Lezione 22

Gestione delle eccezioni (CPU multiciclo)
La CPU pipeline

A. Borghese, F. Pedersini

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano

Eccezioni ed Interrupt

- Eccezioni: eventi che alterano l'esecuzione di un programma
  - **Eccezioni** – Generate internamente al processore
    (e.g. overflow)
  - **Interrupts** – Generati esternamente al processore
    (e.g. richiesta di attenzione da parte di una periferica).

Come si comporta la CPU a fronte di eccezioni?

- **In modo funzionalmente equivalente ad una `jump`**

Tipo di evento	Provenienza	Terminologia MIPS
Richiesta di un dispositivo di I/O	Esterna	*Interrupt*
Uso di un'istruzione non definita	Interna	*Eccezione*
Overflow aritmetico	Interna	*Eccezione*
Chiamata al S.O. da parte di un programma (es. `syscall`)	Interna	*Eccezione*
Malfunzionamento hardware	Entrambe	*Eccezione / Interrupt*

Azioni in risposta ad un'eccezione

Come reagisce la CPU ad un'eccezione?

1. Salvataggio dell'indirizzo dell'istruzione coinvolta

   indirizzo istruzione corrente: ($PC – 4)

2. Trasferimento del controllo al Sistema Operativo: **`jump`**

   il quale deve gestire la condizione di eccezione.

- Operazioni da svolgere:
  1. *Salvataggio ambiente,*
  2. *Riconoscimento e gestione eccezione,*
  3. *Ripristino ambiente*
  - Gestione software – operazione "delicata".
    Per questo le eccezioni vengono gestite dal **Sistema Operativo**

Hardware addizionale:

- Registro **EPC** (Exception Program Counter)
  - registro a 32 bit utilizzato per memorizzare l'indirizzo dell'istruzione coinvolta
- Registro **Causa**
  - registro utilizzato per memorizzare la causa dell'eccezione; (MIPS: 32 bit)

    $bit_0 = 0$ → Causa: Istruzione indefinita (*Illegal OpCode*)

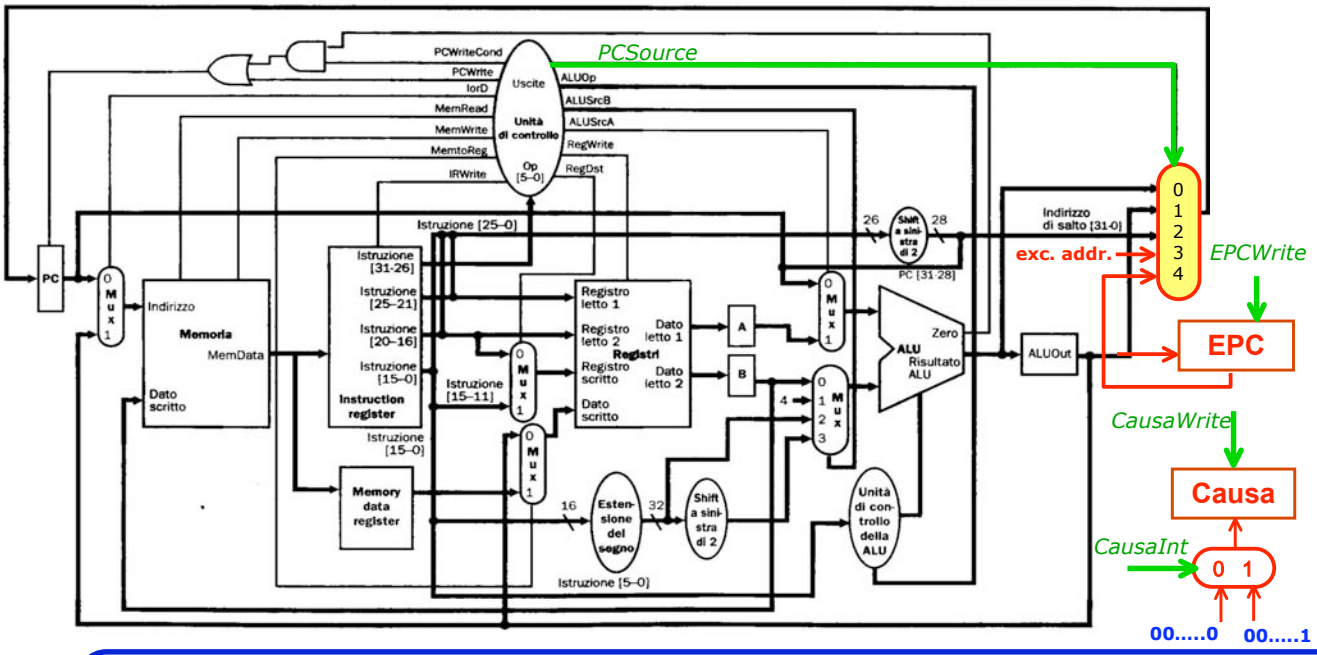
    $bit_0 = 1$ → Causa: Overflow aritmetico

- Segnali di controllo aggiuntivi
  - ***EPCWrite*** – scrittura nel registro EPC.
  - ***CausaWrite*** – scrittura nel registro Causa.
  - ***CausaInt*** – dato per il registro Causa.

