
Architetture RISC

Considerazioni di base x la soluzione RISC 1/3

- nei codici oggetto prodotti dai compilatori per i processori CISC la distribuzione delle istruzioni macchina non è uniforme: circa il 10% delle istruzioni ricorrono nel 90% dei casi, mentre le più complesse vengono raramente usate; inoltre le istruzioni più usate sono anche quelle più semplici (come il trasferimento dati da memoria a registro e viceversa, operazioni tra il contenuto di registri);
- le istruzioni complesse, con differenti tipi di indirizzamenti, necessitano di una unità di controllo complessa, con conseguente rallentamento dell'esecuzione del ciclo macchina e considerevole occupazione di spazio fisico nel chip del processore;
- il tempo di esecuzione delle istruzioni macchina è fortemente dipendente dal tempo di accesso alla memoria di lavoro (RAM statiche o dinamiche) per prelevare le istruzioni (fetch) e per leggere/scrivere i dati;

Considerazioni di base x la soluzione RISC 2/3

- le istruzioni macchina che comportano solo operazioni tra il contenuto dei registri interni del processore, non dovendo interagire con la memoria di lavoro per la lettura/scrittura dei dati, hanno un tempo di esecuzione molto limitato;
- le limitazioni di velocità di accesso alla memoria esterna al processore sono principalmente dovute alle comunicazioni in-out tra chip (con conseguente uso di bus per poter utilizzare dispositivi standard – off the shelf);
- il miglioramento delle tecnologie di integrazione VLSI permette di integrare nello stesso dispositivo (chip) sia il processore sia le memorie cache;
- è possibile organizzazione il codice oggetto in modo che istruzioni consecutive non aggiornino lo stesso insieme di registri, ciò consente di poter eseguire contemporaneamente più istruzioni purché utilizzino sottosistemi indipendenti dello SCA del processore.

Considerazioni di base x la soluzione RISC 3/3

Ciò ha portato a:

- utilizzare il minor numero di tipi di istruzione;
- utilizzare solo due tipi di istruzioni per la lettura e scrittura dei dati dalla memoria;
- utilizzare istruzione di manipolazione dei dati che facciano riferimento solo ad operandi memorizzati nei registri interni del processore;
- eseguire più istruzioni contemporaneamente;
- usare una struttura pipeline nel sottosistema di calcolo del processore (data path);
- utilizzare una cache per le istruzioni da eseguire ed una cache per i dati da elaborare.

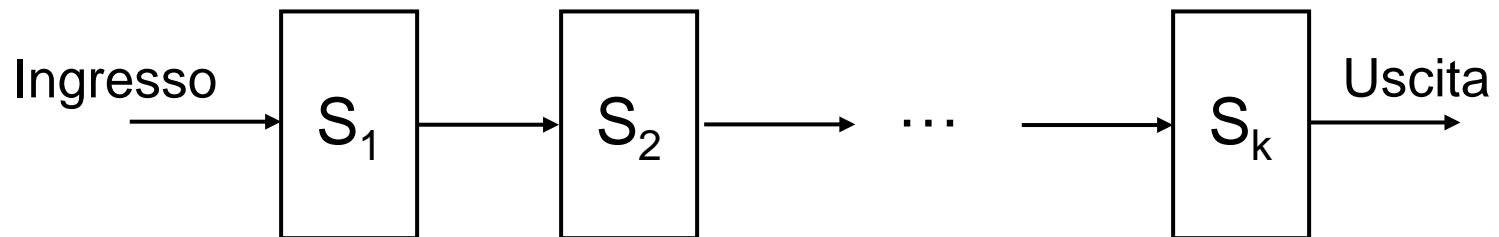
Il pipelining: tecniche di base

Il pipelining

- E' una tecnica
 - per migliorare le prestazioni del processore
 - basata sulla **sovrapposizione** dell'esecuzione di **più istruzioni** appartenenti ad un flusso di esecuzione sequenziale
- Analogia con la catena di montaggio

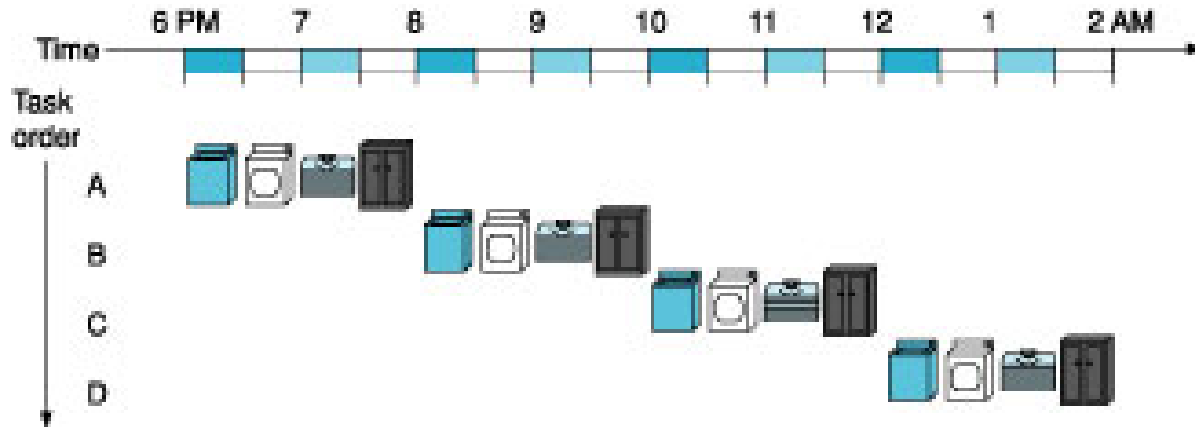
Idea base

- Il lavoro svolto da un processore con pipelining per eseguire un'istruzione è diviso in passi (*stadi della pipeline*), che richiedono una frazione del tempo necessario all'esecuzione dell'intera istruzione
- Gli stadi sono connessi in maniera seriale per formare la pipeline; le istruzioni:
 - entrano da un'estremità della pipeline
 - vengono elaborate dai vari stadi secondo l'ordine previsto
 - escono dall'altra estremità della pipeline



Un esempio pratico

Soluzione sequenziale/uniclo



Compiti

– Lavaggio



– Asciugatura



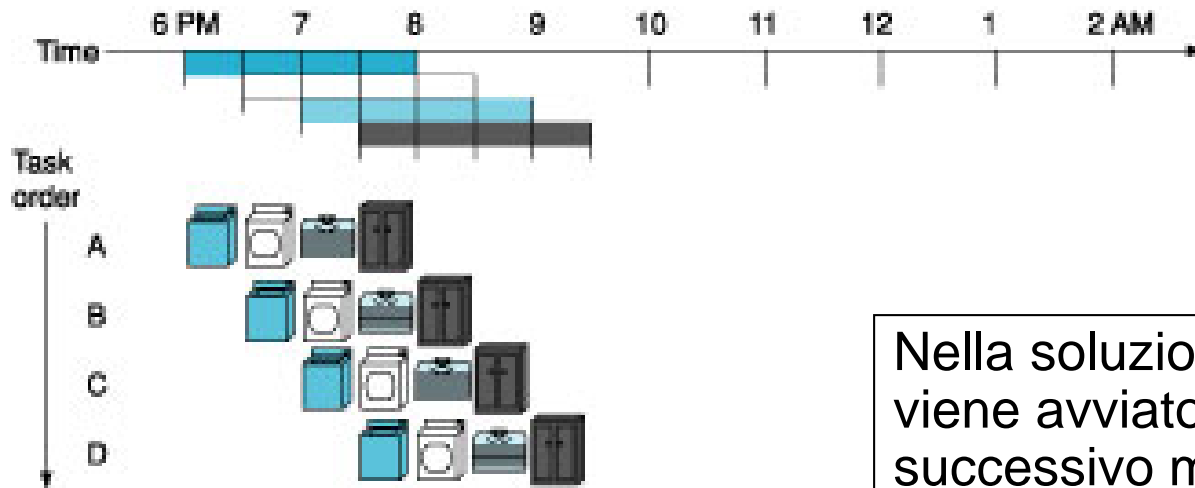
– Stiratura



– Riordino



Soluzione con pipeline



Nella soluzione con pipeline viene avviato il ciclo di lavaggio successivo mentre quello precedente è ancora in esecuzione in un'altra fase

Set istruzioni processore didattico a 32 bit

Tipi di istruzioni:

- logiche/aritmetiche su numeri interi;
- caricamento/memorizzazione;
- salto condizionato;
- non operativa.

Istruzioni logiche/aritmetiche (istruzioni di tipo L/A)

Istruzione	Sintassi	Semantica
Somma	add regsorg1, regsorg2, regdest	$(\text{regdest}) = (\text{regsorg1}) + (\text{regsorg2})$
Sottrazione	sub regsorg1, regsorg2, regdest	$(\text{regdest}) = (\text{regsorg1}) - (\text{regsorg2})$
Prod. Logico	and regsorg1, regsorg2, regdest	$(\text{regdest}) = (\text{regsorg1}) \text{ and } (\text{regsorg2})$
Som. Logica	or regsorg1, regsorg2, regdest	$(\text{regdest}) = (\text{regsorg1}) \text{ or } (\text{regsorg2})$
Neg. logica	not regsorg1, regdest	$(\text{regdest}) = \text{not } (\text{regsorg1})$

Formato istruzioni

31-26	25-21	20-16	15-11	10-0
opcode	regsorg1	regsorg2	regdestL/A	non utilizzati

Esempi di codice

add 2, 3, 1 -- somma il contenuto dei registri 2 e 3, il risultato mettilo nel registro 1

000001	00010	00011	00001	-----
--------	-------	-------	-------	-------

sub 5, 6, 4 -- sottrai il contenuto del registro 5 con quello di 6, il risultato mettilo nel registro 4

000010	00101	00110	00100	-----
--------	-------	-------	-------	-------

and 2, 3, 1 -- fai l'and tra il contenuto dei registri 2 e 3, il risultato mettilo nel registro 1

000011	00010	00011	00001	-----
--------	-------	-------	-------	-------

Istruzioni caricamento/memorizzazione (istruzioni di tipo C/M)

Istruzione	Sintassi	Semantica
Caricamento di parola	load regdest, offset(regbase)	(regdest) = memoria[offset+(regbase)]
Memorizzazione di parola	store regsorgM, offset(regbase)	memoria[offset+(regbase)] = (regsorg)

Formato istruzioni load ed esempio di codice

31-26	25-21	20-16	15-0
opcode	regbase	regdestC	Offset

Load 1, 32(3) --

trasferisci il contenuto della locazione di memoria il cui indirizzo è dato dalla somma di 32 con il contenuto del registro .

000110	00101	00001	0000000000100000
--------	-------	-------	------------------

Formato istruzioni store ed esempio di codice

31-26	25-21	20-16	15-0
opcode	regbase	regsorgM	Offset

Store 7, 16(4) --

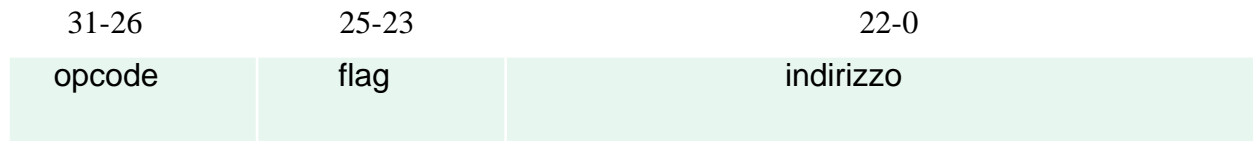
trasferisci il contenuto del registro 7 nella locazione di memoria il cui indirizzo è dato dalla somma di 16 con il contenuto del registro 4.

000111	00100	00111	00000000000010000
--------	-------	-------	-------------------

Istruzioni di salto condizionato (istruzioni di tipo S)

Istruzione	Sintassi	Semantica
salta se flag X == 1 (dove X può essere uno dei flag del registro di stato)	jumpX indirizzo	se flag X == 1 allora $PC = PC + S + \text{indirizzo}$ (dove S è un intero che dipende da come è organizzata la memoria, per esempio in un processore a 32 bit con il banco di memoria della cache costituita con moduli di memoria di 8 bit il suo valore è pari a 4)

Formato ed esempio di codice



jumpC 1026 --

se il bit di CARRY == 1 allora metti il PC ad un valore pari a (PC)+S+1026.

001000	001	00000000000001000000010
--------	-----	-------------------------

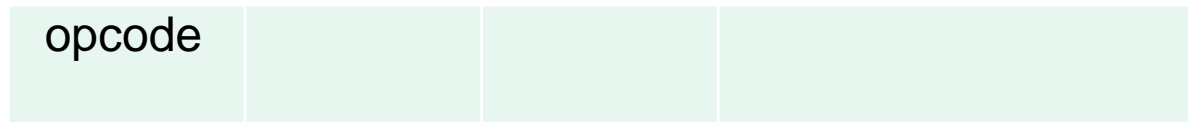
Istruzione non operativa (NOP)

Istruzione	Sintassi	Semantica
Nulla	Nop	Non modificare nulla

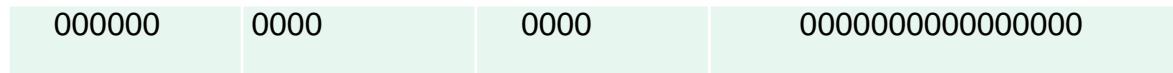
Formato e esempio di codice

31-26

25-0



nop -- non fare nulla



Considerazioni sul formato

- il campo codice-operativo è sempre contenuto nei bit 31-26;
- i due registri sorgente delle operazioni **logiche/aritmetiche** sono specificati, rispettivamente, nei bit 25-21 e 20-16, nella stessa posizione troviamo specificato anche il registro base e registro sorgente dell'istruzione **store**, mentre il registro base dell'istruzione **load** è specificato nei bit 25-21; questa disposizione semplificherà la progettazione del data path, in quanto, come si vedrà, sarà possibile accedere al banco dei registri per prelevare gli operandi indipendentemente dal tipo di istruzione;
- il registro destinazione delle operazioni **logiche/aritmetiche** è specificato nei bit 15-11, mentre quello dell'istruzione **load** nei bit 20-16; questo disallineamento purtroppo complicherà un po' l'architettura del data path;
- l'offset sia nelle istruzioni **load** che **store** è memorizzato nei bit 15-0.

Fasi esecuzione istruzioni logiche/aritmetiche

- prelevare l'istruzione dalla memoria ed incrementare di S il PC (**FETCH**);
- decodificare l'istruzione e leggere il contenuto dei registri sorgente (**DECODE**);
- effettuare l'operazione logica o aritmetica specificata (**EXECUTE**);
- memorizzare nel registro destinazione il risultato dell'operazione (**WRITE BACK**).

Fasi esecuzione istruzione di load

- prelevare l'istruzione dalla memoria ed incrementare di S il PC (**FETCH**);
- decodificare l'istruzione e leggere il contenuto del registro base (**DECODE**);
- calcolare l'indirizzo della locazione di memoria da cui prelevare il dato (**EXECUTE**);
- leggere il contenuto della locazione di memoria in cui è memorizzato il dato (**MEMORY**);
- memorizzare nel registro destinazione ciò che è stato letto dalla memoria (**WRITE BACK**).

Fasi esecuzione istruzione di store

- prelevare l'istruzione dalla memoria ed incrementare di S il PC (**FETCH**);
- decodificare l'istruzione e leggere il contenuto del registro sorgente del dato da memorizzare e il contenuto del registro base (**DECODE**);
- calcolare l'indirizzo della locazione di memoria in cui scrivere il dato letto dal registro sorgente (**MEMORY**);
- memorizzare nella locazione di memoria ciò che è stato letto dal registro sorgente (**WRITE BACK**).

Fasi di esecuzione di una istruzione di salto condizionato

- prelevare l'istruzione dalla memoria ed incrementare di S il PC (**FETCH**);
- decodificare l'istruzione (**DECODE**);
- calcolare l'indirizzo della locazione di memoria da cui prelevare l'istruzione successiva nel caso in cui il flag di stato selezionato è pari ad 1 (**EXECUTE**);
- dipendendo dal valore del flag selezionato aggiornare il PC o lasciarlo invariato.

Fasi di esecuzione di una NOP

- prelevare l'istruzione dalla memoria ed incrementare di S il PC (**FETCH**);
- decodificare l'istruzione (**DECODE**).

Esecuzione delle istruzioni nel processore con pipeline

F Instruction Fetch	D Instruction Decode	E Execute	M Memory access	WB Write-Back
-------------------------------	--------------------------------	---------------------	---------------------------	-------------------------

• Istruzioni logico-aritmetiche

F Prel. istr. e incr. PC	D Lettura reg. sorgente	E Op. ALU su dati letti		WB Scrittura reg. dest.
------------------------------------	-----------------------------------	-----------------------------------	--	-----------------------------------

• Istruzioni di load

I Prel. istr. e incr. PC	D Lettura reg. base	E Somma	M Prelievo dato da M	WB Scrittura reg. dest.
------------------------------------	-------------------------------	-------------------	--------------------------------	-----------------------------------

• Istruzioni di store

F Prel. istr. e incr. PC	D Lettura reg. base e reg sorg M	E Somma	M Scrittura dato in M	
------------------------------------	---	-------------------	---------------------------------	--

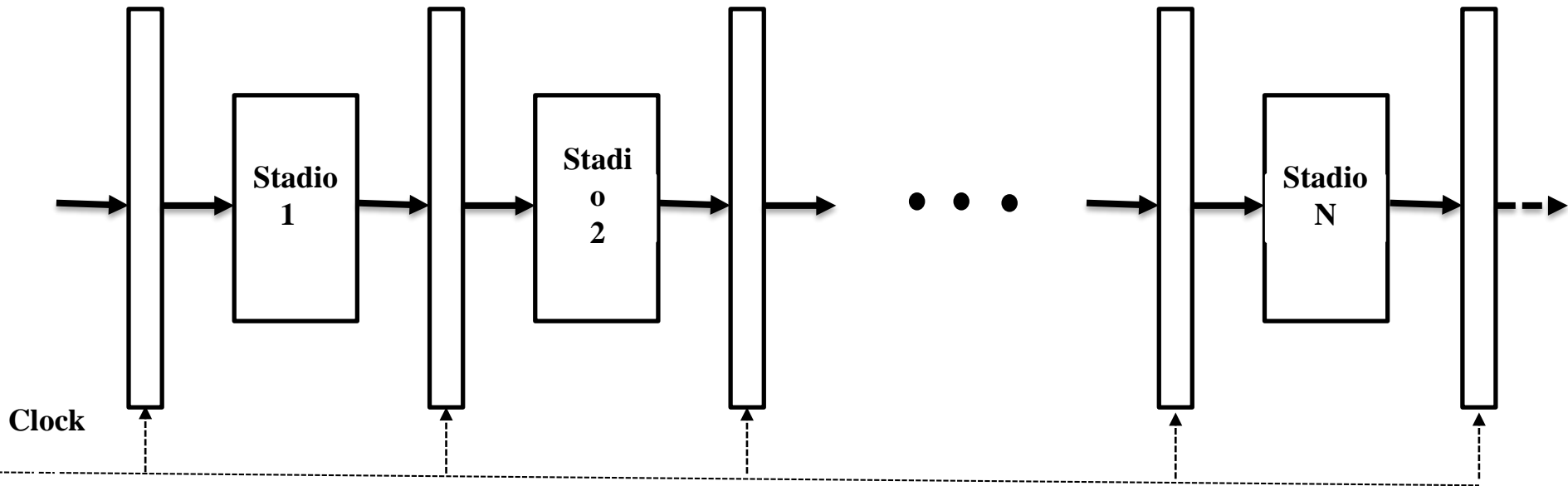
• Istruzioni di jump

F Prel. istr. e incr. PC	D Decodifica	E Somma	M Scrittura PC	
------------------------------------	------------------------	-------------------	--------------------------	--

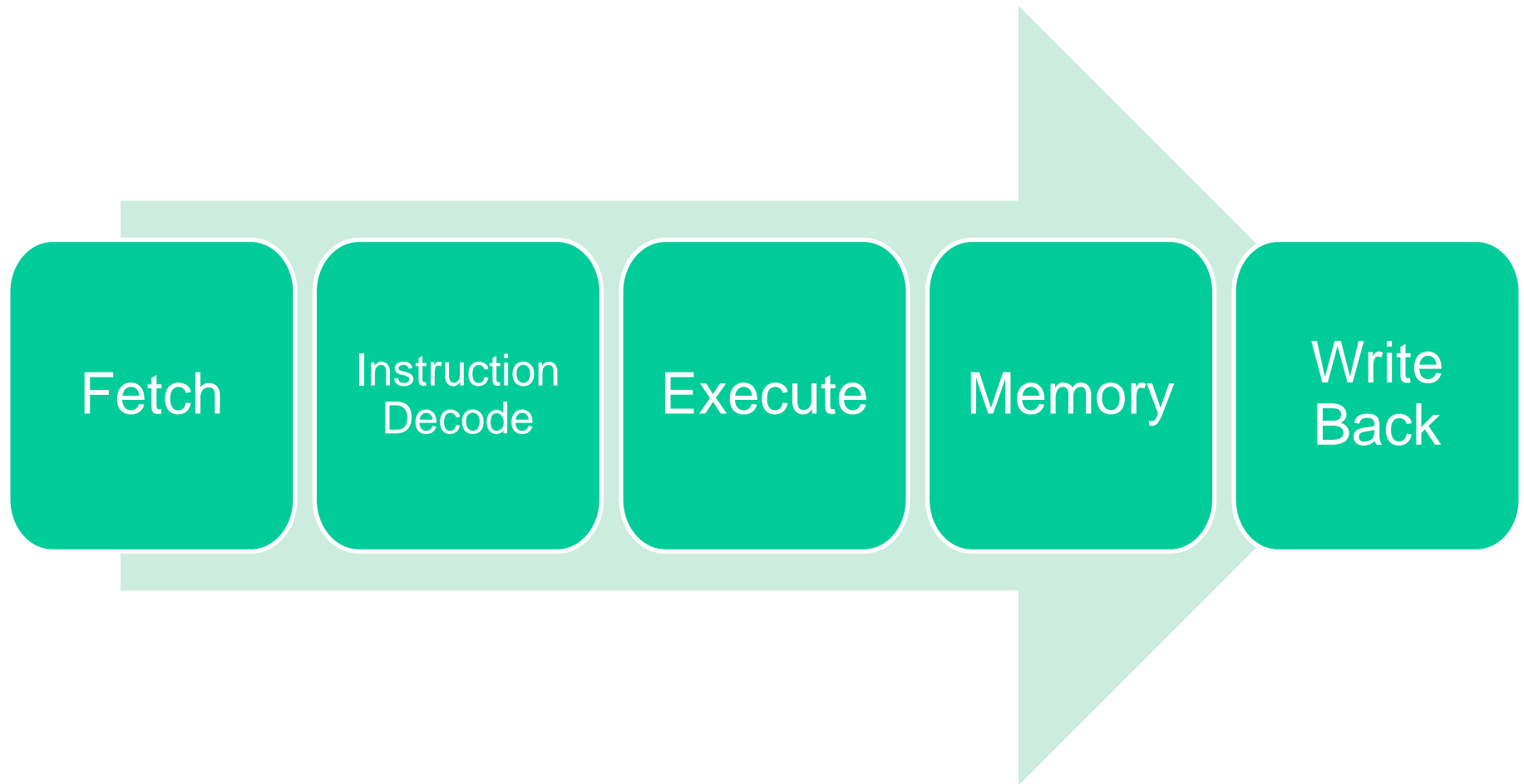
Come progettare l'unità di elaborazione?

