il raggiungimento del massimo di parallelismo teorico (**stallo**)
  - **Sbilanciamento delle fasi**
    - Durata diversa per fase e per istruzione
  - **Problemi strutturali**
    - La sovrapposizione totale di tutte le (fasi di) istruzioni causa conflitti di accesso a risorse limitate e condivise

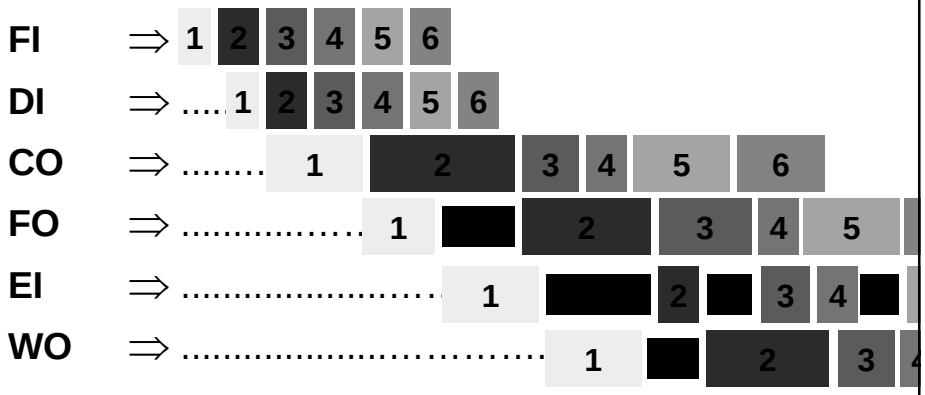
Pipeline
Problemi 2

- **Dipendenza dai dati**
  - L'operazione successiva dipende dai risultati dell'operazione precedente
- **Dipendenza dai controlli**
  - Istruzioni che causano una violazione di sequenzialità (p.es.: salti condizionali) invalidano il principio del *pipelining* sequenziale

Pipeline
Sbilanciamento delle fasi 1

- La suddivisione in fasi va fatta in base all'istruzione più onerosa
- Non tutte le istruzioni richiedono le stesse fasi e le stesse risorse
- Non tutte le fasi richiedono lo stesso tempo di esecuzione
  - P.es.: lettura di un operando tramite registro rispetto ad una mediante indirizzamento indiretto

Pipeline
Sbilanciamento delle fasi 2



Pipeline
Sbilanciamento delle fasi 3

Possibili soluzioni allo sbilanciamento:

- Decomporre fasi onerose in più sottofasi
  - Costo elevato e bassa utilizzazione
- Duplicare gli esecutori delle fasi più onerose e farli operare in parallelo
  - CPU moderne hanno una ALU in aritmetica intera ed una in aritmetica a virgola mobile

Pipeline
Problemi strutturali

Problemi

- Maggiori risorse interne (**severità bassa**): l'evoluzione tecnologica ha spesso permesso di duplicarle
- Colli di bottiglia (**severità alta**): l'accesso alle risorse esterne, p.es.: memoria, è molto costoso e molto frequente (anche 3 accessi per ciclo di clock)

Soluzioni

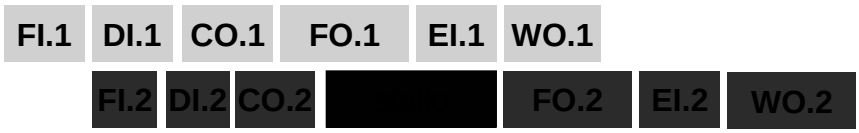
- Suddividere le memorie (accessi paralleli)
- Introdurre fasi non operative (***nop***)

Pipeline
Dipendenza dai dati 1

- Un dato modificato nella fase **EI** dell'istruzione corrente può dover essere utilizzato dalla fase **FO** dell'istruzione successiva

  INC [0123]

  CMP [0123], AL



Pipeline
Dipendenza dai dati 2

Soluzioni

- Introduzione di fasi non operative (***nop***)
- Individuazione del rischio e prelievo del dato direttamente all'uscita dell'ALU (**data forwarding**) [vedremo un esempio su MIPS]
- Risoluzione a livello di compilatore
- Riordino delle istruzioni (**pipeline scheduling**)