- Ingressi aggiuntivi alla UC (FSM)
  - Aggiunta di **Overflow** (proveniente dalla ALU)

HW per risposta ad eccezioni

- Salvataggio del PC:
  - EPC ← PC – 4;
  - Selezione della causa (0...0 , 0...1)
  - PC ← exception address;

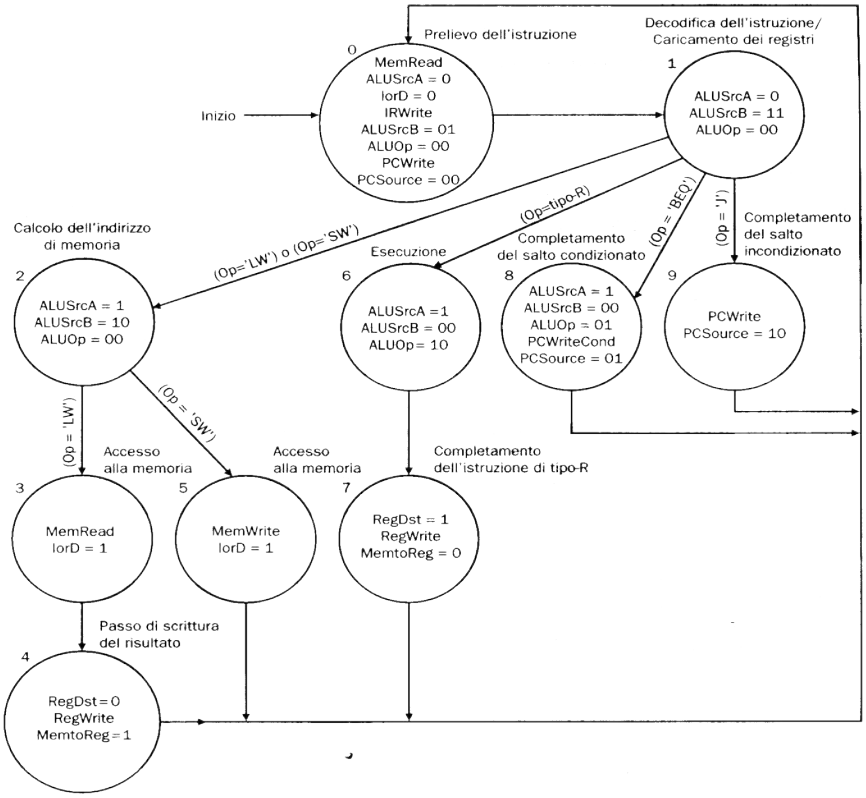


Gestione eccezioni: esempi

Esempio: gestione di 2 tipi di eccezione:

1. Istruzione indefinita:
   - Non esiste, al passo 2 (stato 1 – Decodifica), uno stato futuro valido.
   - Nuovo stato futuro denominato: "**Invalid OpCode**"
     - Tale stato viene raggiunto al verificarsi dell'eccezione.
2. Overflow aritmetico:
   - Al passo 4 di esecuzione dell'operazione (**ALU** ha già eseguito l'operazione – manca la fase di Write-back nel RF) lo stato futuro è scelto in funzione del segnale di Overflow
   - Overflow → input aggiuntivo alla FSM della UC
   - Necessità di uno stato aggiuntivo: "**Overflow**"

