Calcolatori Elettronici B
a.a. 2008/2009

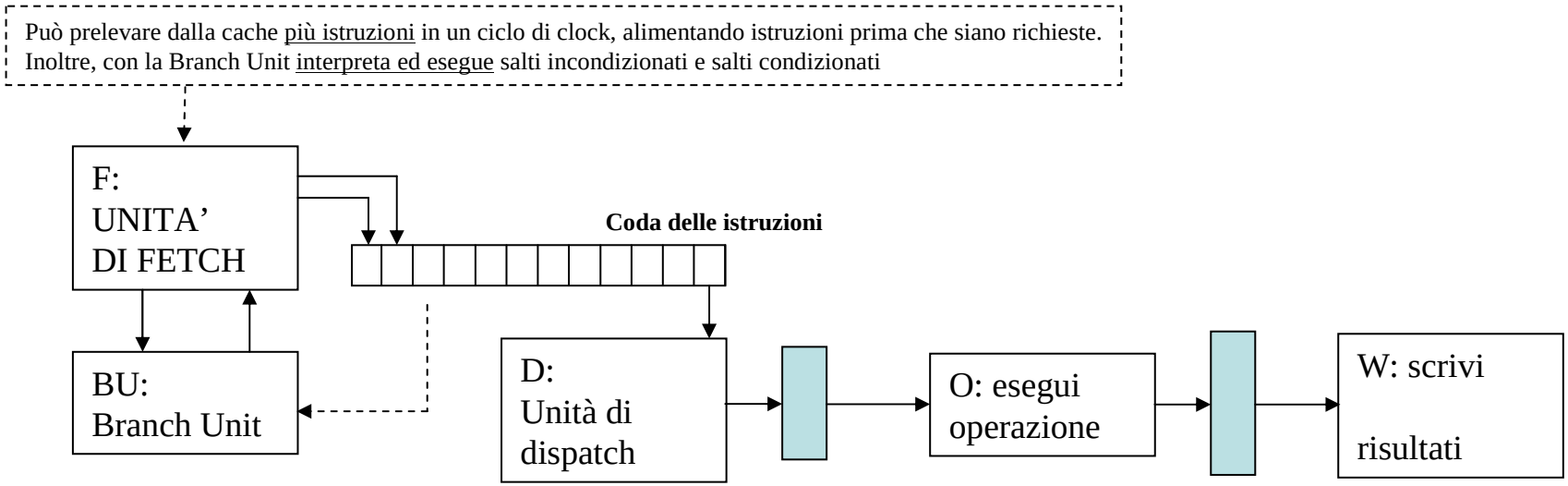
Pipeline:
Tecniche Avanzate

(Instruction Level Parallelism ILP)

Massimiliano Giacomin

PIPELINE:

LA CODA DELLE ISTRUZIONI



- Per assicurare continuità del flusso di esecuzione istruzioni, unità di prelievo cerca di mantenere sempre piena la coda delle istruzioni: preleva più istruzioni in un ciclo di clock dalla cache e le pone in coda; in particolare,
  l'unità di prelievo può decodificare istruzioni di salto e inserire in coda le istruzioni da eseguire, nell'ordine corretto: vedremo che questo permette di risparmiare cicli.
- Unità di smistamento preleva l'istruzione dalla testa della coda e la decodifica

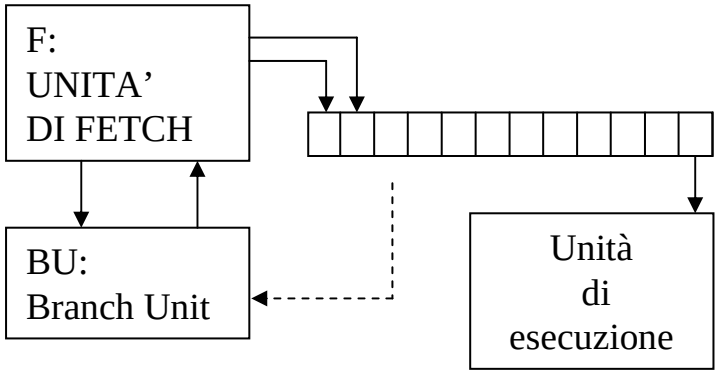
⇨ Può permettere di evitare cicli di stallo nei seguenti casi:

- Fallimento di accesso alla cache [criticità strutturale]
- Salti incondizionati e condizionati

Primo caso: fallimento di accesso alla cache

- Unità di prelievo preleva istruzioni indipendentemente da unità di smistamento:
  coda delle istruzioni mantenuta piena (per quanto possibile)
- Fallimento accesso cache ⇒
  smistamento può continuare (l'unità di smistamento continua a prelevare istruzioni dalla coda finchè non è vuota)
  intanto, il blocco della cache viene letto dalla memoria principale (o dalla cache secondaria)
- Se la frequenza con cui si possono prelevare istruzioni dalla cache è sufficientemente elevata, la coda non si svuota completamente e non vi sono stalli dovuti a fallimento di accesso alla cache (altrimenti la coda si svuota e l'unità di smistamento è comunque in grado di mettere in stallo la pipeline – ovvero, di non inviare nuove istruzioni all'unità di esecuzione)

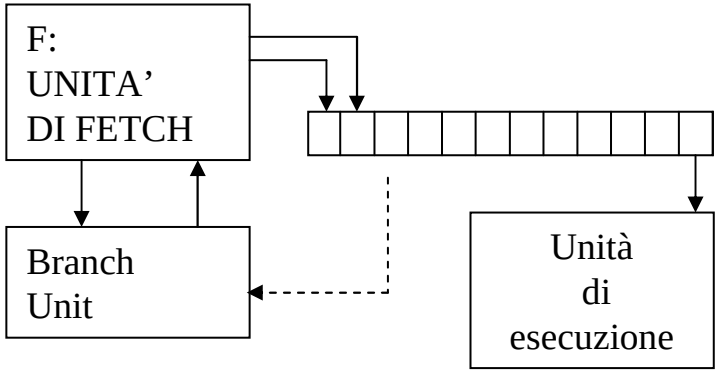
Secondo caso: salti (incondizionati e condizionati – per ora senza predizione)



(estensione della semplice pipeline a due stadi)

- Ciclo di clock C1: l'unità di fetch carica più istruzioni in coda da cache veloce
- Ciclo di clock C2: le istruzioni sono disponibili in coda. L'unità di fetch (+ BU) esamina ed "esegue" le istruzioni di salto presenti in coda – supponiamo che il calcolo indirizzo destinazione e (per salti condizionati) la condizione di salto si possano fare entrambi durante lo stesso ciclo di clock C2:
  - salto incondizionato: calcolo destinazione (C2) e aggiornamento del "PC"
  - salto condizionato, se è già possibile valutare la condizione di salto:
    - > salto effettuato: come nel salto incondizionato (PC aggiornato alla fine di C2)
    - > salto non effettuato: si prosegue con istruzioni successive
- Ciclo di clock C3: eventuale caricamento della coda delle istruzioni, a partire dalla istruzione di destinazione – notare che le istruzioni sono scritte nella coda alla fine di C3 (saranno disponibili in C4)

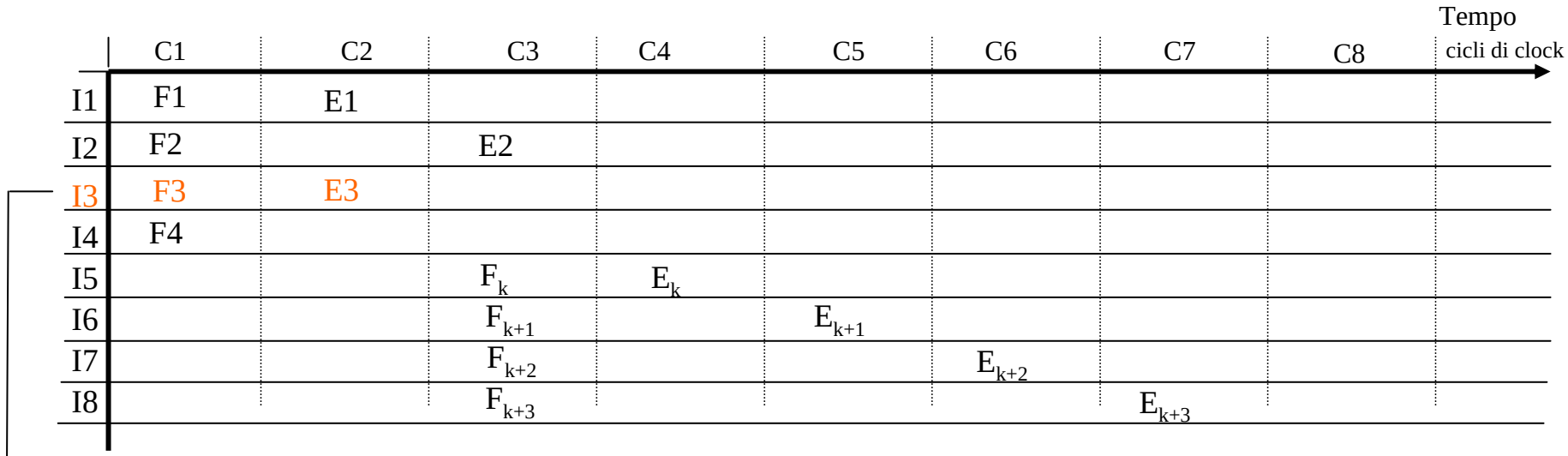
Esempio: salti incondizionati



Ipotesi:

- fetch [da cache veloce] **4** istruzioni alla volta
- nel ciclo di clock successivo a quello di fetch, l'unità di fetch e BU interpreta istruzioni di salto incondizionato e calcola l'indirizzo di destinazione.
- se si incontra un salto incondizionato, l'unità di fetch è quindi in grado (nel ciclo successivo) di caricare la coda con nuove istruzioni a partire dalla destinazione.

Esempio (supponendo solo due fasi per le istruzioni – fetch + "esecuzione")



I3 è un *salto incondizionato* "eseguito" dall'unità fetch.
Si noti che la penalità di salto è di –1 (!!!).

Salti condizionati

Per i salti condizionati, la decisione di saltare non può essere presa finchè la condizione di salto non è stata calcolata (dipende da istruzioni precedenti).

- Può essere che, quando l'unità di prelievo "incontra" l'istruzione di salto, la condizione di salto sia già stata valutata da un'altra unità: in tal caso l'unità di prelievo può lasciare la coda inalterata [se non si salta] oppure cominciare a prelevare (come nel caso del salto incondizionato) le istruzioni a partire dalla destinazione [se si salta]
- Però in generale la condizione di salto potrebbe essere ancora da valutare

⇒ Si usano le tecniche di predizione viste in precedenza (utilizzate però con la coda delle istruzioni)

Predizione dei salti condizionati con la coda delle istruzioni

Una volta prelevate le istruzioni, nel ciclo successivo al prelievo (C2) posso calcolare la destinazione dei salti; per i salti condizionati di cui non si è in grado di valutare la condizione di salto, uso la predizione:

- Predizione di salto effettuato ⇒ prelievo a partire da destinazione
- Predizione di salto non effettuato ⇒ "prelievo" da istruzione successiva
  [o mantengo istruzioni in coda]

Quando è possibile, l'unità di prelievo valuta la condizione di salto:

Predizione corretta ⇒ si ottengono gli stessi benefici del salto incondizionato

Predizione errata ⇒ scartare le istruzioni prima che modifichino registri

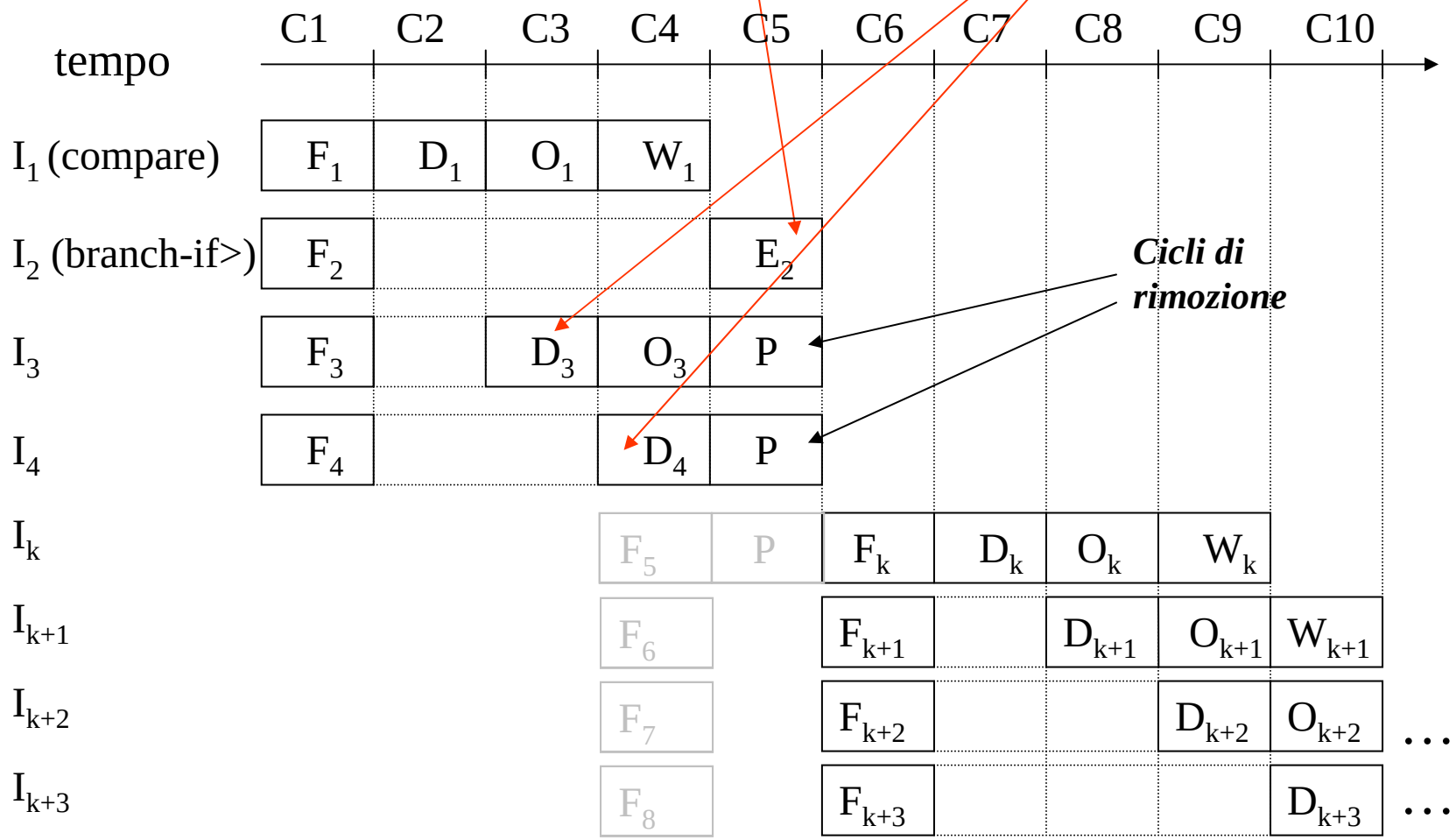
I passi di scrittura delle istruzioni eseguite su base speculativa (ovvero, le istruzioni "previste") devono avvenire dopo la valutazione della condizione di salto
[eventualmente, i passi di scrittura vengono ritardati]

Esempio: predizione di “salto non effetuato” nel caso di predizione errata

(pipeline a 4 stadi, prelievo di 4 istruzioni per ciclo)

Assunzione: calcolo condizione di branch e calcolo nuovo indirizzo in un ciclo. Però Branch Unit (parte dell’unità di prelievo) deve attendere il risultato del Compare
[NB: SI ASSUME DI NON DISPORRE DI UNITA’ DI PROPAGAZIONE]

