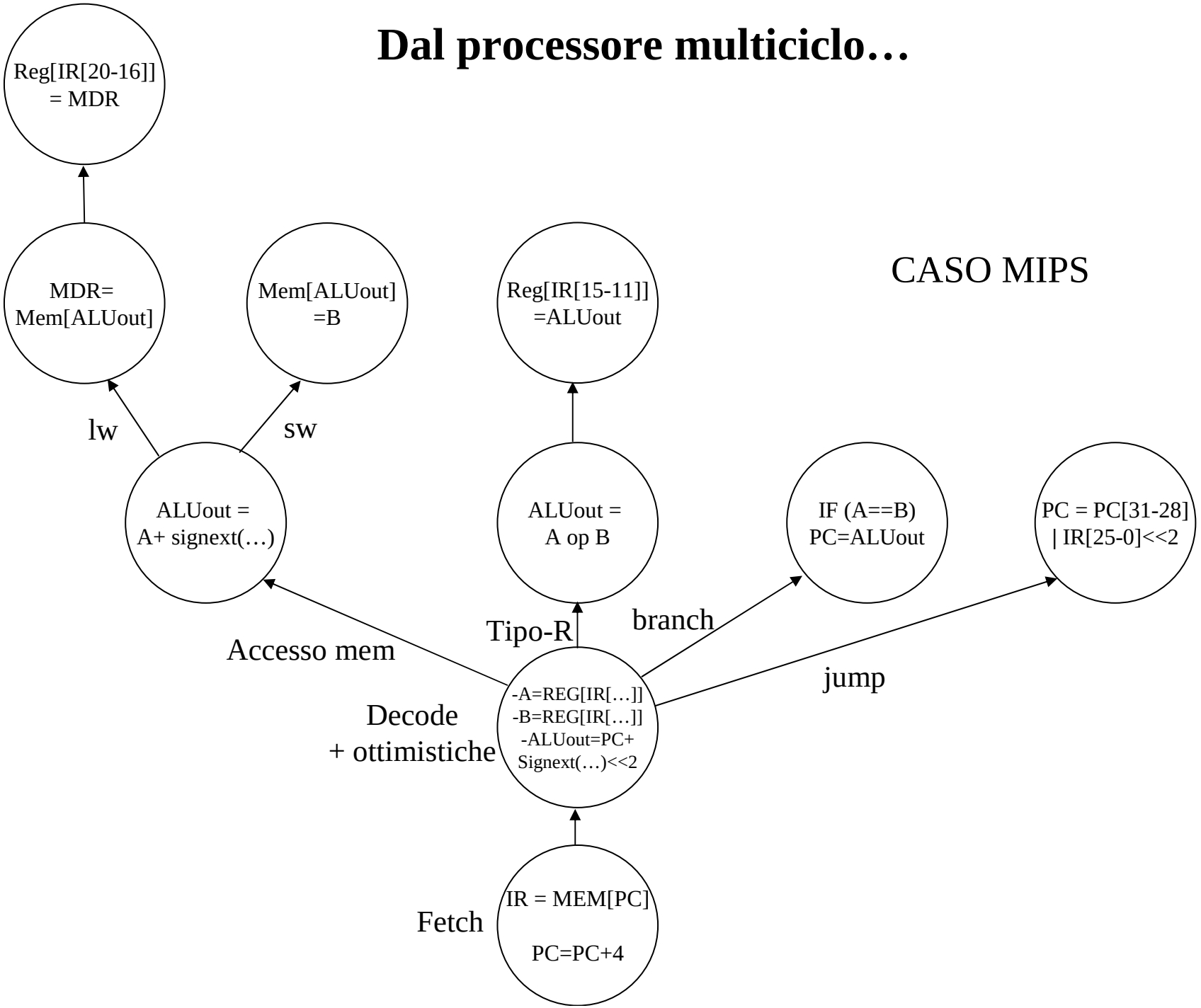
Calcolatori Elettronici B
a.a. 2007/2008

Tecniche Pipeline: Elementi di base

Massimiliano Giacomin

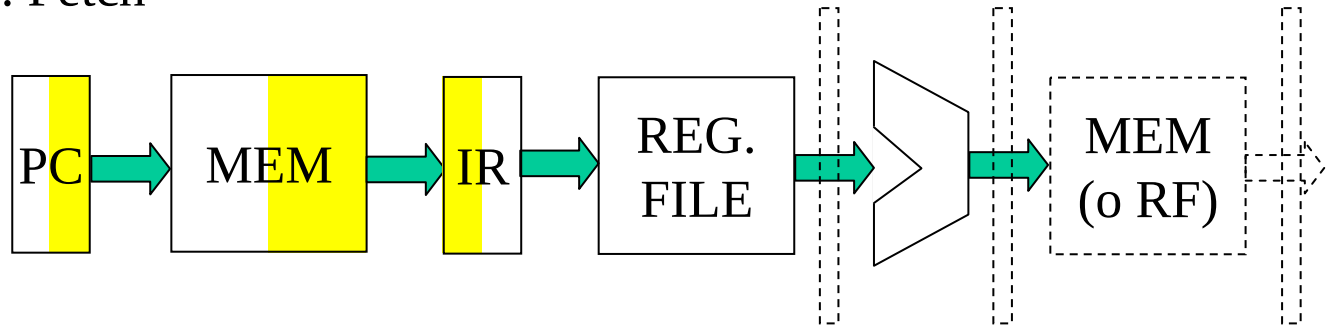
Dal processore multiciclo…



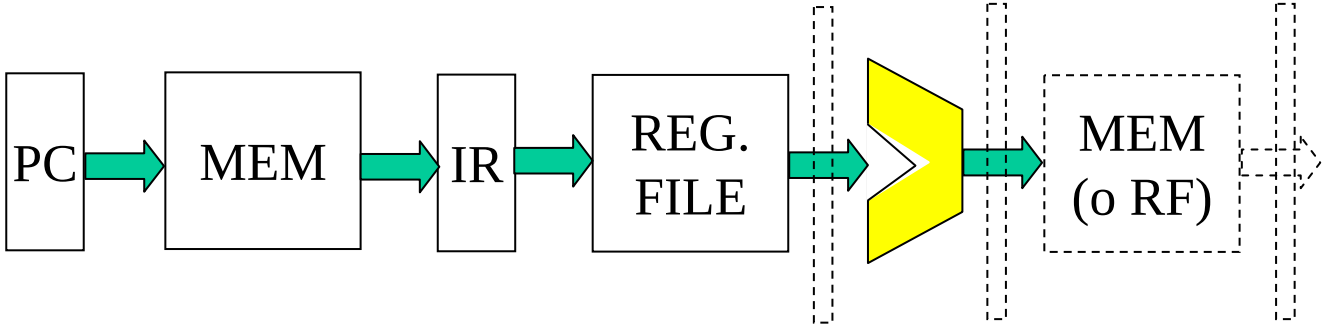
…alla pipeline

• Ogni stato richiede in genere solo una parte distinta dell'unità di elaborazione (il tratteggio indica eventuali registri temporanei)

P.es. Fetch



P.es. esecuzione operazione in ALU in istruzione di TIPO-R



• IDEA: come in una catena di montaggio, cominciare l'esecuzione dell'istruzione successiva non appena si libera lo stadio di fetch, procedendo per stadi successivi

L'idea di base

Pipeline = "oleodotto" | "catena di montaggio"

- Tecnica di implementazione in cui più istruzioni sono sovrapposte durante l'esecuzione.
  Esempio "pratico": la costruzione di automobili in catena di montaggio.

- La CPU è organizzata in un certo numero di "stadi" sequenziali
  - ogni istruzione richiede tutti gli stadi per la sua esecuzione
  - non appena uno stadio si libera, viene impegnato per l'istruzione seguente

  $\Rightarrow$ sfruttamento del parallelismo (ogni stadio è impegnato) nell'esecuzione di un flusso sequenziale di istruzioni.

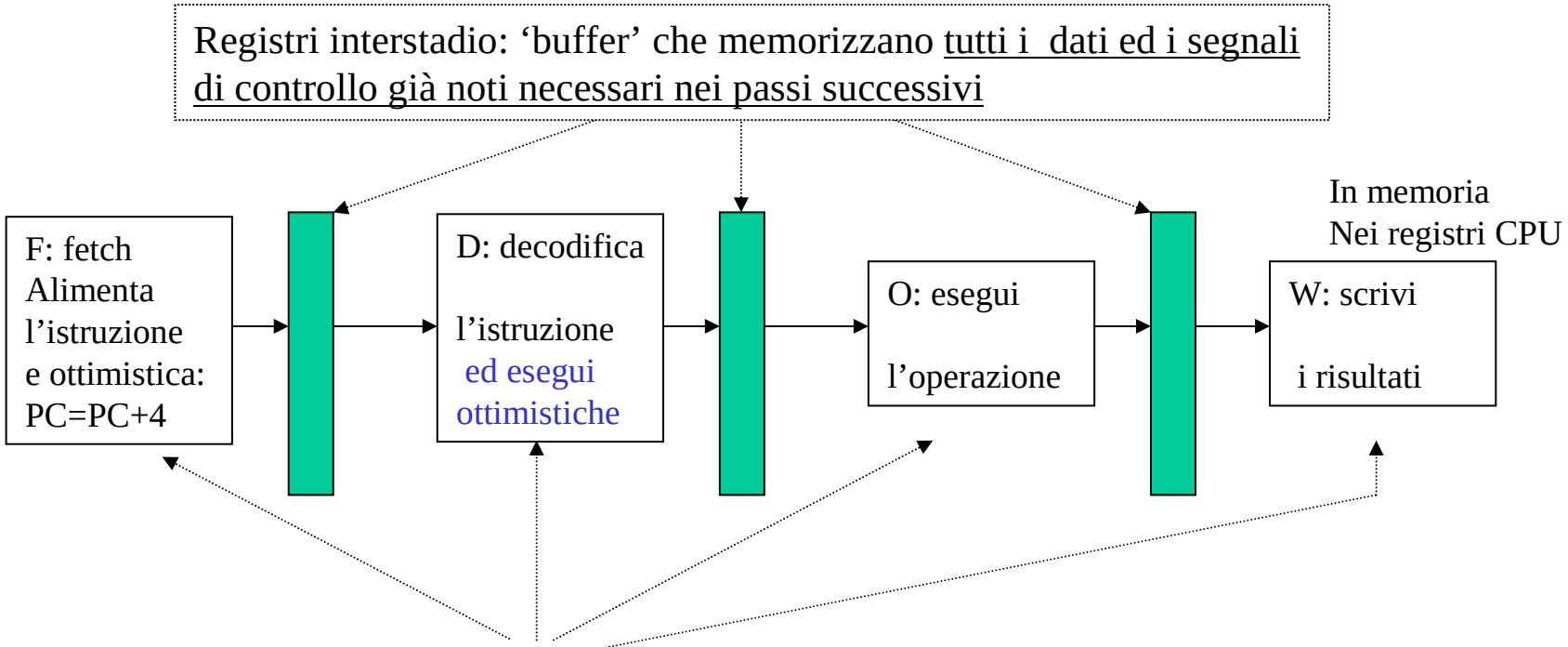
- Il tempo di esecuzione delle singole istruzioni non diminuisce, ma aumenta il "throughput" ["frequenza delle istruzioni"]

NB: seguiremo un approccio un po' diverso dal testo [concetti generali rispetto a cui l'implementazione del MIPS rappresenta una "istanza"]

PIPELINE:

ORGANIZZAZIONE E DATAPATH

Pipeline: l'organizzazione



L'esecuzione avviene in stadi (unità di esecuzione), che completano generalmente la propria attività in un ciclo di clock.

Ogni stadio è attivo in ogni ciclo di clock.

