Architettura degli Elaboratori II

Corso di Laurea Triennale in Informatica

Università degli Studi di Milano

Dipartimento di Informatica “Giovanni Degli Antoni”

Edizione 2 (Cognomi H-Z), A.A. 2021-2022, Nicola.Basilico@unimi.it

Informazioni sull’insegnamento

Organizzazione del corso

Parte di teoria (36 ore, Basilico)

- Lunedì 11:00 – 12:30 e Mercoledì, ore 9:00-10:30
- Lezioni in modalità mista con live streaming su Microsoft Teams (team code lfwcu31)
- Le lezioni (solo la parte di spiegazione) saranno registrate e rese disponibili sul sito del corso

Parte di laboratorio (24 ore, Tarini, Rivolta)

- Mercoledì, 13:30-17:30
- Il laboratorio inizia Mercoledì 16marzo le info organizzative saranno pubblicate sulla pagina web del corso

- Sito web (teoria e lab): https://nbasilicoae2.ariel.ctu.unimi.it/

Programma

Per la parte di teoria:

- CPU a singolo ciclo e a ciclo multiplo: pipeline, criticità, interrupt, eccezioni e loro gestione;
- memoria statica e dinamica, memoria virtuale, gerarchia di memoria;
- input/output;
- metriche di valutazione delle prestazioni.

Per la parte di laboratorio, programmazione in Assembly:

- uso della memoria;
- system calls;
- controllo di flusso;
- definizione e chiamata a procedure;
- gestione delle eccezioni.

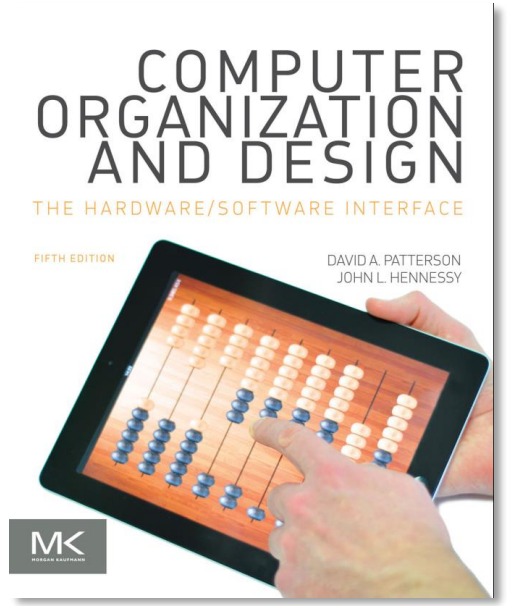
Esame

- **Esame di teoria**: prova scritta
  - Se a distanza viene svolta con **Exam.net** (o con sistemi equivalenti)
  - Orale solo in casi eccezionali (problemi di consegna, sospetto copiatura o altro) e comunque solo di verifica
- **Esame laboratorio**: prova scritta (esercizi di programmazione Assembly)
  - I dettagli saranno forniti dai docenti di laboratorio
- **Voto finale**
  - Voto in trentesimi per entrambe le prove, sufficienza ≥ 18
  - Vanno superate con sufficienza nell'arco di 6 mesi (o 3 appelli successivi)
  - Voto finale media pesata di teoria (peso $\frac{2}{3}$) e laboratorio (peso $\frac{1}{3}$)
  - Una volta superate entrambe le prove il voto viene verbalizzato dal docente, non serve fare richiesta

Materiale didattico

Testi di riferimento

- Computer organization and design: the hardware/software interface *di David A. Patterson e John L. Hennessy, Morgan Kaufmann* (Edizione italiana: «Struttura e Progetto dei Calcolatori», *Zanichelli)*



Materiale aggiuntivo

- Slide e video delle lezioni, esercizi proposti in laboratorio con esempi di soluzioni
- Temi d’esame resi disponibili man mano che verranno svolti

Domande frequenti (con risposta «No»)

- **Posso cambiare Edizione?** No, a meno di motivazioni ragionevolmente giustificate e comunque dopo aver avuto l'ok da entrambi i docenti di teoria
- **Può pubblicare le slide prima della lezione?** No! Le slide verranno pubblicato solo dopo essere state mostrate a lezione
- **Posso diffondere i materiali prodotti dal docente su canali «non ufficiali»?** Assolutamente no! Le registrazioni, le slide e eventuale altro materiale del corso resteranno sempre disponibili su Ariel fino all'inizio dell'edizione del prossimo A.A., durante il quale ne verrà rilasciata una nuova versione
- **Ho passato lo scritto e mi piacerebbe provare ad alzare il voto con un orale, posso farlo?** No, l'esame è solo scritto

Prima di iniziare …

- A distanza tenere video e microfono **spenti**, usare il proprio nome e cognome, per le domande scrivere in chat
- I concetti presentati nel corso vanno:
  - Assimilati **gradualmente** *(no «netflixing» a 1.75x la notte prima dell'esame)*
  - Rielaborati *(**dormici** su, fatti delle domande)*
  - Applicati *(metti in pratica quello che impari su un problema che conosci)*
- Le slide non bastano! Usare il libro

Architettura degli Elaboratori II

Corso di Laurea Triennale in Informatica

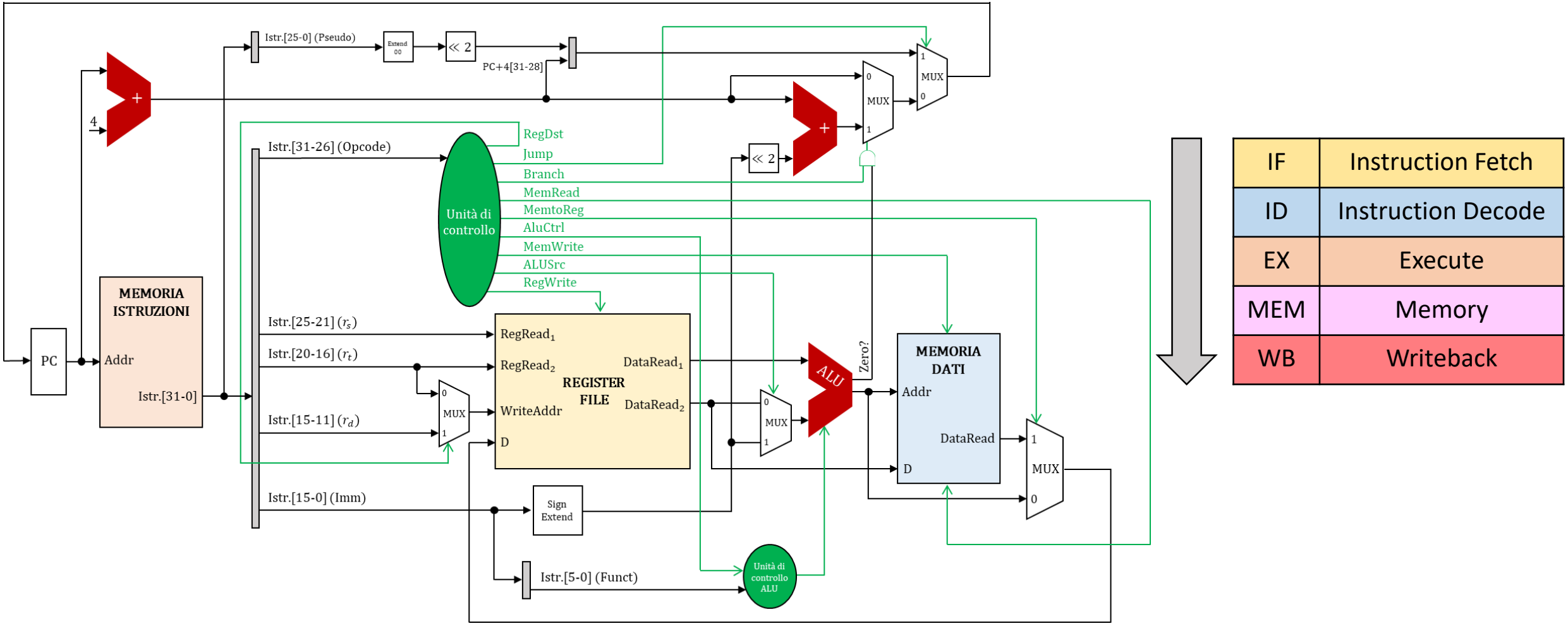
Università degli Studi di Milano

Dipartimento di Informatica “Giovanni Degli Antoni”

Edizione 2 (Cognomi H-Z), A.A. 2021-2022, Nicola.Basilico@unimi.it

CPU a ciclo multiplo

La nostra precedente CPU (singolo ciclo)



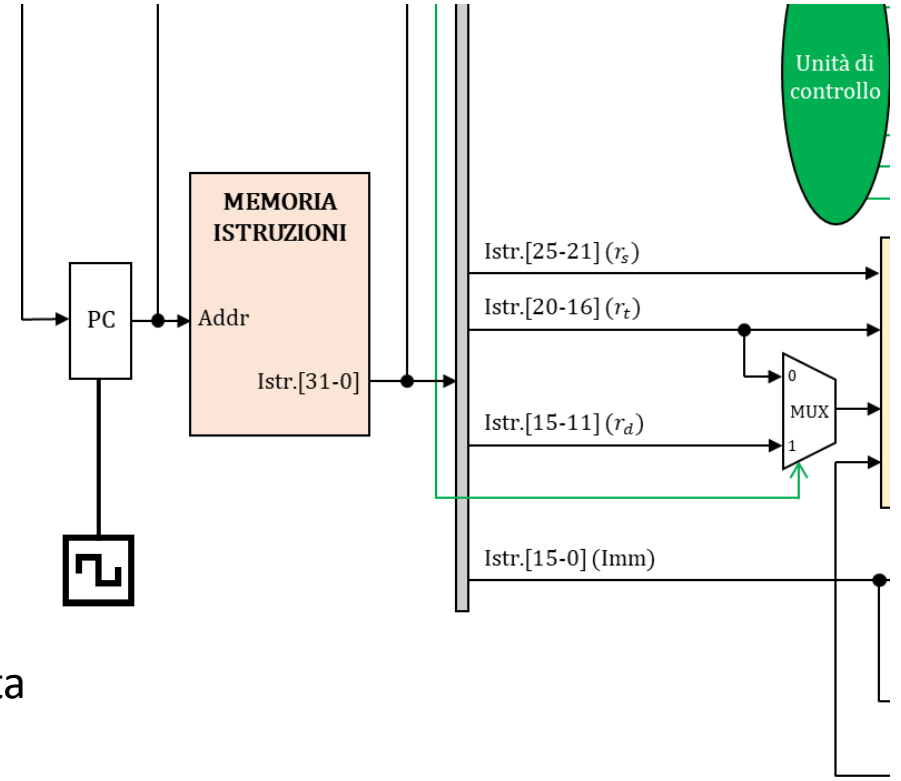
IF	Instruction Fetch
ID	Instruction Decode
EX	Execute
MEM	Memory
WB	Writeback

Pregi e difetti della CPU a singolo ciclo

- In sintesi: una implementazione semplice ma inefficiente
- **Semplicità**
  - L'esecuzione avviene in un **singolo ciclo di clock**, i segnali attraversano un datapath diverso a seconda dell'istruzione che si deve eseguire
  - L'unità di controllo (che manovra gli scambi sul datapath) è una funzione combinatoria
- **Inefficienza**
  - Ogni istruzione richiede un tempo T_{ck} per essere eseguita, dove T_{ck} è la durata del ciclo di clock, ma alcune istruzioni potrebbero richiedere di meno!
  - Replicazione dei componenti
    - Se una istruzione ha bisogno di svolgere due operazioni aritmetiche (es. beq) servono due ALU
    - Due unità di memoria: una per il fetch dell'istruzione e una per la lettura/scrittura dei dati

Pregi e difetti della CPU a singolo ciclo

- Il **Program Counter** (PC) riceve il clock
- Il clock determina la scrittura di un nuovo indirizzo nel PC da cui poi scaturiscono, una dopo l'altra, le varie fasi dell'esecuzione: fetch (IF), decode (ID), execute (EX), memory (MEM) e writeback (WB)
- Tutte e 5 le fasi devono concludersi **prima** del prossimo ciclo (entro un tempo T_{ck}), che innescherà l'esecuzione della prossima istruzione



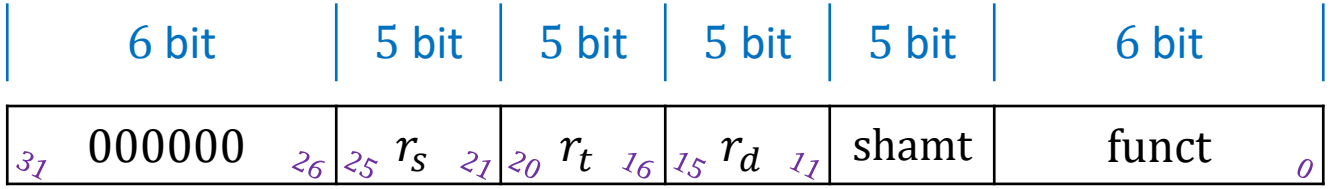
- **Vincolo di progettazione**: T_{ck} va dimensionato sull'istruzione più lenta
- Questa è l'istruzione il cui datapath ha il cammino critico **più lungo**

Prestazioni della CPU a singolo ciclo

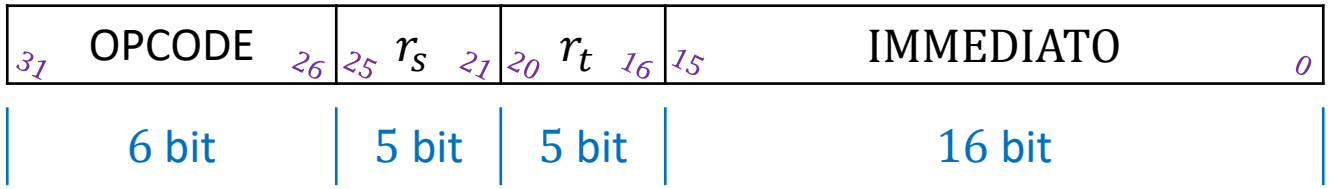
- La nostra CPU supporta 5 tipi di istruzioni
  - Istruzioni Aritmetico-Logiche (A/L)
  - Istruzioni di accesso alla memoria dati: (lw, sw)
  - Salti condizionati (beq)
  - Salti non condizionati (j)
- Il cammino critico della CPU è il più lungo tra i cammini critici delle istruzioni
- Quale è?

Ripasso sulla codifica delle istruzioni

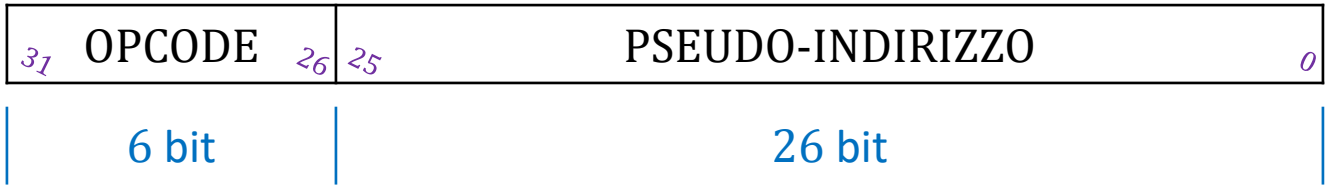
Formato R:
es. add rd, rs, rt



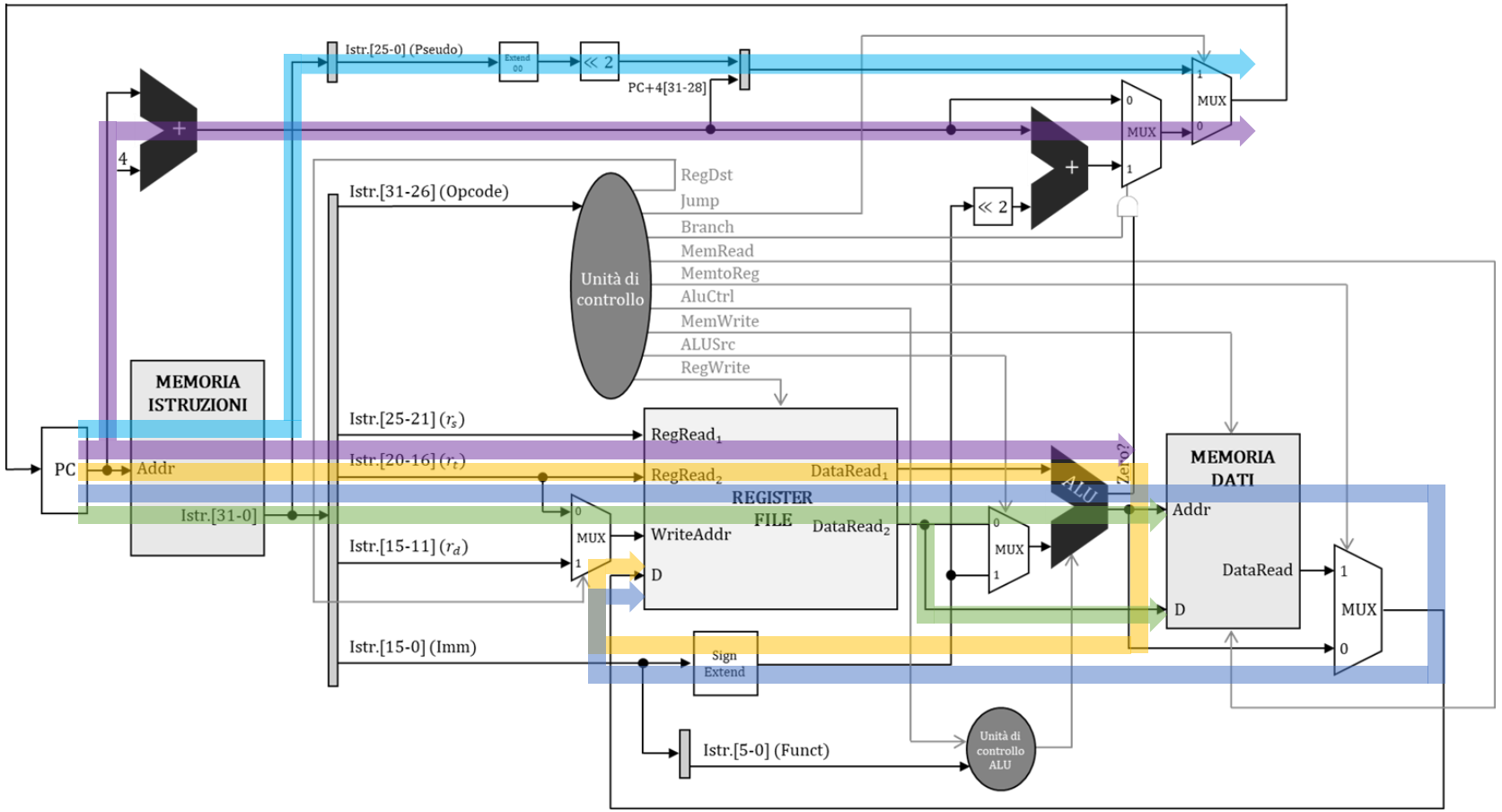
Formato I:
beq rs, rt, OFFSET
lw rt, OFFSET(rs)
sw rt, OFFSET(rs)



Formato J:
J ADDRESS



Prestazioni della CPU a singolo ciclo



- La lw è l'istruzione che richiede il cammino critico più lungo
- Sono necessarie tutte e 5 le fasi
- Nessuna istruzione potrà essere eseguita in un tempo inferiore a quello della lw!

Cammini seguiti sul data path (e fasi necessarie):

- Istruzione Aritmetico/Logica: IF→ID → EX → WB
- Accesso a memoria in lettura (lw): IF→ID → EX → MEM → WB
- Accesso a memoria in scrittura (sw): IF→ID → EX → MEM
- Salto condizionato (beq): IF→ID → EX
- Salto non condizionato (j): IF→ID

Prestazioni della CPU a singolo ciclo

- Assumiamo che queste operazioni abbiano la seguente durata
  - Accesso a memoria (lettura o scrittura): $2\ ns$
  - Operazioni A/L: $2\ ns$
  - Accesso al Register File: $1\ ns$
  - Le altre operazioni assumiamo che siano trascurabili

Durate minime:

- Istruzione Aritmetico/Logica: $2 + 1 + 2 + 1 = 6$ (IF + ID + EX + WB)
- Accesso a memoria in lettura (lw): $2 + 1 + 2 + 2 + 1 = 8$ (IF + ID + EX + MEM + WB)
- Accesso a memoria in scrittura (sw): $2 + 1 + 2 + 2 = 7$ (IF + ID + EX + MEM)
- Salto condizionato (beq): $2 + 1 + 2 = 5$ (IF + ID + EX)
- Salto non condizionato (j): $2 + 1 = 3$ (IF + ID)

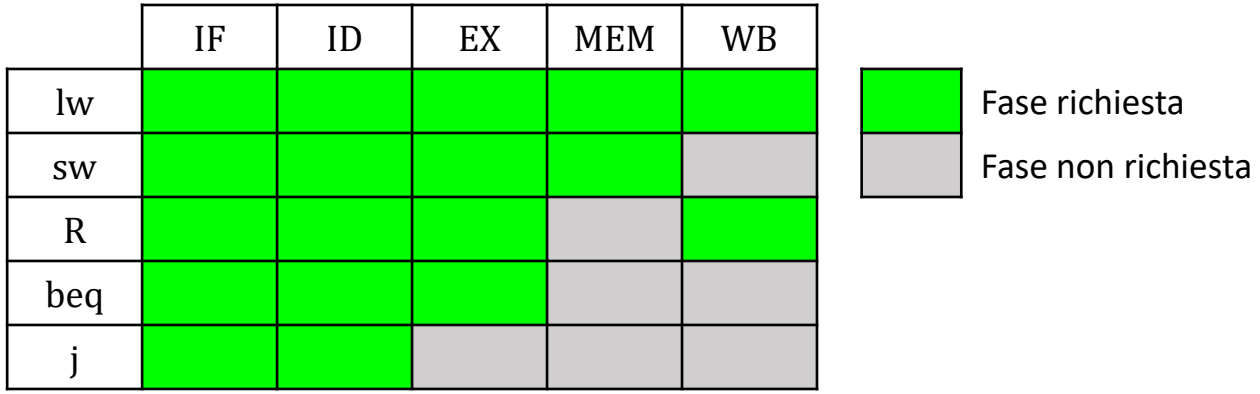
Prestazioni della CPU a singolo ciclo

	A/L	lw	sw	beq	j	Durata media istruzione in CPU singolo ciclo	Durata media istruzione in CPU ciclo variabile
Durata minima	6	8	7	5	3		
Programma 1	30%	5%	20%	30%	15%	8	5.55
Programma 2	50%	40%	5%	2%	3%	8	6.74

- In una CPU a singolo ciclo tutte le istruzioni hanno durata pari a quella massima, questa durata definisce anche la durata del ciclo di clock, in questo esempio $T_{ck} = 8$ $(0.125\ GHz)$
- Se potessimo costruire una CPU «ideale» dove la durata del ciclo di clock fosse variabile le performance potrebbero migliorare

CPU a ciclo multiplo

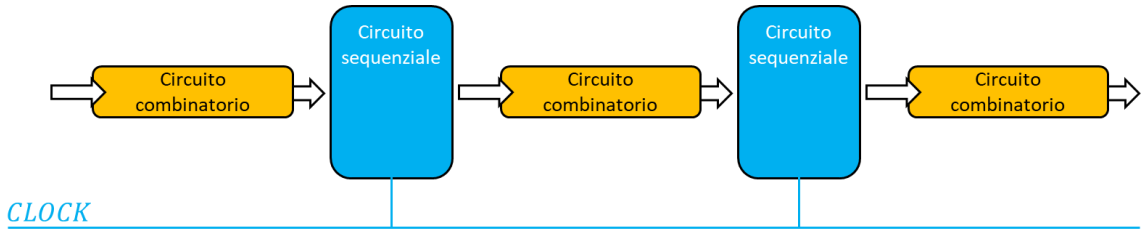
- L'esecuzione di una istruzione è organizzata in sotto-fasi (le 5 fasi che conosciamo)
- **Idea:**
  - Costruiamo una CPU che, in un ciclo di clock, esegua una singola sotto-fase anziché tutte e 5
  - Consentiamo a diverse istruzioni di impegnare la CPU per un **numero diverso di cicli di clock**
- Performance migliori: ogni istruzione richiederà un numero di verso di cicli di clock, solo quelli delle sotto-fasi che necessita



- Costi ridotti: un componente che serve a più sotto-fasi può essere usato in cicli di clock diversi, non serve replicarlo

CPU a ciclo multiplo

- Per poter distribuire l'esecuzione di una istruzione su più cicli di clock, la CPU deve
  1. Tenere traccia dell'esecuzione di ogni istruzione attraverso diversi cicli di clock, sincronizzare le sotto-fasi
     - *Quali sotto-fasi di questa istruzione ho eseguito? Quali restano da eseguire per completarla?*
  2. Una sotto-fase può necessitare di informazioni elaborate in sotto-fasi diverse:
     - *MEM e WB necessitano, rispettivamente di un indirizzo e di un dato calcolato in EX*
     - *EX necessita del contenuto di registri acquisito dal Register File in ID*
  - **Fasi diverse** avvengono, per definizione, in **cicli di clock diversi**: il passaggio di informazioni tra sotto-fasi necessita di una **memoria**
  - **Serve una logica sequenziale!**



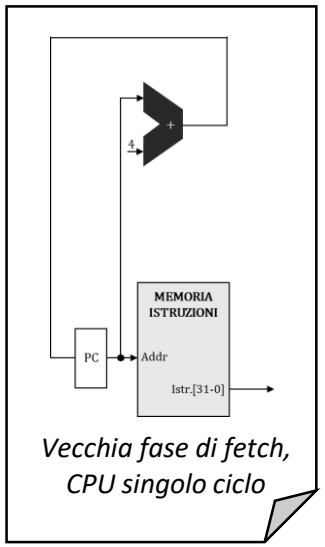
Conseguenze progettuali:

1. La control unit della CPU sarà una macchina a stati finiti (FSM)
2. Il datapath conterrà dei registri di memoria per il passaggio delle info tra sotto-fasi: **registri di transizione**

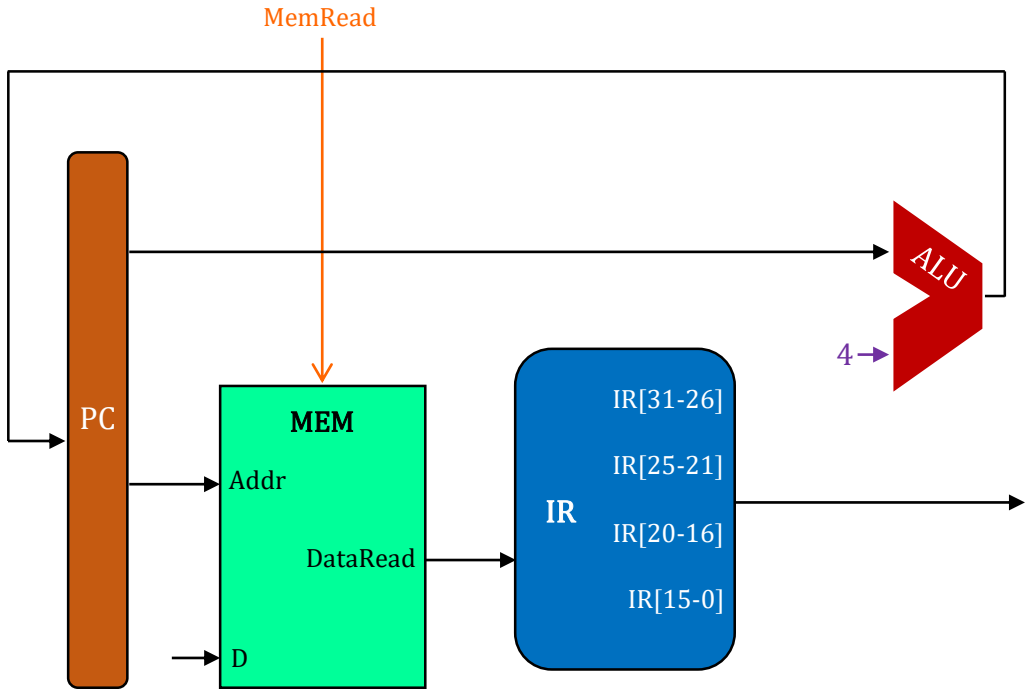
- ***Ricorda!*** CPU = **Datapath (percorsi possibili)** + **Logica di controllo (manovrare gli scambi su quei percorsi)**

CPU a ciclo multiplo: IF

- Ridefiniamo il sotto-circuito per la fase di fetch



Vecchia fase di fetch, CPU singolo ciclo

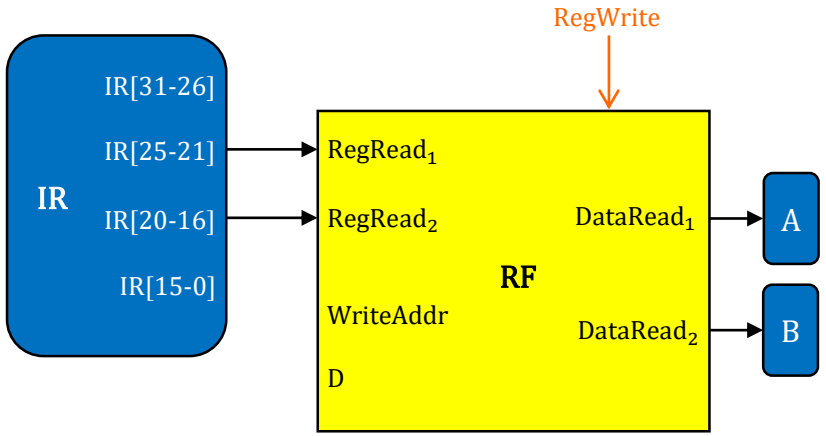


- **PC (Program Counter)**: contiene l'indirizzo dell'istruzione corrente
- **MEM (Memory)**: **unica** unità di memoria con **dati e istruzioni**, non serve più averli in due componenti diversi perché le fasi di IF e MEM non avvengono più nello stesso ciclo di clock!
- **IR (Instruction Register)**: contiene l'istruzione corrente trasferita dalla memoria, è il **registro di transizione** per le sotto-fasi successive
- **ALU**: non è un sommatore dedicato, ma la ALU della CPU, la fase di EX avviene in un ciclo di clock diverso quindi il componente è libero ed usabile per calcolare l'indirizzo della prossima istruzione

- È necessario il segnale di controllo MemRead, nella fase IF è posto a 1 per indicare che la memoria deve lavorare in modalità **lettura di un dato** (i 32 bit che codificano l'istruzione indirizzata da PC)

CPU a ciclo multiplo: ID

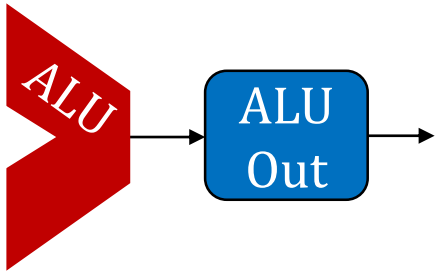
- La fase di decode avviene nel ciclo di clock successivo a quello in cui è avvenuta IF
- Quando la CPU passa a questa fase, assume di avere in IR i 32 bit dell'istruzione da decodificare
- Da IR legge gli indirizzi dei registri rs (IR[20-16]) e rt (IR[25-21]) che vengono dati in input alle porte di lettura del Register File
- Indipendentemente da RegWrite, il RF consegna il contenuto dei due registri indirizzati alle due porte in uscita (DataRead1 e DataRead2)
- I due valori vengono memorizzati in **due registri di transizione A e B**



- I registri di transizione A e B vengono scritti sempre ad ogni decode
- Il loro contenuto servirà alla successiva fase di EX, ma in modo diverso a seconda di quale istruzione la CPU sta eseguendo
- **Istruzioni R e beq:** useranno il contenuto di A e B come operandi della ALU, queste istruzioni infatti fanno operazioni con i contenuti di entrambi i registri rs e rt
- **Istruzioni di accesso a memoria (lw/sw):** useranno solo il contenuto di A (rs), il secondo operando della ALU (l'offset) verrà preso direttamente da IR
- **jump** non li usa: non necessita di una fase EX poiché si sovrascrive direttamente il PC a partire dallo pseudo-indirizzo contenuto in IR

CPU a ciclo multiplo: EX/ALU

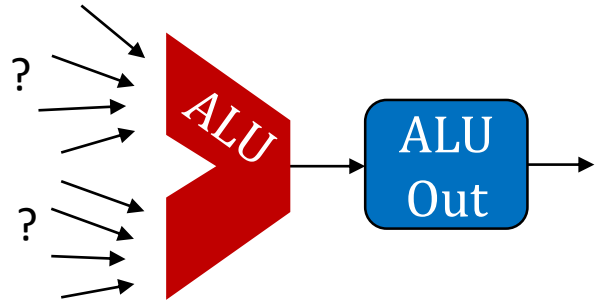
- La fase di EX avviene nel ciclo di clock successivo a ID
- La CPU assume di avere nei registri A e B il contenuto dei registri rs e rt (Attenzione! Potrebbe servirne uno solo o anche nessuno a seconda dell'istruzione)
- In questa fase la ALU fa **sempre** una operazione aritmetico/logica con due operandi, il risultato sarà memorizzato nel **registro di transizione ALUOut**, potrà essere usato in una fase successiva



CPU a ciclo multiplo: EX/ALU

Chi sono gli operandi della ALU?

- Come nella CPU a singolo ciclo cambieranno (tramite segnali di controllo e MUX posti sugli ingressi) a seconda di quale istruzione sta svolgendo la CPU (diversi datapath)
- Aspetto chiave: nella CPU multi-ciclo la ALU verrà sempre usata nella fase di EX, ma può essere usata anche in fasi diverse poiché è disponibile
- Sappiamo già che nella fase di IF la usiamo per fare PC+4, quindi in questo caso i suoi operandi saranno il **contenuto di PC** e **la costante 4**
- Durante la fase di EX di una istruzione A/L i due operandi saranno il **contenuto del registro A** e del **registro B** (il contenuto di, rispettivamente, rs e rt estratti nella fase di ID)
- Durante la fase di EX di una istruzione di accesso a memoria lw/sw i due operandi saranno **il contenuto del registro A** e **l'offset esteso di segno**, la ALU qui è usata per il calcolo dell'indirizzo $(rs) + SignExt(OFFSET)$
- Durante la jump la ALU non è usata



CPU a ciclo multiplo: anticipazione BTA

- La beq pone delle difficoltà aggiuntive! Richiede di usare la ALU **due** volte
  1. per calcolare l'indirizzo di salto $PC + \text{SignExt}(\text{OFFSET}) \times 4$
  2. per verificare la condizione $(rs) - (rt) == 0$

Come fare?

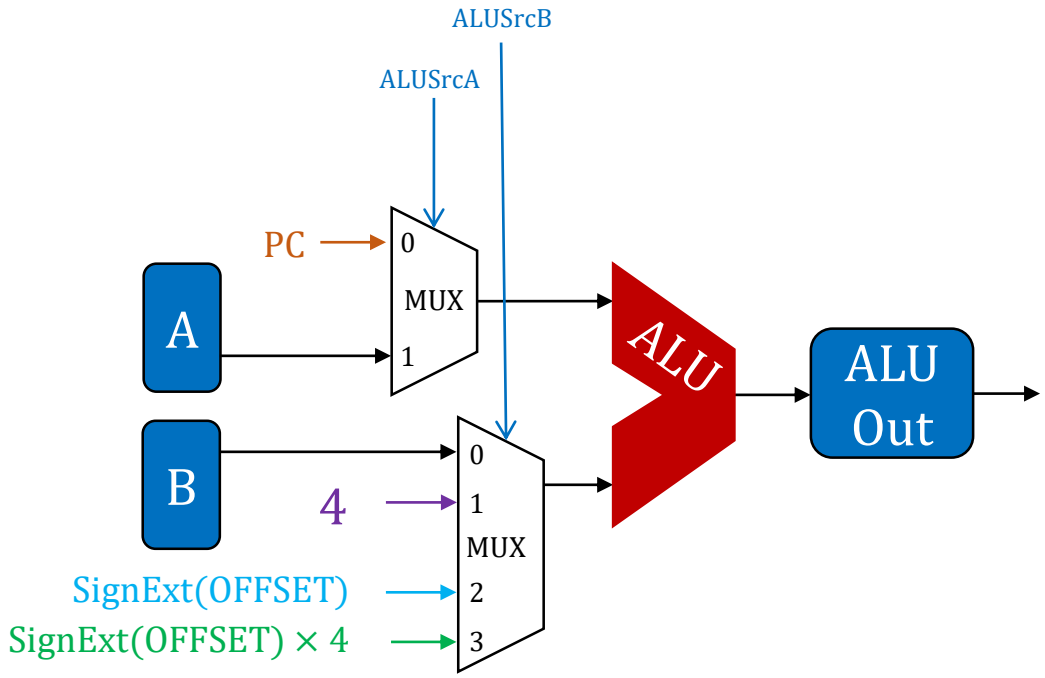
Idea:

- Nella fase di ID nessuno usa la ALU! Sfrutto la disponibilità: le faccio fare $PC + \text{SignExt}(\text{OFFSET}) \times 4$ (tutti dati già disponibili!). Il risultato è il Branch Target Address (BTA) che viene salvato in ALUOut, chiamiamo questa operazione **anticipazione del BTA**
- Quando passo alla successiva fase di EX
  - se non sto eseguendo la beq sovrascrivo ALUOut con il risultato richiesto dall'istruzione che devo eseguire (un valore se A/L o un indirizzo se lw/sw)
  - Se invece sto eseguendo una beq, faccio fare alla ALU $(rs) - (rt)$ e se il bit di zero risulta 1 allora il valore in ALUOut (precedentemente settato al Branch Target Address) verrà scritto nel PC

CPU a ciclo multiplo: EX/ALU

- Uso della ALU nella CPU a ciclo multiplo

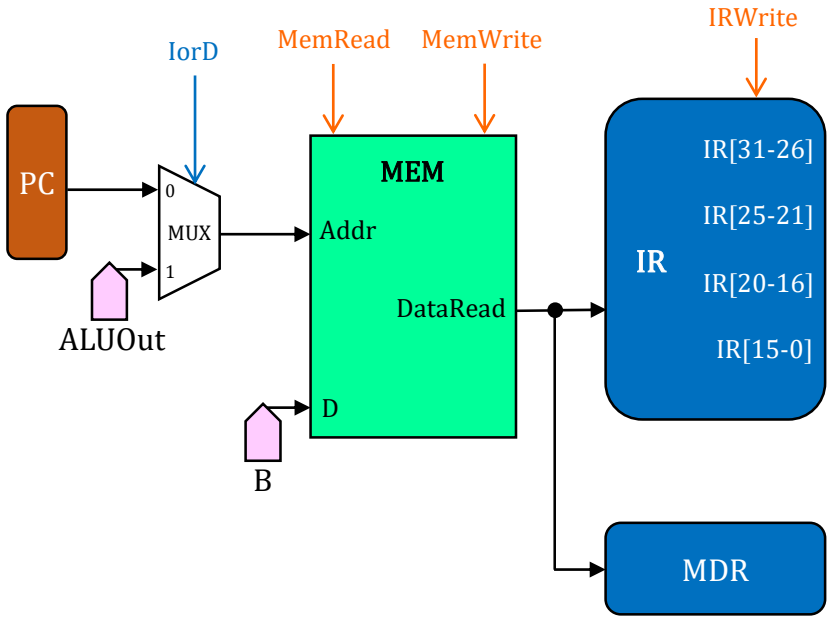
	Primo operando ALU	Secondo operando ALU
IF	PC	4
ID beq	PC	SignExt(OFFSET) × 4
EX A/L	A	B
EX lw/sw	A	SignExt(OFFSET)
EX beq	A	B



- Sarà l'unità di controllo, come al solito, a pilotare i MUX in base a quale sotto-fase e a quale istruzione si sta eseguendo
- Utilizzerà i segnali di selezione ALUSrcA e ALUSrcB

CPU a ciclo multiplo: MEM

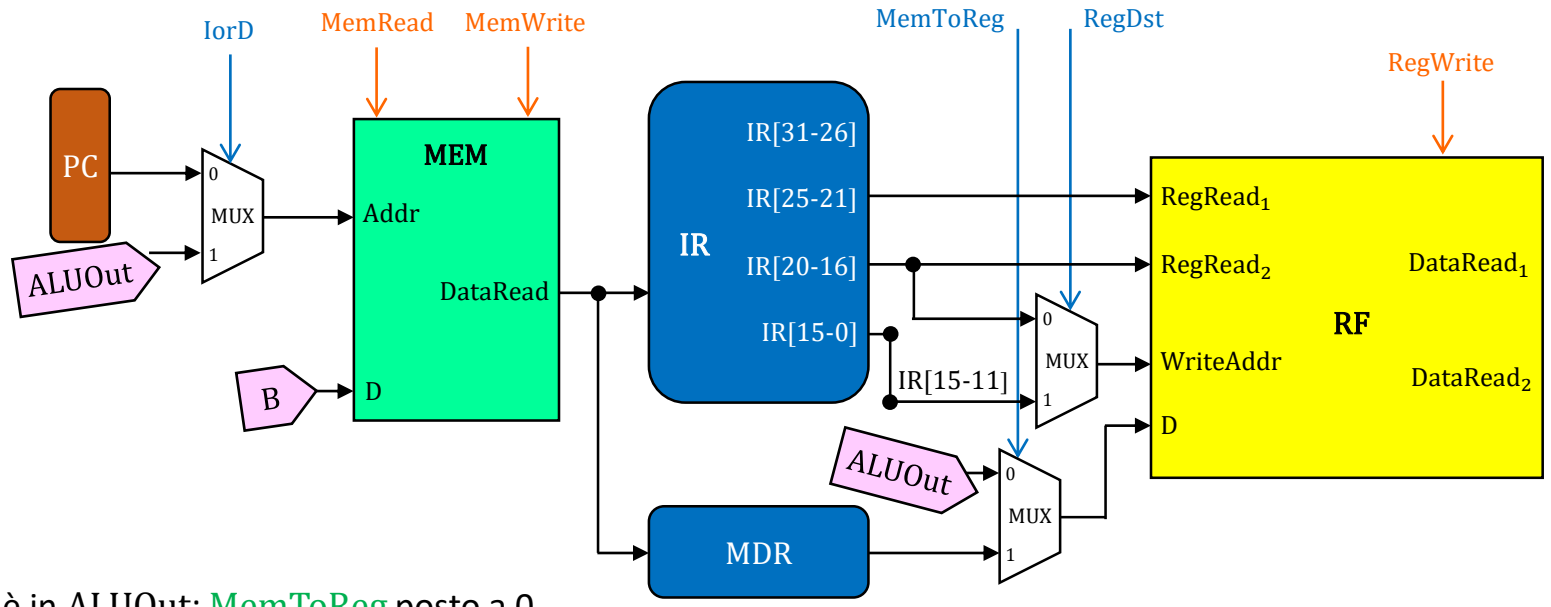
- Quando la CPU esegue una fase MEM assume che in ALUOut vi sia un indirizzo di memoria, calcolato nella precedente fase EX come $(rs) + SignExt(OFFSET)$
- ALUOut, deve quindi poter essere presentato alla porta Addr del modulo di memoria



- Il MUX all'ingresso della memoria permette di selezionare l'indirizzo
  - Instruction: indirizzo di una istruzione, nel segmento testo (sarà in sola lettura)
  - Data: indirizzo di una word nel segmento dati (scrittura o lettura)
- Segnale di selezione IorD (Instruction or Data)
- **lw**: accesso in lettura (MemRead a 1) con l'indirizzo da ALUOut (IorD a 1), il dato viene trasferito al **registro di transizione Memory Data Register (MDR)**
- **sw**: accesso in scrittura (MemWrite a 1) con l'indirizzo da ALUOut (IorD a 1), il dato da scrivere si trova nel **registro di transizione B** $((rt))$

CPU a ciclo multiplo: WB

- Questa fase riguarda soltanto le istruzioni di tipo A/L e lw



Istruzioni A/L:

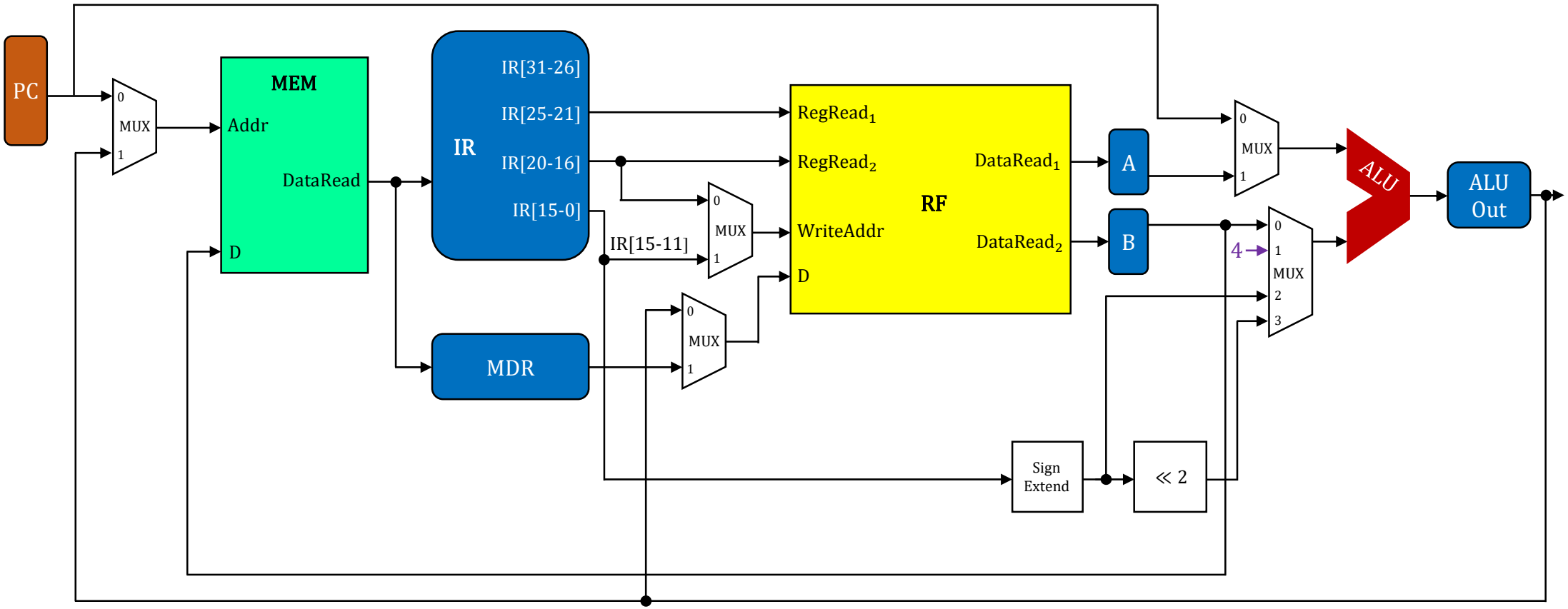
- il dato da scrivere è in ALUOut: MemToReg posto a 0
- l'indirizzo del registro in cui scriverlo è in IR (IR[15-11], ovvero rd): RegDst posto a 1

Load word:

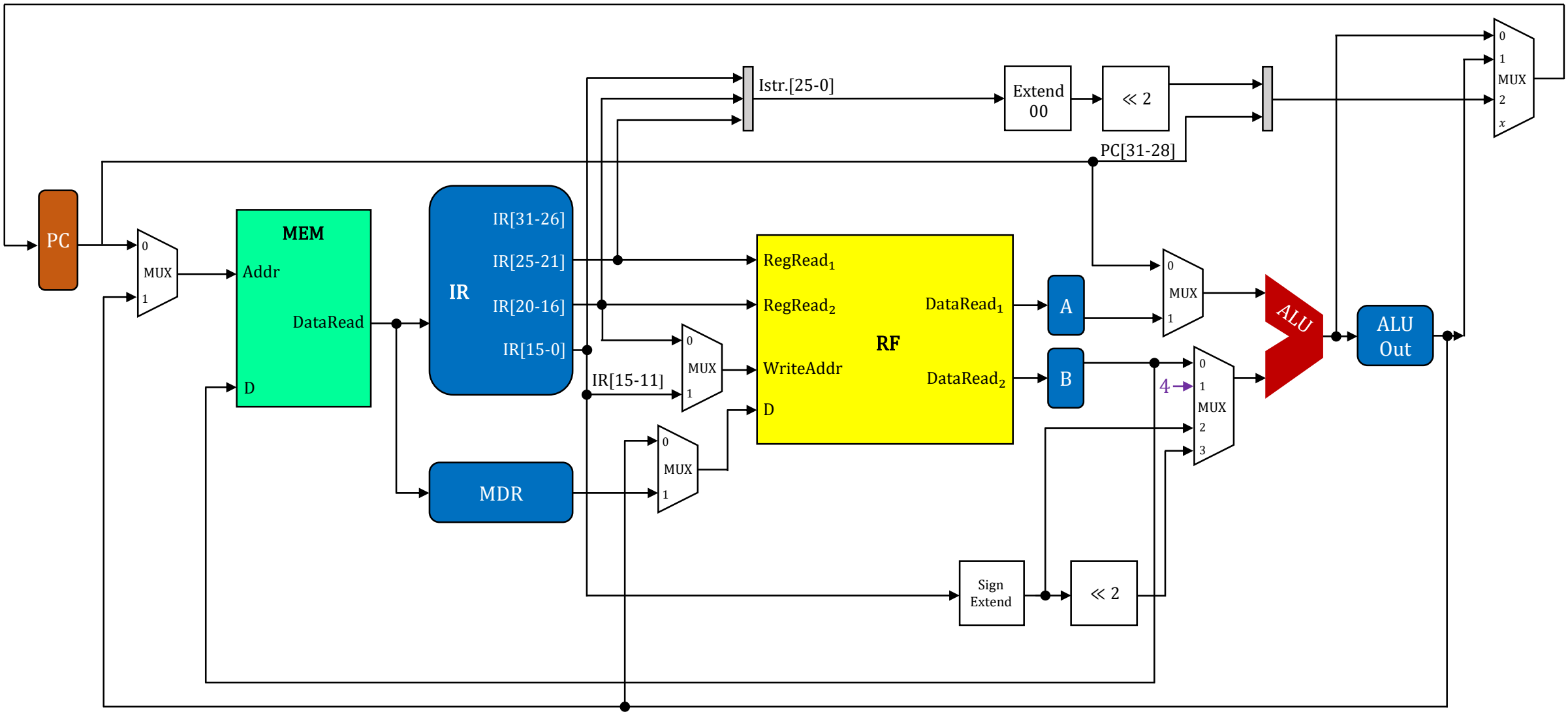
- il dato da scrivere è in MDR: MemToReg posto a 1
- l'indirizzo del registro in cui scriverlo è in IR (IR[20-16], ovvero rt): RegDst posto a 0

CPU a ciclo multiplo

- Primo step: colleghiamo tutti i moduli
- Secondo step: aggiungiamo la parte per la gestione degli indirizzi di salto (branch e jump)



CPU a ciclo multiplo



CPU a ciclo multiplo: segnali di controllo

- Segnali di selezione

ALUSrcA	Selezione del primo operando ALU: PC (0) registro A (1)
ALUSrcB	Selezione del secondo operando ALU: Registro B (00), costante 4 (01), SignExt($OFFSET$) (10), SignExt($OFFSET$) $\times$ 4 (11)
IorD	Selezione per il campo Addr nel modulo memoria: indirizzo istruzione (0), indirizzo dato (1)
PCSrc	Selezione di quale indirizzo mandare al PC: risultato della ALU (00), contenuto di ALUOut (01), indirizzo della jump (10), (11 non usato)
RegDest	Selezione per il campo WriteAddr del RF: rt (0) rd (1)
MemToReg	Selezione del dato presentato in scrittura al RF: contenuto di ALUOut (0), contenuto di MDR (1)
ALUCtrl	Selezione della modalità di operazione della ALU (analogo a CPU singolo ciclo)

Ricorda:

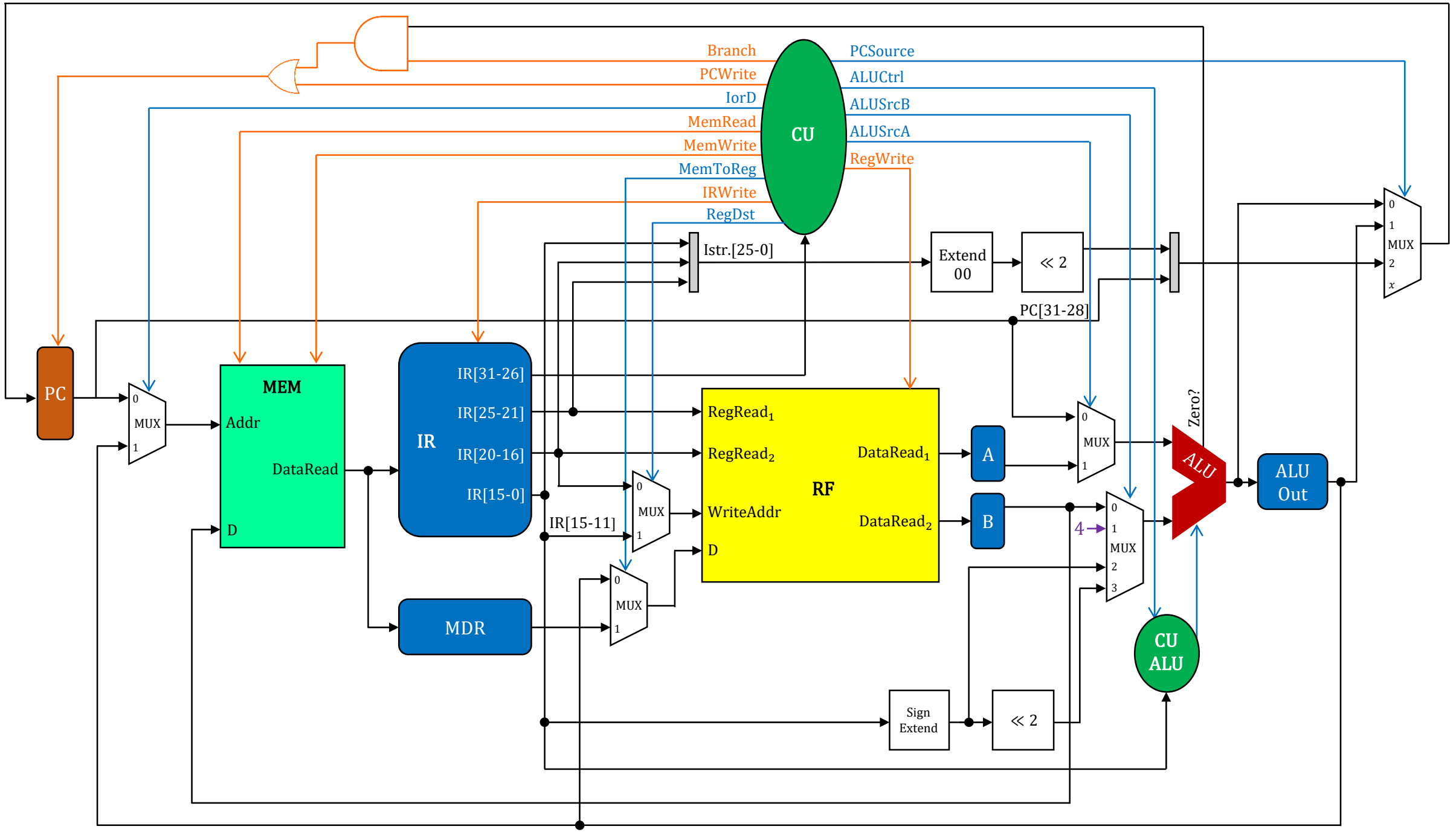
Opcode	AluCtrl
Tipo R, rimanda a funct	10
sw, somma	00
lw, somma	00
beq, sottrazione	01

CPU a ciclo multiplo: segnali di controllo

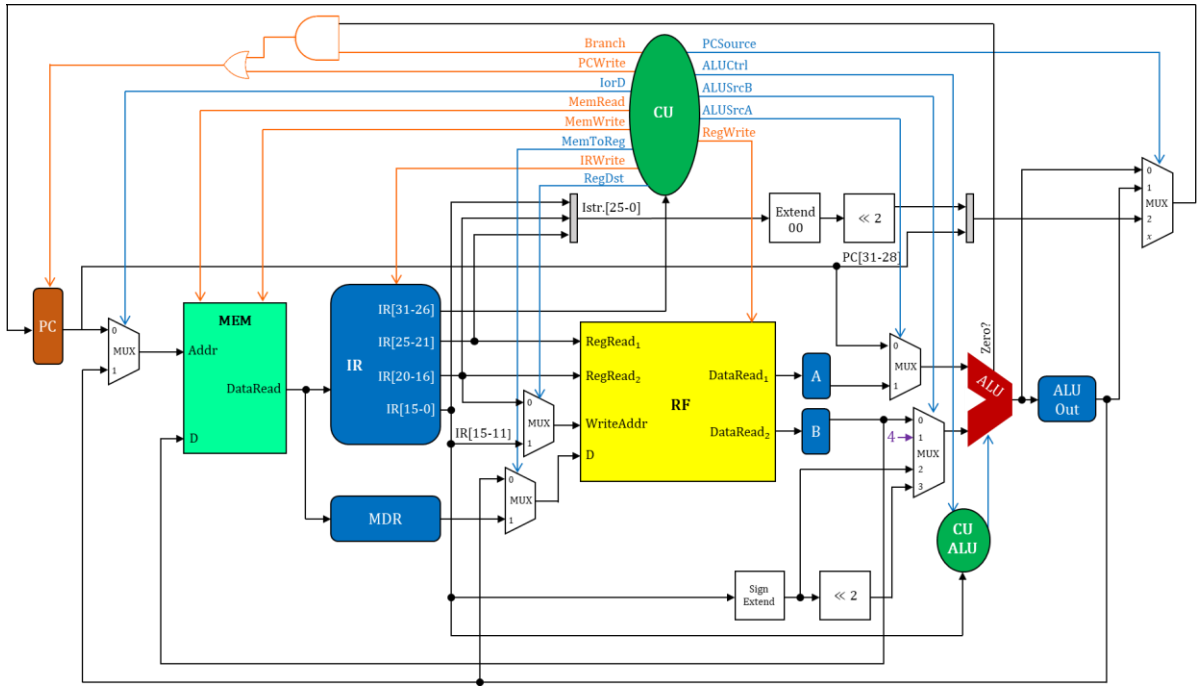
- Segnali di comando

PCWrite	Scrittura del Program Counter
Branch	Se posto a 1 abilita la scrittura del PC quando il bit di zero della ALU vale 1, da usare per i salti condizionati
IRWrite	Scrittura dell'Instruction Register
RegWrite	Scrittura del Register File
MemWrite	Scrittura della memoria
MemRead	Lettura della memoria

- Ricorda: nel Register File si può leggere e scrivere nello stesso ciclo di clock, ma non nella memoria!
- La scrittura in tutti gli altri registri non menzionati è data dal clock, vengono scritti ad ogni ciclo di clock (fronte di discesa)


Branch
PCWrite
IorD
MemRead
MemWrite
MemToReg
IRWrite
RegDst
CU
PCSource
ALUCtrl
ALUSrcB
ALUSrcA
RegWrite
Istr.[25-0]
Extend 00
≪ 2
PC[31-28]
PC
MUX
0
1
MEM
Addr
DataRead
D
IR
IR[31-26]
IR[25-21]
IR[20-16]
IR[15-0]
IR[15-11]
MDR
RF
RegRead1
RegRead2
WriteAddr
D
DataRead1
DataRead2
A
B
4
Zero?
ALU
ALU Out
MUX
2
3
x
Sign Extend
CU ALU

Unità di controllo CPU (CU)



- I segnali di controllo rappresentano le «leve» con cui possiamo far svolgere alla CPU una delle sotto-fasi di esecuzione per ciascuna istruzione
- L'unità di controllo ha il compito di sincronizzare queste sotto-fasi per ogni istruzione

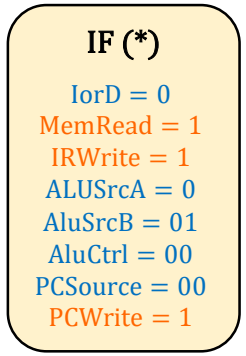
- Questo tipo di controllo non può essere svolto con una rete combinatoria: serve una memoria
- La CU deve ricordarsi che al ciclo di clock precedente ha eseguito, per esempio, la fase EX per una istruzione A/L per poter svolgere, nel ciclo corrente, la fase di WB per quella istruzione
- Il controllo dell'esecuzione necessita di uno stato: la CU è una macchina a stati finiti

Unità di controllo CPU (CU)

Come è progettata la FSM della CU nella CPU a ciclo multiplo?