- La suddivisione dell'istruzione in 5 stadi implica che in ogni ciclo di clock siano in esecuzione 5 istruzioni
 - La struttura di un processore con pipeline a 5 stadi deve essere scomposta in 5 parti (o *stadi di esecuzione*), ciascuna della quali corrispondente ad una delle fasi della pipeline
- Occorre introdurre una separazione tra i vari stadi
 - *Registri di pipeline*
- Inoltre, diverse istruzioni in esecuzione nello stesso istante possono richiedono risorse hardware simili
 - Replicazione delle risorse hardware
- Riprendiamo lo schema dell'unità di elaborazione a ciclo singolo ed identifichiamo i 5 stadi

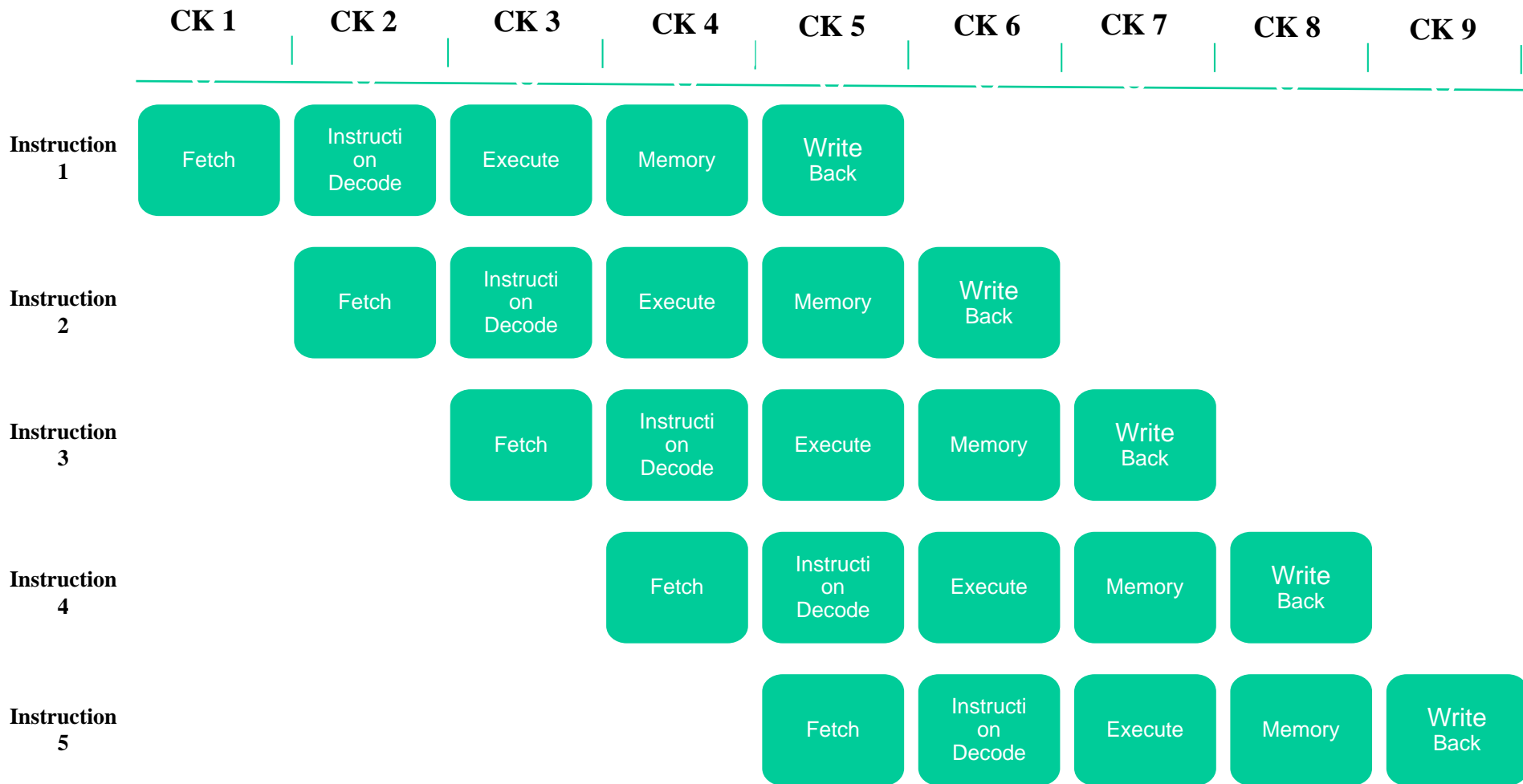
Schema di principio delle architetture pipeline



Sequenza passi per l'esecuzione di ogni istruzione



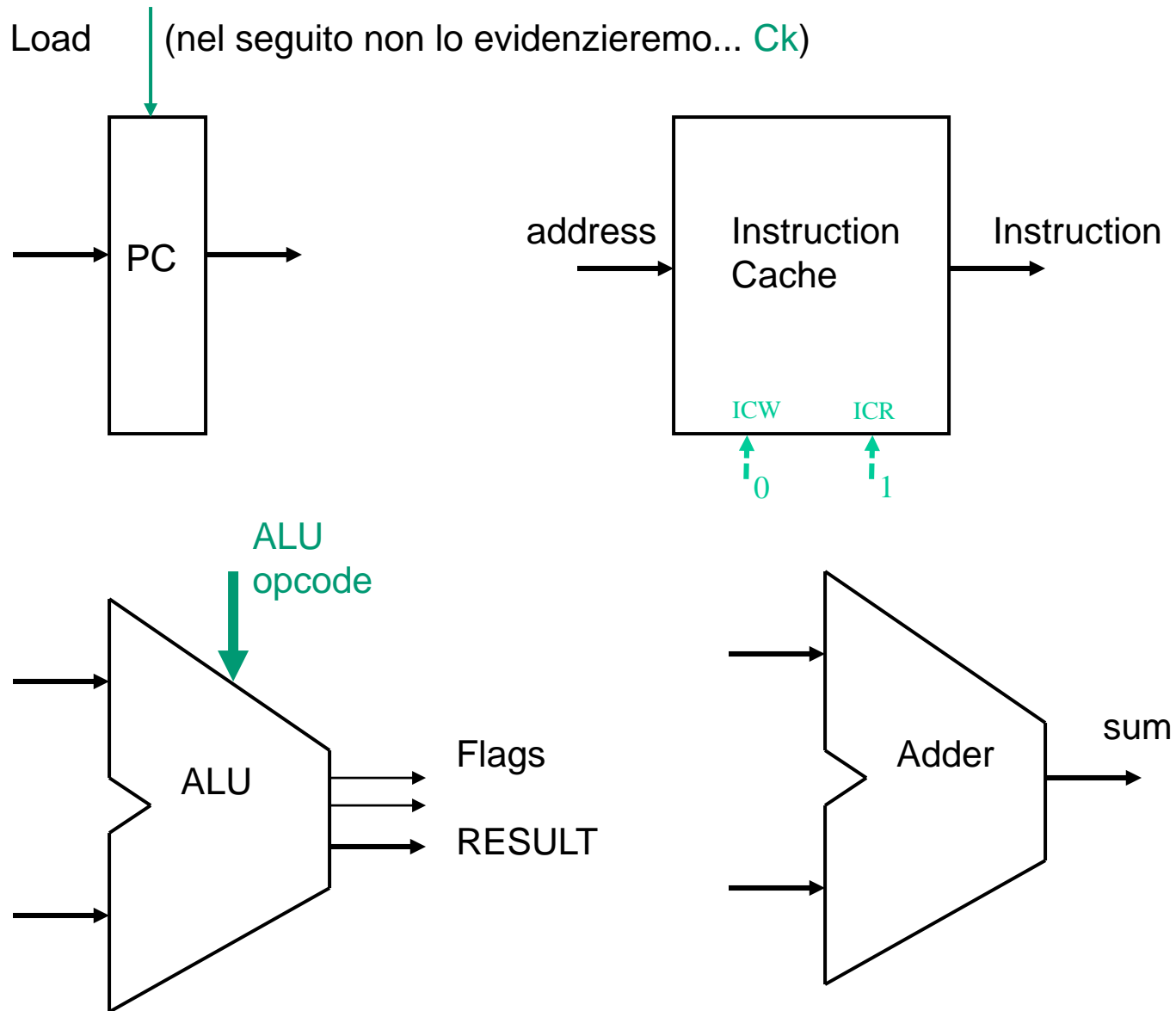
Schema sovrapposizione esecuzione di più istruzioni



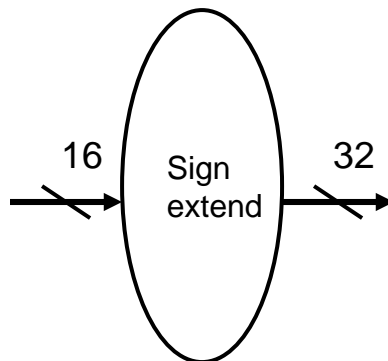
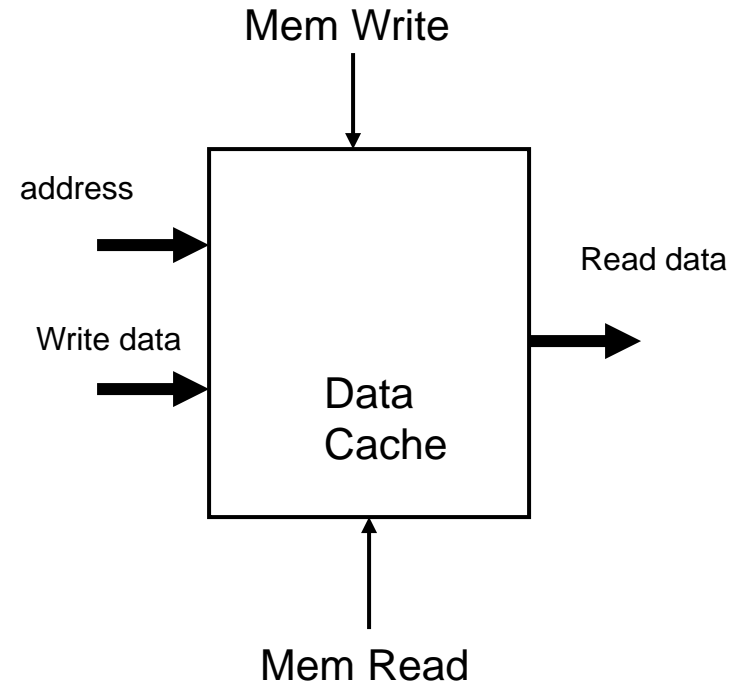
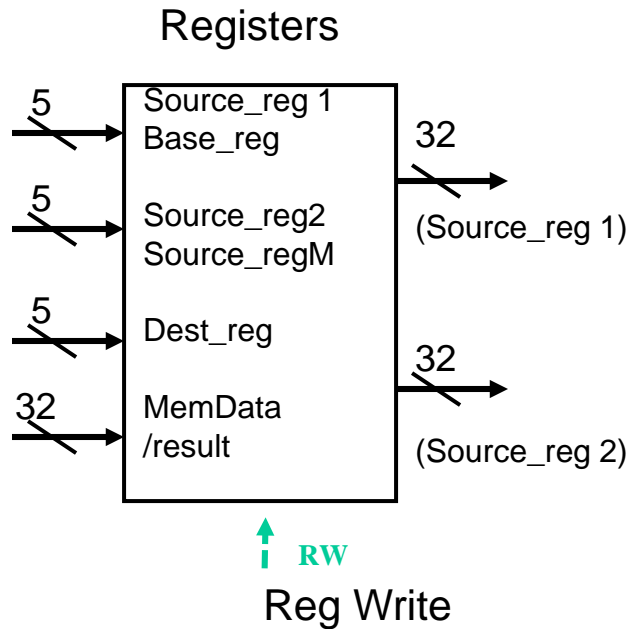
L'unità di elaborazione con pipeline

- Principio guida:
 - Permettere il riuso delle componenti per l'istruzione successiva
- Introduzione di *registri di pipeline* (registri interstadio)
 - Ad ogni ciclo di clock le informazioni procedono da un registro di pipeline a quello successivo
 - Il nome del registro è dato dal nome dei due stadi che separa
 - Registro **F/D** (Instruction Fetch / Instruction Decode)
 - Registro **D/E** (Instruction Decode / Execute)
 - Registro **E/M** (Execute / Memory access)
 - Registro **M/WB** (Memory access / Write Back)
 - Il PC può essere considerato come un registro di pipeline per lo stadio FETCH

Componenti di base (1)

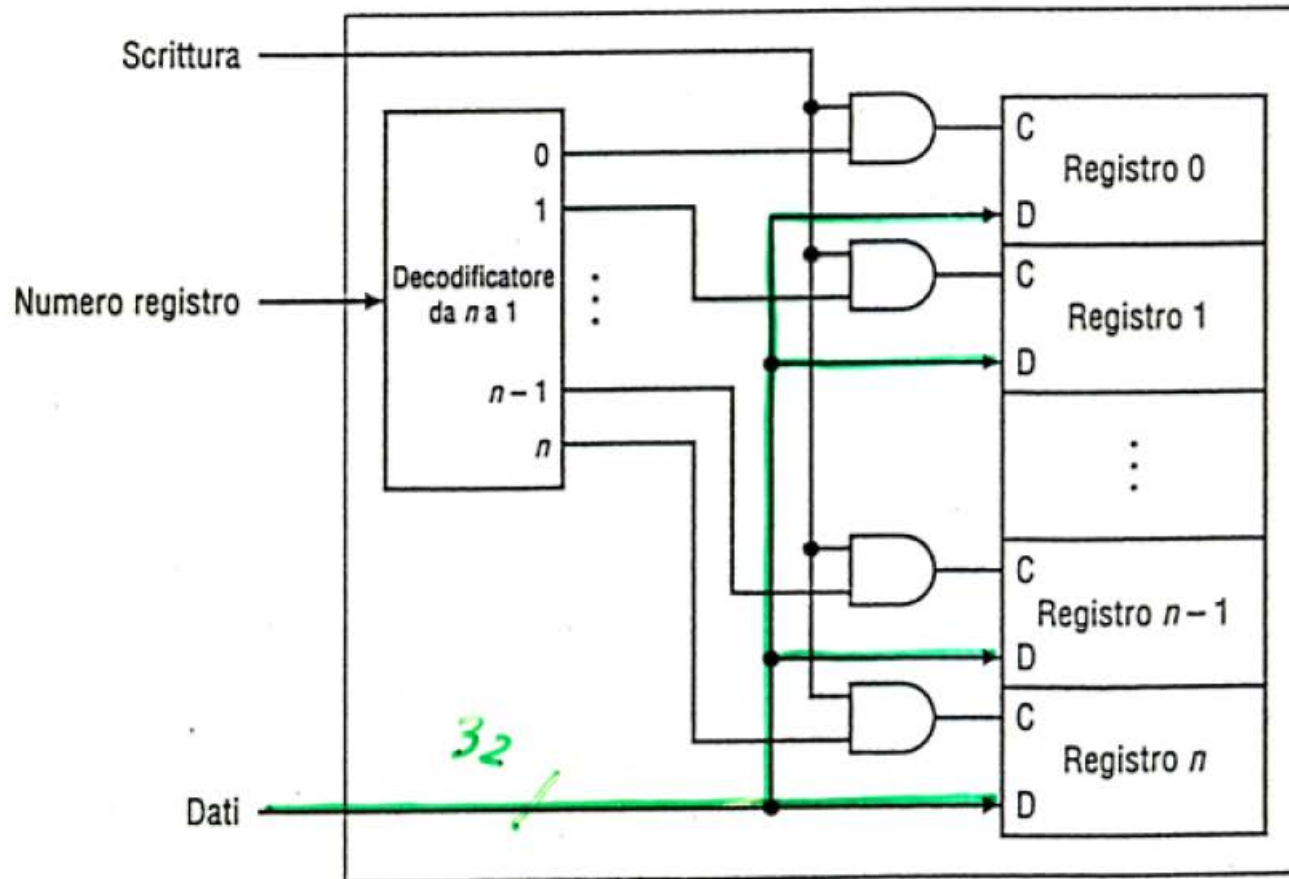


Componenti di base (2)



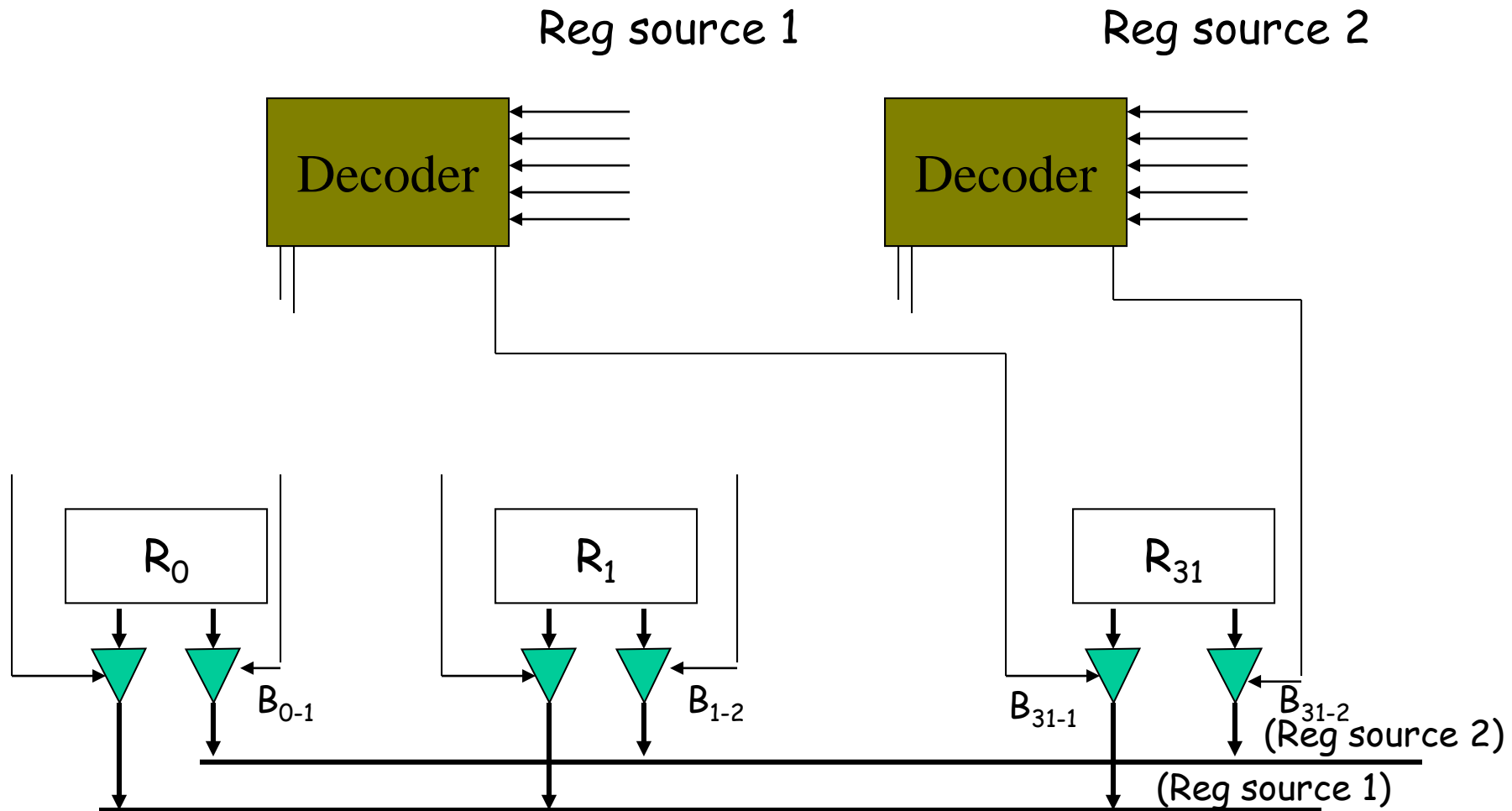
I registri commutano
sul fronte positivo
dei segnali di abilitazione
**Flip/flop positive/negative edge
triggered**