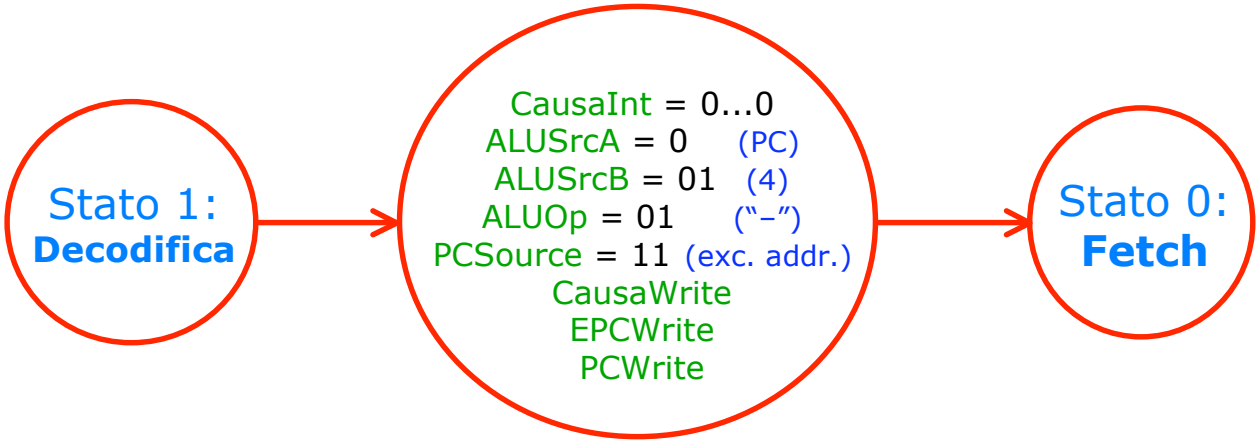
FSM di CPU Multi-ciclo: – STG



Stato 10: *Invalid OpCode*

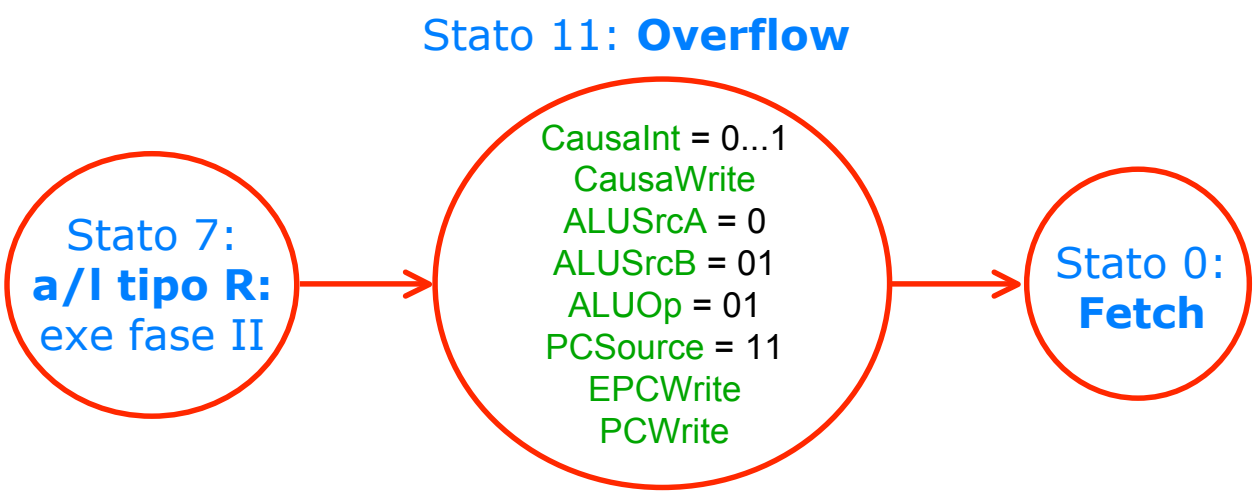
- Nuovo stato nella FSM dell'unità di controllo:
  - **Stato: "Invalid OpCode" – stato 10**