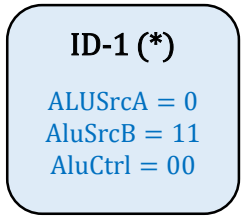
- Stato: SOTTO-FASE + TIPO-ISTRUZIONE-CORRENTE
- Input: il tipo di istruzione (**opcode**)
- Uscita: valore dei segnali di controllo (indicheremo solo quelli necessari, quelli non riportati sono o ovvi dal contesto o ininfluenti)
- Implementata come una **macchina di Moore**:
  - l'uscita dipende solo dallo stato corrente
  - Lo stato prossimo dipende dallo stato corrente e dall'input
- Vediamo quali stati vanno codificati e come comporli nel grafo delle transizioni
- La sintesi dell'unità viene lasciata come esercizio da svolgere autonomamente

Unità di controllo CPU (CU)



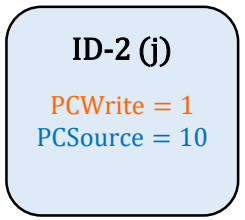
Instruction Fetch: stesso stato per ogni istruzione (*), è anche lo stato iniziale

- I segnali di controllo sono settati in modo che:
  - L'istruzione indirizzata da PC venga letta dalla memoria e scritta in IR
  - La ALU svolga $PC + 4$ (viene scritto in ALUOut, ma non serve)
  - Il risultato della ALU viene scritto nel PC



Instruction Decode 1: stesso stato per ogni istruzione (*)

- Grazie alla precedente fase IF, IR contiene i 32 bit dell'istruzione i cui campi vanno in ingresso al RF
- Il RF legge il contenuto dei registri e li carica in A e B (scritti ad ogni ciclo di clock)
- I segnali di controllo sono settati in modo che venga svolta l'**anticipazione del BTA**:
  - La ALU svolge $PC + \mathrm{SignExt}(\mathrm{OFFSET}) \times 4$
  - Il risultato della ALU viene scritto in ALUOut



Instruction Decode 2 per j: dopo ID-1, solo per la jump

- I segnali di controllo sono settati in modo che venga scritto nel PC l'indirizzo di salto

Unità di controllo CPU (CU)

EX (beq)

ALUSrcA = 1
AluSrcB = 00
AluCtrl = 01
PCSource = 01
Branch = 1

Execute di beq

- Grazie alla precedente fase ID-1, in ALUOut c'è il BTA
- I segnali di controllo sono settati in modo che:
  - La ALU svolga $(rs) - (rt)$ (verrà scritto in ALUOut, ma sul fronte di discesa)
  - Se il bit di zero vale 1, il contenuto di ALUOut viene scritto nel PC

EX (A/L)

ALUSrcA = 1
AluSrcB = 00
AluCtrl = 10

Execute di una qualsiasi istruzione A/L

- Grazie alla precedente fase ID-1, in A e B ci sono i contenuti di rs e rt
- I segnali di controllo sono settati in modo che:
  - La ALU svolga $(rs)\ op\ (rt)$
  - Il risultato della ALU venga scritto in ALUOut

EX (lw/sw)

ALUSrcA = 1
AluSrcB = 10
AluCtrl = 00

Execute di lw o sw

- Grazie alla precedente fase ID-1, in A c'è il contenuto di rs (base address)
- I segnali di controllo sono settati in modo che:
  - La ALU svolga $(rs) + \mathrm{SignExt}(\mathrm{OFFSET})$ (calcolo dell'indirizzo)
  - Il risultato della ALU venga scritto in ALUOut

Unità di controllo CPU (CU)

MEM (sw)

IorD = 1
MemWrite = 1

Memory di sw

- Grazie alla precedente fase ID-1, in B c'è il contenuto di rt (la parola da scrivere in memoria). B è collegato direttamente con l'ingresso dati del modulo di memoria
- Grazie alla precedente fase di EX(lw/sw) ALUOut contiene l'indirizzo della parola di memoria in cui scrivere
- I segnali di controllo sono settati in modo che rt venga scritto in memoria all'indirizzo contenuto in ALUOut

MEM (lw)

IorD = 1
MemRead = 1

Memory di lw

- Grazie alla precedente fase di EX(lw/sw) ALUOut contiene l'indirizzo della parola di memoria da cui leggere
- I segnali di controllo sono settati in modo che la parola in memoria all'indirizzo contenuto in ALUOut venga letta dalla memoria e scritta in MDR

Unità di controllo CPU (CU)

WB (A/L)

RegDst = 1
MemToReg = 0
RegWrite = 1

Writeback di una qualsiasi istruzione A/L

- Grazie alla precedente fase di EX(A/L) ALUOut contiene il risultato dell'operazione
- I segnali di controllo sono settati in modo che il contenuto in ALUOut venga scritto nel registro rd

WB (lw)

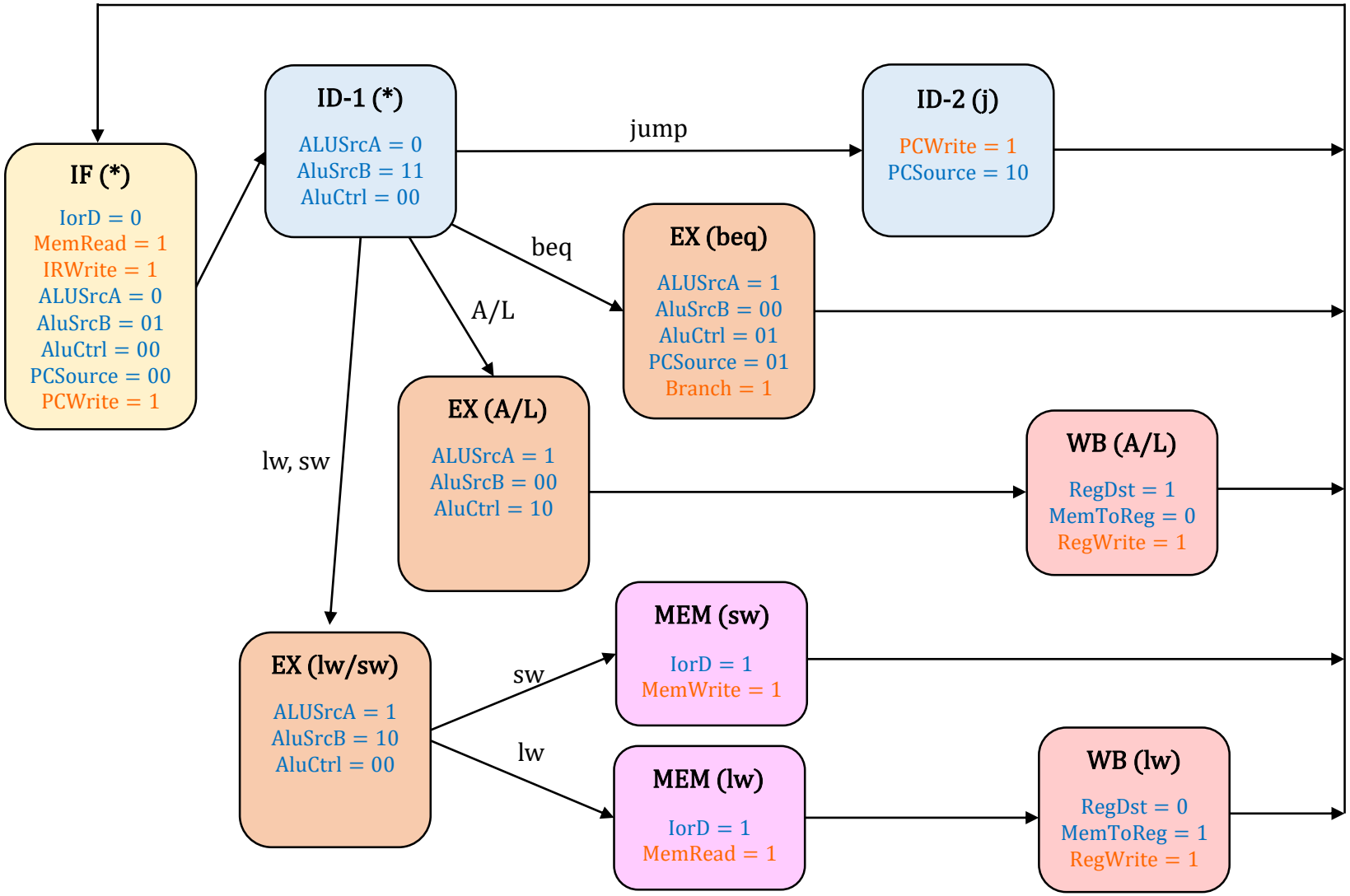
RegDst = 0
MemToReg = 1
RegWrite = 1

Writeback di lw

- Grazie alla precedente fase di MEM(lw) MDR contiene il dato letto dalla memoria
- I segnali di controllo sono settati in modo che il contenuto di MDR venga scritto nel registro rt

Unità di controllo CPU (CU)

Grafo delle transizioni della FSM che implementa la Control Unit



Architettura degli Elaboratori II

Corso di Laurea Triennale in Informatica

Università degli Studi di Milano

Dipartimento di Informatica “Giovanni Degli Antoni”

Edizione 2 (Cognomi H-Z), A.A. 2021-2022, Nicola.Basilico@unimi.it

CPU a pipeline

La pipeline

- Il concetto di **pipeline**: la catena di montaggio
- **Problema**: gli invitati ad un ricevimento sono in fila per l'antipasto, seguono un percorso fatto da una sequenza di isole gastronomiche dove gli viene servito un assaggio
- Il servizio su ogni isola richiede 1 unità di tempo, non ci sono stalli (blocco della coda) e trascuriamo il tempo degli spostamenti

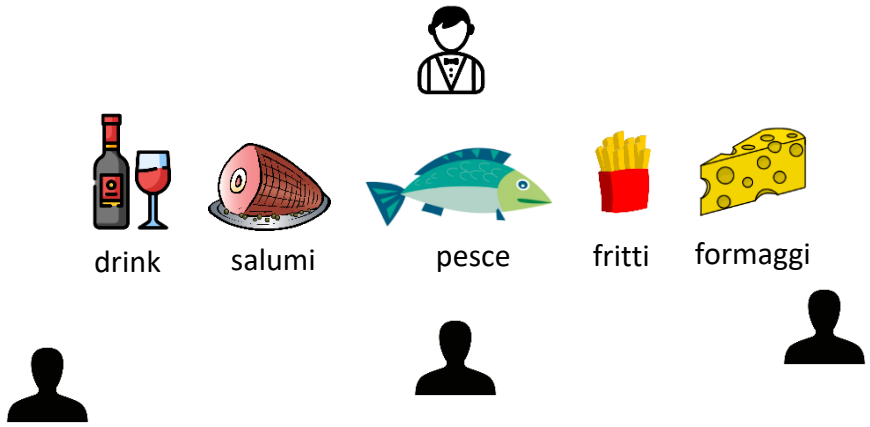
- Dalla coda esce un invitato ogni 5 unità di tempo
- **Frequenza 0.2**

- Dalla coda esce un invitato ogni 1 unità di tempo
- **Frequenza 1 (5 volte più veloce!)**

La pipeline

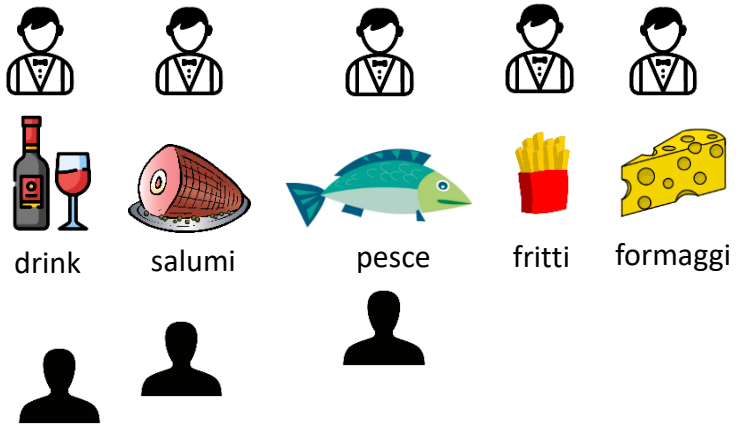
- Il concetto di **pipeline**: la catena di montaggio
- **Problema**: gli invitati ad un ricevimento sono in fila per l'antipasto, seguono un percorso fatto da una sequenza di isole gastronomiche dove gli viene servito un assaggio
- Il servizio su ogni isola richiede 1 unità di tempo, non ci sono stalli (blocco della coda) e trascuriamo il tempo degli spostamenti

Primo scenario: c'è un solo cameriere



- Dalla coda esce un invitato ogni 5 unità di tempo
- **Frequenza 0.2**

Secondo scenario, approccio pipeline: un cameriere per ogni isola

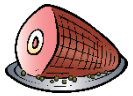


- Dalla coda esce un invitato ogni 1 unità di tempo
- **Frequenza 1 (5 volte più veloce!)**

La pipeline

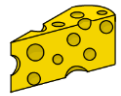
Istruzione da eseguire


drink


salumi

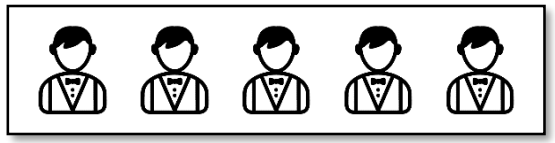

pesce


fritti


formaggi

Fasi dell'esecuzione (IF, ID, EX, MEM, WB)

CPU a singolo ciclo (ma anche a **ciclo multiplo**): le fasi lavorano in sequenza e in un dato ciclo di clock la CPU sta eseguendo una fase di una singola istruzione



CPU a pipeline: le fasi lavorano in **parallelo**! In un dato ciclo di clock ci possono essere fino a 5 istruzioni contemporaneamente in esecuzione, ma in fasi diverse

- Consideriamo un programma fatto da N istruzioni, 5 fasi di esecuzione ciascuna di durata t_j con $j \in \{1,2,3,4,5\}$ e la seguente funzione:

$$f(i,j) = \begin{cases} 1, & \text{l'istruzione } i \text{ necessita della fase } j \\ 0, & \text{altrimenti} \end{cases}$$

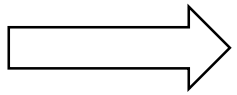
- La durata del ciclo di clock T_{ck} è costante, quindi dobbiamo assumere che tutte le fasi durino in realtà $t = \max_j\{t_j\}$
- Calcoliamo T definito come il tempo totale impiegato per eseguire il programma
  - Nella CPU a singolo ciclo $T_S = N5t$ e abbiamo il vincolo che $T_{ck} \geq 5t$
  - Nella CPU a ciclo multiplo $T_M = \sum_{i=1}^{N}\sum_{j=1}^{5} f(i,j)t$ e abbiamo il vincolo che $T_{ck} \geq t$
  - Nella CPU a pipeline $T_P = 5t + t(N-1)$ e abbiamo ancora il vincolo che $T_{ck} \geq t$

La pipeline

- Confronto dei tempi totali: $T_P \leq T_M \leq T_S$

Throughput (tasso di produzione), definizione: $\rho = \lim_{Lavoro \to \infty} \frac{Lavoro}{Tempo\ speso}$

- Nella CPU singolo ciclo: $\rho_S = \lim_{N \to \infty} \frac{N}{N5t} = \frac{1}{5t}$
- Nella CPU a pipeline: $\rho_P = \lim_{N \to \infty} \frac{N}{5t + t(N-1)} = \frac{1}{t}$



$\rho_P = 5\rho_S$, la CPU a pipeline è 5 volte più veloce di una CPU a singolo ciclo

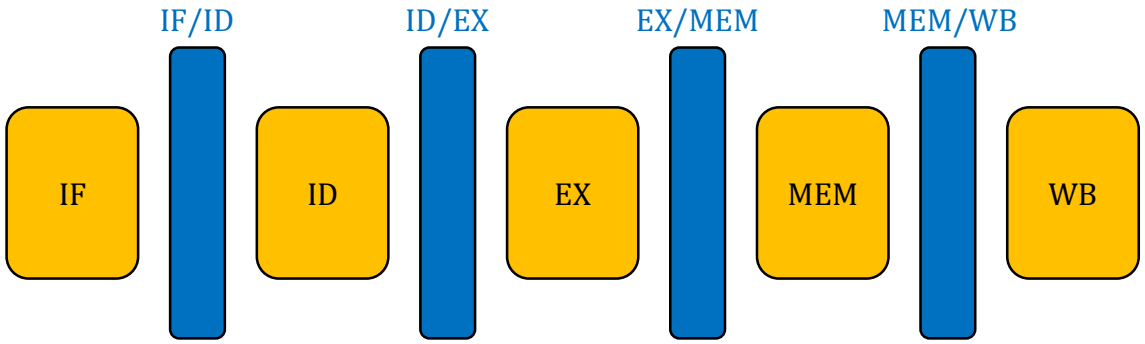
Il risultato si generalizza rispetto al numero di stadi:

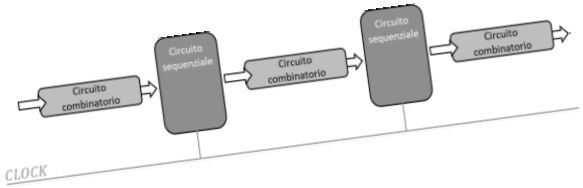
a parità di durata delle fasi, una CPU a pipeline con k stadi è k volte più veloce di una CPU a singolo ciclo

- E la CPU a ciclo multiplo?
  - **Worst case**: tutte le istruzioni richiedono 5 fasi: $\rho_M = \frac{1}{5t}$ (come CPU a singolo ciclo)
  - **Best case**: tutte le istruzioni richiedono 2 fasi: $\rho_M = \frac{1}{2t}$
- In ogni caso più lenta della CPU a pipeline, nonostante con la pipeline richiediamo ad ogni istruzione di attraversare tutte le 5 fasi, anche quando non serve!

CPU pipeline

- Come dobbiamo progettare il datapath in modo da implementare un'esecuzione a pipeline?
- **Concetto chiave della pipeline**: le fasi lavorano in **parallelo** ma devono essere applicate a ciascuna istruzione in **sequenza**!
- Perché possano lavorare in parallelo serve che ogni fase venga eseguita in una **regione** (sotto-circuito) dedicata del datapath
- Perché possano essere applicate in sequenza serve un modo per rendere accessibile il risultato parziale di una fase alla fase successiva: **registri di transizione**

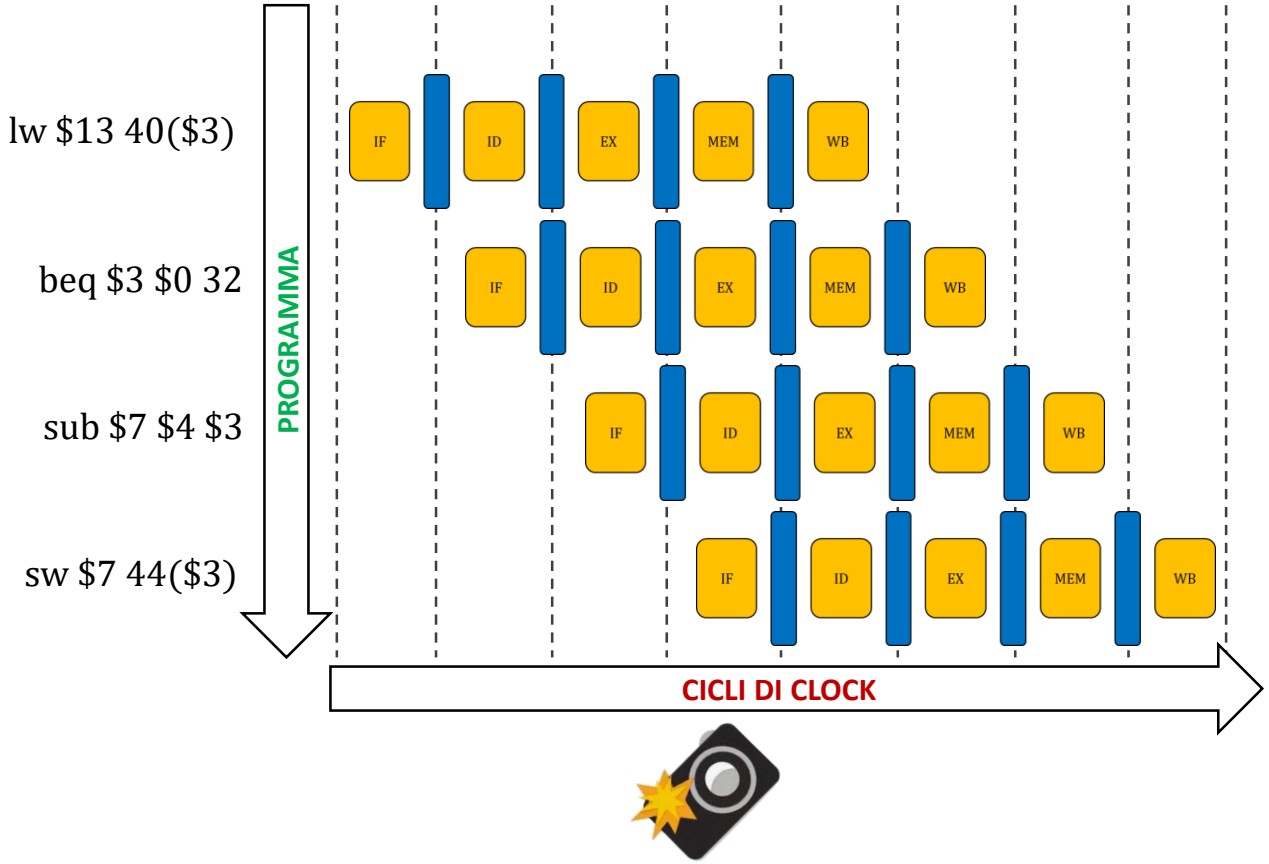




- I registri di transizione servono per trasferire nella fase $i + 1$ il risultato dell'elaborazione svolta nella fase i

CPU pipeline

- Rappresentiamo su un piano cartesiano l'esecuzione di un programma
- Per ogni ciclo di clock (asse x) vediamo in quale fase si trova ciascuna istruzione (asse y) correntemente in esecuzione nella CPU



- Facciamo uno snapshot della CPU al quarto ciclo di clock

lw	MEM
beq	EX
sub	ID
sw	IF

- Tutte e quattro le istruzioni sono in esecuzione, ciascuna in una fase diversa

- Se manteniamo una sola ALU e una sola unità di Memoria, istruzioni che sono contemporaneamente in esecuzione possono andare in **conflitto sull'uso di una risorsa**

Esempi:

- sw e sub vanno in conflitto sulla ALU ($PC + 4$ vs anticipazione BTA)
- sw e lw vanno in conflitto sulla Memoria (lettura istruzione vs lettura dato, ricordiamo che abbiamo un'unica unità di memoria)

Criticità strutturali

Criticità strutturali

- Le **criticità strutturali** (**structural hazards**) sono dei conflitti sulle risorse della CPU a pipeline, dovute al fatto che più di una istruzione può trovarsi in esecuzione nello stesso ciclo di clock
- Partiamo dal datapath della CPU a ciclo multiplo e analizziamo questi potenziali conflitti

		ALU	MEM	RF
IF		✔	✔	
ID		✔		✔ R
EX	A/L	✔		
	lw	✔		
	sw	✔		
	beq	✔		
	j			
MEM	lw		✔	
	sw		✔	
WB	A/L			✔ W
	lw			✔ W

- ALU e MEM possono svolgere una sola operazione in un ciclo di clock: se sulla colonna di ALU o di MEM compaiono ≥ 2 ✔ su sfondi di colore diverso (fasi diverse) possiamo avere criticità strutturali su ALU e/o MEM
- RF può svolgere un massimo di due operazioni in un ciclo di clock, ma devono essere in modalità diversa (read e write): ogni volta che su sfondi di colore diverso compaiono > 2 ✔ oppure 2 ✔ con la stessa modalità c'è una potenziale criticità strutturale su RF
- Sia su ALU che MEM abbiamo potenziali criticità, su RF nessuna!

Criticità strutturali

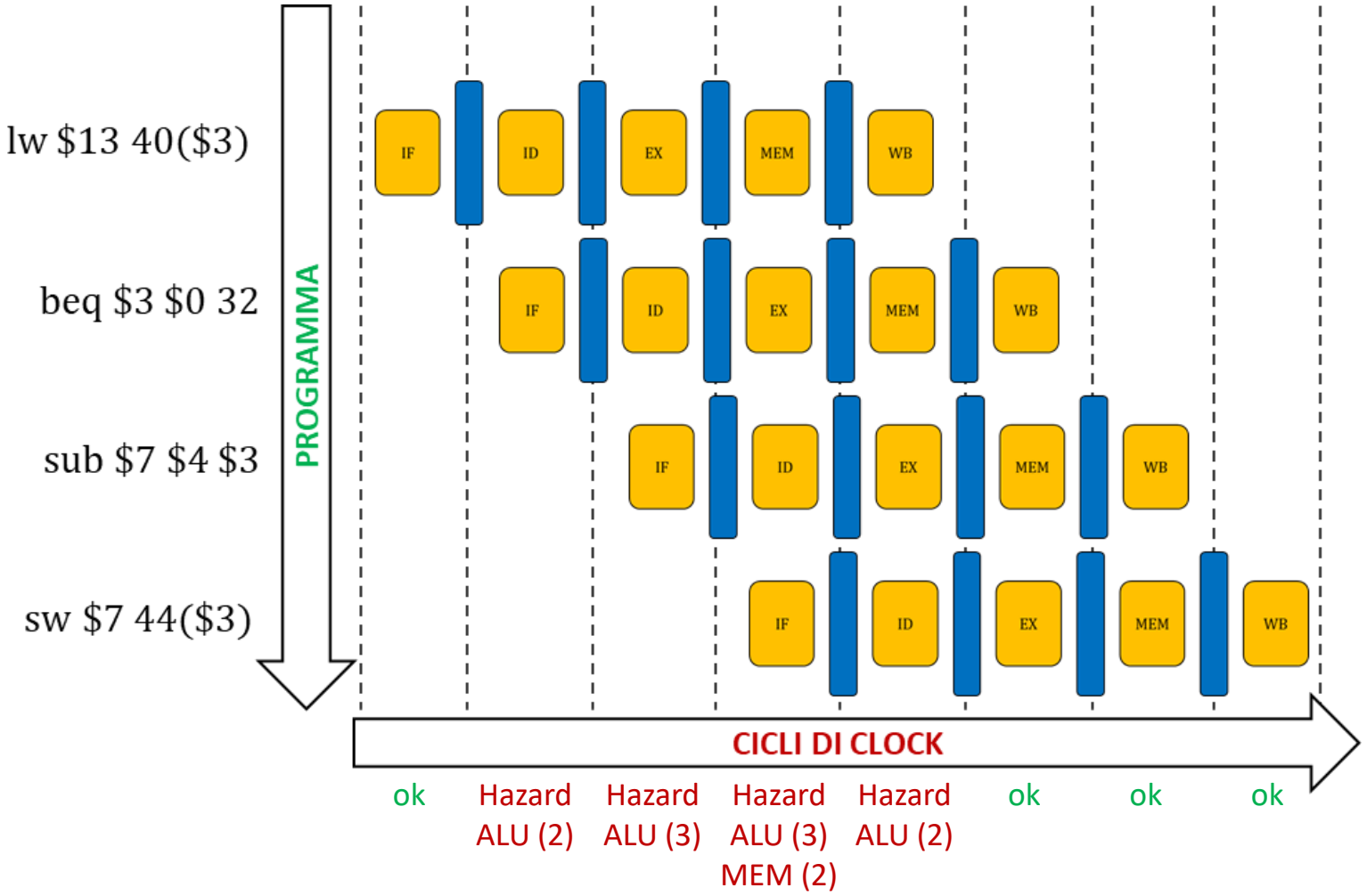
- Come si risolve una criticità strutturale?
- Dobbiamo modificare il datapath e tornare alla duplicazione dei componenti!

		ALU	MEM	RF
IF		✔	✔	
ID		✔		✔ R
EX	A/L	✔		
	lw	✔		
	sw	✔		
	beq	✔		
	j			
MEM	lw		✔	
	sw		✔	
WB	A/L			✔ W
	lw			✔ W

- Quanti RF servono?
  - Uno solo, il RF non presenta criticità
- Quante ALU servono?
  - Su quanti sfondi diversi abbiamo la ✔ nella colonna ALU? Tre! Servono tre ALU
- Quante unità di memoria servono?
  - Su quanti sfondi diversi abbiamo la ✔ nella colonna MEM? Due! Servono due MEM

Criticità strutturali

- Tutte le criticità strutturali dall'esempio precedente

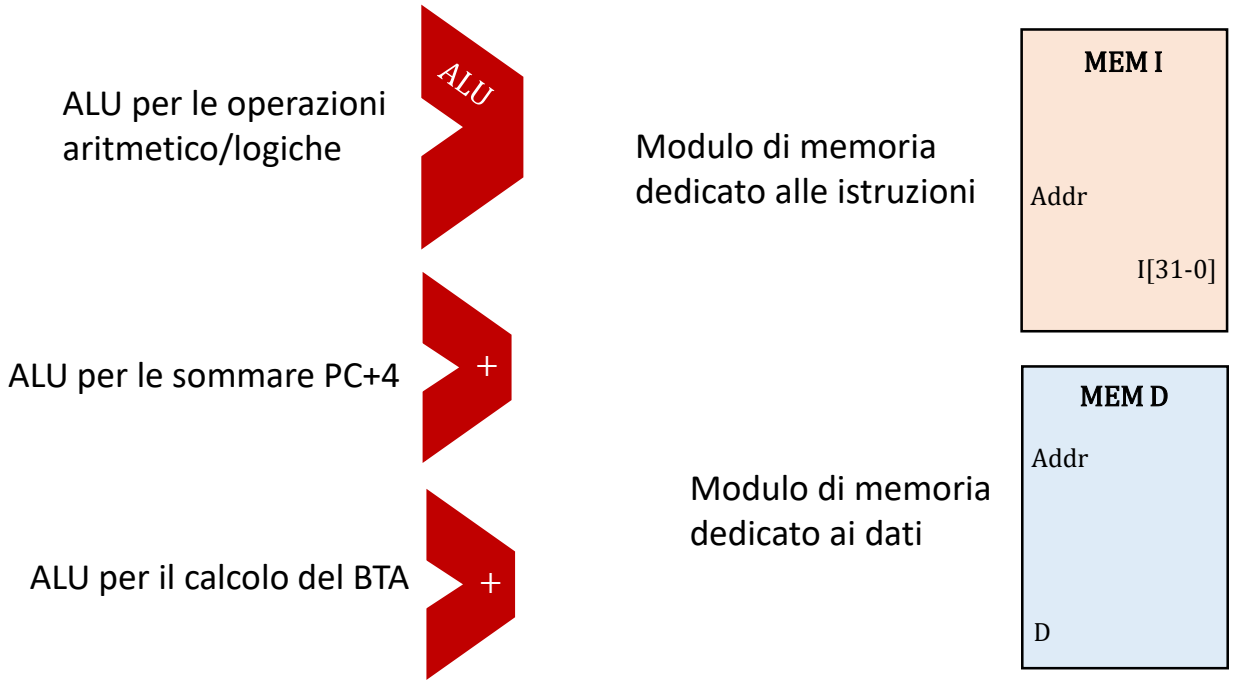


		ALU	MEM	RF
IF		✔	✔	
ID		✔		✔ R
EX	A/L	✔		
	lw	✔		
	sw	✔		
	beq	✔		
	j			
MEM	lw		✔	
	sw		✔	
WB	A/L			✔ W
	lw			✔ W

Criticità strutturali

- Duplichiamo le risorse ritornando alla configurazione che avevamo nella CPU a singolo ciclo

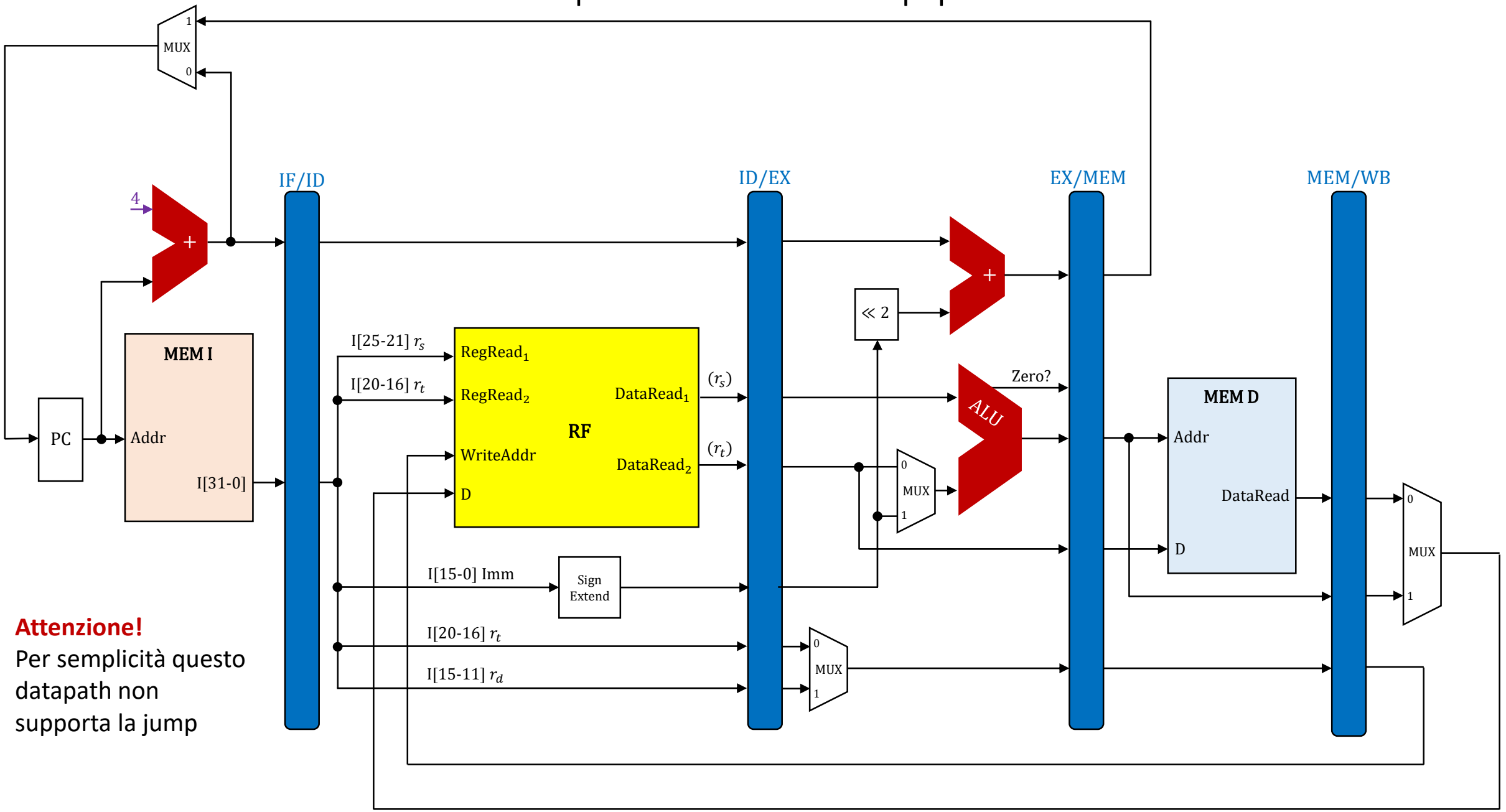
		ALU	MEM	RF
IF		✔	✔	
ID		✔		✔ R
EX	A/L	✔		
	lw	✔		
	sw	✔		
	beq	✔		
	j			
MEM	lw		✔	
	sw		✔	
WB	A/L			✔ W
	lw			✔ W



- **Importante**: non abbiamo più necessità di anticipate il calcolo del BTA per sfruttare la disponibilità della ALU, ora ne abbiamo una dedicata! Possiamo tranquillamente farlo nella fase di EX (dove è più giusto che sia) in parallelo alla verifica della condizione

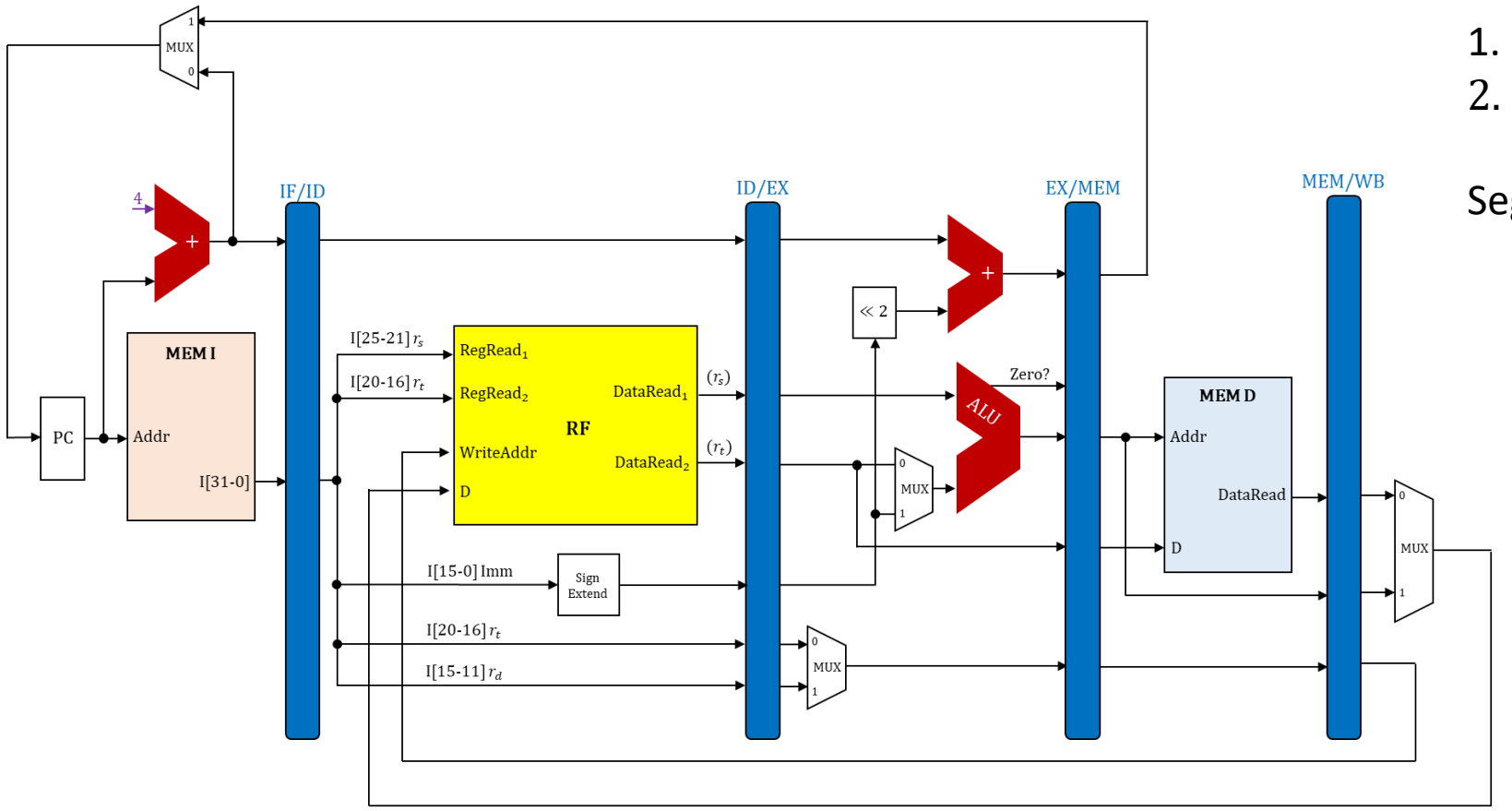
Datapath della CPU a pipeline

Datapath della a CPU pipeline



Attenzione!
Per semplicità questo datapath non supporta la jump

Esempio di esecuzione



Esempio

Consideriamo un semplice programma fatto da sole due istruzioni:

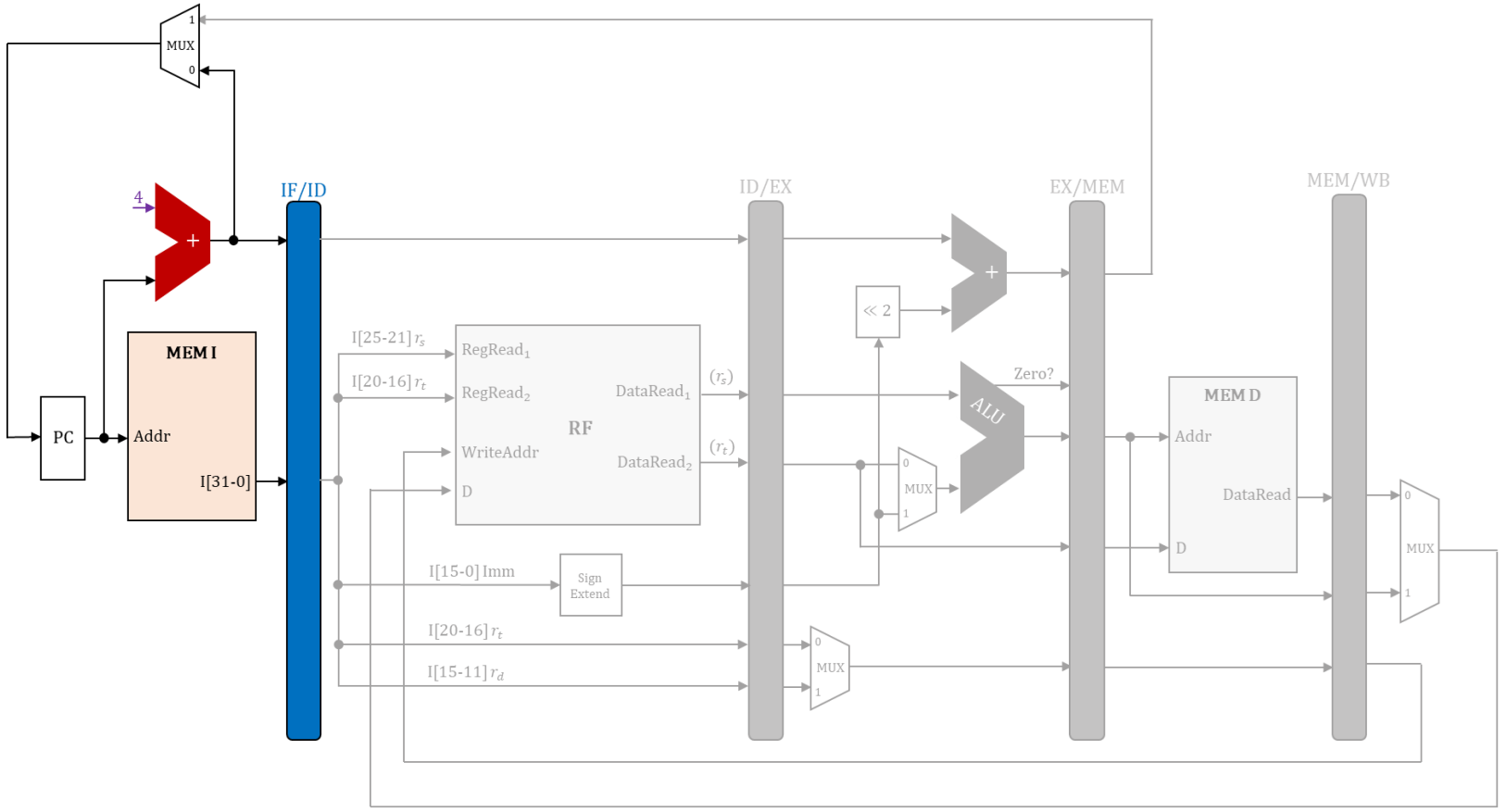
1. lw \$13 40(\$3)
2. beq \$3 \$0 32

Seguiamo l'esecuzione un passo alla volta

Attenzione!
Per ora assumiamo che le istruzioni siano indipendenti l'una dall'altra anche se, come vedremo più avanti, non è quasi mai vero

Esempio di esecuzione (ciclo di clock 1)

- Il PC contiene l'indirizzo dell'istruzione lw $13 40($3)
- L'istruzione successiva è beq $3 $0 32

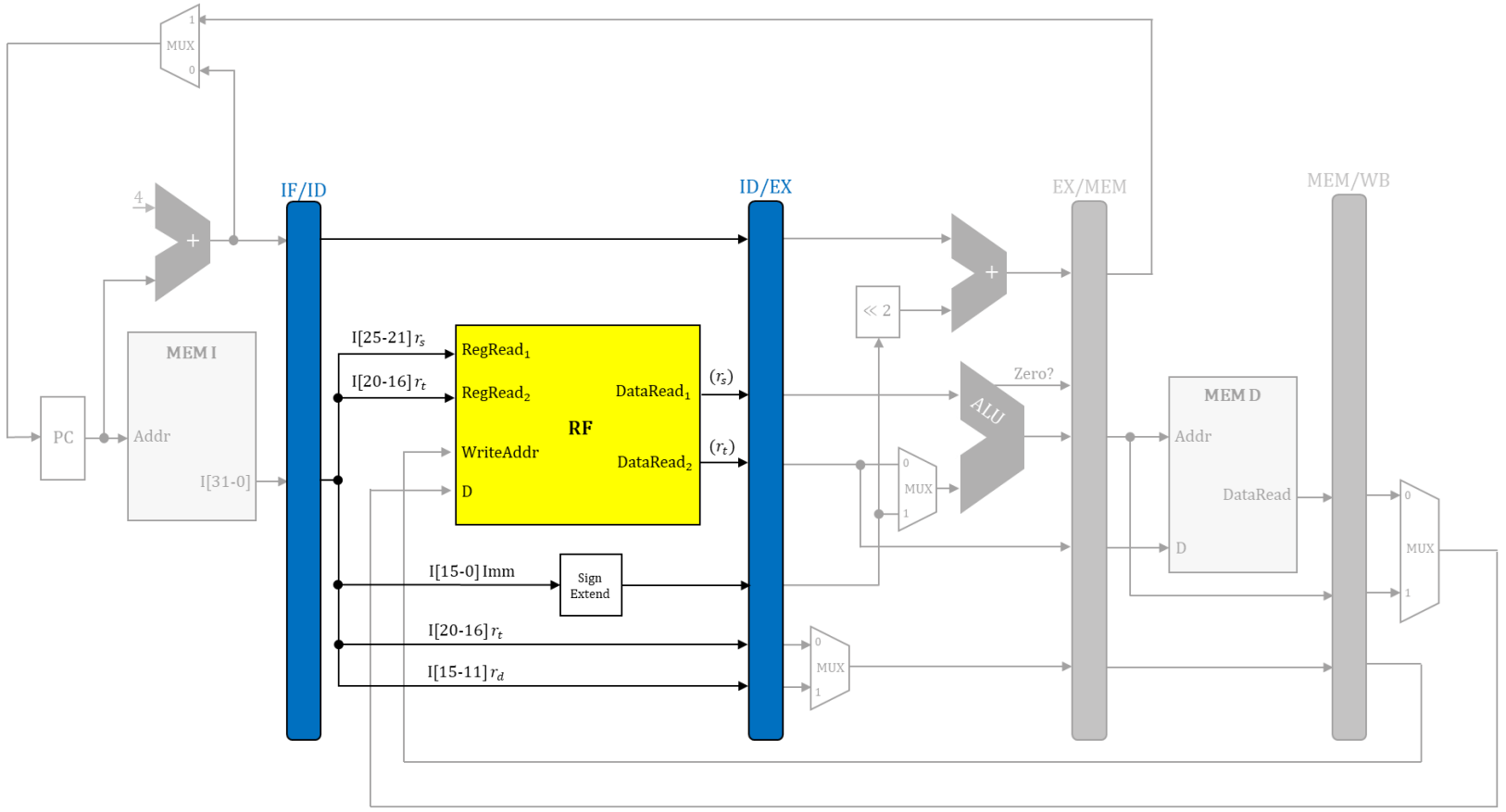


Ciclo di clock 1

- La lw viene prelevata da MEM I e salvata nel registro di transizione IF/ID (campo I)
- Il PC viene sovrascritto con PC+4 (l'indirizzo della beq)

Esempio di esecuzione (ciclo di clock 2)

- Il PC contiene l'indirizzo dell'istruzione beq $3 $0 32
- **lw $13 40($3) in fase di ID**

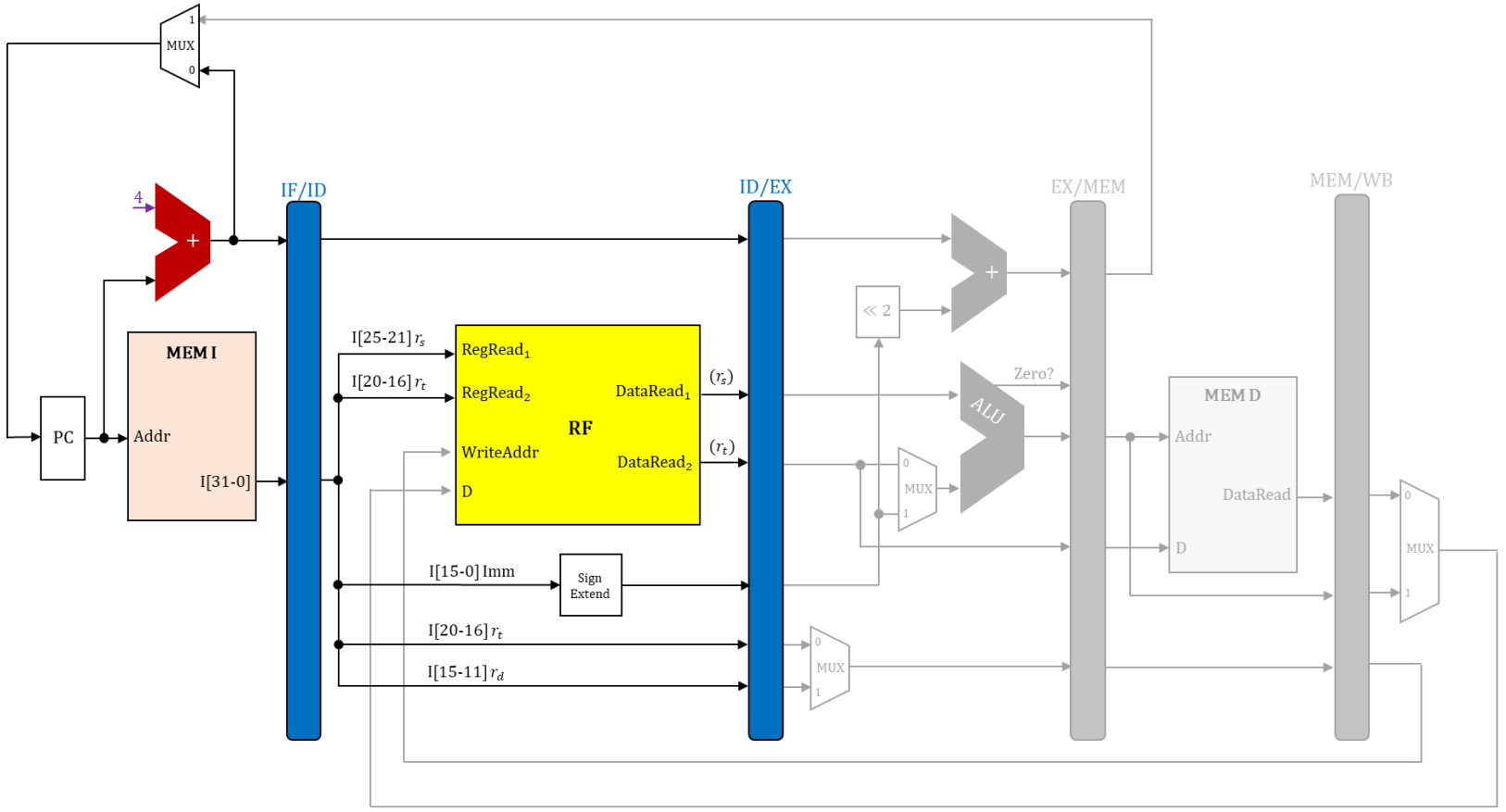


Ciclo di clock 2

- Il contenuto dei registri 3 e 13, (numeri letti dal campo I di IF/ID) viene prelevato dal RF e caricato in ID/EX (campi A e B)
- Gli altri campi di I che serviranno ad una fase successiva vengono copiati in ID/EX (**forwarding di informazioni**)

Esempio di esecuzione (ciclo di clock 2)

- **Il PC contiene l'indirizzo dell'istruzione beq $3 $0 32**
- lw $13 40($3) in fase di ID



Ciclo di clock 2

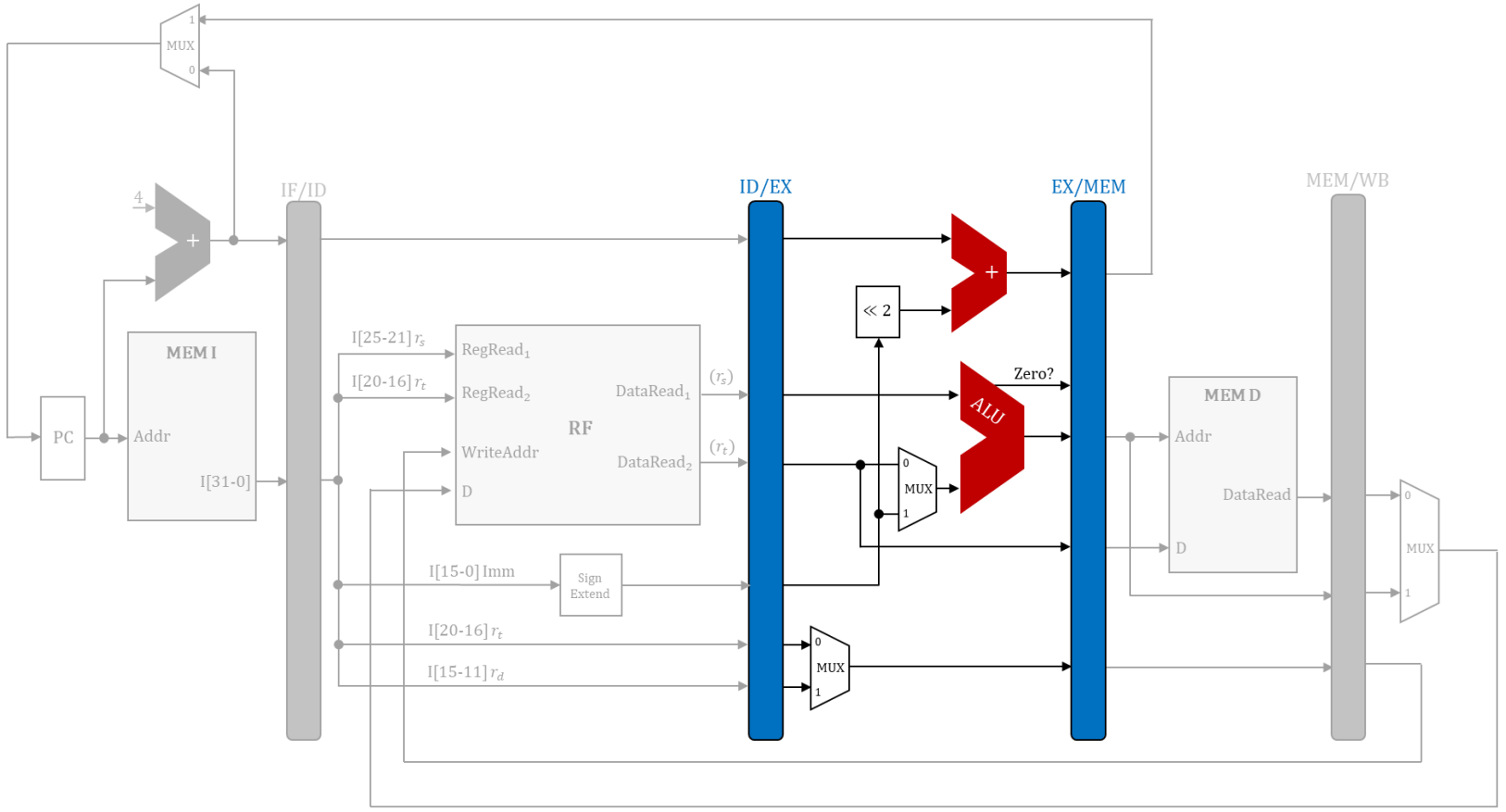
- Il contenuto dei registri 3 e 13, (numeri letti dal campo I di IF/ID) viene prelevato dal RF e caricato in ID/EX (campi A e B)
- Gli altri campi di I che serviranno ad una fase successiva vengono copiati in ID/EX (**forwarding di informazioni**)