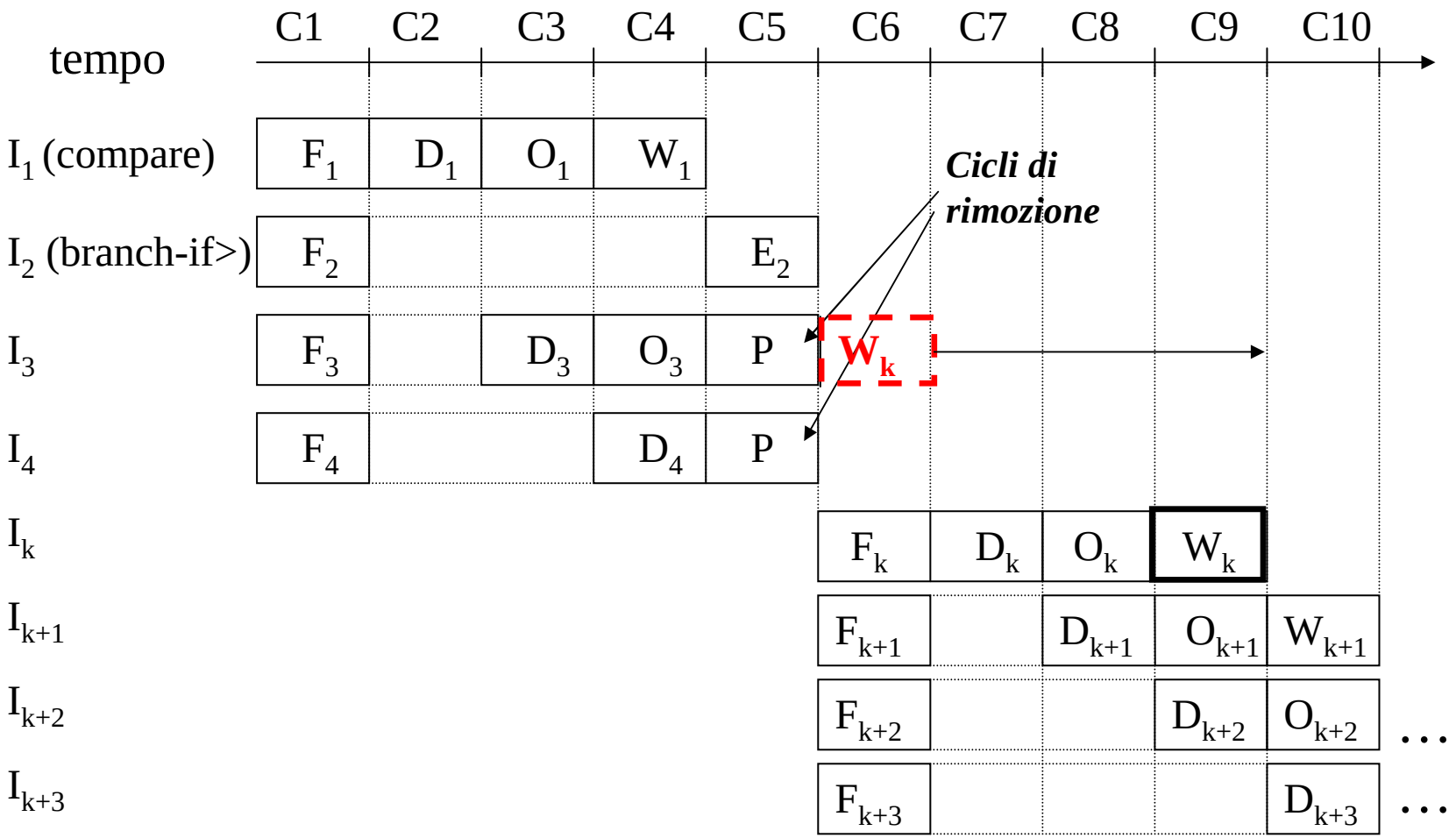
Eseguite perché si predice salto non effettuato



Chiarimenti sull'esempio

- L'unità di prelievo predice che il salto non verrà effettuato, quindi le istruzioni I_3 e I_4 vengono fatte proseguire
- La decisione finale sul salto viene presa dall'unità di prelievo (stadio indicato con E_2) dopo che i flag dei codici di condizione sono stati aggiornati al termine dell'istruzione I_1 (e quindi al termine di W_1)
- Quindi, dopo il passo W_1, l'unità di prelievo si accorge che la predizione era errata e che le istruzioni I_3 e I_4 vanno rimosse (cicli 'P' in figura)
- Quattro nuove istruzioni, da I_k a I_{k+3} vengono prelevate a partire dall'indirizzo calcolato al passo E_2

Chiarimento: penalità è di tre cicli



Chiarimenti sull'esempio (2)

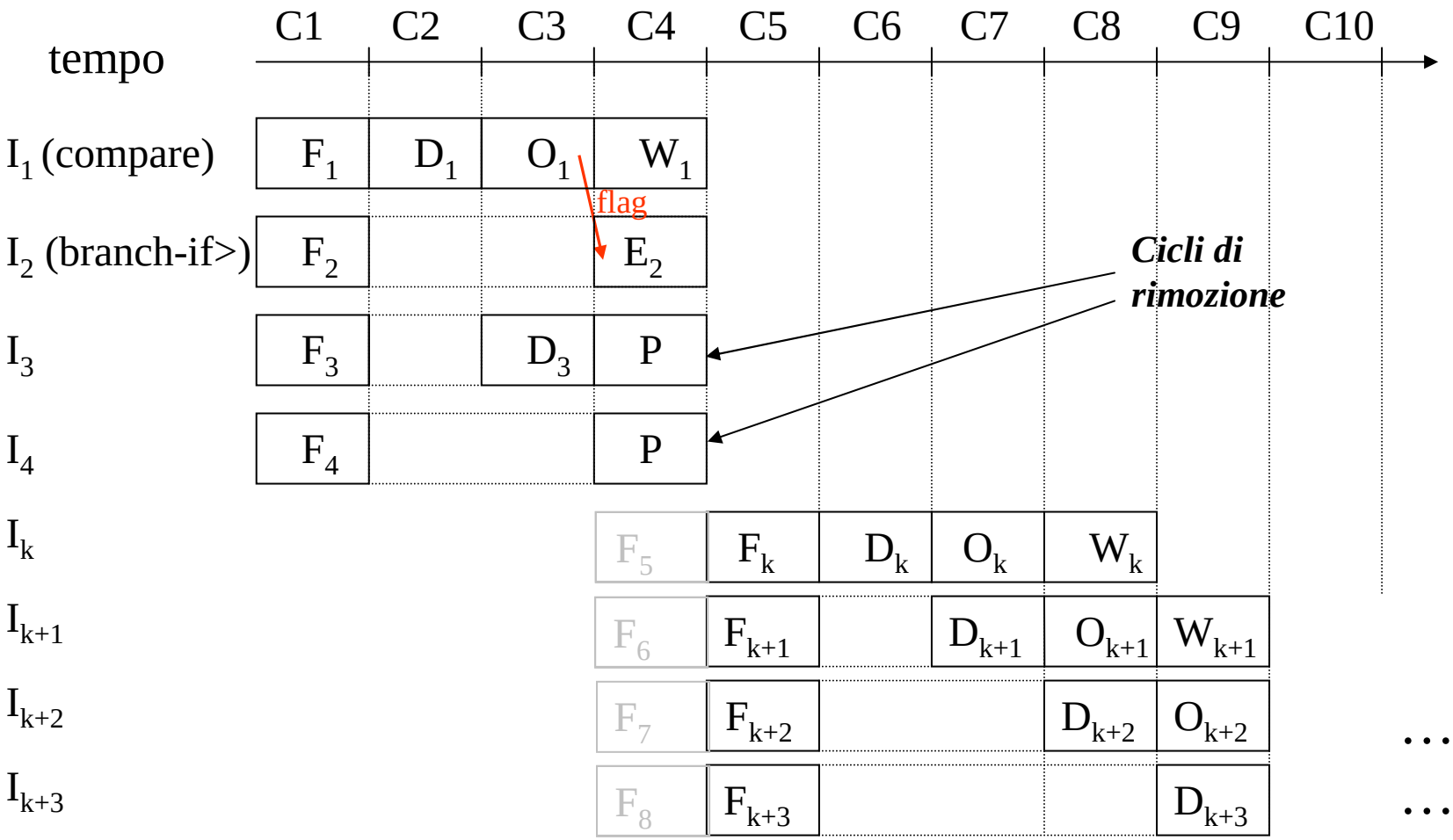
- Con pipeline a 4 stadi, in caso di predizione errata si ha una penalità di salto pari a 3 cicli
- Infatti, osservando la figura precedente si vede che l'istruzione di destinazione (in questo caso I_k) terminerebbe al ciclo 6 invece che al ciclo 9 nel caso ideale (senza stalli)
- Tuttavia, se si attua anche l'anticipo degli operandi (propagazione dei dati) è possibile ridurre la penalità a 2 cicli…vedi lucido successivo
- Si noti infine che, se il risultato della condizione di salto fosse previsto correttamente, la penalità di salto sarebbe –1 (come già analizzato in precedenza nel caso di pipeline a 2 stadi)

Esempio di predizione errata (pipeline a 4 stadi, prelievo di 4 istruzioni per ciclo)

Supponiamo semplice previsione di salto non effettuato

Assunzione: calcolo condizione di branch e calcolo nuovo indirizzo in un ciclo.

[CON UNITA' DI PROPAGAZIONE VERSO BRANCH UNIT!]