Pipeline
Dipendenza dai controlli

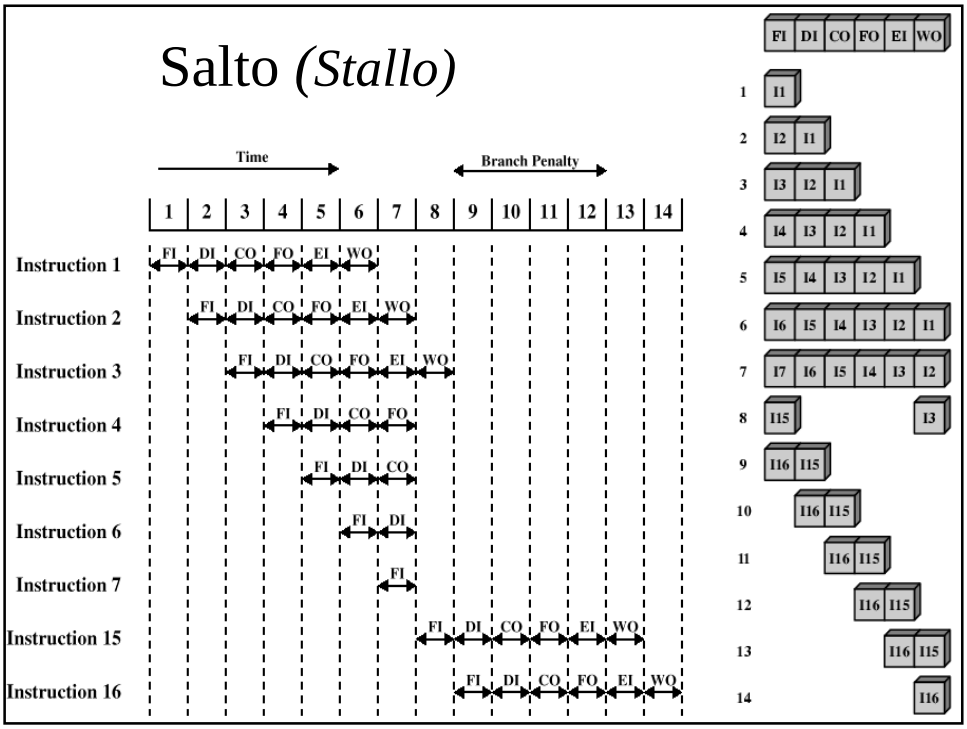
- Tutte le istruzioni che modificano l'IP (salti condizionati e non, chiamate a e ritorni da procedure, interruzioni) invalidano il pipeline
- La fase **fetch** successiva carica l'istruzione seguente, che può *non essere* quella giusta
- Tali istruzioni sono circa il 30% del totale medio di un programma

Pipeline
Dipendenza dai controlli

Soluzioni

- Mettere in **stallo** il pipeline fino a quando non si è calcolato l'indirizzo della prossima istruzione
  - Pessima efficienza, massima semplicità
- Individuare le istruzioni critiche per anticiparne l'esecuzione, eventualmente mediante apposita logica di controllo
  - Compilazione complessa, hardware specifico

Salto *(Stallo)*



Pipeline
Dipendenza dai controlli

Alcune soluzioni per salti condizionati

- flussi multipli (multiple streams)
- prelievo anticipato della destinazione (prefetch branch target)
- buffer circolare (loop buffer)
- predizione del salto (branch prediction)
- salto ritardato (delayed branch)

Pipeline
Dipendenza dai controlli

Flussi multipli: replicare le parti iniziali della pipeline, una che contenga l'istruzione successiva a quella corrente di salto (nel caso il salto non avvenga), e l'altra l'istruzione destinazione (*target*) del salto (nel caso in cui il salto avvenga)

Problemi di questa soluzione:

- conflitti nell'accesso alle risorse (registri, memoria, ALU,…) da parte delle 2 pipeline
- presenza di salti condizionali in sequenza che entrano nelle 2 pipeline prima che si sia risolta la condizione del primo salto condizionale (occorrerebbero 2 pipeline aggiuntive per ogni ulteriore salto condizionale…)

Pipeline
Dipendenza dai controlli

Prelievo anticipato della destinazione: quando si incontra un salto condizionato si effettua il fetch anticipato della istruzione di destinazione del salto in modo da trovarla già caricata nel caso in cui il salto debba avvenire.