Contemporaneamente

- La beq viene prelevata da MEM I e salvata nel registro di transizione IF/ID (campo I)
- Il PC viene sovrascritto con PC+4 (l'indirizzo della prossima istruzione)

Esempio di esecuzione (ciclo di clock 3)

- beq $3 $0 32 in fase di ID
- **lw $13 40($3) in fase di EX**

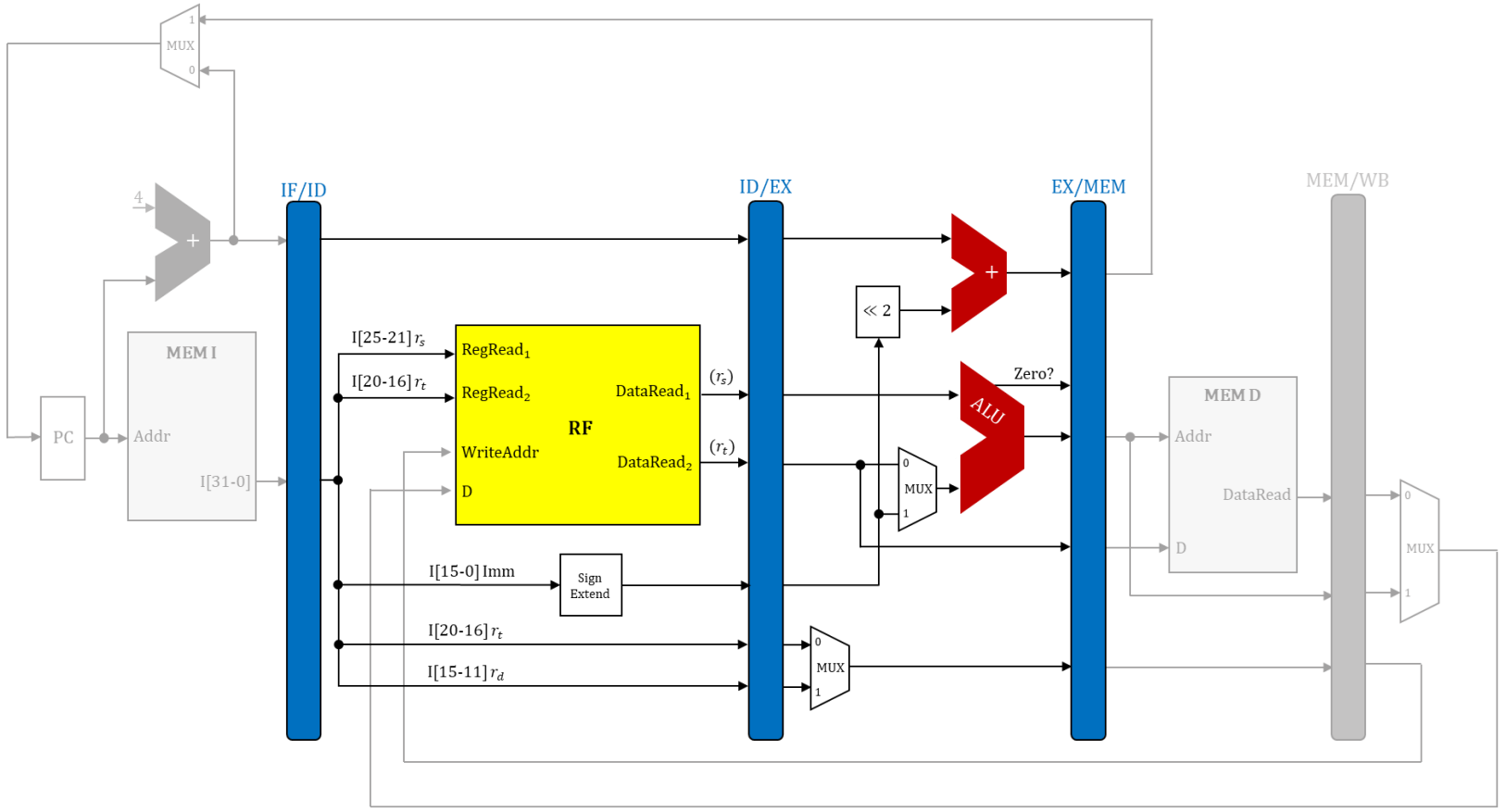


Ciclo di clock 3

- La ALU calcola l'indirizzo a cui accederà la lw per recuperare il dato, viene salvato in EX/MEM
- Il sommatore, parallelo, calcola un BTA, viene salvato in EX/MEM, è un scarto residuo

Esempio di esecuzione (ciclo di clock 3)

- beq $3 $0 32 in fase di ID
- **lw $13 40($3) in fase di EX**



Ciclo di clock 3

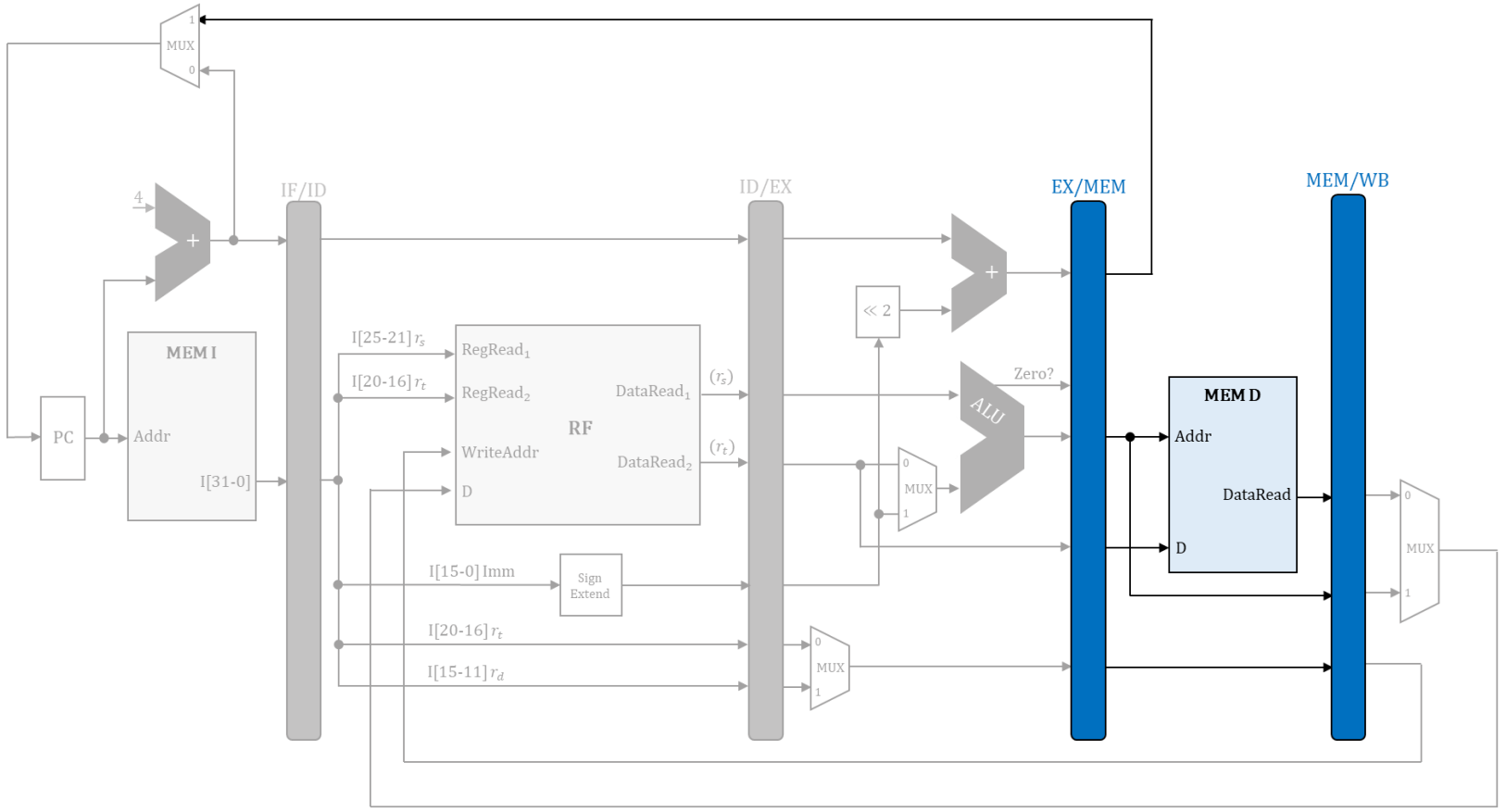
- La ALU calcola l'indirizzo a cui accederà la lw per recuperare il dato, viene salvato in EX/MEM
- Il sommatore, parallelo, calcola un BTA, viene salvato in EX/MEM, è un scarto residuo

Contemporaneamente

- Il contenuto dei registri 3 e 0, (numeri letti dal campo I di IF/ID) viene prelevato dal RF e caricato in ID/EX
- Gli altri campi di I che serviranno ad una fase successiva , tra cui l'offset per il calcolo del BTA, vengono copiati in ID/EX

Esempio di esecuzione (ciclo di clock 4)

- beq $3 $0 32 in fase di EX
- **lw $13 40($3) in fase di MEM**

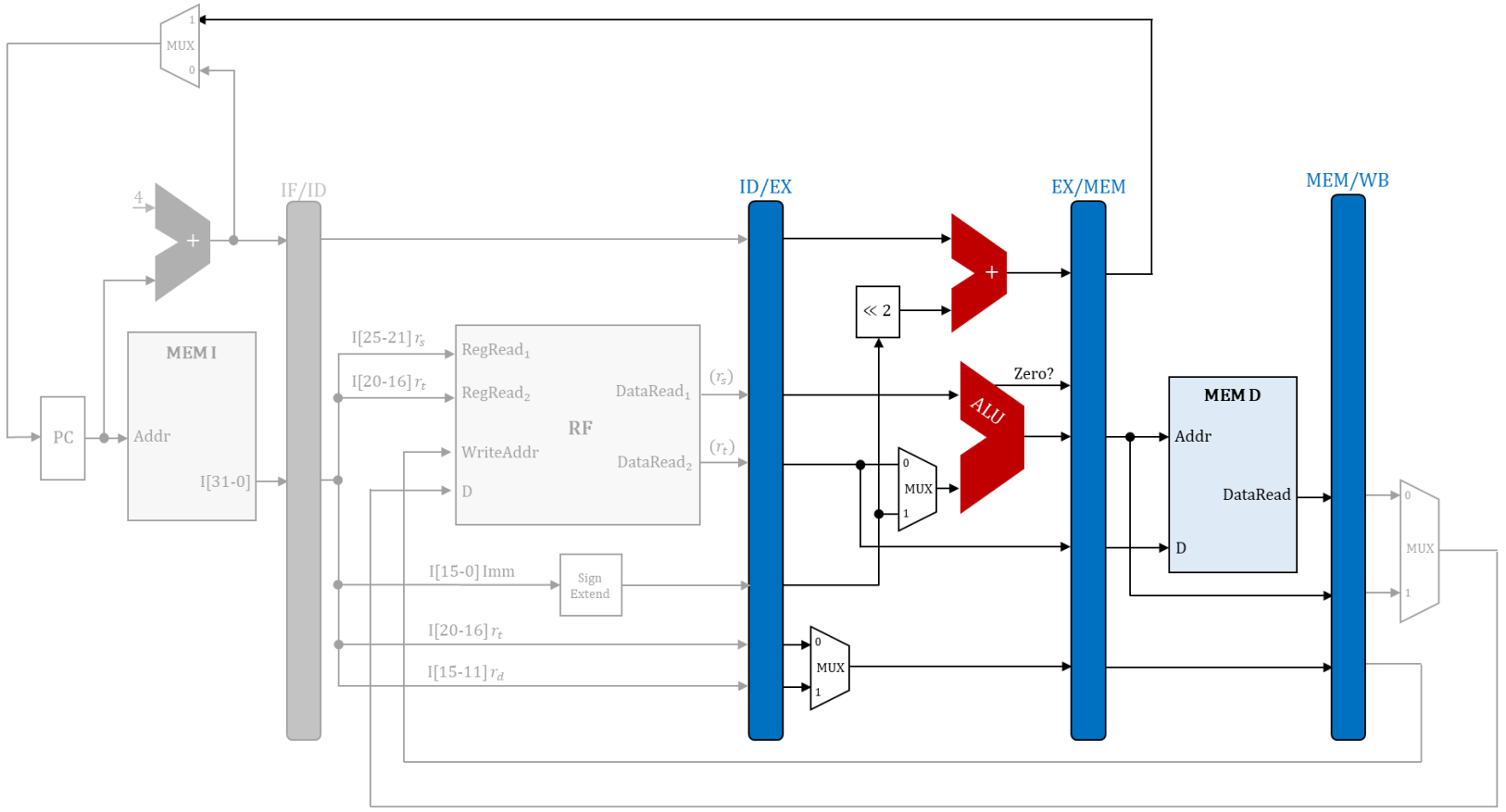


Ciclo di clock 4

- La MEM D recupera il dato per la lw, l'indirizzo di accesso è letto da EX/MEM, il dato è scritto in MEM/WB

Esempio di esecuzione (ciclo di clock 4)

- **beq $3 $0 32 in fase di EX**
- lw $13 40($3) in fase di MEM



Ciclo di clock 4

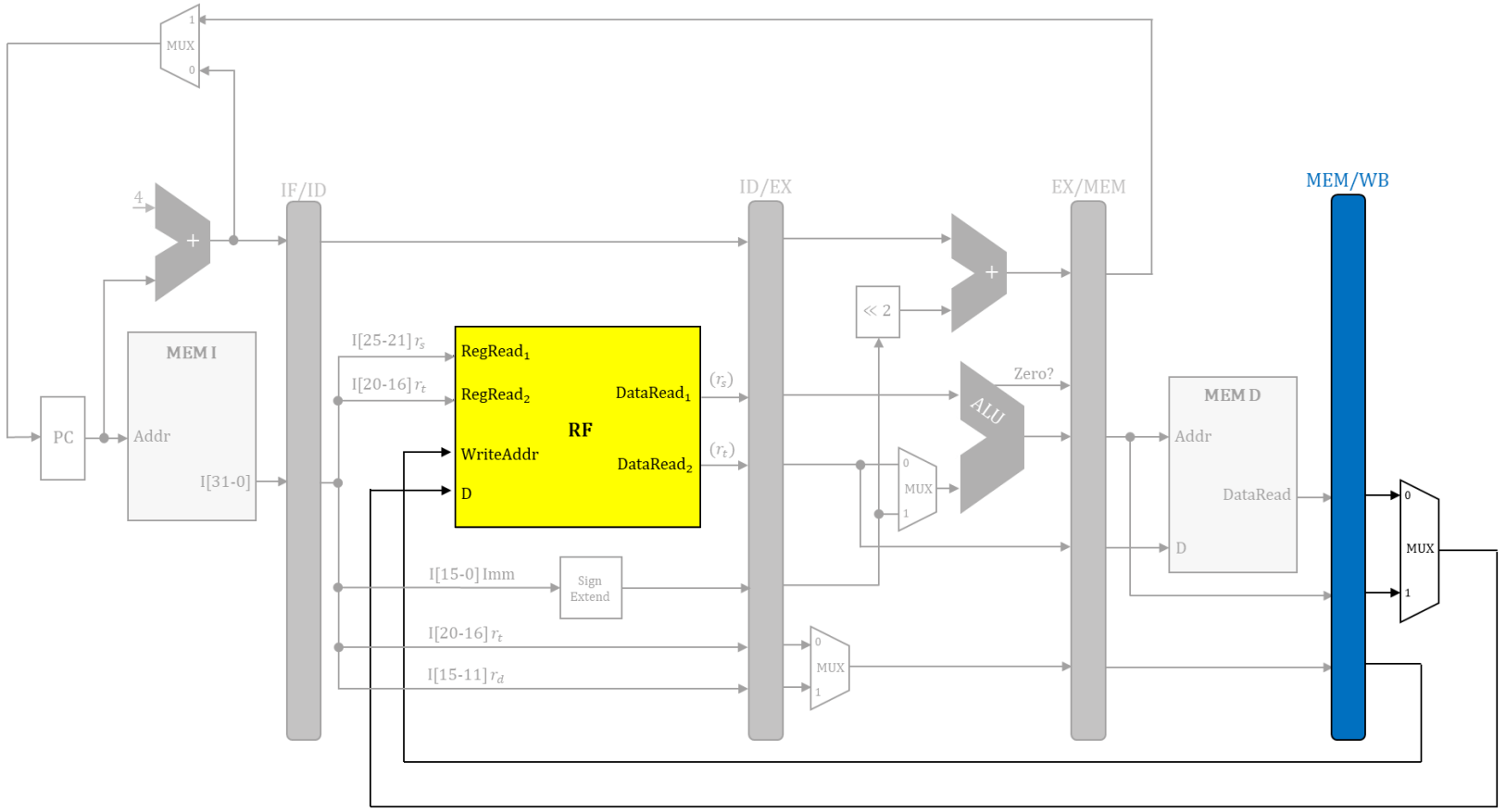
- La MEM D recupera il dato per la lw, l'indirizzo di accesso è letto da EX/MEM, il dato è scritto in MEM/WB

Contemporaneamente

- La ALU fa la differenza tra i contenuti dei registri 3 e 0, il risultato e il bit di zero sono salvati in EX/MEM
- Il sommatore, in parallelo, calcola il BTA che viene salvato in EX/MEM

Esempio di esecuzione (ciclo di clock 5)

- beq $3 $0 32 in fase di MEM
- **lw $13 40($3) in fase di WB**

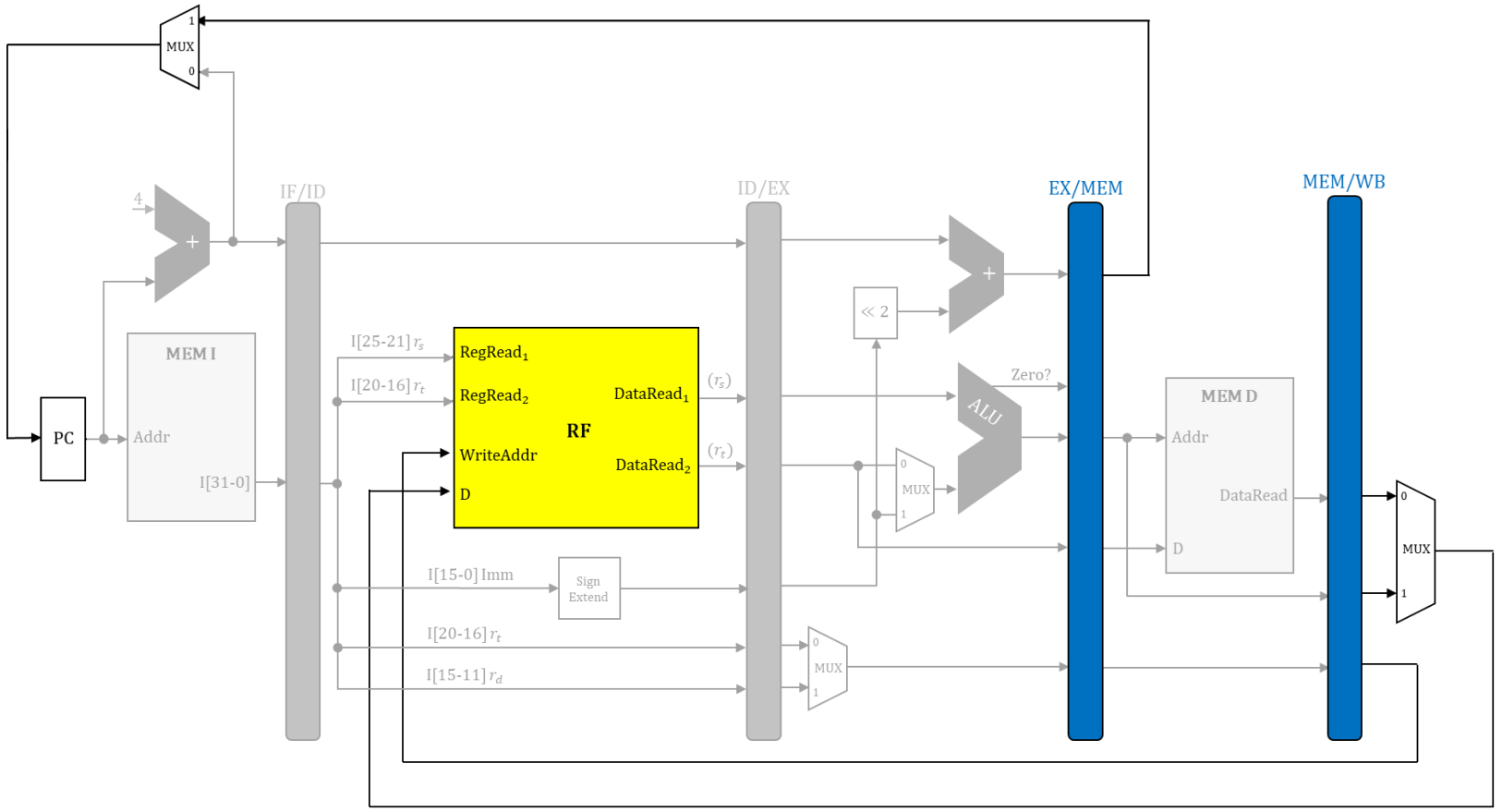


Ciclo di clock 5

- Il dato recuperato viene scritto nel RF nel registro 13

Esempio di esecuzione (ciclo di clock 5)

- **beq $3 $0 32 in fase di MEM**
- lw $13 40($3) in fase di WB



Ciclo di clock 5

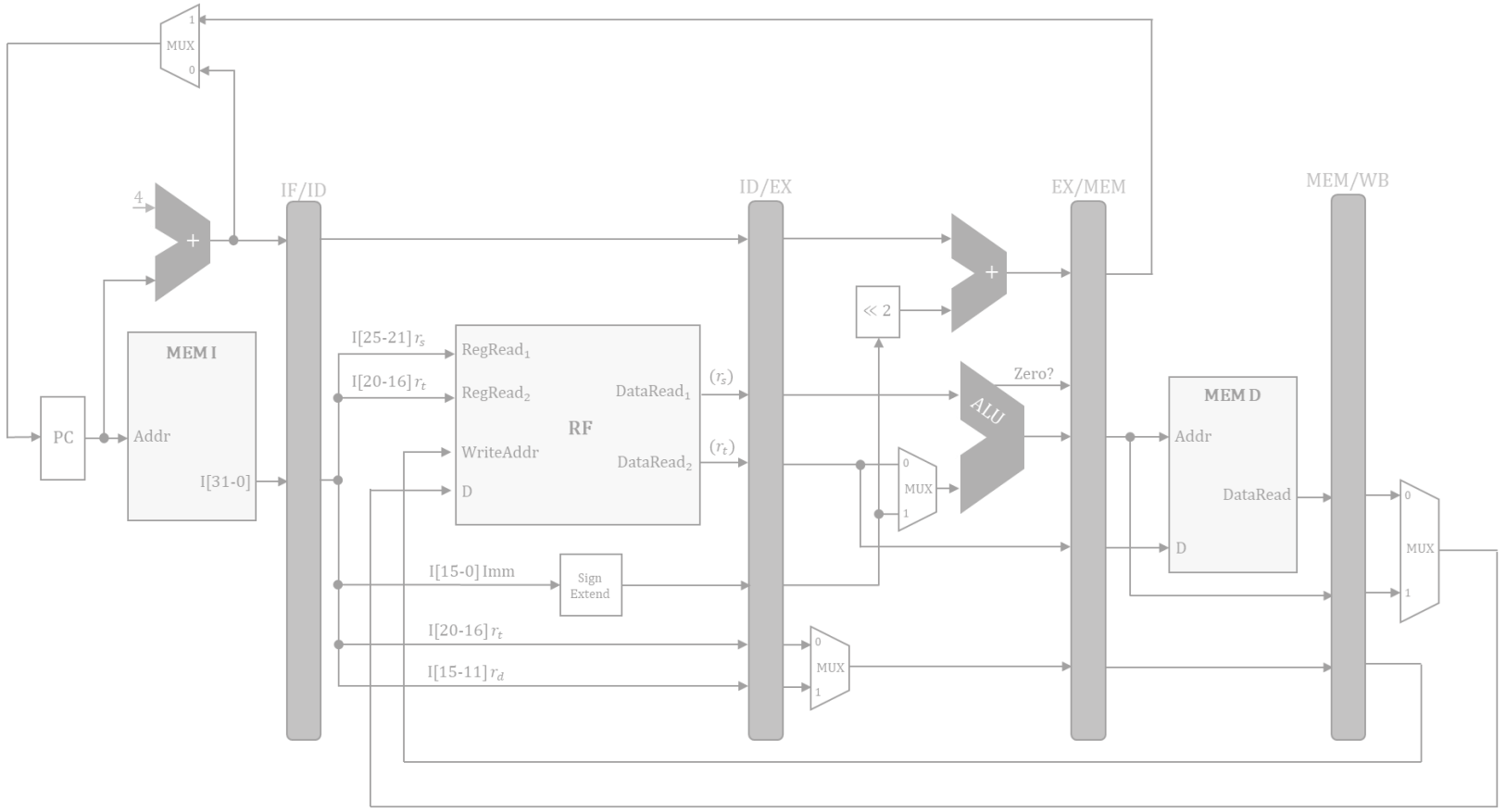
- Il dato recuperato viene scritto nel RF nel registro 13

Contemporaneamente

- Se il bit di zero (presente in EX/MEM) è pari ad 1, viene scritto il BTA (in EX/MEM) nel PC
- Altrimenti non succede niente

Esempio di esecuzione (ciclo di clock 6)

- **beq $3 $0 32 in fase di WB**
- lw $13 40($3) fuori dalla pipeline!

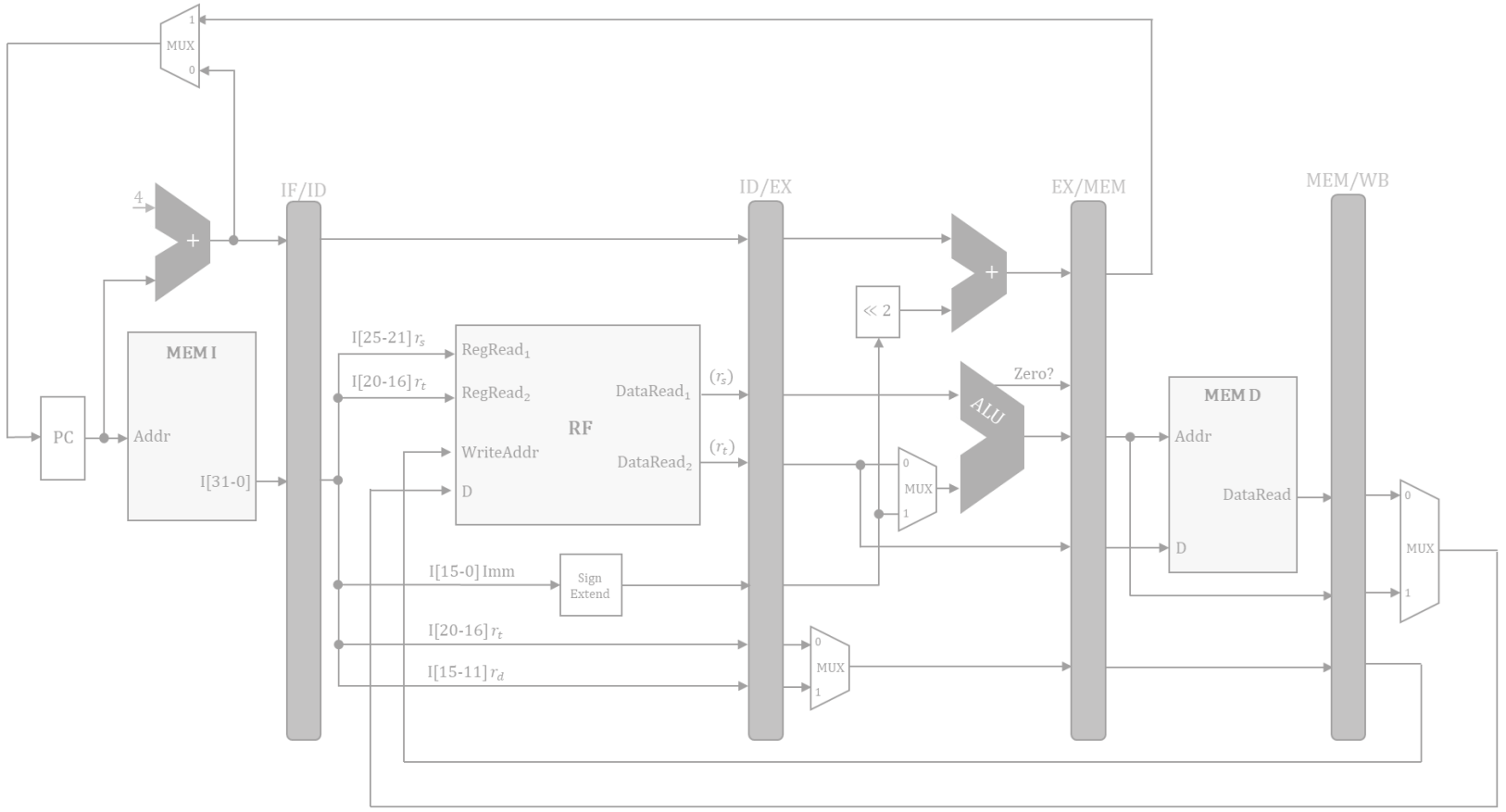


Ciclo di clock 6

- La beq non deve mai scrivere nel RF, non succede niente

Esempio di esecuzione (ciclo di clock 7)

- beq \$3 \$0 32 fuori dalla pipeline!
- lw \$13 40(\$3) fuori dalla pipeline!



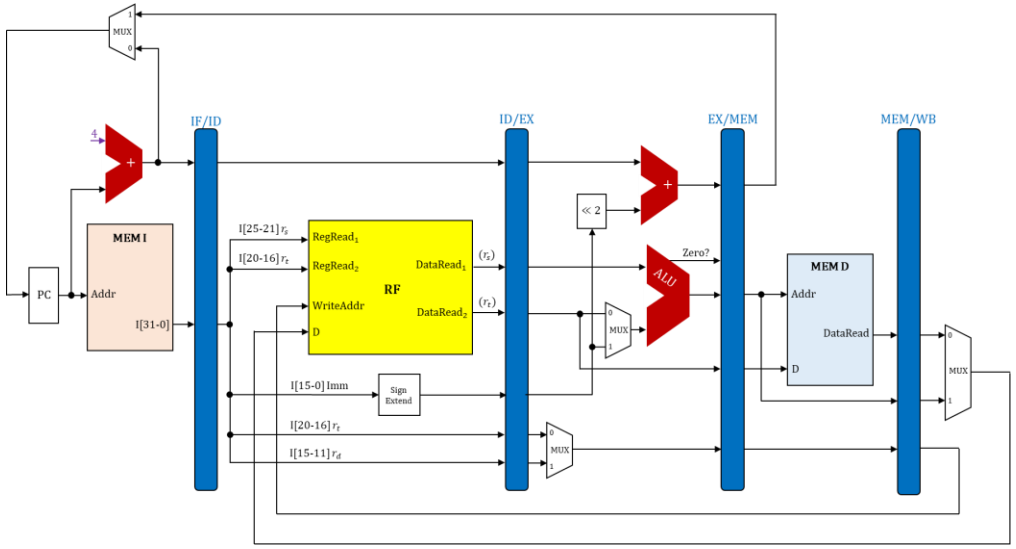
Segnali di controllo della CPU a pipeline

CPU a pipeline: segnali di controllo

- Segnali di selezione

ALUSrc	Selezione del secondo operando ALU: (rt) (0) immediato esteso di segno (1)
RegDest	Selezione campo WriteAddr del RF: rt (0), rd (1)
MemToReg	Selezione del dato presentato in scrittura al RF: risultato ALU (0), contenuto di dato letto da MEM D (1)
AluCtrl	Selezione della modalità di operazione della ALU (analogo a CPU singolo ciclo)
Branch	Indica se istruzione corrente è una beq

PCSrc	Selezione di quale indirizzo mandare al PC: PC+4 (0), BTA (1)



- **Nota 1**: PCSrc sarà calcolato direttamente nel datapath come Branch AND Zero
- **Nota 2**: se aggiungessimo il supporto a jump, il MUX controllato da PCSrc dovrebbe avere una linea in più e servirebbe un altro segnale di selezione

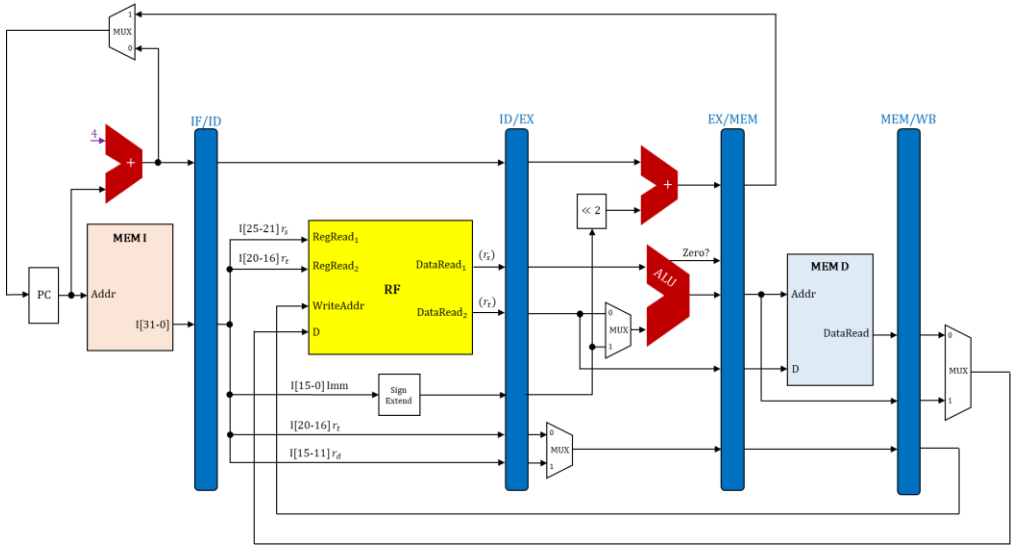
Ricorda:

Opcode	AluCtrl
Tipo R, rimanda a funct	10
lw, somma	00
sw, somma	00
beq, sottrazione	01

CPU a pipeline: segnali di controllo

- Segnali di comando

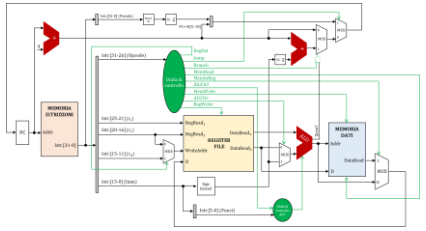
RegWrite	Scrittura del Register File
MemWrite	Scrittura della memoria
MemRead	Lettura della memoria



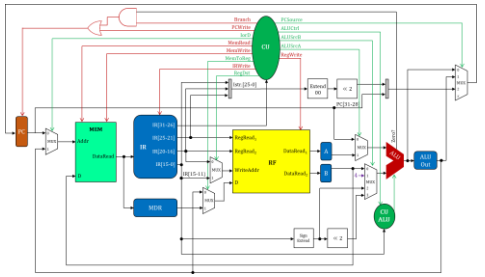
- La scrittura in tutti gli altri registri non menzionati è data dal clock, vengono scritti ad ogni ciclo di clock (fronte di discesa)
- **Problema**: come organizziamo i segnali di controllo nel datapath?

CPU a pipeline: logica di controllo

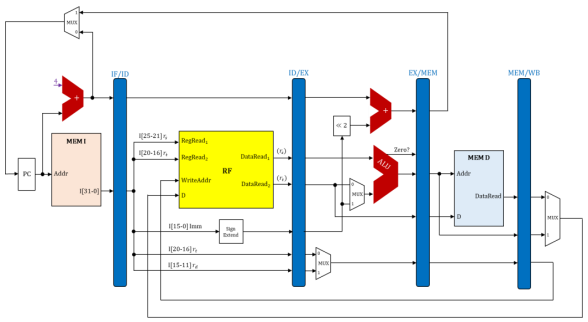
- La logica di controllo di una CPU a pipeline presenta alcuni aspetti peculiari, dovuti a questa specifica architettura
- Data una istruzione i l'unità di controllo calcola tutti i segnali di selezione e comando



- Nella **CPU a singolo ciclo**: i segnali configurano il datapath in modo che, attraversandolo, i svolga tutte e 5 le fasi di esecuzione



- Nella **CPU a ciclo multiplo**: i segnali configurano il datapath in modo che, attraversandolo, i svolga una singola fase di esecuzione



- Nella **CPU a pipeline**: i segnali devono configurare il datapath **di ogni stadio** in modo coerente con l'istruzione che è in esecuzione in quello stadio in quel ciclo di clock (quindi diversamente a seconda dell'istruzione)
- Visto dal punto di vista di una singola istruzione i: in ogni stadio della pipeline che attraversa avrà bisogno che quello stadio venga configurato come lei necessita, attraverso i **suoi** segnali di controllo!

Ricorda:
Nella pipeline ci possono essere fino a 5 istruzioni in esecuzione!

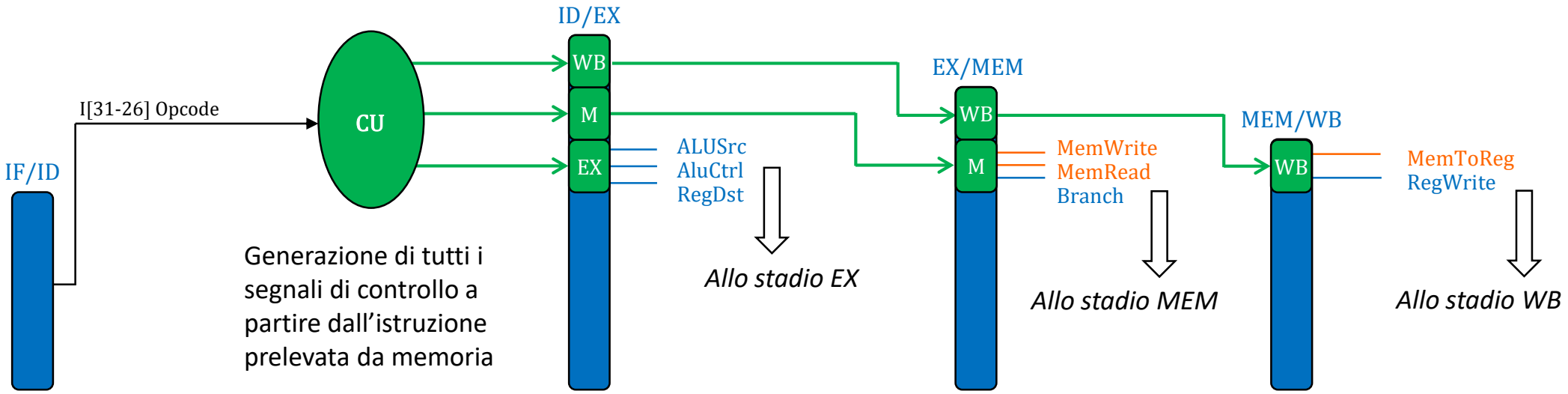
CPU a pipeline: logica di controllo

- Idea: «trasporto» nella pipeline i segnali di controllo di ogni singola istruzione, analogamente con quanto fatto per i risultati parziali della sua elaborazione
- All'ingresso di uno stadio della pipeline, nel registro pipeline da cui lo stadio legge avremo i risultati parziali e i segnali di controllo che servono a quella istruzione per quello stadio
- **Quali sono i segnali che dobbiamo trasportare nella pipeline?**
- **Nella fase IF** non esistono segnali di controllo
- **Nella fase ID** non si dipende dai segnali di controllo che in questa fase vengono tutti generati a partire dall'istruzione

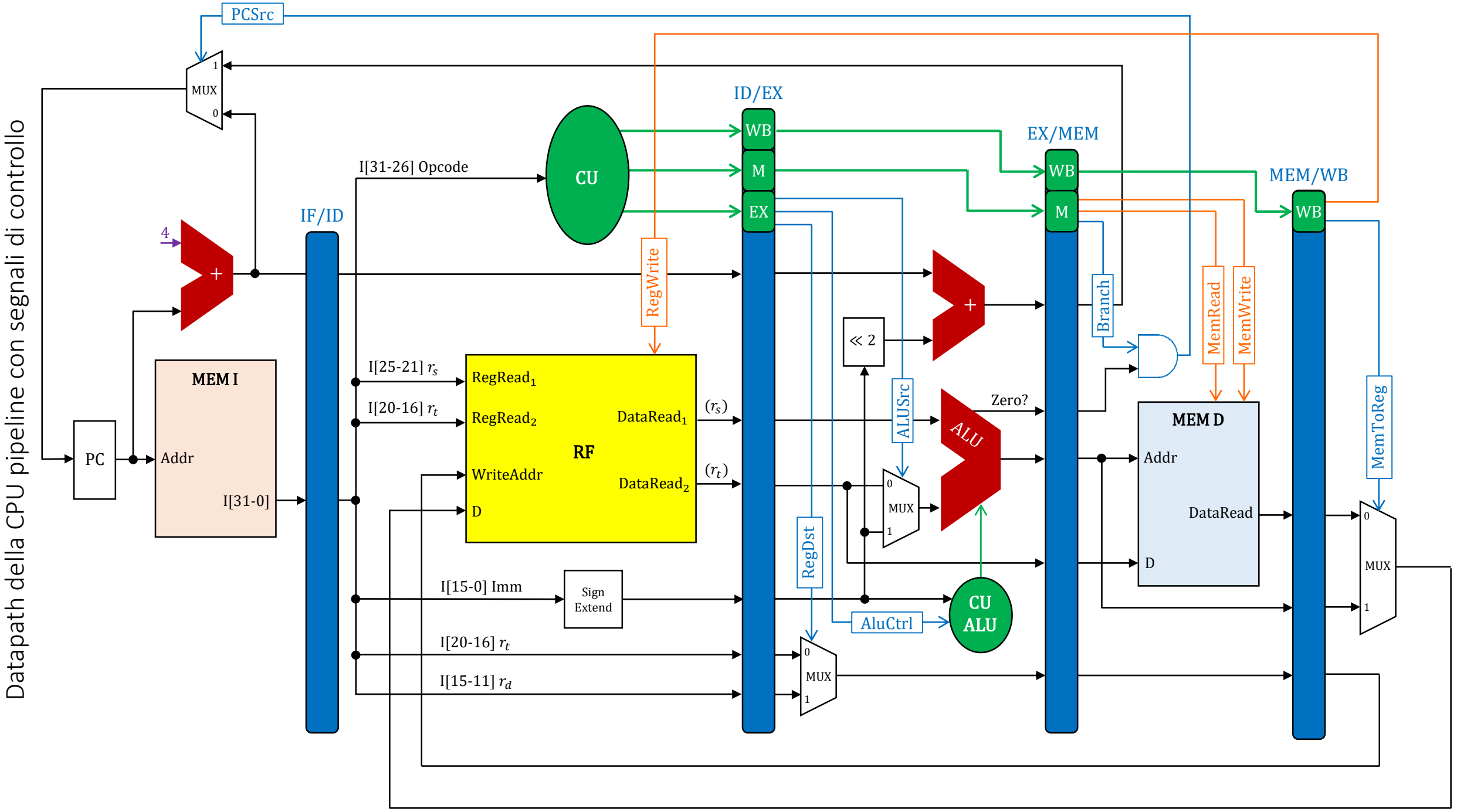
	EX			MEM			WB	
	ALUSrc	RegDst	AluCtrl	Branch	MemWrite	MemRead	MemToReg	RegWrite
A/L	0	1	10	0	0	0	0	1
lw	1	0	00	0	0	1	1	1
sw	1	x	00	0	1	0	x	0
beq	0	x	01	1	0	0	x	0

CPU a pipeline: datapath logica di controllo

- Estendiamo i registri pipeline e il datapath per supportare il trasporto dei segnali di controllo stadio dopo stadio, in modo solidale con l'istruzione



- Ogni stadio può lavorare in parallelo all'istruzione correntemente al suo interno leggendo i risultati parziali e i segnali di controllo dal registro pipeline alla sua sinistra
- Verso lo stadio successivo verranno inoltrati sia i risultati parziali che i segnali di controllo che serviranno più avanti


Datapath della CPU pipeline con segnali di controllo
PCSrc
MUX
1
0
4
+
PC
MEM I
Addr
I[31-0]
IF/ID
I[31-26] Opcode
CU
ID/EX
WB
M
EX
RegWrite
I[25-21] r_s
I[20-16] r_t
RegRead$_1$
RegRead$_2$
RF
WriteAddr
D
DataRead$_1$
DataRead$_2$
(r_s)
(r_t)
I[15-0] Imm
Sign Extend
I[20-16] r_t
I[15-11] r_d
RegDst
≪ 2
ALUSrc
ALU
Zero?
AluCtrl
CU ALU
EX/MEM
Branch
MemRead
MemWrite
MEM D
Addr
DataRead
D
MEM/WB
MemToReg

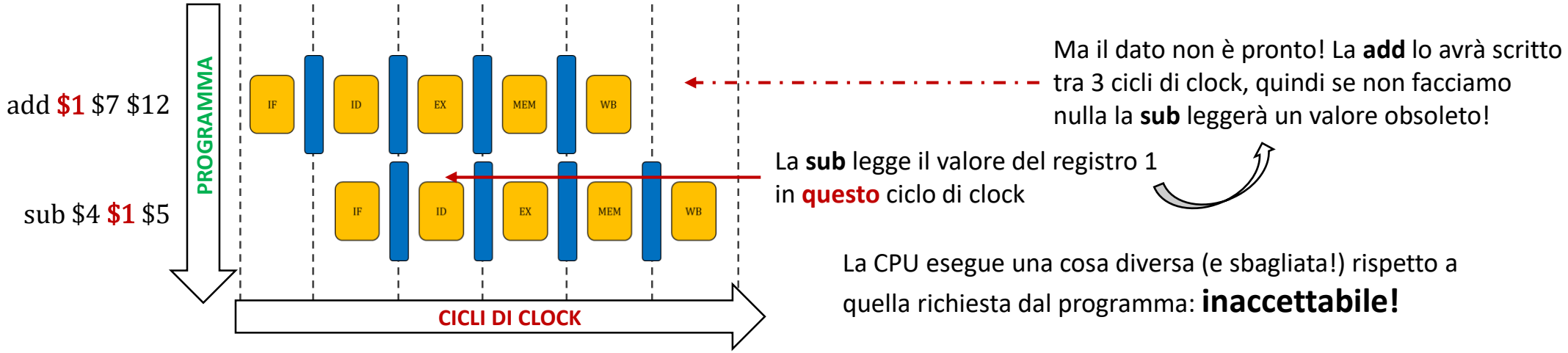
Data hazards

Criticità

- L'architettura a pipeline che abbiamo introdotto ci ha già richiesto di risolvere due problematiche strutturali
  - Ridefinizione del datapath con duplicazione delle risorse (criticità strutturali)
  - Estensione del datapath e dei registri di transizione per lavorare in pipeline anche con i segnali di controllo
- Le abbiamo risolte, ma i problemi più interessanti sono dati da criticità che possono affliggere non le risorse hardware ma la **correttezza del flusso di esecuzione**
- **Problema principale**: le istruzioni non sono indipendenti l'una dall'altra!
- **Esempi**
  - Una **add** potrebbe dover usare un operando che è il risultato di una **sub** precedente, la **add** potrebbe trovarsi nella fase di EX quando la **sub** non ha ancora concluso la fase di WB!
  - Una **j** potrebbe non dover essere eseguita se viene svolto un salto condizionato di una **beq** che la precede, ma la sua esecuzione potrebbe essere iniziata prima di sapere l'esito della condizione nella branch!
- La CPU a pipeline che abbiamo progettato non è ancora in grado di prevenire/gestire questi eventi che possono compromettere la correttezza dell'esecuzione di un programma

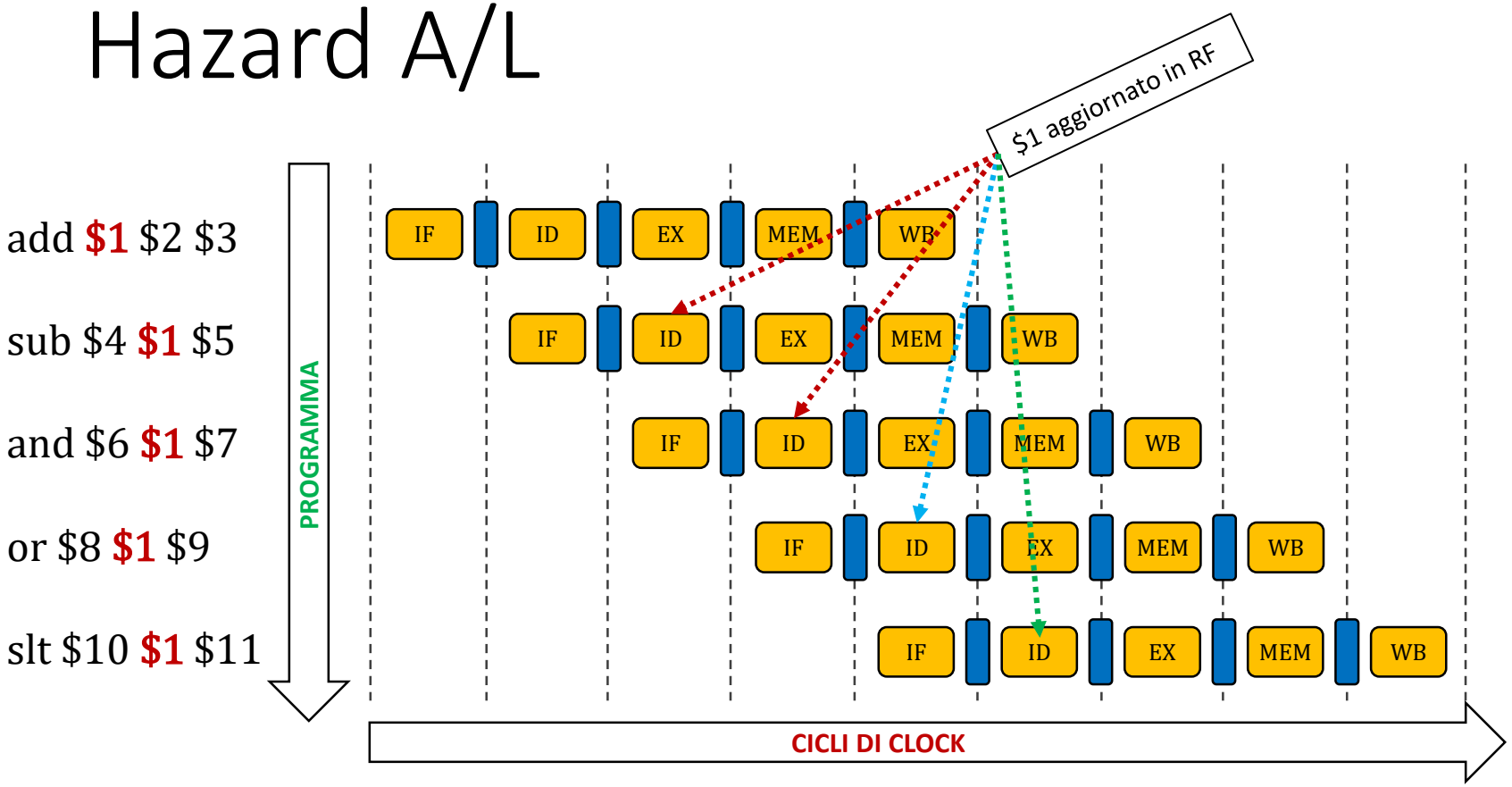
Criticità di dato

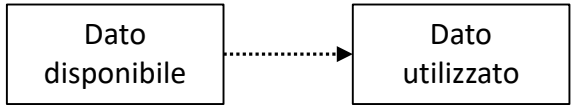
- Criticità strutturali (structural hazards): **risolte**
- **Criticità di dato** (data hazards)
  - Un'istruzione j deve scrivere un valore nel registro r
  - Un'istruzione i, successiva a j, deve usare il contenuto di r
  - Quando i recupera (r) dal RF (fase ID) j è ancora in pipeline e non ha svolto la WB



- **Data hazard**: opero su un dato prima che la fase di WB, che aggiorna il suo valore nel RF, sia conclusa
- **Ma quel valore corretto, calcolato da una istruzione precedente non ancora conclusa, dove viene prodotto all'interno della pipeline?**
  - Caso 1: il valore corretto del dato è prodotto (calcolato) nella fase EX (istruzioni A/L)
  - Caso 2: il valore corretto del dato è prodotto (prelevato) nella fase MEM (lw)

Hazard A/L





add **$1** $2 $3

sub $4 **$1** $5

and $6 **$1** $7

or $8 $1 $9

slt $10 $1 $11

Hazard A/L solo tra:

- Una istruzione e la precedente
- Una istruzione e la precedente della precedente

Hazard! La sub e la and operano su $1, ma il valore che assumono di avere non è ancora stato calcolato!