Banco dei registri (scrittura)

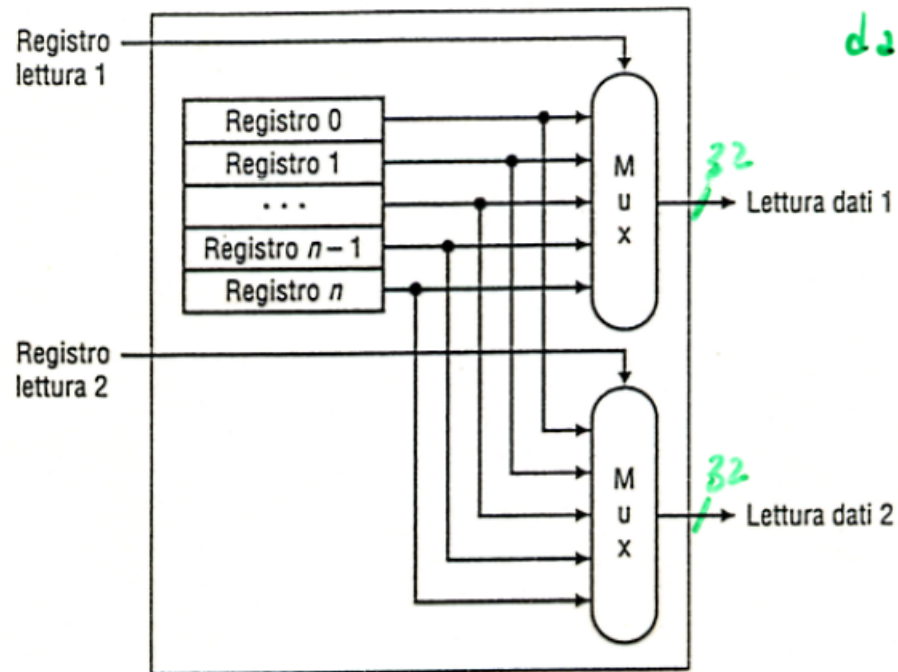


in ogni regis
ci sono 32 F
di tipo D

Banco dei registri (lettura) con decoder



Banco dei registri (lettura) con MUX



Sincronizzazione tra circuiti sequenziali

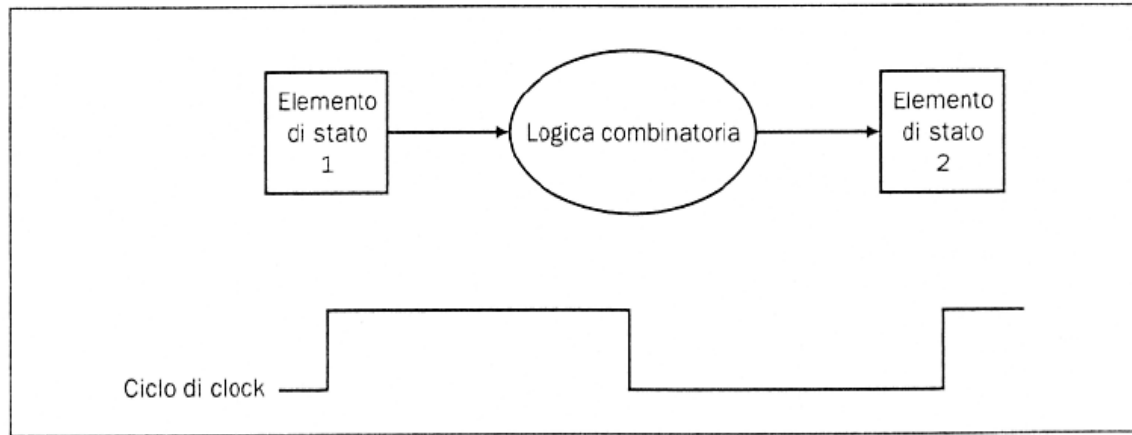


Figura 5.2 Logica combinatoria, elementi di stato e clock sono interdipendenti. In un sistema digitale sincrono il clock determina quando gli elementi di stato debbono scrivere i valori di ingresso nella loro memoria interna. Gli ingressi di un elemento di stato devono raggiungere uno stato stabile (ossia, devono aver raggiunto un valore destinato a non cambiare almeno fino al prossimo fronte del clock) prima che il fronte attivo del clock causi un aggiornamento dello stato; si assumerà che tutti gli elementi di stato, incluse le memorie, siano sensibili ai fronti.

I registri commutano
sul fronte positivo
dei segnali di abilitazione
Flip/flop positive edge triggered

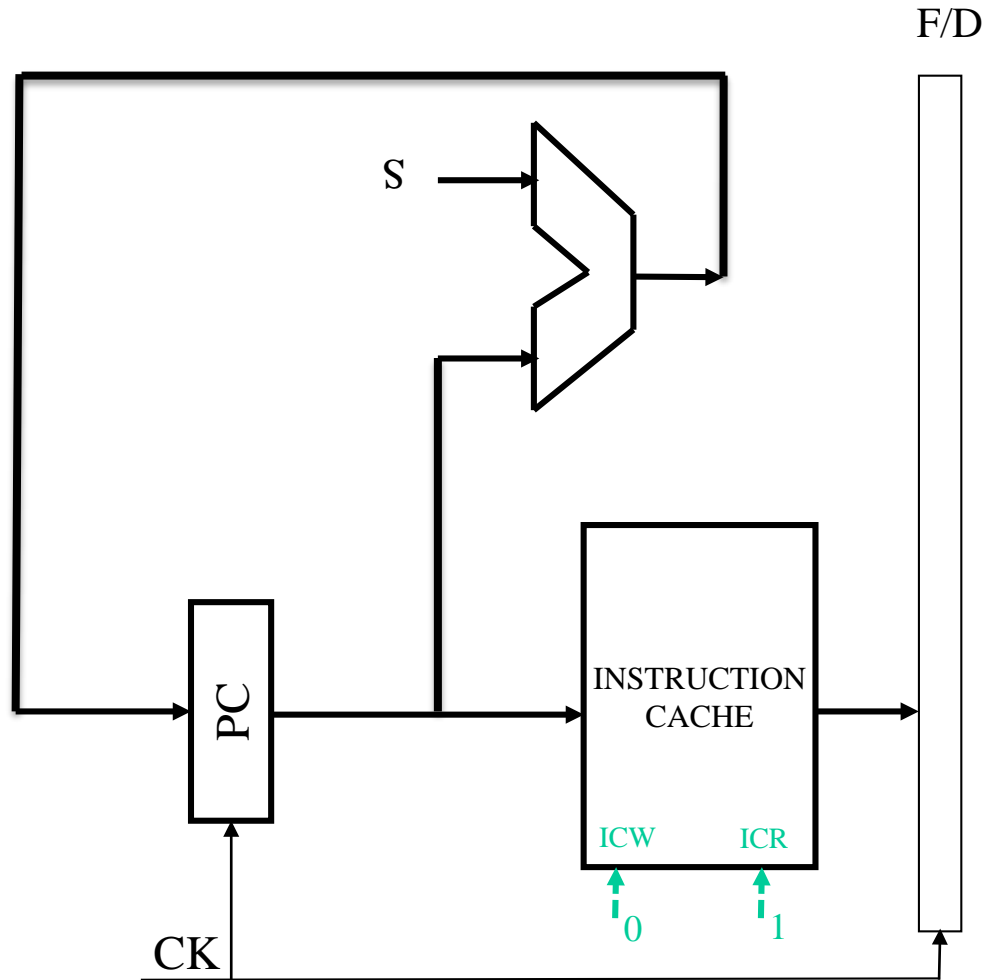
Passi logici per progettare il Data path

- Per ogni tipo di istruzione progettare l'hardware necessario per eseguirlo in ogni stadio
- Aggregare l'hardware di ogni stadio per ogni istruzione usando eventualmente aggiungendo dei mux

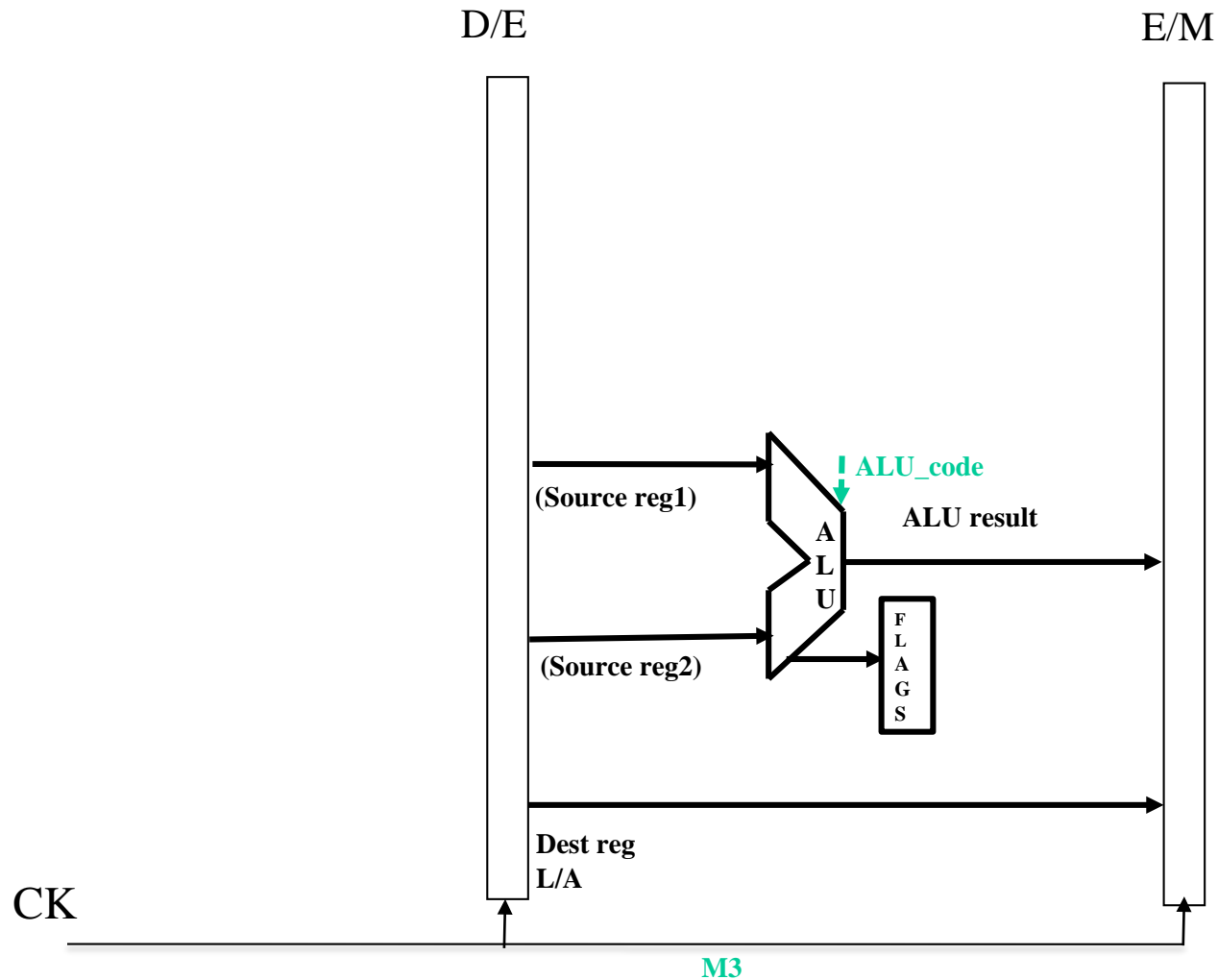
istruzioni di tipo L/A – formato

31-26	25-21	20-16	15-11	10-0
opcode	regsorg1	regsorg2	regdestL/A	non utilizzati

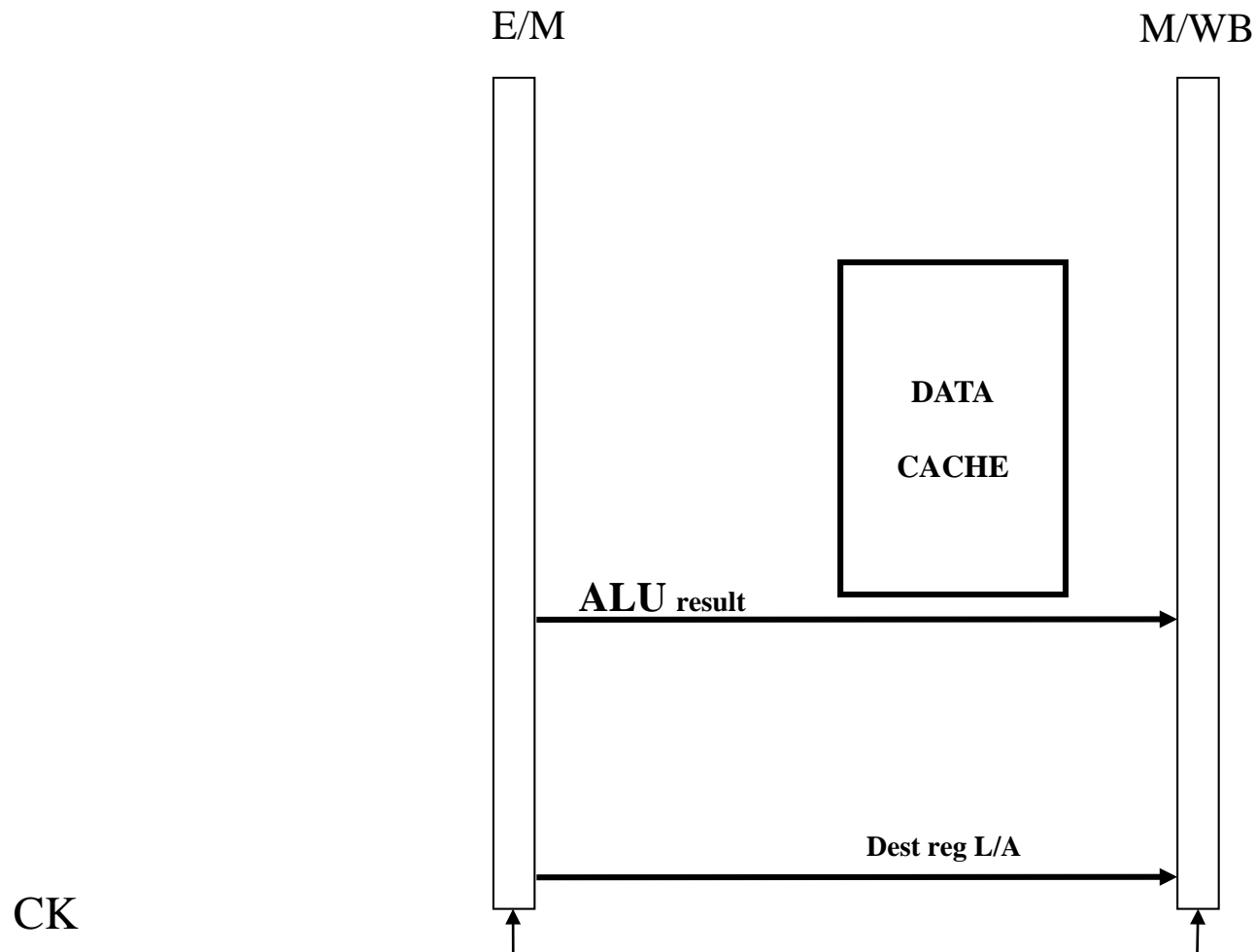
istruzioni di tipo L/A: Data path dello stadio di fetch



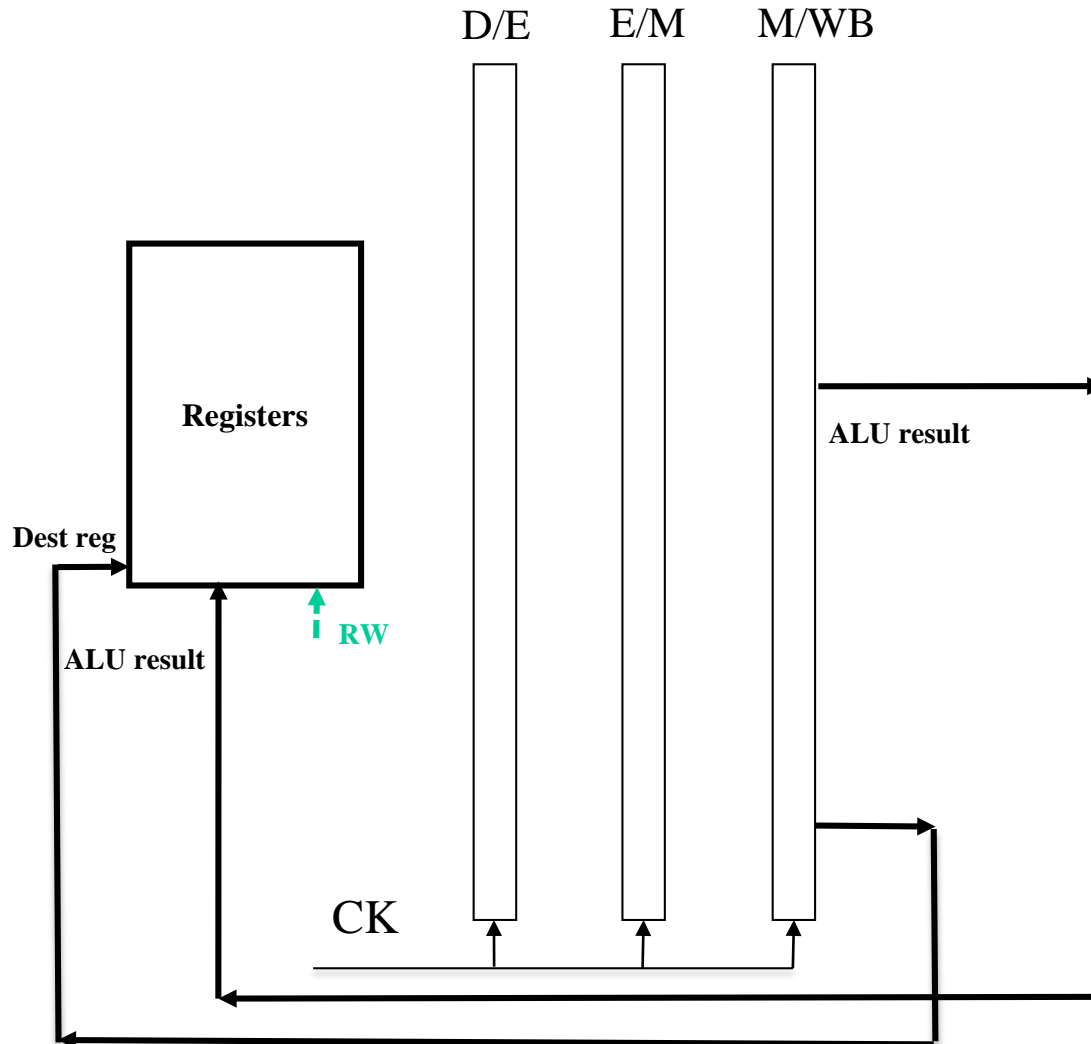
istruzioni di tipo L/A: Data path dello stadio di Execute



istruzioni di tipo L/A: Data path dello stadio di Memory



istruzioni di tipo L/A: Data path dello stadio di Write Back



Istruzioni caricamento/memorizzazione (istruzioni di tipo C/M)