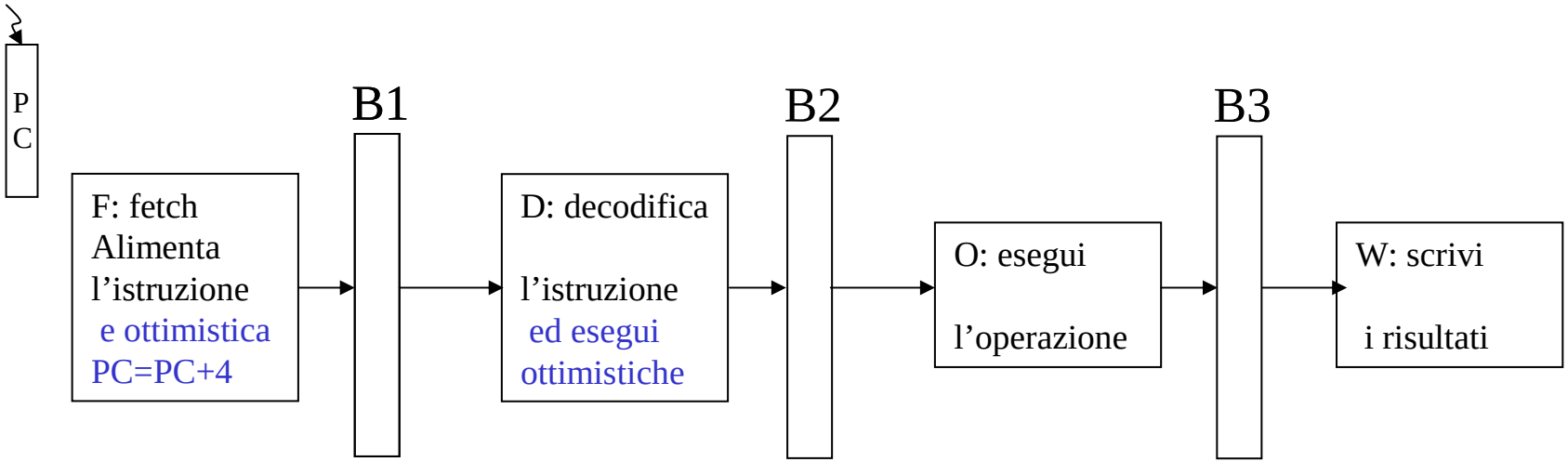
NB: a differenza del MIPS (cfr. testo) stiamo considerando una suddivisione in 4 stadi.

NB: come operazioni ottimistiche in stadio D consideriamo il prelievo degli operandi
[Potrebbero esserci anche altre operazioni, p.es. come nel caso multiciclo
il calcolo indirizzo di salto, però in questo caso non "paga" molto…]

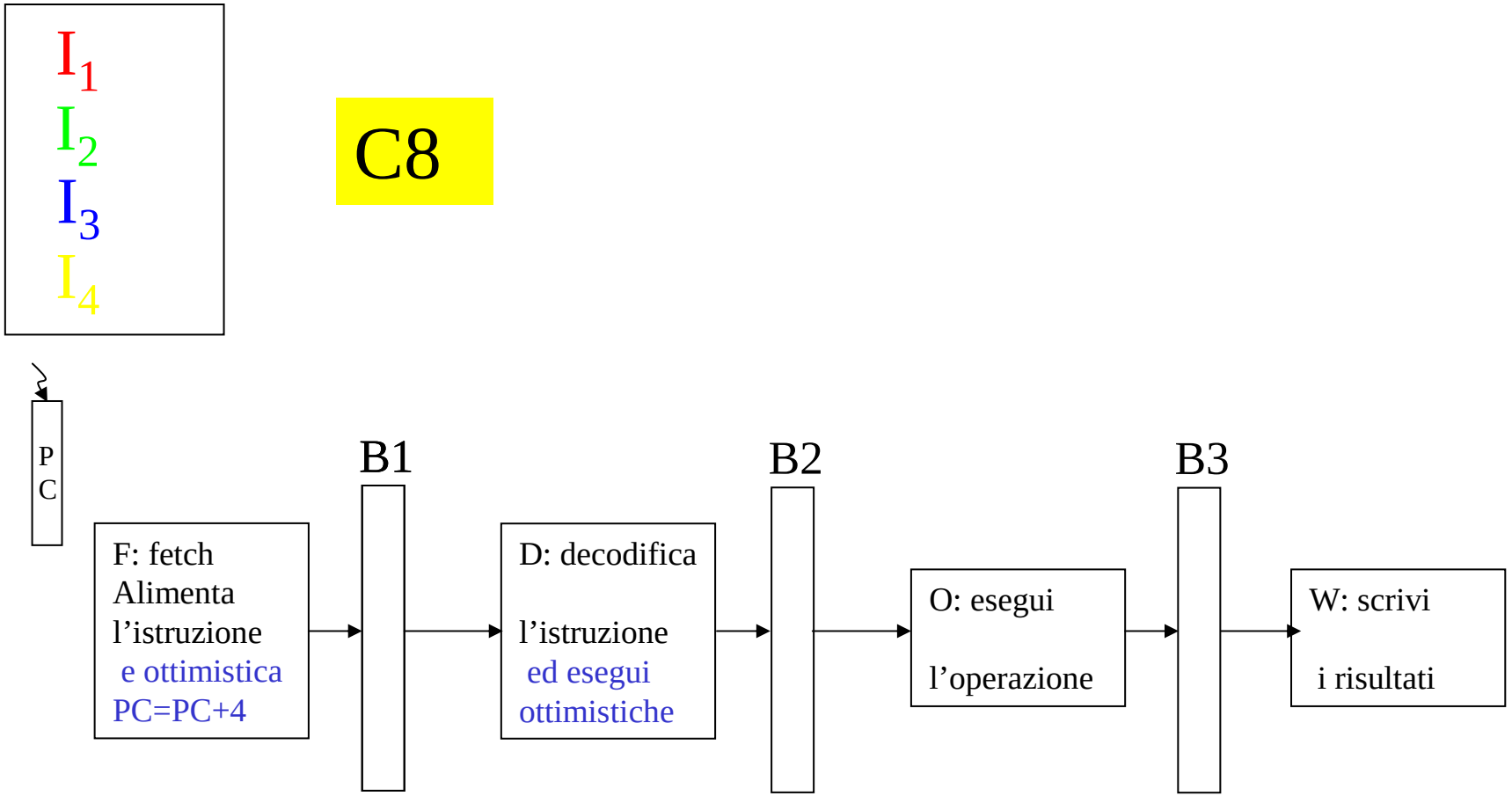
Animazione: esecuzione di una istruzione

I_1

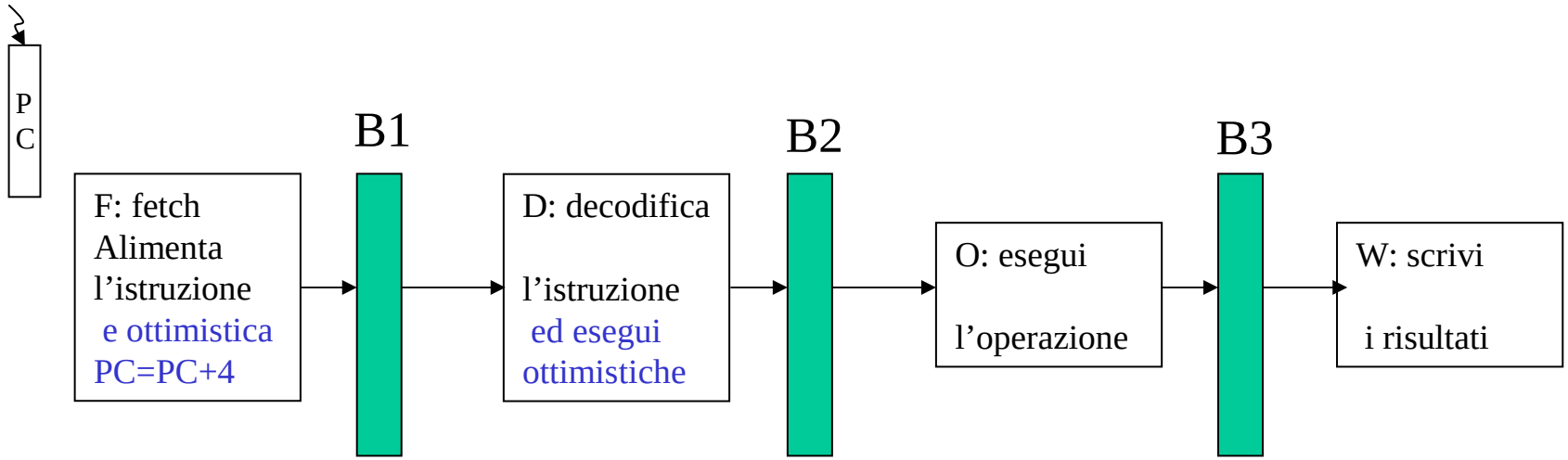
C4

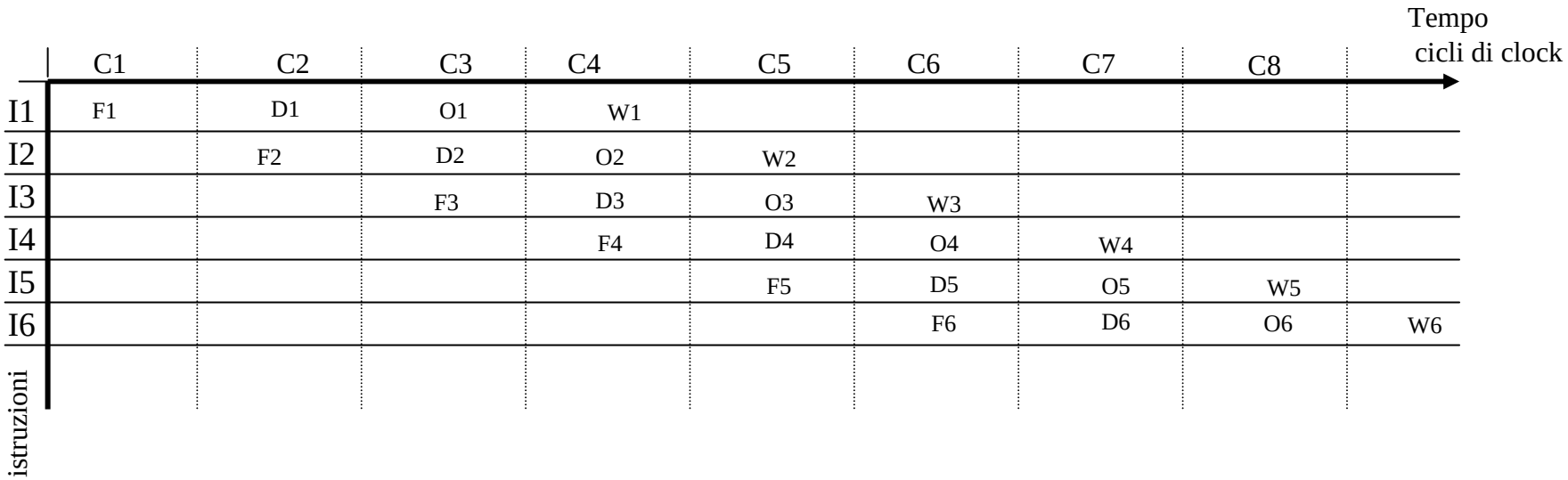


Animazione: esecuzione di quattro istruzioni



Esempio
di schema temporale dell'esecuzione





C1:

durante il ciclo: viene letta l'istruzione I_1 dalla memoria

alla fine del ciclo: in PC viene scritto il valore PC+4

nel buffer B1 viene scritta I_1

C2:

durante il ciclo: lettura I_2 dalla memoria + decodifica di I_1 (presente in B1)

lettura degli operandi di I_1

alla fine del ciclo: in PC viene scritto il valore PC+4

in B1 viene scritta I_2,

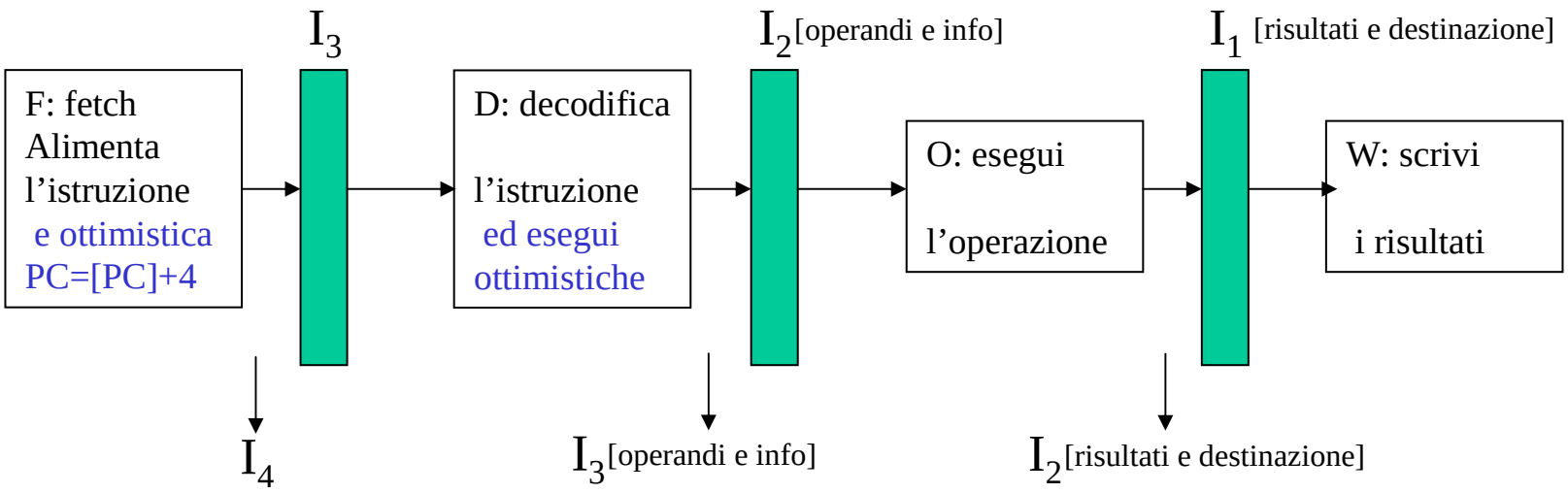
in B2 vengono scritti operandi sorgente I_1