Dipendenza interna alla pipeline: le scritture nel RF sono fatte nella prima metà del ciclo di clock, le letture nella seconda, grazie a questo la or accede al dato corretto, nessun problema

Esecuzione corretta

Hazard A/L (approccio SW)

- **Prima soluzione**: intervenire sul software

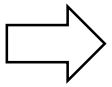
```
add $1 $2 $3          add $1 $2 $3
sub $4 $1 $5          nop
and $6 $1 $7    ⇨     nop
or $8 $1 $9           sub $4 $1 $5
slt $10 $1 $11        and $6 $1 $7
                      or $8 $1 $9
                      slt $10 $1 $11
```

- **nop**: istruzione speciale che svolge un'operazione fittizia che **non provoca effetti** sulla normale esecuzione (no operation)
- **Esempio** add $0 $0 $0 (in MIPS il registro $0 contiene sempre la costante $0 per convenzione)
- Effetto di una nop: la CPU aspetta per 1 ciclo di clock senza fare niente, come se inserissimo una bolla nella pipeline
- Chi dovrebbe svolgere questo lavoro sul codice? Idealmente il compilatore
- **Pro**: con questo approccio risolviamo i data hazard
- **Contro**: molto inefficiente, si perdono un sacco di cicli di clock

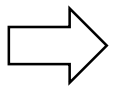
Hazard A/L (approccio SW)

- Per mitigare il problema il compilatore potrebbe provare a riorganizzare il codice

```
add $1 $2 $3
sub $4 $1 $5
and $6 $1 $7
or $8 $1 $9
slt $10 $1 $11
```



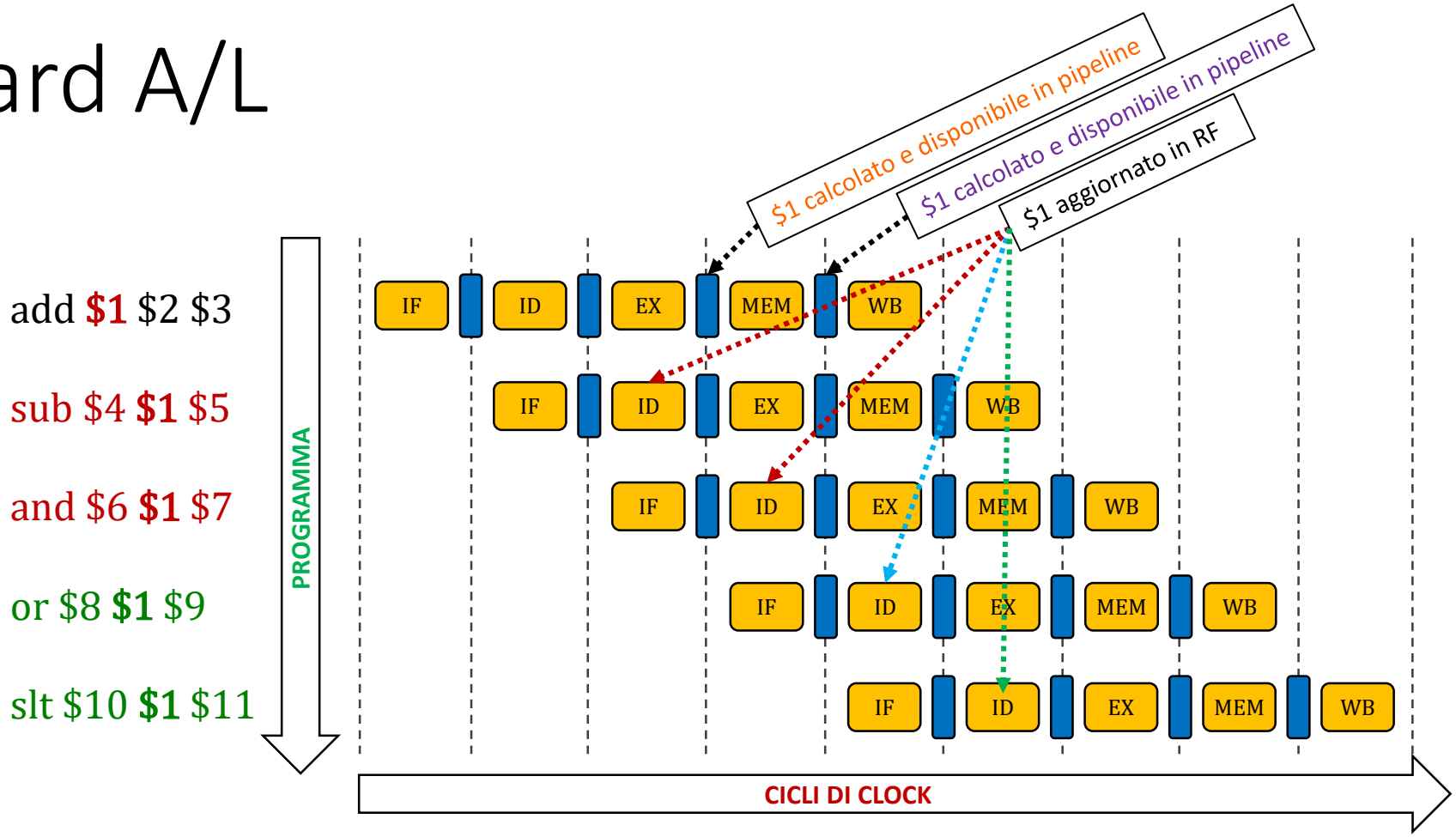
```
add $1 $2 $3
nop
nop
sub $4 $1 $5
and $6 $1 $7
or $8 $1 $9
slt $10 $1 $11
```



```
add $1 $2 $3
add $31 $30 $29
add $25 $26 $27
sub $4 $1 $5
and $6 $1 $7
or $8 $1 $9
slt $10 $1 $11
⋮
```

- Queste sono istruzioni che sarebbero venute dopo, ma la cui esecuzione può essere anticipata senza compromettere la correttezza del programma: sarebbero comunque state eseguite, non generano data hazard
- Fare questo genere di ottimizzazioni sul codice è molto complesso e comunque non vi è garanzia di riuscire a riempire tutte le «bolle»

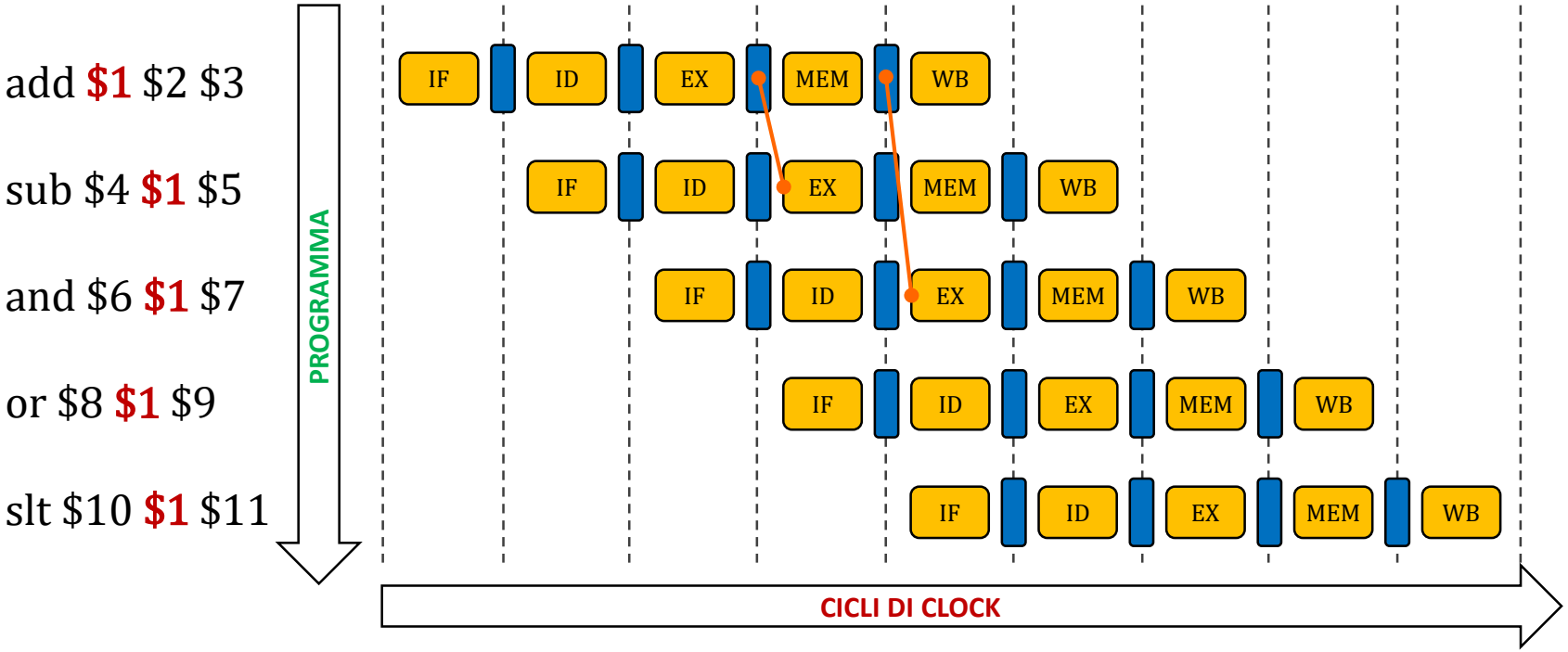
Hazard A/L



- Le due istruzioni su cui si verifica l'hazard **non devono necessariamente aspettare che il dato venga scritto nel RF!**
  - Quando sub entra in EX, in ID/EX *(alla sua sinistra)* c'è il valore obsoleto di $1, **ma il valore corretto è in pipeline dentro il registro EX/MEM** (calcolato nel ciclo precedente, *subito alla sua destra!*)
  - Quando and entra in EX, in ID/EX *(alla sua sinistra)* c'è il valore obsoleto di $1, **ma il valore corretto è in pipeline dentro il registro MEM/WB** (calcolato due cicli di clock prima, *nel secondo registro pipeline alla sua destra!*)

Hazard A/L (approccio HW)

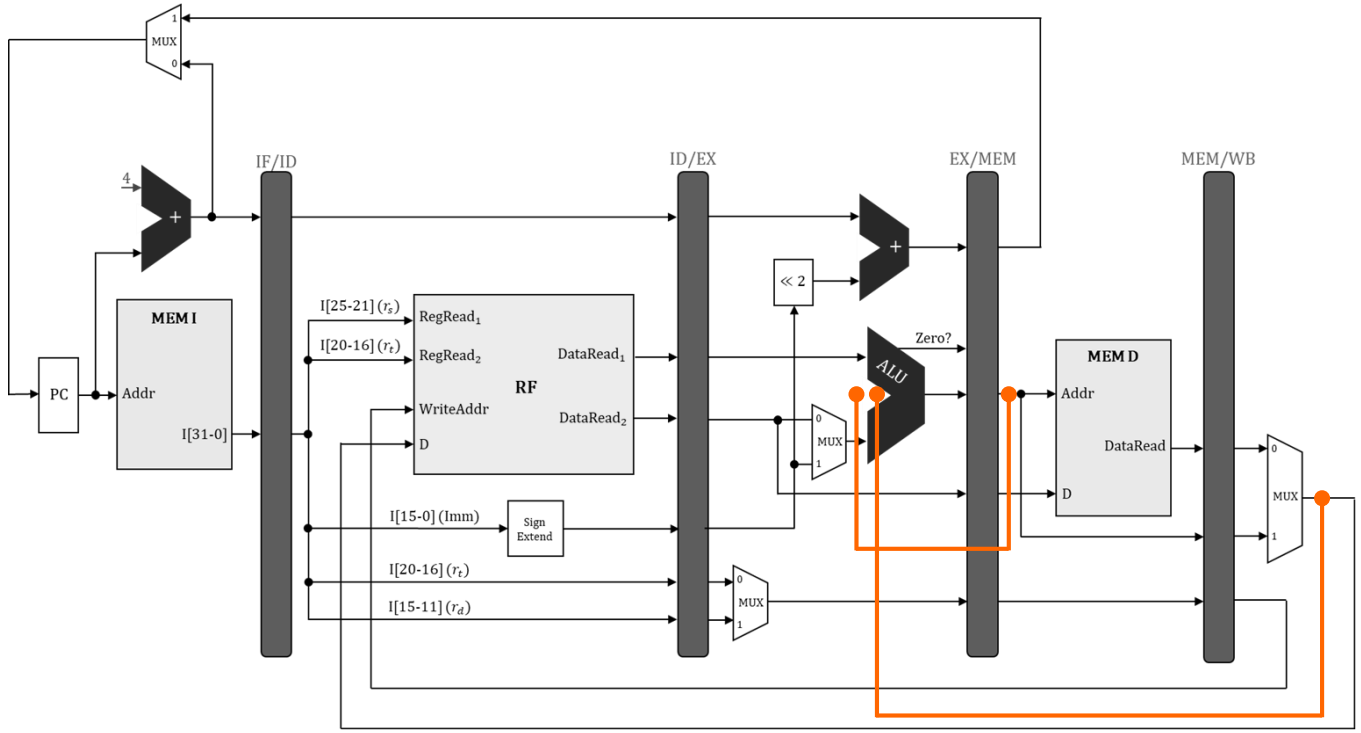
- **Seconda soluzione**: intervenire sull'hardware
- **Idea**: costruiamo delle scorciatoie all'interno della pipeline in modo che un dato pronto possa essere inoltrato all'indietro nella pipeline **dallo stadio che lo ha disponibile ad uno precedente**
- Anticipiamo il momento in cui un'istruzione successiva può usare il valore di un registro, non deve più aspettare che l'istruzione che lo aggiorna concluda l'attraversamento dalla pipeline



- Collegamento aggiuntivo nel datapath (la scorciatoia)
- Consente ad un risultato della fase EX di saltare all'indietro nel punto in cui è richiesto
- Questa tecnica si chiama **forwading**

Hazard A/L: forwarding

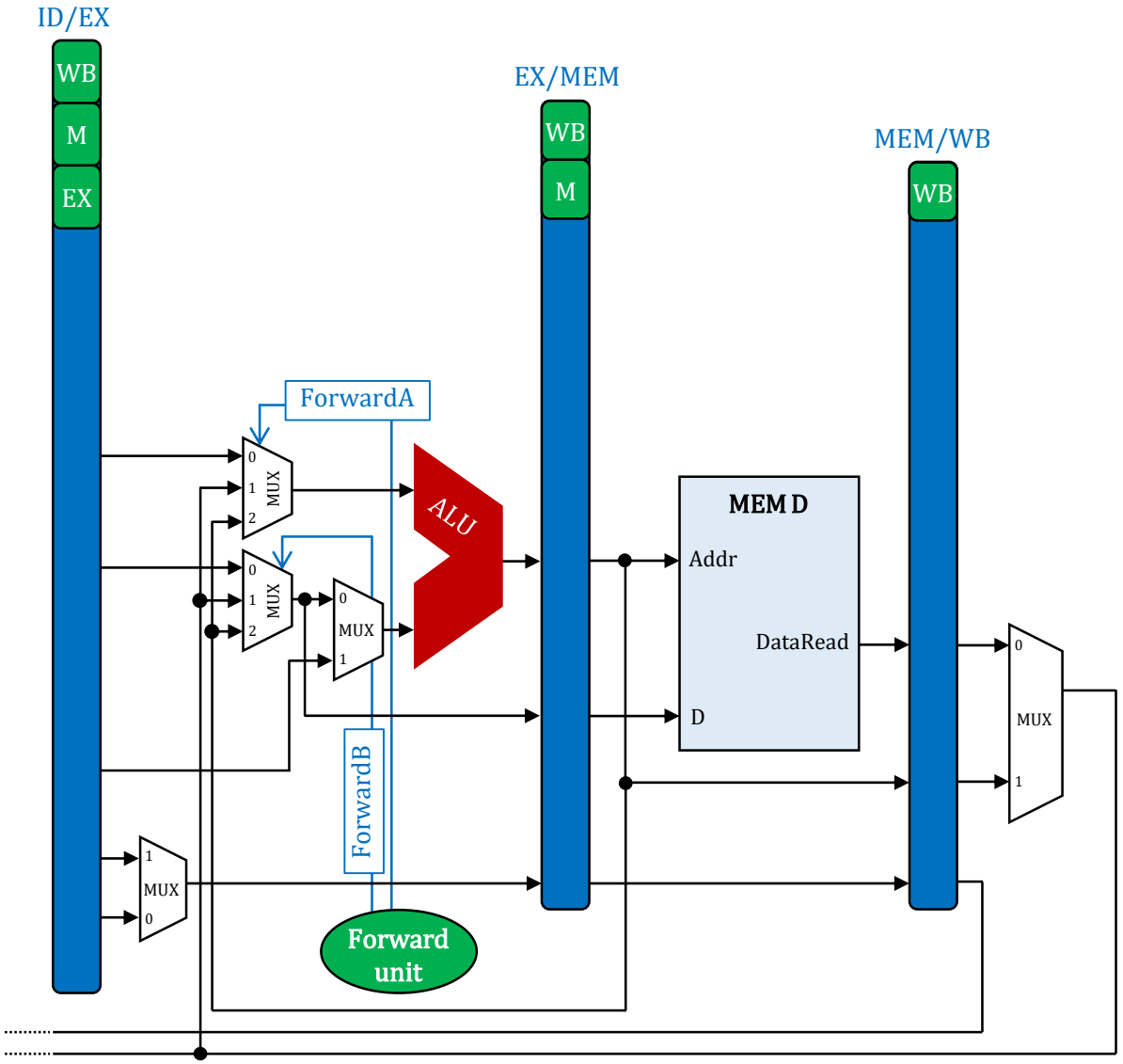
- Dobbiamo modificare il datapath



- Usiamo la dot notation per riferirci a particolari registri (campi) contenuti in un registro pipeline, ad es. $\mathrm{IF/ID.I}$, $\mathrm{ID/EX.}(rt)$
- Colleghiamo $\mathrm{EX/MEM.ALUout}$ alla ALU
- Colleghiamo il dato in uscita dalla fase di WB alla ALU: questo è il dato che va al RF per essere scritto in un registro, può essere o il risultato della ALU (quello che ci serve ora) o una dato recuperato dalla memoria (ci tornerà utile dopo)

- Collegare alla ALU significa aggiungere altre possibilità tra i dati che possono presentarsi ai suoi ingressi
- Serviranno **altri MUX**, **altri segnali di controllo** (selezione) e la **logica di controllo** per capire quando e come usare le nuove scorciatoie

Hazard A/L: forwarding datapath



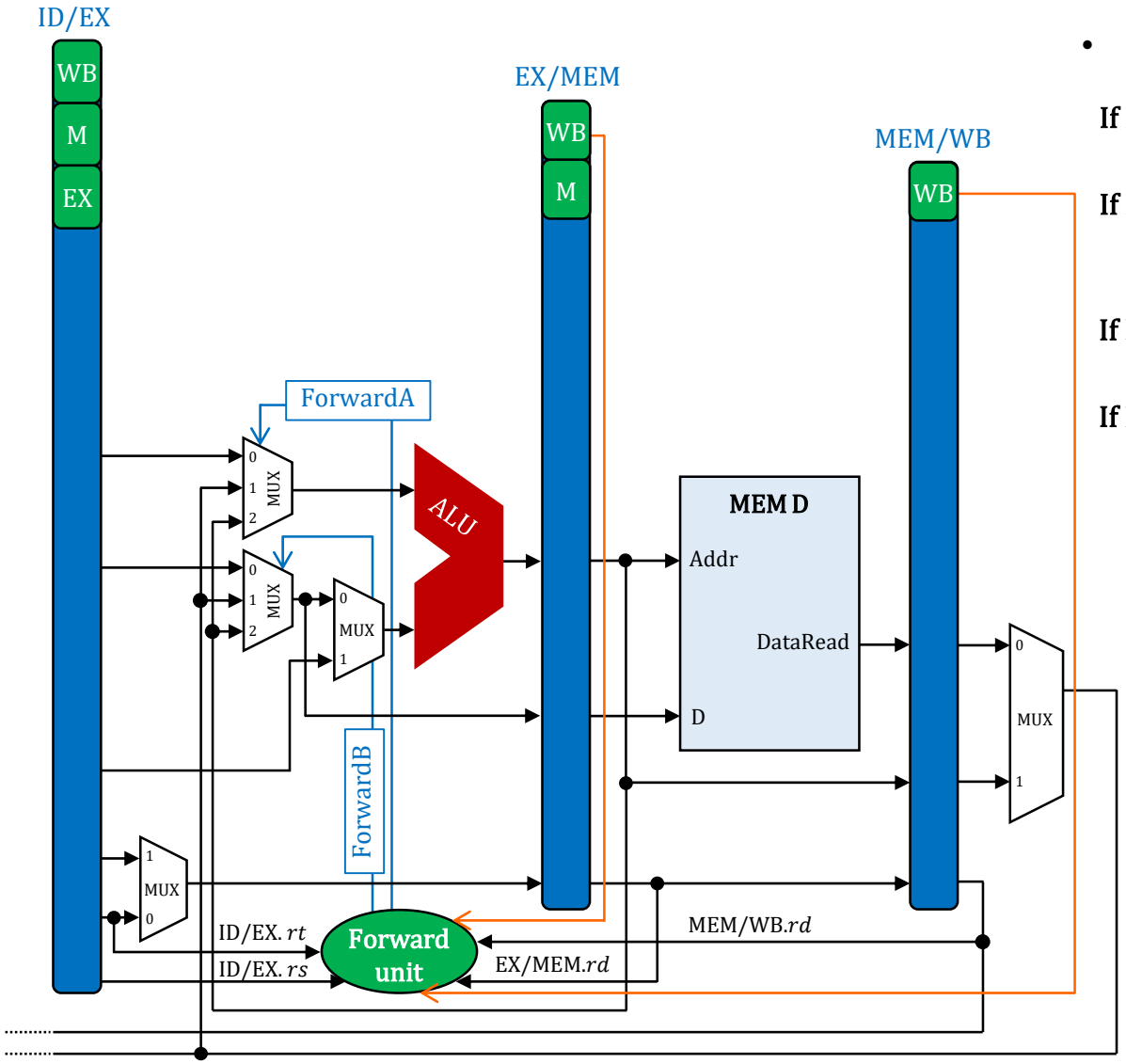
- Aggiungiamo i collegamenti necessari
- Indichiamo gli input della ALU con A (primo, in alto) e B (secondo, in basso)
- Serviranno due nuovi segnali controllo (selezione) per A e B e una unità di controllo dedicata per settarli quando serve: **forward unit**

valore	ForwardA	ForwardB
00	$A \leftarrow (rs)$	$B \leftarrow (rt)$
01	$A \leftarrow$ dato WB	$B \leftarrow$ dato WB
10	$A \leftarrow$EX/MEM.ALUout	$B \leftarrow$EX/MEM.ALUout
11	*non usato*	*non usato*

- Di quali input necessita la forward unit?

Hazard A/L: forward unit

valore	ForwardA	ForwardB
00	$A \leftarrow (rs)$	$B \leftarrow (rt)$
01	$A \leftarrow$ dato WB	$B \leftarrow$ dato WB
10	$A \leftarrow$EX/MEM.ALUout	$B \leftarrow$EX/MEM.ALUout
11	*non usato*	*non usato*



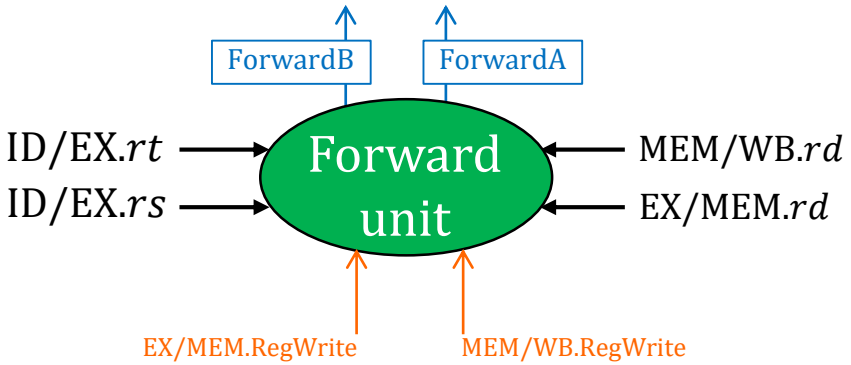
- La forward unit implementa questa funzione logica (combinatoria!):

If ID/EX.rs == EX/MEM.rd && EX/MEM.RegWrite && EX/MEM.rd != 0
ForwardA $\leftarrow$ 10
If ID/EX.rt == EX/MEM.rd && EX/MEM.RegWrite && EX/MEM.rd != 0
ForwardB $\leftarrow$ 10

⇦ Riconosce e risolve hazard A/L tra istruzione e la precedente

If ID/EX.rs == MEM/WB.rd && MEM/WB.RegWrite && MEM/WB.rd != 0
ForwardA $\leftarrow$ 01
If ID/EX. rt == MEM/WB.rd && MEM/WB.RegWrite && MEM/WB.rd != 0
ForwardB $\leftarrow$ 01

⇦ Riconosce e risolve hazard A/L tra istruzione e la precedente della precedente



Nota: prestare bene attenzione a chi «davvero» è MEM/WB.rd o EX/MEM.rd

Hazard A/L: forward unit

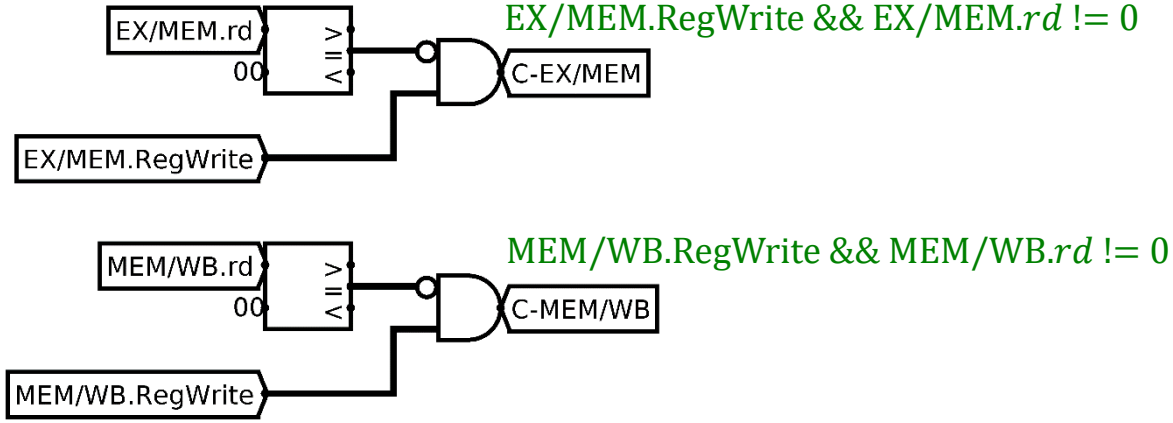
valore	ForwardA	ForwardB
00	$A \leftarrow (rs)$	$B \leftarrow (rt)$
01	$A \leftarrow$ dato WB	$B \leftarrow$ dato WB
10	$A \leftarrow$EX/MEM.ALUout	$B \leftarrow$EX/MEM.ALUout
11	*non usato*	*non usato*

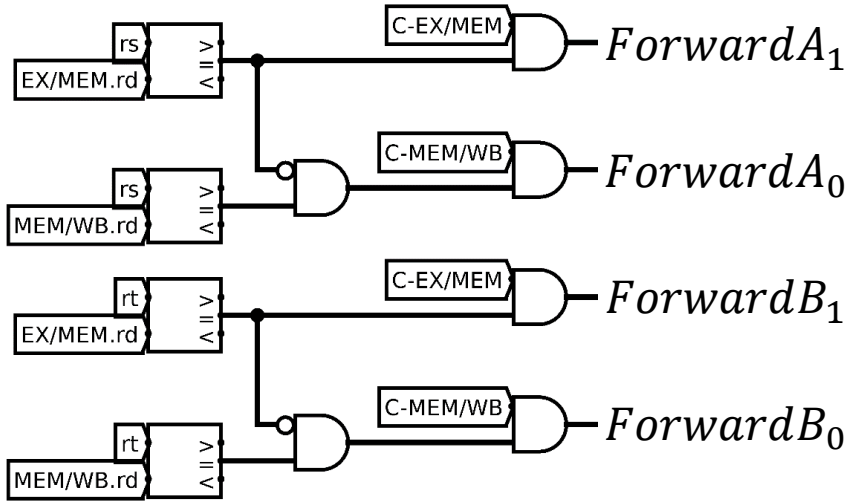
If ID/EX. rs == EX/MEM.rd && EX/MEM.RegWrite && EX/MEM.rd != 0
ForwardA $\leftarrow 10$

If ID/EX.rt == EX/MEM.rd && EX/MEM.RegWrite && EX/MEM.rd != 0
ForwardB $\leftarrow 10$

If ID/EX.rs == MEM/WB.rd && MEM/WB.RegWrite && MEM/WB.rd != 0
ForwardA $\leftarrow 01$

If ID/EX.rt == MEM/WB.rd && MEM/WB.RegWrite && MEM/WB.rd != 0
ForwardB $\leftarrow 01$





- Dobbiamo stare attenti a quando una coppia di condizioni (le due blu o arancioni) sono entrambe vere:
  - Devo evitare output 11 (non usato)
  - Se entrambe sono vere significa che ho un valore per quel dato sia in EX/MEM (calcolato dall'istruzione precedente) che in MEM/WB (calcolato dall'istruzione precedente alla precedente)
  - Devo fare il forward del più aggiornato: quello che sta in EX/MEM (ForwardX $\leftarrow 10$)

Esercizio: cosa succede con questi due blocchi di istruzioni?

add \$1 \$2 \$2	add \$1 \$1 \$1
add \$1 \$3 \$3	add \$1 \$1 \$1
add \$1 \$1 \$1	add \$1 \$1 \$1

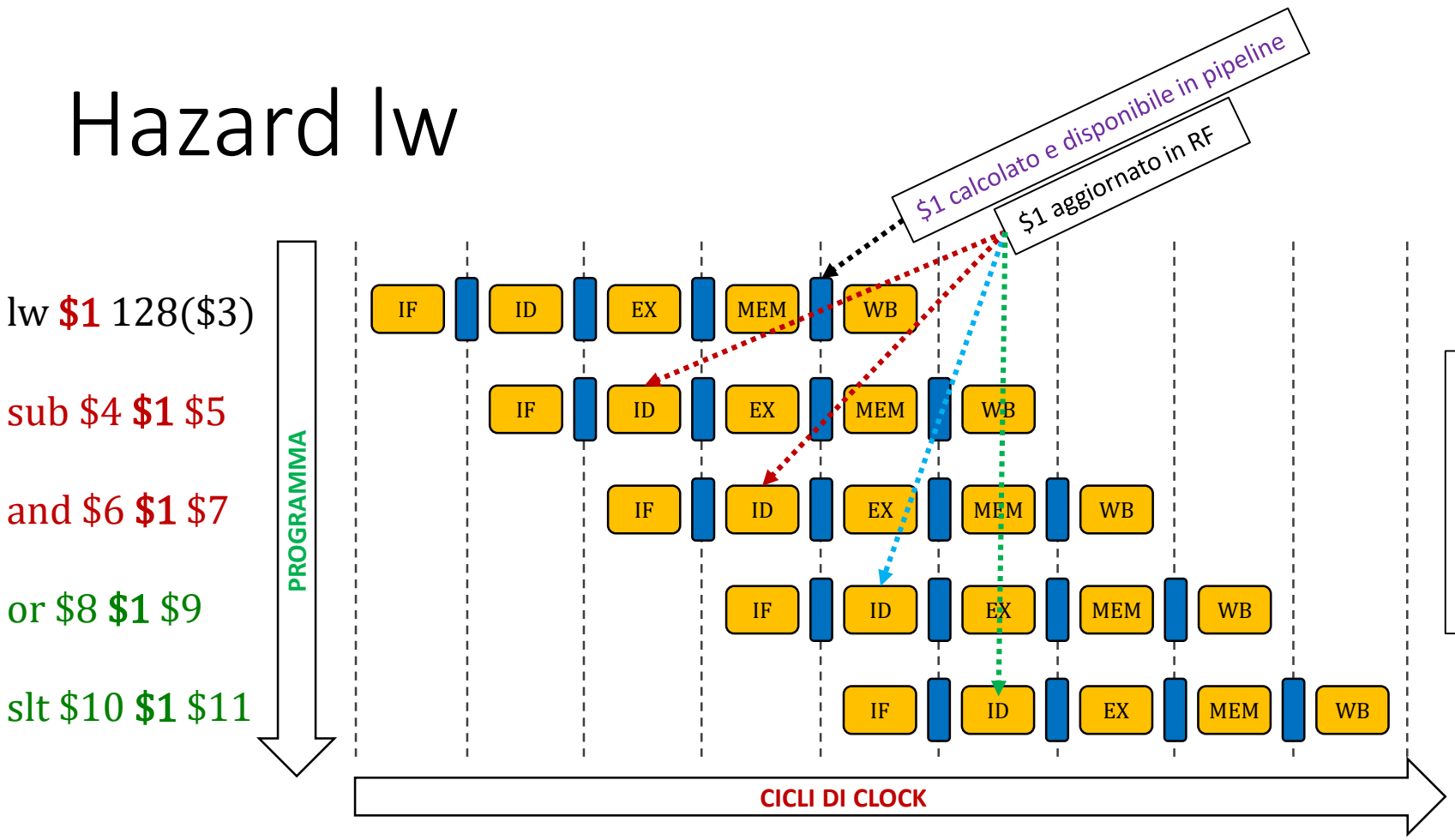
Hazard lw

- Negli hazard considerati fino ad ora il dato da anticipare era sempre **prodotto** nella fase di EX (e anticipato prelevandolo o da EX/MEM o da MEM/WB)
- C'è un secondo caso problematico, un hazard su un dato che viene prodotto nella fase di MEM (per noi l'unica istruzione che produce un dato da scrivere in un registro nella fase MEM è la load word)

```
lw $1 128($3)
sub $4 $1 $5
and $6 $1 $7
or $8 $1 $9
slt $10 $1 $11
```

- Consideriamo lo stesso esempio di prima dove nella prima istruzione ho una lw
- Il registro coinvolto negli hazard è sempre **$1**

Hazard lw



Stesso profilo di prima,

hazard lw solo tra:

- Una istruzione e la precedente
- Una istruzione e la precedente della precedente

- **Ma le opportunità di forwarding sono ridotte! Il dato è prodotto più tardi!**
- Quando sub entra in EX, in ID/EX *(alla sua sinistra)* c'è il valore obsoleto di $2, **il valore corretto non esiste ancora! Non possiamo fare forwarding!**
- Quando and entra in EX, in ID/EX *(alla sua sinistra)* c'è il valore obsoleto di $2, **ma il valore corretto è in pipeline dentro il registro MEM/WB** (letto da memoria due cicli di clock prima, *nel secondo registro pipeline alla sua destra!*)

Hazard lw

WB lw **$1** 128($3)

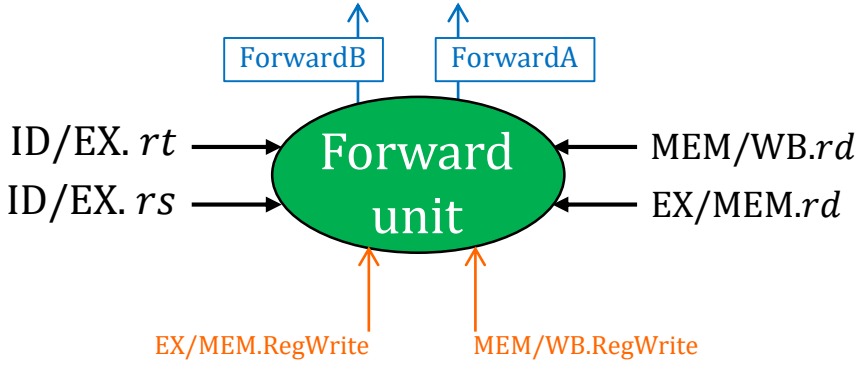
MEM sub $4 **$1** $5

EX and $6 **$1** $7

ID or $8 $1 $9

IF slt $10 $1 $11

- L'hazard sulla seconda istruzione che segue la lw può essere risolto tramite la logica di forwarding
- Quando la and entra in fase di EX la lw sta svolgendo WB, abbiamo che:
  - ID/EX.rs=1 (per la and)
  - MEM/WB.rd=1 (rt della lw)
  - MEM/WB.RegWrite=1 (la lw deve scrivere nel RF)
- **Ci pensa già la nostra Forward unit!**



if ID/EX.rs == EX/MEM.rd && EX/MEM.RegWrite && EX/MEM.rd != 0
ForwardA ← 10
if ID/EX. rt == EX/MEM.rd && EX/MEM.RegWrite && EX/MEM.rd != 0
ForwardB ← 10

if ID/EX. rs == MEM/WB.rd && MEM/WB.RegWrite && MEM/WB.rd != 0
ForwardA ← 01
if ID/EX. rt == MEM/WB.rd && MEM/WB.RegWrite && MEM/WB.rd != 0
ForwardB ← 01

Hazard lw

WB lw **$1** 128($3)

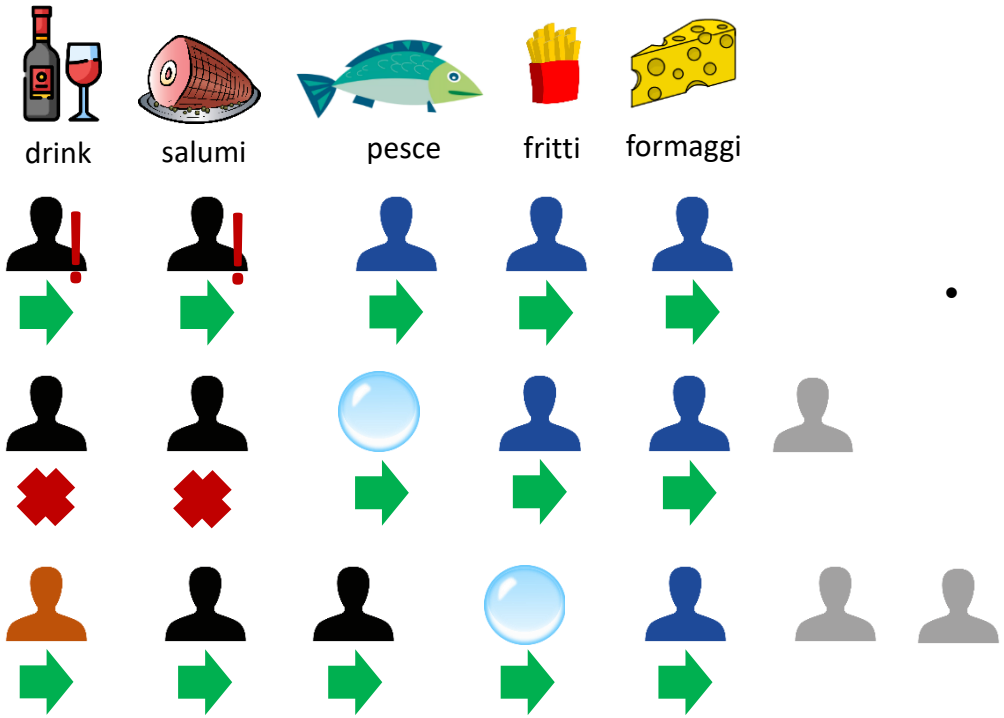
MEM sub $4 **$1** $5

Risolto con FW → EX and $6 **$1** $7

ID or $8 **$1** $9

IF slt $10 **$1** $11

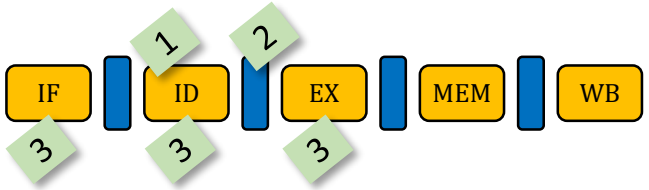
- L'hazard sulla prima istruzione che segue la lw non può essere risolto tramite la logica di forwarding, non possiamo anticipare l'arrivo di un dato se questo ancora non esiste
- Unica opzione: **mandare in stallo la pipeline**



- Effetto simile a quello di una nop, **ma viene fatto dall'hardware durante l'esecuzione!**

Hazard lw

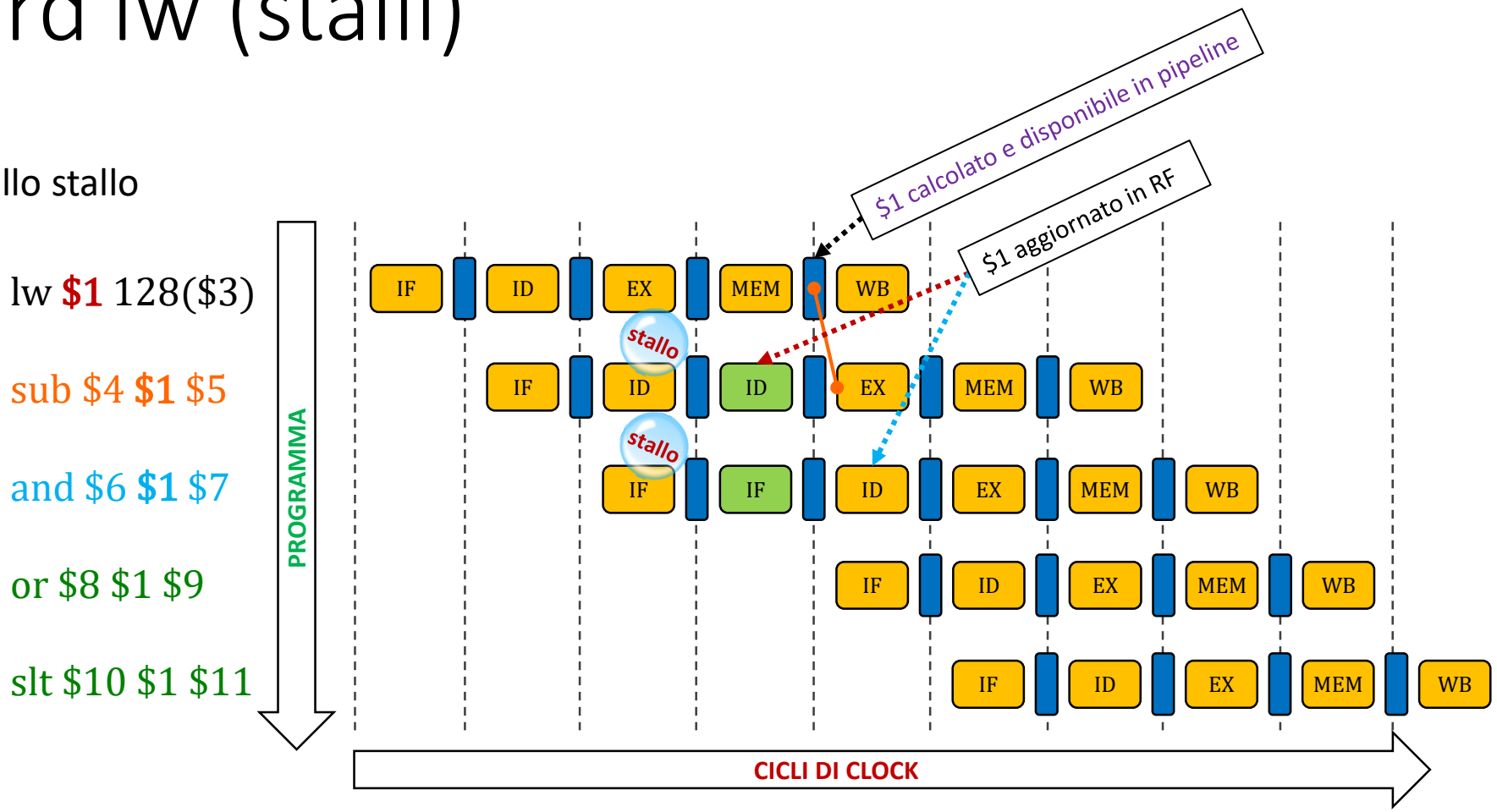
- Per implementare la logica di stallo dobbiamo seguire questi step:
    1. **Rilevare che vi sarà l'hazard**: va fatto il prima possibile, ovvero nella fase ID
    2. Nello stesso ciclo di clock in cui viene rilevato l'hazard, **scrivere nel registro ID/EX dei segnali di controllo di una nop** anziché quelli dell'istruzione correntemente in ID
    3. Nel ciclo di clock successivo:
        - **IF e ID vanno ripetute** come al ciclo precedente
        - EX svolge una nop
        - MEM e WB continuano senza alterazioni



- **Dopo uno ciclo di stallo gli hazard diventano risolvibili con forwarding e dipendenza interna**
- **Dobbiamo modificare il datapath**

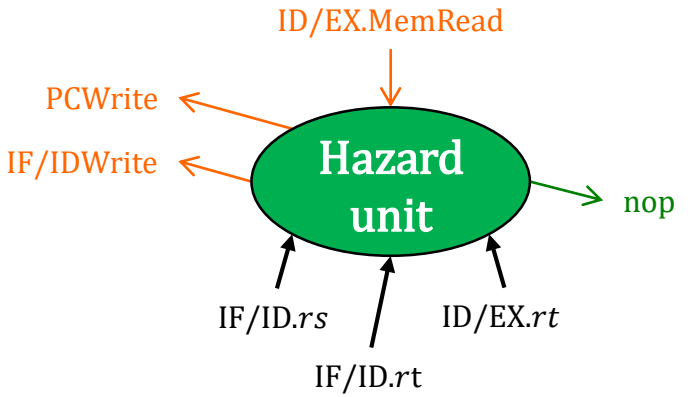
Hazard lw (stalli)

- Effetti dello stallo



- Per il data hazard della sub ora scatta la **logica di forwarding**, che lo risolve
- Per la and non ci sono problemi, dipendenza interna alla pipeline risolta automaticamente

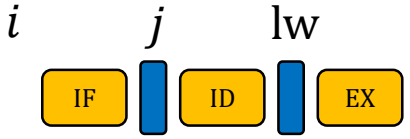
Hazard lw (stalli)



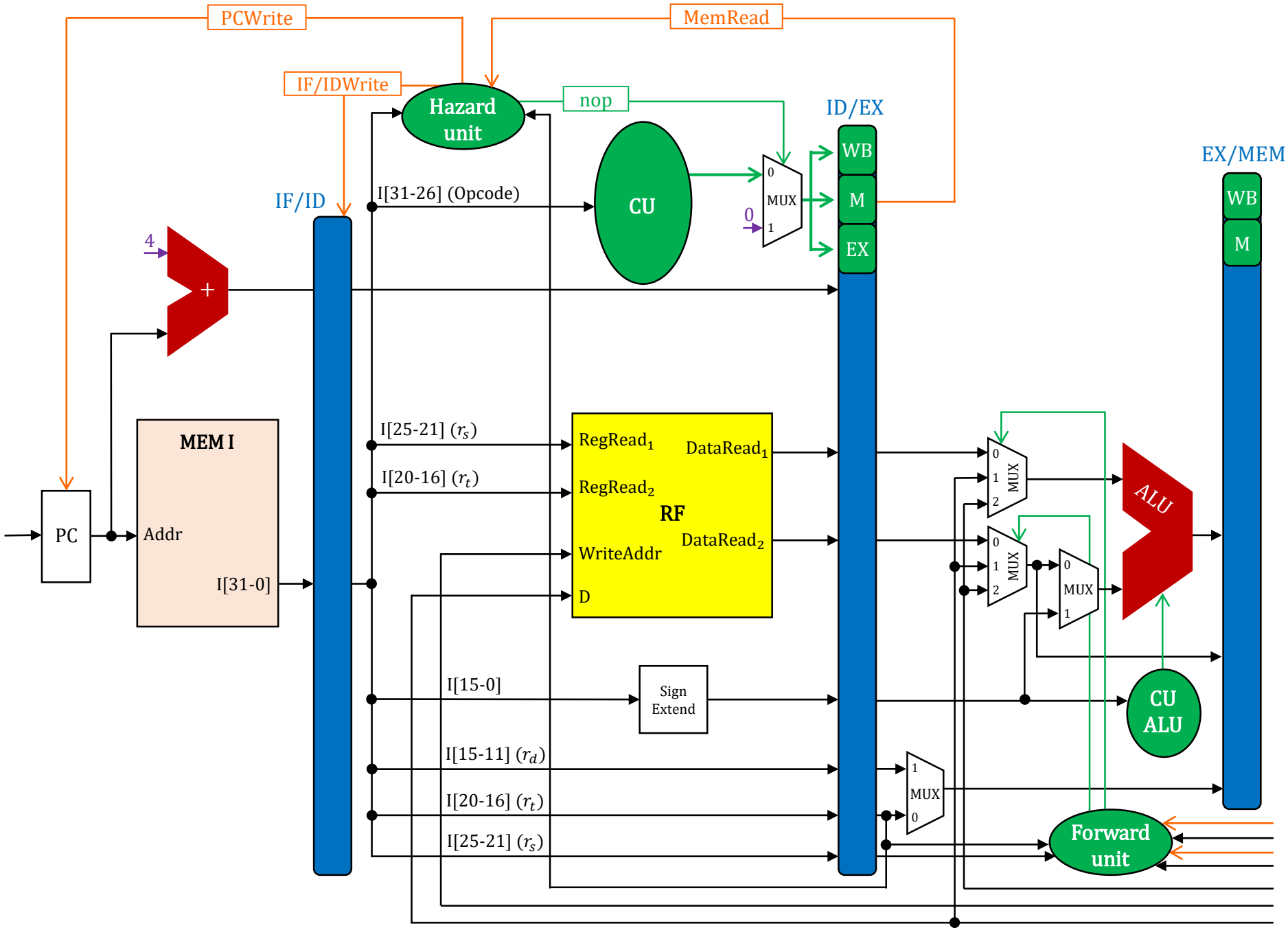
If ID/EX.MemRead && (ID/EX.rt ==IF/ID.rs || ID/EX.rt ==IF/ID.rt)

Stallo di IF e ID

- La Hazard Unit lavora nella fase di ID, assumiamo di essere all'inizio di quel ciclo di clock (possiamo anche pensare che questi step avvengano immediatamente prima che si effettui la fase ID)
- Se la condizione dell' **If** è vera allora:
  - Nel ciclo prima era stata codificata una lw, che in questo ciclo fa la EX
  - L'istruzione successiva j, che è ora in ID, opera sul valore letto dalla lw (con rs o con rt)



- Stallo di IF:
  - PCWrite $\leftarrow 0$ impedisco, per questo ciclo di clock, la scrittura di PC+4 nel PC, nel prossimo ciclo di clock la CPU ripeterà IF di i che aveva svolto in questo ciclo
  - IF/IDWrite $\leftarrow 0$ impedisco, per questo ciclo di clock, l'aggiornamento di IF/ID con l'istruzione i, nel prossimo ciclo di clock la CPU ripeterà quindi ID di j che aveva svolto in questo ciclo
  - nop $\leftarrow 1$ nel registro ID/EX vengono scritti tutti segnali di controllo pari a 0 anziché i segnali di controllo della j, nel prossimo ciclo di clock la CPU fa EX di una nop (imporre tutti i segnali di controllo a 0 è il modo più semplice che usiamo noi per implementare la nop)



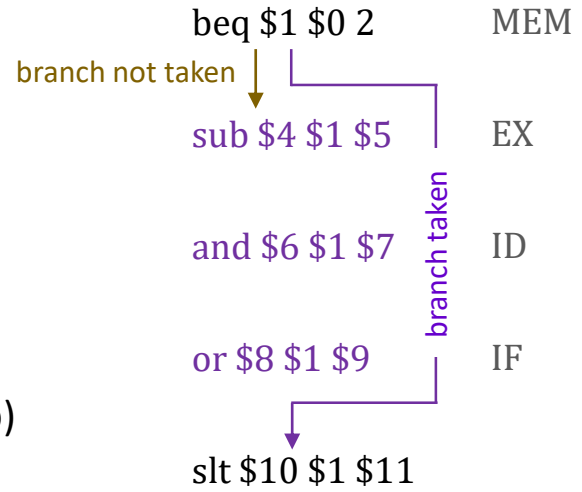
nop è un segnale di selezione che controlla un MUX tra:

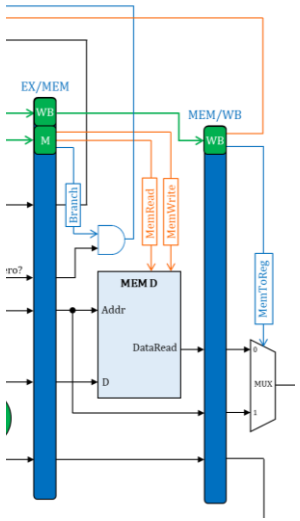
- Segnali di controllo dell'istruzione decodificata
- Tutti 0

Control hazards

Criticità di controllo

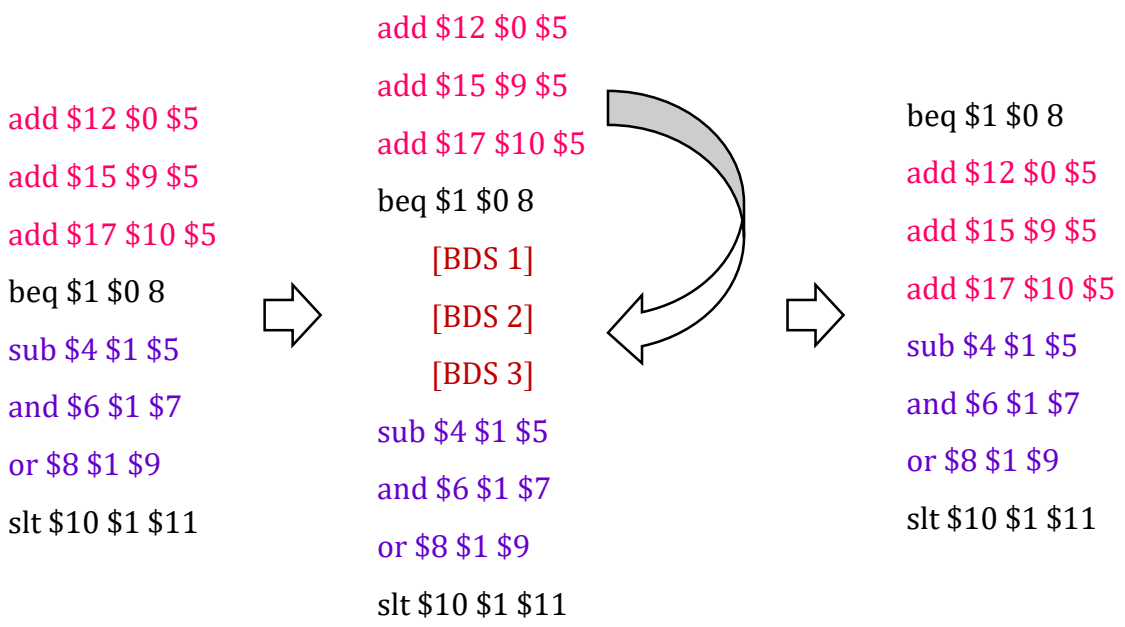
- Criticità strutturali (structural hazards): **risolte**
- Criticità di dato (data hazards)
  - Caso 1 (dato necessario prodotto in EX): **risolte** con forwarding
  - Caso 2 (dato necessario prodotto in MEM): **risolte** con stallo e forwarding
- **Criticità di controllo** (control hazards)
  - Un'istruzione j determina un salto nel flusso di controllo del programma (branch oppure jump)
  - Un'istruzione i, successiva a j, non deve essere eseguita a causa del salto
  - Quando j sovrascrive il PC con l'indirizzo di salto, i è già entrata in pipeline e la CPU la sta elaborando
- Consideriamo lo stesso esempio di prima dove nella prima istruzione ho una beq
- Nel caso in cui il salto venga fatto (branch taken) le istruzioni sub, and e or non devono essere eseguite
- **Aspetto chiave:** quando la beq effettua la sovrascrittura del PC con il BTA?
- Durante la sua fase di MEM! **Ma mentre questo avviene**:
  - sub svolge la sua fase di EX
  - and svolge la sua fase di ID
  - or svolge la sua fase di IF





Criticità di controllo (gestione SW)

- Come gestiamo il problema?
- Come per i data hazard, abbiamo la possibilità di intervenire via software facendo fare del lavoro extra al compilatore
- **Delayed branch slot:** ogni branch avviene di fatto in ritardo rispetto a quando entra nella CPU
- Vedo questo ritardo come slot di esecuzione su cui riorganizzare il codice (analogamente a quando abbiamo fatto per la gestione software dei data hazard)



- Tecnica più semplice: riempire i BDS con istruzioni prese da **prima del salto** (*from before*)
- Se non è possibile:
  - Pescare dalla zona di salto (scommettere su branch taken, *from target*)
  - Pescare dalla zona tra la beq e salto (scommettere su branch not taken, *from fall-through*)

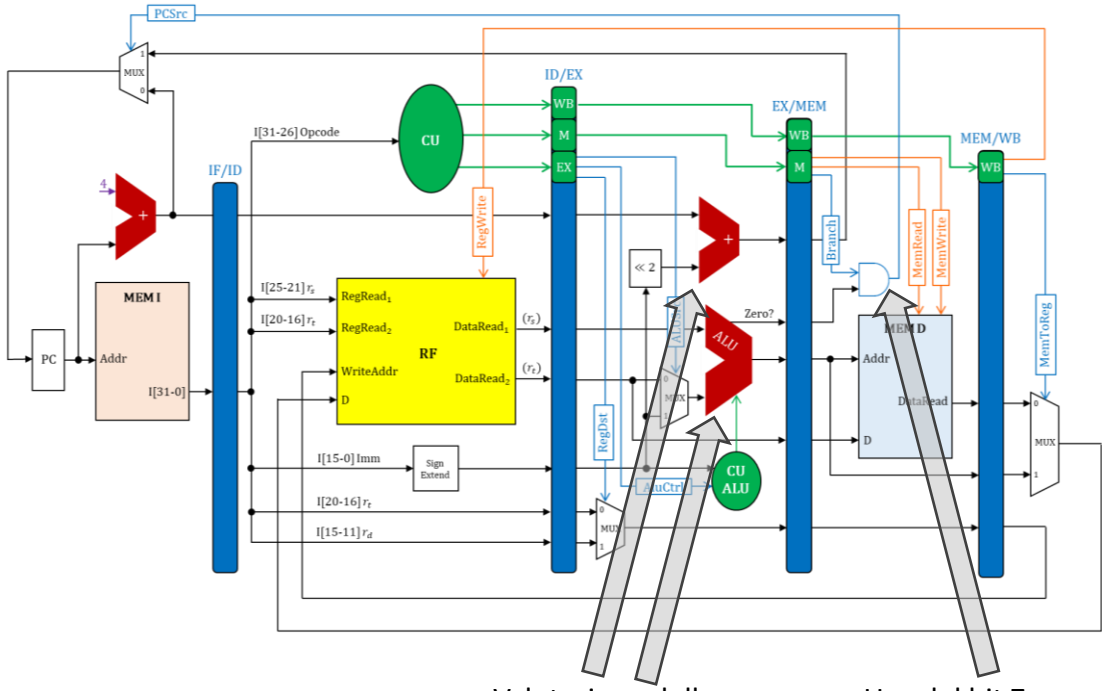
Criticità di controllo (gestione HW)

- **Primo semplice approccio**: lasciare che il problema accada e porvi rimedio via hardware
- L'esecuzione procede normalmente
  - **Branch not taken**: tutto fila liscio, non ci sono hazard
  - **Branch taken**: ho in pipeline 3 istruzioni che non devono esserci, vanno cancellate
- Come si cancella una istruzione che è già entrata in pipeline?
- Si scrive 0 in ogni suo segnale di controllo nel registro pipeline alla sua destra, nel procedere in avanti «si trasforma» in una nop: l'operazione si chiama **flush del registro pipeline**
- Nel nostro caso dobbiamo fare flush di: IF/ID, ID/EX, EX/MEM
- La CPU scommette sempre su branch not taken, quando perde paga 3 cicli di clock (uno per ogni flush)!

Questo costo sembra un po' alto, non possiamo fare nulla per abbassarlo?