Istruzione	Sintassi	Semantica
Caricamento di parola	load regdest, offset(regbase)	(regdest) = memoria[offset+(regbase)]
Memorizzazione di parola	store regsorgM, offset(regbase)	memoria[offset+(regbase)] = (regsorgM)

Formato istruzione STORE ed esempio di codice

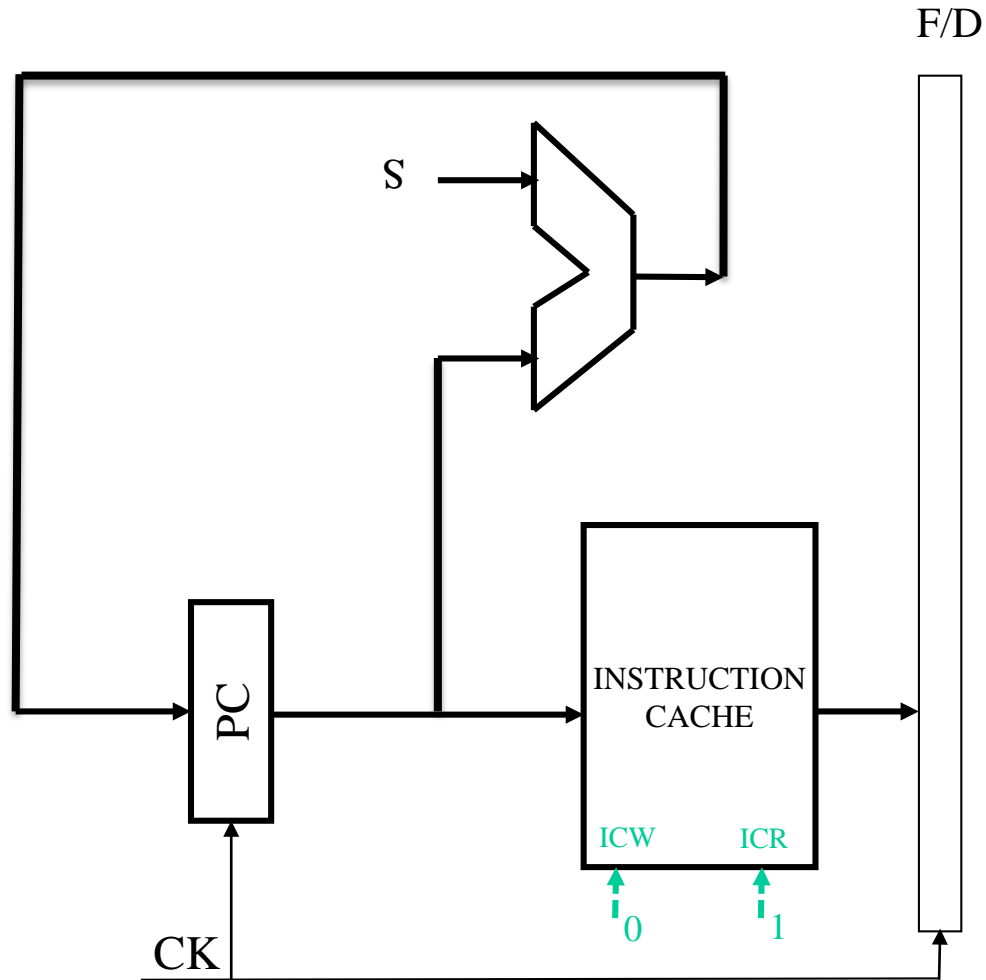
31-26	25-21	20-16	15-0
opcode	regbase	regsorgM	Offset

Store 7, 16(4) --

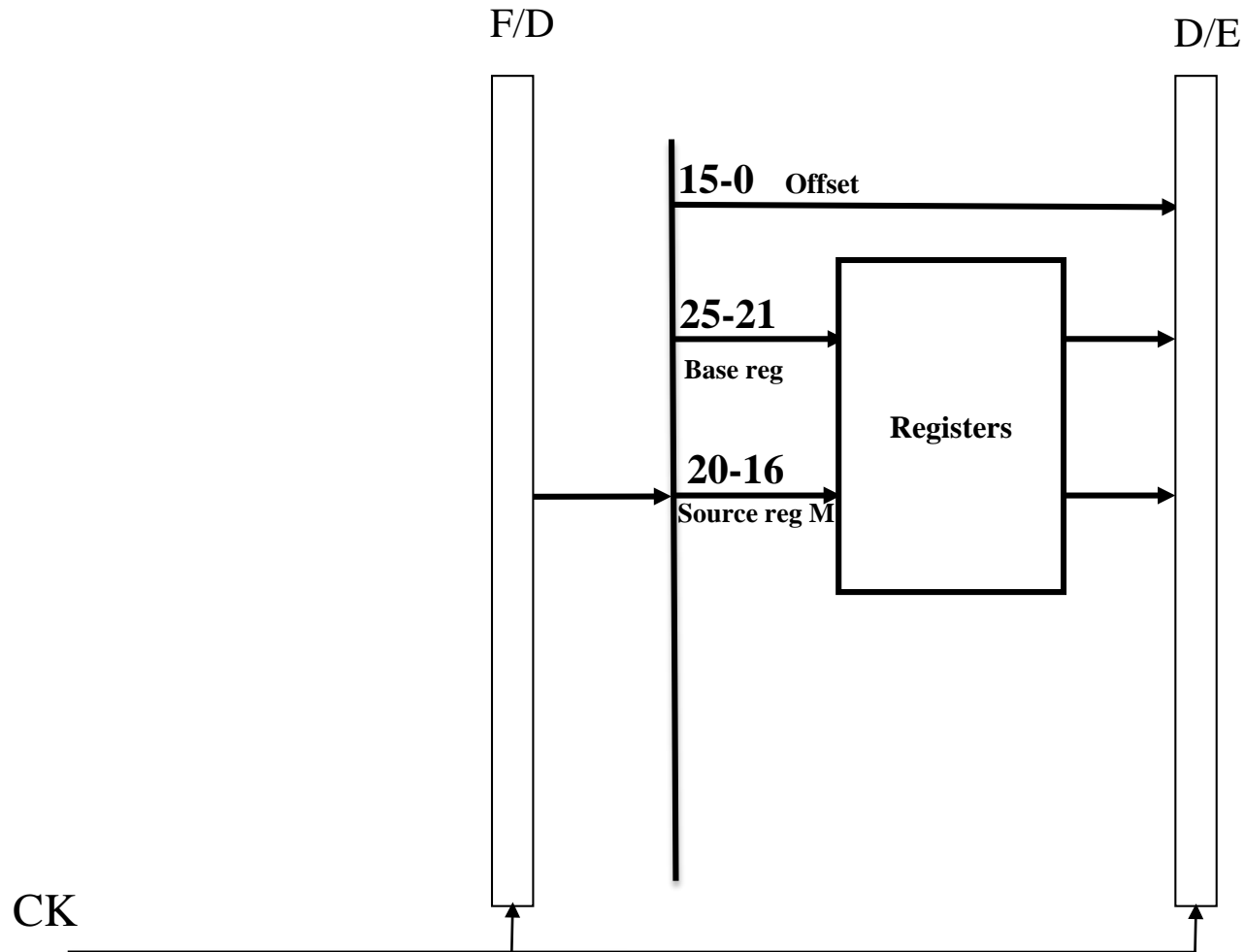
trasferisci il contenuto del registro 7 nella locazione di memoria il cui indirizzo è dato dalla somma di 16 con il contenuto del registro 4.

000111	00100	00111	0000000000010000
--------	-------	-------	------------------

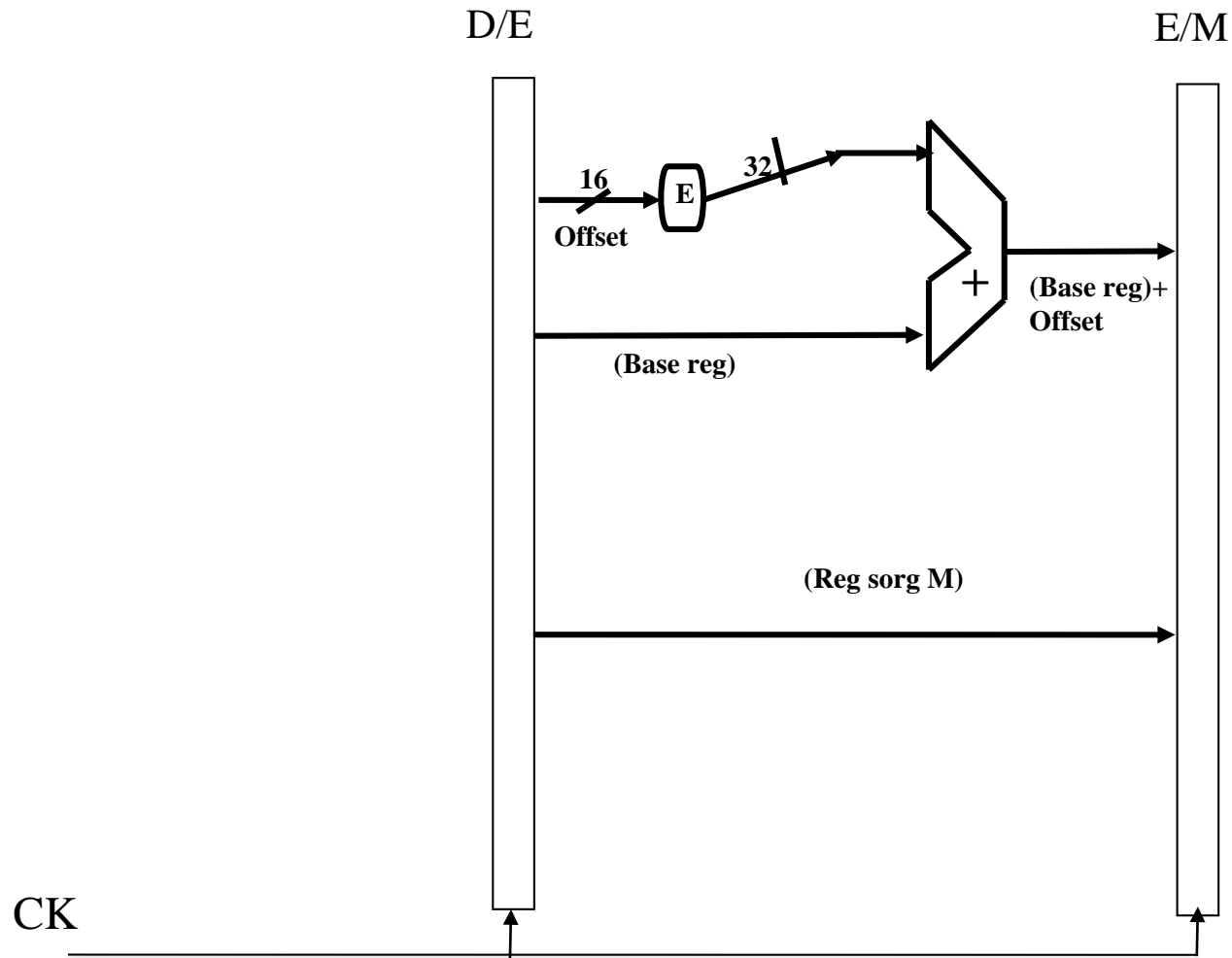
Store: Data path dello stadio di fetch



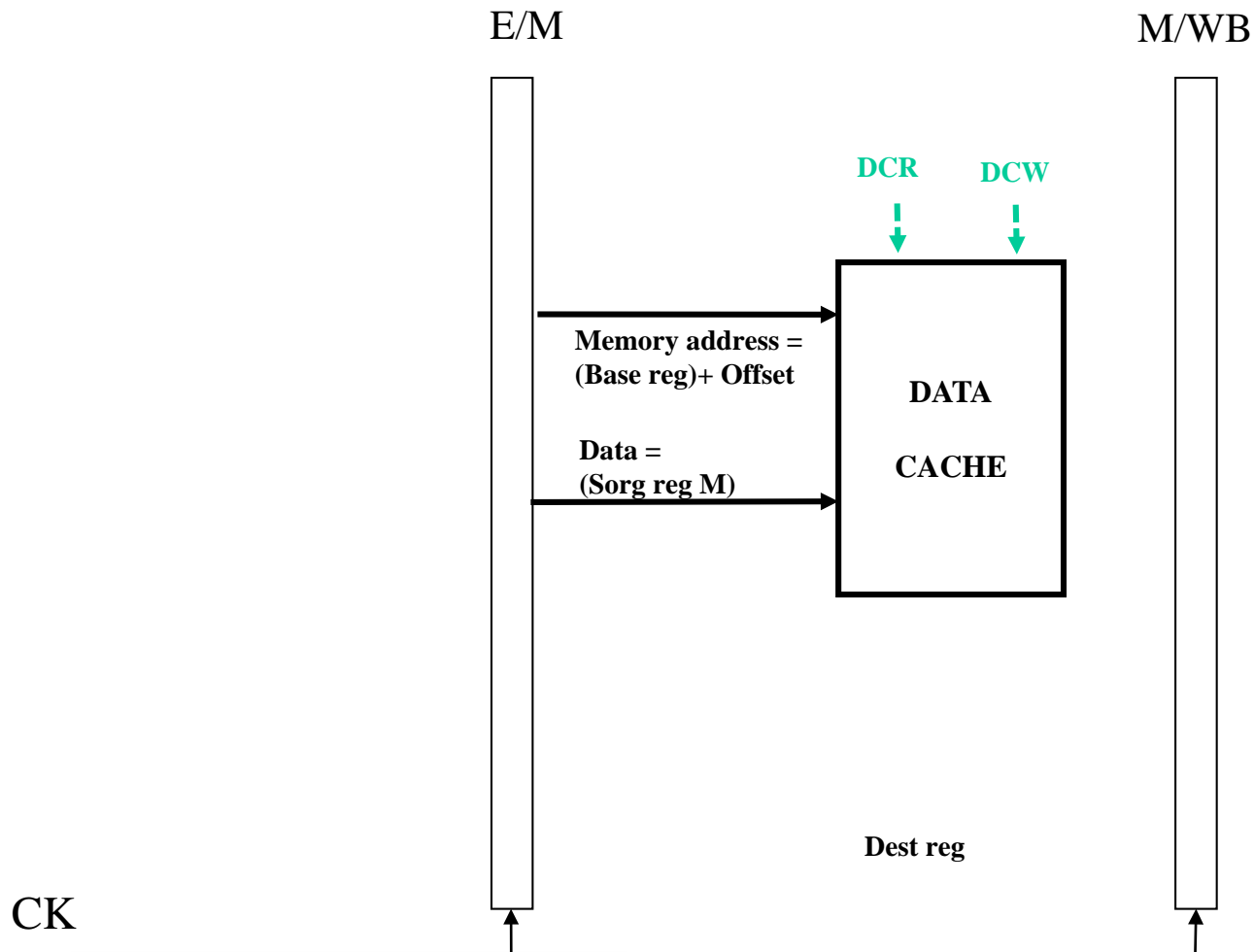
Store: Data path dello stadio Instruction Decode



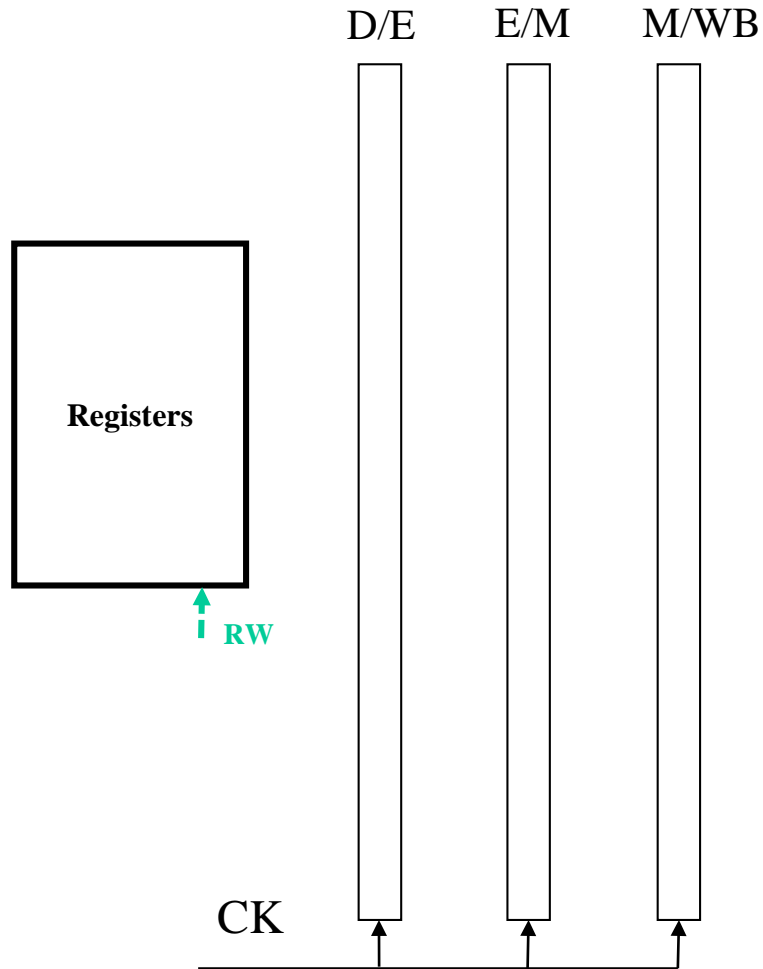
Store: Data path dello stadio di Execute