Stato 10: **invalid OpCode**

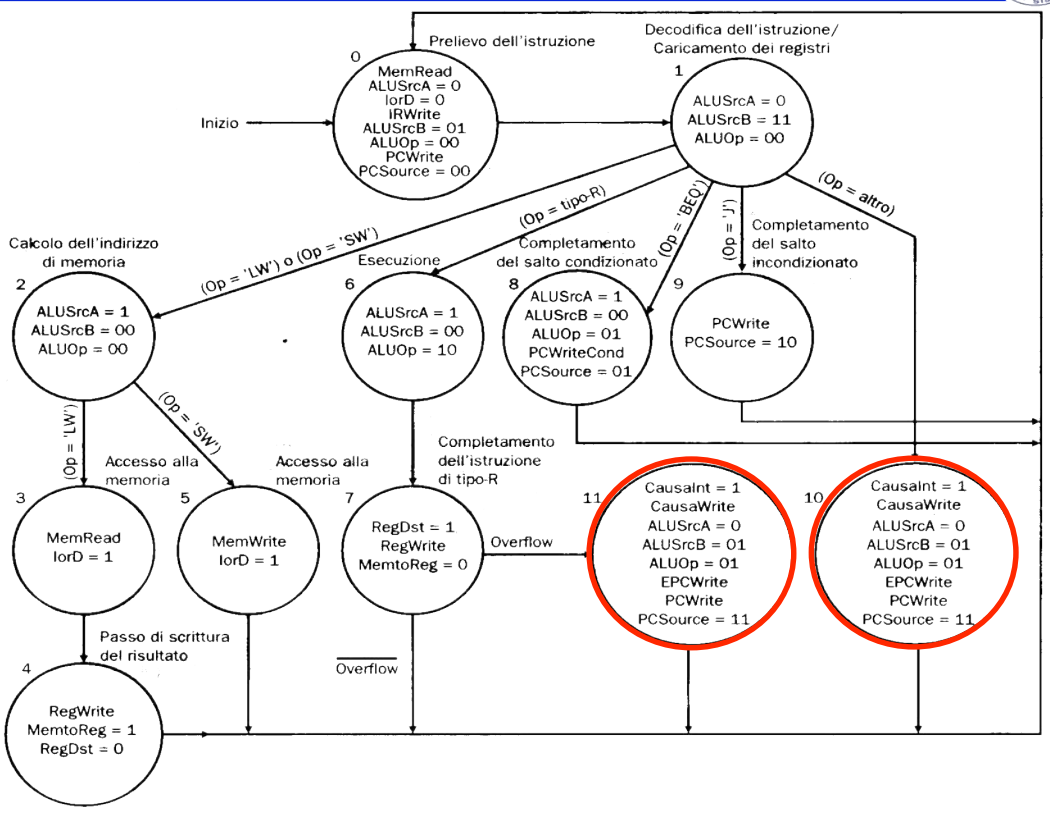


Stato 11: Overflow

- Nuovo stato nella FSM dell'unità di controllo:
  - Stato: **"Overflow" – stato 11**
  - raggiungibile solo per istruzioni tipo R, dopo l'esecuzione del calcolo (Esecuzione – fase II)



CPU multi-ciclo con gestione eccezioni: FSM



Risposta ad un'eccezione

Eccezione → Risposta di tipo **software**

Esistono diverse tecniche di reazione ad un'eccezione:

- **Vettorializzata:**
  - Ogni eccezione rimanda ad un indirizzo diverso del SO
  - Dall'indirizzo si può ricavare la causa dell'eccezione
  - Gli indirizzi sono in genere equispaziati (es. ogni 8 parole)
- Tramite **registro** (MIPS: *registro **causa***):
  - Il SO ha un unico *entry point* per la gestione delle eccezioni.
  - La prima istruzione è di decodifica della causa dell'eccezione
  - L' *entry point* è forzato tramite: `PCSource = 11`

MIPS: Co-processore 0

Coprocessore 0 – presente nelle architetture MIPS

- Raccolta e gestione delle informazioni e gli indirizzi per la gestione delle eccezioni
- Registri:

Registro	N. registro C0	Utilizzo / Contenuto
Bad/Addr	**8**	Indirizzo di memoria a cui si è fatto riferimento
Stato	**12**	Maschera delle interruzioni e bit di abilitazione
Causa	**13**	Tipo di interruzione / interruzione pendente
EPC	**14**	Indirizzo istruzione che ha causato l'interruzione

- Istruzioni che interessano il coprocessore 0 (`c0`):
  - Lettura/scrittura da **memoria** a **registri di `c0`**

    `lwc0 $<reg_c0> <offset>($reg)`

    `swc0 $<reg_c0> <offset>($reg)`
  - Move from/to, tra **registri** general-purpose e **registri di `c0`**

    `mfc0 $<reg>, $<reg_c0>` (pseudo-istruzione)

    `mtc0 $<reg_c0>, $<reg>` (pseudo-istruzione)

Sommario

- La gestione delle eccezioni in una CPU multi-ciclo
- Introduzione sulla Pipe-line

Principio intuitivo della pipe-line

- Anna, Bruno, Carla e Dario devono fare il bucato.

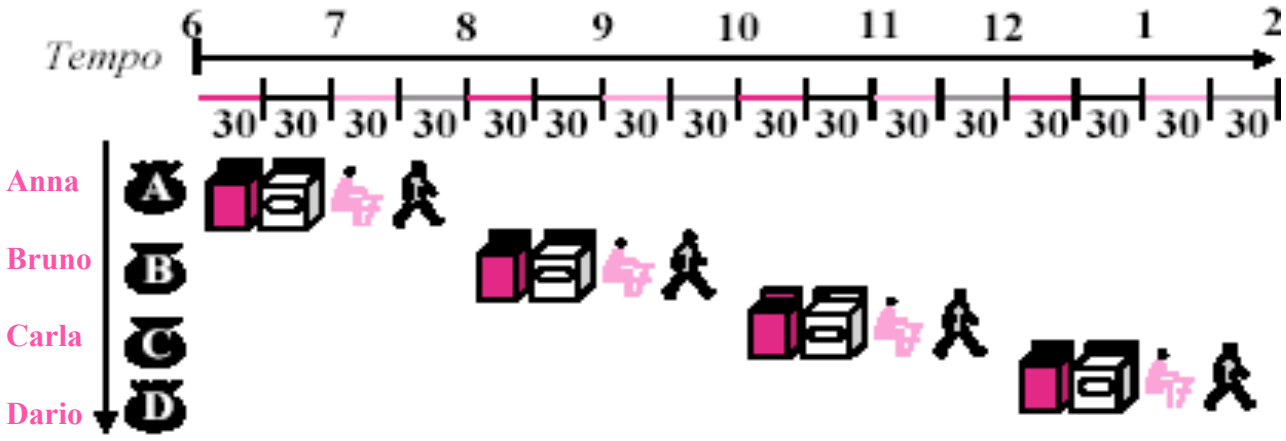
Ciascuno deve:

1. lavare,
2. asciugare,
3. stirare
4. mettere via

un carico di biancheria

- La lavatrice richiede: 30 minuti
- L'asciugatrice: 30 minuti
- Stirare richiede: 30 minuti
- Piegare/mettere via: 30 minuti

La lavanderia **sequenziale**



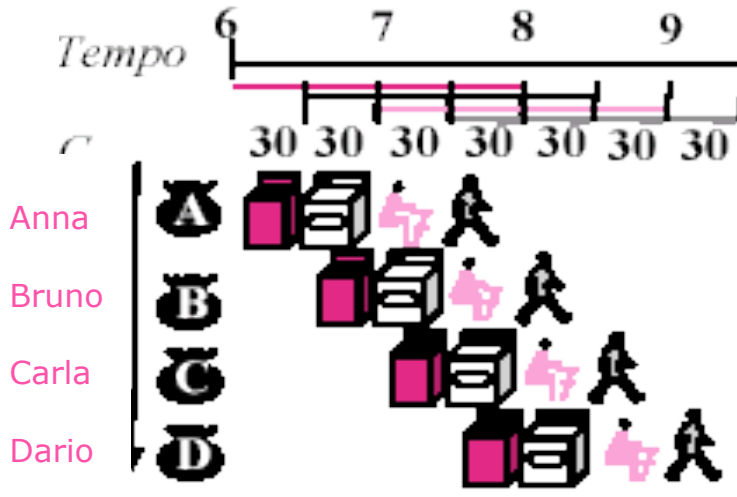
- le singole operazioni vengono svolte una alla volta

  T_totale = T_CicloCompleto * N_Utenti =

  = (T_SingolaFase * N_Fasi) * N_Utenti

- Tempo totale richiesto: **8 ore**

La lavanderia **pipe-line**



- le singole operazioni vengono svolte **in parallelo**
  - **T_totale = T_SingolaFase * (N_Fasi + N_Utenti – 1)**

- Tempo totale richiesto: **3,5 ore**

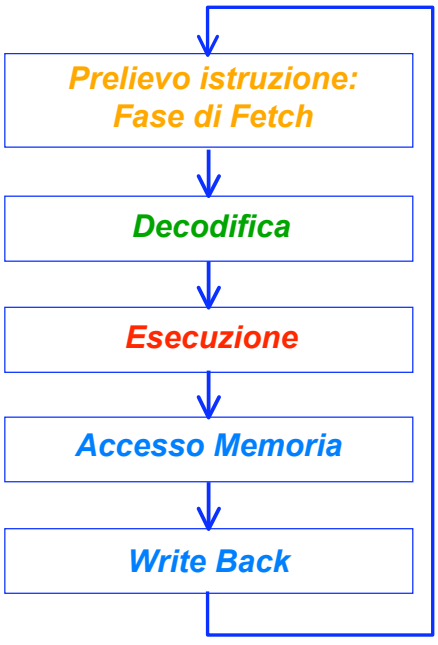
Osservazioni sulla pipe-line

- Il tempo di ciascuna operazione elementare non viene ridotto.
- Gli stadi della pipe-line lavorano in contemporanea perché utilizzano unità funzionali differenti.
- Le unità funzionali lavorano sequenzialmente (in passi successivi) su istruzioni successive

  ➔ aumenta il ***"Throughput"***
  (throughput = quantità di lavoro / tempo)