Anticipazione dei salti

- Paghiamo un costo di 3 perché il salto viene fatto tardi, nella fase di MEM
- È una conseguenza dell'architettura del nostro datapath: la valutazione della condizione e il calcolo del BTA vengono svolte nella fase EX
- Quindi il primo ciclo di clock utile per usare queste informazioni (e, eventualmente, svolgere il salto) è quello successivo: fase di MEM



Valutazione della condizione e calcolo BTA

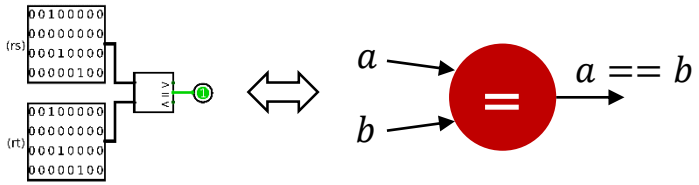
Uso del bit Zero per sovrascrittura del PC con BTA

- Dagli hazard lw abbiamo imparato una regola fondamentale: **per prevenire gli stalli bisogna rilevare la criticità il prima possibile e cioè nella fase ID**
- Come posiamo rilevare questo tipo di criticità nella fase di ID?
- **Anticipazione del salto**: modifico il datapath in modo che sia il calcolo del BTA che il controllo della condizione vengano svolti nella fase ID
- Mi accorgo subito se il salto si fa o non si fa
- In caso il salto si faccia bisogna fare flush solo di IF, quindi **perdo un solo ciclo di clock anziché tre**

Anticipazione dei salti

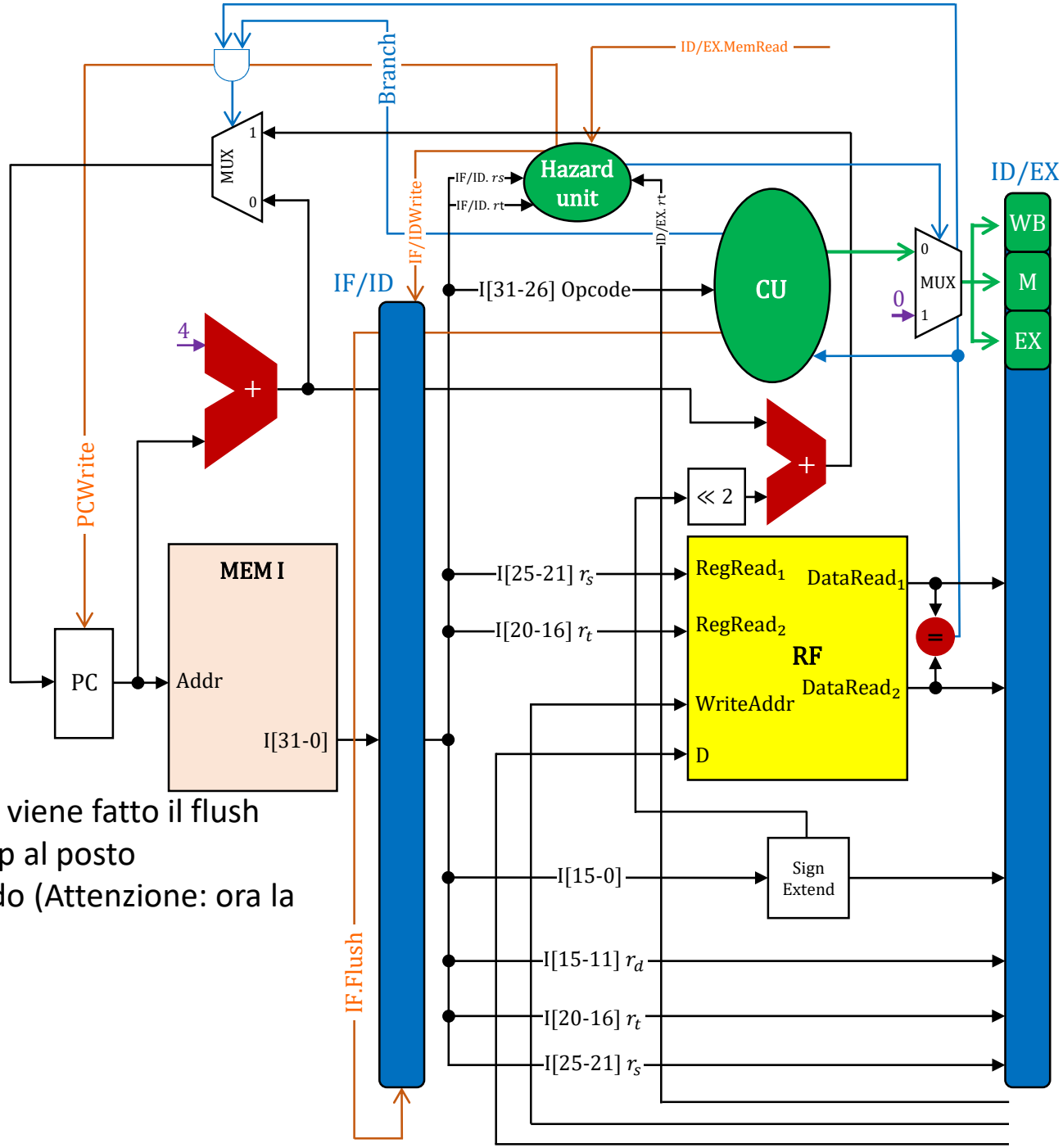
- Il sommatore per il calcolo del BTA ritorna nello stadio ID (come nella CPU a ciclo multiplo)

- Va aggiunto un comparatore dedicato per la verifica della condizione di uguaglianza



- Il comparatore è collegato direttamente alle uscite del RF per confrontare (rs) e (rt)

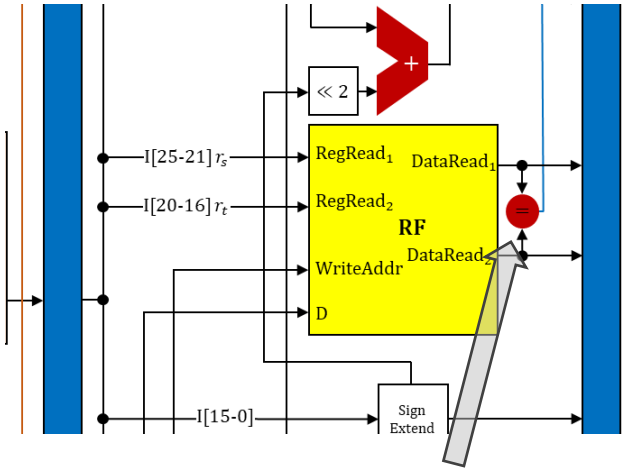
- La control unit viene aggiornata: in caso di salto effettuato viene fatto il flush (comando IF.Flush) del registro IF/ID, viene cioè scritta nop al posto dell'istruzione successiva alla beq che la CPU sta prelevando (Attenzione: ora la nop va implementata come parte dell'ISA)
- La CPU scommette sempre su **branch not taken**
  - Se indovina l'esecuzione prosegue a pieno regime
  - Se sbaglia paga con un flush (un ciclo di clock)



Anticipazione dei salti

- L'anticipazione dei salti è una modifica del datapath più forte rispetto a quanto fatto nella CPU a ciclo multiplo: oltre al calcolo del BTA spostiamo nello stadio di ID anche la valutazione della condizione (prima era anticipazione del BTA, ora dei salti)
- Questa modifica ha due effetti importanti (e nascosti):
  - I Branch Delay Slot passano da 3 a 1
  - Serve una logica di **data hazard detection e forwarding** anche nello stadio ID!

```
add $12 $0 $5
add $15 $9 $5
add $17 $10 $5
beq $1 $0 8
    [BDS]
sub $4 $1 $5
and $6 $1 $7
or $8 $1 $9
slt $10 $1 $11
```

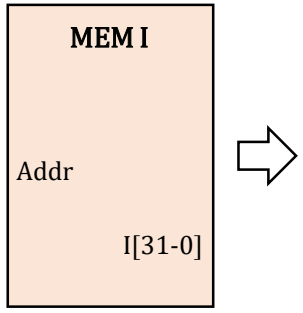


Hazard detection e forwarding necessari

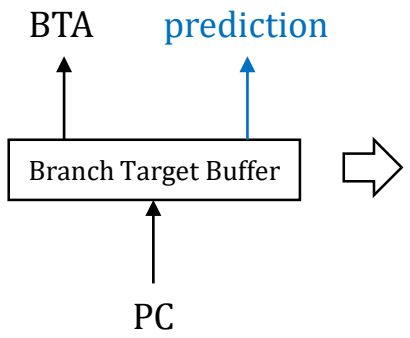
- Per non appesantire troppo lo schema del datapath non raffigureremo esplicitamente la logica di forwarding aggiuntiva
- Ricordiamoci che è comunque presente e che lavora in modo totalmente analogo alla forward unit che abbiamo nello stadio di EX

Dynamic branch prediction

- L'architettura attuale «scommette» sempre su branch not taken, è possibile fare meglio?
- **Dynamic branch prediction**: la CPU scommette (branch prediction) su un esito diverso (dynamic) caso per caso
- Come fare la predizione? Uso di un **Branch Target Buffer**: memoria associativa che contiene uno storico delle branch più recenti
- Ogni branch è indicizzata con la parte bassa del proprio indirizzo
- Per ognuna, il buffer memorizza il suo target address e un bit che codifica l'esito del salto più recente (nell'ultima volta che quella istruzione è stata eseguita): 0 branch not taken, 1 branch taken (schema di predizione a 1 bit)



Indirizzo	Istruzione
$0x00400024$	beq \$1 \$2 28
⋮	
$0x004000A8$	beq \$7 \$5 44
⋮	
$0x004000B8$	beq \$3 \$0 36
⋮	
$0x00400108$	beq \$11 \$0 24
⋮	
$0x00400114$	beq \$13 \$15 8



Indirizzo istruzione (parte bassa)	Target address	Bit di salto
024	$0x00440324$	1
0A8	$0x00440364$	0
108	$0x0044037C$	0
114	$0x00441744$	1
0B8	$0x00441570$	1

- **Il bit di salto è la predizione**: la CPU scommette che l'esito del salto sarà lo stesso dell'ultima volta che quella branch è stata eseguita
- Si fa guidare da una sorta di principio di continuità

Dynamic branch prediction

- Questo principio di continuità è ben riposto? È una buona euristica con cui provare a indovinare i salti?
- **Argomento a favore**: un programma spesso ha dei loop (while, for, etc.) nei loop una stessa branch viene eseguita ripetutamente

```
        addi $1 $0 1          # $1 ← 1
        addi $2 $0 15         # $2 ← 15
loop:   beq $2 $0 break       # while($2≠0)
        sub $2 $2 $1
        …
        # fai qualcosa
        …
        j loop
break:  …
        …
```

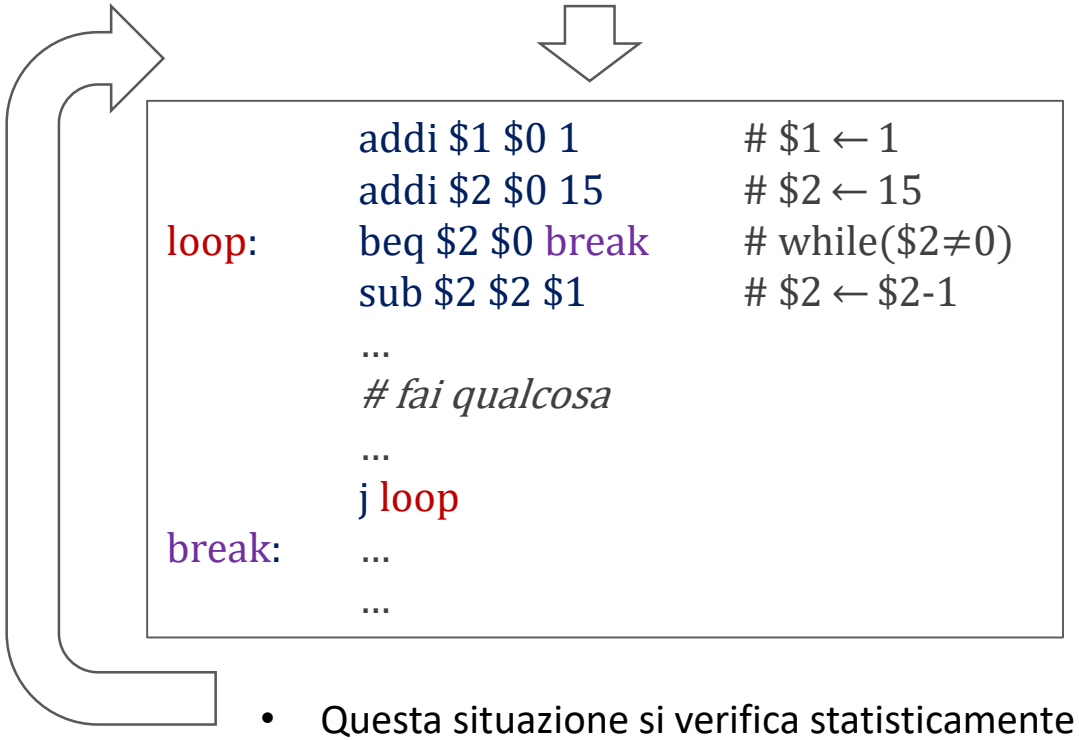
- Quante volte viene eseguita la branch?
  - 16
- Quante volte indoviniamo il salto con dynamic branch prediction?
  1. Scommetto not taken (initial guess non informato), indovinato!
  2. Scommetto not taken, indovinato!
  3. Scommetto not taken, indovinato!

  ⋮

  16. Scommetto not taken, sbagliato!

Dynamic branch prediction

- **Argomento a sfavore**: cosa c'è di ancora più frequente di un loop? **Il doverlo ripetere più volte!**
- Esempi: loop annidati, oppure il loop parte di una funzione viene chiamata spesso, etc.
- Cosa succede se il ciclo dell'esempio di prima viene ripetuto una seconda volta dopo poco tempo?



```
        addi $1 $0 1          # $1 ← 1
        addi $2 $0 15         # $2 ← 15
loop:   beq $2 $0 break       # while($2≠0)
        sub $2 $2 $1          # $2 ← $2-1
        …
        # fai qualcosa
        …
        j loop
break:  …
        …
```

- Quante volte viene eseguita la branch?
  - 16
- Quante volte indoviniamo il salto con dynamic branch prediction?
  1. Scommetto taken, sbagliato!
  2. Scommetto not taken, indovinato!
  3. Scommetto not taken, indovinato!

  ⋮

  16. Scommetto not taken, sbagliato!

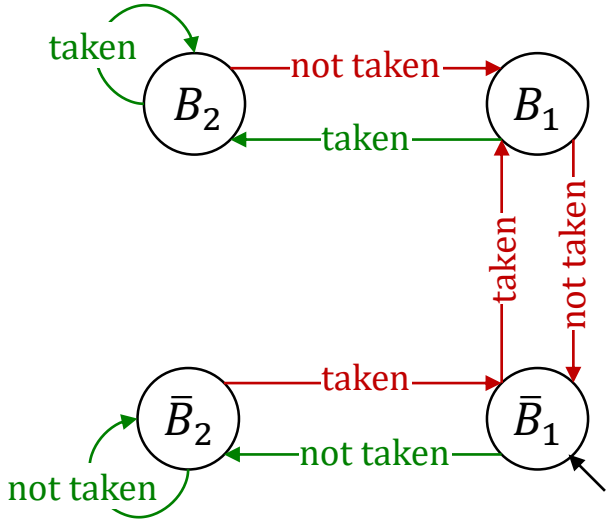
- Questa situazione si verifica statisticamente così tanto spesso che
  - Di fatto ogni ciclo costa 2
  - Conviene progettare un sistema di predizione più sofisticato per questo caso specifico e cercare di ridurre tale costo

Dynamic branch prediction

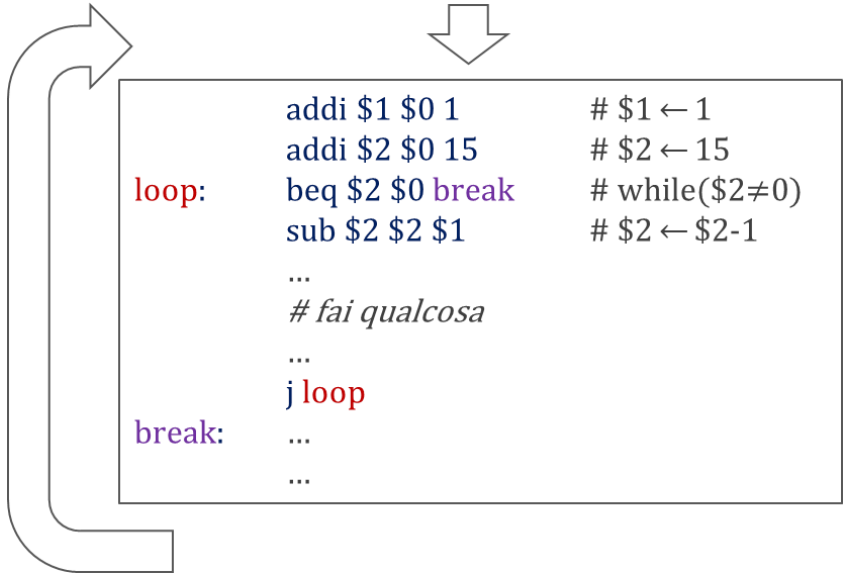
- Il limite principale del meccanismo che stiamo usando è dato dal bit di predizione: ne stiamo usando uno solo e quindi possiamo tenere traccia solo dell'esito di salto più recente, l'ultimo osservato in CPU
- Estendiamo a 2 bit: **2-bit prediction**
- I due bit di salto ora rappresentano lo stato corrente di una FMS (Moore), **una per ogni branch (riga della tabella)**
- **L'uscita** è il bit di salto usato per la predizione, il modello di transizione
- **Modello di transizione**: cambio la predizione solo dopo due errori consecutivi (con 1 bit era dopo ogni errore)

Indirizzo istruzione (parte bassa)	Target address	Stato
024	$0x00440324$	01
$0A8$	$0x00440364$	10
108	$0x0044037C$	10
114	$0x00441744$	11
$0B8$	$0x00441570$	00

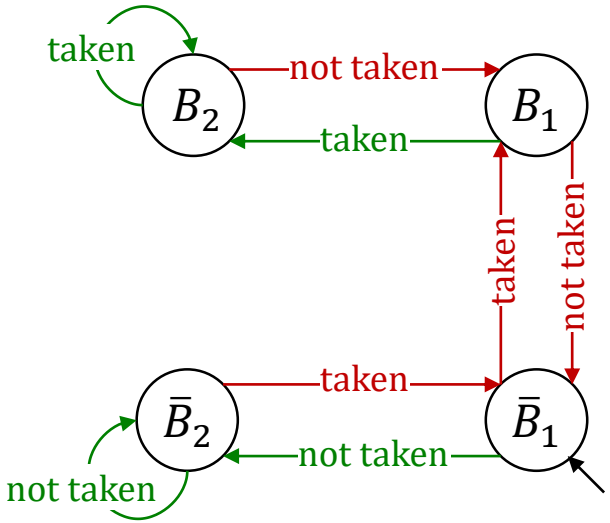
Nome dello stato	Codifica	Uscita (prediction)
$\bar{B}_1$	00	0
$\bar{B}_2$	01	0
B_1	10	1
B_2	11	1



Dynamic branch prediction



```
        addi $1 $0 1          # $1 ← 1
        addi $2 $0 15         # $2 ← 15
loop:   beq $2 $0 break       # while($2≠0)
        sub $2 $2 $1          # $2 ← $2-1
        ...
        # fai qualcosa
        ...
        j loop
break:  ...
        ...
```



- Prima esecuzione
  1. Scommetto not taken (sono in $\bar{B}_1$, stato iniziale), indovinato! (vado in $\bar{B}_2$)
  2. Scommetto not taken (sono in $\bar{B}_2$), indovinato! (resto in $\bar{B}_2$)
  3. Scommetto not taken (sono in $\bar{B}_2$), indovinato! (resto in $\bar{B}_2$)

  ⋮

  16. Scommetto not taken (sono in $\bar{B}_2$), sbagliato! (vado in $\bar{B}_1$)

1 flush su 16 esecuzioni

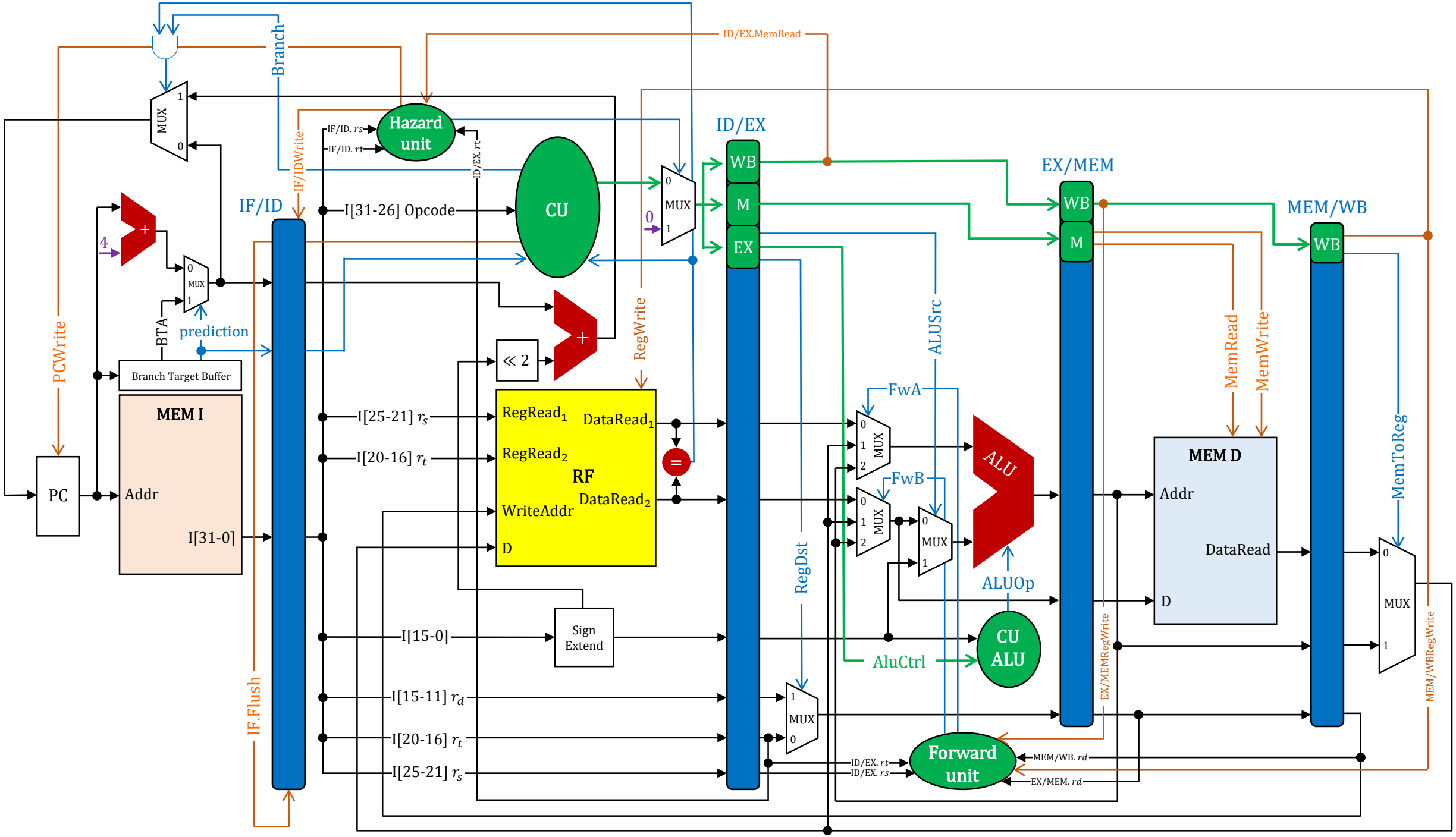
- Seconda esecuzione
  1. Scommetto not taken (sono in $\bar{B}_1$), indovinato! (vado in $\bar{B}_2$)
  2. Scommetto not taken (sono in $\bar{B}_2$), indovinato! (resto in $\bar{B}_2$)
  3. Scommetto not taken (sono in $\bar{B}_2$), indovinato! (resto in $\bar{B}_2$)

  ⋮

  16. Scommetto not taken (sono in $\bar{B}_2$), sbagliato! (vado in $\bar{B}_1$)

1 flush su 16 esecuzioni

- Dalla seconda esecuzione in poi si sbaglia una volta sola anziché due!


Branch
ID/EX.MemRead
MUX
1
0
Hazard unit
IF/ID. rs
IF/ID. rt
IF/IDWrite
ID/EX. rt
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
CU
I[31-26] Opcode
0
MUX
1
IF/ID
4
+
BTA
prediction
PCWrite
Branch Target Buffer
MEM I
PC
Addr
I[31-0]
≪ 2
+
RegWrite
RegRead1
DataRead1
RegRead2
RF
DataRead2
WriteAddr
D
=
I[25-21] rs
I[20-16] rt
I[15-0]
Sign Extend
I[15-11] rd
I[20-16] rt
I[25-21] rs
IF.Flush
RegDst
ALUSrc
FwA
FwB
ALU
ALUOp
CU ALU
AluCtrl
MemRead
MemWrite
MemToReg
MEM D
Addr
DataRead
D
EX/MEMRegWrite
MEM/WBRegWrite
Forward unit
ID/EX. rt
ID/EX. rs
MEM/WB. rd
EX/MEM. rd

Architettura degli Elaboratori II

Corso di Laurea Triennale in Informatica

Università degli Studi di Milano

Dipartimento di Informatica “Giovanni Degli Antoni”

Edizione 2 (Cognomi H-Z), A.A. 2021-2022, Nicola.Basilico@unimi.it

Eccezioni

Eccezioni

- Abbiamo visto come la gestione dei control hazard sia particolarmente critica in quanto implica sia rischi sulla correttezza che potenziali costi di performance
- Le branch e le jump causano deviazioni **strutturate** del flusso di controllo: sono appositamente progettate dal programmatore affinché il programma svolga il compito desiderato, sono **previste** e **fondamentali** alle funzioni svolte
- Esistono deviazioni del flusso causate da eventi imprevisti: **eccezioni**
- Terminologia MIPS:
  - **Eccezione**: una deviazione imprevista del flusso di controllo la cui causa può genericamente verificarsi internamente o esternamente alla CPU
  - **Interrupt**: un'eccezione la cui causa si verifica esternamente alla CPU

Evento	Origine	Terminologia MIPS
Istruzione non riconosciuta (es. opcode non valido)	Interna	Eccezione
Overflow	Interna	Eccezione
Syscall (il programma invoca un servizio del sistema operativo)	Interna	Eccezione
Richiesta da una periferica I/O	Esterna	Interrupt
Malfunzionamento hardware	Interna/Esterna	Eccezione/Interrupt

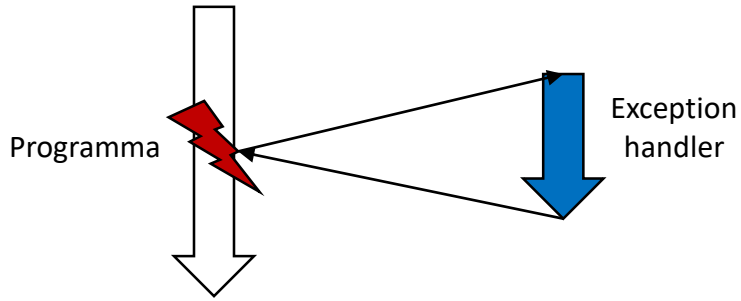
Eccezioni

- Trattiamo queste due possibili eccezioni che possono verificarsi nella nostra architettura:
  1. **Istruzione non riconosciuta**: è stato svolto il fetch di una istruzione che la CPU non riesce a decodificare nella fase di ID
  2. **Overflow aritmetico**: un'istruzione aritmetica nella fase di EX ha generato un overflow
- Non abbiamo controllo sugli eventi che generano le eccezioni, ma l'hardware può riconoscere quando avvengono e svolgere delle operazioni speciali per gestirle
- **Esempi di riconoscimento**:
  - Nella fase di ID la CU legge un opcode non valido oppure l'istruzione è in formato R (opcode 000000) ma il campo funct non è valido → **istruzione non riconosciuta**
  - Nella fase di EX la ALU svolge un'operazione e il bit di overflow si attiva → **overflow aritmetico**

Gestione di un'eccezione

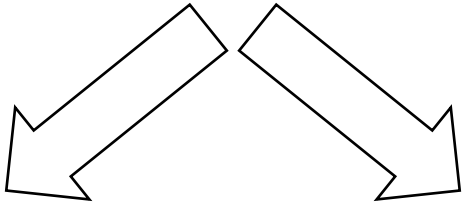
- Nella nostra architettura semplificata per poter gestire un'eccezione all'atto della sua rilevazione vanno identificate due informazioni chiave:
  - **La causa dell'eccezione** (istruzione non riconosciuta o overflow?)
  - **L'indirizzo della *offending instruction*** e cioè di quella istruzione che ha causato l'eccezione (questa informazione è definita in modo diverso e più generale quando si considerano eccezioni aggiuntive e interrupt)

- Risposta all'eccezione: trasferimento del controllo verso un blocco di istruzioni che gestisce l'evento, è un programma detto **exception handler**, tipicamente fa parte del sistema operativo
- Che cosa fa l'exception handler? Gestisce la situazione provando a ripristinare l'esecuzione del programma, ad esempio:
  - Stampare un messaggio di errore
  - Terminare l'esecuzione del programma
  - Ripristinare l'esecuzione del programma



Gestione di un'eccezione

1. L'eccezione viene rilevata dall'hardware
2. L'indirizzo dell'istruzione successiva alla *offending instruction* viene scritto in un registro speciale chiamato **EPC** (Exception Program Counter), nel ciclo di clock in cui avviene l'eccezione questo indirizzo è $PC + 4$ calcolato al ciclo precedente e che, nel ciclo corrente, è stato scritto nel registro IF/ID



Gestione basata sul registro di stato, in MIPS chiamato **registro causa**

Cause

Gestione basata su interrupt vettorizzati: **interrupt vector table**

Gestione di un’eccezione (registro causa)

3. dopo averla rilevata, la CPU scrive un codice dentro un registro speciale, detto **registro causa** (Il codice rappresenta la causa dell’eccezione)

Eccezione	Codice
Istruzione non riconosciuta	10
Overflow aritmetico	12

Spesso chiamata Reserved Instruction: questa eccezione può essere usata per estendere l’ISA con nuove istruzioni non supportate in modo nativo o per segnalare istruzioni privilegiate che operano in modalità kernel (con gli stessi privilegi del S.O.)

- Viene fatto un salto non condizionato all’indirizzo $0x80000180$, dove risiede la prima istruzione dell’exception handler
- L’handler, come prima cosa, legge il codice dell’eccezione dal registro causa e agisce di conseguenza (come uno switch)

Gestione di un'eccezione (IVT)

3. L'hardware effettua un salto ad un indirizzo diverso a seconda della causa dell'eccezione

Eccezione	Indirizzo di salto
Istruzione non riconosciuta	$0x800000$
Overflow aritmetico	$0x800180$

- La corrispondenza tra eccezione e indirizzo a cui saltare è memorizzata dentro una struttura dati chiamata **Interrupt Vector Table**: è una tabella dove, data una causa di eccezione, si ottiene l'indirizzo a cui saltare per cedere il controllo all'handler specifico di quella eccezione
- MIPS non usa questo metodo, ma il metodo basato sul registro causa
- Tipicamente gli indirizzi di salto sono equispaziati (ad es. 32 byte, 8 istruzioni entro cui l'handler può salvare il codice causa e eventualmente fare un altro salto)
- Spesso non viene riportato un indirizzo ma uno pseudo-indirizzo: ad es. nelle architetture x86 in real mode si usano due campi, un Code Segment (un base address da estendere) a cui viene sommato un Instruction Pointer (un offset)

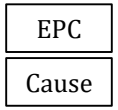
Gestione di un’eccezione (conclusione)

- Una volta concluse le operazioni l’handler può:
- Terminare il programma: ad esempio in caso di overflow si è verificato un errore critico e non c’è modo di continuare senza incorrere in errori (cfr. il problema dell’overflow, Architettura 1)
- Ripristinare l’esecuzione: ad esempio se usiamo l’eccezione «istruzione non riconosciuta» per estendere l’ISA vogliamo che dopo la gestione dell’eccezione l’esecuzione del programma prosegua da dove era stata interrotta
- In questo secondo caso l’exception handler farà uso dell’informazione contenuta in EPC, nello specifico un salto all’indirizzo contenuto in tale registro (l’indirizzo dell’istruzione successiva a quella che ha causato l’eccezione e che ora è stata gestita)

Datapath per le eccezioni

- Quali modifiche dobbiamo fare al nostro datapath per poter implementare la gestione delle due eccezioni che abbiamo considerato?

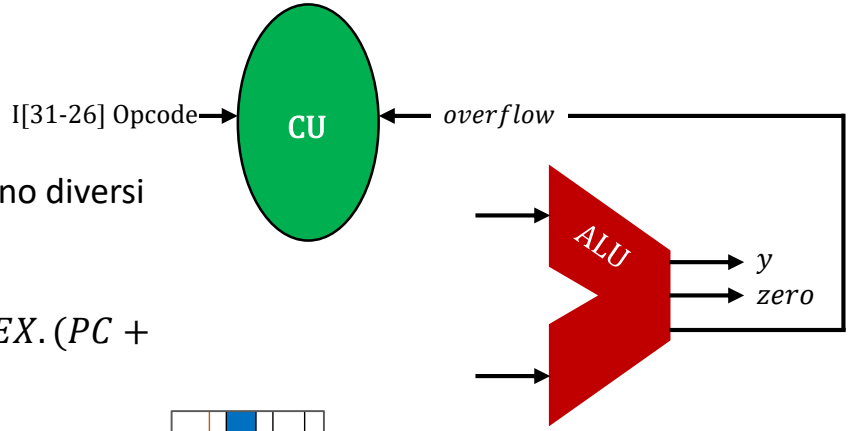
1. Aggiungiamo i registri EPC e Cause



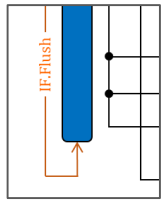
2. Estendiamo la CU in modo che:

   a) Se durante la fase di ID l'opcode (o il campo funct in caso di formato R) sono diversi da quelli conosciuti scriva $IF/ID.(PC+4)$ in EPC e il codice 10 in Cause

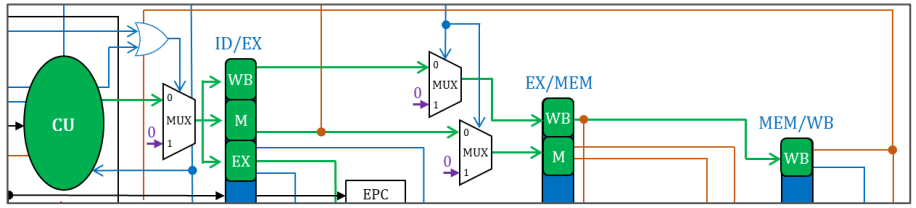
   b) Se durante la fase di EX la ALU attiva in uscita il bit di overflow scriva $ID/EX.(PC+4)$ in EPC e il codice 12 in Cause



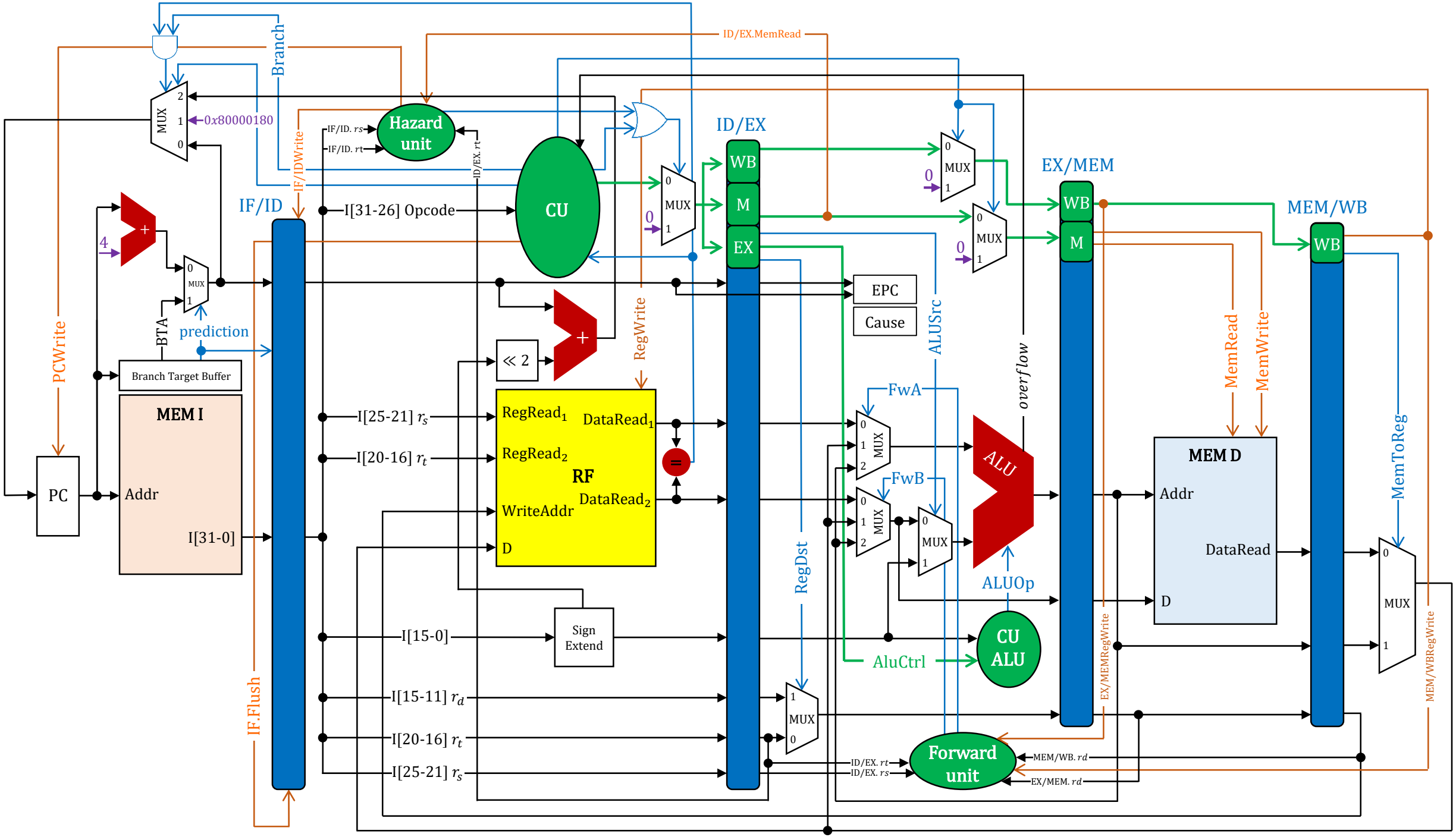
3. In caso si sia verificato a), la CU deve scartare la offending instruction dalla fase di ID e la successiva che in quello stesso ciclo di clock ha completato la fase IF



4. In caso si sia verificato b), la CU deve scartare la offending instruction dalla fase di EX e le due successive che in quello stesso ciclo di clock hanno completato, rispettivamente, le fase di ID e IF



5. Aggiungere $0x80000180$ tra i segnali da poter scrivere nel PC (con logica di selezione)


Branch
ID/EX.MemRead
MUX
2
1
0
0x80000180
Hazard unit
IF/ID. rs
IF/ID. rt
ID/EX. rt
IF/IDWrite
CU
ID/EX
WB
M
EX
EX/MEM
MEM/WB
IF/ID
I[31-26] Opcode
4
+
BTA
prediction
Branch Target Buffer
PCWrite
PC
MEM I
Addr
I[31-0]
≪ 2
RegWrite
RegRead1
RegRead2
RF
WriteAddr
D
DataRead1
DataRead2
=
EPC
Cause
ALUSrc
overflow
FwA
FwB
ALU
ALUOp
CU ALU
AluCtrl
RegDst
MemRead
MemWrite
MEM D
Addr
DataRead
D
MemToReg
I[25-21] rs
I[20-16] rt
I[15-0]
Sign Extend
I[15-11] rd
I[20-16] rt
I[25-21] rs
IF.Flush
Forward unit
ID/EX. rt
ID/EX. rs
MEM/WB. rd
EX/MEM. rd
EX/MEMRegWrite
MEM/WBRegWrite

Esempio

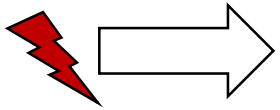
$0x40$ add \$12 \$12 \$13

$0x44$ add \$12 \$12 \$13

$0x48$ add \$12 \$12 \$13

$0x4C$ and \$1 \$2 \$4

$0x50$ beq \$12 \$0 \$1



Overflow

Fasi svolte nel ciclo di clock (appena concluso) in cui è stata rilevata l'eccezione?

WB

MEM

EX

ID

IF

Valori di questi registri?

PC
$0x54$

EPC $0x4C$
Cause 12

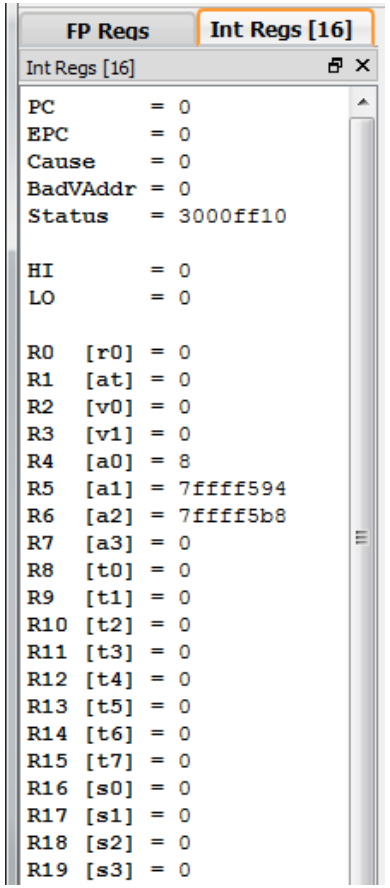
Eccezioni in MIPS

- Nella pratica i tipi di eccezioni riconosciute da una architettura sono più di due e includono anche interrupt causati da eventi esterni, nella nostra architettura pipeline semplificata non le abbiamo considerate

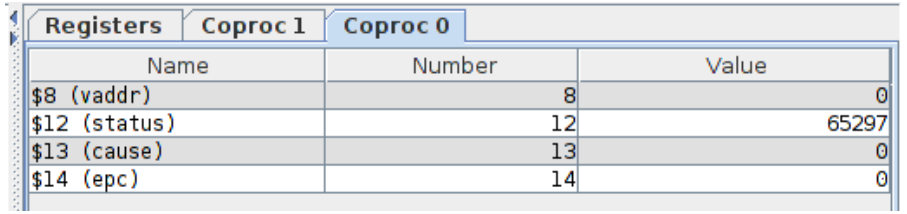
Number	Name	Cause of exception
0	Int	interrupt (hardware)
4	AdEL	address error exception (load or instruction fetch)
5	AdES	address error exception (store)
6	IBE	bus error on instruction fetch
7	DBE	bus error on data load or store
8	Sys	syscall exception
9	Bp	breakpoint exception
10	RI	reserved instruction exception
11	CpU	coprocessor unimplemented
12	Ov	arithmetic overflow exception
13	Tr	trap
15	FPE	floating point

- Se ammettiamo anche cause esterne (ad esempio la richiesta di una periferica di I/O) il concetto di offending instruction **non è sufficiente** a descrivere l'informazione che viene salvata in EPC
- *Chi è la offending instruction in caso di interrupt?* La risposta non è ovvia e richiede delle scelte architetturali su dove interrompere la pipeline
- **Scelta più logica**: non fare flush di nessuna istruzione, eseguire l'interrupt handler e ripristinare l'esecuzione dall'istruzione all'indirizzo che era contenuto in PC prima che iniziasse la successiva fase di IF

Eccezioni in MIPS

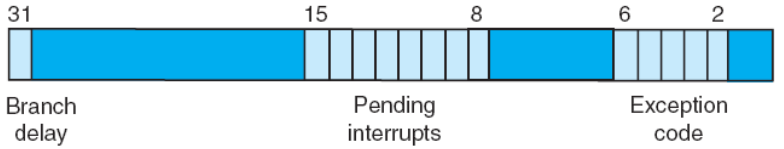


- **Coprocessore 0:** unità dedicata alla gestione delle eccezioni

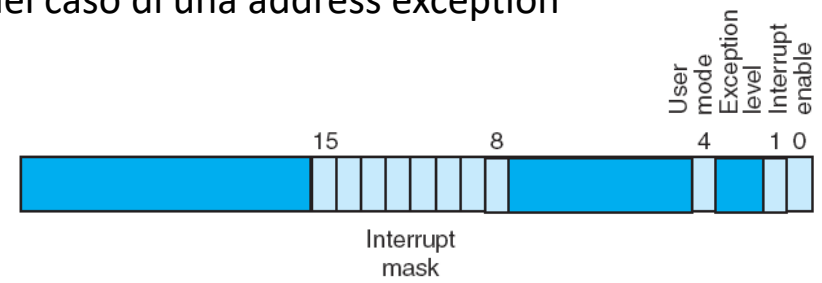


Name	Number	Value
$8 (vaddr)	8	0
$12 (status)	12	65297
$13 (cause)	13	0
$14 (epc)	14	0

- **Registro Cause**: causa dell'eccezione specificata con Exception Code (unsigned int, bit 2-6). Indica anche gli intterupt che al momento dell'eccezione erano ancora da servire



- **Registro EPC**: Exception Program Counter
- **Registro BadVaddr**: indirizzo errato nel caso di una address exception



- **Registro Status**: interrupt mask e bit di controllo

Architettura degli Elaboratori II

Corso di Laurea Triennale in Informatica

Università degli Studi di Milano

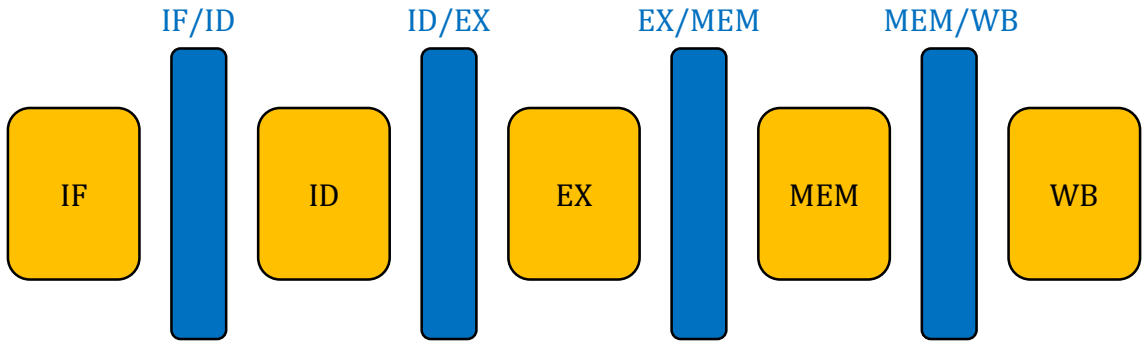
Dipartimento di Informatica “Giovanni Degli Antoni”

Edizione 2 (Cognomi H-Z), A.A. 2021-2022, Nicola.Basilico@unimi.it

Aumentare il parallelismo a livello di istruzione

Parallelismo a livello di istruzione

- **ILP**: instruction level parallelism, è il tipo di parallelismo sfruttato dalla pipeline
- Nella nostra architettura a 5 stadi ci possono essere fino a 5 diverse istruzioni contemporaneamente in elaborazione nella CPU, una per ogni stadio



- L'ILP è ciò che sta alla base del risultato teorico che caratterizza la performance della pipeline: k stadi significa (best case teorico) k volte più veloce dell'implementazione a singolo ciclo
- Siamo già 5 volte più veloci della singolo ciclo, possiamo fare meglio?

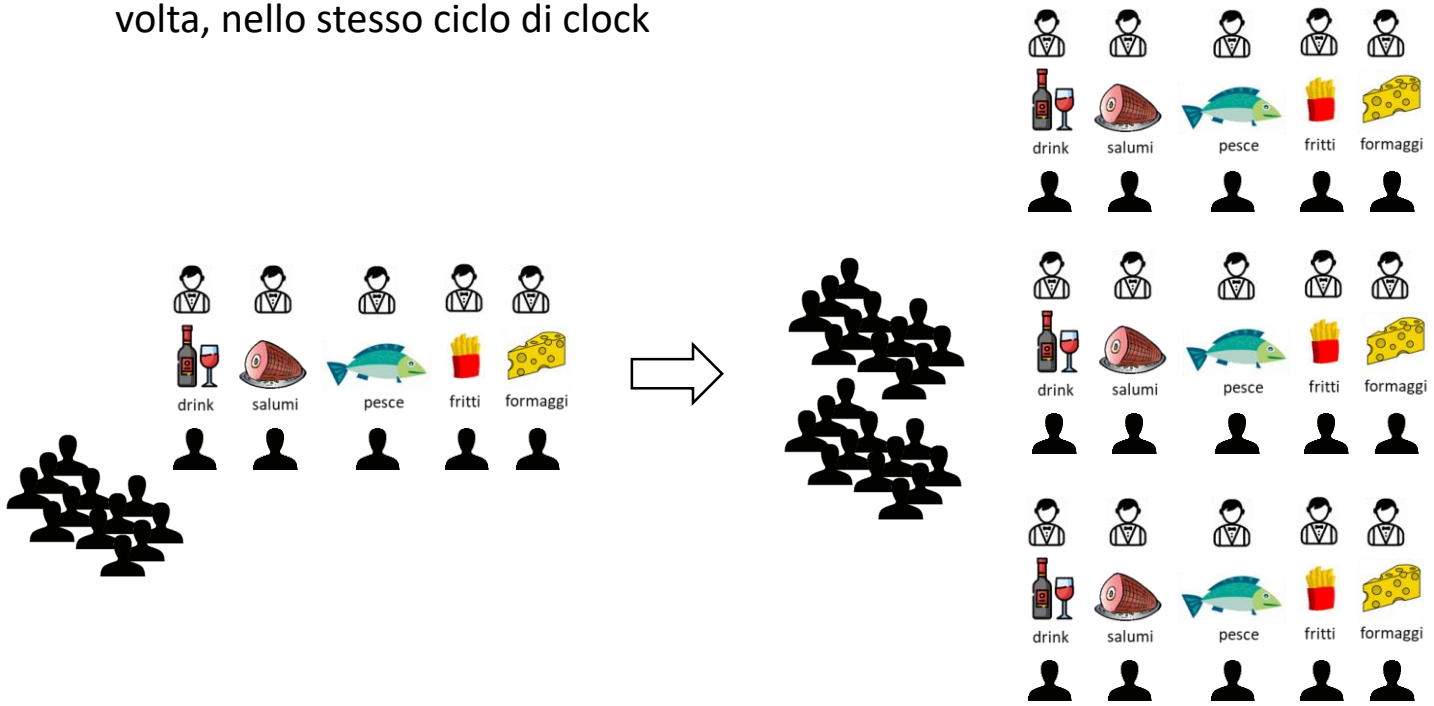
Profondità della pipeline

- Per poter aumentare l'ILP, e cioè aumentare il massimo numero di istruzioni contemporaneamente in esecuzione, esistono **due approcci**
- **Primo approccio (orizzontale)**: aumentare gli stadi della pipeline
- Ogni stadio svolge micro-operazioni più elementari
- La durata del ciclo di clock (che comunque deve essere la stessa per ogni stadio) diminuisce e il throughput aumenta
- Con instruction set più complesse l'esecuzione di una istruzione può essere «spezzettata» in più di 5 sotto-fasi

Num. stadi	Architettura
5	MIPS, Pentium
6	UltraSPARC T1
7	PowerPC G4e
8	UltraSPARC T2/T3, Cortex-A9
10	Athlon, Scorpion, Pentium 3
11	Krait
12	Pentium Pro/II/III, Athlon 64/Phenom, Apple A6
13	Denver
14	UltraSPARC III/IV, Core 2, Apple A7/A8, Skylake, Kabylake, Sandy Bridge, Pentium Pro
14/19	Core i2/i3 Sandy/Ivy Bridge, Core i4/i5 Haswell/Broadwell, Core i7
15	Cortex-A15/A57
16	PowerPC G5, Core i1 Nehalem, Bonnell
14 – 17	Silvermont
18	Bulldozer/Piledriver, Steamroller
20	Pentium 4, NetBurst (Willamette), NetBurst (Northwood)
31	Pentium 4E Prescott, NetBurst (Prescott), NetBurst (Cedar Mill)

Pipeline multi-issue

- **Secondo approccio (verticale)**: replicare le risorse hardware in modo che possano entrare in pipeline più istruzioni alla volta, nello stesso ciclo di clock



- Una pipeline di questo tipo si chiama pipeline **multi-issue** (a lancio multiplo) a k vie (nel nostro esempio $k = 3$)
- il numero delle vie è anche chiamato **IPC** (Instructions Per clock Cycle): il numero massimo di istruzioni che la pipeline è in grado lanciare in un singolo ciclo di clock
- Le CPU moderne hanno un IPC tra 3 e 6 e raramente inferiore a 2

- Problema: Il lancio multiplo implica la necessità di riconsiderare le problematiche legate ai data e control hazard e di riadattare le strategie di risoluzione dei conflitti/dipendenze
- **Esempio**: $\texttt{add \$1 \$1 \$2}$ e $\texttt{lw \$2 0(\$1)}$ non possono essere lanciate insieme, indipendentemente dall'ordine in cui compaiono nel codice

Pipeline multi-issue

- Problema centrale nell'implementazione di una pipeline multi-issue: come riempire gli issue slot?
- **Issue slot**: la «rampa di lancio delle istruzioni», in una pipeline con k vie ad ogni ciclo di clock dobbiamo riempire uno slot con k istruzioni che verranno lanciate nella pipeline contemporaneamente
- Non sempre è possibile riempire completamente uno slot con k istruzioni, in questi casi i posti «vuoti» verranno lasciati tali o, equivalentemente, occupati da delle nop
- Chi ha la responsabilità di **organizzare ad ogni ciclo di clock i lanci multipli** (ordine di esecuzione e riempimento degli slot, quando/chi) in modo che non si generino conflitti ed errori? Questa operazione si chiama anche **scheduling**
- Approccio software: ci pensa il compilatore, **multi-issue statica**
- Approccio hardware: ci pensa la CPU, **multi-issue dinamica**
- I due approcci spesso adottano principi simili e non sono mutuamente esclusivi, tipicamente sono presenti entrambi

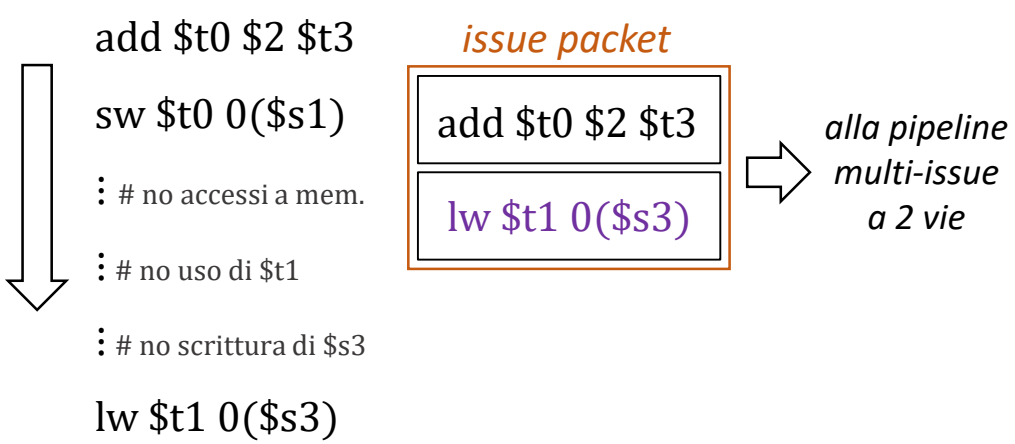
Nota sulla notazione registri

- Da ora adotteremo, dove serve, le convenzioni di nome per i registri MIPS

Identificatore	Numero	Utilizzo in MIPS
$zero	0	contiene la costante 0
$at	1	riservato all'assembler
$v0, $v1	2,3	valori di ritorno di una procedura
$a0,$a1, $a2, $a3	4,5,6,7	parametri da passare a una procedura
$t0,$t1, …, $t7	8,9, … , 15	registri temporanei (non preservati su chiamata di procedura)
$s0,$s1, …, $s7	16,17, … , 23	registri salvati (preservati su chiamata di procedura)
$t8,$t9	24,25	registri temporanei (non preservati su chiamata di procedura)
$k0,$k1	26,27	gestione delle eccezioni
$gp	28	global pointer
$sp	29	stack pointer
$s8	30	frame pointer
$ra	31	return address

Speculazione

- Sia con multi-issue statica (SW) che dinamica (HW) la maggiore difficoltà nel fare scheduling è quella di **non conoscere** gli effetti delle istruzioni prima di averle eseguite
- **Ma è proprio da quegli effetti che dipende la correttezza dello scheduling**
- **Esempio**: come posso riempire il posto libero nell'issue packet che sta per essere lanciato?



- Posso inserire la sw?
  - **No!** Le istruzioni che vengono lanciate insieme hanno un parallelismo totalmente sincronizzato sulle sotto-fasi (quando una è in una certa fase X anche l'altra lo è!) e la sw ha bisogno del risultato della add, il forwarding è impossibile
- Posso inserire la lw?
  - **Forse!** Se l'indirizzo a cui accede è diverso da quello della sw allora si potrebbe! Ma questo, in generale, non è deducibile dall'analisi del codice (per capirlo andrebbe eseguito!)