Store: Data path dello stadio di Memory

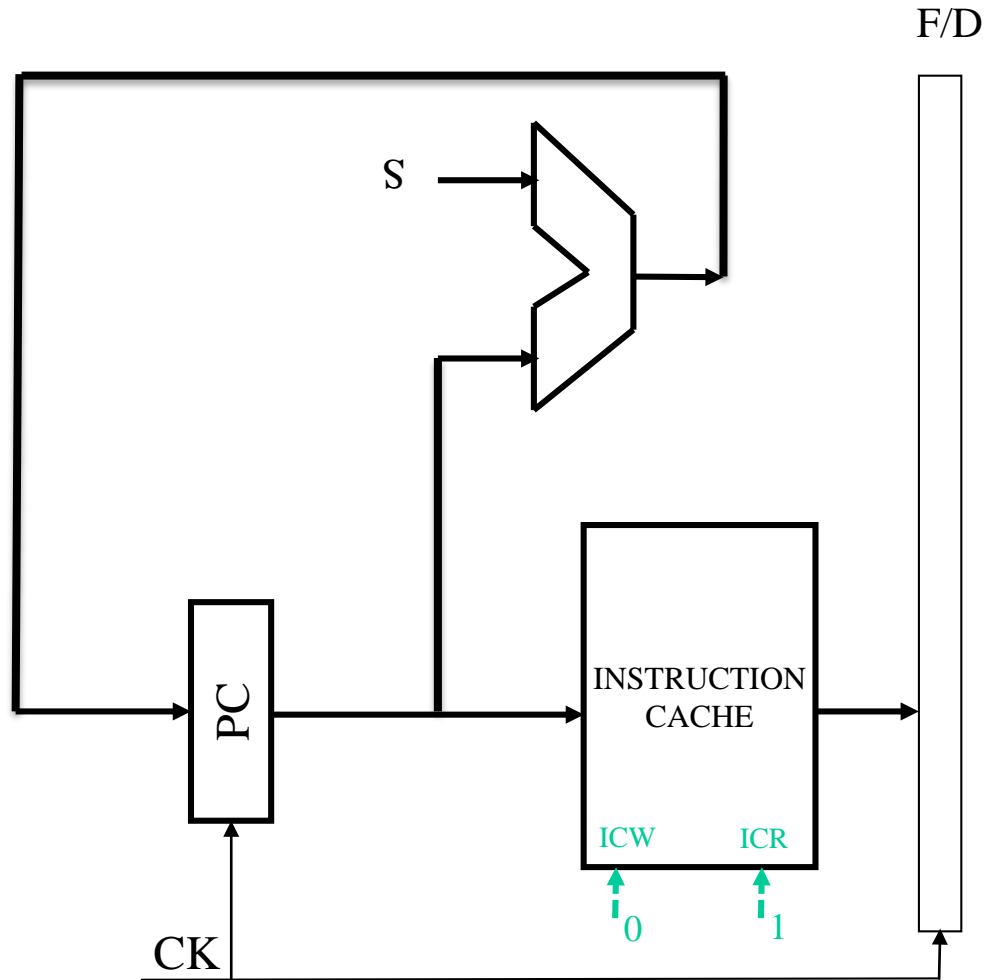


Store: Data path dello stadio di Write Back

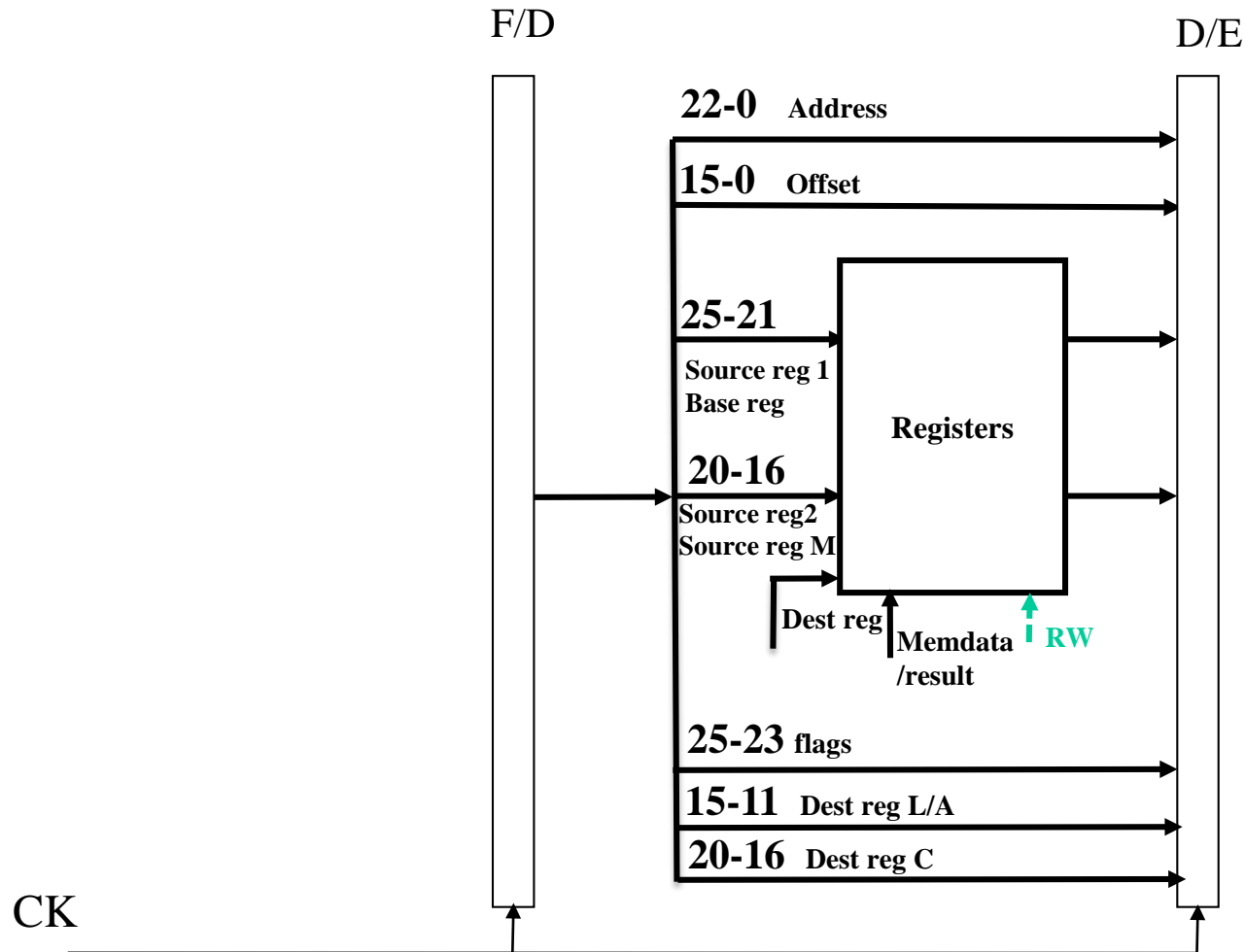


Mettendo insieme tutti i data path

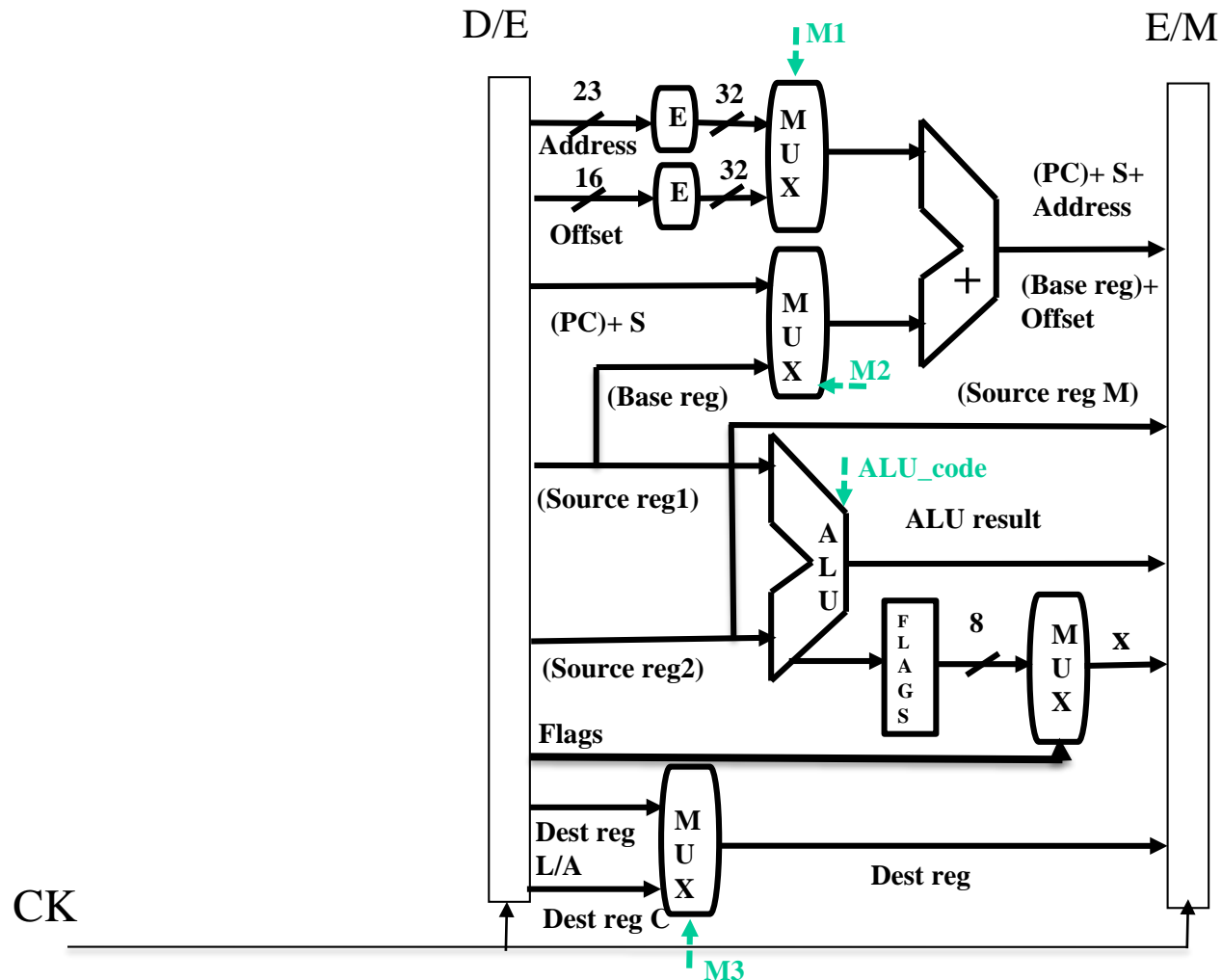
Data path dello stadio di fetch



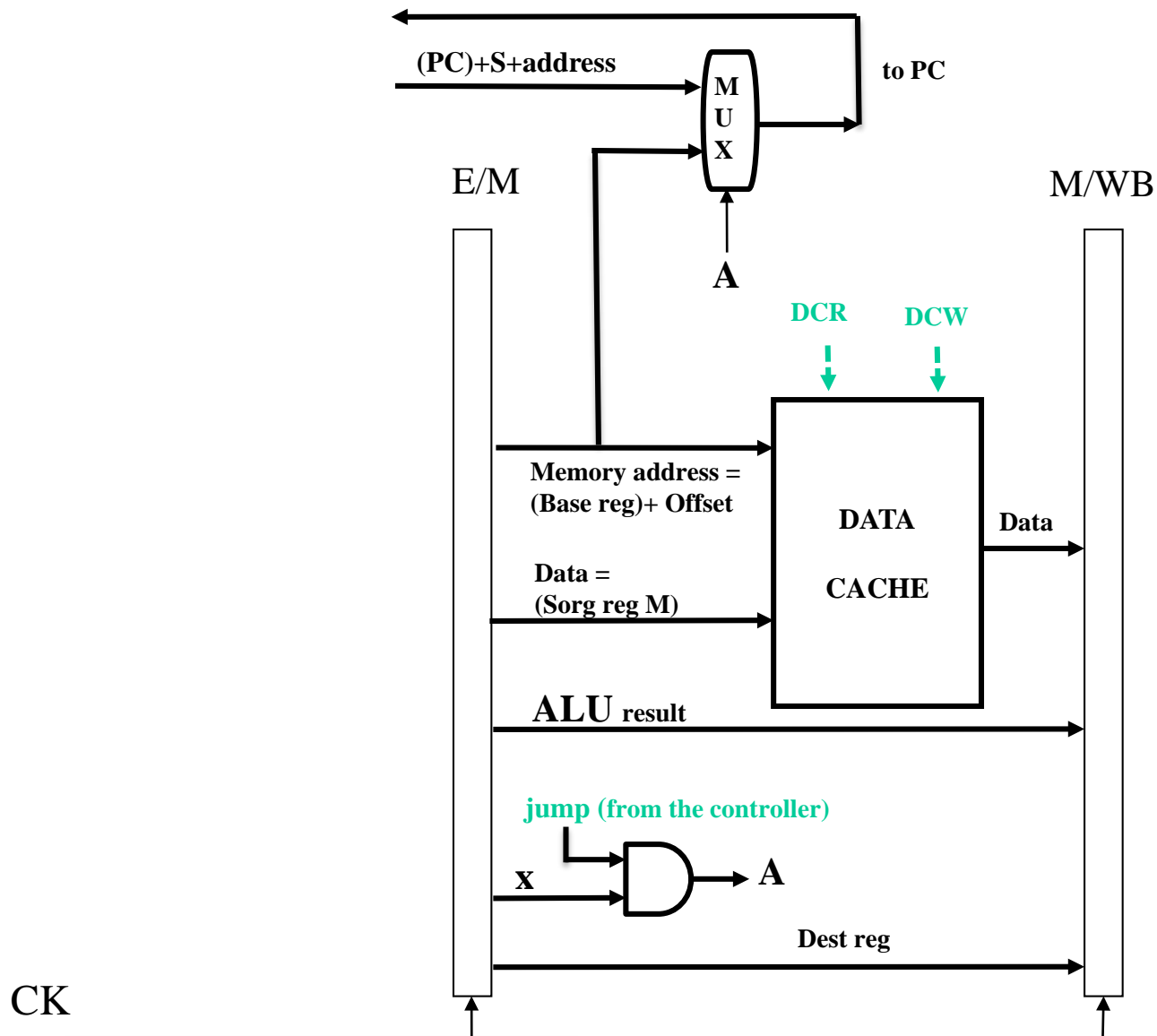
Data path dello stadio Instruction Decode



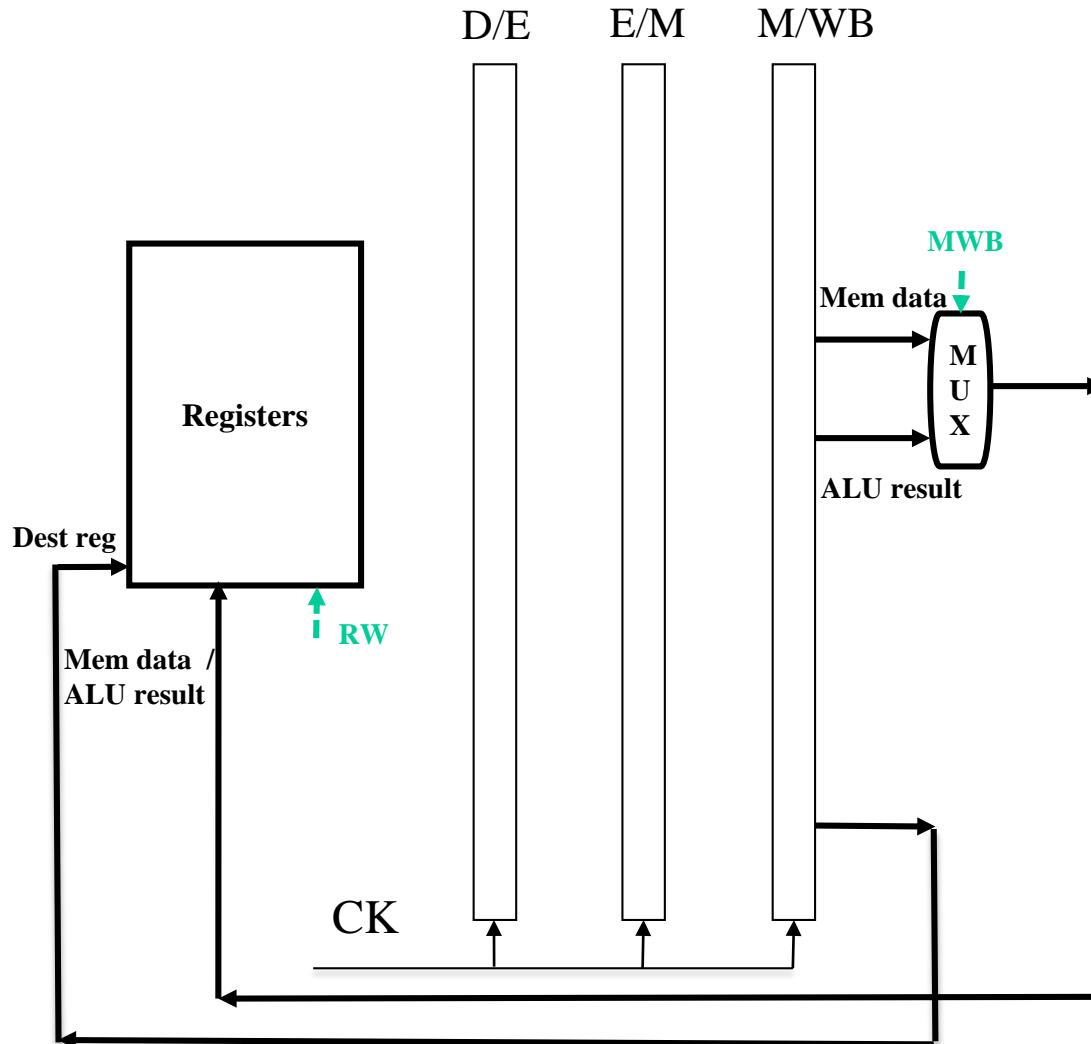
Data path dello stadio di Execute



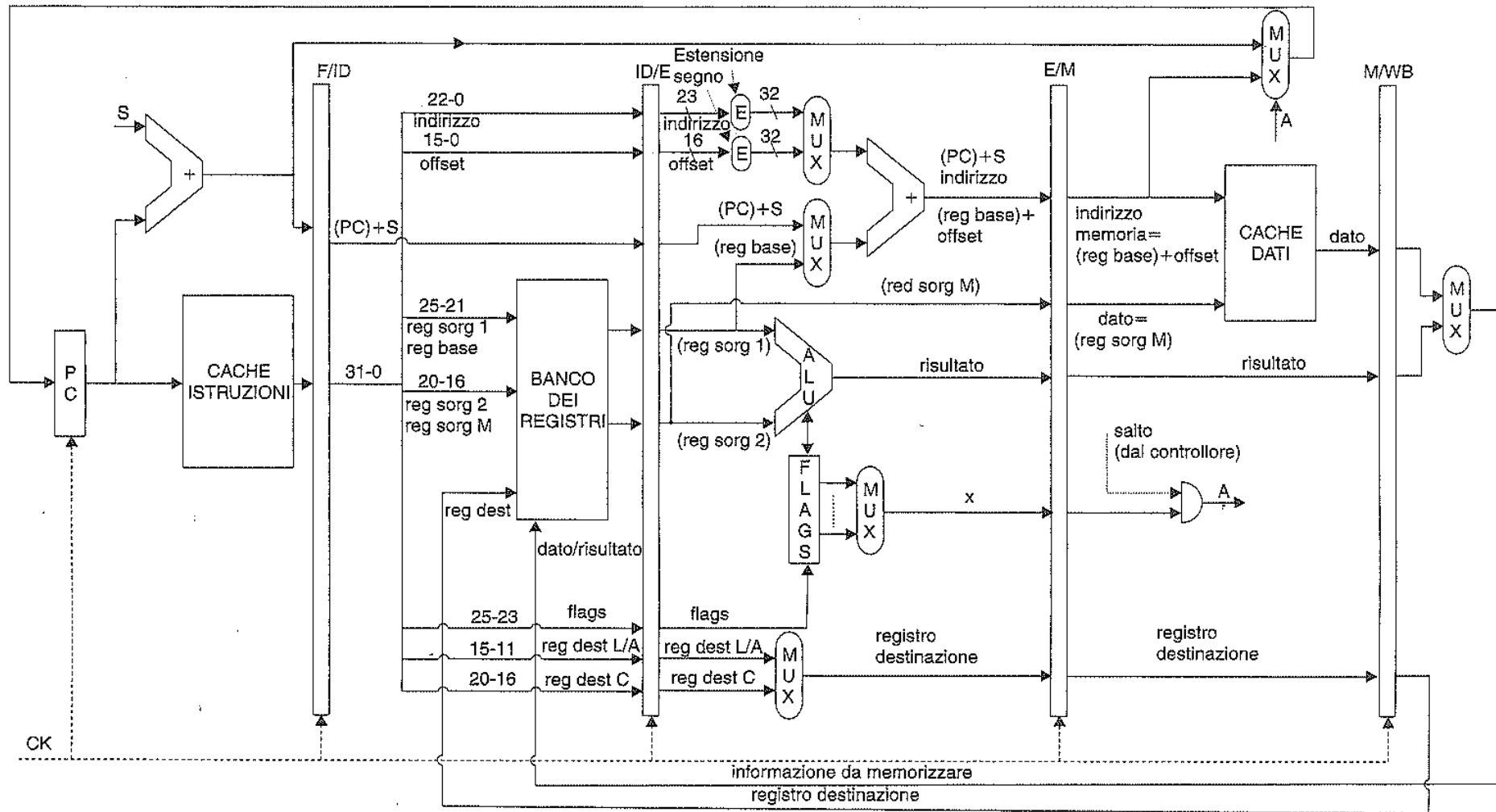
Data path dello stadio di Memory



Data path dello stadio di Write Back



Architettura complessiva



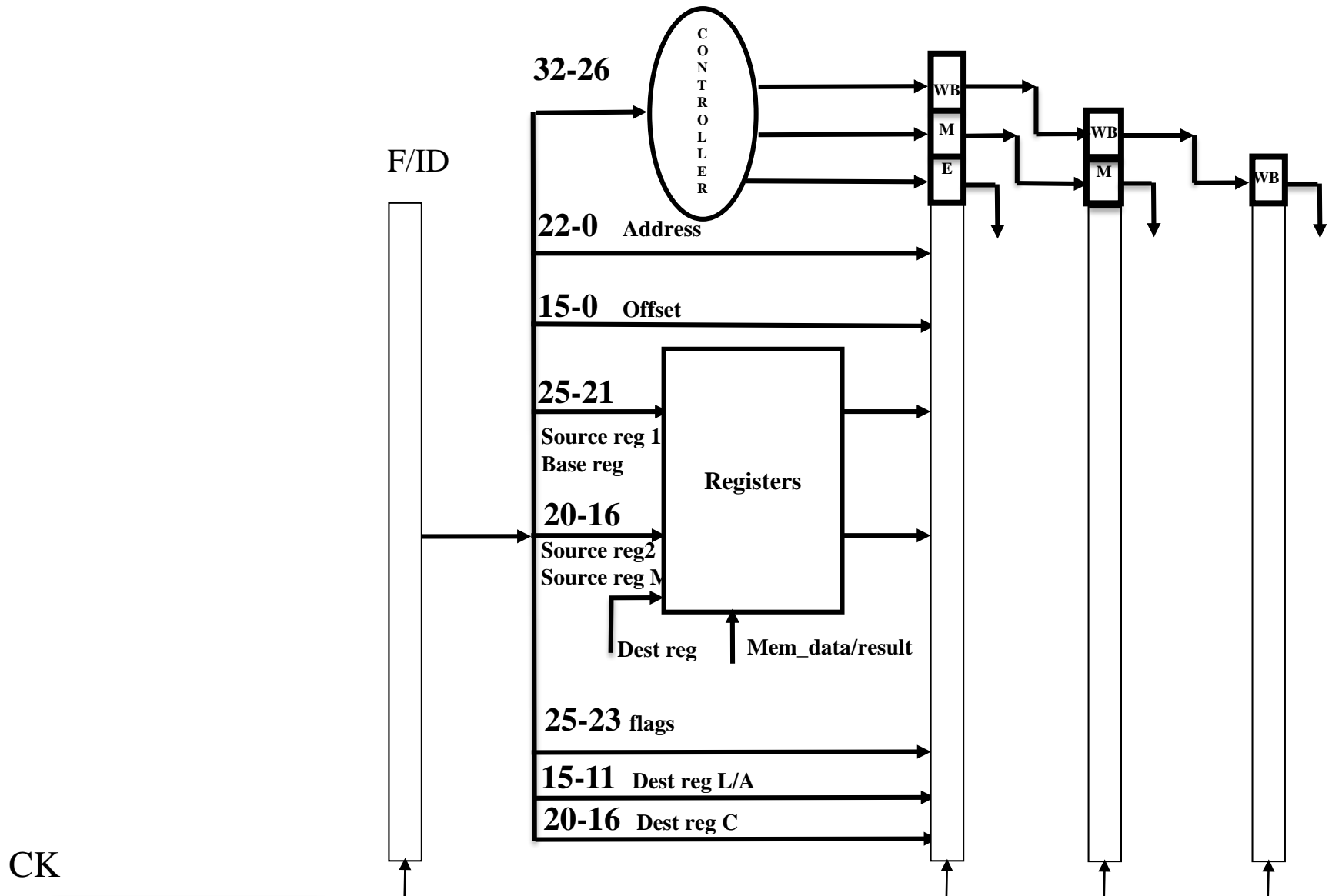
Come vengono eseguite le istruzioni

- Come viene eseguita un'istruzione nei vari stadi della pipeline?
- Consideriamo per prima l'istruzione lw
 - Prelievo dell'istruzione
 - Decodifica dell'istruzione e lettura dei registri
 - Esecuzione (uso dell'ALU per il calcolo dell'indirizzo)
 - Lettura dalla memoria
 - Scrittura nel registro
- Analizziamo poi l'esecuzione dell'istruzione sw
- Infine consideriamo l'esecuzione contemporanea di più istruzioni

Controllore dell'architettura pipeline

- I dati viaggiano attraverso gli stadi della pipeline
- Tutti i dati appartenenti ad un'istruzione devono essere mantenuti all'interno dello stesso stadio
- Le informazioni si trasferiscono solo tramite i registri della pipeline
- Le informazioni di controllo di una istruzione devono “viaggiare” con gli operandi / i dati dell'istruzione stessa
- Non sono necessari segnali di controllo per la scrittura dei registri di pipeline (si usa il clock), come per il PC

Posizionamento del controllore



I segnali di controllo (1/2)

- nello stadio di EXECUTE sono previsti:
 - **M1** per pilotare il primo multiplexer nella selezione del primo operando da mandare all'addizionatore tra l'indirizzo di memoria e l'Offset;
 - **M2** per pilotare il secondo multiplexer nella selezione del secondo operando da mandare all'addizionatore tra il valore di (PC)+ e (Base_reg)::;
 - **M3** per selezionare l'appropriato registro destinazione tra Dest_reg_L/A e Dest_reg_C
 - **ALU_OPCODE** per comandare l'ALU ad effettuare l'operazione appropriata in funzione del tipo di istruzione da eseguire.