PIPELINE SUPERSCALARI E DINAMICHE

• ILP (Instruction Level Parallelism): parallelismo tra istruzioni. Come aumentarlo?
  - più stadi per la pipeline (riduzione T_{clock})
  - **multiple-issue**: più istruzioni cominciano l'esecuzione nello stesso ciclo!

⇒ CPI < 1

E' necessario:
- determinare quante e quali istruzioni lanciare in un ciclo di clock (**issue slot**)
- gestire le criticità sui dati e sul controllo

• Due modalità alternative per realizzare multiple-issue:
  - STATICA: decisioni su issue slot e gestione criticità demandate al compilatore
    [Raggruppa esplicitamente le istruzioni in modo che:
    - siano effettivamente eseguibili con le risorse HW del processore
    - dipendenze tra dati e stalli sono risolti]
  - DINAMICA: decisioni prese "a run-time" dall'hardware
    (ma il compilatore ha un ruolo nell'ottimizzare prestazioni, ad esempio schedulando il codice in modo da evitare dipendenze e consentire che più istruzioni possano essere lanciate in parallelo)

NB: sono possibili approcci "ibridi" [es. il compilatore decide issue slot, ma HW offre qualche supporto per la gestione delle dipendenze e criticità sul controllo]

Concetto di “speculazione”:

- Si “presuppone” il risultato di un’istruzione in modo da rimuovere la dipendenza da parte di altre istruzioni che vengono eseguite senza attendere il risultato stesso. P.es:
  - Predizione di salto:
    salto previsto effettuato o non effettuato ed esecuzione del codice
  - avendo *sw r1, offset(r2)* e *lw r3, offset(r4)* in sequenza,
    lw e sw eseguite in parallelo (o in ordine diverso) presupponendo non facciano riferimento allo stesso indirizzo
- Se la speculazione è errata, SW o HW devono “rimediare”:
  - se un gruppo di istruzioni non doveva essere eseguito, va eliminato
  - se lw si riferisce allo stesso indirizzo di sw, è necessario rimediare eseguendo la lw sul dato in memoria modificato da sw
  - se un’eccezione coinvolge istruzioni che non dovevano essere eseguite, è necessario eliminarne gli effetti

⇒ Soluzione software: procedure di “ripristino”

⇒ Soluzione hardware:
  - i risultati “speculativi” posti in un buffer e trasferiti effettivamente in registri (o memoria) quando è sicuro che predizione era corretta
  - bufferizzazione delle eccezioni, considerate quando è lecito farlo

Multiple issue statica

• Il compilatore codifica “pacchetti” di istruzioni da eseguire in parallelo, con vincoli relativamente restrittivi [in pratica, istruzioni che comprendono operazioni multiple!]

Esempio di riferimento (semplice 2-issue MIPS)

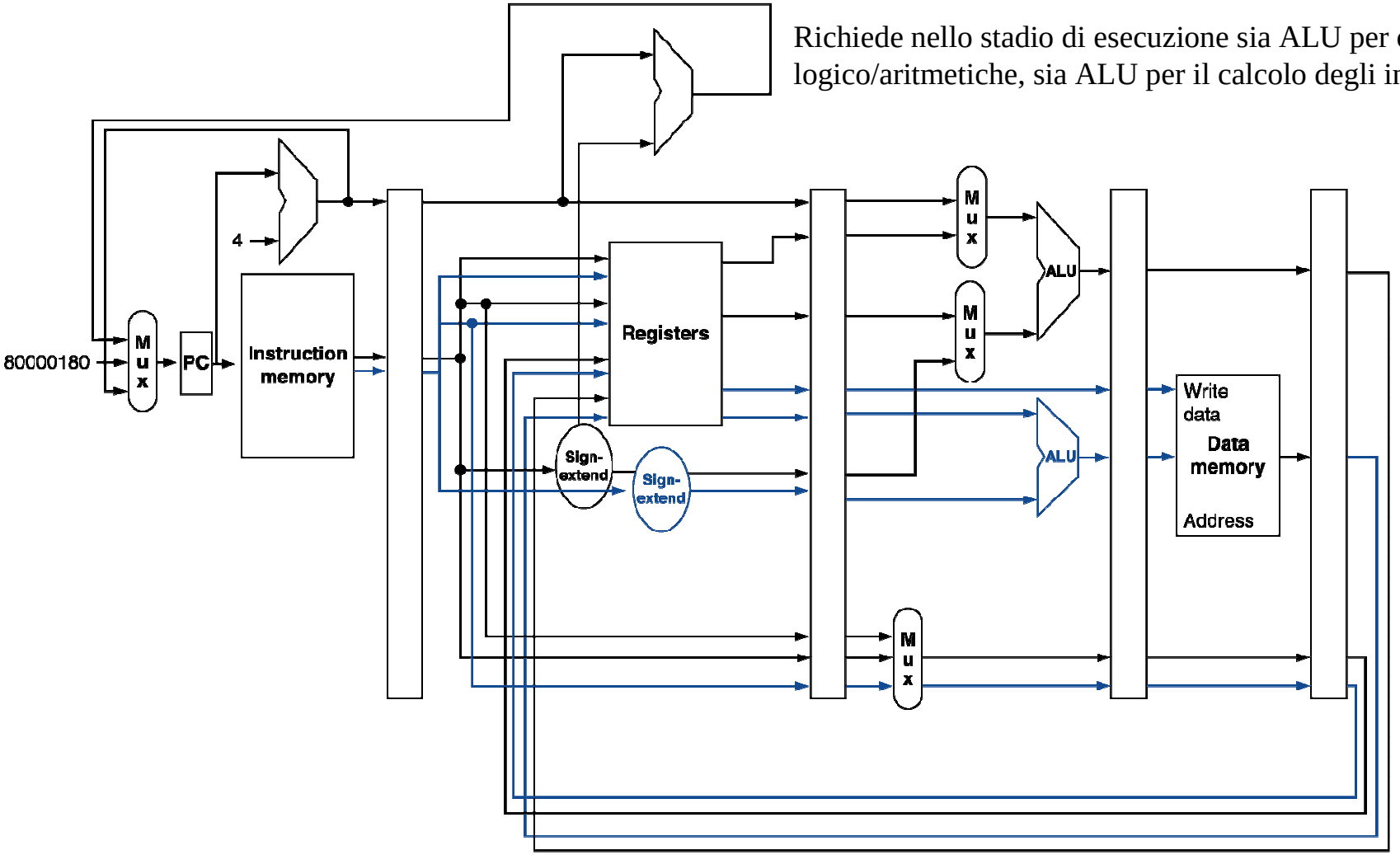
• Si vogliono eseguire in parallelo per ogni ciclo di clock:
  - un’istruzione aritmetico/logica o un salto condizionato
  - un’istruzione di trasferimento dati (lw o sw)
• Le istruzioni devono essere allineate e raggruppate due a due (64 bit); il compilatore inserisce NOP se non riesce a trovare la coppia di istruzioni

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

⇒ datapath del tipo…

Richiede modifiche al data path di “basso livello” (p.es. porte addizionali nel register file per leggere 4 registri e scrivere due registri in un ciclo di clock).

Richiede nello stadio di esecuzione sia ALU per operazioni logico/aritmetiche, sia ALU per il calcolo degli indirizzi



Esegue in parallelo:

istruzione logico/aritmetica – salto || accesso in memoria

Supporto dell'HARDWARE alle criticità:

- In alcuni approcci, il compilatore ha l'intera responsabilità sulla gestione (schedula il codice in modo opportuno)
- Qui, assumiamo che l'hardware gestisca criticità (con stallo e propagazione) tra slot diversi, mentre non gestisce nulla all'interno dei singoli pacchetti (es. se la seconda istruzione all'interno di uno slot dipende da un dato prodotto dalla prima, il valore da essa usato sarà quello precedente alla modifica)

  $\Rightarrow$ Anche se non indicate nel datapath, si assume che siano presenti le unità di propagazione dei dati e di gestione delle criticità "inter-slot"

Prestazioni

IDEALI: CPI = 0.5

Ma le penalità per le criticità sono maggiori!
(vedi prox lucido)

Esempio: criticità sui dati

```
add     $t0, $t1, $t2
sw      $s0, 20($t0)
```

No nello stesso pacchetto!
(necessario attendere un ciclo di clock!)

Altro esempio: criticità carica-e-usa

```
Istruzione TIPO-R o branch

lw      $s0, 20($t0)

add     $s2, $s2, $s2
sw      $t3, 20($s0)
```

Ho lo stallo dello slot per un ciclo di clock!