e informazioni necessarie all'esecuzione

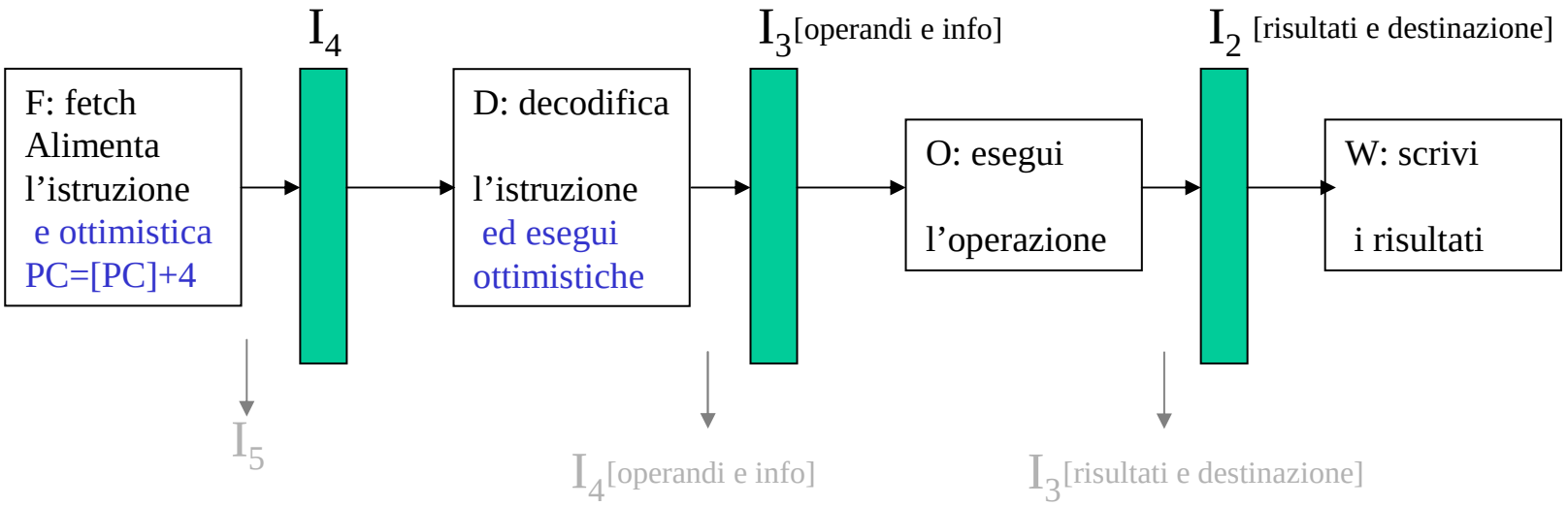
di I_1 negli stadi successivi

. . .

durante C4:



dopo il fronte di salita del clock (inizio di C5):



Datapath: prestazioni e suddivisione in stadi

- Abbiamo visto che “buone prestazioni” sono favorite (tra le altre cose) da:
  - velocità dei circuiti [alto grado di integrazione e densità circuitale]
  - diminuzione lunghezza del “cammino critico” per abbassare T_{clock}

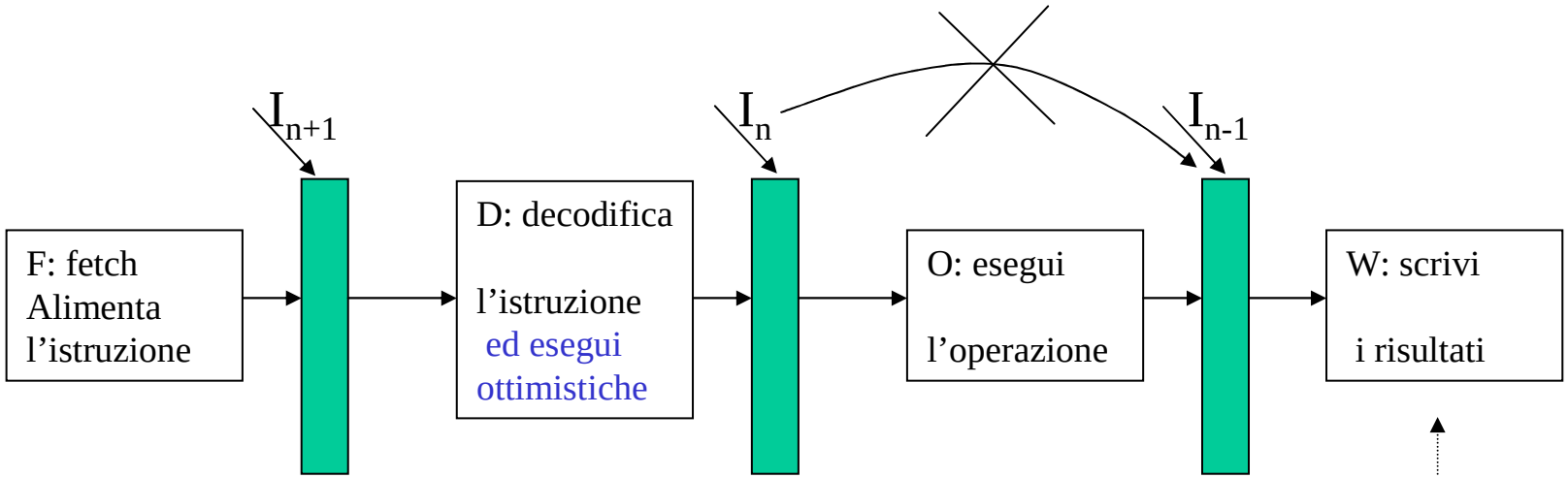
- Rispetto al processore a singolo ciclo, il multi-ciclo abbassa il tempo di esecuzione delle istruzioni più frequenti, che “non devono adeguarsi” alle istruzioni che hanno il “cammino critico” più lungo.
  Tuttavia, ogni istruzione deve aspettare la fine della precedente…

- La tecnica con pipeline aumenta il numero di istruzioni completate in un ciclo di clock (throughput), ovvero diminuisce il CPI (cicli per istruzione): idealmente, un’istruzione non deve mai aspettare la fine della precedente (CPI=1)!

- Viceversa, il tempo di esecuzione della singola istruzione è in generale aumentato!
  In generale, servono più cicli di clock per una singola istruzione, pari al numero degli stadi, anche per le istruzioni che ne usino un sottoinsieme!
  Ciò però non ha alcuna importanza, visto che comunque “comincia” un’istruzione per ciclo di clock!

Infatti: supponiamo che una istruzione I_n non usi lo stadio di esecuzione

Alla fine di un certo ciclo di clock si ha:

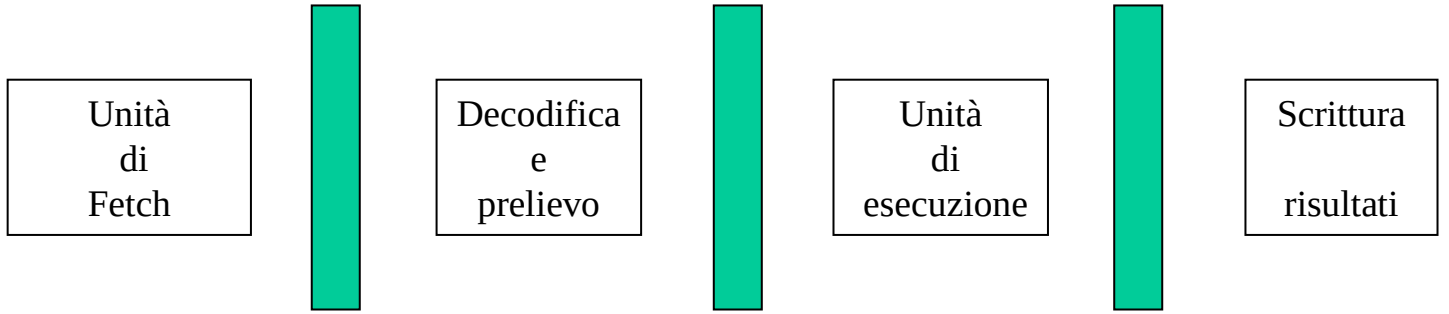


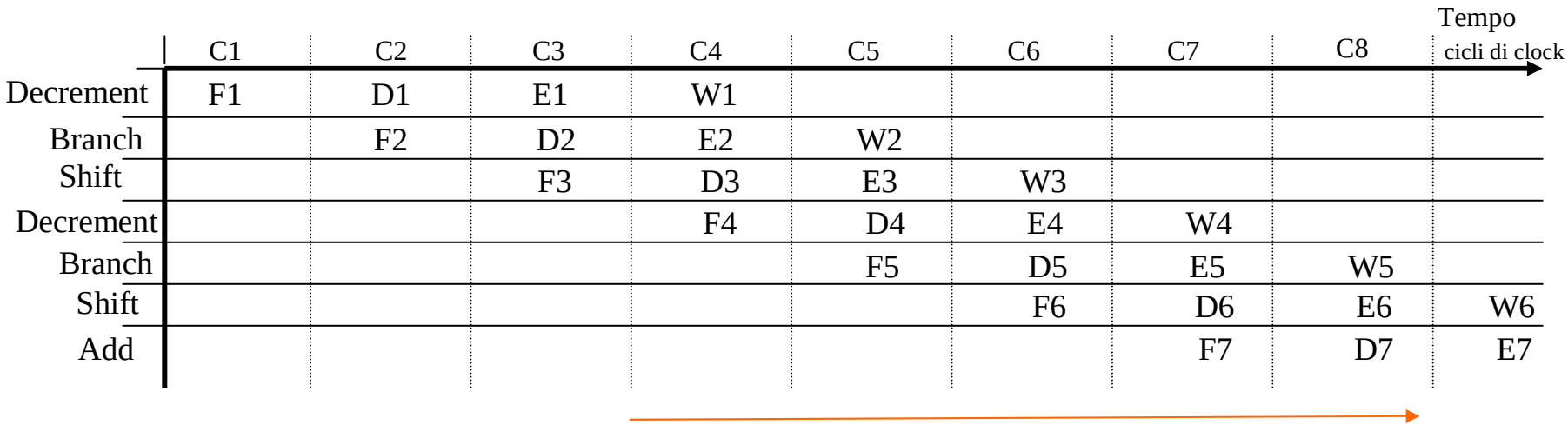
Non è possibile per I_n utilizzare nel ciclo successivo lo stadio W, perché già impegnato per l'esecuzione di I_{n-1}

D'altra parte, ciò non comporterebbe un vantaggio sostanziale: la decodifica di I_{n+1} (già caricata) comincia comunque, nessuna istruzione deve più aspettare la fine della precedente!

