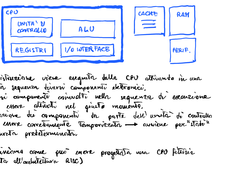
COME LA CPU ESEGUE LE ISTRUZIONI

Fino ad ora abbiamo assunto che ogni istruzione venga eseguita **istantaneamente** dalla CPU



Ogni istruzione viene eseguita dalla CPU attraverso in una catena sequenza diversi componenti elettronici.
Ciascun componente esaurisce nella sequenza di esecuzione una certa attività ed inoltre meccanismi di sincronizzazione dei componenti in punti dell'unità di controllo fanno come conseguenza temporizzato → avviene per stadi di questa predeterminata.

Immaginiamo come per mero progetto una CPU fittizia (ispirata all'architettura 8-bit)

1. **FETCH (F)**: la CPU preleva dalla memoria la prossima istruzione da eseguire (la cui posizione è data da un apposito registro)
2. **DECODE (D)**: viene interpretata l'istruzione (o i valori in cui sono) legge gli operandi immediati, legge gli eventuali operandi presenti in registri
3. **EXECUTE (E)**: se l'istruzione lo richiede, l'ALU calcola il risultato
4. **MEMORY (M)**: se l'istruzione lo richiede, la memoria viene accedute in lettura o scrittura
5. **WRITE-BACK (W)**: se l'istruzione lo richiede, i risultati di output vengono aggiornati

Esempi:

`movl $8, %eax` — istruzione caricata in memoria
F
D — (nessun operando in memoria)
E — (nessun calcolo)
M — (nessun accesso)
W — viene scritto %eax

`addl %ebx, %eax` — istruzione caricata in memoria
F
D — legge %ebx, %eax
E — esegue somma in ALU
M — (nessun accesso)
W — torna in %eax

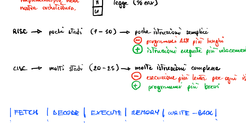
`addl $50, (%eax)` — istruzione caricata in memoria
F
D — legge $50, legge %eax
E — esegue somma
M — legge (%eax)
W — (nessun accesso)

Istruzione non implementabile nella nostra architettura

RISC → pochi stadi (4 - 10) → poche istruzioni semplici
- ⊕ programmi più lunghi
- ⊕ istruzioni eseguite più velocemente

CISC → molti stadi (20 - 25) → molte istruzioni complesse
- ⊖ esecuzione più lenta per ogni istruzione
- ⊖ programmazione più breve

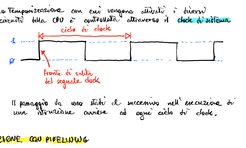
La temporizzazione con cui vengono attivati i diversi circuiti della CPU è controllata attraverso il **clock di sistema**

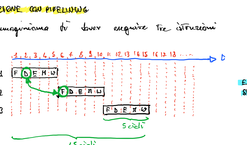


Il passaggio da uno stadi di successivo all'esecuzione di una istruzione avviene ad ogni ciclo di clock.

ESECUZIONE CON PIPELINING

Immaginiamo di dover eseguire tre istruzioni



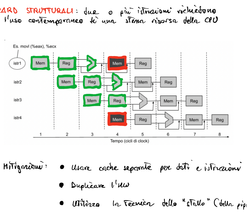


$\Rightarrow$ AUMENTO IL THROUGHPUT COMPLESSIVO DELLA SISTEMA

ATTENZIONE: pipeline "piena" in realtà i suoi stadi significativi massimo throughput.

Non sempre è possibile mantenere la pipeline piena a causa della presenza di **hazard**

HAZARD STRUTTURALI: due o più istruzioni richiedono l'uso contemporaneo di una stessa risorsa della CPU



Mitigazioni:
- Usare cache separate per dati e istruzioni
- Duplicare l'ALU
- Utilizzo la tecnica della "stalla" (della pipeline)

HAZARD SUI DATI: si verificano quando esistono dipendenze tra dati in cui una istruzione istruzione diverse

Es.:
1) `movl $2, %eax`
2) `movl %eax, %ebx`
3) `|`

Soluzione:

TECNICA DELLO STALLO: la CPU ritarda la pipeline in stalla (in una nuova istruzione) per il numero minimo di cicli di clock necessari a risolvere l'hazard

L'introduzione di bolle per mettere in stallo la pipeline risolve SEMPRE qualsiasi hazard.
ATTENZIONE: le stalle riducono il throughput

RIORDINAMENTO DEL CODICE: consiste l'ordine di esecuzione delle istruzioni per evitare l'hazard

Es.:
```
movl $1, %eax
movl %eax, %ebx
movl $2, %ecx
movl $3, %edi
movl $4, %esi
movl %ecx, %edx
```

HAZARD SUL CONTROLLO DI FLUSSO: l'esecuzione di una istruzione dipende dalla valutazione di una precedente condizione.

Es.:
```
    testl %eax, %eax
    je L1
    movl %eax, %edx
L1: ...
```

Mitigazioni:

BRANCH PREDICTION: la CPU cerca di "predire" quale sarà la prossima istruzione da eseguire prima che la relativa condizione venga completamente valutata.

USO DI CODICE ALTERNATIVO: costruzione sequenze di istruzioni che non generano hazard sul controllo

Es.
```
    testl %eax, %eax
    cmovel %ecx, %edx
```

INTERRUZIONI

Un "interrupt" è una notifica generata dalla CPU in HW o attraverso apposite istruzioni SW.
La notifica rappresenta un evento (tipicamente asincrono) che richiede l'attenzione del sistema.

In caso di interrupt, l'esecuzione del processo attualmente viene sospesa, il processo passa in READY (tipicamente) e al suo posto si lancia una routine di gestione dell'evento detta **INTERRUPT HANDLER**.

Eventi diversi sono rappresentati da interrupt diversi associato con il proprio handler.
In memoria, tra interrupt e handler è gestita dal SO attraverso la "**interrupt vector table**" (IVT)

INTERRUPT

ASINCRONI

Interruzioni HW generate attraverso apposite linee di segnale elettrico. Sono tipicamente generate da periferiche.

SINCRONI

TRAP: interruzioni generate dal programma. In x86-32 le trap possono essere generate con l'istruzione `int N` dove N è l'id della interruzione che si vuole generare.
Dopo l'esecuzione dell'handler il processo interrotto riprende dall'istruzione successiva.

ABORT: interrupt normalmente non intenzionali e non recuperabili

FAULT: generati dalla CPU al verificarsi di errori gravi. Sono spesso recuperabili. Dopo l'esecuzione dell'handler il processo interrotto riprende dall'istruzione che ha generato il fault.