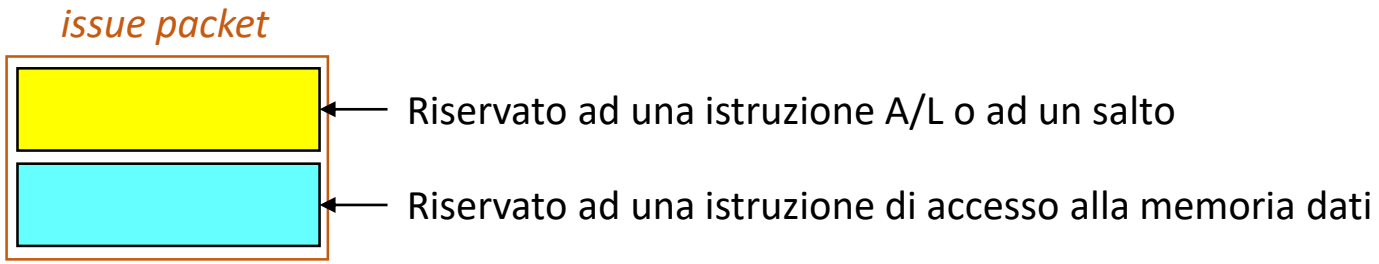
- Esempio di una **speculazione**: scommettere che l'indirizzo di accesso della lw sarà diverso e inserire l'istruzione nello issue packet

Speculazione

- La speculazione, come in ogni suo ambito di applicazione, comporta di agire sotto assunzioni di cui non si ha certezza di realizzazione
- Se le assunzioni si realizzano si ottengono dei vantaggi, se non si realizzano si paga un costo
- Nel nostro scenario questo aspetto si traduce in due requisiti operativi sia per la multi-issue statica che per quella dinamica:
    1. Controllare se la speculazione è andata a buon fine (**check**)
    2. Se non è andata a buon fine attuare una procedura di recovery (**roll back**): annullare/prevenire gli effetti delle istruzioni che sono state eseguite sotto l'assunzione di una predizione errata
- **Per il compilatore**: prevedere delle istruzioni speciali che controllino l'esito della speculazione e, nel caso, riparino gli errori
- **Per la CPU**: il risultati delle istruzioni speculate vengono temporaneamente mantenuti in un buffer fino a quando non è possibile stabilire l'esito della speculazione
    - Se l'esito è positivo i risultati vengono scritti nel RF o in memoria, questa operazione spesso si chiama **commit**
    - Se l'esito è negativo i risultati vengono scartati e si mandano in esecuzione le istruzioni corrette
- **Problema delle eccezioni**: se una istruzione speculata genera un'eccezione come ci si comporta?
- l'approccio è quello di aspettare a servire l'eccezione fino a che la speculazione non sia risolta

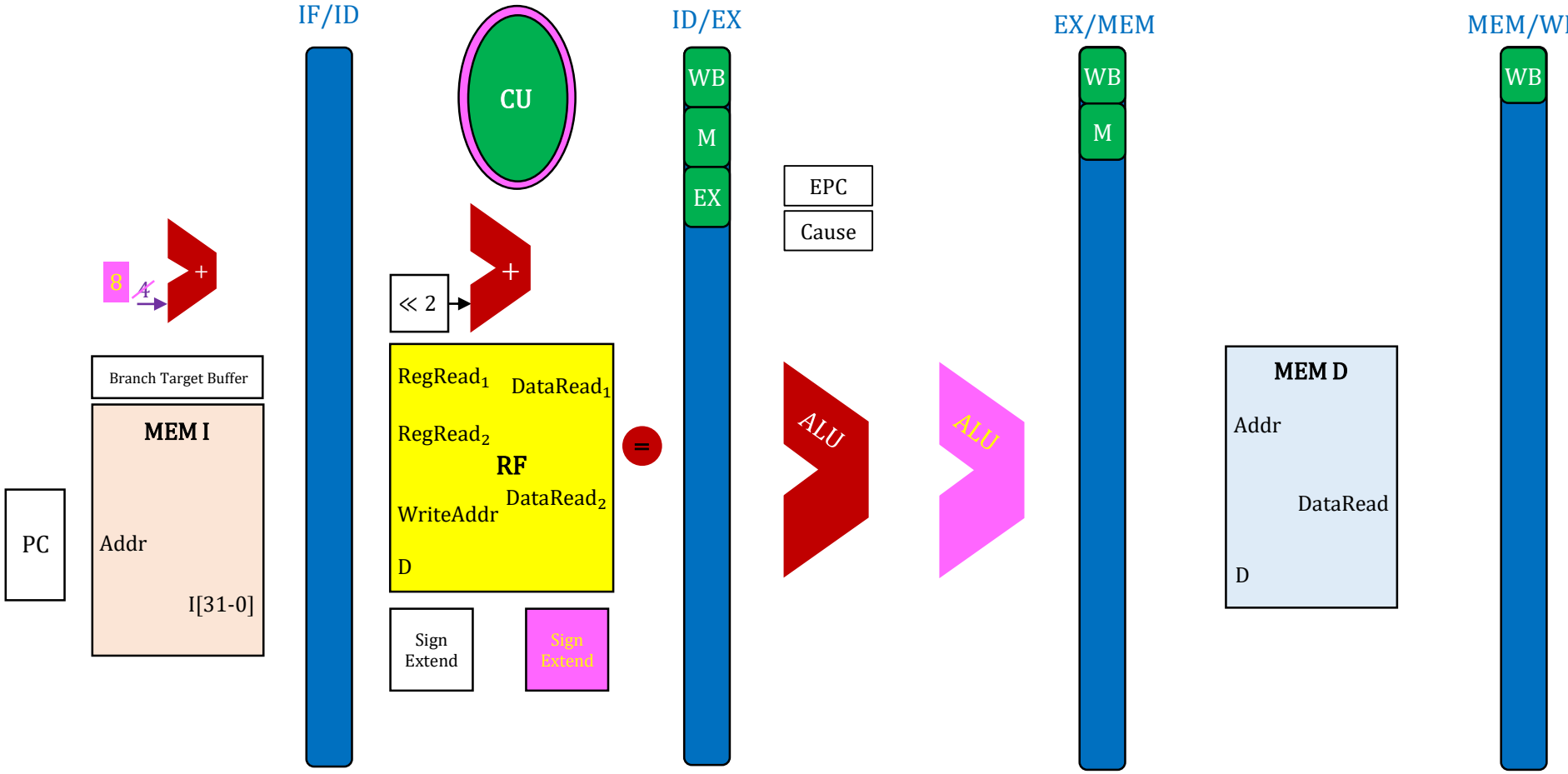
Multi-issue statica

- Consideriamo una CPU con pipeline multi-issue a 2 vie
- Primo aspetto implementativo: lo issue packet non può essere arbitrario, vengono imposti dei vincoli in modo da semplificare la gestione degli hazard



- Il compilatore ha l'onere di riordinare il codice assumendo che la CPU esegua dei fetch da 64 bit, cioè due istruzioni per volta, nel fare ciò deve:
    1. garantire che non ci siano hazard tra le istruzioni all'interno dello stesso issue packet
    2. cercare di minimizzare eventuali hazard tra istruzioni che stanno in issue packet diversi
- La CPU comunque rileva e risolve gli hazard tra issue packet diversi

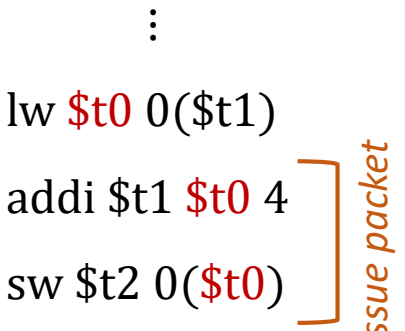
Multi-issue statica (data path)



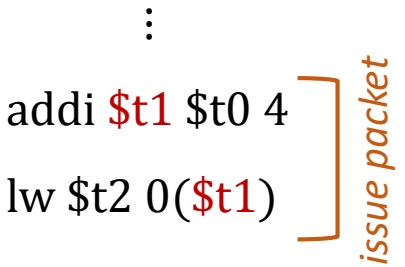
- Il prossimo indirizzo a cui fare il fetch è $PC + 8$
- I registri pipeline vanno estesi per poter contenere i risultati parziali di 2 istruzioni
- LA CU (e la logica di controllo in generale) va arricchita per poter decodificare 2 istruzioni alla volta
- Va aggiunta una ALU in più, due istruzioni potrebbero fare EX contemporaneamente
- Va aggiunto un modulo di estensione di segno per la seconda ALU (2 istruzioni possono necessitare dell'estensione di segno nella stessa fase, ad es. beq e lw)
- Bisogna aggiungere collegamenti (non li rappresentiamo per non appesantire la notazione)

Multi-issue statica (scheduling)

- L'essere passati a questa architettura ha un impatto sui data hazard, consideriamo gli hazard lw

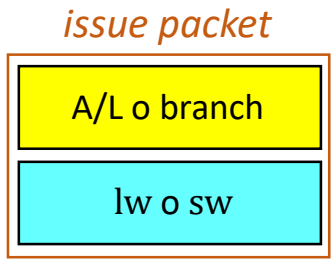


- La lw impone uno stallo al ciclo di clock successivo che però ora impatta su due istruzioni!



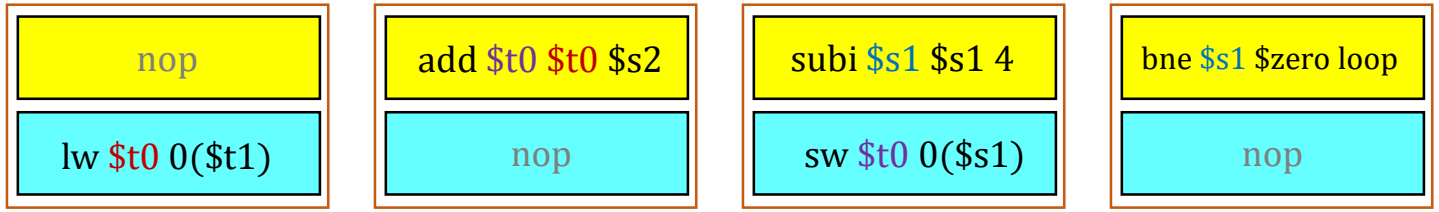
- La lw non può usare il risultato della add se si trova appaiata con lei nello stesso packet, se non si trova un sostituto da inserire nel pacchetto al posto della lw anche una istruzione A/L può generare uno stallo

Multi-issue statica (scheduling)

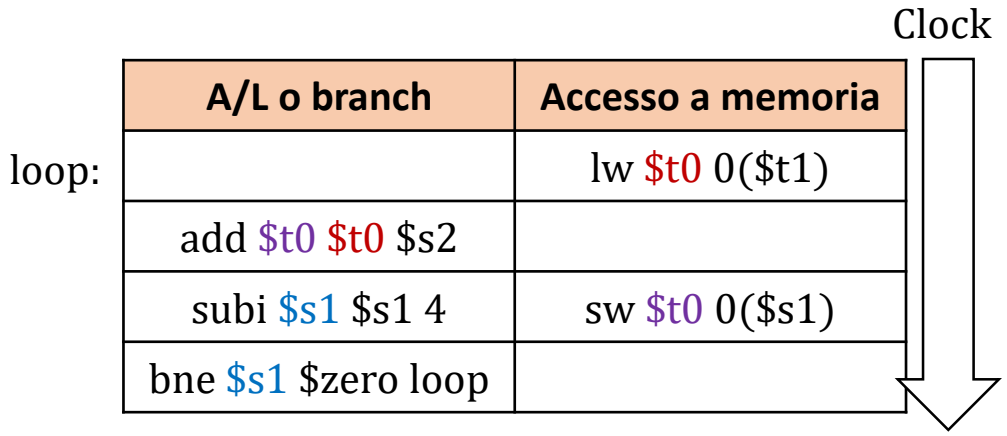


- Scheduling di questa sequenza di istruzioni

```
loop:   lw $t0 0($t1)
        add $t0 $t0 $s2
        sw $t0 0($s1)
        subi $s1 $s1 4
        bne $s1 $zero loop
```



- **Primo packet**: add usa il risultato della lw, subi causerebbe un errore nella sw, bne usa risultato di subi, serve nop
- **Secondo packet**: add può eseguire (CPU farà stallo), sw usa risultato di add, serve nop
- **Terzo packet**: $t0 è pronto, eseguo sw; subi può eseguire
- **Quarto packet**: bne



Multi-issue statica (loop unrolling)

- Supponiamo che, analizzando il codice, il compilatore scopra che il ciclo svolga **sempre almeno 4 iterazioni**

```
loop:   lw $t0 0($s1)
        add $t0 $t0 $s2
        sw $t0 0($s1)
        subi $s1 $s1 4
        bne $s1 $zero loop
```

unrolling di 4 iterazioni ⇨

```
        subi $s1 $s1 12
iterazione 1
        lw $t0 12($s1)
        add $t0 $t0 $s2
        sw $t0 12($s1)
iterazione 2
        lw $t0 8($s1)
        add $t0 $t0 $s2
        sw $t0 8($s1)
iterazione 3
        lw $t0 4($s1)
        add $t0 $t0 $s2
        sw $t0 4($s1)
loop:
iterazione 4
        lw $t0 0($s1)
        add $t0 $t0 $s2
        sw $t0 0($s1)
        subi $s1 $s1 4
        bne $s1 $zero loop
```

register renaming ⇨

```
        subi $s1 $s1 12
iterazione 1
        lw $t3 12($s1)
        add $t3 $t3 $s2
        sw $t3 12($s1)
iterazione 2
        lw $t2 8($s1)
        add $t2 $t2 $s2
        sw $t2 8($s1)
iterazione 3
        lw $t1 4($s1)
        add $t1 $t1 $s2
        sw $t1 4($s1)
loop:
iterazione 4
        lw $t0 0($s1)
        add $t0 $t0 $s2
        sw $t0 0($s1)
        subi $s1 $s1 4
        bne $s1 $zero loop
```

- **Loop unrolling**: esplicito le prime iterazioni del ciclo con un codice che produce un effetto equivalente, le iterazioni successive rimangono implicite (notare gli offset)
- **Register renaming**: elimino le **anti-dipendenze**, quelle dipendenze che dipendono dall'uso dello stesso registro ma non dello stesso dato

Multi-issue statica (loop unrolling)

```
        subi $s1 $s1 12
iter 1  lw $t3 12($s1)
        add $t3 $t3 $s2
        sw $t3 12($s1)
iter 2  lw $t2 8($s1)
        add $t2 $t2 $s2
        sw $t2 8($s1)
iter 3  lw $t1 4($s1)
        add $t1 $t1 $s2
        sw $t1 4($s1)
loop:   lw $t0 0($s1)
iter 4  add $t0 $t0 $s2
        sw $t0 0($s1)
        subi $s1 $s1 4
        bne $s1 $zero loop
```

Con loop unrolling e register renaming

A/L o branch	Accesso a memoria
subi $s1 $s1 12	lw $t3 **0**($s1)
	lw $t2 8($s1)
add $t3 $t3 $s2	lw $t1 4($s1)
add $t2 $t2 $s2	lw $t0 0($s1)
add $t1 $t1 $s2	sw $t3 12($s1)
add $t0 $t0 $s2	sw $t2 8($s1)
subi $s1 $s1 4	sw $t1 4($s1)
bne $s1 $zero loop	sw $t0 **4**($s1)

$IPC_{max} = 2, IPC_{min} = 1$

IPC empirico: $\text{IPC}_e = \frac{num.di\ istruzioni}{num.cilci\ di\ clock} = \frac{15}{8} = 1{,}875$

percentuale di efficienza: $\rho = \frac{IPC_e - IPC_{min}}{IPC_{max} - IPC_{min}} = 1{,}875 - 1 = 0{,}875$

Senza loop unrolling

A/L o branch	Accesso a memoria
	lw $t0 0($s1)
add $t0 $t0 $s2	
subi $s1 $s1 4	sw $t0 0($s1)
bne $s1 $zero loop	

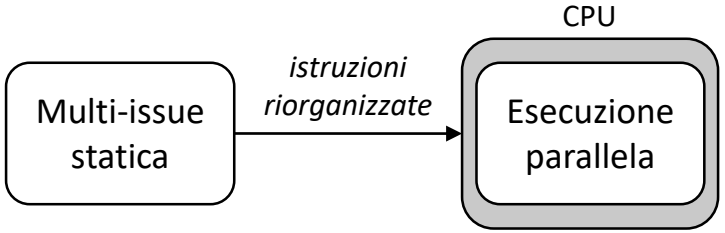
$$\text{IPC}_e = \frac{5}{4} = 1{,}25$$

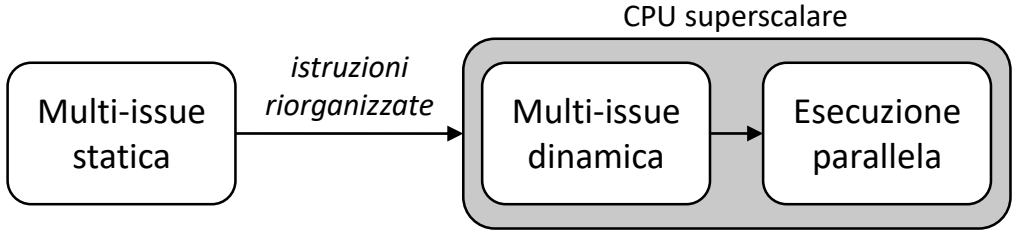
$$\rho = \frac{IPC_e - IPC_{min}}{IPC_{max} - IPC_{min}} = 1{,}25 - 1 = 0{,}25$$

- Applicando il loop unrolling otteniamo un miglioramento del 60%

Multi-issue dinamica

- CPU a pipeline con multi-issue dinamica: le decisioni su come riempire gli slot sono prese dall'hardware, man mano che le istruzioni vengono prelevate
- Questo tipo di processori sono detti **superscalari**
- **Approccio base**:
  - Le istruzioni vengono lanciate sempre nello stesso ordine in cui sono allocate nel segmento testo
  - La CPU decide, per ogni ciclo di clock, se non lanciarne nessuna (stallo), lanciarne una sola o lanciarne più di una contemporaneamente
- La decisione di scheduling dipende, come prima, dai vincoli di packaging delle istruzioni: hazard di tipo diverso possono limitare la parallelizzazione





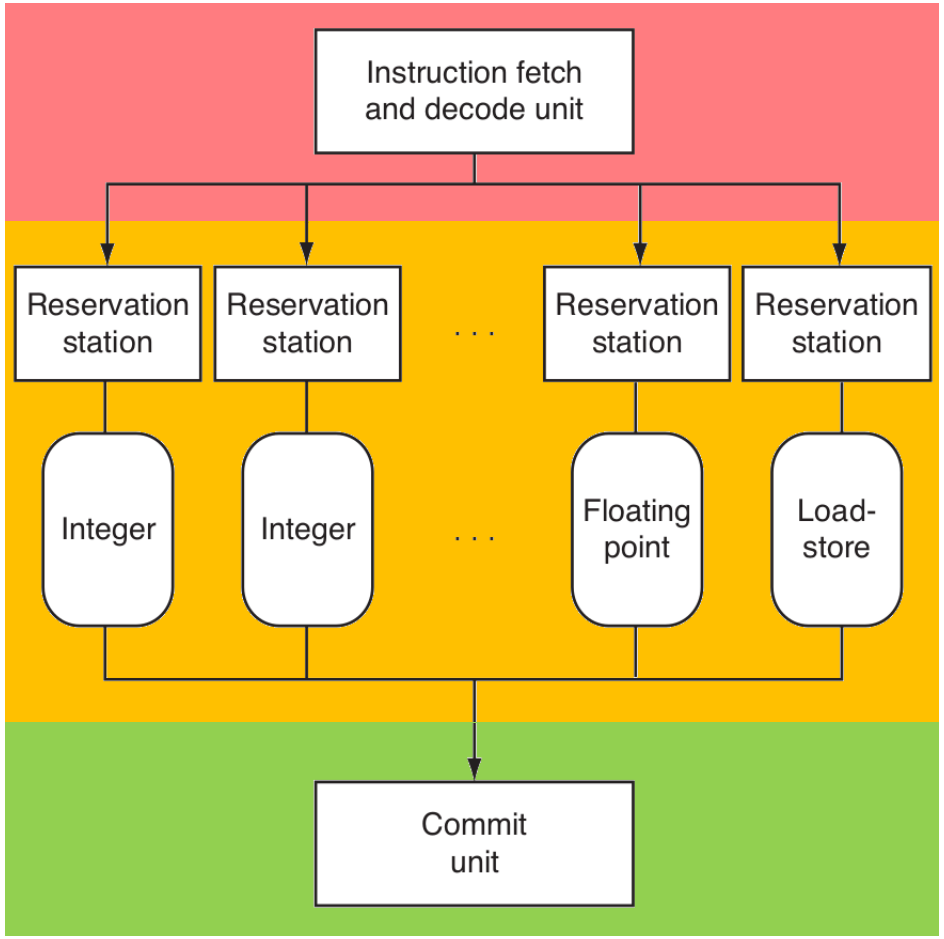
- La riorganizzazione del codice è fortemente legata allo specifico processore (non all'ISA!)
- Se si cambia processore **il codice va ricompilato** per evitare errori o grosse perdite di performance

- La multi-issue dinamica è supportata dal lavoro del compilatore: le istruzioni sono riorganizzate in modo da minimizzare gli hazard
- Con la multi-issue dinamica la CPU garantisce sempre la correttezza di esecuzione anche quando la riorganizzazione del compilatore genererebbe errori (ad es. perché fatta per un'altra CPU)

Scheduling dinamico

- L’approccio base è spesso arricchito con la possibilità di fare riordinamento delle istruzioni a livello hardware: **schedulazione dinamica della pipeline**
- L’hardware opera a sua volta una esecuzione riordinata che, sempre garantendo la correttezza dell’esecuzione, aumenti le performance prevenendo gli stalli
- Per il momento non consideriamo la speculazione, quindi il problema è dato semplicemente dagli hazard
- Per poter fare schedulazione dinamica della pipeline si utilizza una architettura diversa composta da 3 macro-stadi

Scheduling dinamico



Prelievo e decodifica in-order: le istruzioni vengono prelevate una alla volta nell'ordine in cui sono presenti nel segmento testo (quello deciso dal compilatore) e vengono decodificate, in questo momento si rilevano gli hazard

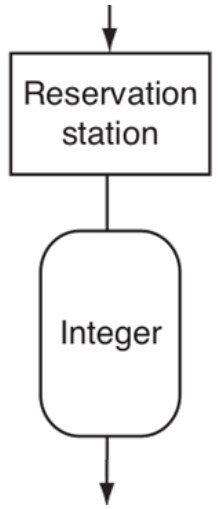
Esecuzione out-of-order: è il "banco di lavoro" della pipeline composto da un numero di unità funzionali (di solito ≥ 12)

- ogni unità funzionale si occupa di eseguire un particolare tipo di istruzione, lavorano tutte in parallelo
- Potendo terminare in tempi diversi questo tipo di esecuzione può comportare un riordinamento delle istruzioni

Commit in-order: è un buffer per i risultati delle istruzioni che vengono scritti nel RF o nella memoria nell'ordine corretto (quello indicato dal segmento testo)

Scheduling dinamico

- Ciascuna unità funzionale ha un buffer al suo ingresso chiamato **reservation station**
- Dopo essere stata prelevata e decodificata, una istruzione i viene lanciata verso l'unità funzionale associata (ad esempio un'istruzione aritmetica su interi viene lanciata ad una unità «Integer», un accesso a memoria invece verso una unità «Load-Store»)
- All'arrivo di *i* presso l'unità funzionale, nella reservation station viene creato un record che contiene:
  - L'operazione che i deve svolgere, ad esempio add
  - Il **valore** degli operandi su cui deve svolgere l'operazione
- **Attenzione**: se per via di un hazard uno o più valori di operandi non sono disponibili nel RF, **il valore dell'operando non viene copiato e l'ingresso di i nell'unità funzionale è posto in attesa**
- La i ha «prenotato» (reservation) la sua unità funzionale, ma attende di poter iniziare: durante questa attesa gli operandi mancanti potrebbero diventare disponibili (essere calcolati in un'altra unità funzionale) e venire trasferiti **tempestivamente** dentro il reservation record di i
- Non appena il reservation record di i è completo (tutti gli operandi sono stati copiati) e l'unità funzionale è libera i può accedere all'unità e il risultato viene calcolato
- Questo risultato viene **copiato dentro la commit unit** e **inviato verso tutte quelle reservation station che hanno una istruzione ferma in attesa per quell'operando**



Scheduling dinamico

Registri

s1	44	s3	33
s2	23	s4	7

Memoria

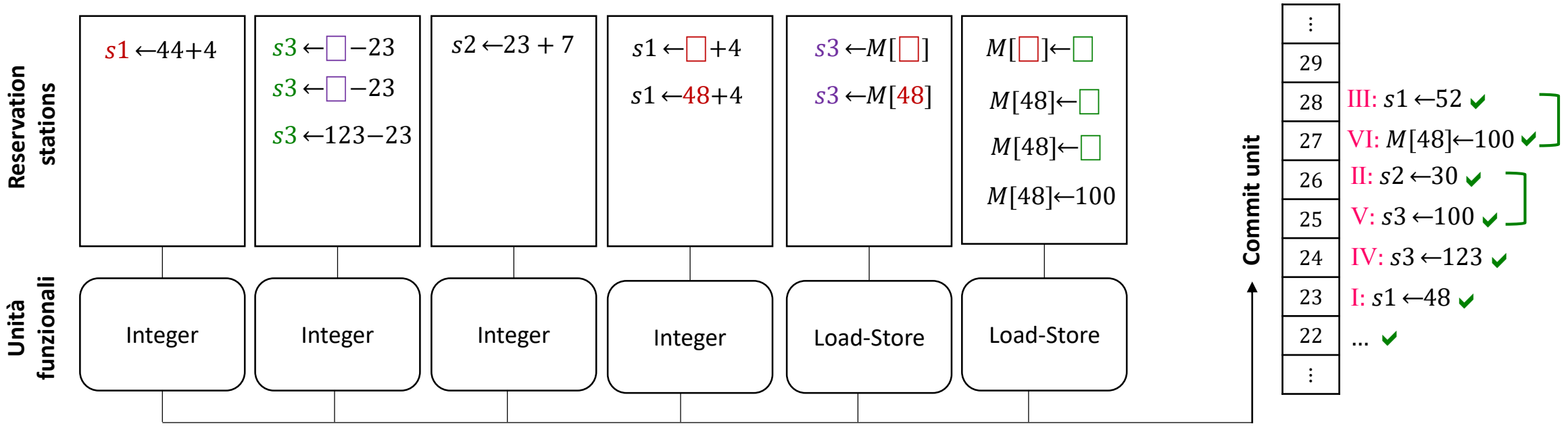
⋮	…
0x52	…
0x48	123
0x44	…
⋮	…

```
0x4C …
0x50 addi $s1 $s1 4
0x54 lw $s3 0($s1)
0x58 sub $s3 $s3 $s2
0x5C add $s2 $s2 $s4
0x60 sw $s3 0($s1)
0x64 addi $s1 $s1 4
```

Legenda

- □: operando non ancora pronto (il colore indica la dipendenza)
- I, II,III, … : ordine di arrivo alla commit unit
- ✔ : commit effettuato
-] commit pronti nello stesso momento

- Si assuma che tutte e 6 le istruzioni siano nello stesso momento le prossime ad avere la reservation dell'unità funzionale assegnatagli (si veda la figura)



Scheduling dinamico e renaming

```
0x50 lw $t0 0($t1)
0x54 addi $t0 $t0 10
0x58 sw $t0 0($t1)
0x5C lw $t0 4($t1)
0x60 addi $t0 $t0 10
0x64 sw $t0 4($t1)
0x68 lw $t0 8($t1)
0x6C addi $t0 $t0 10
0x70 sw $t0 8($t1)
```

Registri

t0	x
t1	40

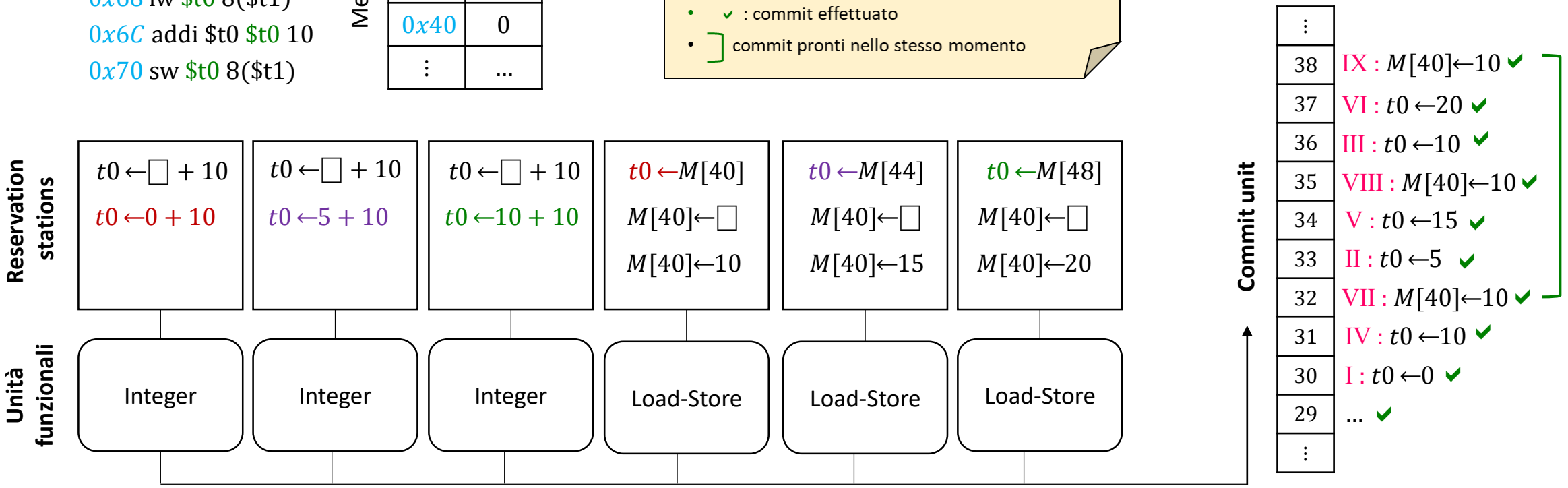
Memoria

⋮	...
0x48	10
0x44	5
0x40	0
⋮	...

- Il calcolo dei risultati (fase di EX) avviene all'interno di un buffer nella reservation station, **è una forma di register renaming fatto dallo hardware**

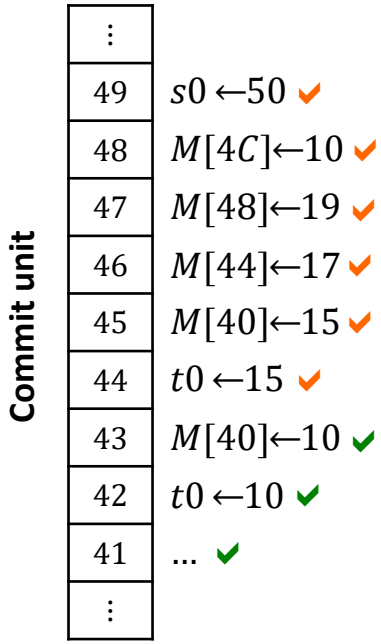
Legenda

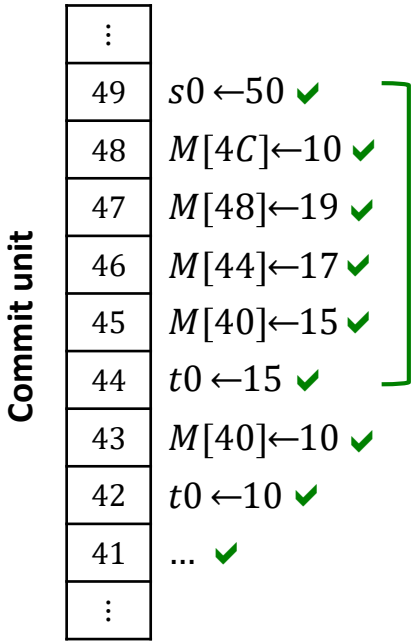
- □: operando non ancora pronto (il colore indica la dipendenza)
- I, II,III, ... : ordine di arrivo alla commit unit
- ✔ : commit effettuato
-] commit pronti nello stesso momento

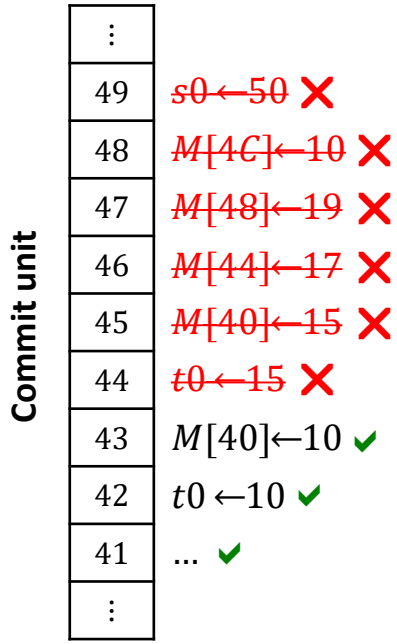


Scheduling dinamico: speculazione

- I risultati delle istruzioni speculate vengono tenute in attesa ma marcate come pronte per il commit: ✔
- **Speculazione corretta**: si procede con i commit ✔
- **Speculazione errata**: i risultati vengono scartati ✘, eventualmente fare anche flush delle unità funzionali coinvolte







Scheduling dinamico: vantaggi

- Permette di fare scheduling a runtime utilizzando informazioni sullo stato dell'esecuzione del programma che non possono essere note al compilatore (ad esempio, vedremo che non tutti gli stalli sono predicibili)
- Se la CPU sta speculando sui branch usando un approccio dinamico le predizioni vengono fatte a runtime (e dipendono dallo storico di risoluzione dei salti): non è quindi possibile stabilire un ordine delle istruzioni a compile time poiché non si conosce la predizione, senza multi-issue dinamica perderemmo i vantaggi della speculazione
- Maschera i dettagli dell'implementazione hardware e permette di concentrarsi solo sull'ISA: un codice compilato per una implementazione diversa dell'ISA (legacy) può comunque girare senza errori

Scheduling dinamico e multi-issue statica: limiti

- Uno dei fenomeni che può limitare molto l'ILP in una pipeline multi-issue è l'aliasing tramite puntatori
- **Esempio**:

```
int foo(int *a, int *b) {
    *a = 4;
    *b = 5;
    return *a;
}
```

MIPS gcc 5.4.0

no flag di ottimizzazione

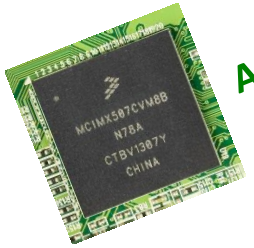
```
foo:    addiu $sp,$sp,-8
        sw $fp,4($sp)
        move $fp,$sp
        sw $4,8($fp)
        sw $5,12($fp)

        lw $2,8($fp)        # $2<-a
        li $3,4             # $3<-4
        sw $3,0($2)         # M[a]<-4

        lw $2,12($fp)       # $2<-b
        li $3,5             # $3<-4
        sw $3,0($2)         # M[b]<-5

        lw $2,8($fp)        (1)
        nop                 |  (2) li $2,4
        lw $2,0($2)         |

        move $sp,$fp
        lw $fp,4($sp)
        addiu $sp,$sp,8
        j $31
```

Identificatore	Numero	Utilizzo in MIPS
$zero	0	contiene la costante 0
$at	1	riservato all'assembler
$v0, $v1	2,3	valori di ritorno di una procedura
$a0,$a1, $a2, $a3	4,5,6,7	parametri da passare a una procedura
$t0,$t1, ..., $t7	8,9, ... , 15	registri temporanei (non preservati su chiamata di procedura)
$s0,$s1, ..., $s7	16,17, ... , 23	registri salvati (preservati su chiamata di procedura)
$t8,$t9	24,25	registri temporanei (non preservati su chiamata di procedura)
$k0,$k1	26,27	gestione delle eccezioni
$gp	28	global pointer
$sp	29	stack pointer
$s8	30	frame pointer
$ra	31	return address

(1) vs (2)

- Quale delle due dovrebbe produrre il compilatore?
- La 2 sarebbe molto più efficiente: una semplice scrittura di registro contro 2 lw e una nop!
- Ma il compilatore non può inserirla perché non può escludere *a priori* che la funzione non venga mai invocata passando lo stesso indirizzo come primo e secondo parametro
- Se il compilatore non può stabilire se una dipendenza ci sarà o non ci sarà ha due alternative
  - speculare sugli indirizzi (può essere molto complicato)
  - **assumere che la dipendenza c'è**
- Questi stessi limiti possono affliggere anche l'hardware a runtime

ARM Cortex-A8 e Intel Core i7



ARM Cortex A8

Intel Core i7 920

Target	Dispositivi mobili	Server, Cloud, Workstation
TDP*	2 W	130 W
Ciclo di clock / Frequenza	1 ns / 1 GHz	0,37 ns / 2,66 GHz
Multi-issue?	Sì, dinamica	Sì, dinamica
IPC	2	4
Stadi pipeline	14	14
Scheduling pipeline	Statico (esecuzione in-order)	Dinamico (esecuzione out-of-order) con speculazione

* **TDP**: Thermal Design Power, è la potenza media dissipata dal processore con tutti i core attivi (viene misurata su un benchmark)

Architettura degli Elaboratori II

Corso di Laurea Triennale in Informatica

Università degli Studi di Milano

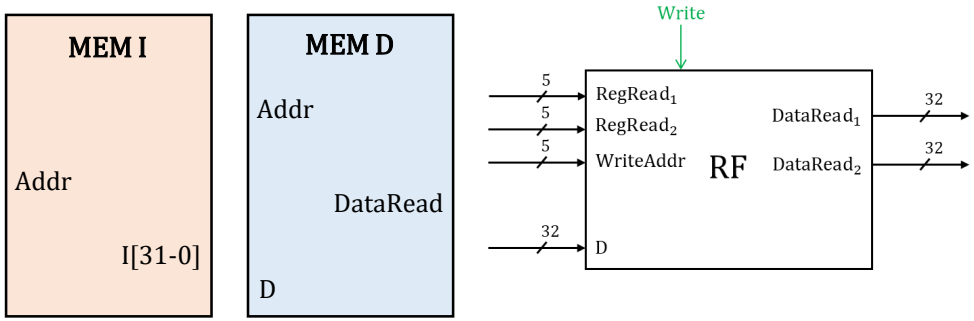
Dipartimento di Informatica “Giovanni Degli Antoni”

Edizione 2 (Cognomi H-Z), A.A. 2021-2022, Nicola.Basilico@unimi.it

Memorie

Architettura di un sistema di memoria

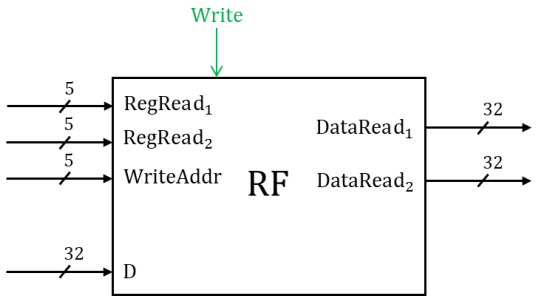
- Fino ad ora la memoria è stata rappresentata come un componente nel datapath che abbiamo utilizzato tramite la sua interfaccia: data una richiesta fatta dalla CPU (lettura o scrittura) il componente la eseguiva

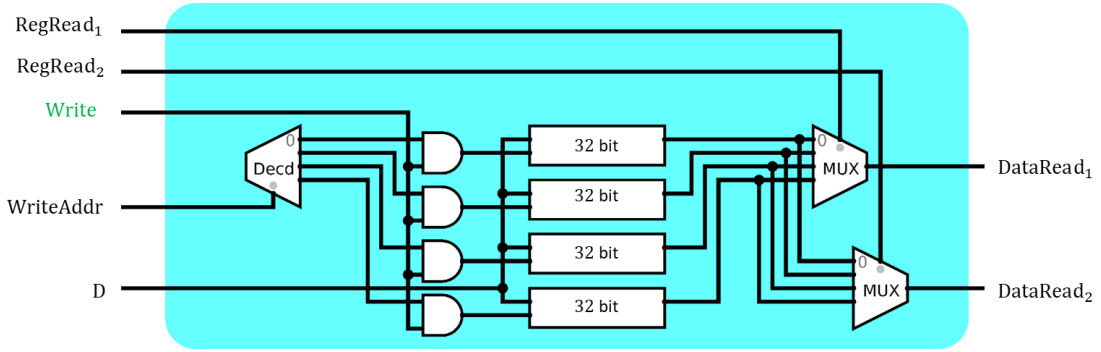


- **MEM I**: la memoria che contiene le istruzioni (segmento testo)
- **MEMD D**: la memoria che contiene i dati su cui lavorano le istruzioni
- **Register File**: il banco di lavoro della CPU

- La memoria rappresenta un punto critico legato alla performance di una CPU che necessita di essere trattato **a livello architetturale** (non solo migliorando l'efficienza della tecnologia)
- Per capire il motivo analizziamo le tecnologie di queste memore e i dati storici delle performance associate, considerando specialmente gli effetti descritti dalla legge di Moore

Il register file





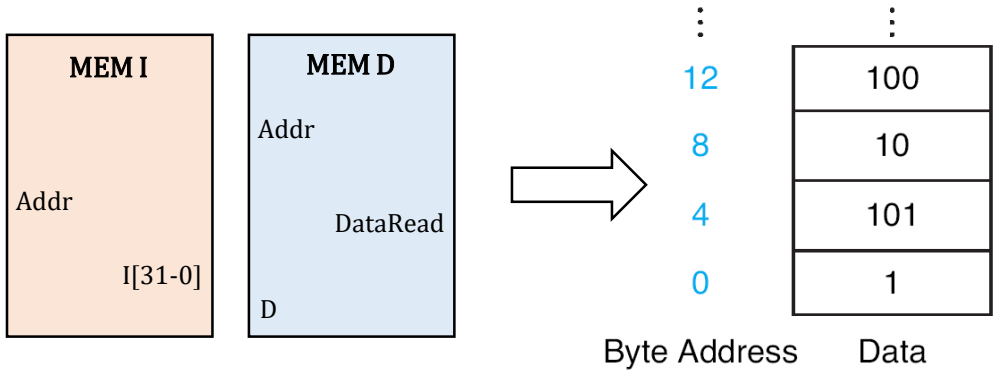
- Memoria di lavoro della CPU, installata al suo interno
- Tempo di accesso:
  - In **lettura** dipende dal cammino critico del multiplexer
  - In **scrittura** dipende dal cammino critico dei decoder

⇒ I cammini critici aumentano con l'aumentare del numero di registri e quindi della capacità

- È una memoria **volatile**, conserva il dato finché alimentata altrimenti il dato svanisce
- Tecnologia **SRAM** (Static Random Access Memory):
  - Random Access: letture e scritture hanno costi simili che **non dipendono da dove il dato è fisicamente immagazzinato**
  - Static: **non** è necessario aggiornare (refresh) le celle di memoria per mantenere il dato

⇒
- Per memorizzare 1 bit usa da 6 a 8 transistor in modo da evitare fenomeni di degradamento dell'informazione
- Alte performance, ma costi elevati

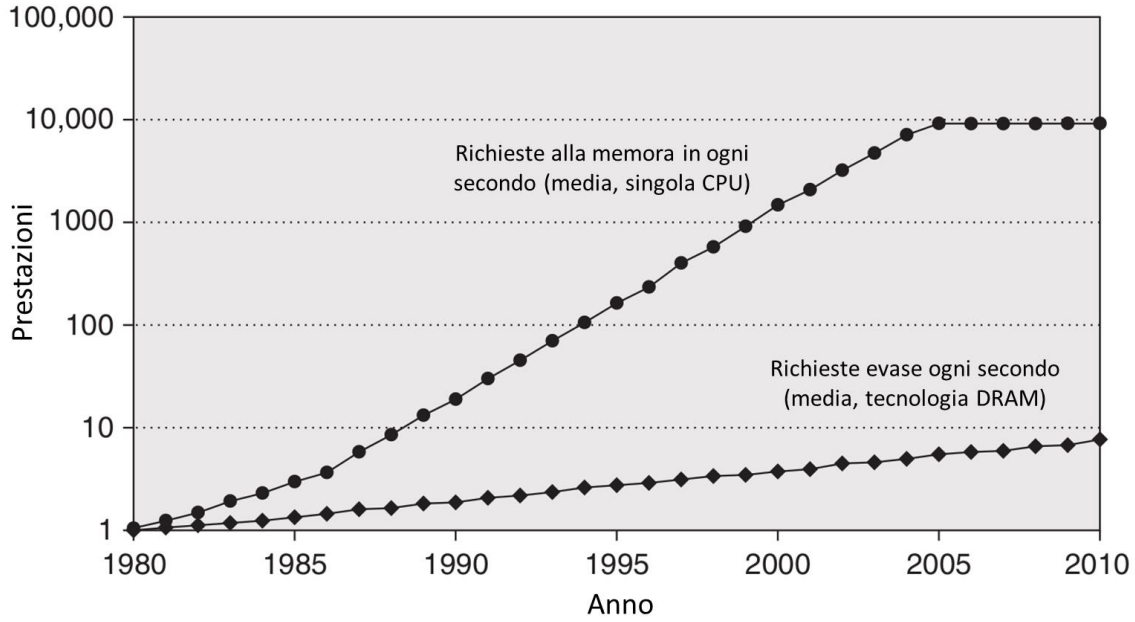
La memoria principale (Main Memory)



- Modellata come un array unidimensionale dove ogni elemento è una parola di memoria a cui è associato un indirizzo
- In MIPS: indirizzi di 32 bit **sui singoli byte**, parole di 4 byte, allineamento su multipli di 4

- Memoria esterna alla CPU da cui prelevare istruzioni e dati che alimentano l'elaborazione
- È una memoria **volatile**, conserva il dato finché alimentata altrimenti il dato svanisce
- Tecnologia **DRAM** (Dynamic Random Access Memory):
  - Random Access: letture e scritture hanno costi simili che **non dipendono da dove il dato è fisicamente immagazzinato**
  - Dynamic: **è necessario** aggiornare (refresh) le celle di memoria per mantenere il dato (logica dei condensatori)
    - Refresh di una cella di memoria: (1) lettura del valore della cella, (2) riscrittura del valore letto nella cella
    - Tipicamente avviene ogni $\sim 65\text{ms}$; nel caso in cui il dato debba cambiare, in (2) subentra il nuovo valore al posto del vecchio
- Per memorizzare 1 bit usa **1 transistor**: performance più basse, ma anche i costi si abbassano
- SDRAM: DRAM che integra un clock per sincronizzazione automatica con la CPU
- DDR SDRAM: **Double Data Rate**, è in grado di servire richieste di trasferimento su entrambi i fronti di un ciclo di clock!

Processor-Memory bottleneck



- Lo sviluppo tecnologico nei processori (anche grazie ad architetture sempre più complesse) ha generato progressi di performance più sostenuti rispetto a quanto successo nel campo delle memorie
- Il gap ha un impatto significativo sulle performance perché gli accessi costituiscono l'attività principale della CPU (90% del tempo) che le consente di accedere alle istruzioni e ai dati
- Non possiamo sperare che sia la tecnologia a livello di singolo componente di memoria a colmare questo gap, si deve puntare su come la tecnologia è organizzata: **progettare una architettura anche per il sistema di memoria**

Richiamo sulle unità di misura della memoria

Prefissi basati su Sistema Internazionale (SI)

Nome	Abbreviazione	Fattore
KiloByte	KB	10^3
MegaByte	MB	10^6
GigaByte	GB	10^9
TeraByte	TB	10^{12}
PetaByte	PB	10^{15}
ExaByte	EB	10^{18}
ZettaByte	ZB	10^{21}
YottaByte	YB	10^{24}

Prefissi basati su Sistema Binario (IEC)

Nome	Abbreviazione	Fattore
KibiByte	KiB	2^{10}
MebiByte	MiB	2^{20}
GibiByte	GiB	2^{30}
TebiByte	TiB	2^{40}
PebiByte	PiB	2^{50}
ExbiByte	EiB	2^{60}
ZebiByte	ZiB	2^{70}
YobiByte	YiB	2^{80}

Architettura del sistema di memoria

- **Compito**: scrivere una relazione su un tema assegnatoci, ad esempio *la storia d'Italia*
- Abbiamo accesso ad una grande biblioteca, per scrivere la nostra relazione dovremo consultare un certo numero di volumi di storia
- Problema: come organizziamo il nostro lavoro sapendo che dovremo avvalerci del servizio offerto dalla biblioteca?

Approccio 1

- Vado in biblioteca e prelevo il primo libro di cui ho bisogno: *«L'Italia del Risorgimento (183-1861)»*
- Porto il libro sulla scrivania e inizio il lavoro; poco dopo mi accorgo che mi serve un altro libro *«L'Italia del Settecento (1700-1789)»*
- Torno alla biblioteca e prelevo il libro che, tra l'altro, era riposto vicino al precedente
- Porto il libro sulla scrivania e continuo il lavoro; poco dopo mi accorgo che mi serve un altro libro *«L'Italia dei Notabili (1861-1900)»*
- Torno alla biblioteca …

Approccio 2

- Vado in biblioteca e prelevo un po' di libri (il massimo numero consentito dalla biblioteca) dallo scaffale *«Storia d'Italia»*
- Porto i libri sulla scrivania e inizio il lavoro consultando il primo libro; poco dopo mi accorgo che mi serve un altro libro, ma è già sulla scrivania, non devo tornare in biblioteca!
- Dopo un tempo maggiore necessito di un libro che non avevo prelevato, torno in biblioteca riconsegno i libri di prima e ne prendo di nuovi
- …

- **Quale dei due approcci permette di finire prima la relazione?**

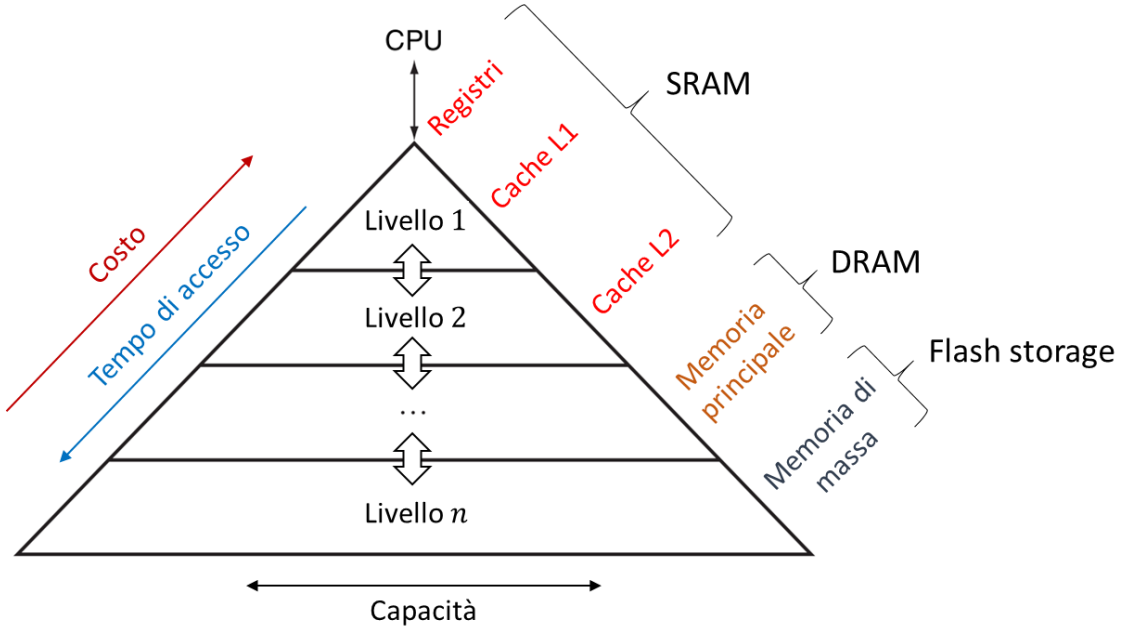
Architettura del sistema di memoria

- Il secondo approccio funziona meglio perché sfrutta due **principi di località**
- **Località temporale**: se dopo un tempo t si ha necessità di accedere ad un dato è molto probabile che dopo un tempo $t + \epsilon$ (con $\epsilon \ll t$) si avrà necessità di accedere ancora allo stesso dato
- **Località spaziale**: se dopo un tempo t si ha necessità di accedere ad un dato è molto probabile che dopo un tempo $t + \epsilon$ (con $\epsilon \ll t$) si avrà necessità di accedere ad un dato fisicamente vicino
- Questi due principi descrivono abbastanza bene il comportamento dei programmi, che rappresentano insiemi di operazioni eseguite in modo fortemente strutturato:
  - spesso si fanno cose ripetute (es. incrementare il valore di una variabile)
  - spesso si fanno cose in sequenza (es. scorrere gli elementi di un array)

- In media, gli accessi a memoria di un programma seguono questi due principi di località
- **Idea**: strutturiamo il sistema di memoria in modo che aumenti l'efficienza del lavoro sfruttando questo principio (come quanto fatto per la biblioteca)
- La biblioteca rappresenta la memoria principale, la scrivania sarà un'altra memoria (più piccola, ma più veloce) dove al tempo t mantenere quei dati (i libri) che, secondo i due principi di località, è molto probabile siano necessari al programma
- L'architettura che studiamo si chiama: **gerarchia di memoria**

Gerarchia di memoria

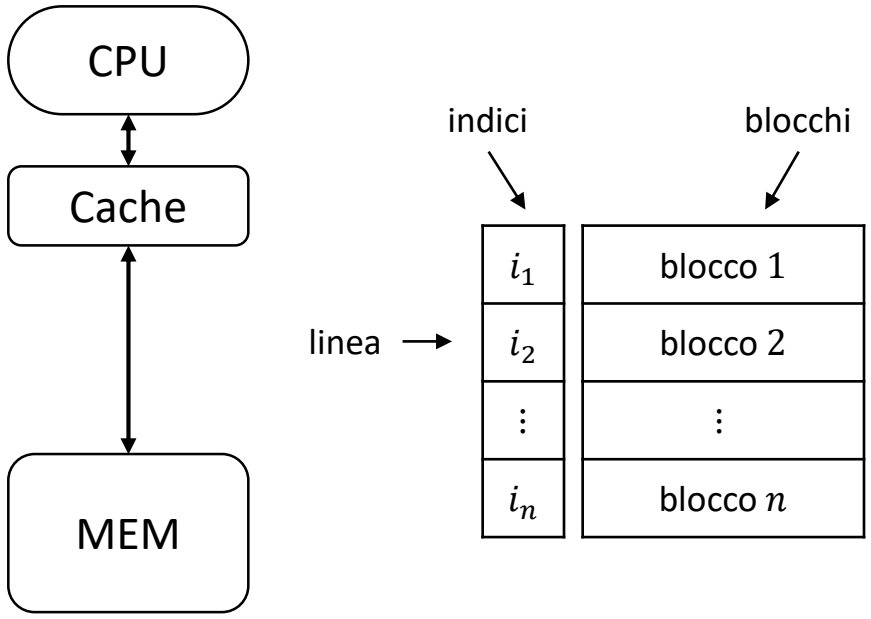
- Il sistema di memoria è progettato come una gerarchia a livelli
- I trasferimenti possono avvenire solo tra livelli adiacenti
- Anche per i dati viene mantenuta una organizzazione gerarchica: il livello i contiene un sotto-insieme dei dati immagazzinati al livello $i + 1$
- I livelli che stanno tra il banco registri e la memoria principale vengono chiamati **cache** (il *nascondiglio*, una memoria nascosta)
- Livelli bassi: alte performance, ma costi alti quindi bassa capacità
- Livelli alti: basse performance, ma costi bassi quindi alta capacità



- Le richieste di accesso (lettura o scrittura) sono gestite seguendo la gerarchia
- Richiesta di un dato d al livello i, il dato è contenuto nel livello i?
  - **Sì**: la richiesta viene evasa (lettura o scrittura)
  - **No**: la richiesta viene inoltrata al livello $i + 1$ sottostante

Modello della cache

- Consideriamo una singola cache dati tra il processore e la memoria principale
- La cache può essere modellata, come per la memoria principale, con un array monodimensionale dove ogni elemento è una linea indirizzata da un **indice** e contenente un **blocco** di dati



Terminologia del funzionamento della cache

Linea/Blocco	La minima unità informativa che può essere presente o non presente nella cache
Hit	Evento che indica che una richiesta di accesso può essere servita dalla cache perché il blocco richiesto è presente
Miss	Evento che indica che una richiesta di accesso non può essere servita dalla cache perché il blocco richiesto non è presente
Hit rate	Percentuale di hit su totale di richieste
Miss rate	Percentuale di miss su totale di richieste
Hit time	In caso di hit, tempo di accesso alla cache che include lo stabilire anche se hit o miss
Miss penalty	In caso di miss, tempo per recuperare e trasferire in cache il blocco mancante + tempo di accesso

- **Split cache**: la cache istruzioni può essere separata dalla cache dati, in ogni caso valgono le stesse considerazioni
- Con una split cache non abbiamo competizione di accesso tra dati e istruzioni, ma non possiamo ripartire dinamicamente l'uso della memoria tra le due tipologie di informazione

Notazione

d	un dato che ha un indirizzo in memoria principale, per noi è il **byte** che può anche rappresentare una parola di 32 bit
$M(d)$	l'indirizzo del dato d in memoria principale, per noi è sempre su 32 bit
L	numero di linee della cache o, in alternativa, il numero di blocchi che la cache può contenere
B	numero di dati contenuti sequenzialmente in un blocco (la dimensione di un blocco), **in byte**
$N(d)$	numero del blocco in memoria in cui si trova d: se guardiamo la memoria come ad una sequenza di blocchi di dimensione B numerati progressivamente con $0,1,2,\ldots$ quale è il numero del blocco che contiene d?
$div(a,b)$	quoziente della divisione intera tra a e b: $\left\lfloor \frac{a}{b} \right\rfloor$
$mod(a,b)$	resto della divisione intera tra a e b: $a - \left\lfloor \frac{a}{b} \right\rfloor b$
$I(d)$	indice della linea di cache a cui si trova il blocco che contiene il dato d

- Assunzione temporanee (che affronteremo dopo):
  - trascuriamo la gestione dei cache miss
  - consideriamo soltanto accessi in lettura

Mappatura diretta

- **Primo problema**: determinare l'indice di linea $I(d)$ per un dato d
- **Mappatura diretta**: ad ogni **blocco** in memoria principale viene associato un **indice univoco** della cache
- **Attenzione**: la corrispondenza non è biunivoca, il numero di blocchi in memoria è di norma maggiore del numero di linee della cache, quindi un indice di linea è condiviso da più blocchi della memoria principale

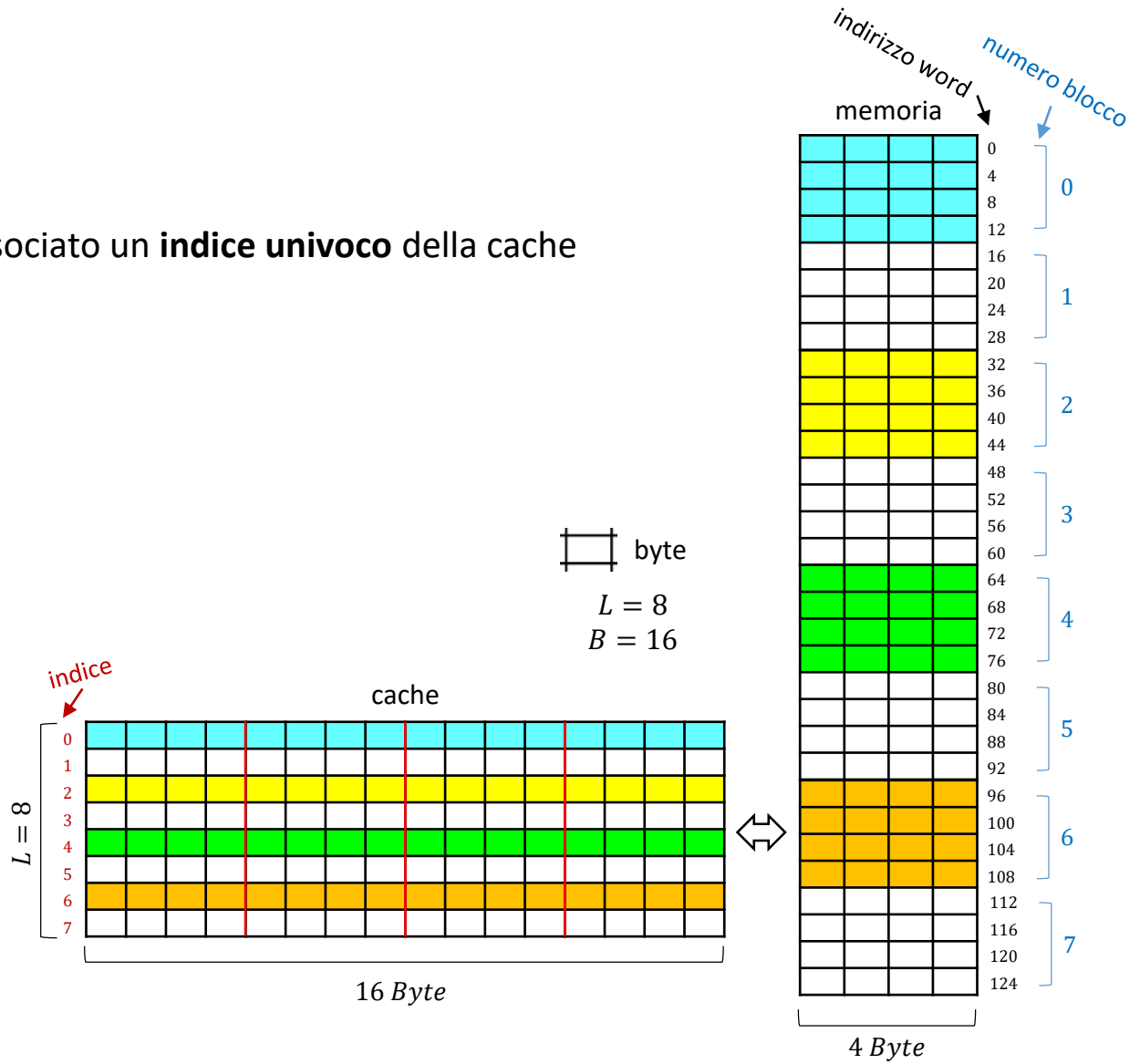
Soluzione

1. Determinare il **numero di blocco** in memoria a partire dall'indirizzo di d:

$$N(d) = div(M(d), B)$$

2. Determinare l'**indice** di cache a cui è assegnato il blocco $N(d)$:

$$I(d) = mod(N(d), L)$$



Mappatura diretta

$N(d)=div(M(d),B)$

$I(d)=mod(N(d),L)$

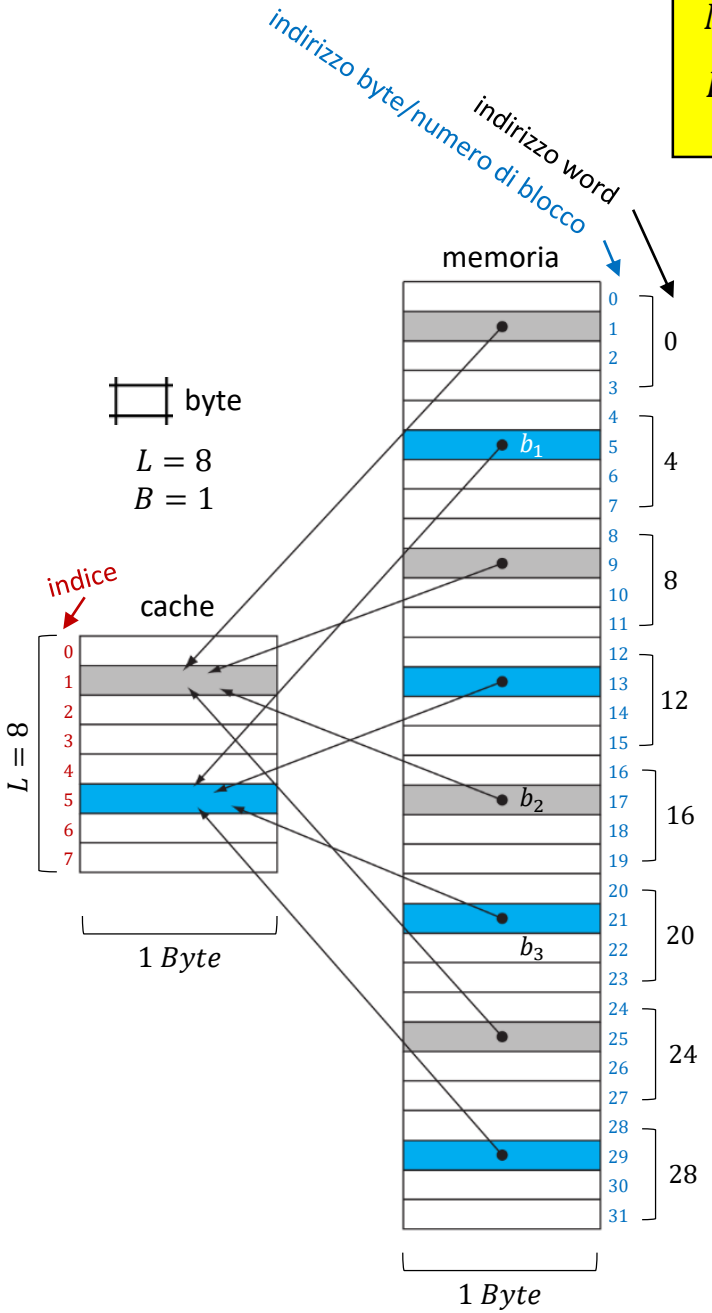
Esempio:

- A quale indice di linea di cache dovrebbero trovarsi i byte b_1, b_2 e b_3?
- **Attenzione!** In questo esempio un blocco contiene un byte (l'elemento indirizzato in memoria), quindi l'indirizzo in memoria coincide con il numero di blocco!