⇒ Ciò che conta è il throughput…

Throughput come numero di istruzioni eseguite per ciclo di clock



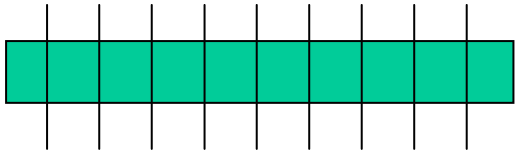


- Consideriamo la situazione “a regime” (quando la pipeline è piena), in questo caso a partire da C4
- Da C4 in poi viene conclusa un’istruzione per ciclo

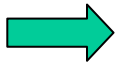
(es.tra C4 e C8 5 istruzioni in 5 cicli: throughput di 1 istruzione per ciclo)

• Ovviamente, il numero di istruzioni eseguite al secondo aumenta diminuendo T_{clock}

• Idealmente, aumento throughput proporzionale al numero di stadi:



T_{clock} nel caso di singolo ciclo



Mi basta aumentare il numero di stadi in cui suddivido l'esecuzione dell'istruzione per aumentare le prestazioni…

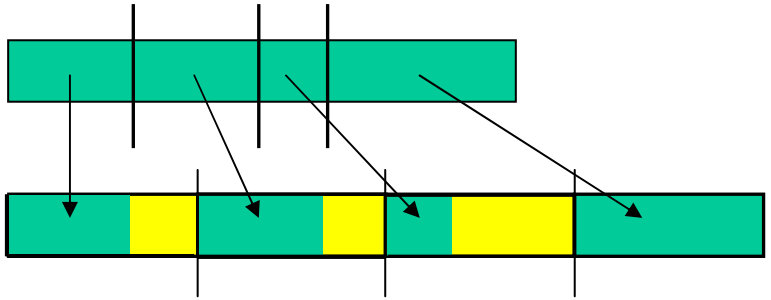
• Putroppo le cose non sono così semplici:

– C'è un limite "fisico" al numero di stadi (le unità elementari di calcolo!)

– Gli stadi devono essere **bilanciati**!!!

T_{clock} deve essere maggiore dell'operazione più lunga tra i vari stadi! (stessa cosa accadeva nel caso del multiciclo)

Quindi in realtà:



– aumento numero di stadi aumenta problemi nella gestione delle criticità (lo vedremo in seguito)

PROBLEMATICA:

in quanti e quali stadi suddividere l'esecuzione?

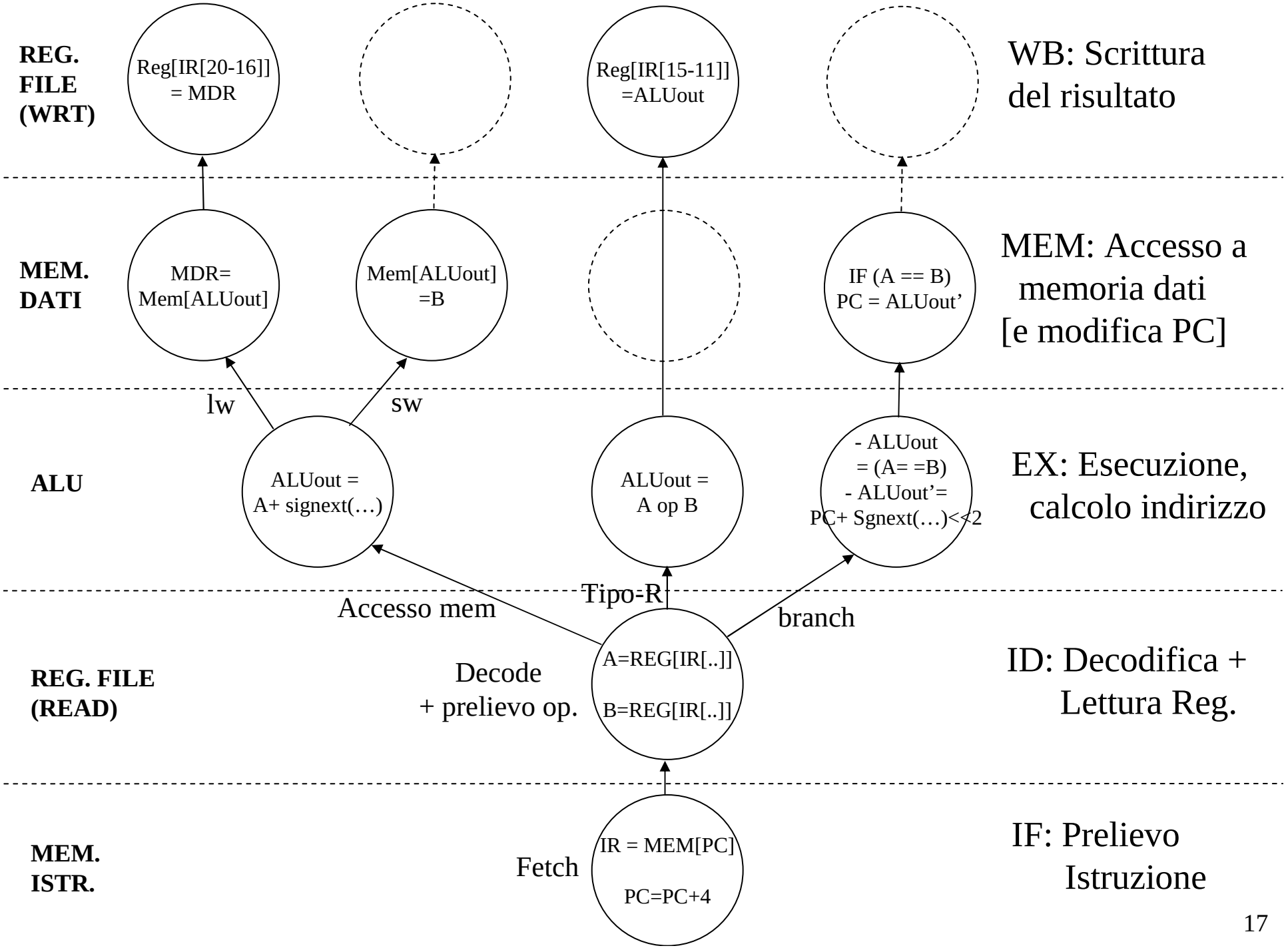
➡ Occorre che gli stadi siano "bilanciati"

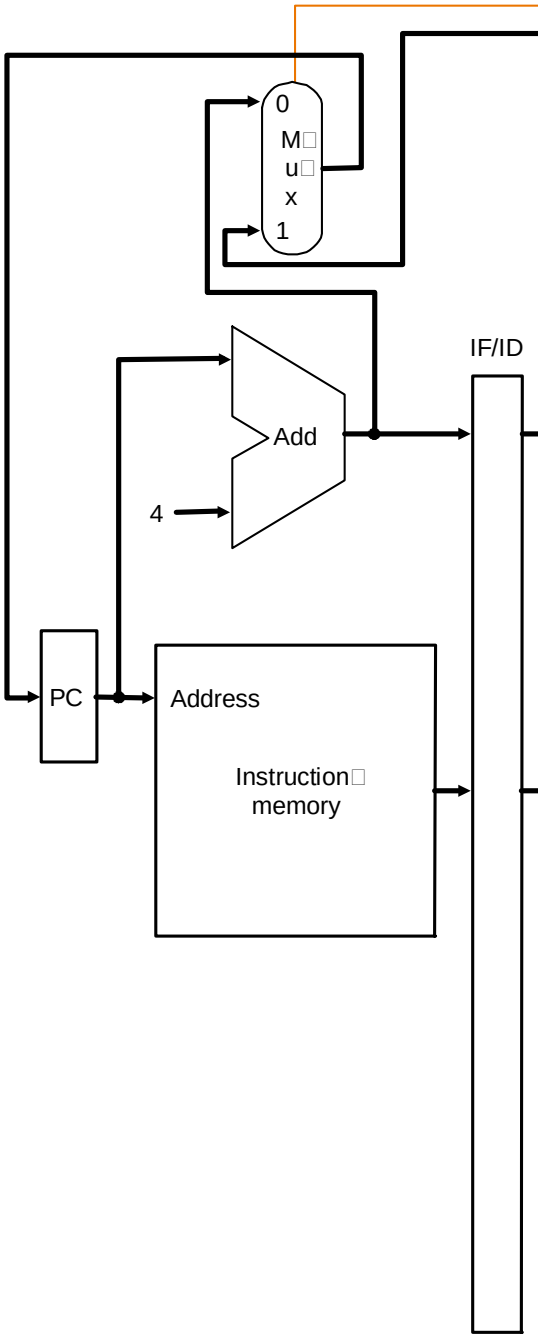
- Nei casi reali, il numero va da 2-3 a qualche decina…

- **Vediamo il caso MIPS didattico** presentato nel
  Patterson & Hennessy
  (in cui si considerano le istruzioni Tipo-R, lw, sw, beq)
  $\Rightarrow$ Suddivisione in stadi, ciascuno con distinte risorse HW.

  Per rendere bilanciati gli stadi, ci riferiamo alle unità funzionali più critiche: come nel caso del multiciclo, Memoria, Register File, ALU mai in serie

  NB: si considera memoria istruzioni distinta da memoria dati
  (vedremo subito perché)


REG. FILE (WRT)
Reg[IR[20-16]] = MDR
Reg[IR[15-11]] =ALUout
WB: Scrittura del risultato
MEM. DATI
MDR= Mem[ALUout]
Mem[ALUout] =B
IF (A == B) PC = ALUout'
MEM: Accesso a memoria dati [e modifica PC]
lw
sw
ALU
ALUout = A+ signext(…)
ALUout = A op B
- ALUout = (A= =B) - ALUout'= PC+ Sgnext(…)<<2
EX: Esecuzione, calcolo indirizzo
Accesso mem
Tipo-R
branch
REG. FILE (READ)
Decode + prelievo op.
A=REG[IR[..]]
B=REG[IR[..]]
ID: Decodifica + Lettura Reg.
MEM. ISTR.
Fetch
IR = MEM[PC]
PC=PC+4
IF: Prelievo Istruzione

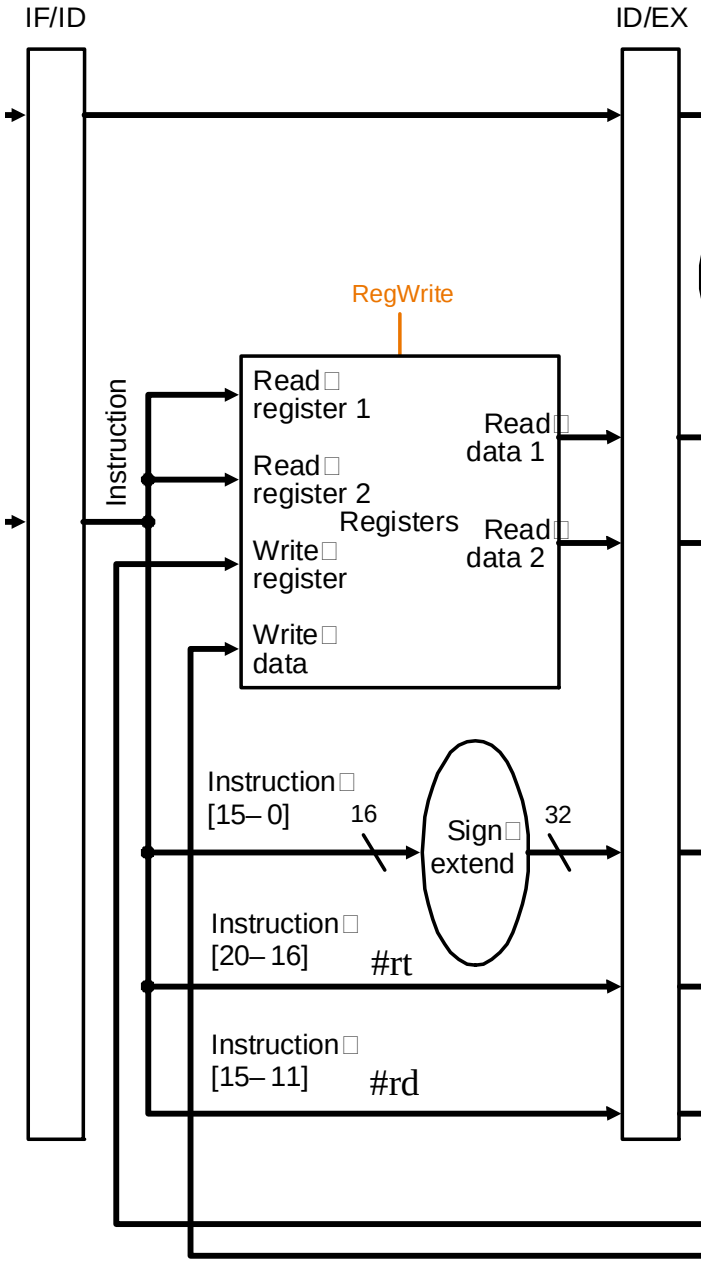


PRIMO STADIO: INSTRUCTION FETCH

- Operazione di fetch identica per qualunque Istruzione venga eseguita

- PC è aggiornato ad ogni ciclo di clock per poter caricare un'istruzione ad ogni ciclo.