I segnali di controllo (2/3)

- nello stadio MEMORY sono previsti:
 - DCR per abilitare la lettura del dato dalla memoria
 - DCW per abilitare la scrittura del dato in memoria
 - JMP per consentire il salto di sequenza nell'esecuzione di un programma
- nello stadio di WRITE BACK sono previsti:
 - MWB per pilotare il multiplexer nella selezione tra dato e risultato
 - RW per abilitare la scrittura nel banco dei registri di quanto selezionato dal multiplexer

I segnali di controllo (3/3)

		EXECUTE	MEMORY	WRITE BACK
Instruction	OPCODE	M1 M2 M3 OP ₃ OP ₂ OP ₁	DCW DCR JMP	MWB RW
ADD	000001	- - 0 0 0 1	0 - 0	1 1
SUB	000010	- - 0 0 1 0	0 - 0	1 1
OR	000011	- - 0 0 1 1	0 - 0	1 1
AND	000100	- - 0 1 0 0	0 - 0	1 1
NOT	000101	- - 0 1 0 1	0 - 0	1 1
LOAD	000110	1 1 1 - - -	0 1 0	0 1
STORE	000111	1 1 - - - -	1 0 0	- 0
JMPC	001000	0 0 - - - -	0 0 1	- 0
NOP	000000	- - - 0 0 0	0 - 0	- 0

SCO-SCA

Prestazioni del pipelining

- La presenza della pipeline aumenta il numero di istruzioni *contemporaneamente* in esecuzione
- Quindi, introducendo il pipelining nel processore, *aumenta il throughput* ...
 - Throughput: numero di istruzioni eseguite nell'unità di tempo
- ... ma *non si riduce la latenza* della singola istruzione
 - Latenza: tempo di esecuzione della singola istruzione, dal suo inizio fino al suo completamento
 - Un'istruzione che richiede 5 passi, continua a richiedere 5 cicli di clock per la sua esecuzione con pipelining, mentre una che ne richiederebbe 4 necessita di 5 cicli di clock

Stadi della pipeline

- Il tempo necessario per fare avanzare un'istruzione di uno stadio lungo la pipeline corrisponde ad un ciclo di clock di pipeline
- Poiché gli stadi della pipeline sono collegati in sequenza, devono operare in modo sincrono
 - avanzamento nella pipeline sincronizzato dal clock
 - durata del ciclo di clock del processore con pipeline determinata dalla durata dello stadio più lento della pipeline
 - Es.: 200 ps per l'esecuzione dell'operazione più lenta
 - per alcune istruzioni, alcuni stadi sono cicli sprecati
- Obiettivo dei progettisti: bilanciare la durata degli stadi
- Se gli stadi sono *perfettamente bilanciati* e non ci sono istruz. con cicli sprecato, lo *speedup ideale* dovuto al pipelining è pari al numero di stadi della pipeline

$$\text{Speedup ideale}_{\text{pipeline}} = \frac{\text{tempo tra istruzioni}_{\text{no pipeline}}}{\text{tempo tra istruzioni}_{\text{pipeline}}} = \text{num. stadi pipeline}$$

Stadi della pipeline (2)

- Ma, in generale, gli stadi della pipeline non sono perfettamente bilanciati
- L'introduzione del pipelining comporta quindi costi aggiuntivi
 - L'intervallo di tempo per il completamento di un'istruzione è superiore al minimo valore possibile
 - Lo *speedup reale* sarà minore del numero di stadi di pipeline introdotto
 - In genere una pipeline a 5 stadi non riesce a quintuplicare le prestazioni

Miglioramento delle prestazioni

- *In generale*: partendo dalla pipeline vuota con k stadi, per completare n istruzioni occorrono $k + (n-1)$ cicli di clock:
 - k cicli per riempire la pipeline e completare l'esecuzione della prima istruzione
 - $n-1$ cicli per completare le rimanenti $n-1$ istruzioni

Miglioramento delle prestazioni (2)

- All'aumentare del numero di istruzioni n , il rapporto tra i tempi totali di esecuzione su macchine senza e con pipeline si avvicina al limite ideale
 - Il tempo per riempire/svuotare la pipeline diventa trascurabile rispetto al tempo totale per completare le istruzioni
- Esempio:
- 1000 istruzioni load senza pipeline: $5 \times 200 \times 1000 \text{ ps} = 1.000.000 \text{ ps}$
 - 1000 istruzioni lw con pipeline: $1000 + 200 \times (1000 - 1) = 200.800 \text{ ps}$
 - 200800 ps invece di 1000000 ps ($1000000/200800 = 4.98$ circa)

Miglioramento delle prestazioni (3)

- Nel caso asintotico ($n \rightarrow \infty$)
 - La **latenza** di alcune istruzioni potrebbe **peggiorare**, per esempio le istruzioni logiche/aritmetiche nel caso di processore multicycle necessitano
 - passa da 800 ps (senza pipelining) a 1000 ps (con pipelining)
 - Il **throughput** però **migliora** di 4 volte
 - Passa da 1 istruzione completata ogni 800 ps (senza pipelining) ad 1 istruzione completata ogni 200 ps (con pipelining)
- CASO IDEALE
 - Se consideriamo un processore multicycle con un clock da 200 ps ed un processore con pipelining (con 5 stadi da 200 ps ciascuno) nel caso asintotico
 - La **latenza** della singola istruzione rimane **invariata** e pari a 1000 ps
 - Il **throughput migliora** di 5 volte
 - Passa da 1 istruzione completata ogni 1000 ps (senza pipelining) ad 1 istruzione completata ogni 200 ps (con pipelining)

Miglioramento delle prestazioni (4)

- Quindi il pipelining **potrebbe incrementare il throughput** del processore (numero di istruzioni completate nell'unità di tempo), ma **sicuramente non** riduce il tempo di esecuzione (latenza) della singola istruzione
- Anzi, in generale il pipelining aumenta il tempo di esecuzione della singola istruzione, a causa di sbilanciamenti tra gli stadi della pipeline e overhead di controllo della pipeline
 - Lo sbilanciamento tra gli stadi della pipeline riduce le prestazioni
 - Il clock non può essere minore del tempo necessario per lo stadio più lento della pipeline
 - L'overhead della pipeline è causato
 - dai ritardi dei registri di pipeline e dal clock skew (ritardo di propagazione del segnale di clock sui fili)
 - dalla presenza di **criticità**

Le criticità

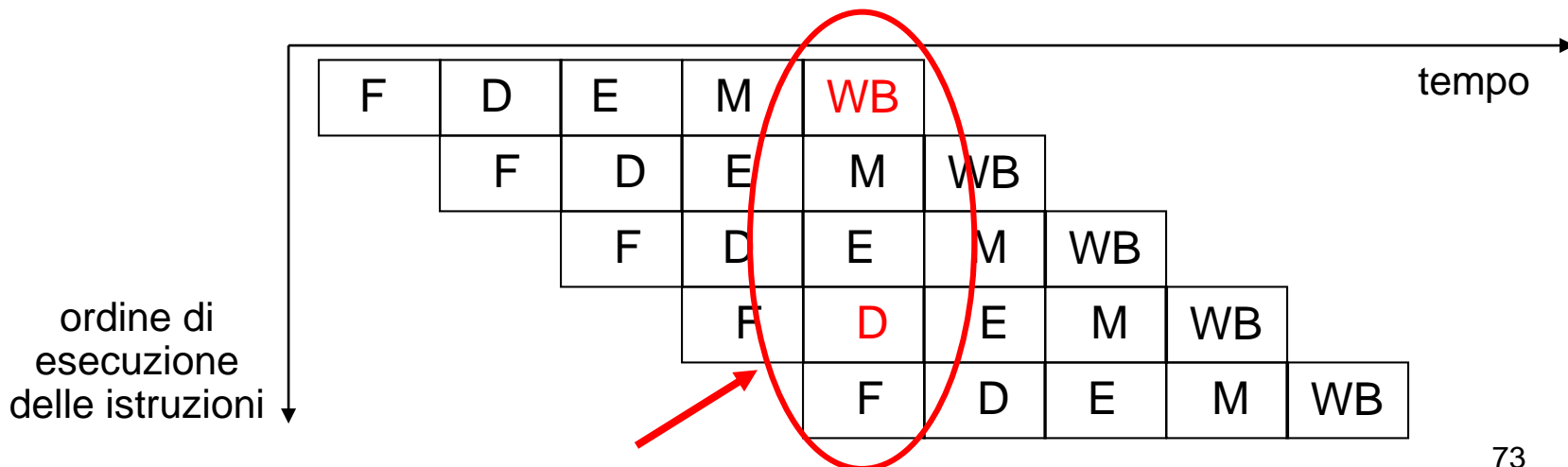
- Le *criticità* (o *conflitti* o *alee*) sorgono nelle architetture con pipelining quando non è possibile eseguire un'istruzione nel ciclo immediatamente successivo senza cambiare la semantica del programma eseguito in una struttura multiciclo
- Tre tipi di criticità:
 - Criticità strutturali
 - Criticità sui dati
 - Criticità sul controllo

Le criticità (2)

- Criticità **strutturale**
 - Tentativo di usare la stessa risorsa hardware da parte di diverse istruzioni in modi diversi nello stesso ciclo di clock, come nella lettura e scrittura contemporanea di un registro del banco dei registri da una istruzione in fase di decode ed una in fase di write back.
 - Da notare che si avrebbe un'altra criticità strutturale se nel processore didattico avessimo un'unica memoria per le istruzioni e i dati
- Criticità **sui dati**
 - Tentativo di usare un risultato prima che sia disponibile
 - Es.: istruzione che dipende dal risultato di un'istruzione precedente che è ancora nella pipeline
- Criticità **sul controllo**
 - Nel caso di salti condizionati, decidere quale prossima istruzione sia da eseguire prima che la condizione sia valutata

Criticità strutturali

- Nel processore didattico c'è un conflitto strutturale sul banco dei registri
- Si potrebbe evitare se il banco dei registri viene progettato per evitare conflitti tra la lettura e la scrittura nello stesso ciclo
 - Soluzione
 - **Scrittura** del banco dei registri nella **prima metà** del ciclo di clock
 - **Lettura** del banco dei registri nella **seconda metà** del ciclo di clock



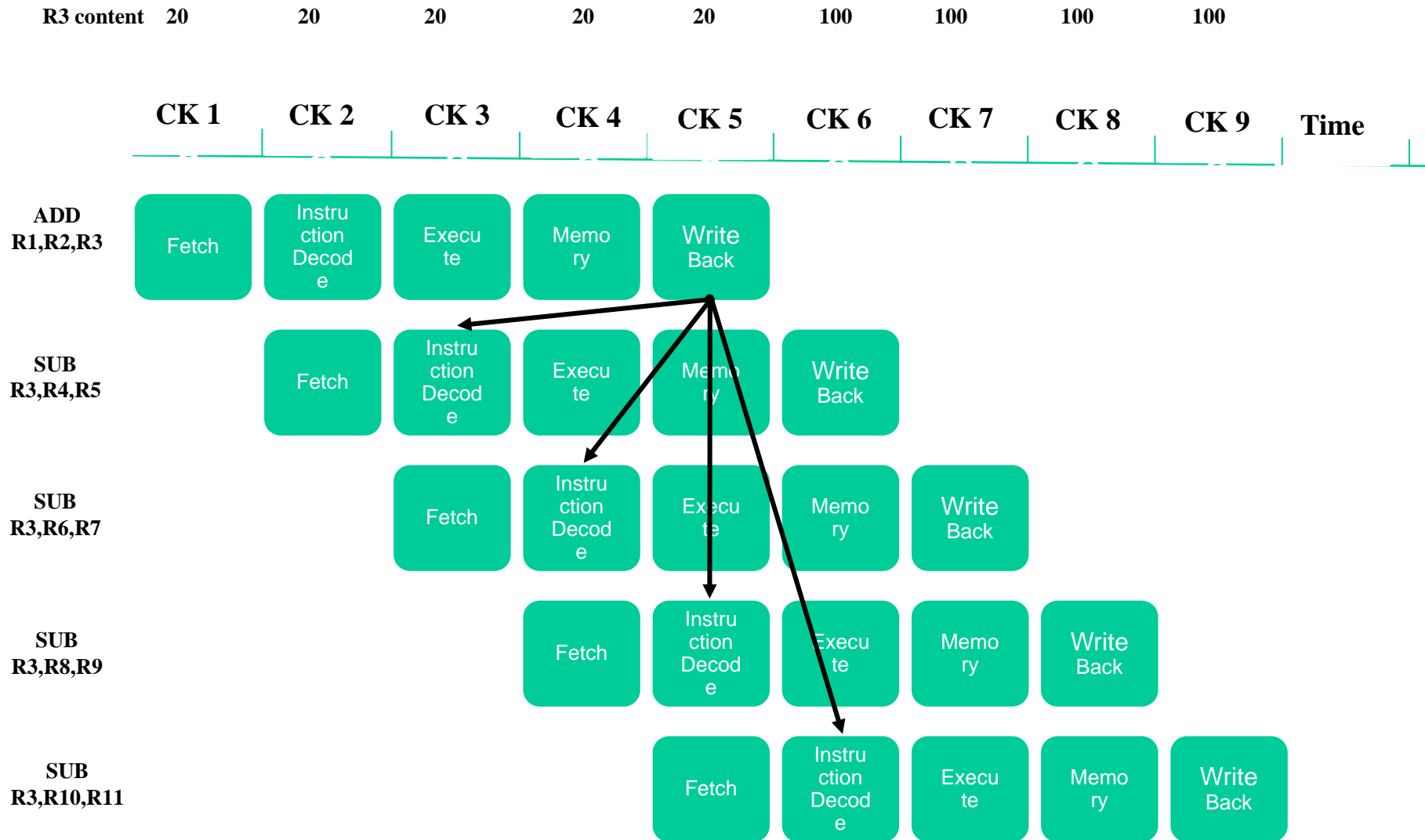
Criticità sui dati

- Un'istruzione dipende dal risultato di un'istruzione precedente che è ancora nella pipeline
- Esempio 1:
 - `add R3, R4, R5`
 - `sub R3, R5, R6`
 - uno degli operandi sorgente di sub (R5) è prodotto da `add`, che è ancora nella pipeline
 - Criticità sui dati di tipo *define-use*
- Esempio 2:
 - `load R3, 122(R1)`
 - `sub R5, R3, R6`
 - uno degli operandi sorgente di sub (R3) è prodotto da `load`, che è ancora nella pipeline
 - Criticità sui dati di tipo *load-use*

Frammento di programma con criticità define use

- add R1, R2, R3
- sub R3, R4, R5
- sub R3, R6, R7
- add R3, R8, R9
- sub R3, R10, R11

Schema temporale di esecuzione di frammento di programma con conflitti di tipo define use

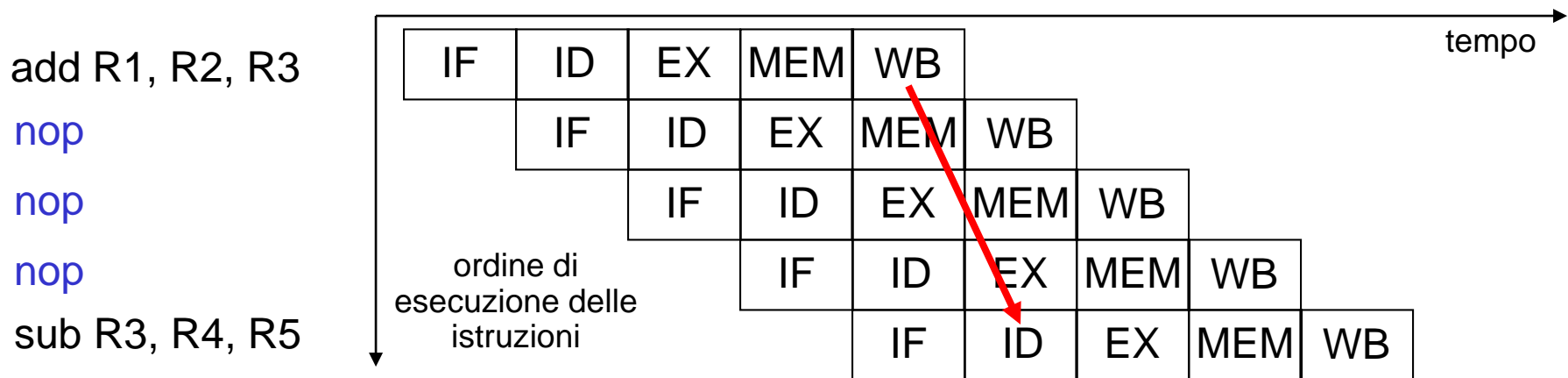


Soluzioni per criticità sui dati

- Soluzioni di tipo software
 - inserimento di istruzioni *nop* (no operation)
 - peggiora il throughput
 - riordino delle istruzioni
 - *spostare* istruzioni “innocue” in modo che esse eliminino la criticità
- Soluzioni di tipo hardware
 - inserimento di bolle (*bubble*) o stalli nella pipeline
 - si inseriscono dei tempi morti
 - peggiora il throughput
 - propagazione o scavalciamento (*forwarding* o bypassing)
 - si propagano i dati in avanti appena sono disponibili verso le unità che li richiedono