⇒ RUOLO DEL COMPILATORE CRUCIALE

Esempio di schedulazione superscalare del codice

[somma del valore in $2 agli elementi di un vettore]

```
Ciclo:  lw    $t0, 0($s1)
        addu  $t0, $t0, $s2
        sw    $t0, 0($s1)
        addi  $s1, $s1, -4
        bne   $s1, $zero, Ciclo
```

Per evitare il più possibile le condizioni di stallo, si può procedere così:

1: lw deve essere la prima
2: addu allora deve seguirla dopo 2 cicli di clock
3: sw deve venire dopo

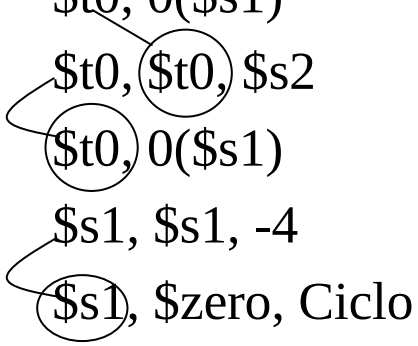
	ALU - beq	lw, sw	Cicli di clock
Ciclo:		lw $t0, 0($s1)	1
			2
	addu $t0, $t0, $s2		3
		sw $t0, 0($s1)	4

⇒ Dopo le prime tre istruzioni, schedulo le altre…

Esempio di schedulazione superscalare del codice

[somma del valore in $2 agli elementi di un vettore]

```
Ciclo:  lw    $t0, 0($s1)
        addu  $t0, $t0, $s2
        sw    $t0, 0($s1)
        addi  $s1, $s1, -4
        bne   $s1, $zero, Ciclo
```



Per evitare il più possibile le condizioni di stallo, si può procedere così:

1: lw deve essere la prima
2: addu allora deve seguirla dopo 2 cicli di clock
3: sw deve venire dopo

	ALU - beq	lw, sw	Cicli di clock
Ciclo:		lw $t0, 0($s1)	1
	addi $s1, $s1, -4		2
	addu $t0, $t0, $s2		3
	bne $s1, $zero, Ciclo	sw $t0, **4**($s1)	4

Si vede che:

- con due operazioni per ciclo di clock, idealmente CPI = 0.5
- invece, in questo caso si hanno 5 istruzioni in 4 cicli: CPI = 0.8

⇒ Occorrono tecniche sofisticate di compilazione, p.es. espansione dei cicli

```
Ciclo:  lw      $t0, 0($s1)
        addu    $t0, $t0, $s2
        sw      $t0, 0($s1)
        addi    $s1, $s1, -4
        bne     $s1, $zero, Ciclo
```

Assumendo che l'indice del ciclo sia multiplo di quattro:

- devo anticipare il più possibile le lw [creano lo stallo]: effettuo quattro lw come primo blocco di istruzioni, utilizzando 4 registri temporanei;
- pongo le quattro istruzioni di addu separate da due cicli di clock (per evitare lo stallo)
- sistemo le quattro istruzioni di sw subito sotto quelle di lw

⇒ Ottengo in prima battuta…

	ALU - beq	lw, sw	Cicli di clock
Ciclo:		lw $t0, 0($s1)	1
		lw $t1, -4($s1)	2
	addu $t0, $t0, $s2	lw $t2, -8($s1)	3
	addu $t1, $t1, $s2	lw $t3, -12($s1)	4
	addu $t2, $t2, $s2	sw $t0, 0($s1)	5
	addu $t3, $t3, $s2	sw $t1, -4($s1)	6
		sw $t2, -8($s1)	7
		sw $t3, -12($s1)	8

- Ora pongo negli slot liberi l'aggiornamento dell'indice [addi $s1, $s1, -16] e il salto bne, sistemando di conseguenza gli offset

⇒ Si ottiene…

	ALU - beq	lw, sw	Cicli di clock
Ciclo:	addi $s1, $s1, -16	lw $t0, 0($s1)	1
		lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t3, $s2	sw $t1, 12($s1)	6
		sw $t2, 8($s1)	7
	bne $s1, $zero, Ciclo	sw $t3, 4($s1)	8

- in 8 cicli di clock 14 istruzioni utili:

  CPI = 8/14 = 0.57 (contro 0.8 di prima e 0.5 ideale)

- abbiamo però utilizzato 4 registri temporanei ($t0--$t3) al posto di uno ($t0) e abbiamo un codice più lungo!

Esempio reale

Architettura IA-64 a 64 bit, detta EPIC (Explicitly Parallel Instruction Computer)

- ciascuna “macroistruzione” è lunga 128 bit e contiene 3 istruzioni macchina, ciascuna di 41 bit (gli altri 5 bit specificano gli stop e le specifiche unità di esecuzione utilizzate da ciascuna delle tre istruzioni)
- Il compilatore, utilizzando indicazioni dette “stop” può dividere le istruzioni in “gruppi”, ciascuno dei quali contiene istruzioni senza dipendenze sui dati tra loro
  (informazione utilizzata dall’hardware che può eseguire le istruzioni in parallelo)

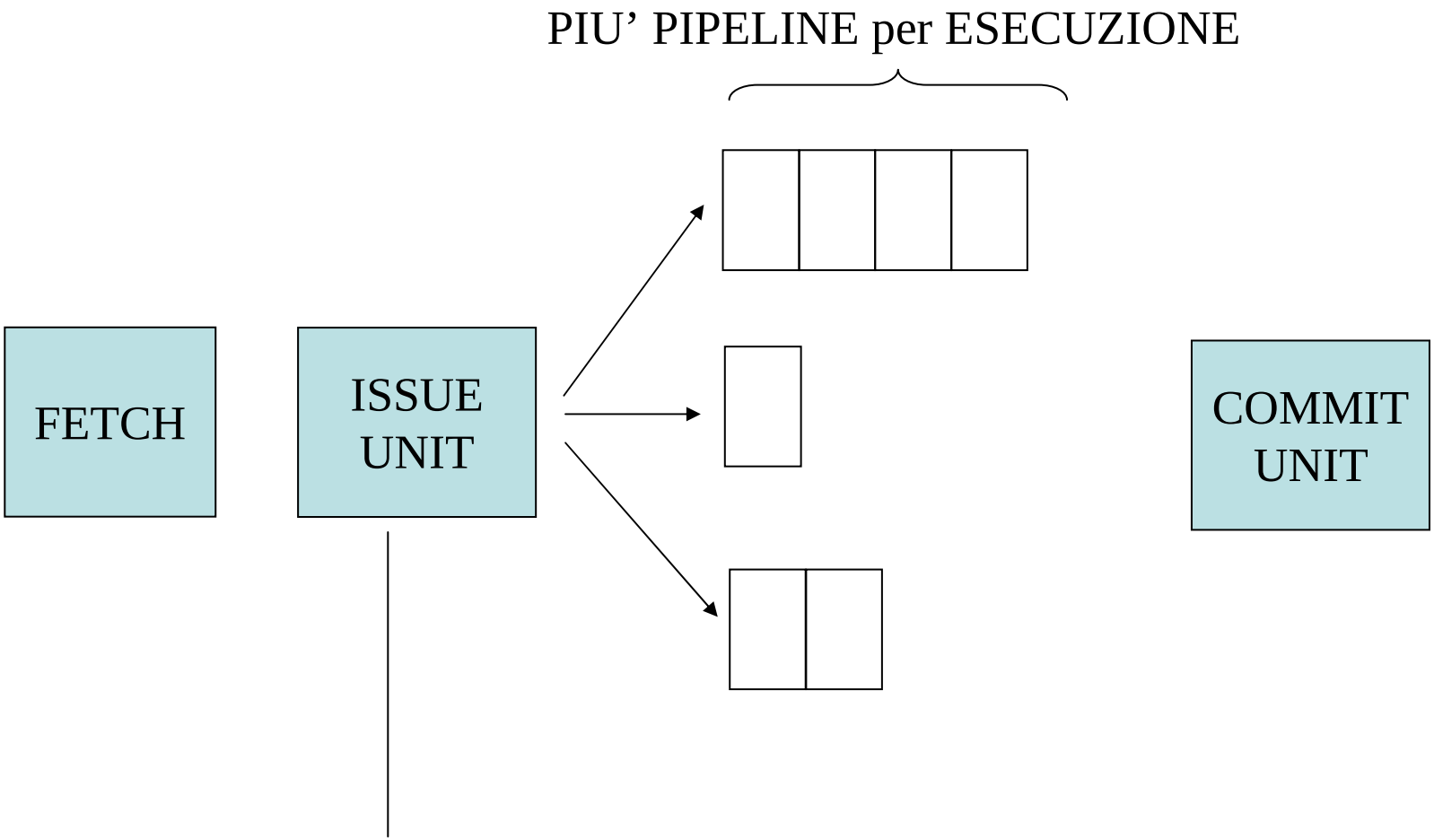
Multiple issue dinamica

• Come visto, nelle pipeline statiche il compilatore deve gestire completamente alcuni tipi di criticità e soddisfare vincoli che dipendono da come è fatta la pipeline [MIPS visto in precedenza: due istruzioni di tipo diverso – HW non fornisce supporto per le istruzioni che fanno parte dello stesso issue slot]

⇒ Se cambia la struttura del processore, il codice deve essere ricompilato!