- informazioni in IF/ID:
  - Istruzione da decodificare ed eseguire
  - PC+4 (può essere utilizzato per il calcolo dell'indirizzo nel caso della beq)

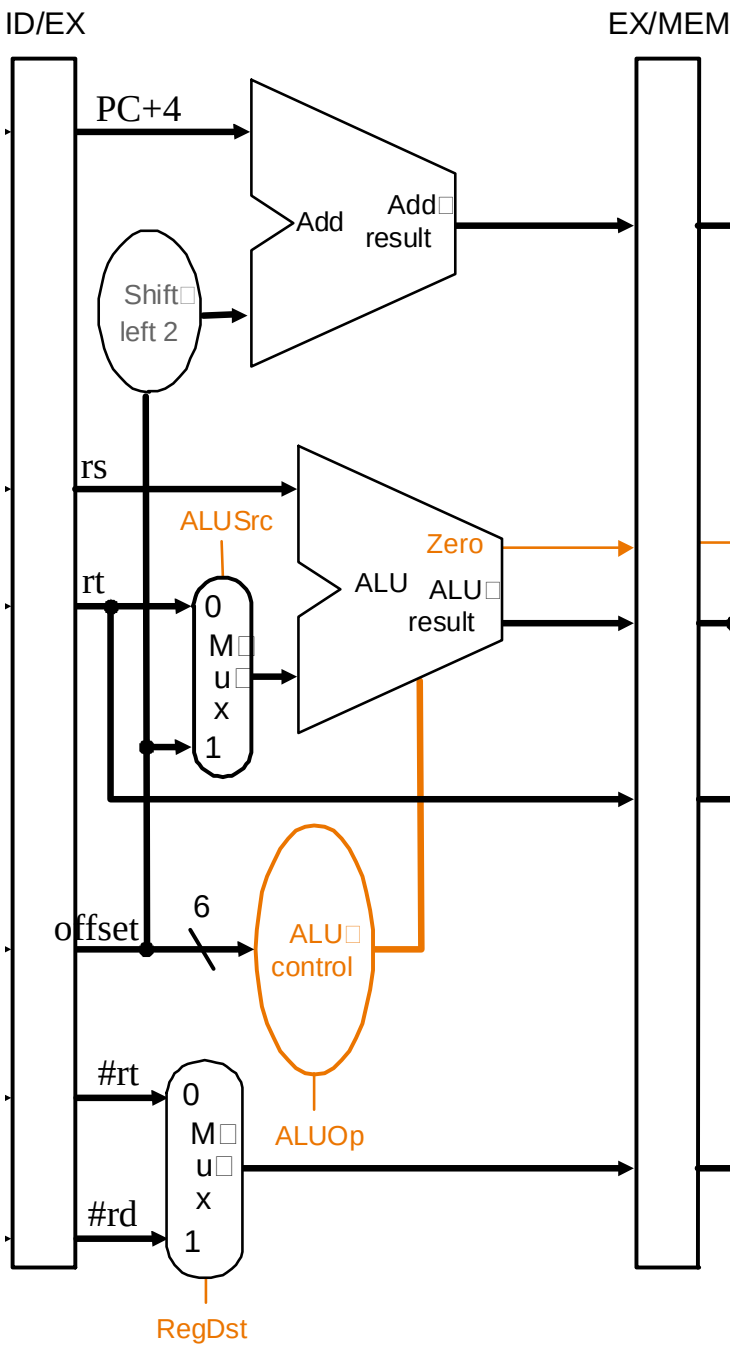


SECONDO STADIO: INSTRUCTION DECODE

- Operazione di decode identica per qualunque Istruzione venga eseguita

- informazioni in IF/ID (input):
  - Istruzione da decodificare ed eseguire
  - PC+4 (può essere utilizzato per il calcolo dell'indirizzo nel caso della beq)

- informazioni prodotte in ID/EX (output):
  - Valori registri rs e rt letti da Register File
  - numero dei registri rt e rd (possibili destinazioni)
  - Offset con estensione del segno (cfr. lw, sw, beq)
  - PC+4 (direttamente da IF/ID) che può servire in seguito

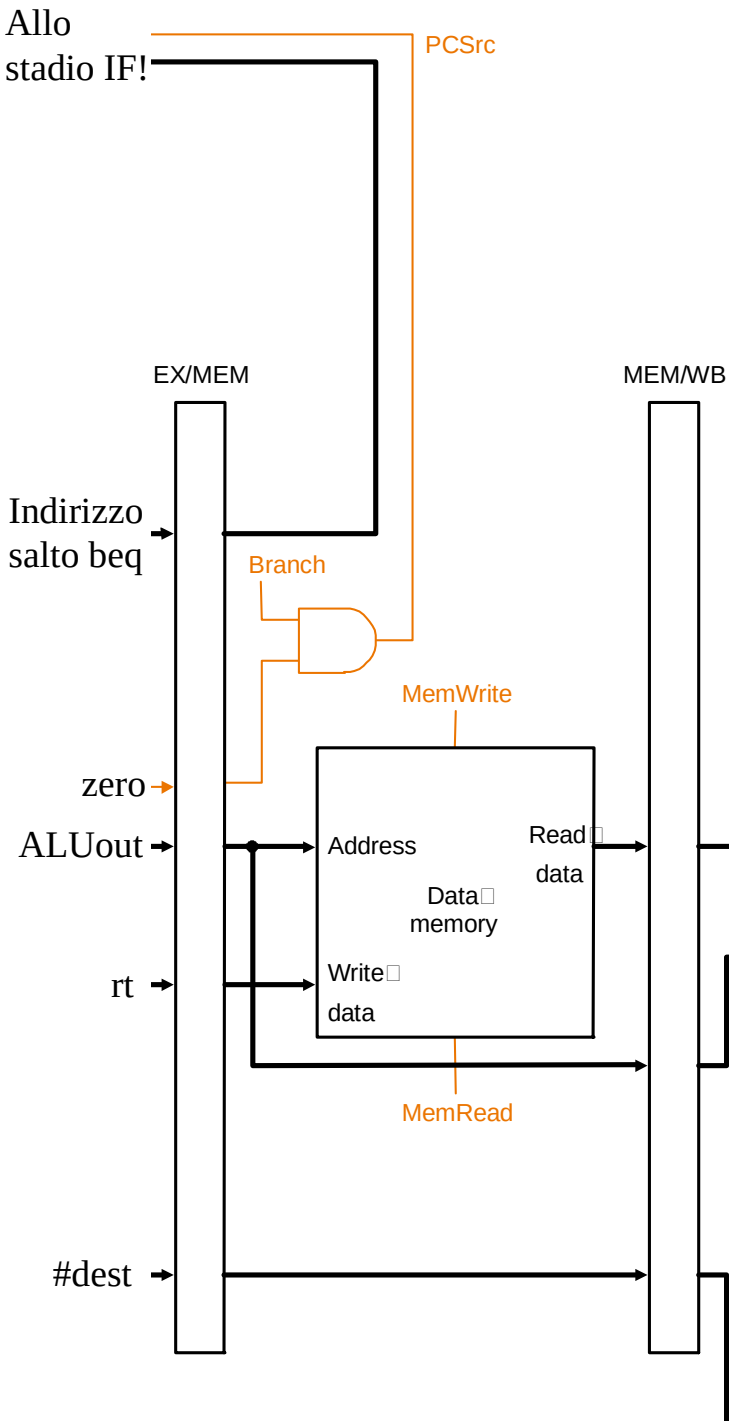


TERZO STADIO: EXECUTE

- Utilizzo della ALU + determinazione del registro da usare come (eventuale) destinazione: ovviamente variano a seconda dell'istruzione

- informazioni in ID/EX (input):
  - Valori registri rs e rt letti da Register File
  - numero dei registri rt e rd (possibili destinazioni)
  - Offset con estensione del segno (cfr. lw, sw, beq)
  - PC+4 (direttamente da IF/ID)

- informazioni in EX/MEM (output):
  - indirizzo utilizzato dal salto condizionato (beq)
  - risultato della ALU (per lw e sw è l'indirizzo!) e zero (verrà utilizzato dalla beq)
  - rt (se l'istruzione è sw, questo dato verrà scritto in memoria!)
  - numero del registro destinazione, che ora può essere determinato (rd per TIPOR, rt per lw)

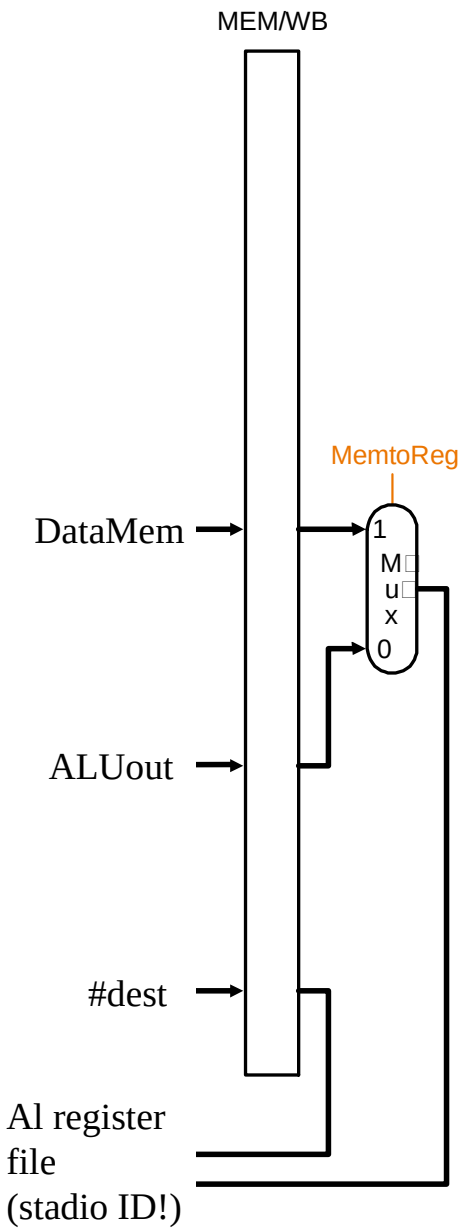


QUARTO STADIO: MEMORY ACCESS

- Utilizzo della memoria (lettura e scrittura) ed eventuale aggiornamento PC (beq con salto da eseguire)

- informazioni in EX/MEM (input):
  - indirizzo utilizzato dal salto condizionato (beq)
  - risultato della ALU (per lw e sw è l'indirizzo!) e zero
  - rt
  - numero del registro destinazione

- informazioni in MEM/WB (output):
  - dato letto dalla memoria (per lw)
  - risultato calcolo ALU effettuato nello stadio precedente riportato da EX/MEM (per TIPO-R)
  - # registro destinazione riportato da EX/MEM (per lw e TIPO-R)



QUINTO STADIO: WRITE BACK

- Scrittura del risultato nel register file

  $\Rightarrow$ Register File "in scrittura" fa parte di questo stadio!