Define use (soluzione SW) Inserimento di nop

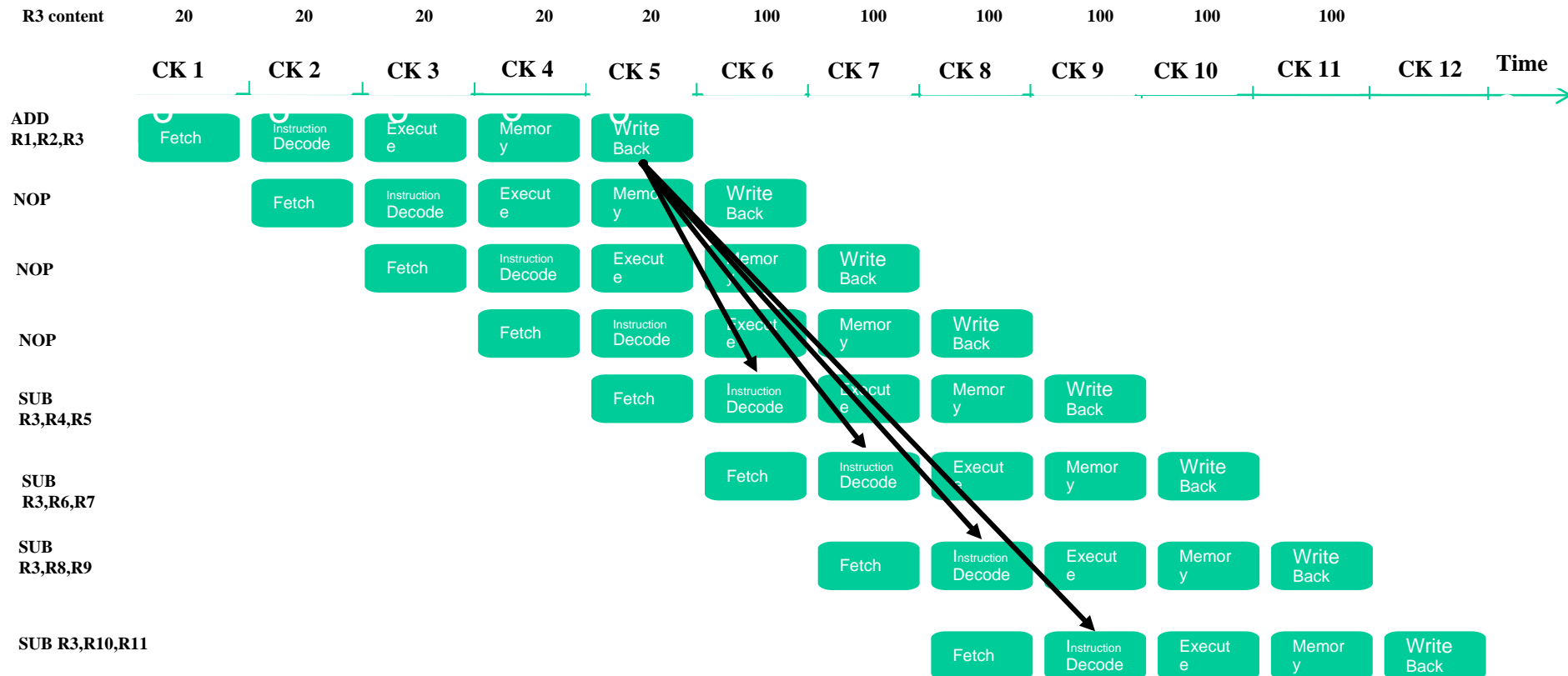
- Esempio 1: l'assemblatore deve inserire tra le istruzioni add e sub tre istruzioni nop, facendo così scomparire il conflitto (caso di presenza di criticità strutturale sul banco registri)
 - L'istruzione nop è l'equivalente software dello stallo



Nel caso di assenza di criticità strutturale sul banco dei registri

sarebbero sufficienti **due** solo *nop*

Define use - Inserimento nop



Define use(soluzione SW): riordino delle istruzioni

- L'assemblatore o il programmatore riordina le istruzioni in modo da impedire che istruzioni correlate siano troppo vicine
- L'assemblatore o il programmatore cerca di inserire tra le istruzioni correlate (che presentano dei conflitti) delle istruzioni *indipendenti* dal risultato delle istruzioni precedenti
 - Quando l'assemblatore non riesce a trovare istruzioni indipendenti deve inserire istruzioni nop
- Esempio

Codice con criticità

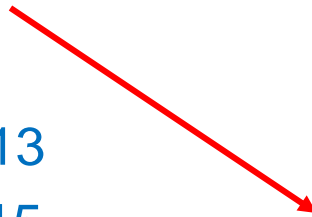
criticità

add R1, R2, R3
sub R3, R6, R7

sub R8, R9, R10
add R11, R12, R13
sub R14, R15, R15

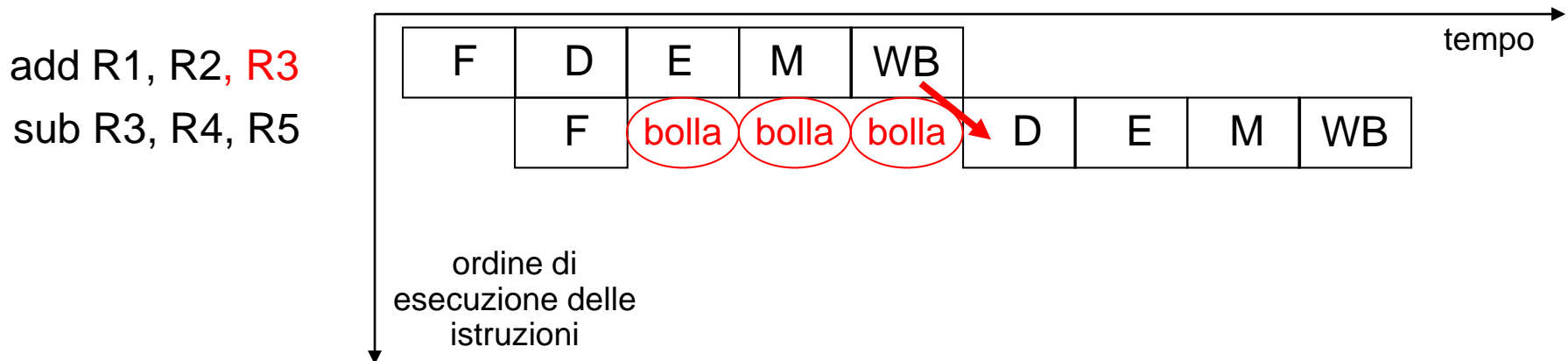
Codice senza criticità

add R1, R2, R3
sub R8, R9, R10
add R11, R12, R13
sub R14, R15, R16
sub R3, R6, R7



Define use (soluzione HW) Inserimento di bolle (1/3)

- Si inseriscono delle bolle nella pipeline, ovvero si blocca il flusso di istruzioni nella pipeline finché il conflitto non è risolto
 - **STALLO**: stato in cui si trova il processore quando le istruzioni sono bloccate
- Esempio 1: occorre inserire **tre bolle** per fermare l'istruzione sub affinché possano essere letti i dati corretti
 - **Due bolle** se assenza di conflitti strutturali sul banco dei registri



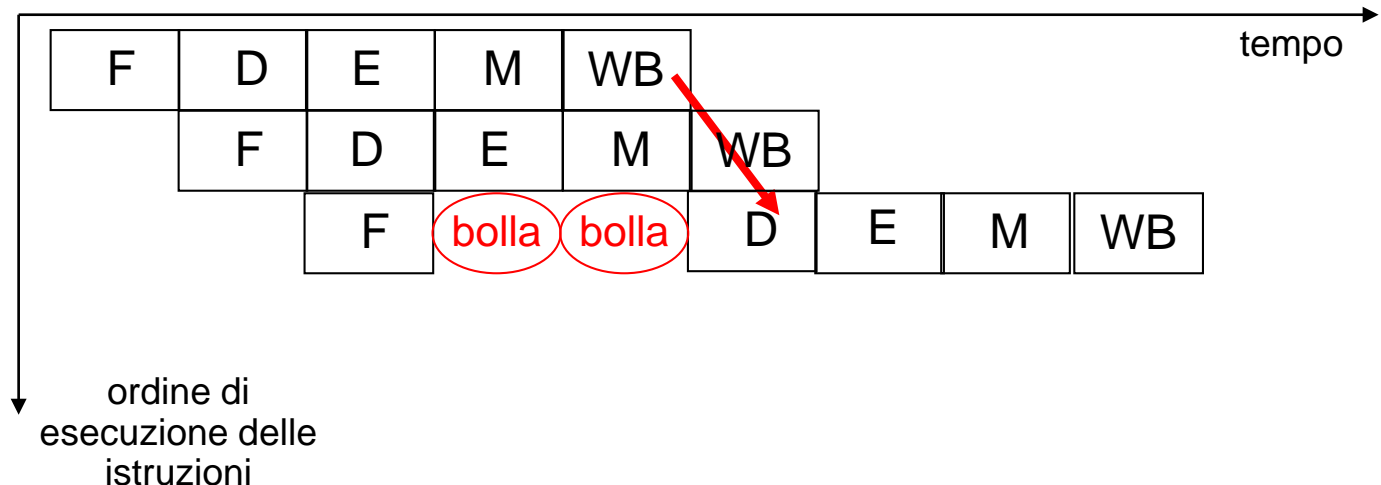
Define use (soluzione HW) Inserimento di bolle (2/3)

- Esempio 2: occorre inserire **due bolle** per fermare l'istruzione sub affinché possano essere letti i dati corretti
 - **una bolla** se assenza di conflitti strutturali sul banco dei registri

add R1, R2, **R3**

add R7, R8, R9

sub R3, R4, R5



Define use (soluzione HW) Inserimento di bolle (3/3)

Le bolle si generano:

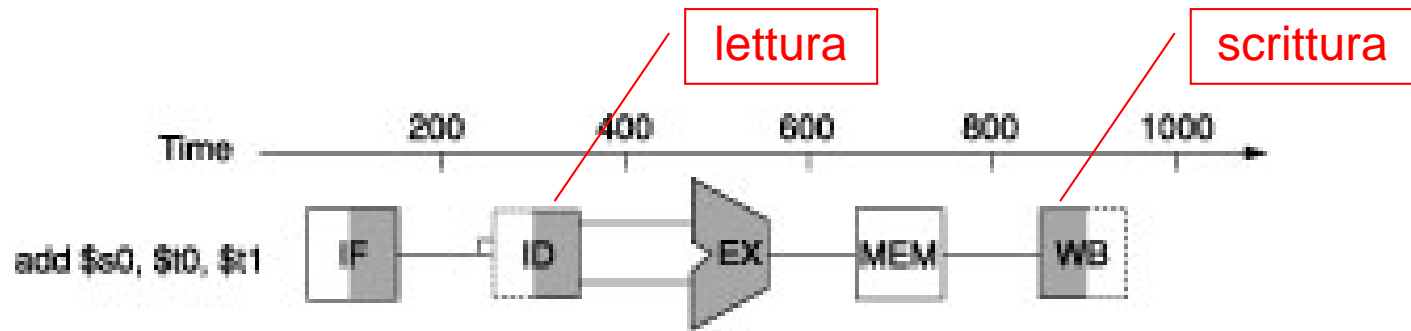
- bloccando il FETCH delle istruzioni
- facendo propagare il codice operativo della nop invece che quella dell'istruzione nello stato di DECODE

A tal fine si utilizza un dispositivo **HAZARD DETECTION UNIT**, che controllando la presenza del conflitto effettua le operazioni precedenti.

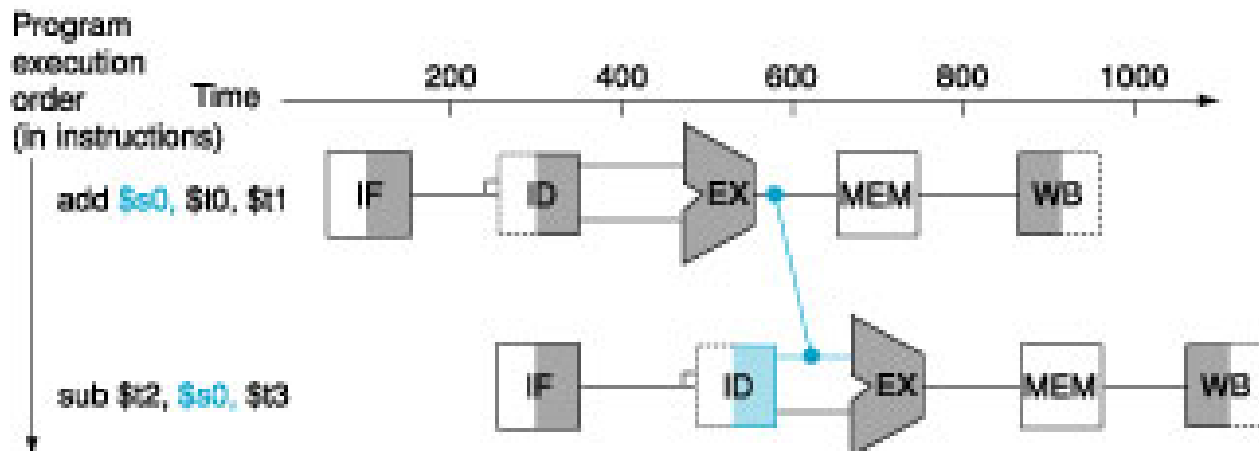
La presenza dei conflitti viene effettuata comparando l'identificativo dei registri destinazione e sorgenti e verificando che l'istruzione davanti abbia una fase di Write Back (valore del segnale di controllo RW pari ad 1)

Notare che tale dispositivo è un semplice circuito combinatorio in grado di fare comparazioni

(soluzione HW) Propagazione (o forwarding)



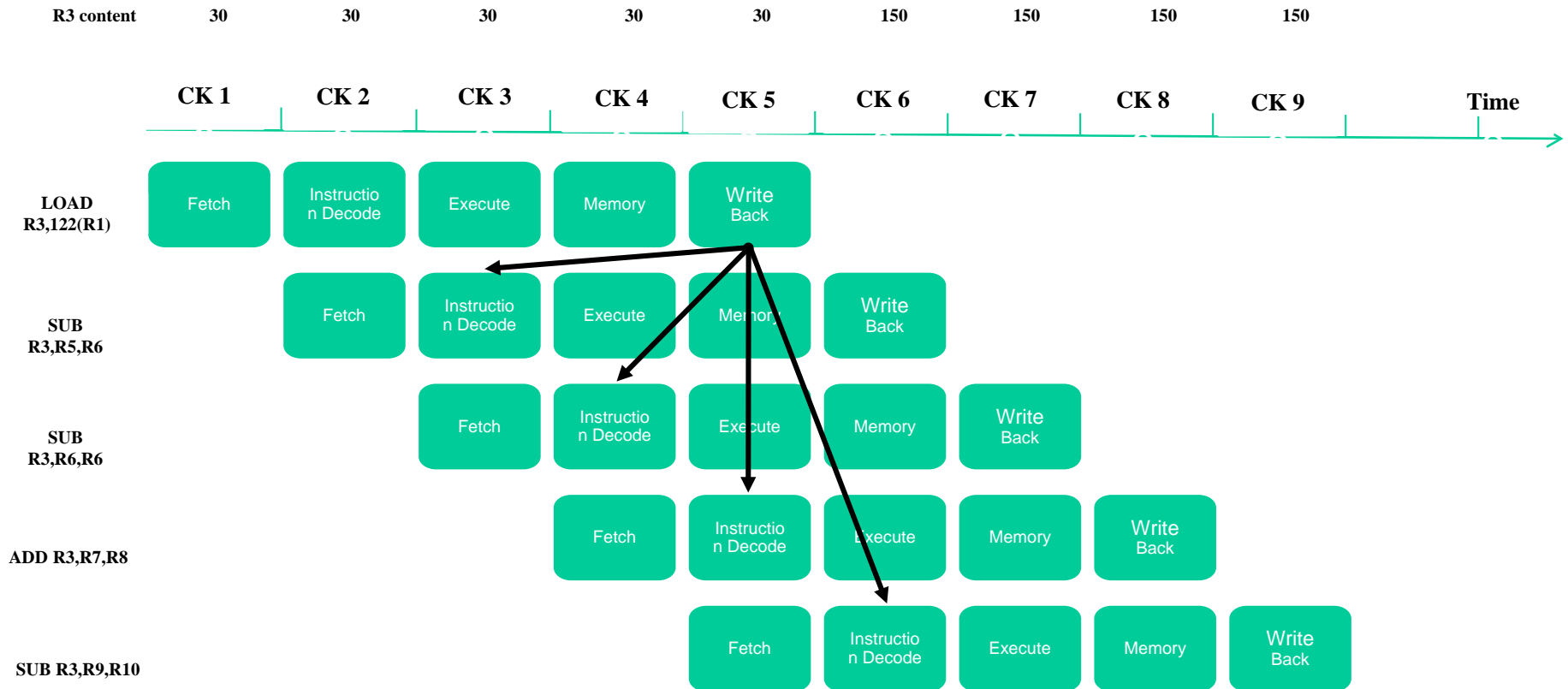
- Esempio 1: quando la ALU genera il risultato, questo viene *subito* messo a disposizione per il passo dell'istruzione che segue tramite una *propagazione in avanti*



Criticità load use

- load **R3**, 122(R1)
- sub **R3**, R5, R6
- sub **R3**, R6, R6
- add **R3**, R7, R8
- sub R3, R9, R10

Schema temporale di criticità load use



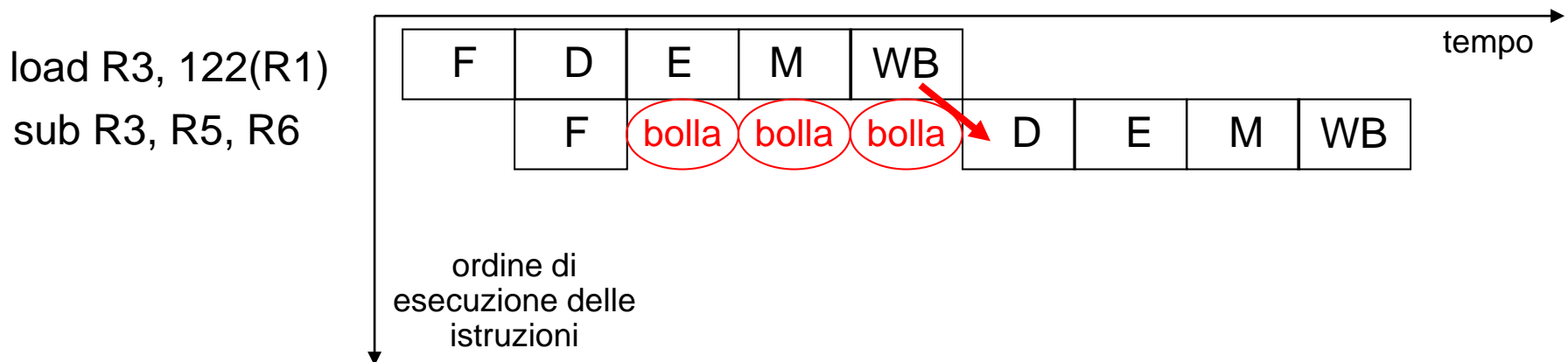
Load use (soluzione SW) Propagazione e stallo

- load **R3**, 122(R1)
- nop
- nop
- nop
- sub R3, R5, R6
- sub R3, R6, R6
- add R3, R7, R8
- sub R3, R9, R10

N.B. Nel caso di modifica banco registri per evitare conflitti strutturali necessitano solo due nop

load use (soluzione HW) inserimento bolle

- Ciò che può essere fatto a SW può essere fatto in HW
- Esempio 1: occorre inserire **tre bolle** per fermare l'istruzione sub affinché possano essere letti i dati corretti
 - **Due bolle** se assenza di conflitti strutturali sul banco dei registri



define use (soluzione HW) Propagazione e stallo

- Esempio 2:

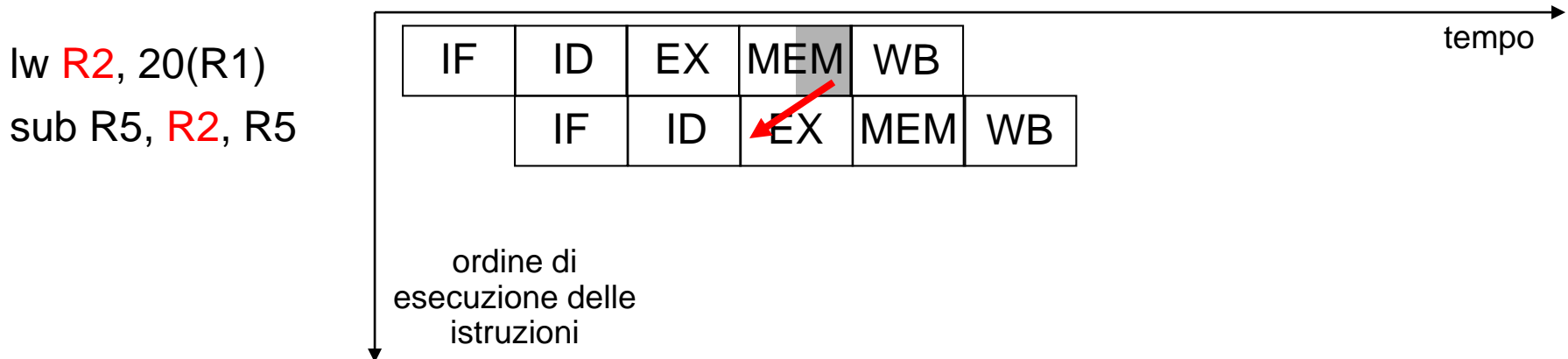
lw R2, 20(R1)

sub R5, R2, R5

- E' una criticità sui dati di tipo *load-use*

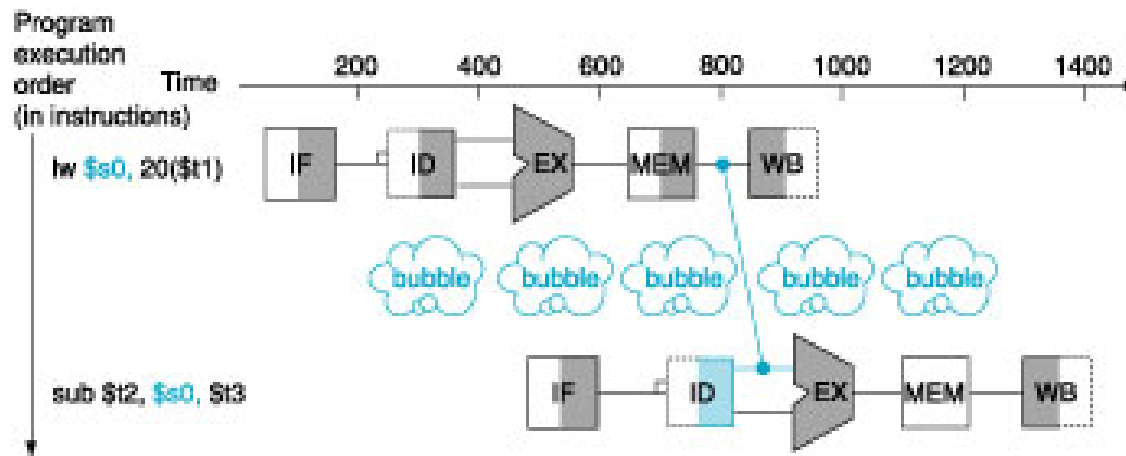
- Il dato caricato dall'istruzione di load non è ancora disponibile quando viene richiesto da un'istruzione successiva