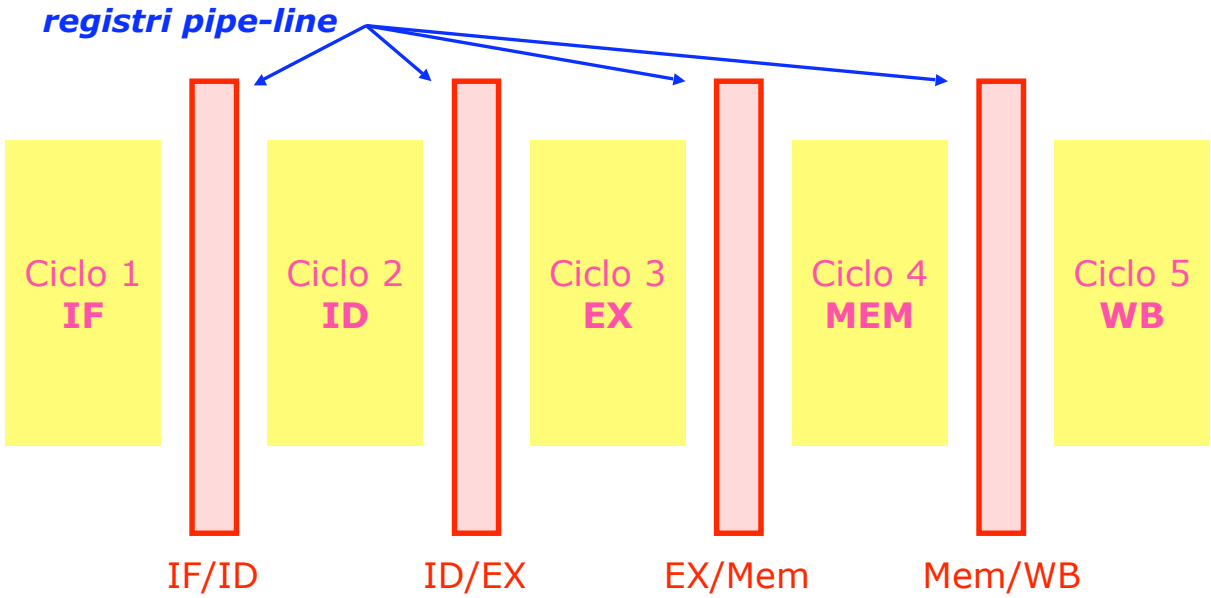
Ciclo di esecuzione: istruzioni MIPS

- MIPS è un'architettura pipeline
  - le istruzioni richiedono 5 passi
- **MIPS:** Pipeline a **5 stadi:**
  1. **IF**: Prelievo istruzione (Instruction Fetch)
  2. **ID**: Decodifica istruzione (+ lettura RF)
  3. **EX**: Esecuzione
  4. **MEM**: Accesso a memoria (Read/Write)
  5. **WB**: Scrittura del register file



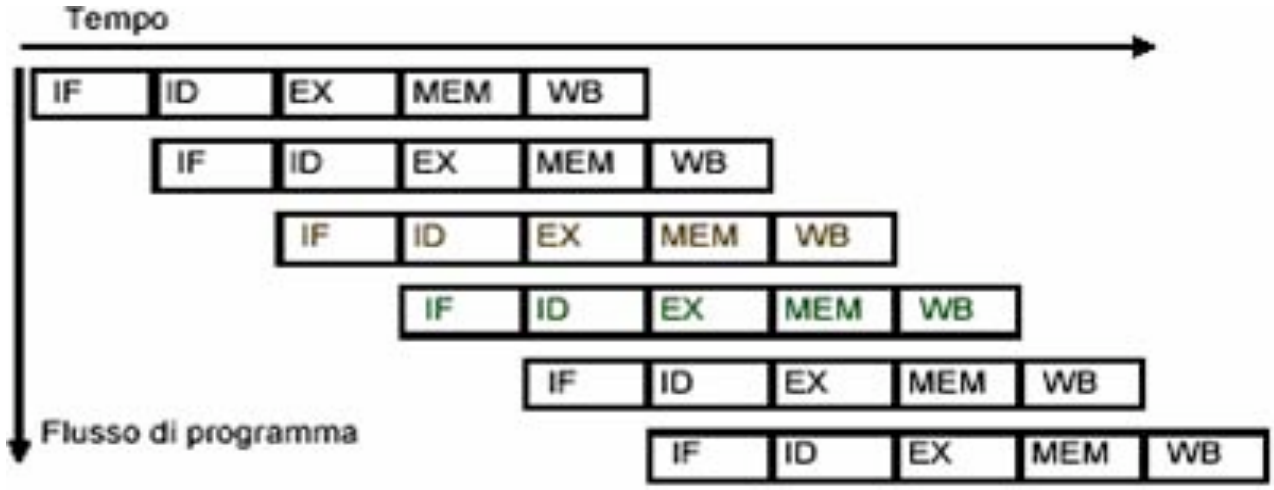
I 5 stadi della pipeline

- Tra cicli contigui sono posti i **registri di pipe-line**
  - Uno stadio inizia il suo lavoro quando il clock va basso e trasferisce in quello stadio l'elaborazione effettuata dallo stadio precedente