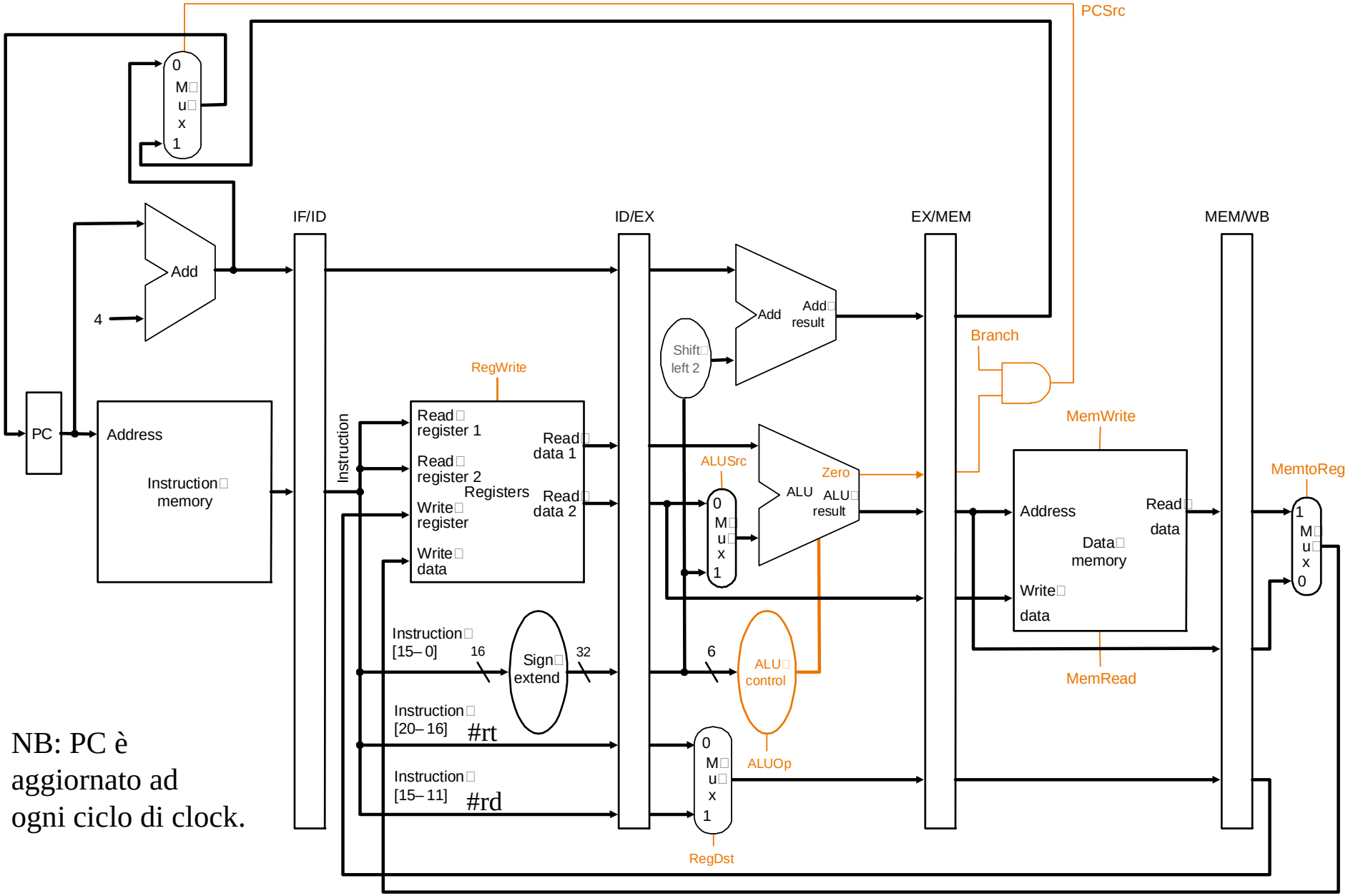
- informazioni in MEM/WB (input):
  - dato letto dalla memoria (per lw)
  - risultato calcolo ALU effettuato nello stadio precedente riportato da EX/MEM (per TIPO-R)
  - # registro destinazione riportato da EX/MEM (per lw e TIPO-R)

Operazione eseguita: se l'istruzione è lw o Tipo-R, scrive nel registro destinazione (#dest) il valore calcolato dalla ALU (per Tipo-R) o letto dalla memoria (per lw)

$\Rightarrow$ deve essere utilizzato il segnale RegWrite

$\Rightarrow$ non occorre un registro interstadio di output: il risultato è scritto nel register file!

L’unità di elaborazione corrispondente è: [Fig. 6.22 Patterson & Hennessy]



NB: PC è aggiornato ad ogni ciclo di clock.

PIPELINE:

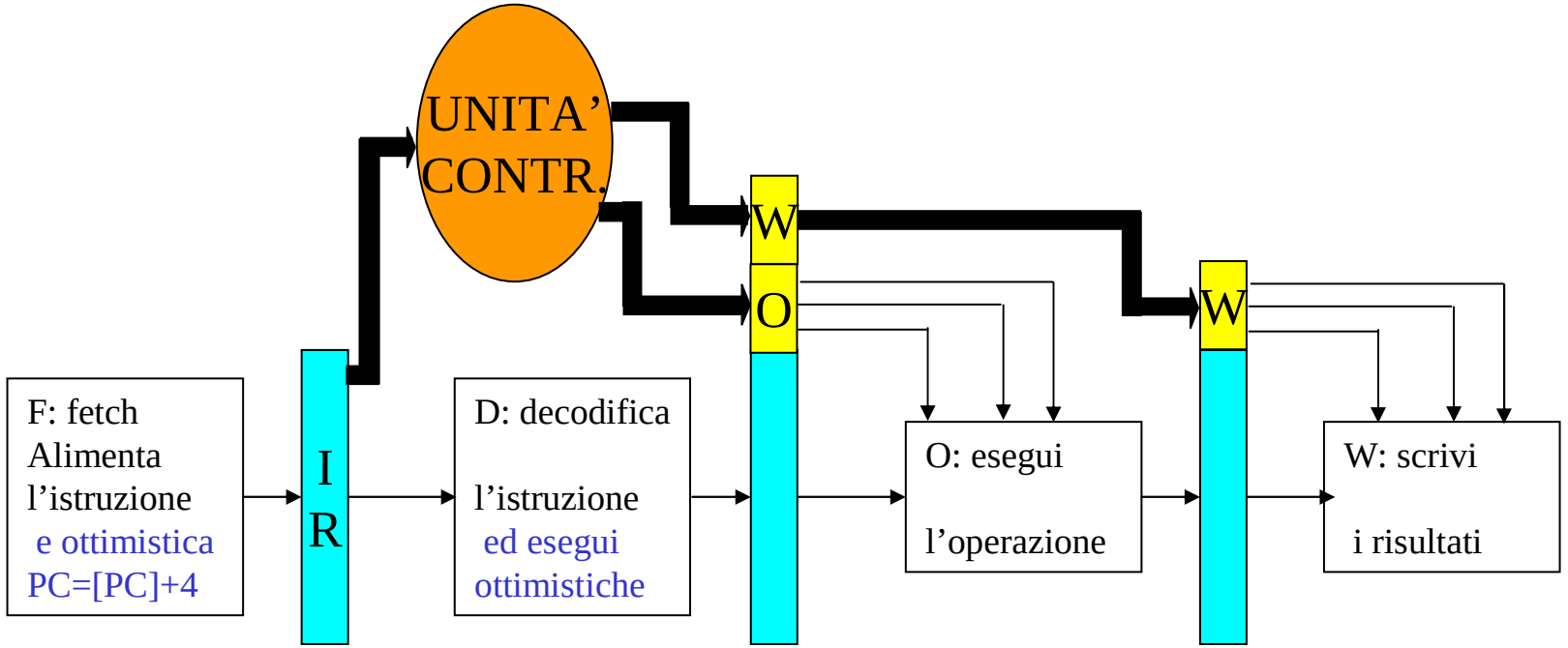
IL CONTROLLO

NB: come visto, al datapath arrivano segnali di controllo, che dovranno essere impostati dall'unità di controllo del processore

PROBLEMA: in un ciclo di clock, ogni stadio è dedicato ad un'istruzione diversa!

SOLUZIONE: segnali di controllo visti come "prodotto" dello stadio ID necessario per gli stadi successivi, depositato nel registro ID/EX