- La sola propagazione è insufficiente per risolvere questo tipo di criticità – necessità di almeno una bolla per far completare la lettura del dato



Propagazione e stallo (2)

- Soluzione possibile: **propagazione e uno stallo**

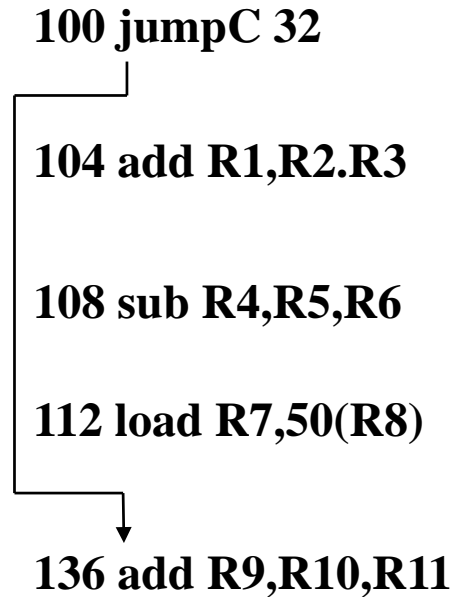


- Senza propagazione e ottimizzazione del banco dei registri, sarebbero stati necessari **tre stalli**

Criticità sul controllo

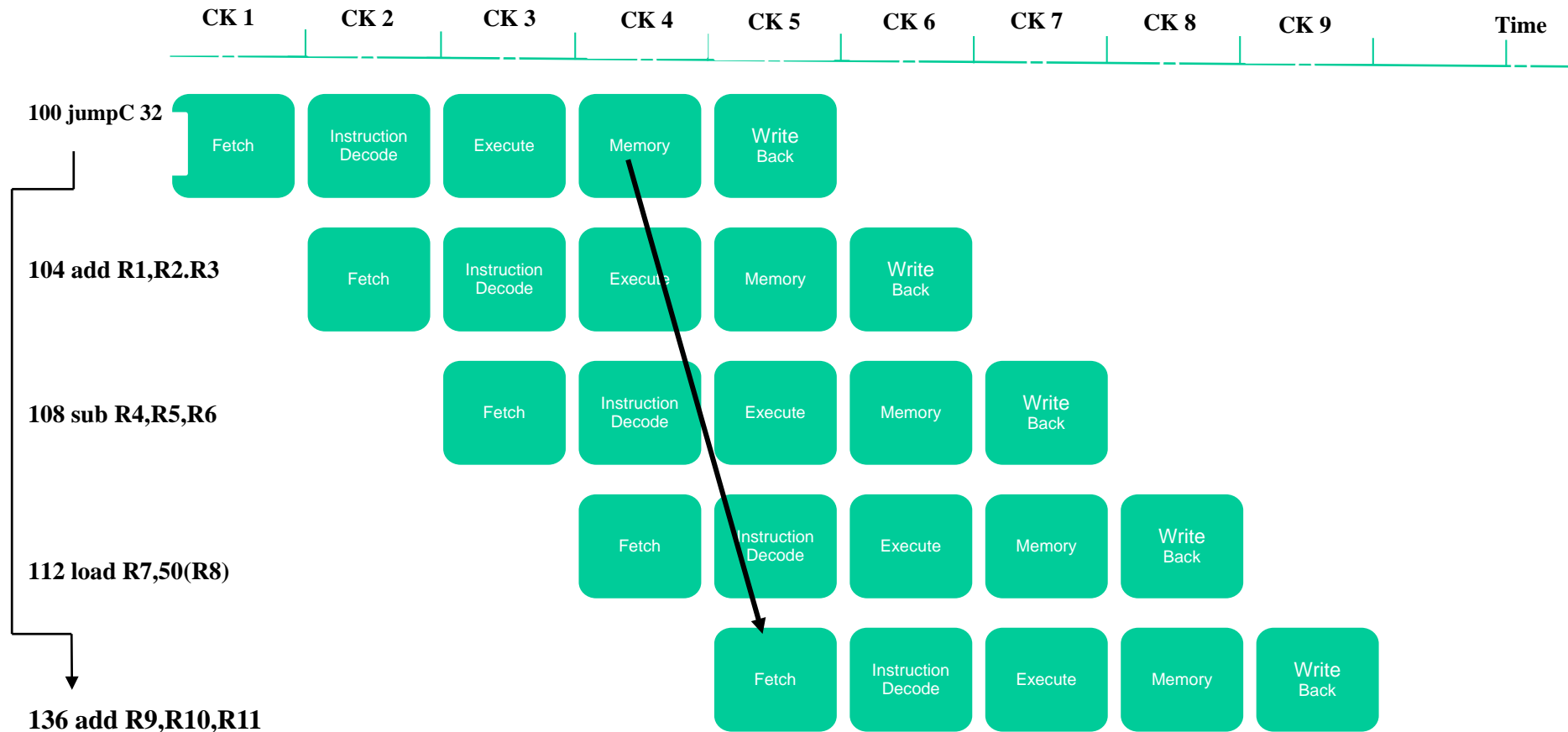
- Per alimentare la pipeline occorre inserire un'istruzione ad ogni ciclo di clock
- Tuttavia, nel processore didattico la decisione sul salto condizionato non viene presa fino al quarto passo (MEMORY) dell'istruzione *jumpX*
- Comportamento desiderato del salto:
 - se il bit selezionato è pari a 0 continuare l'esecuzione con l'istruzione successiva a *jumpx*
 - se il bit selezionato è pari a 1 non eseguire le istruzioni successive alla *jumpx* e saltare all'indirizzo specificato

Esempio di criticità sul controllo



(Note that the value 136 is obtained by $100 + S + 32$ in case S is equal to 4)

Rappresentazione spazio/temporale della criticità



(Note that the value 136 is obtained by $100 + S + 32$ in case S is equal to 4)

Soluzioni per criticità sul controllo

- Approccio **pessimistico**: non si eseguono istruzioni significative fino al completamento della scelta
- Approccio **ottimistico**: si prospetta che non venga effettuato il salto
- Vantaggi/svantaggi:
 - **pessimistico**: attesa della decisione (perdita di tempo)
 - **ottimistico**: necessità di annullare gli effetti delle istruzioni eseguite nel caso di salto

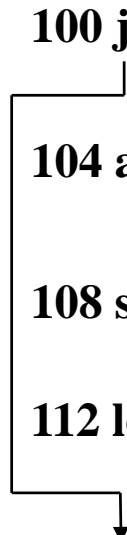
Soluzioni pessimistiche

- Software: inserimento di 3 nop
- Hardware: inserimento di 3 bolle
 - si blocca la pipeline finché non è noto il risultato del confronto della *jumpX* e si sa quale è la prossima istruzione da eseguire

Conflitto sul controllo (soluzione pessimistica software)

Codice iniziale

100 jumpC 32
104 add R1,R2,R3
108 sub R4,R5,R6
112 load R7,50(R8)
136 add R9,R10,R11



```
graph TD; 100[100 jumpC 32] --> 104[104 add R1,R2,R3]; 104 --> 108[108 sub R4,R5,R6]; 108 --> 112[112 load R7,50(R8)]; 112 --> 136[136 add R9,R10,R11];
```

Codice modificato

100 jumpC 44
104 nop
108 nop
112 nop
116 add R1,R2,R3
120 ub R4,R5,R6
124 load R7,50(R8)
.
.
.
148 add R9,R10,R11

Conflitto sul controllo (soluzione pessimistica hardware)

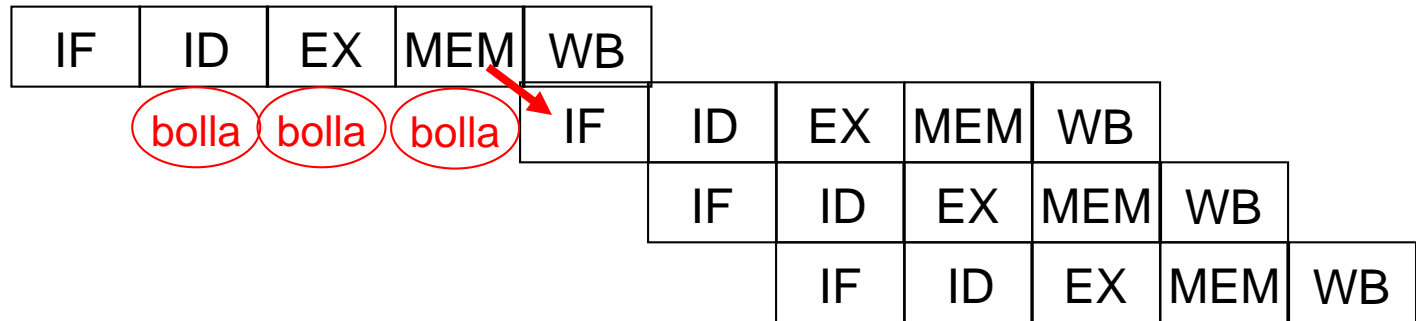
Esempio 1: caso in cui il salto non viene effettuato

100 jumpC 32

104 add R1,R2,R3

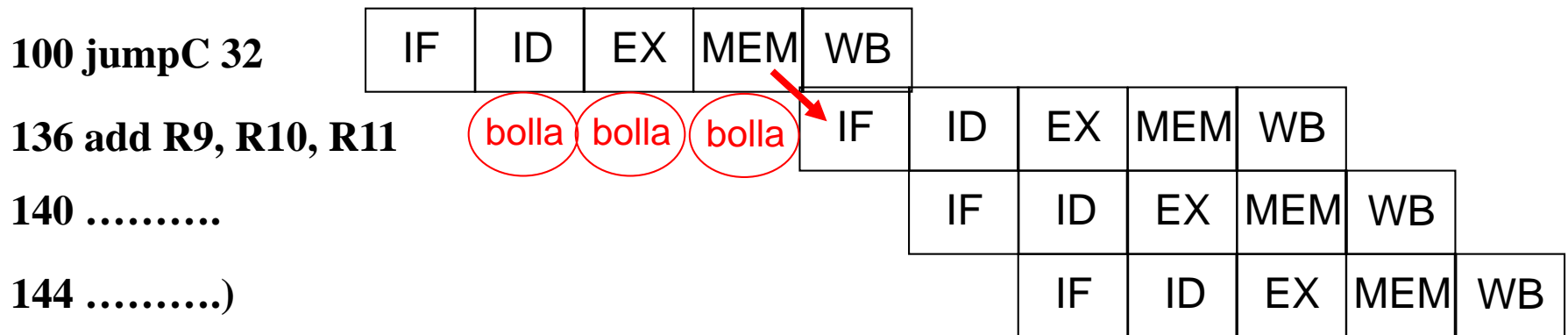
108 sub R4,R5,R6

112 load R7,50(R8)



Conflitto sul controllo (soluzione pessimistica hardware)

Esempio 2: caso in cui il salto viene effettuato



Conflitto sul controllo (soluzione ottimistica hardware) 1/2

Approccio **ottimistico**: si prospetta che non venga effettuato il salto

Quindi si eseguono le istruzioni in sequenza all'istruzione di salto:

- se il salto non doveva essere effettuato OK
- se il salto doveva essere effettuato è necessario annullare gli effetti dell'esecuzione delle tre istruzioni eseguite (RECOVERY dello stato antecedente all'esecuzione delle tre istruzioni erroneamente eseguite)

Conflitto sul controllo (soluzione ottimistica hardware) 1/2

- Se il salto viene eseguito
 - Si scartano le istruzioni che sono state nel frattempo caricate nella pipeline
 - Nell'esempio del lucido 3 istruz. (add, sub, load) sono da scartare
 - Si puliscono gli stadi F, D, E
 - *Flushing* (annullamento) delle istruzioni
 - La pipeline viene caricata a partire dall'istruzione di destinazione del salto (nell'esempio del lucido l'istruzione add R9, R10, R11)
 - Non è stato modificato nessun registro del banco reg. perché nessuna istruzione successiva al salto ha raggiunto lo stadio WB
 - Necessità di *ripristinare lo Status Register* al valore relativo all'istruzione precedente il salto – NORMALMENTE VIENE FATTO USANDO UNA COPPIA DI REGISTRI, di cui uno memorizza il valore del secondo quando si è sicuri che tale valore non verrà mai annullato – *Richiami al checkpointing*

Ottimizzazione dei prestazioni: predizione del salto

- Tecniche di predizione dinamica:
 - Locali
 - Globali

Esempio di tecnica di predizione locale: BIMODALE

- utile nel caso di salti collegati a cicli (loop)
- migliora le prestazioni pesantemente in funzione del numero di iterazioni da effettuare prima di uscire dal loop

Tecnica bimodale

- Uso di una tabella di contatori, a cui si accede utilizzando i bit meno significativi dell'indirizzo delle istruzioni di salto
- Tali contatori, di due bit, possono assumere quattro valori:
 - 00, fortemente non scelto;
 - 01, debolmente non scelto;
 - 10, debolmente scelto;
 - 11, fortemente scelto.
- Ad ogni predizione di salto, se la condizione è verificata si passa da un valore a quello crescente (e quindi se si è già nello stato 11 si rimane in tale stato), mentre se la condizione non è verificata si passa al valore inferiore (anche in questo caso se si è già nello stato 00 si rimane in tale stato).

Altri problemi della pipeline

- FPU
- Gestione interruzioni/eccezioni

FPU

- L'FPU ha tempi di elaborazione molto lunghi rispetto a quelli del singolo stadio visto fino ad ora, necessità quindi di mettere in stallo per lunghi periodi il processore

Eccezioni/Interruzioni imprecise

- In un calcolatore con pipelining è difficile associare sempre in modo corretto un'eccezione all'istruzione che l'ha provocata (ci sono in esecuzione un numero di istruzioni pari al numero degli stadi) o quale istruzione è in esecuzione al momento dell'arrivo di una interruzione
- Eccezione/Interruzione *imprecisa*: non è associata alcuna istruzione in modo esatto, come nel multiciclo che ha causato l'eccezione
- Eccezione/Interruzione *precisa*: è sempre associata all'istruzione esatta che ha causato l'eccezione o all'istruzione in fase di execute durante l'arrivo dell'interruzione.

Architetture di calcolo avanzate basate sul pipelining

- Superpipeline

Fetch	Instruction Decode	Execute	Execute	Execute	Memory	Write Back
-------	-----------------------	---------	---------	---------	--------	---------------

- Superscalari

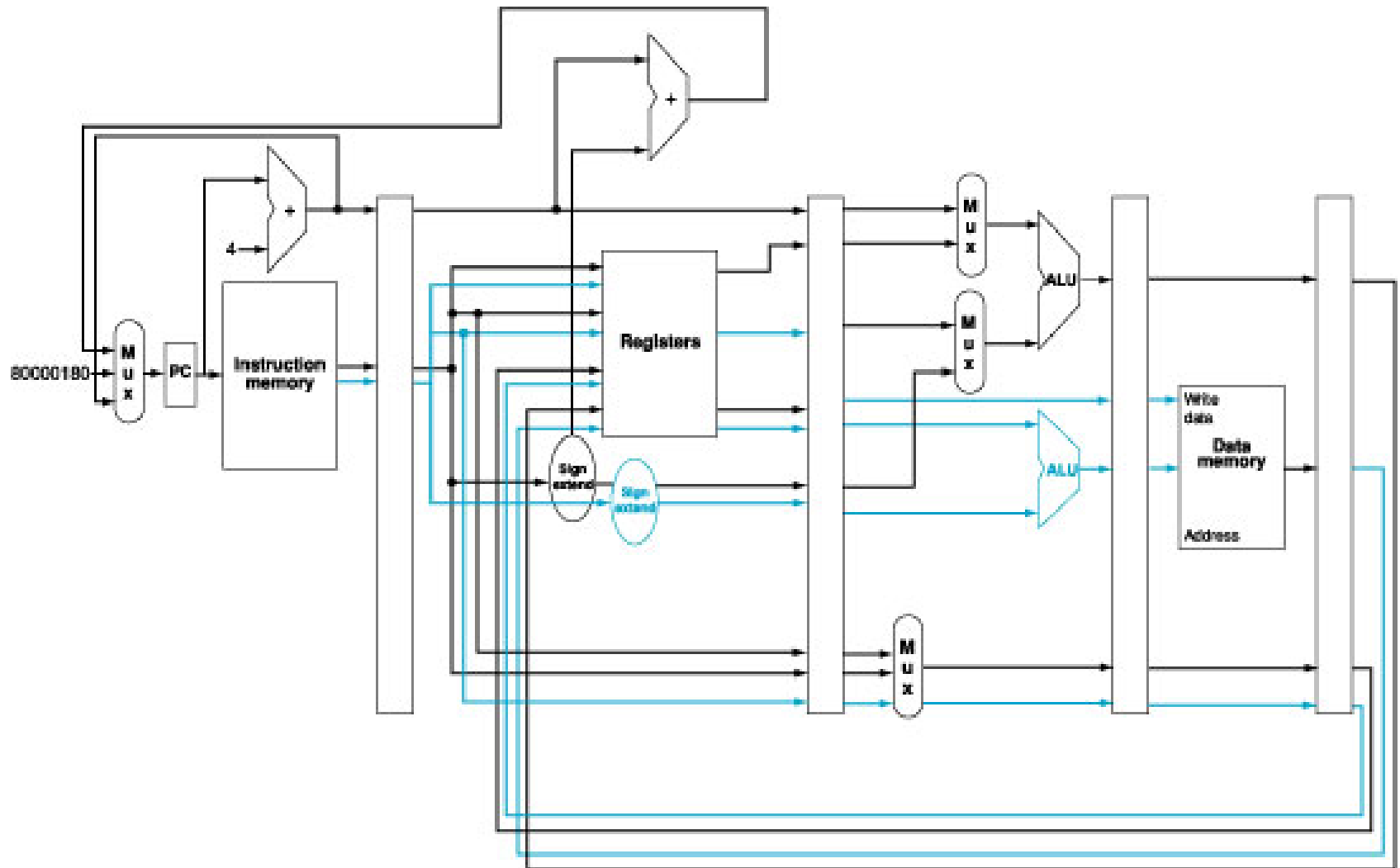
Fetch	Instruction Decode	Execute	Memory	Write Back
Fetch	Instruction Decode	Execute	Memory	Write Back

Superscalare a due vie

- Si leggono da 2 istruzioni alla volta (dimensione complessiva dell'Instruction Cache pari a 64 bit)
- Ogni programma è organizzato in modo da alternare una istruzione logica/aritmetica o di salto con una di load o store

Istruzione	Stadi della pipeline							
ALU o branch	IF	ID	EX	MEM	WB			
Load o store	IF	ID	EX	MEM	WB			
ALU o branch		IF	ID	EX	MEM	WB		
Load o store		IF	ID	EX	MEM	WB		
ALU o branch			IF	ID	EX	MEM	WB	
Load o store			IF	ID	EX	MEM	WB	
ALU o branch				IF	ID	EX	MEM	WB
Load o store				IF	ID	EX	MEM	WB

MIPS: superscalare a due vie



Considerazioni prestazionali

Aumentare:

- il numero di stadi (per il superpipeline)
- Il numero di vie per il superscalare

NON MIGLIORA IN ASSOLUTO LE PRESTAZIONI
in quanto si aumenta la probabilità di conflitto