- $M(b_1) = 5$
- $N(b_1) = div(5,1) = 5$
- $I(b_1) = mod(5,8) = 5$

- $M(b_2) = 17$
- $N(b_2) = div(17,1) = 17$
- $I(b_1) = mod(17,8) = 1$

- $M(b_3) = 22$
- $N(b_3) = div(22,1) = 22$
- $I(b_3) = mod(22,8) = 6$

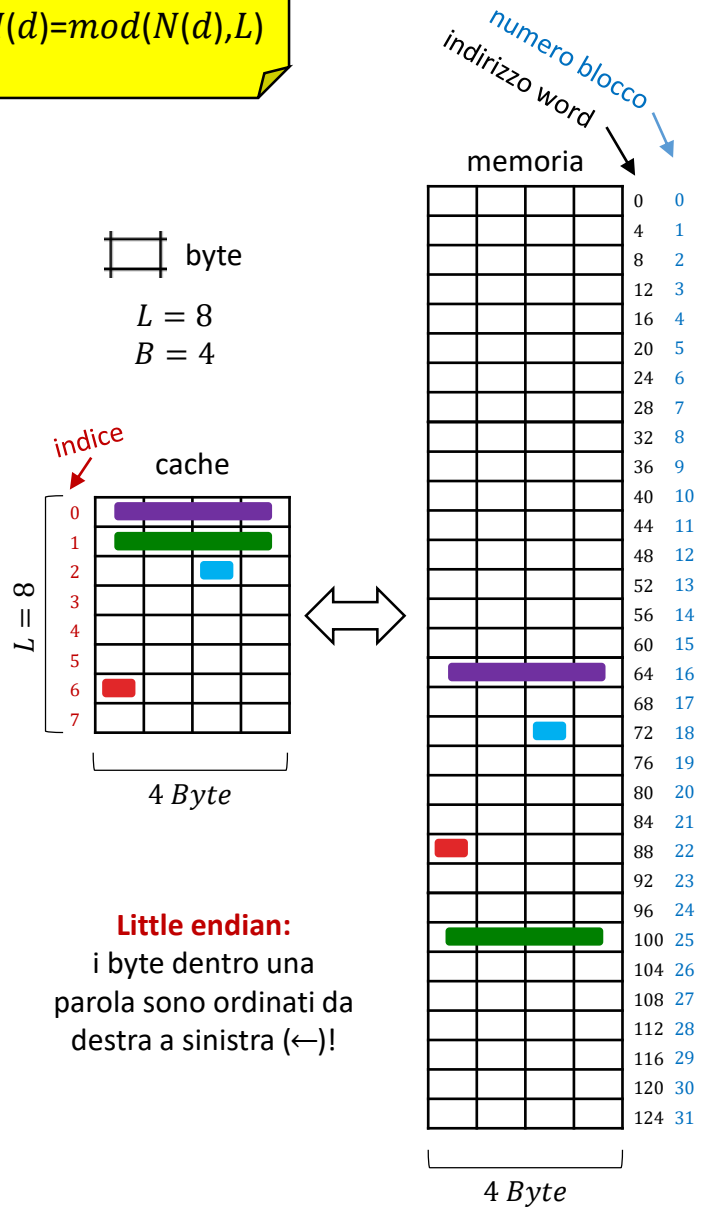


Mappatura diretta

$N(d)=div(M(d),B)$

$I(d)=mod(N(d),L)$

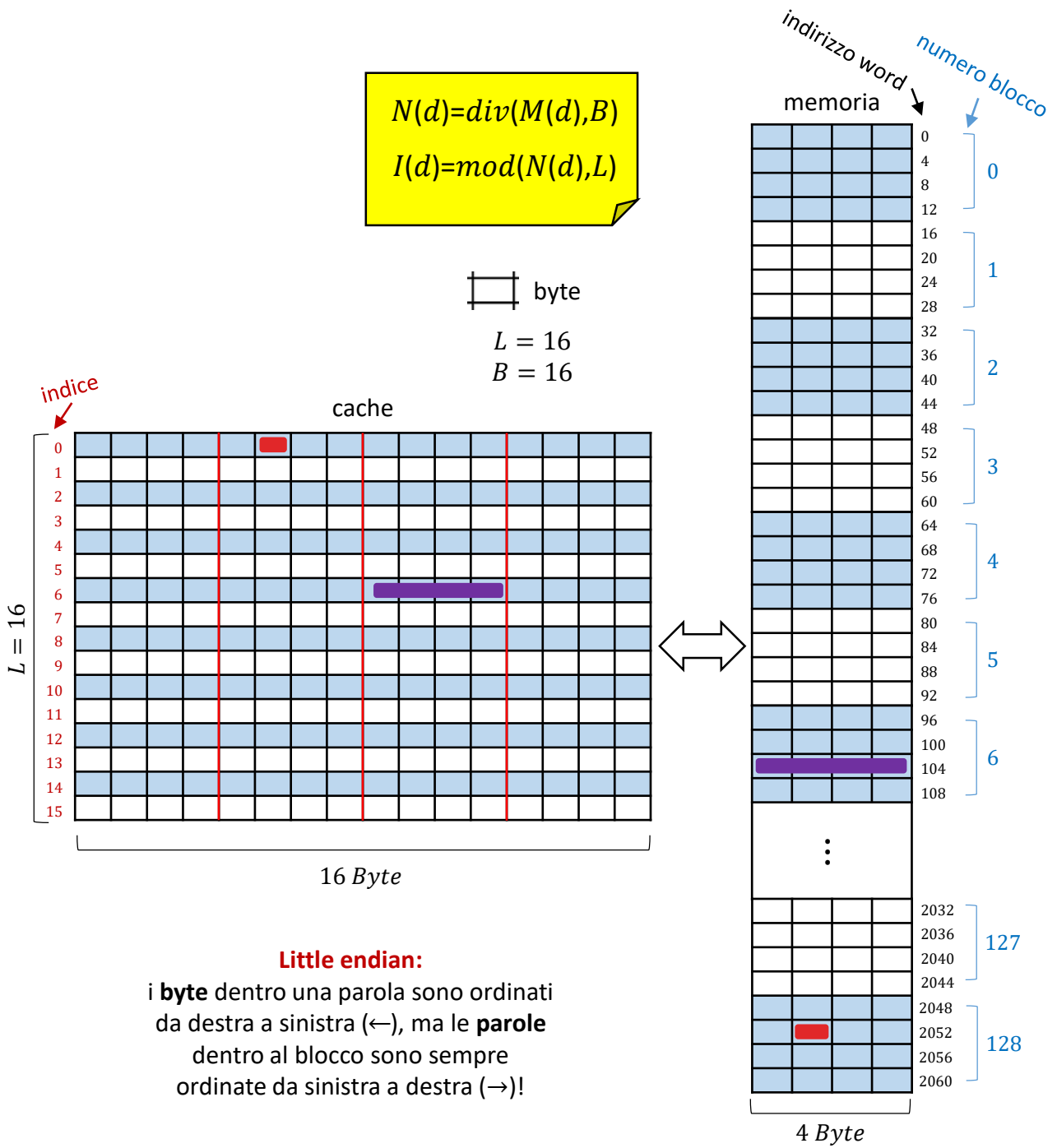
- A quale indice si trova la parola con indirizzo 100?
  - $M(d) = 100$
  - numero di blocco $N(d) = div(100,4) = 25$ (resto 0, primo byte dentro al blocco 25)
  - Indice di cache: $I(d) = mod(25,8) = 1$
- A quale indice si trova **il byte** con indirizzo 73?
  - $M(d) = 73$
  - numero di blocco $N(d) = div(73,4) = 18$ (resto 1, secondo byte dentro al blocco 18)
  - Indice di cache: $I(d) = mod(18,8) = 2$
- A quale indice si trova la parola con indirizzo 64?
  - $M(d) = 64$
  - numero di blocco $N(d) = div(64,4) = 16$ (resto 0, primo byte dentro al blocco 16)
  - Indice di cache: $I(d) = mod(16,8) = 0$
- A quale indice si trova **il byte** con indirizzo 91?
  - $M(d) = 91$
  - numero di blocco $N(d) = div(91,4) = 22$ (resto 3, quarto byte dentro al blocco 22)
  - Indice di cache: $I(d) = mod(22,8) = 6$



Little endian:
i byte dentro una parola sono ordinati da destra a sinistra (←)!

Mappatura diretta

- A quale indice si trova la **parola** con indirizzo 104?
  - $M(d) = 104$
  - numero di blocco $N(d) = div(104,16) = 6$ (resto 8, quindi **nono byte** dentro al blocco 6)
  - Quale parola dentro al blocco? Divido il resto per la dimensione della parola (4 byte): $div(8,4) = 2$ quindi è la **terza parola** nel blocco 6
  - Indice di cache: $I(d) = mod(6,16) = 6$
- A quale indice si trova **il byte** con indirizzo 2054?
  - $M(d) = 2054$, numero di blocco $N(d) = div(2054,16) = 128$ (resto 6, quindi **settimo byte** dentro al blocco 128)
  - Quale parola dentro al blocco? Divido il resto per la dimensione della parola (4 byte): $div(6,4) = 1$ quindi è la **seconda parola** nel blocco 128
  - Indice di cache: $I(d) = mod(128,16) = 0$



Little endian:
i **byte** dentro una parola sono ordinati da destra a sinistra (←), ma le **parole** dentro al blocco sono sempre ordinate da sinistra a destra (→)!

Divisione intera per potenza della base

1. Le cache che consideriamo non hanno dimensioni arbitrarie
   - Il numero di linee è sempre una potenza di 2: $L = 2^k$
   - La dimensione di un blocco è tipicamente definita da un numero intero di parole, se nel blocco ci sono 2^m parole allora $B = 2^{m+2}$, quindi anche B è una potenza di 2
2. I calcoli visti precedentemente sono, quando eseguiti nella CPU, divisioni intere tra **un intero binario senza segno** (unsigned) e **una potenza di 2**

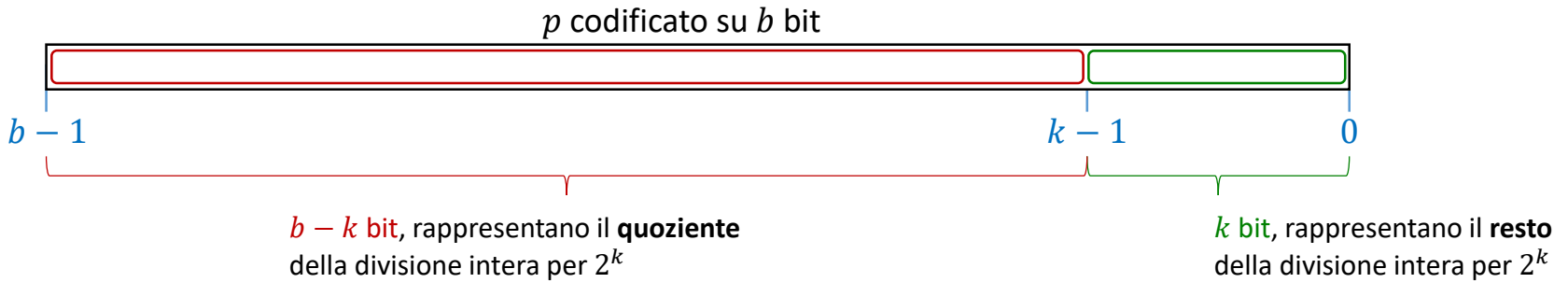
In base **dieci**:

- $div(104,16) = div(104,2^4) = 6$
- $mod(104,16) = mod(104,2^4) = 8$

Rappresentando il dividendo in base **due**:

- $div(1101000,2^4) = \mathbf{110}1000$
- $mod(1101000,2^4) = 1101\mathbf{000}$

- Dato un intero binario senza segno p codificato su b bit, dalla divisione intera tra p e la k-esima potenza di due 2^k si ha che:
  - Il quoziente della divisione è dato dalle $b - k$ cifre **più** significative di p (estraibili con uno shift a destra di k posizioni applicato a p)
  - Il resto della divisione è dato dalle k cifre **meno** significative di p



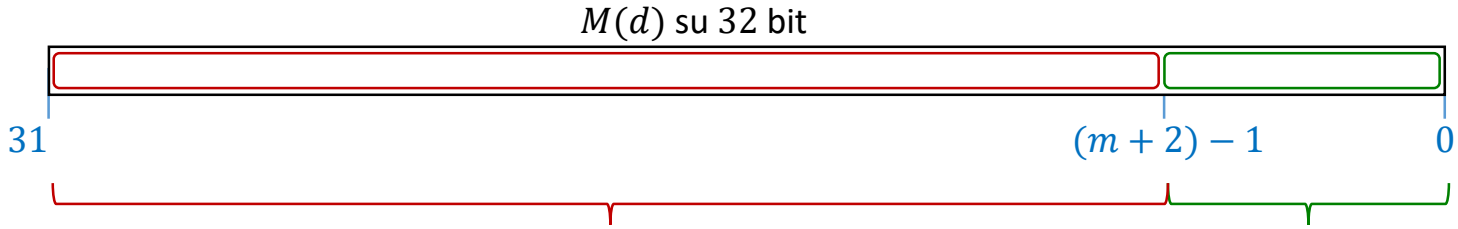
Accesso alla cache

$N(d)=div(M(d),B)$

$I(d)=mod(N(d),L)$

- Sfruttando questa proprietà possiamo implementare in modo semplice l'accesso alla cache con mappatura diretta
- Consideriamo cache con $L = 2^k$ e $B = 2^{m+2}$, come calcoliamo l'indice di linea di un dato d?

Calcolo del numero di blocco

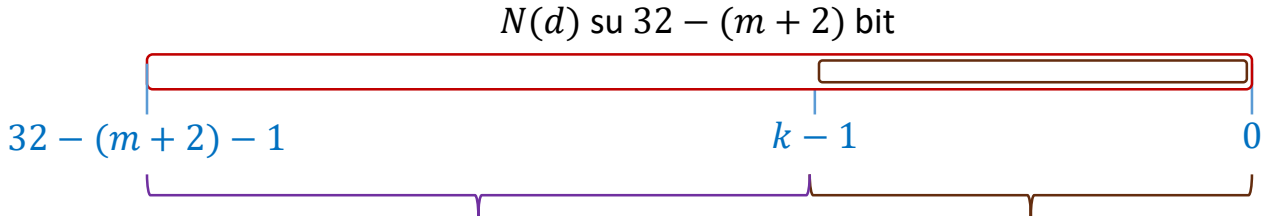


$32-(m+2)$ bit, rappresentano il **quoziente** della divisione intera tra $M(d)$ e $B = 2^{m+2}$, ovvero il **numero di blocco** $N(d)$

$m+2$ bit, rappresentano il **resto** della divisione intera tra $M(d)$ e 2^{m+2}, ovvero … **l'offset all'interno del blocco!**

⇩

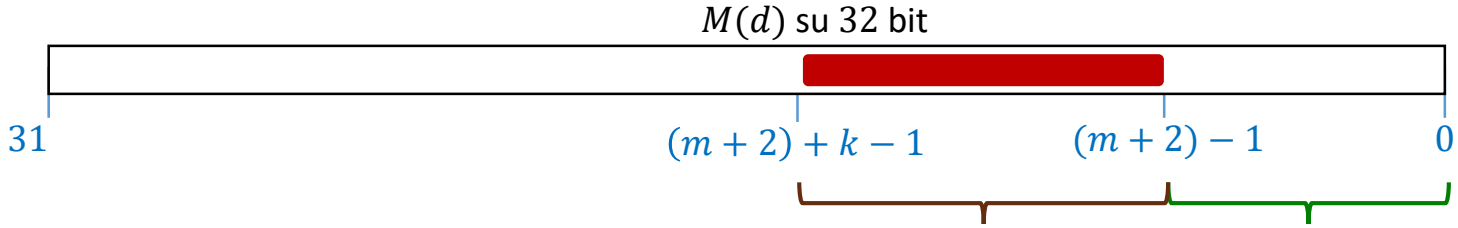
Calcolo dell'indice di cache



k bit, rappresentano il **resto** della divisione intera tra $N(d)$ e 2^k, **l'indice della linea di cache!**

Mappatura diretta

- Ricapitolando, data una cache con L linee ciascuna contenenti blocchi da B byte e dato $M(d)$ (l'indirizzo del byte o della parola a cui si deve accedere):



i successivi $k = \log_2 L$ bit sono l'indice di linea della cache a cui trovare il blocco che contiene d

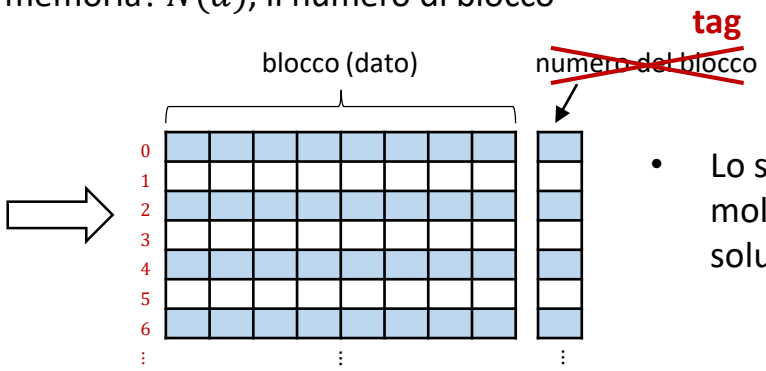
i primi $m + 2 = \log_2 B$ bit sono l'offset che d ha all'interno del blocco che lo contiene: m bit per l'offset di parola dentro al blocco, 2 bit per l'offset di byte dentro alla parola

- Proprietà della mappatura diretta: tutti i blocchi di memoria che sono assegnati alla stessa linea di cache condividono nel proprio indirizzo i bit dalle posizioni $m + 2$ fino a $(m + 2) + k - 1$ (l'indice di linea)
- Corollario: conoscendo l'indice di linea di un blocco $I(d)$, l'offset di parola e l'offset di byte di d, ricavo i primi $m + 2 + k$ bit dell'indirizzo di d in memoria principale
- Queste tre informazioni (indice, offset di parola, offset di byte) costituiscono una singola richiesta alla cache

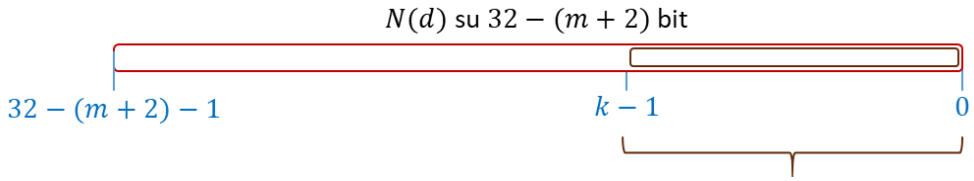
Mappatura diretta

- **Secondo problema**: quando si accede ad un blocco immagazzinato dentro ad una linea di cache, come si può stabilire se è quello effettivamente richiesto visto che quella stessa linea può essere condivisa da diversi blocchi della memoria?
- **Soluzione**: nella linea di cache aggiungo, oltre al blocco stesso, l'informazione per identificarlo
- Cosa identifica in modo univoco un blocco in memoria? $N(d)$, il numero di blocco

- **Idea**: aggiungo il numero di blocco sulla linea di cache, in questo modo so sempre quale effettivo blocco, tra tutti quelli possibili, c'è in quel momento sulla linea



- Lo spazio di memoria nella cache costa molto! Siamo sicuri che questa sia la soluzione più efficiente? **No!** Perché?

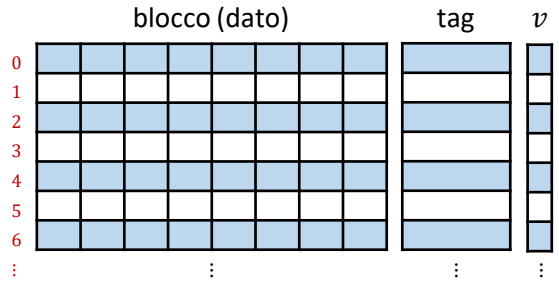


- i k bit meno significativi di $N(d)$ coincidono con l'indice della cache
- Sono implicitamente determinati dalla posizione dalla linea di cache in cui sta quel blocco! Non serve memorizzarli esplicitamente!
- Per identificare il blocco basta memorizzare i $32-(m+2)-k$ bit più significativi del numero di blocco
- Dal corollario precedente sono anche i $32-(m+2)-k$ bit più significativi dell'indirizzo del dato!

- Questi bit aggiuntivi da includere nella linea di cache prendono il nome di **tag**

Mappatura diretta

- **Terzo problema**: il blocco presente in una linea di cache potrebbe non essere valido (per esempio perché la linea non è stata ancora inizializzata oppure il blocco è diventato obsoleto)
- **Soluzione**: nella linea di cache aggiungo, oltre al blocco stesso e al tag, un bit di validità v



Dimensionamento della cache:

- Quanta memoria totale (in byte) serve per una cache con L linee e dimensione di blocco B?

$$D_{tot} = L \times \frac{8B + (32 - \log_2 B - \log_2 L) + 1}{8}$$

- Se consideriamo solo i dati: $D_{data} = L \times B$, di norma si riporta solo questa, ad esempio quando si dice «una cache da $16\ KiB$» si intende una cache dove $D_{data} = 16\ KiB$

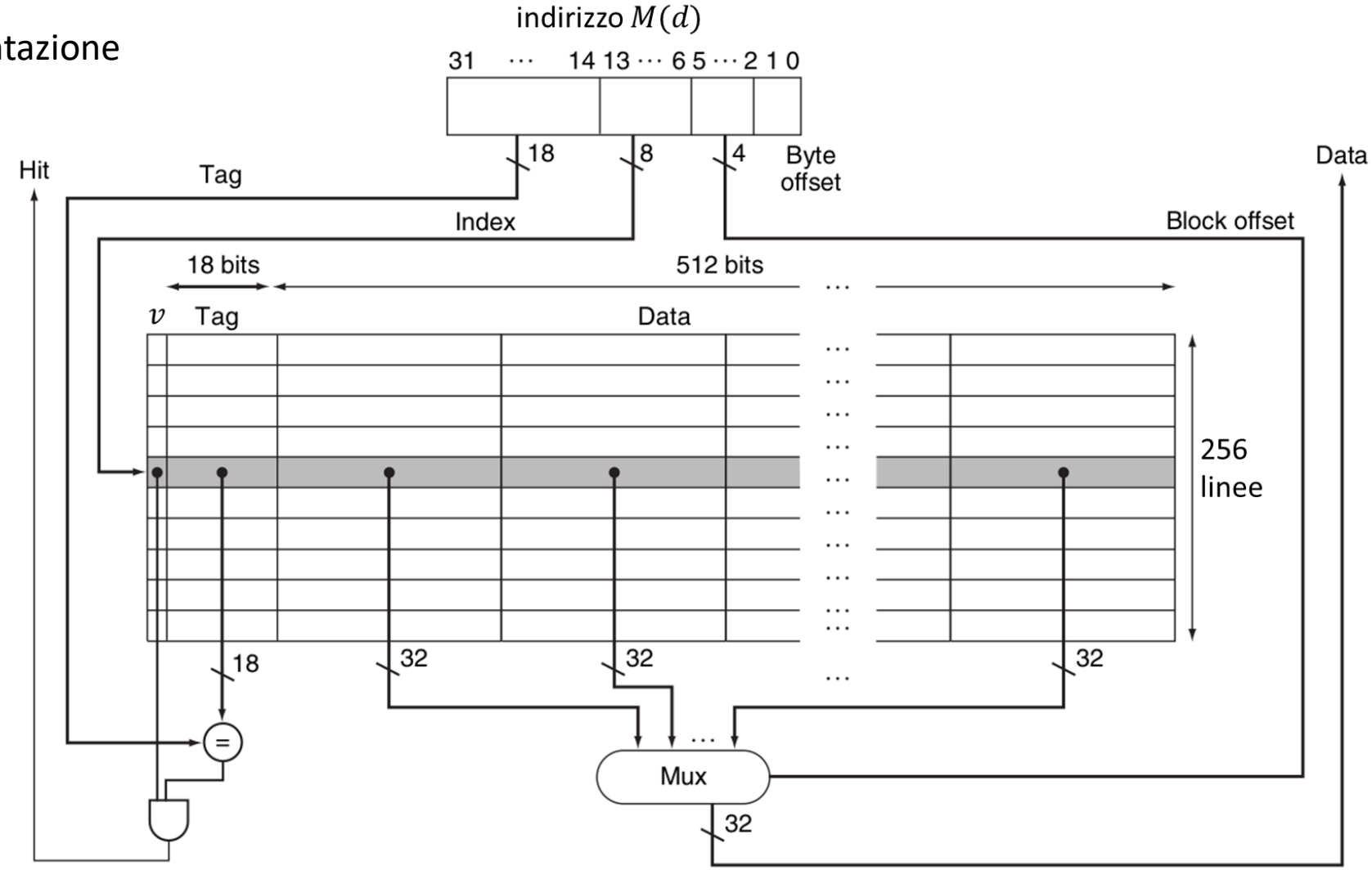
Mappatura diretta

Esercizio: dimensionare e implementare una cache con $D_{data} = 16\ KiB$ e dove ogni blocco contiene 16 parole di memoria

- $D_{data} = 16KiB = (16 \times 2^{10})B = 2^{14}B$
- $B = (16 \times 4)B = 64B = 2^{6}B$ (a destra dell'uguale B indica Bytes)
- $L = \frac{D_{data}}{B} = \frac{2^{14}B}{2^{6}B} = 2^{8} = 256$
- Quanti bit per l'indice di linea? $k = \log_2 L = \log_2 2^{8} = 8$
- Quanti bit per l'offset dentro al blocco? $B = 2^{m+2} \rightarrow m = \log_2 B - 2 = 6 - 2 = 4$
- Quanti bit per il tag? $32 - (m + 2) - k = 32 - 6 - 8 = 18$
- Dimensione totale: $D_{tot} = 256 \times \frac{(8 \times 64 + (32 - 6 - 8) + 1)}{8} = 16992B \cong 16{,}6\ KiB$, servono $608\ B$ in aggiunta alla capacità dati

Mappatura diretta

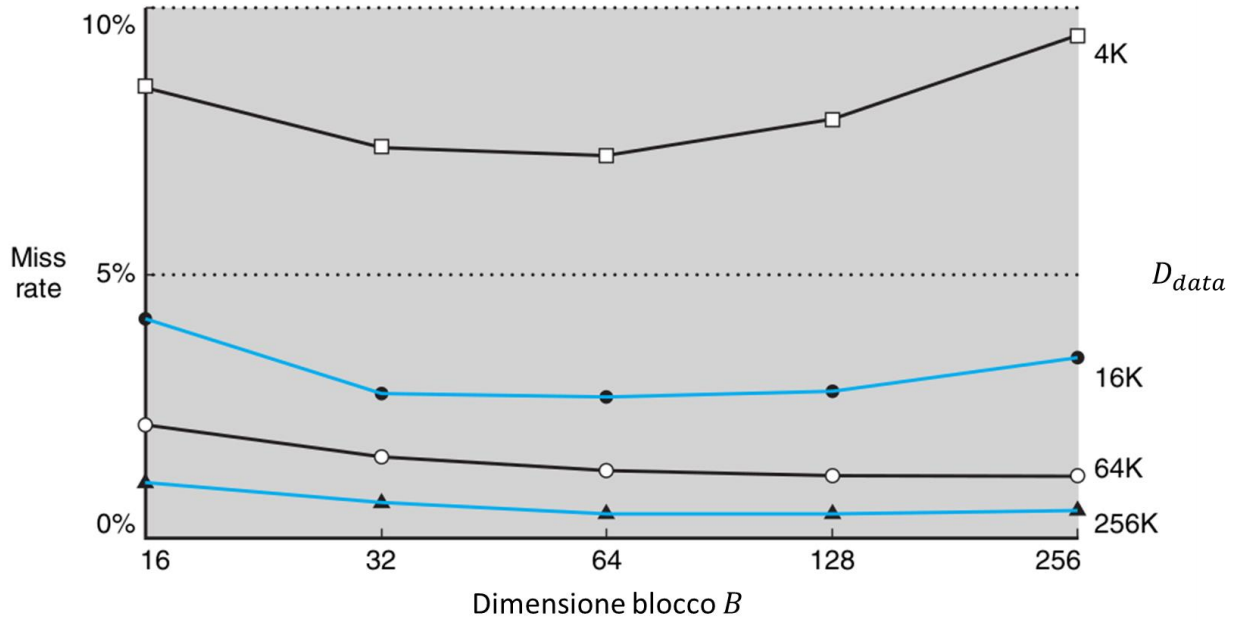
- Implementazione



Dimensione della cache

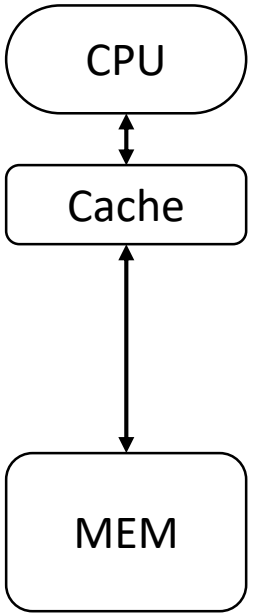
- Come dimensionare una cache?
- Aumentando la dimensione del blocco sfruttiamo meglio la località spaziale ma …
- … se il blocco è troppo grande rispetto alla dimensione della cache (D_{data}) ci sarà molta competizione per le linee di cache tra i vari blocchi

- Questo tradeoff emerge da un'analisi empirica della **miss rate** media in una cache
- Con blocchi più grandi anche la **miss penalty** tende ad aumentare



Gestione delle miss

- Dato l'indirizzo di un dato $M(d)$ sappiamo dove cercarlo in cache e come stabilire se è il dato corretto e se è valido
- Cosa succede quando una delle due condizioni non si verifica? Si assuma che un'istruzione (es. una lw) richieda accesso in lettura ad un dato d



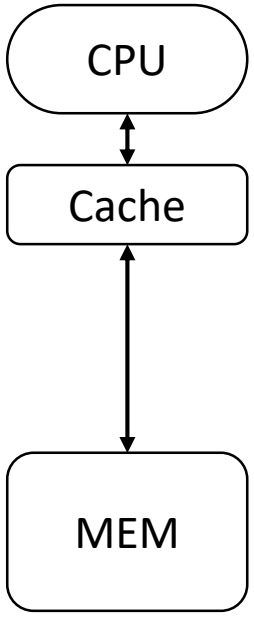
1. Accesso alla cache, linea $I(d)$
2. Check sul bit di validità e confronto tra tag e parte alta di $M(d)$ → **cache miss**
3. La CPU va in stallo
4. Trasferimento dalla Memoria Principale alla cache del blocco $N(d)$ nel campo dati della linea $I(d)$
5. Scrittura della parte alta di $M(d)$ nel campo tag della linea $I(d)$
6. Settaggio a 1 del bit di validità della linea $I(d)$
7. La CPU può ripartire, **cache hit**

Miss penalty

- Se la CPU può eseguire istruzioni fuori ordine, l'attesa può essere occupata con l'esecuzione di altre istruzioni

Accessi in scrittura

- Si assuma che un'istruzione (es. una sw) richieda accesso in **scrittura** ad un dato d
- Il primo step non cambia, è necessario verificare se il blocco $N(d)$ è disponibile in cache oppure no
  - Se è disponibile: **write hit**, in questo caso l'operazione di scrittura viene svolta sulla copia del dato nella cache
  - Se non è disponibile: **write miss**, si gestisce la miss con la procedura vista precedentemente, una volta che il blocco che contiene il dato è stato trasferito in cache si procede come con una write hit



- Nasce il problema della **coerenza della cache**: dopo l'operazione di scrittura il dato in cache è aggiornato, ma la sua versione corrispondente in memoria è obsoleta
- Come ristabilire la coerenza tra cache e memoria principale?
- Schema di tipo **write-through**
  - quando il dato viene scritto in cache, viene subito scritto anche in memoria
  - si può usare un write buffer per migliorare le prestazioni, ma se si riempie serve uno stallo
- Schema di tipo **write-back**
  - le operazioni di scrittura avvengono solo in cache
  - quando nella cache il blocco viene rimpiazzato, prima di sovrascriverlo si aggiorna la sua versione in memoria

Cache associative

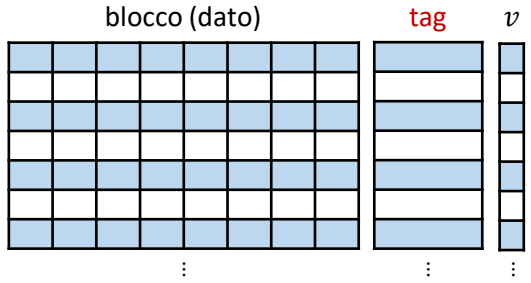
- Nelle cache a mappatura diretta il dato d può essere immagazzinato unicamente nella linea $I(d)$ della cache
- Vantaggio: le operazioni di accesso sono semplici
- Svantaggio: poca flessibilità nell'uso efficiente delle memoria
- **Esempio**: un programma che induce, su alcune linee di cache, una forte competizione tra i blocchi mappati su di essi mentre altre linee di cache sono molto poco utilizzate
- Sarebbe interessante poter rimuovere il vincolo di mappatura e dare ad un blocco di memoria **più di una alternativa** rispetto alla linea di cache a cui essere destinato, in modo da bilanciare la competizione

un dato può trovarsi
su **una sola linea** della cache

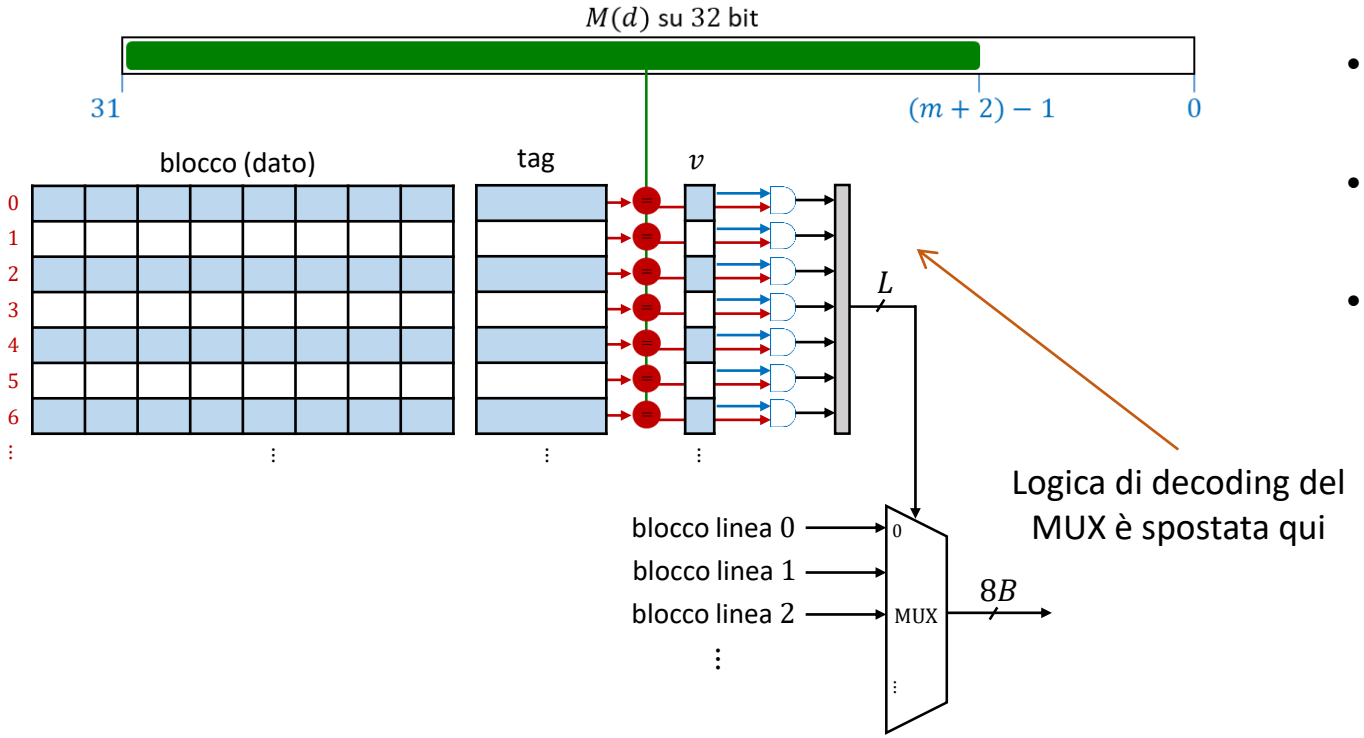
un dato può trovarsi
su **una qualsiasi linea** della cache

Cache associative

- Perché si usa l'aggettivo «*associativo*»?
- In una memoria classica l'accesso al contenuto (l'**informazione**) avviene tramite l'uso di un indirizzo (una **meta-informazione**) che identifica univocamente il punto a cui accedere
- Le cache a mappatura diretta possono funzionare in questo modo perché un dato ha un solo punto possibile (la linea assegnata) a cui essere acceduto: determinando l'indice di linea (meta-informazione) accedo al dato (informazione)
- Ma come si opera l'accesso se per un dato ci sono **più indirizzi possibili** a cui potrebbe trovarsi?
- Idea: provo tutti gli indirizzi possibili
  - Non una buona idea: nelle cache fully-associative significa, nel caso pessimo, controllare ogni linea di cache
- Idea migliore: identificare il punto di accesso tramite il suo contenuto, non usando la meta-informazione ma l'informazione stessa!
- **Memoria associativa**: l'accesso non avviene tramite un indirizzo, ma con una parte del contenuto del dato che lo identifica univocamente (un valore detto **chiave**)
- **Domanda**: quale parte del **contenuto** di una linea di cache identifica univocamente il blocco memorizzato in quella linea?
- **Risposta**: **il tag**
- **Attenzione**
  - non esiste l'indice di linea, è come se ci fosse un'unica linea, $L = 1$, quindi $k \leftarrow 0$
  - il tag è quindi dato da tutti i $32 - (m + 2)$ bit del numero di blocco



Cache fully-associative



- Il tag viene confrontato su tutte le linee di cache, **darà risultato positivo su al più una linea**
- il bit del test di uguaglianza sul tag va in AND con il bit di validità della linea corrispondente
- Il MUX seleziona per la porta di lettura la linea di cache su cui c'è stata la hit

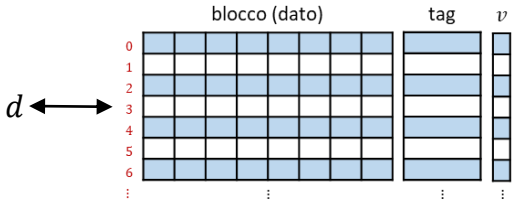
- Questo tipo di memoria costa!
  - serve la logica di test (tag e validità) replicata su ogni linea, più la logica di selezione
  - Il tag ha un numero più alto di bit ($k = 0$) quindi ogni linea di cache contiene più informazioni

Cache fully-associative

- **Esempio:** quanti dati può contenere (D_{data}) una cache completamente associativa dove il blocco contiene 16 parole di memoria e $D_{tot} = 33.6875\ KiB$?
- $L = 1$, quindi $k = 0$
- $m + 2 = \log_2(4 \times 16) = 6$ (4 bit per offset di parola, 2 bit per offset di byte)
- Il tag è su $32 - 6 = 26$ bit
- Per ogni blocco servono quindi 539 bit:
  - $4 \times 16 \times 8 = 512$ bit per il dato
  - 26 bit per il tag
  - 1 bit di validità
- Capacità in numero di blocchi: $\frac{33.6875 \times 2^{10+3}}{539} = 512$
- Capacità dei soli dati $D_{data} = 512 \times 16 \times 4 = 32768\ B = 32\ KiB$

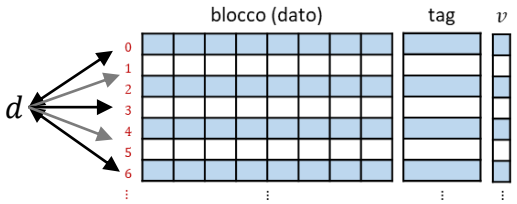
Cache set-associative

Cache Direct-Mapped



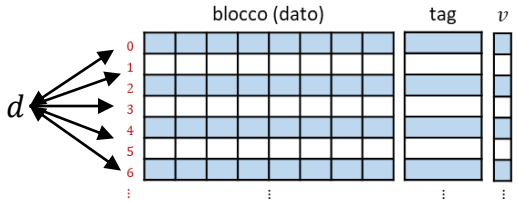
un dato può trovarsi
su **una sola linea** della cache

Cache Set-Associative



la cache è partizionata in insiemi (**set**), ciascun set ha posto per n blocchi, un dato è univocamente associato ad un set, ma al suo interno può prendere uno qualsiasi degli n posti

Cache Fully-Associative



un dato può trovarsi
su **una qualsiasi linea** della cache

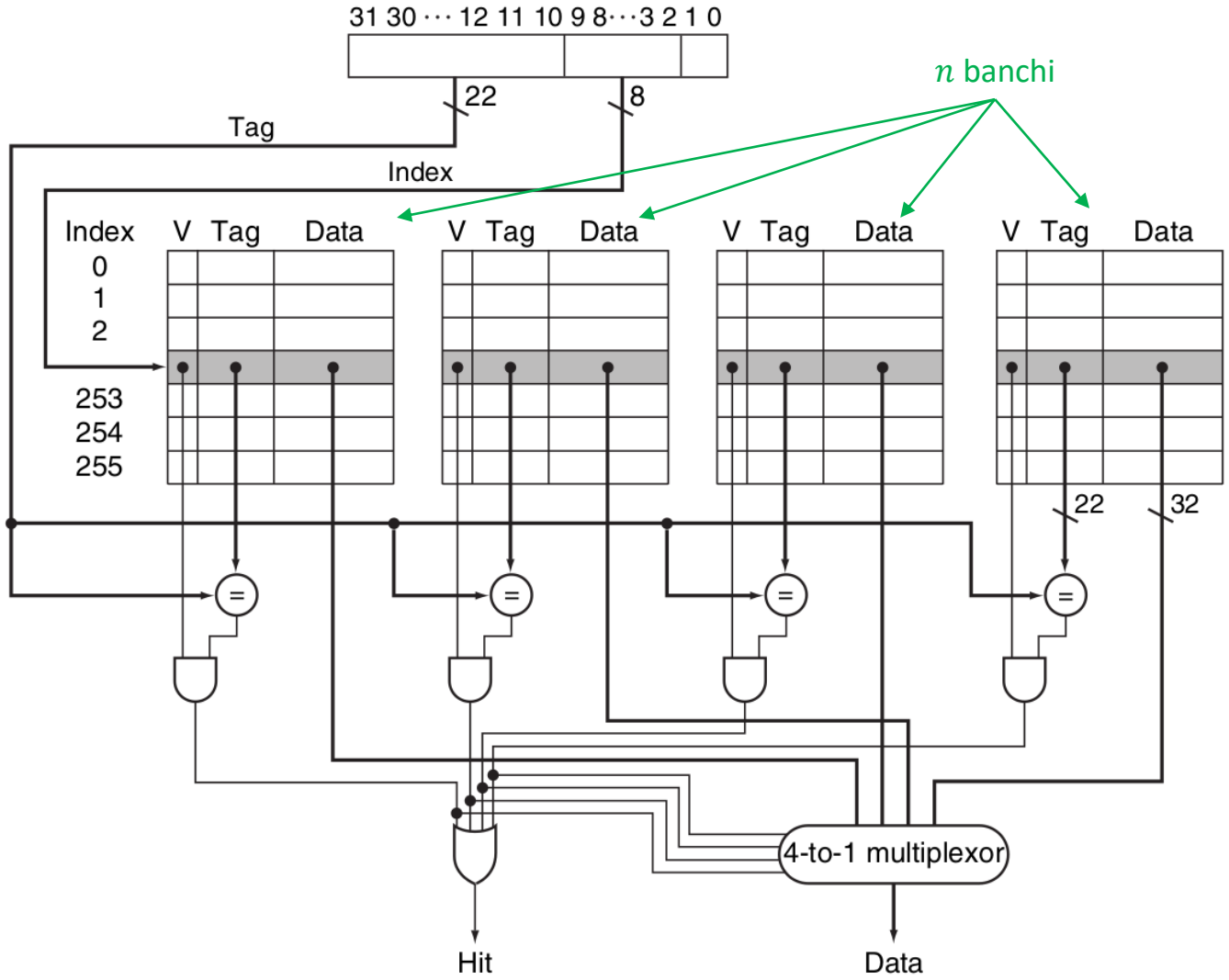
- Il set può essere pensato come ad una generalizzazione della linea, ammettendo che abbia un numero generico n di posti per contenere blocchi
- Per indicare il numero di set continuiamo ad utilizzare il parametro L
- **Direct-mapped**: $L = \frac{D_{data}}{B}$ set (linee) da $n = 1$ posto, chi è assegnato a quel set (linea) può occupare solo quel posto
- **Fully-associative**: $L = 1$ set (linee) da $n = \frac{D_{data}}{B}$ posti, assegnazione di set implicita, i blocchi possono occupare qualsiasi posto disponibile in cache
- **Set-Associative a n vie**: $L = \frac{D_{data}}{n \times B}$ da n posti, chi è assegnato ad un set (linea) può occupare uno qualsiasi degli n posti (se n è pari al numero totale di blocchi memorizzabili in cache, $n = \frac{D_{data}}{B}$, allora la cache diventa completamente associativa, se invece $n = 1$ allora è direct-mapped)
- Il parametro $n \in \left[1, \frac{D_{data}}{B}\right]$, numero di posti, viene più spesso chiamato numero di **banchi** della cache o numero di **vie**
- Dobbiamo generalizzare la formula di D_{data}: $D_{data} = L \times n \times B$

Cache set-associative

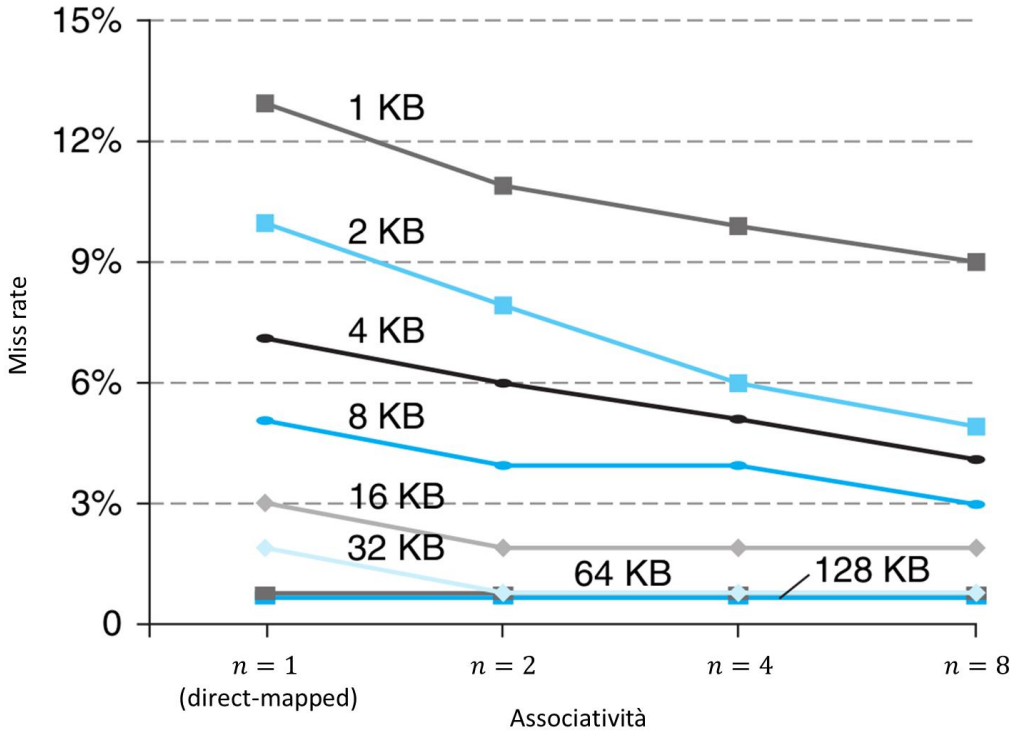
- L'accesso alle cache set-associative combina i due meccanismi che abbiamo visto fino ad ora:
  - Determinazione del set (linea): metodo direct-mapped con calcolo dell'indice
  - Determinazione del banco (posto) all'interno del set: metodo fully-associative, con test di uguaglianza del tag e check su bit di validità

Esempio:

- Cache associativa a 4 vie da $4\ KiB$ e blocchi da 1 parola
- numero di set $L = \frac{4 \times 2^{10}}{4 \times 4} = 256$, quindi $k = \log_2 256 = 8$ bit di indice
- $m + 2 = \log_2 B = 2$, quindi $m = 0$ (c'è solo offset di byte perché un blocco coincide con una parola)
- Tag su $32 - 2 - 8 = 22$ bit



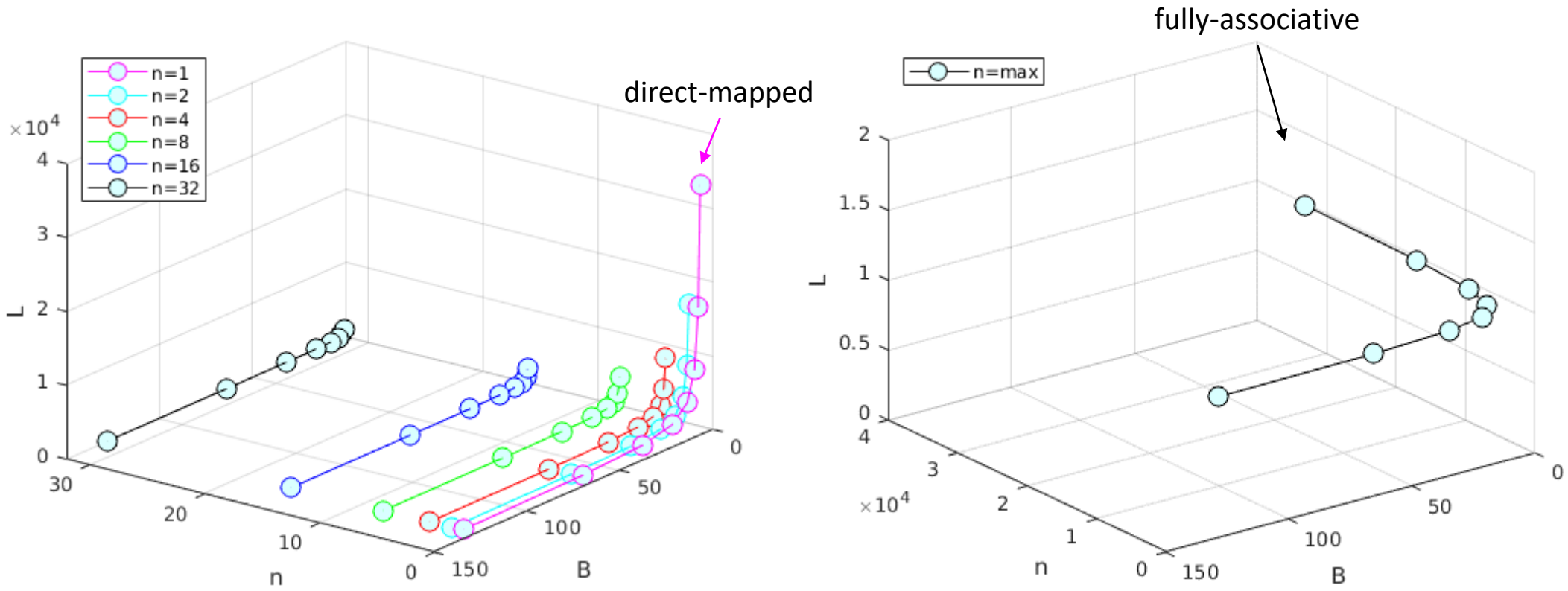
Grado di associatività



- Aumentare il numero di vie (n) comporta, in generale un abbassamento della miss rate
- I benefici sono decrescenti (ad es. passare da 1 a 2 vie introduce una maggiore diminuzione della miss rate rispetto a passare da 2 a 4)
- Se la miss rate è già bassa i benefici introdotti sono limitati

Numero dei set

- Cache da $32\ KiB$



Accessi direct-mapped

- Cache da 16 B, direct-mapped con $L = 4$, $B = 4$
- $k = 2, m = 0$, tag su 28 bit

Memoria

48	63
⋮	…
28	55
⋮	…
16	20

$M(d)$	$M(d)_2$	$N(d)$	$I(d)$	tag_2
16	00010000	4	0	0001
28	00011100	7	3	0001
48	00110000	12	0	0011

lw $s0 16($zero) **miss!**
lw $s1 48($zero) **miss!**
lw $s4 16($zero) **miss!**
lw $s3 28($zero) **miss!**
lw $s1 48($zero) **miss!**

block	tag	v
–	–	0
–	–	0
–	–	0
–	–	0

⇨

block	tag	v
20	1	1
–	–	0
–	–	0
–	–	0

⇨

block	tag	v
63	3	1
–	–	0
–	–	0
–	–	0

⇨

block	tag	v
20	1	1
–	–	0
–	–	0
–	–	0

⇨

block	tag	v
20	1	1
–	–	0
–	–	0
55	1	1

⇨

block	tag	v
63	3	1
–	–	0
–	–	0
55	1	1

i bit alti non riportati nelle codifiche binarie di indirizzi e tag sono da considerarsi implicitamente pari a 0 (vale in questa slide e nelle due successive)

Accessi set-associative

- Cache da 16 B, set-associative a 2 vie, $L = 2$, $B = 4$ (due set da 2 posti)
- $k = 1, m = 0$, tag su 29 bit

Memoria

48	63
⋮	…
28	55
⋮	…
16	20

$M(d)$	$M(d)_2$	$N(d)$	$I(d)$	tag_2
16	00010000	4	0	00010
28	00011100	7	1	00011
48	00110000	12	0	00110

lw $s0 16($zero) **miss!**
lw $s1 48($zero) **miss!**
lw $s4 16($zero) **hit!**
lw $s3 28($zero) **miss!**
lw $s1 48($zero) **hit!**

block	tag	v	block	tag	v
–	–	0	–	–	0
–	–	0	–	–	0

⇨

block	tag	v	block	tag	v
20	00010	1	–	–	0
–	–	0	–	–	0

⇨

block	tag	v	block	tag	v
20	00010	1	63	00110	1
–	–	0	–	–	0

⇨

block	tag	v	block	tag	v
20	00010	1	63	00110	1
–	–	0	–	–	0

⇨

block	tag	v	block	tag	v
20	00010	1	63	00110	1
55	00011	1	–	–	0

⇨

block	tag	v	block	tag	v
20	00010	1	63	00110	1
55	00011	1	–	–	0

Accessi fully-associative

- Cache da 16 B, fully-associative, $L = 1$, $B = 4$ (unico set da 4 posti)
- $k = 0$, $m = 0$, tag su 30 bit

Memoria

48	63
⋮	…
28	55
⋮	…
16	20

lw \$s0 16(\$zero) miss!
lw \$s1 48(\$zero) miss!
lw \$s4 16(\$zero) hit!
lw \$s3 28(\$zero) miss!
lw \$s1 48(\$zero) hit!