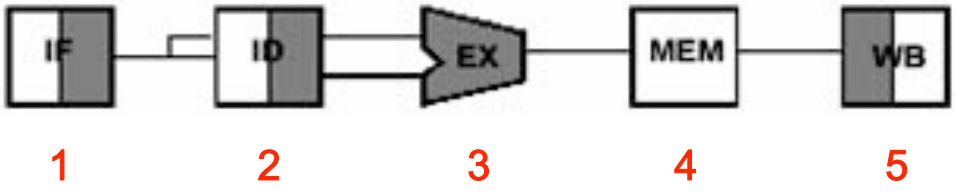
Rappresentazione della pipeline

- Rappresentazione convenzionale:
  - Ascisse (X): tempo
  - Ordinate (Y): Flusso di programma



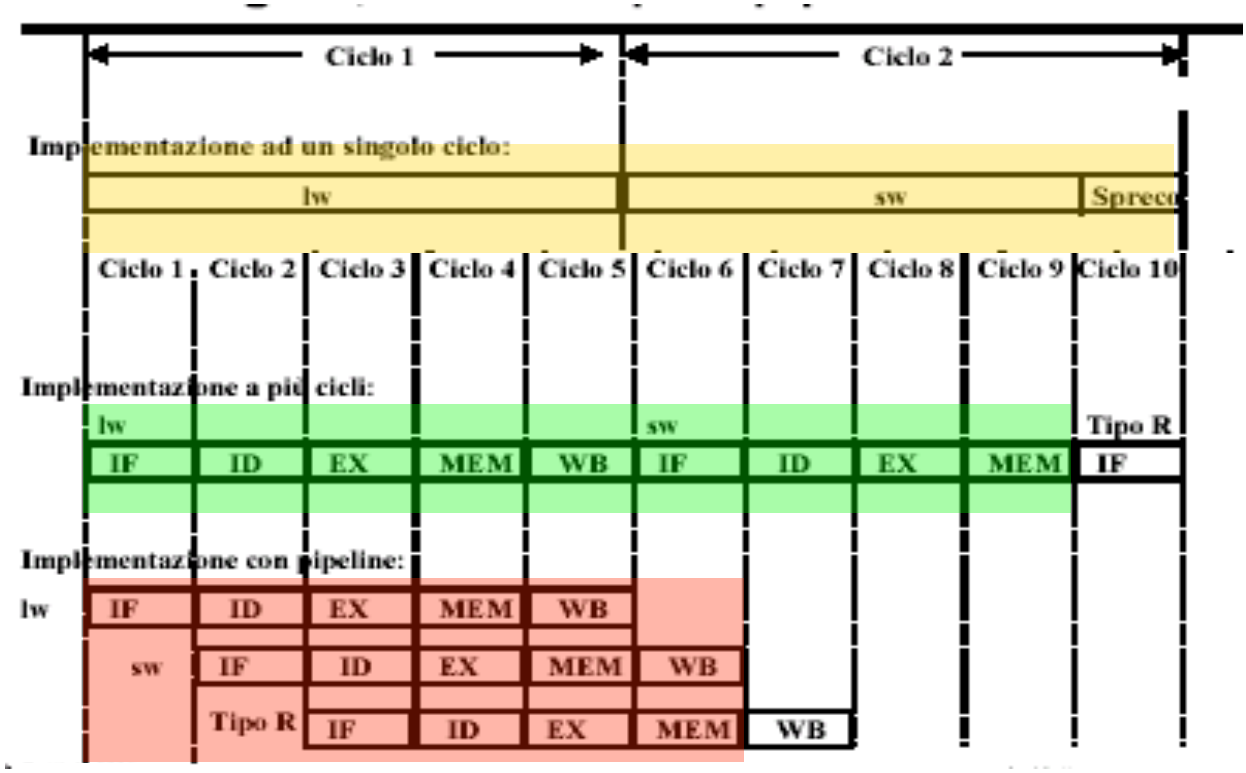
Rappresentazione grafica della pipeline

Esempio: `add $s0, $t0, $t1`



- Rappresentazione:
  - I rettangoli grigi a destra indicano lettura, a sinistra indicano scrittura.
  - I componenti bianchi, indicano il loro non utilizzo

I vantaggi della pipeline



Miglioramento delle prestazioni

- Il miglioramento massimo è una riduzione del tempo di un fattore pari al numero di stadi della pipe-line
  - Nell'esempio precedente (2 istruzioni **lw**, 2ns ad istruzione), il tempo richiesto con la pipe-line è di **12 ns** contro i **20 ns** senza pipe-line.
  - Il miglioramento teorico prevedrebbe **4ns**
- Il *Throughput* migliora comunque!
  - Miglioramento relativo al lavoro globale (con pipe-line senza stalli)

Utilizzo unità funzionali della CPU

- Quali unità funzionali (**ALU**, **RF**, **Mem**) vengono utilizzate in ciascuna fase ?