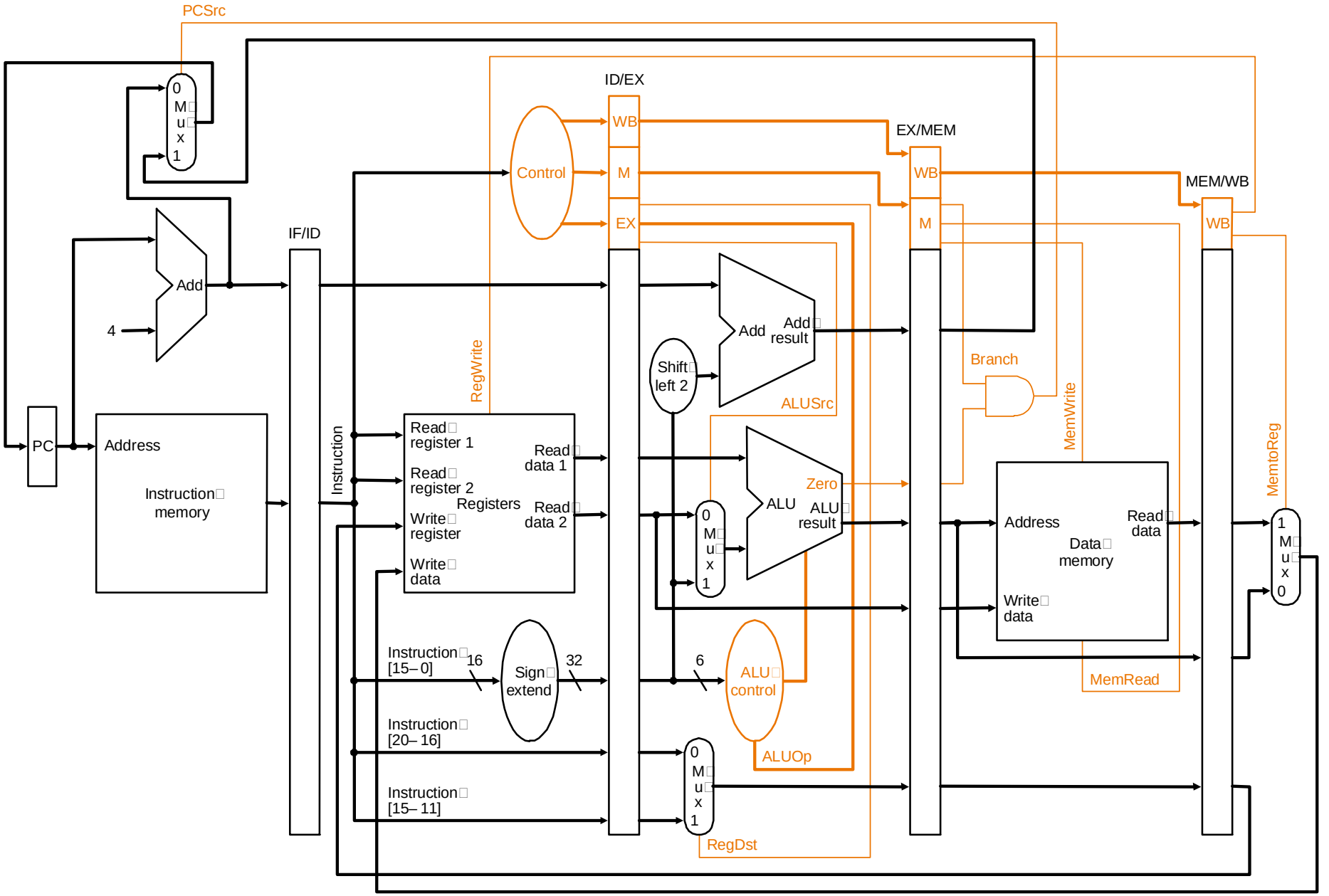
L’idea di base in astratto

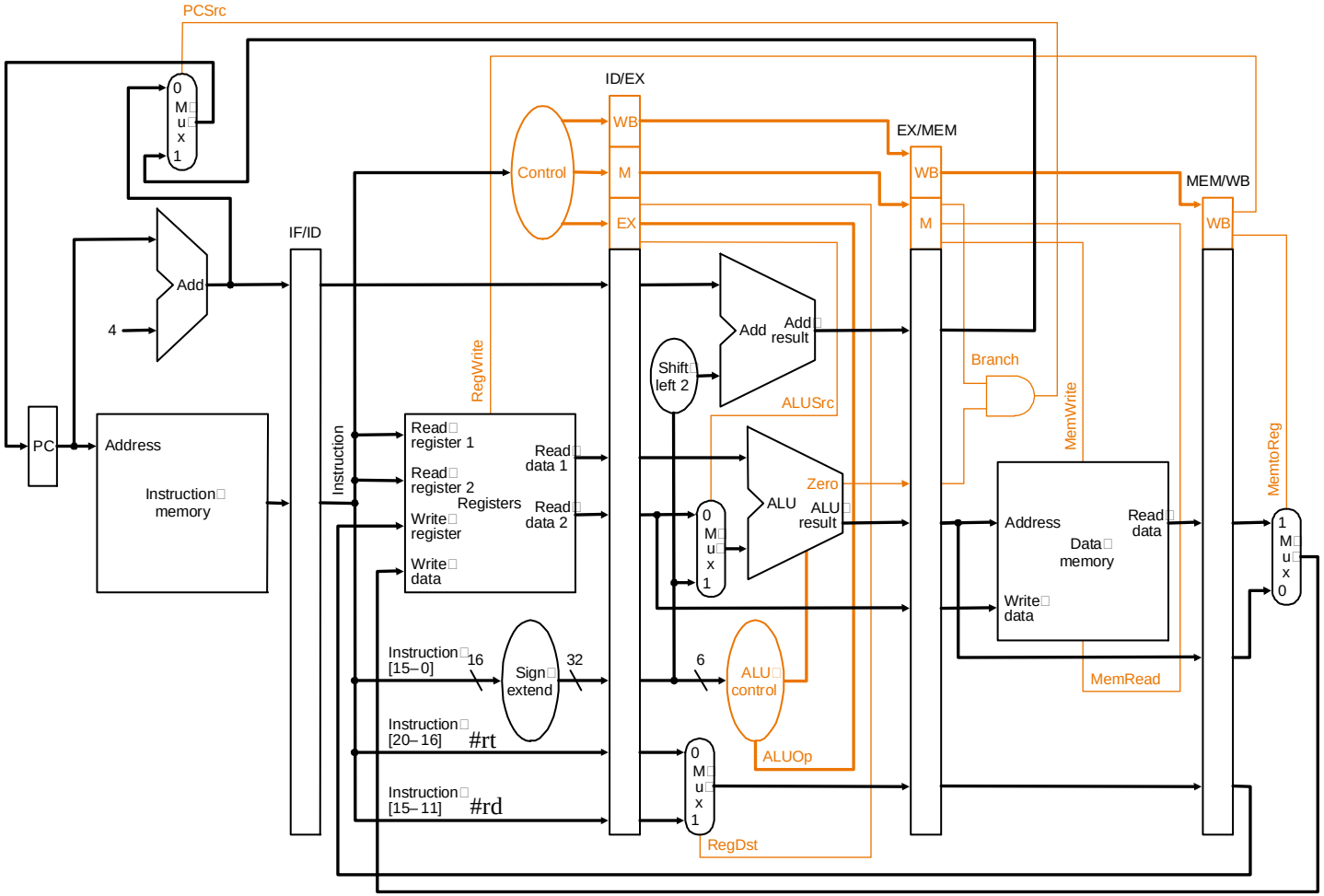


- L’unità di controllo è combinatoria: in fase di decodifica (D) calcola tutti i segnali di controllo che saranno necessari in tutti gli stadi.
- I segnali di controllo si spostano lungo la pipeline allo stesso modo delle altre informazioni (p.es. gli operandi)
- Questo è il modo più conveniente di realizzare un comportamento sequenziale: la decodifica è fatta nel primo stadio, che nel ciclo di clock successivo dovrà essere impegnato dall’istruzione successiva!

Il caso MIPS a 5 stadi

[Fig. 6.27 Patterson & Hennessy]





	EX			MEM			WB	
	RegDst	ALUOp	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemToReg
Tipo-R	1	funct	0	0	0	0	1	0
lw	0	sum	1	0	1	0	1	1
sw	X	sum	1	0	0	1	0	X
beq	X	sub	0	1	0	0	0	X

PIPELINE:

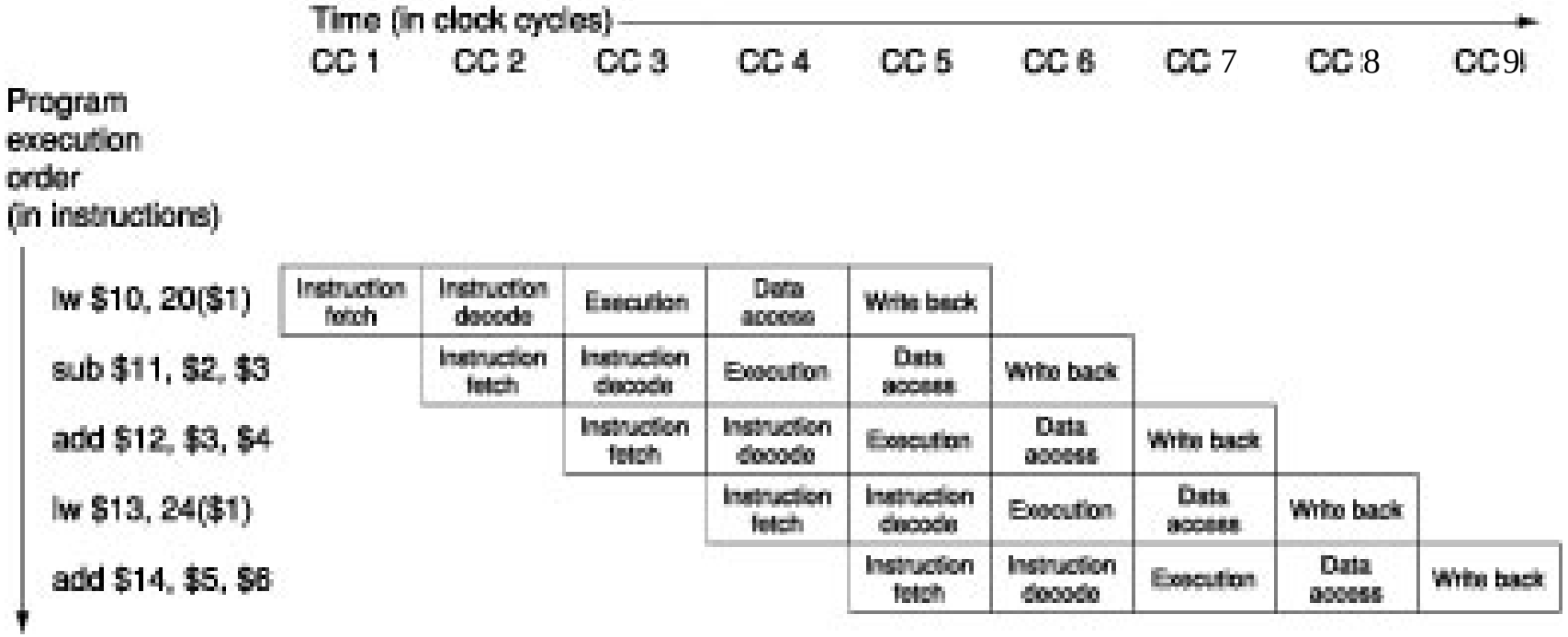
- DIAGRAMMI DI RAPPRESENTAZIONE

- PRESTAZIONI [NEL CASO IDEALE]

Rappresentazione grafica della Pipeline

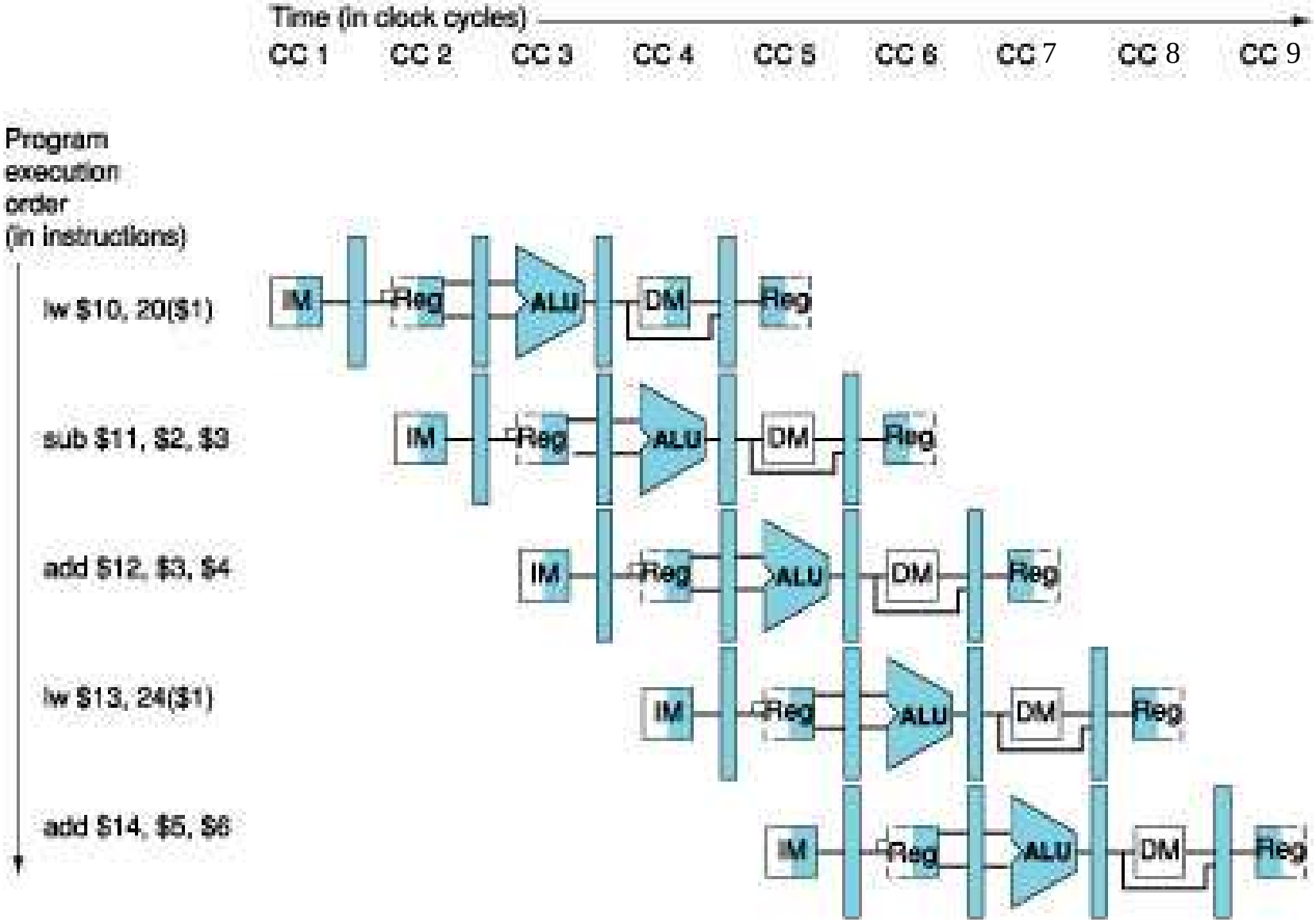
I diagrammi che abbiamo utilizzato (e che utilizzeremo) possono essere di due tipi:

1) Diagramma della pipeline “a più cicli di clock”: esempio riferito al MIPS



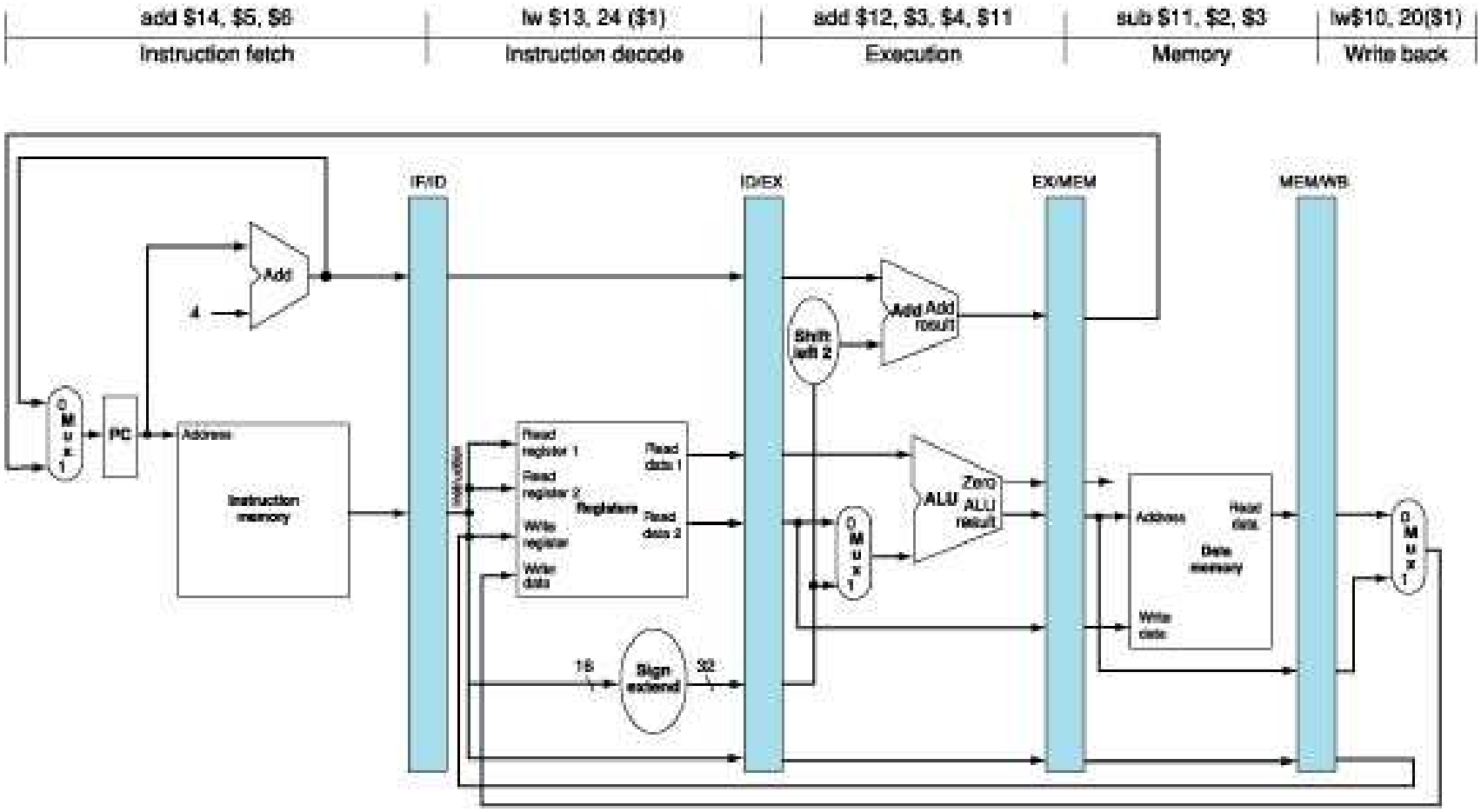
NB: quello che si userà prevalentemente negli esercizi…