• IDEA: è l'hardware che decide quante (e quali) istruzioni sono eseguite in un ciclo di clock, smistandole nella pipeline in grado di eseguirle

⇒ Le diverse istruzioni possono completare la loro esecuzione in un ordine diverso rispetto a quello del programma (le pipeline possono avere un numero di stadi diverso)



Due modalità: smistamento in ordine / smistamento fuori ordine

Pipeline superscalare “semplice” (smistamento in ordine)

HW è in grado di decidere il numero delle istruzioni successive che nel prossimo ciclo di clock cominciano l’esecuzione, in modo da garantire una esecuzione corretta del codice. L’ordine delle istruzioni nel programma è rispettato (nessuna istruzione “scavalca” la precedente

[in pratica, è come se nella pipeline statica inserisse NOP automaticamente]

⇒ Non richiede la ricompilazione del codice

(ma il compilatore può contribuire a massimizzare prestazioni!)

Non molto efficiente nella gestione delle dipendenze…

Pipeline superscalare “con schedulazione dinamica del codice”

… un esempio…

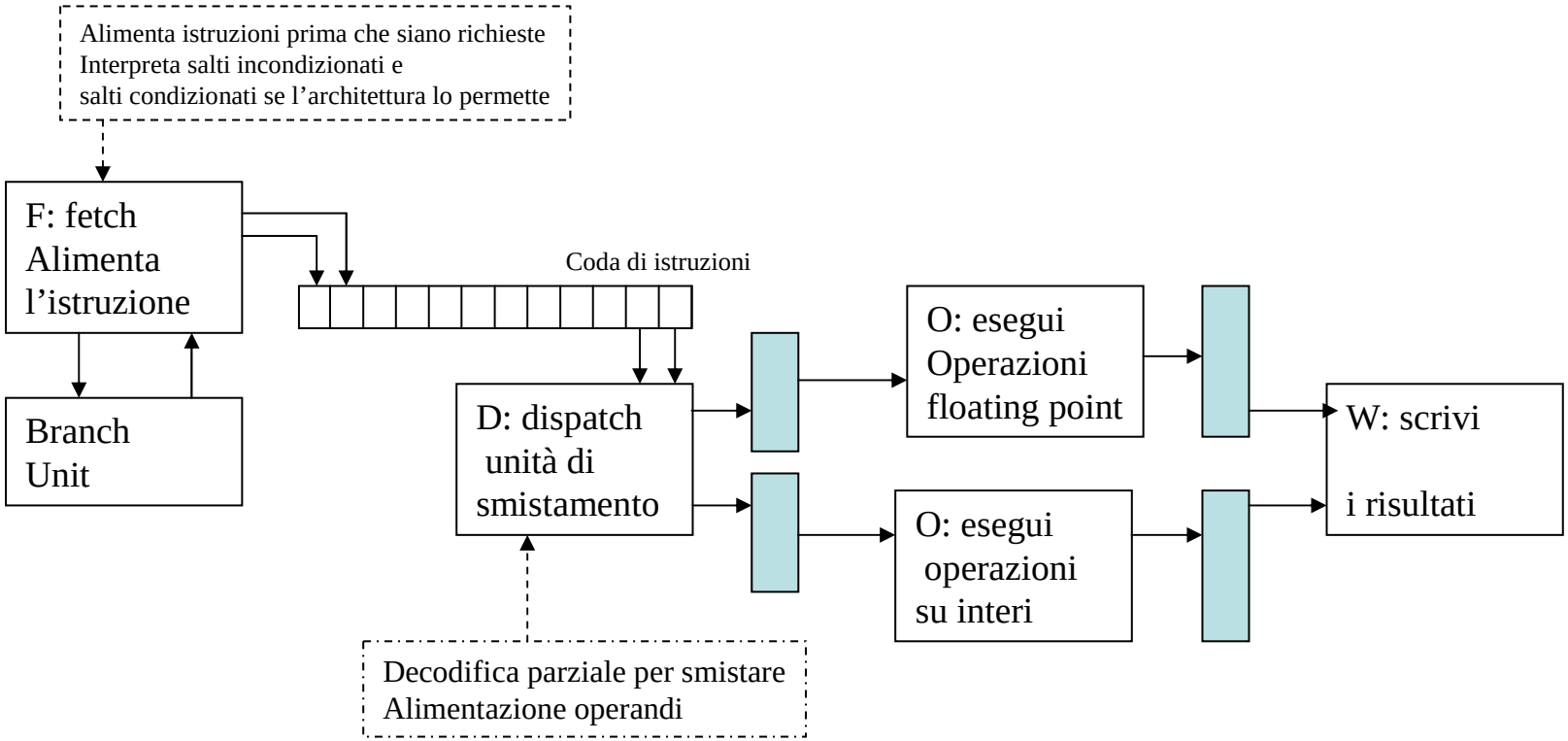
```
lw      $t0, 20($t2)
addu    $t1, $t0, $t2
sub     $s4, $s4, $t3
slti    $t5, $s4, 20
```

lw causa stallo di addu per un ciclo di clock – inoltre, se ho un miss di cache, i cicli di clock persi da lw si ripercuotono non solo su addu, ma bloccano anche sub e slti che potrebbero essere eseguite
[non hanno dipendenze rispetto a lw e addu]

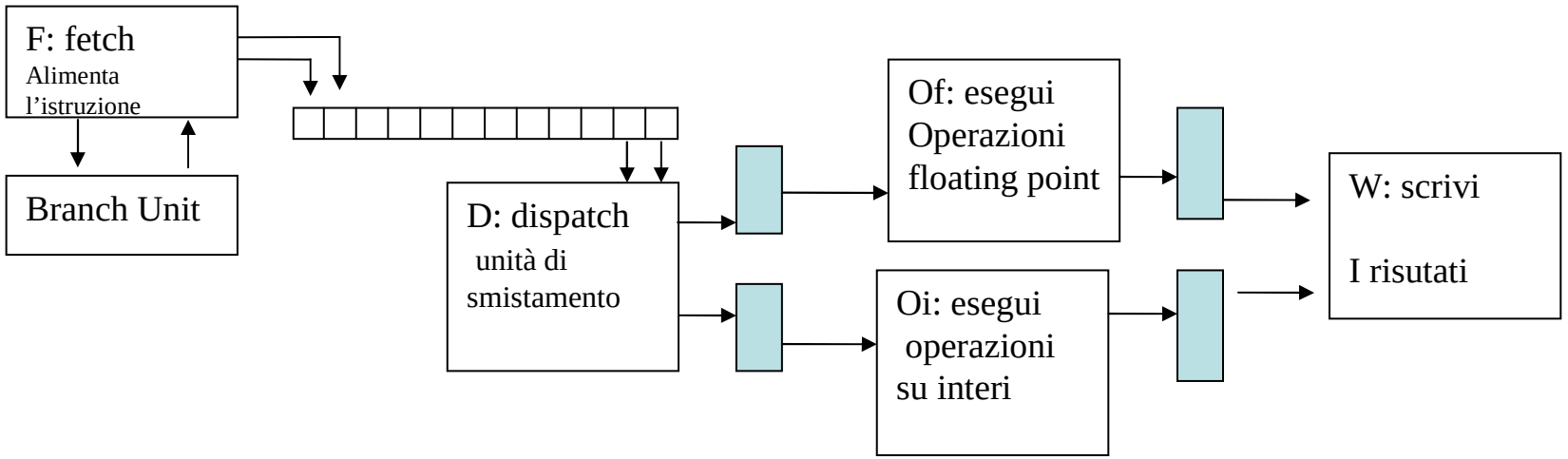
⇒ Per sfruttare i vantaggi servono:

- compilatori molto sofisticati, oppure
- tecniche di schedulazione hardware dinamica:
  pipeline capaci di schedulare dinamicamente le istruzioni, ovvero sub e slti possono procedere!