Problemi di questa soluzione:

- non evita l'eventuale svuotamento della pipeline con conseguente perdita di prestazioni

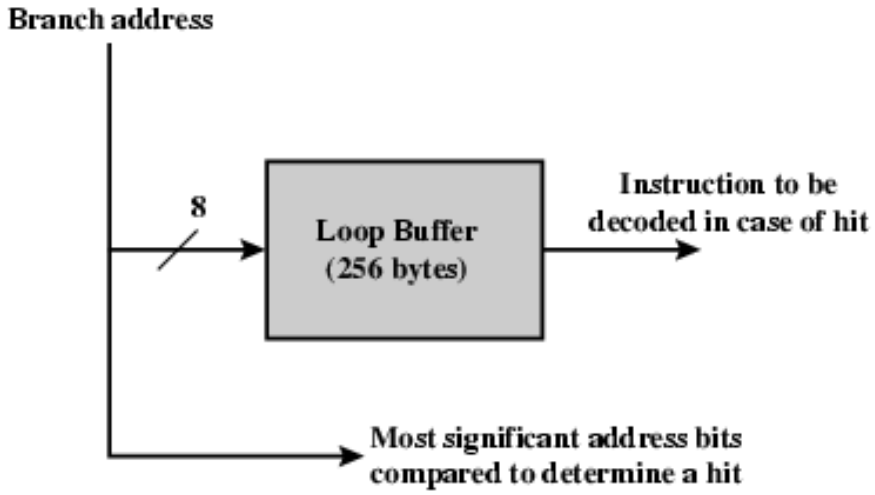
Pipeline
Dipendenza dai controlli

Buffer circolare: si utilizza una memoria piccola e molto veloce (il buffer circolare) dove mantenere le ultime *n* istruzioni prelevate. In caso di salto, si controlla se l'istruzione destinazione è già presente nel buffer, così da evitare il fetch della stessa.

Vantaggi:

- anticipando il fetch, alcune delle istruzioni successive a quella corrente saranno già presenti nel buffer e se non si ha salto non ci sarà bisogno di caricarle dalla memoria
- se si salta in avanti di poche istruzioni (vedi trattamento del costrutto IF-THEN-ELSE), l'istruzione destinazione sarà già presente nel buffer
- se il salto condizionale realizza un ciclo le cui istruzioni possono essere tutte contenute nel buffer, non c'è bisogno di effettuare fetch ripetuti delle stesse istruzioni

Pipeline
Buffer circolare



Pipeline
Dipendenza dai controlli

Predizione dei salti: si cerca di prevedere se il salto sarà intrapreso oppure no.

Varie possibilità:

- previsione di saltare sempre
- previsone di non saltare mai
- previsione in base al codice operativo

} *approcci statici*

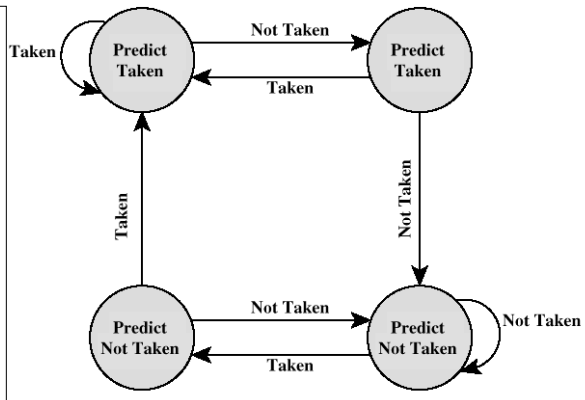
- bit *taken/not taken*
- tabella della storia dei salti

} *approcci dinamici*

Approcci dinamici di predizione: cercano di migliorare la qualità della predizione sul salto memorizzando la storia delle istruzioni di salto condizionato di uno specifico programma.

bit *taken/not taken* :

- ad ogni istruzione di salto condizionato si associano uno o più bit che codificano la storia recente.
- bit memorizzati non in memoria centrale ma in una locazione temporanea ad accesso molto veloce



esempio con 2 bit

Problemi di bit *taken/not taken* :

- quando si decide di saltare, bisogna aspettare la decodifica dell'indirizzo destinazione prima di poter prelevare l'istruzione destinazione
- si può anticipare il prelievo a patto di salvare opportune info nel *branch target buffer* o *branch history table*

tabella della storia dei salti:

- piccola memoria associata allo stadio fetch della pipeline
- ogni riga della tabella è costituita da 3 elementi:
  1. indirizzo istruzione salto,
  2. l'indirizzo destinazione del salto (o l'istruzione destinazione stessa),
  3. alcuni bit di storia che descrivono lo stato dell'uso dell'istruzione

Lezione Architettur