… il cui significato è…



2) Diagramma della pipeline “a singolo ciclo di clock”

Ad esempio, nel ciclo di clock 5 del precedente diagramma, si ha:



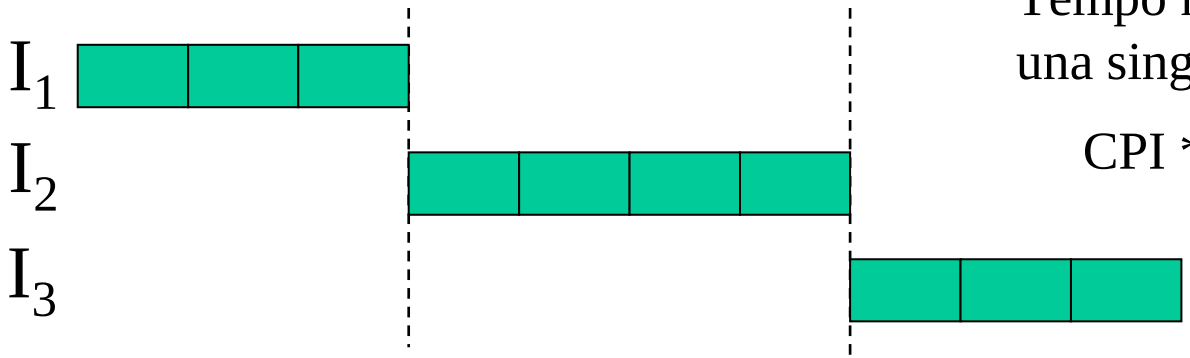
Cfr. CD del libro:

- “For More Practice” – CAP. 6 – pag. 6.14.7

Prestazioni per processore con pipeline

Si ricorda che, per un processore multiciclo, era:

$$T_{esecuzione} = \#istruzioni * CPI * T_{clock}$$



Tempo medio per eseguire una singola istruzione:

$$CPI * T_{clock}$$

L'esecuzione di ogni istruzione comincia dopo che l'esecuzione della precedente è conclusa:

$$T_{esecuzione} = \#istruzioni * (\text{Tempo medio per eseguire una istruzione})$$

Con la pipeline, i parametri hanno un significato diverso:
le istruzioni "non aspettano" le precedenti e viene completata
(a parte il caso di stalli che vedremo in seguito) una istruzione per ciclo di clock!!!

Facciamo un esempio con la pipeline…

- Supponiamo di avere una pipeline a 5 stadi (prelievo, decodifica/lettura registri, esecuzione, accesso in memoria, scrittura registri)
- Supponiamo che la durata delle unità funzionali sia la seguente:
  - Prelievo, esecuzione con ALU, accesso in memoria = 2 ns
  - Decodifica/lettura registri, scrittura registri = 1 ns
- Ogni stadio di clock richiede un ciclo di clock: il periodo di clock deve essere abbastanza lungo da contenere l'operazione più lenta

  $\Rightarrow$ il ciclo di clock deve essere posto a 2 ns

➡ Tempo per eseguire una qualunque istruzione:

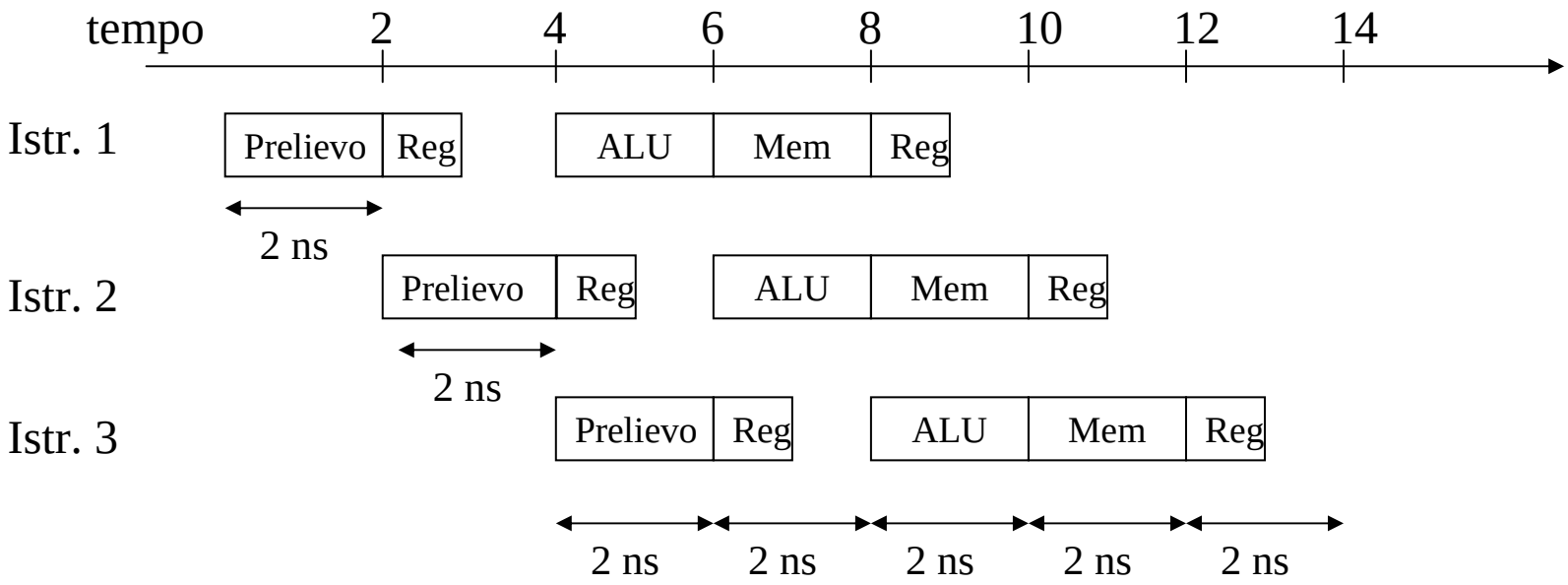
5 stadi * T_{clock} = 10 ns

Tuttavia, $T_{esecuzione}$ <u>non</u> è

#istruzioni * (Tempo per eseguire una istruzione)

Vediamo di determinare $T_{esecuzione}$ …

Esecuzione in pipeline di 3 istruzioni

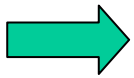


- Il tempo totale di esecuzione di 3 istruzioni è 14 ns, a causa del tempo di startup della pipeline
- "Tempo medio di esecuzione per istruzione" = 14/3 ≈ 4,67 ns
  ⇒ tempo non significativo, non siamo nella situazione a regime !!!

Tempo di esecuzione per processore con pipeline a regime

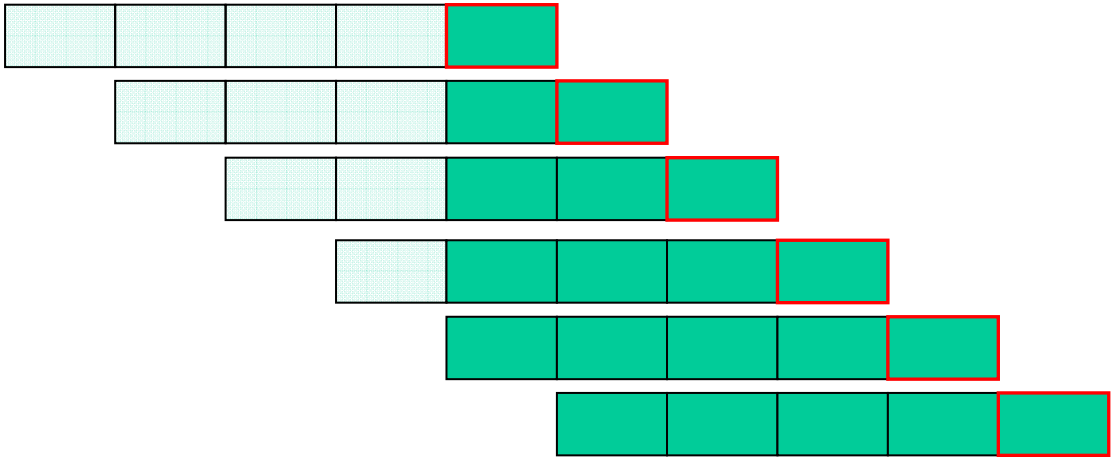
- Il tempo medio di esecuzione prima calcolato non è significativo: occorre considerare la situazione a regime
- Ad esempio, supponiamo di aggiungere 1000 istruzioni alle 3 di cui sopra:
  - Ogni istruzione in più fa aumentare il tempo di esecuzione di 2 ns
  - Il tempo totale di esecuzione di 1003 istruzioni è quindi:

    $14 \text{ ns} + 1000 \times 2 \text{ ns} = 2014 \text{ ns}$
  - Quindi, il tempo medio di esecuzione diventa = $2014/1003 \approx 2$ ns



Tempo medio di esecuzione pari a 2 ns significa che, a regime, viene completata un'istruzione ogni 2 ns (cioè ad ogni ciclo di clock)

La situazione “a regime”



. . .

Ad ogni ciclo, cioè ogni T_{clock} = 2 ns, viene completata un’istruzione:
il *tempo medio di esecuzione* è il tempo che intercorre
fra il completamento di due istruzioni successive (in questo caso 2 ns)

NB: grazie alla pipeline, aumenta il ***throughput delle istruzioni***
(ovvero il rapporto ***numero istruzioni/ciclo di clock***)
e non il tempo di esecuzione di un’istruzione
(che dipende anche dal numero di stadi della pipeline)

Tempo medio di esecuzione, throughput e CPI

Anche per la pipeline, a regime vale la formula

$$T_{esecuzione} = \#istruzioni * \underbrace{CPI * T_{clock}}_{\text{Tempo medio di esecuzione}}$$

PIPELINE IDEALE:

Si considera **CPI = 1**, in quanto ad ogni ciclo di clock viene completata un' istruzione

⇒ $T_{esecuzione} = \#istruzioni * T_{clock}$

Throughput = 1/CPI = 1 (espresso come istruzioni/ciclo)
= $1/(CPI * T_{clock}) = 1/T_{clock}$ (espresso come istruzioni/sec)

PIPELINE IN PRESENZA DI STALLI (vedremo):

CPI = CPI ideale + Cicli di stallo per istruzione

Pipeline: i problemi

- Idealmente, il throughput è di una istruzione per ciclo di clock!
- Purtroppo, in realtà esistono diverse problematiche:
  - Criticità **strutturali**: HW non può eseguire una certa combinaz. di istruzioni [es. contesa della stessa risorsa da parte di più istruzioni]
  - Criticità **sui dati**: un'istruzione dipende dal risultato di un'istruzione precedente che si trova ancora nella pipeline. E' necessario attendere che il risultato sia pronto.
  - Criticità **sul controllo**: l'istruzione successiva ad un'istruzione di salto deve attendere l'esecuzione della precedente, per sapere se/dove saltare.

<u>Un chiarimento sui termini che useremo</u>

CRITICITA': un'istruzione non può essere eseguita nel ciclo di clock immediatamente seguente (pena comportamenti scorretti)

⇨ **STALLO**: sospensione di una unità della pipeline (e delle precedenti)