  Criticità strutturale: Se istruzioni diverse (in fasi diverse) necessitano contemporaneamente di una stessa risorsa, la risorsa va duplicata, oppure... si aspetta!

Istruzioni → passo ↓	Istruzioni tipo R	Istruzioni di accesso a memoria	Salto condizionato	Salto non condizionato
1. Fetch	IR = Memory[PC] ; PC = PC + 4			
2. Decodifica	A = Reg[IR[25-21]] ; B = Reg[IR[20-16]] ALUout = PC + (sign_ext(IR[15-0]) << 2)			
3. Esecuzione	ALUOut = A oper B	ALUOut = A + sign_ext(IR[15-0])	If (A==B) then PC = ALUOut	PC = PC[31-28] || IR[25-0]<<2
4. Mem: acc. R: WB	Reg([IR[15-11]] = ALUOut	**lw**: MDR=Mem[ALUOut] **sw**: Mem[ALUOut] = B		
5. Mem: WB		**lw**: Reg[IR[20-16]] = MDR		

Utilizzo unità funzionali della CPU

Φ	Passo esecuzione	ALU	Memoria	Register File
1	Fase fetch	Yes (PC+4)	Yes	No
2	Decodifica	Yes (salto)	No	Yes
3	Exec I – beq	Yes (test)	No	No
	Exec I – jump	No	No	No
	Exec I – tipo R	Yes (salto)	No	No
	Exec I – sw	Yes (indirizzo)	No	No
	Exec I – lw	Yes (indirizzo)	No	No
4	Exec II – tipo R	Yes (op)	No	Yes
	Exec II – sw	No	Yes	No
	Exec II – lw	No	Yes	No
5	Exec III – lw	No	No	Yes

Pipeline per l'istruzione: `lw`

- Limitandoci a considerare solo istruzioni `lw`:

Passo esecuzione	ALU	Memoria	Register File
IF (Fase fetch)	Yes	Yes	No
ID (Decodifica)	No	No	Yes
EX (Esecuzione)	Yes	No	No
MEM (Accesso memoria)	No	Yes	No
WB (riscrittura)	No	No	Yes

Tempo → / ↓ Istruzioni	t_0	t_1	t_2	t_3	t_4	t_5
...						
`lw $t0, 8($t2)`	Mem, ALU	RF	ALU	Mem	RF	
`lw $t1, 12($t2)`		Mem, ALU	RF	ALU	Mem	RF
...						

Criticità **(hazards)**

- 3 tipi di Criticità – Hazards
- **Strutturali**
  - Dovrei utilizzare la stessa unità funzionale due volte nello stesso passo.
  - Le unità funzionali non sono in grado di supportare le istruzioni (nelle diverse fasi) che devono essere eseguite in un determinato ciclo di clock.
- **di Dato**
  - Dovrei eseguire un'istruzione in cui uno dei dati è il risultato dell'esecuzione di un'istruzione precedente.
  - `lw $t0, 16($s1)`
  - `add $t1, $t0, $s0`
- **di Controllo/salto**
  - Dovrei prendere una decisione (sull'istruzione successiva) prima che l'esecuzione dell'istruzione sia terminata (e.g. **`branch`**) → generazione di situazioni di **hazard**

Stallo della pipeline

- Stallo:
  - In alcuni istanti di clock non può essere eseguita l'istruzione successiva
  - → La pipeline va in stallo
  - Detti **buchi** o **bolle** (o ***bubbles***)

Tempo → ↓ Istruzioni	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
…..								
`lw $t0, 8($s0)`	FF (Mem, ALU)	DEC (RF)	EXEC (ALU)	MEM (Mem)	WB (RF)	`$t0` disponibile		
`add $t1, $t0, $s0`		Buco (FF)	Buco (DEC)	Buco (EXEC)	Buco (MEM)	Buco (WB)		
`add $t1, $t0, $s0`			Buco	Buco	Buco	Buco	Buco	
`add $t1, $t0, $s0`	devo aspettare `$t0` !			Buco	Buco	Buco	Buco	Buco
`add $t1, $t0, $s0`					FF	DEC	EXEC	MEM

Considerazioni

- La tecnica di progettazione mediante **pipeline** sfrutta il **parallelismo** tra **fasi diverse** di **istruzioni consecutive**.
- Non aumenta la velocità di esecuzione della singola istruzione, ma del lavoro nel suo complesso (**throughput**)
  - Aumento di prestazioni massimo = n. di stadi di pipeline
- L'impossibilità di iniziare ad eseguire una fase di istruzione determina uno stallo della pipeline
- Criticità (hazards) in una pipeline sono suddivisibili in:
  - strutturali,
  - di controllo,
  - sui dati.
- La soluzione a tutte le criticità potrebbe essere … aspettare!