Realizzazione (supponiamo smistamento in ordine)

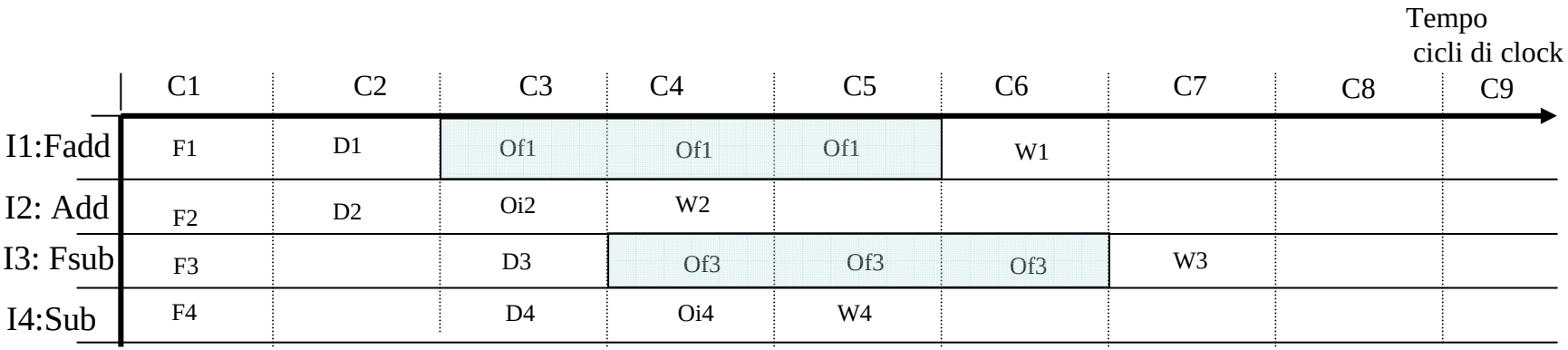


- Operazioni indipendenti possono procedere in parallelo [unità di smistamento può smistare più istruzioni in un ciclo di clock]
- Ciascuna unità O, a seconda della complessità dell'operazione, può essere costituita da più stadi e impiegare quindi diversi cicli di clock [p.es. aritmetica intera vs. virgola mobile]



P.es.

- 4 istruzioni per ciclo di clock prelevate (fetch) dalla cache istruzioni
- Fadd e Fsub sono in virgola mobile [3 stadi per Of], add e sub intere [1 stadio per Oi]
- 2 istruzioni per ciclo di clock smistate da unità D



NB: istruzioni <u>smistate</u> **in ordine** [secondo la sequenza del programma]
istruzioni (in questo esempio) **<u>completate</u> fuori ordine**

• Ovviamente, vi saranno in pratica diverse (più di due) unità di esecuzione, sia per le varie tipologie di istruzioni sia per eseguire più istruzioni dello stesso tipo in uno stesso ciclo di clock [p.es. più unità per aritmetica intera]

Difficoltà per il completamento fuori ordine

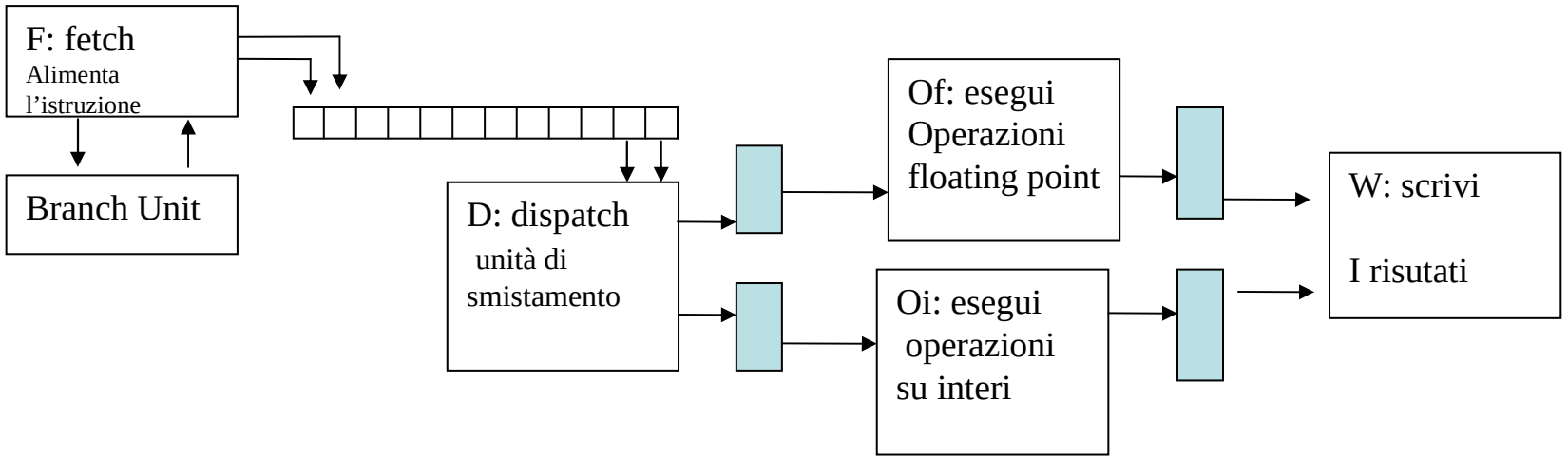
- E' necessario rispettare le dipendenze (tramite propagazione e stallo)
- E' necessario garantire che i valori scritti nei registri corrispondano comunque alla semantica del programma [istruzione precedente non deve "sovrascrivere" un'istruzione seguente!]
- Il problema più grosso si ha con le **eccezioni…**

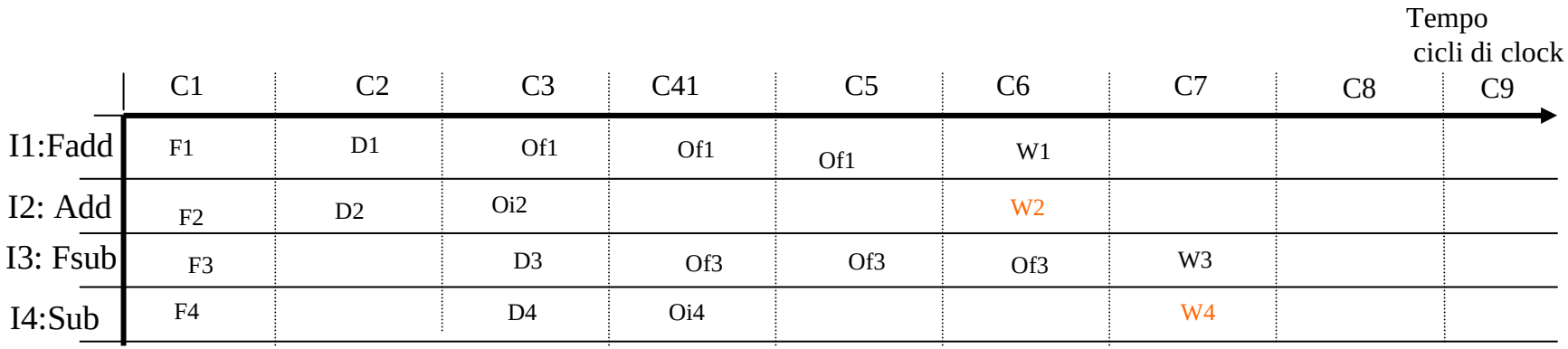
→ Nell'esempio precedente: se un'eccezione accade in Of1 nel ciclo 5, è *eccezione imprecisa* (W2 già avvenuto)

⇨ Lo stato non è coerente, perché deriva da modifiche da parte di istruzioni seguenti quella interrotta!

⇨ RIMEDIO: **completamento in ordine**

Macchine con più di una unità di esecuzione:
Rimedio al problema del 'fuori ordine' in una macchina con coda , alimentazione di 4 istruzioni contemporaneamente dalla cache: ritardare le scritture





Se un'eccezione accade in Of1 nel ciclo 5, è *eccezione precisa* poiché gli unici registri aggiornati sono quelli scritti da istruzioni precedenti I1: è quindi possibile scartare le istruzioni logicamente successive a I1.

COME?

- Usare degli stalli nelle pipeline più corte (poco efficiente)
- Usare una **coda di completamento (reorder buffer)**

• La Commit Unit dispone di una **coda di completamento** che memorizza l'ordine logico delle istruzioni: il completamento è effettuato seguendo la coda

(In-order commit)

•L'unità di smistamento considera le istruzioni in ordine *(In-order issue)* e per ogni istruzione:

- prenota una posizione nella coda di completamento

(segue l'ordine definito nel programma)

- smista l'istruzione nella pipeline in grado di eseguirla

• Ciascuna istruzione non scrive i risultati direttamente nei registri, ma li bufferizza in registri temporanei

• Commit-Unit trasferisce i risultati delle istruzioni nei rispettivi registri permanenti "in ordine", secondo l'ordine fissato dalla coda (liberando le relative posizioni dalla coda di completamento).

Gestione della speculazione

- Previsione di salto:

  vengono eseguite le istruzioni predette, ma non si completano finché non si controlla la condizione effettiva di salto – eventualmente possono essere scartate

- Trattamento eccezioni:

  - possono essere scartate le istruzioni seguenti l'eccezione (non ancora completate)
  - le eccezioni non sono servite su "base speculativa"

ESEMPIO:
POWERPC 603

LSU: Unità load/store
IU: Unità Aritmetica Intera
FPU: Unità Floating Point
SRU: Unità dei registri di Sistema

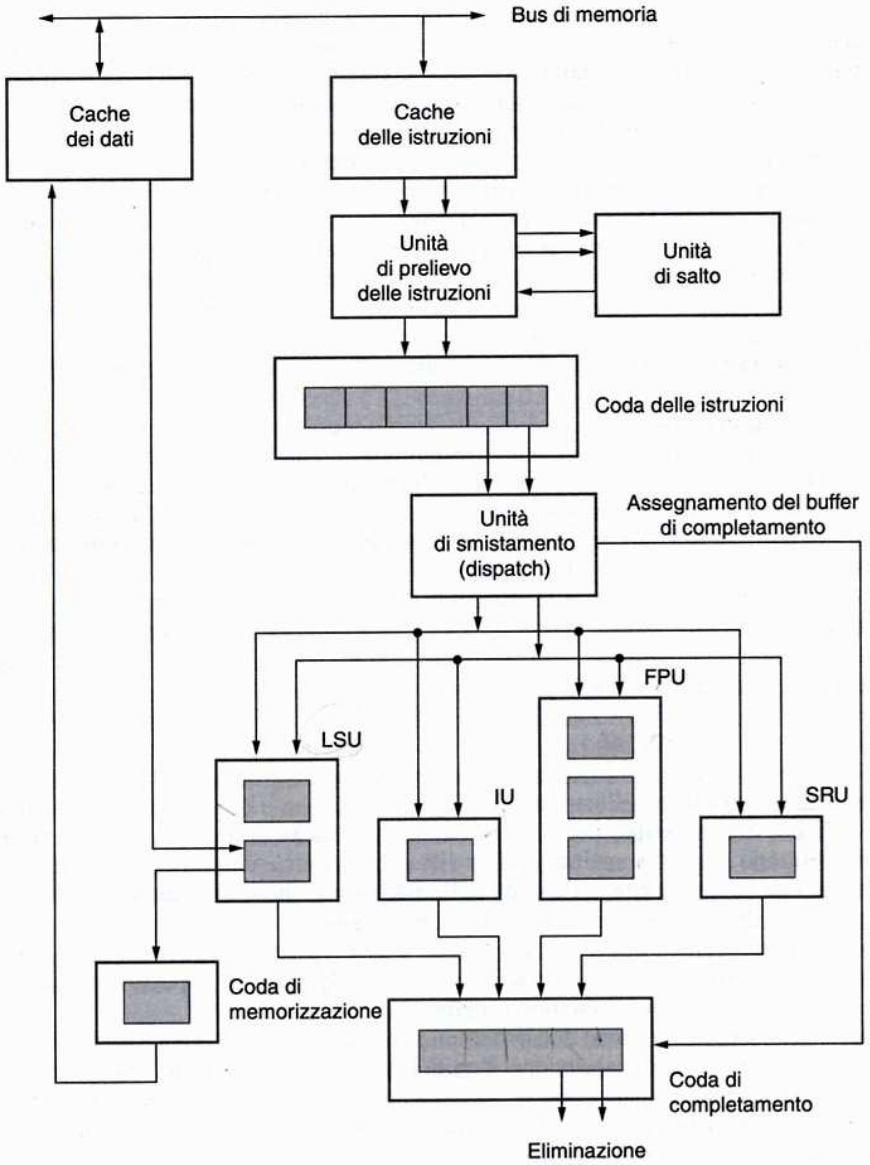
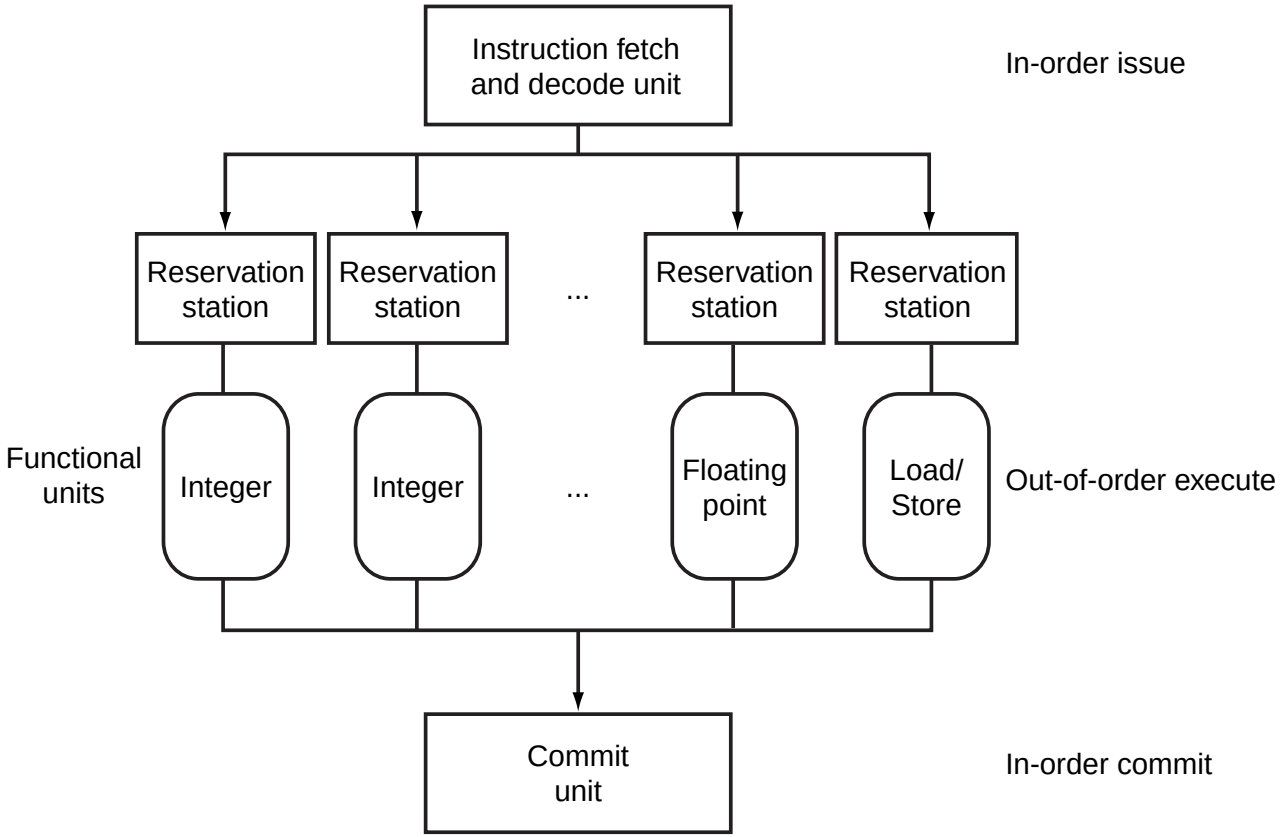
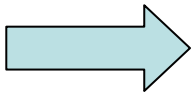


Figura 7.22 Flusso di esecuzione delle istruzioni nel PowerPC 603.

Schedulazione dinamica



- Ogni unità funzionale dispone di una “reservation station”, in grado di memorizzare gli operandi disponibili e l’operazione da svolgere
- Quando un’istruzione è smistata, gli operandi disponibili sono prelevati dal register file (o dal reorder buffer). Gli operandi non disponibili sono “operandi pendenti” nella reservation station
- L’esecuzione dell’istruzione può procedere solo se tutti gli operandi sono disponibili nella stazione (in caso di dipendenze può essere necessario attendere); gli operandi non sono prelevati dai registri, ma dalla reservation station
- Non appena un risultato è prodotto da un’unità funzionale, può essere scritto direttamente nelle reservation station, bypassando i registri (e permettendo eventualmente ad un’istruzione di procedere)



Le istruzioni non procedono secondo l’ordine del programma, ma secondo la disponibilità degli operandi

ESEMPIO: INTEL PENTIUM 4