$M(d)$	$M(d)_2$	$N(d)$	$I(d)$	tag_2
16	00010000	4	–	000100
28	00011100	7	–	000111
48	00110000	12	–	001100

block	tag	v	block	tag	v	block	tag	v	block	tag	v
–	–	0	–	–	0	–	–	0	–	–	0

block	tag	v	block	tag	v	block	tag	v	block	tag	v
20	000100	1	–	–	0	–	–	0	–	–	0

block	tag	v	block	tag	v	block	tag	v	block	tag	v
20	000100	0	63	001100	1	–	–	0	–	–	0

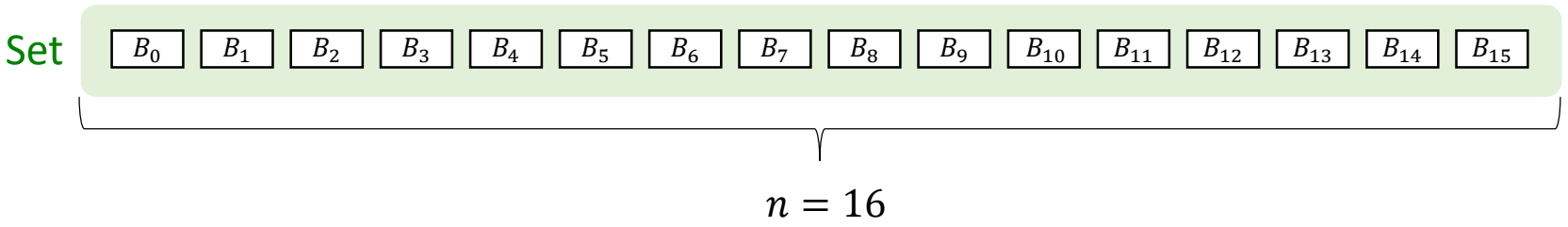
block	tag	v	block	tag	v	block	tag	v	block	tag	v
20	000100	0	63	001100	1	–	–	0	–	–	0

block	tag	v	block	tag	v	block	tag	v	block	tag	v
20	000100	0	63	001100	1	55	000111	1	–	–	0

block	tag	v	block	tag	v	block	tag	v	block	tag	v
20	000100	0	63	001100	1	55	000111	1	–	–	0

Sostituzione dei blocchi

- Se $n > 1$ si pone il problema di scegliere in quale degli n posti sovrascrivere il blocco a fronte di una miss

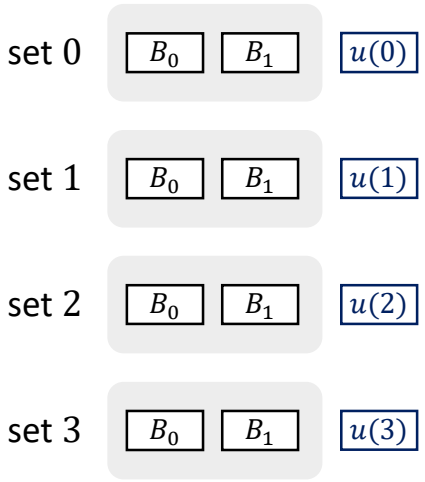


Quando un blocco viene trasferito dentro il set, quale dei 16 bocchi presenti viene sovrascritto?

- La politica con cui viene presa questa decisione impatta fortemente sulla miss rate
- Strategie con cui fare la sostituzione
  - **Random**: soluzione semplice da implementare, con alti gradi di associatività può funzionare bene
  - **Least-Recently Used (LRU):** si sostituisce il blocco che non viene acceduto dal maggior tempo, tra quelli correntemente nel set
  - **Pseudo-LRU**: LRU approssimato

Sostituzione dei blocchi

- Come si implementa la strategia di tipo LRU? Bisogna tenere traccia, in ogni set, di quale è il blocco non usato (letto o scritto) da più tempo
- Questa operazione è semplice da implementare con bassa associatività (2 o 4), ma diventa significativamente più complicata quando il numero di vie aumenta
- Approccio basato sugli **use bits**
- **Esempio** con $n = 2$, ogni set s ha un singolo use bit $u(s)$



$$u(s) = \begin{cases} 1, \text{ultimo accesso avvenuto in } B_1 \\ 0, \text{ultimo accesso avvenuto in } B_0 \end{cases}$$

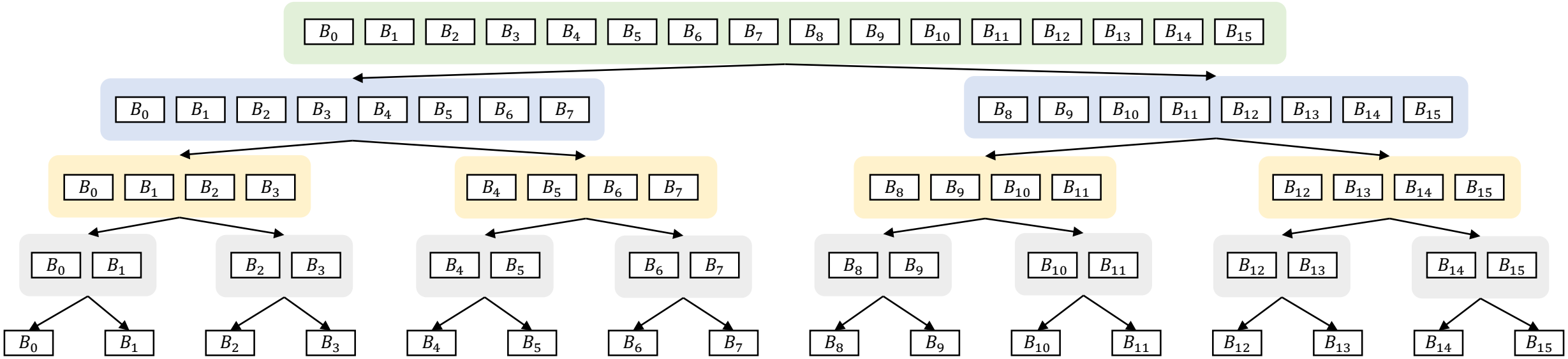
Controllando lo use bit si può stabilire dove sovrascrivere nel set s:

- se $u(s) = 1$ sovrascrivo B_0
- se $u(s) = 0$ sovrascrivo B_1

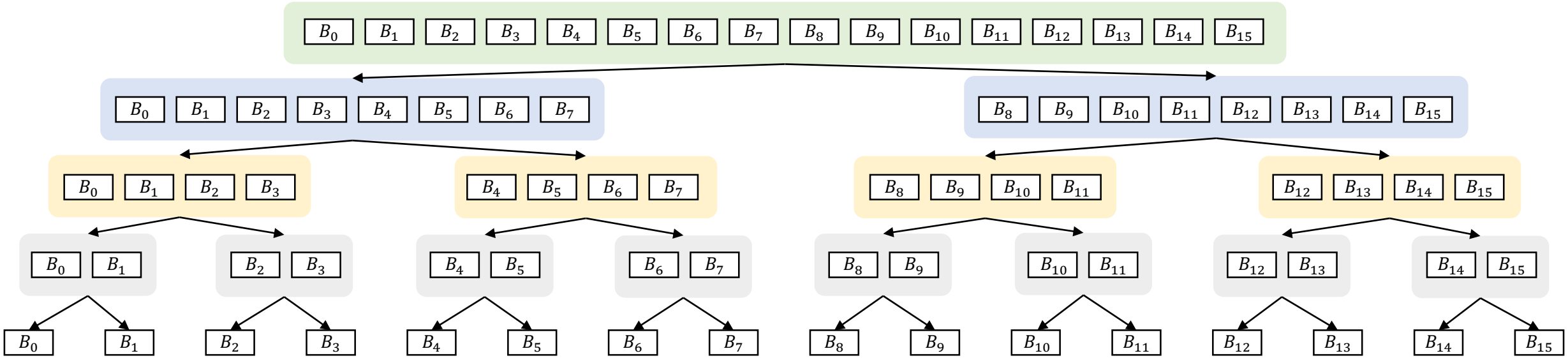
- Questo caso, con $n = 2$, è molto semplice perché, in ogni set, l'ultimo accesso implicitamente localizza la prossima eventuale sovrascrittura
- Conoscere il blocco acceduto più recentemente equivale a conoscere quello non acceduto da più tempo (l'altro)
- Se ci sono più di 2 blocchi in un set le cose si complicano: quando accedo ad un blocco B_i quale altro blocco diventa quello non usato da più tempo?

Sostituzione dei blocchi

- Per ovviare al problema di dover tener traccia in ogni set dello storico degli accessi si possono mettere in campo metodi approssimati
- Un metodo approssimato cerca di identificare quale sia il blocco LRU usando una euristica (un principio pratico), senza garantire di identificarlo correttamente
- **Pseudo-LRU** basato su **albero di ricerca binario**: suddividere i blocchi di un set seguendo la struttura di un Binary Search Tree (BST)



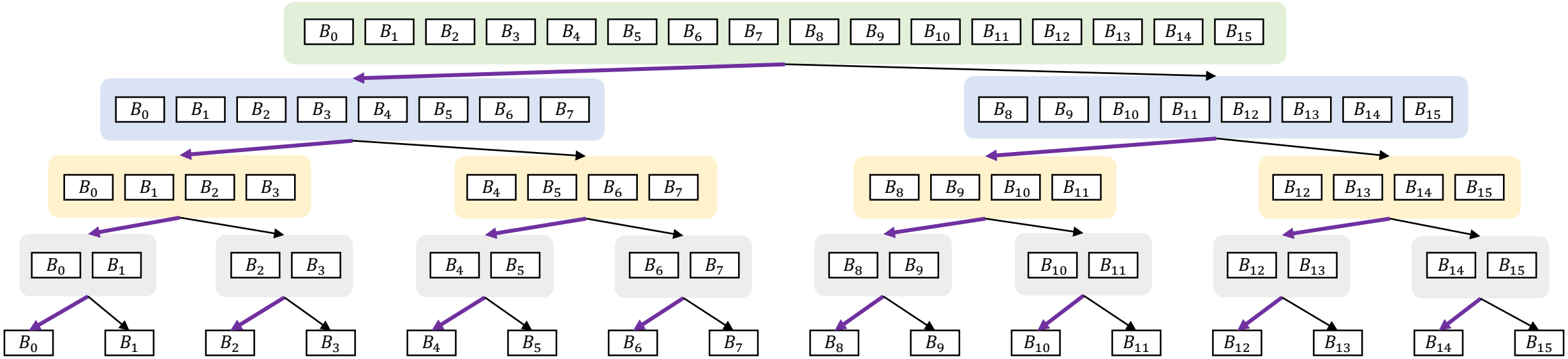
Pseudo-LRU basato su BST



- Un accesso ad un blocco del set è associato ad un ramo dell'albero: un percorso che va dal nodo radice ad un nodo foglia
- Ad ogni nodo lungo il percorso, bisogna prendere una decisione su una biforcazione: sinistra o destra?
- Immaginiamo di allocare 1 bit su ogni biforcazione: 0 significa sinistra, 1 significa destra (con n vie servono $n - 1$ bit)
- Ad ogni accesso, per ogni biforcazione attraversata da quell'accesso:
  - se si è andati a sinistra, si setta il bit a 1 (per suggerire che per trovare LRU si dovrà andare dall'altra parte, a **destra**)
  - se si è andati a destra, si setta il bit a 0 (per suggerire che per trovare LRU si dovrà andare dall'altra parte , a **sinistra**)

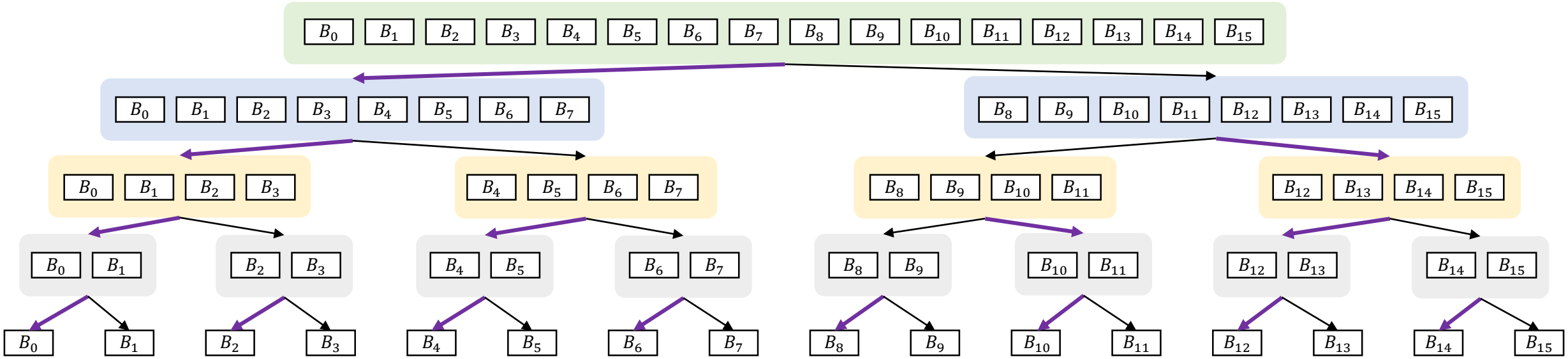
Pseudo-LRU esempio

- Inizialmente tutti i bit delle biforcazioni sono **settati** a 0



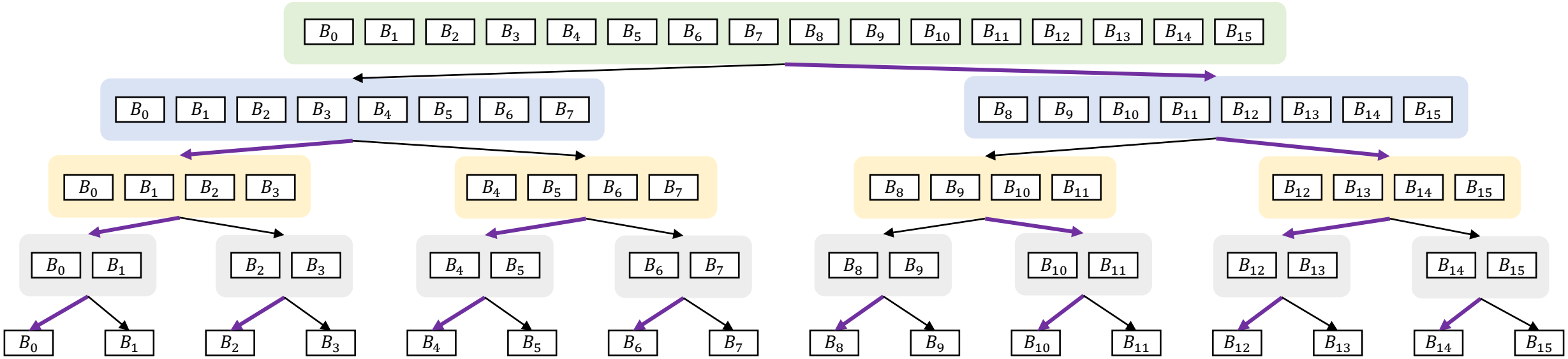
Pseudo-LRU esempio

- Inizialmente tutti i bit delle biforcazioni sono **settati** a 0
- Lettura blocco B_9



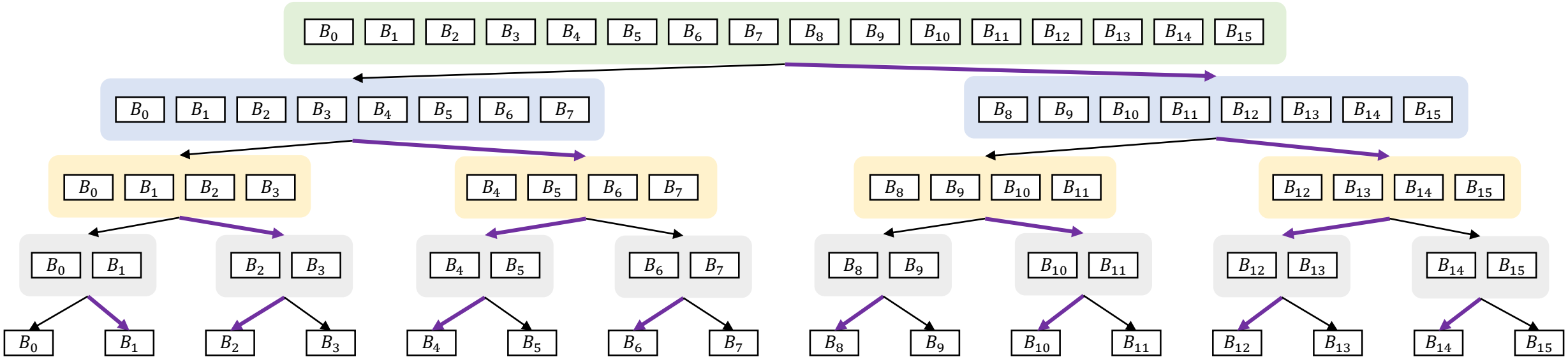
Pseudo-LRU esempio

- Inizialmente tutti i bit delle biforcazioni sono **settati** a 0
- Lettura blocco B_9
- Lettura blocco B_7



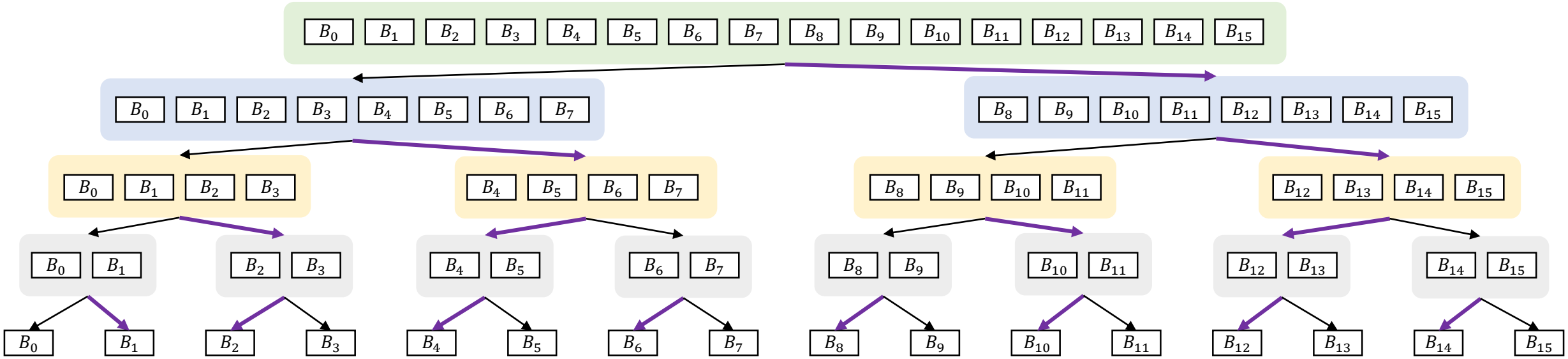
Pseudo-LRU esempio

- Inizialmente tutti i bit delle biforcazioni sono **settati** a 0
- Lettura blocco B_9
- Lettura blocco B_7
- Lettura blocco B_0



Pseudo-LRU esempio

- Inizialmente tutti i bit delle biforcazioni sono **settati** a 0
- Lettura blocco B_9
- Lettura blocco B_7
- Lettura blocco B_0
- Scrittura blocco B_{new}, chi sovrascrivo?
- Riposta: B_{12}



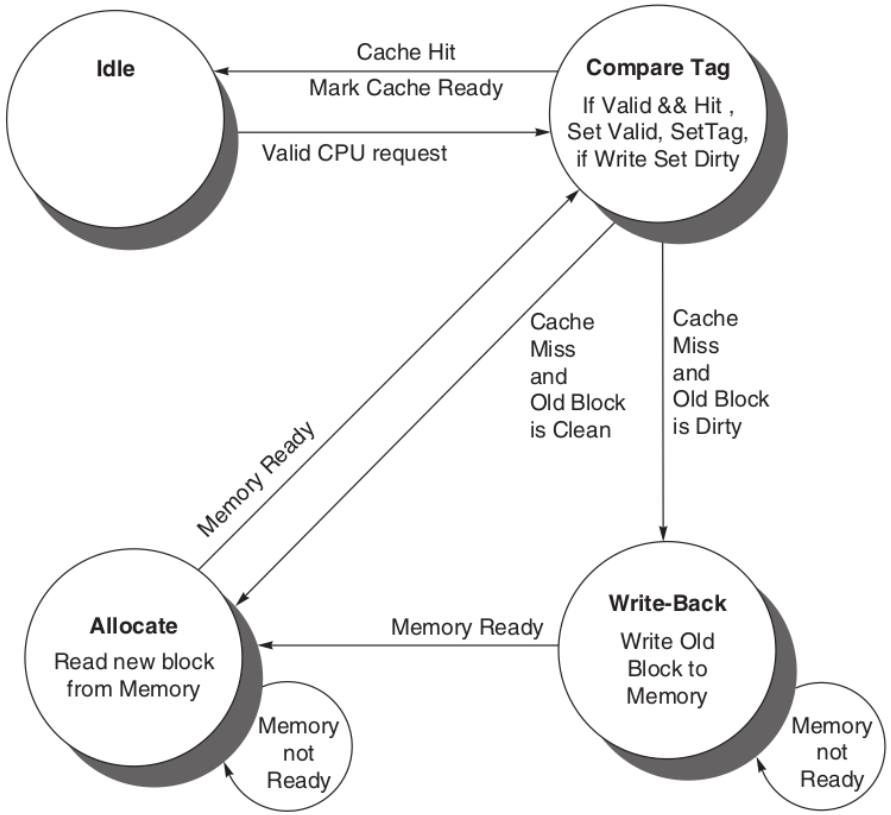
Pseudo-LRU basato su MRU

- Approccio sempre approssimato ma duale: **MRU**, Most Recently Used bit
- Ad ogni blocco i dentro il set associo un bit, l'MRU, U_i
- Ogni volta che si accede a B_i, si setta $U_i \leftarrow 0$
  - se era l'ultimo bit ad essere precedentemente a 1 sul set, si settano a 1 tutti gli altri
- La sovrascrittura avviene nel blocco B_j dove $j = argmin_{j|U_j=1}\{j\}$ (il blocco con ID più basso che ha MRU a 1)

- Con n vie servono n bit, per ogni set

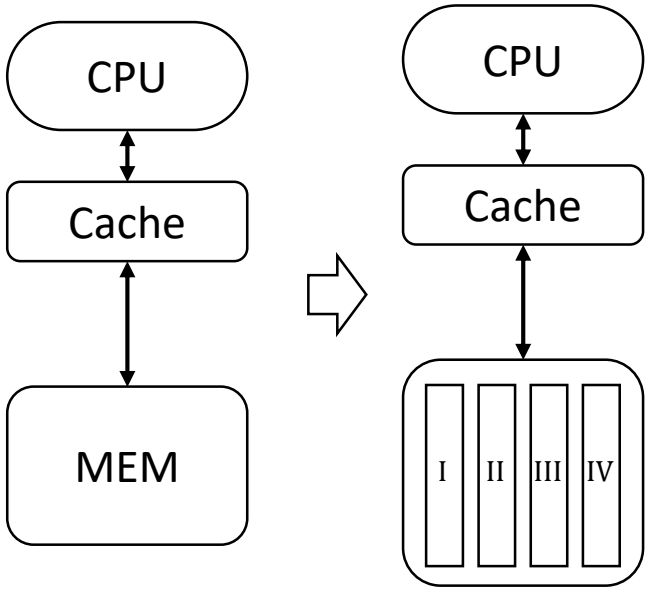
Gestione della cache

- Come sono implementate la gestione delle richieste alla cache che comprendono anche queste policy di sostituzione?
- **FSM** con stati di
  - attesa (idle)
  - verifica di hit/miss (compare tag)
  - scrittura di un blocco dalla cache alla memoria (coerenza, write-back)
  - trasferimento di un blocco dalla memoria alla cache (allocate)

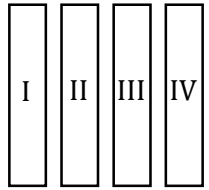


Miglioramento della miss penalty

- Abbiamo visto che aumentando la dimensione del blocco (che deve essere almeno di 1 parola per poter sfruttare almeno il principio di località spaziale) diminuisce la miss rate, ma la miss penalty, anche se pagata meno frequentemente, aumenta
- Perché? Blocchi più grandi richiedono di trasferire un maggior numero di informazioni tra memoria e cache
- Una tecnica per mitigare questo problema si chiama **interleaving**



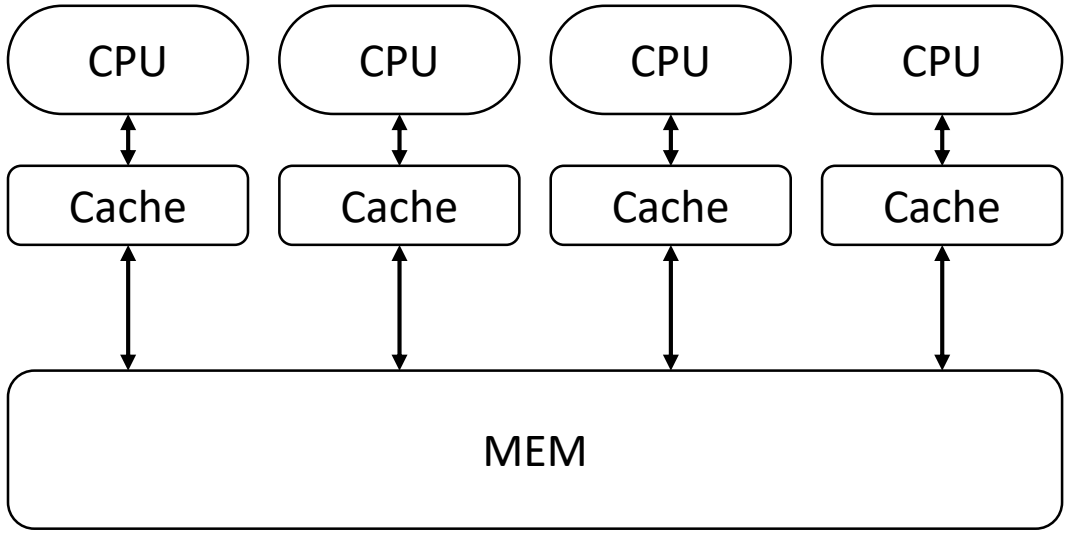
- La memoria viene organizzata in **banchi** che possono lavorare il parallelo
- Grazie ad un principio analogo a quello della pipeline, il throughput delle richieste di accesso aumenta (attenzione! il bus rimane lo stesso, può comunque trasferire una parola per volta)



- Una seconda tecnica è quella di **aumentare l'ampiezza del bus** e consentire trasferimenti di più parole per volta

Sistemi multi-processore

- In un sistema multi-processore abbiamo più CPU (core) che lavorano in parallelo
- Ogni CPU ha una propria cache, ma la memoria principale della macchina è sempre una sola
- Il problema della cache coherence si complica: un accesso in scrittura su una CPU genera una incoerenza che indirettamente coinvolge tutte le altre



Possibili tecniche:

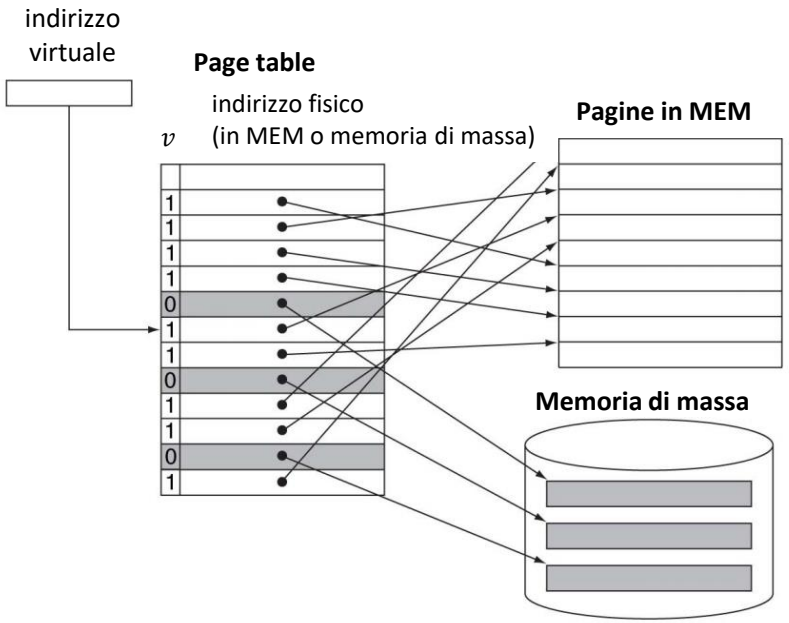
- **Snooping**: le cache lavorano in write-through e ognuna di loro monitora costantemente il bus indirizzi, quando osservano una write su un dato che hanno in cache lo invalidano (write invalidate)
- **Hardware transparency**: ripristino automatico della coerenza, quando un dato viene scritto un circuito dedicato aggiorna il corrispondente in ogni cache
- **Non-cacheable memory**: si definisce una regione della memoria come non allocabile in cache

Memoria Virtuale

- Quando la logica della gerarchia si focalizza tra i livelli di memoria principale e memoria di massa si parla di **memoria virtuale**
- Due obiettivi:
  - consentire una condivisione efficiente della memoria tra diversi programmi
  - rimuovere il vincolo di uno spazio di memoria ridotto ad ogni programma

⇐ **motivazione principale adottata oggi**

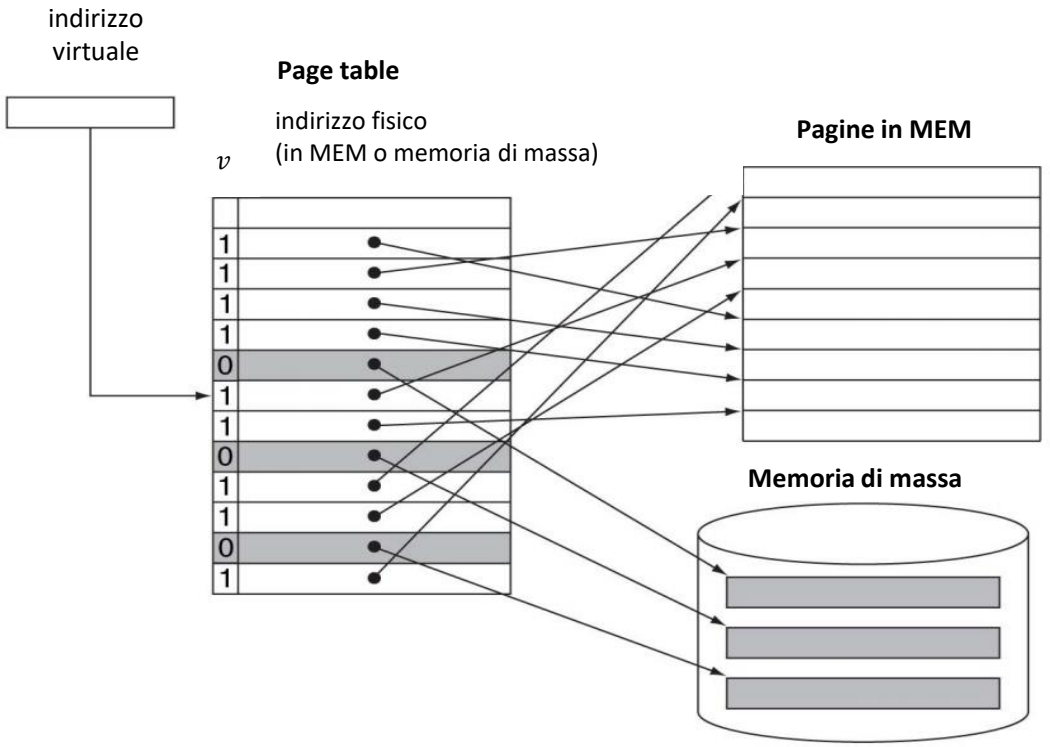
- **Principi di funzionamento**
  - La memoria fisica è suddivisa in pagine (i blocchi della cache)
  - Ogni programma lavora su uno spazio di indirizzamento virtuale, **che non deve tenere conto della presenza di altri programmi**
  - La memoria principale lavora come una cache, dove al posto del blocco ci sono le pagine di memoria (page hit, page fault)
  - In una page table (contenuta in memoria principale) ogni programma ha le proprie corrispondenza tra indirizzo virtuale e fisico



Memoria Virtuale

1. La memoria riceve una richiesta di accesso, tramite un indirizzo virtuale
2. Viene controllata la page table
    1. se la corrispondenza dell'indirizzo virtuale è presente e valida **page hit**: si accede al corrispondente indirizzo fisico in MEM e il dato è recuperato
    2. Altrimenti **page fault**: bisogna trasferire il dato in MEM dalla memoria di massa, **swap**

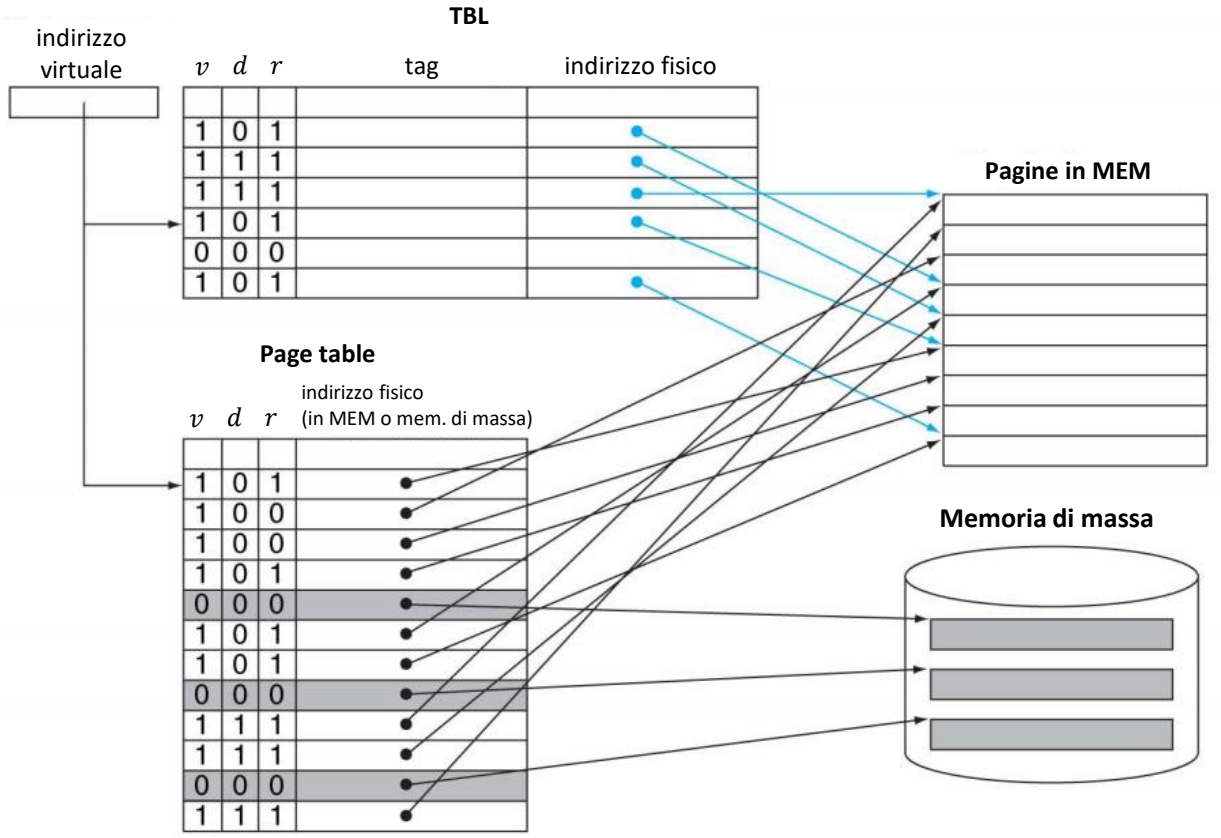
- **Problema:** in ogni caso servono 2 accessi a MEM



Memoria Virtuale

- **Soluzione**: Translation Look-aside buffer
- Una cache per gli indirizzi delle pagine

1. La memoria riceve una richiesta di accesso, tramite un indirizzo virtuale
2. Viene controllato TBL
    1. **hit**: si accede al corrispondente indirizzo fisico in MEM e il dato è recuperato
    2. **miss**: bisogna trasferire il dato in MEM dalla memoria di massa, **swap e aggiornare TBL**



Architettura degli Elaboratori II

Corso di Laurea Triennale in Informatica

Università degli Studi di Milano

Dipartimento di Informatica “Giovanni Degli Antoni”

Edizione 2 (Cognomi H-Z), A.A. 2020-2021, Nicola.Basilico@unimi.it

Rilevamento e correzione di errori

Integrità del dato

- Fino ad ora abbiamo assunto che un dato presente in memoria (a qualsiasi livello della gerarchia) fosse integro e cioè privo di alterazioni causate da fattori esterni alla logica del programma
- **Esempio**: alcuni bit di una parola di memoria cambiano il proprio valore a causa di un malfunzionamento dell'hardware
- Gli effetti possono essere molto gravi, come proteggersi da questi eventi?
- Approcci:
  - costruire memorie più affidabili
  - replicare l'informazione (ridondanza)
  - codici di rilevamento/correzione degli errori: **aggiungiamo** al dato delle informazioni che ci permetteranno di accorgerci se il dato è corrotto
- Se un codice è di correzione, non solo è possibile accorgersi dell'errore nel dato, ma anche correggerlo!

Integrità del dato

Ripasso

- distanza di Hamming tra due informazioni binarie su n bit: il numero di posizioni in cui un bit ha valore diverso tra una informazione e l'altra
- **Esempio**: quanto è la distanza di Hamming tra 11111010 e 11111110? (Risposta: 1)
- Definizione alternativa: la distanza di Hamming è il numero di bit da invertire per rendere le due informazioni identiche

- **Che relazione c'è tra la distanza di Hamming e i codici di rilevamento/correzione degli errori?**
- **Esempio: codice** su 4 bit dove la **distanza di Hamming** tra qualsiasi coppia di elementi del codice è un valore pari:

$$C = \{0000, 0011, 0101, 0110, 1010, 1001, 1100\}$$

1. Le informazioni saranno codificate unicamente attraverso C
2. In questo codice, la distanza di Hamming minima tra due codifiche valide è 2 e, assumendo che lo zero (0000) sia parte del codice, si ha che ogni elemento del codice contiene un numero pari di 1

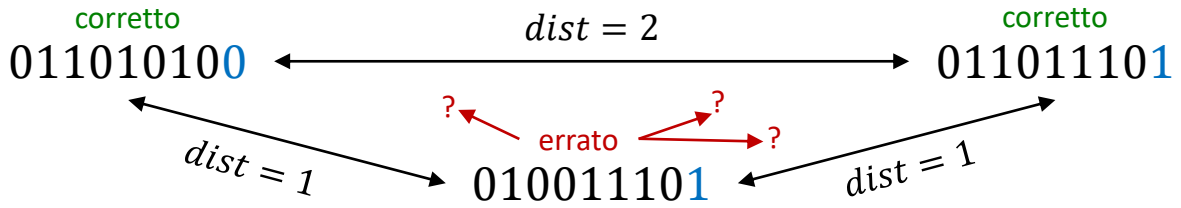
- Da 1 e 2 si deriva una semplice regola di controllo errori: se accedendo ad un dato si osservano un numero dispari di bit pari a 1 (ad esempio se leggo 0111), allora quel dato non appartiene al codice: **non è integro!**
- **Problema**: i codici che usiamo di solito non sono fatti così, come fare per costruire un codice con questa proprietà a partire da un codice qualsiasi (ad esempio da quello dei numeri binari o da quello degli interi in C2, che usiamo di norma per rappresentare le informazioni in memoria)?
- **Soluzione**: il bit di parità

Bit di parità

- Supponiamo di aver calcolato un dato $d = 01101010$ (inizialmente corretto per definizione) e di doverlo salvare in memoria
- Il dato è scritto in un codice qualsiasi (d potrebbe essere un intero senza segno, in C2 o qualsiasi altra cosa)
- Si chiami $\Sigma(d)$ la somma di tutte le cifre (0 e 1) di d interpretate in base 10, nel nostro esempio $\Sigma(d) = 4$
- Si aggiunge a d un bit extra, indicati con $p(d)$ e il cui valore si stabilisce così:

$$p(d) = \begin{cases} 1, \text{se } \Sigma(d) \text{ è dispari} \\ 0, \text{se } \Sigma(d) \text{ è pari} \end{cases}$$

- In memoria salvo $(d, p(d))$ anziché solo d
- Ogni dato salvato in memoria ha sempre un numero pari di 1, se tale numero risultasse dispari allora il dato sarebbe corrotto
- Rileva solo un numero dispari di errori (bit il cui valore non è corretto)
- Si accorge della presenza di un errore, ma non è in grado di correggerlo (non sappiamo quale/quali sono i bit corrotti)



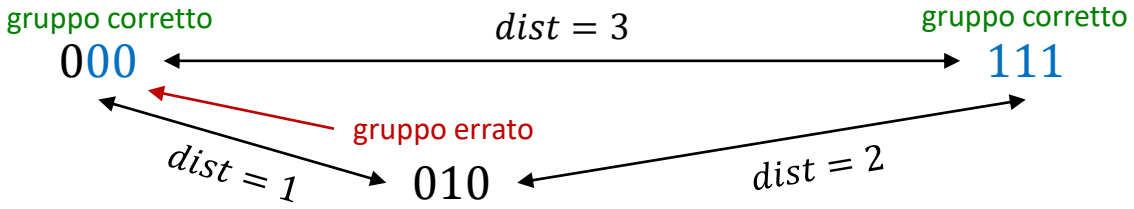
Costo: 1 bit per ogni informazione protetta, può essere il byte o la parola

Codice a ripetizione

- Altro codice basato sull'idea di ridondanza, replico ogni singolo bit n volte
- Ad es. se $n = 2$, $d = 01101010$ diventa

$$000\ 111\ 111\ 000\ 111\ 000\ 111\ 000$$

- Ogni singolo bit viene memorizzato come un gruppo di $n + 1$ bit
- In questo codice la distanza di Hamming minima tra due elementi del codice è pari a $n + 1$, nell'esempio 3



- In ogni gruppo posso rilevare la presenza di **uno o due** errori
- In caso di singolo errore lo posso **correggere**, il gruppo corretto nel codice è quello più vicino secondo la distanza di Hamming!

- In generale, se δ è la distanza di Hamming minima di un codice C allora

1. Quanti errori può **rilevare** C? Risposta: $\delta - 1$
2. Quanti errori può **correggere** C? Risposta: $\left\lfloor \frac{\delta - 1}{2} \right\rfloor$

- Maggiore è δ migliore è il codice, ma il costo aumenta! Con il codice a ripetizione paghiamo n bit aggiuntivi per ogni bit da proteggere

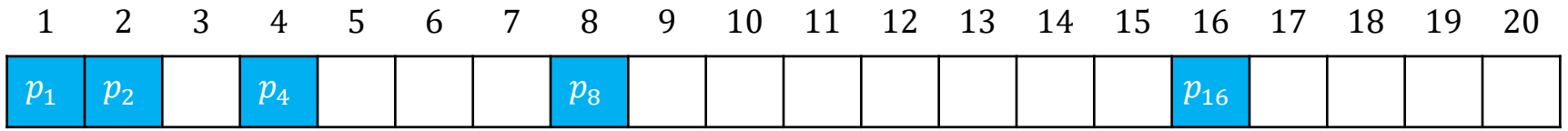
Codice di Hamming

- Il codice di Hamming è un codice a distanza 3 in grado di rilevare e correggere singoli errori
- Ha un costo minore rispetto al codice a ripetizione visto precedentemente

- Il lavoro di Hamming sui codici di rilevamento/correzione di errori gli è valso il prestigioso **Turing Award** nel 1968

Codice di Hamming

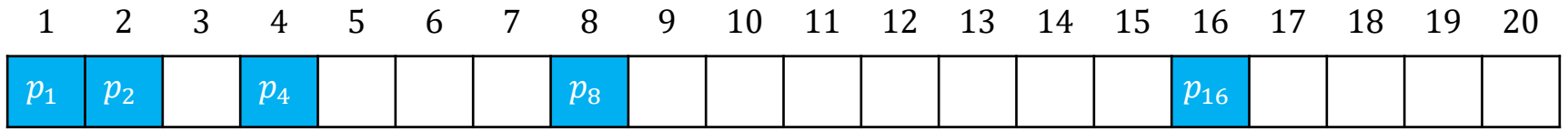
- Codice su n bit dove alcuni bit sono di parità mentre altri sono utilizzati per il dato
- Quali bit hanno il ruolo di parità?
- Numeriamo i bit da 1 a n, da sinistra a destra (al contrario rispetto a quello che facciamo con la codifica binaria!), per non confonderci con la posizione di un bit chiamiamo questo numero **indice**
- I bit di parità sono quelli il cui indice è una potenza di 2
- Esempio con $n = 20$



Bit di parità indicato con p_i dove i è il suo indice

- Se ho 20 bit in totale, vi saranno 5 bit di parità e ne restano 15 per il dato

Codice di Hamming



- Come si usano più bit di parità? Ciascun bit calcola la parità su uno specifico gruppo di altri bit, scelti tra gli n
- **Regola di copertura**: un bit con indice h è protetto dal bit di parità p_i se e solo se h in binario ha un 1 alla **posizione** $\log_2 i$

Operativamente:

- bit di parità p_1, 1 in binario è 00001, protegge tutti gli indici che, in binario, hanno 1 nel bit in posizione 0: 00001, 00011, 000101, …
- bit di parità p_2, 2 in binario è 00010, protegge tutti gli indici che, in binario, hanno 1 nel bit in posizione 1: 00010, 00011, 00110, …
- bit di parità p_4, 4 in binario è 00100, protegge tutti gli indici che, in binario, hanno 1 nel bit in posizione 2: 00100, 00101, 00110, …
- bit di parità p_8, 8 in binario è 01000, protegge tutti gli indici che, in binario, hanno 1 nel bit in posizione 3: 01000, 01001, 01010, …
- bit di parità p_{16}, 16 in binario è 10000, protegge tutti gli indici che, in binario, hanno 1 nel bit in posizione 4: 10000, 10001, 10010, …
- Il bit p_i indica la parità calcolata come fatto precedentemente considerando soltanto i bit a lui assegnati secondo la regola di copertura

indice	indice in binario
1	00001
2	00010
3	00011
4	00100
5	00101
6	00110
7	00111
8	01000
9	01001
10	01010
11	01011
12	01100
13	01101
14	01110
15	01111
16	10000
17	10001
18	10010
19	10011
20	10100

Codice di Hamming

- Pattern di copertura: cosa notiamo?

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	p_1	p_2	d_1	p_4	d_2	d_3	d_4	p_8	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	p_{16}	d_{12}	d_{13}	d_{14}	d_{15}
p_1	■		■		■		■		■		■		■		■		■		■	
p_2		■	■			■	■			■	■			■	■			■	■	
p_4				■	■	■	■					■	■	■	■					■
p_8								■	■	■	■	■	■	■	■					
p_{16}																■	■	■	■	■

1. Ogni bit di parità copre più bit, incluso se stesso
2. **Ogni bit dati è coperto da una combinazione univoca di bit di parità**

- Queste regolarità si mantengono sul pattern per qualsiasi n

- Se riceviamo un dato in cui un gruppo di bit di parità risulta errato allora c'è un solo bit dati che può essere corrotto, l'unico coperto da quel gruppo di bit di parità: lo correggiamo!
- Operativamente: quando leggo il dato calcolo un **codice di errore (sindrome)** definito così (con riferimento all'esempio sopra) $\mathrm{ECC} = c_{16}c_8c_4c_2c_1$, è un intero unsigned dove ogni bit c_i corrisponde ad uno dei bit di parità p_i
- c_i **è a sua volta un bit di parità** calcolato sul gruppo di bit coperti da p_i (ricorda: il gruppo comprende p_i stesso!) , se c'è stato un errore in quel gruppo allora c_i vale 1, altrimenti 0 quindi:
  - Se $ECC = 0$ non ci sono errori
  - Se $ECC = k$ il bit di indice k è errato, lo correggiamo!

Codice di Hamming, leggi del pattern

- Il pattern determina n_p il numero di bit di parità e n_d il numero di bit per il dato, da cui definiamo $n_{tot} = n_p + n_d$
- **Problema 1**: Dato n_p quanto valgono n_d e il massimo n_{tot}?
  - Con n_p bit di parità possono verificarsi 2^{n_p} differenti codici errori (sindromi); i codici, ad esclusione dello 0, sono in corrispondenza biunivoca con gli n_{tot} bit, quindi $n_{tot} = 2^{n_p} - 1$
  - Dalla precedente ricavo immediatamente anche $n_d = 2^{n_p} - 1 - n_p$
- **Problema 2**: Dato n_{tot} quanto valgono n_p e n_d?
  - Per calcolare n_p basta invertire la formula precedente di n_{tot} con una piccola aggiunta: $n_p = \lceil \log_2(n_{tot} + 1) \rceil$
  - L'arrotondamento all'intero successivo è necessario perché, con un n_{tot} arbitrario, un bit di parità potrebbe essere sotto-utilizzato: potrebbe essere in grado di coprire più bit dati di quanti sono presenti in n_{tot}
  - Dalla precedente ricavo immediatamente anche $n_d = n_{tot} - \lceil \log_2(n_{tot} + 1) \rceil$
- **Problema 3**: Dato n_d quanto valgono n_p e n_{tot}?
  - Per trovare n_p potremmo invertire la prima formula di n_{tot}, ma otterremmo un'equazione non risolvibile con le regole elementari dell'algebra basate su logaritmi ed esponenziali, dobbiamo ragionare in modo diverso
  - I codici di errore, che sono 2^{n_p}, devono essere sufficienti per esprimere un codice > 0 per ogni singolo bit degli n_{tot}, quindi $2^{n_p} - 1 \geq n_{tot}$, da cui derivo applicando la definizione di n_{tot} questa relazione: $2^{n_p} - 1 - n_p \geq n_d$ (l'unica incognita qui è n_p)
  - **Soluzione**: trovare il più piccolo n_p che soddisfa la disuguaglianza
  - Operativamente: considero $n_p \in \{2,3,4,5, \dots\}$ in ordine crescente e mi fermo appena la disuguaglianza è soddisfatta

Codice di Hamming

Esempio

- $d = 11100101$
- servono in totale 12 bit, 4 bit di parità e 8 bit per il dato
- p_1 parità su $d_1 + d_2 + d_4 + d_5 + d_7 = 2$, quindi $p_1 \leftarrow 0$
- p_2 parità su $d_1 + d_3 + d_4 + d_6 + d_7 = 3$, quindi $p_2 \leftarrow 1$
- p_4 parità su $d_2 + d_3 + d_4 + d_8 = 3$, quindi $p_4 \leftarrow 1$
- p_8 parità su $d_5 + d_6 + d_7 + d_8 = 2$, quindi $p_8 \leftarrow 0$

- codifica: 011111000101

1	2	3	4	5	6	7	8	9	10	11	12
p_1	p_2	1	p_4	1	1	0	p_8	0	1	0	1

Bit aggiuntivo di parità

- Aggiungendo un ulteriore bit di parità sul pattern otteniamo $\delta = 4$ quindi possiamo identificare e correggere errori singoli e anche identificare (ma non correggere!) errori doppi
- Nuovo bit di parità p_{n_d+1} si applica a tutti gli altri bit (che ora diventano $n_{tot} - 1$)
- Si possono verificare 4 casi in fase di verifica
  1. $ECC = 0$ e p_{n_d+1} corretto → non ci sono errori
  2. $ECC > 0$ e p_{n_d+1} errato → singolo errore che si può correggere
  3. $ECC = 0$ e p_{n_d+1} errato → singolo errore proprio in p_{n_d+1}, si può correggere
  4. $ECC > 0$ e p_{n_d+1} corretto → doppio errore, sappiamo che c'è stato ma non si può correggere

Architettura degli Elaboratori II

Corso di Laurea Triennale in Informatica

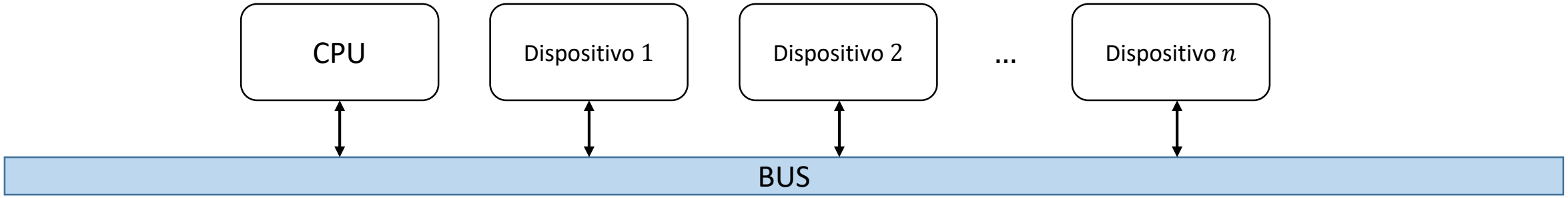
Università degli Studi di Milano

Dipartimento di Informatica “Giovanni Degli Antoni”

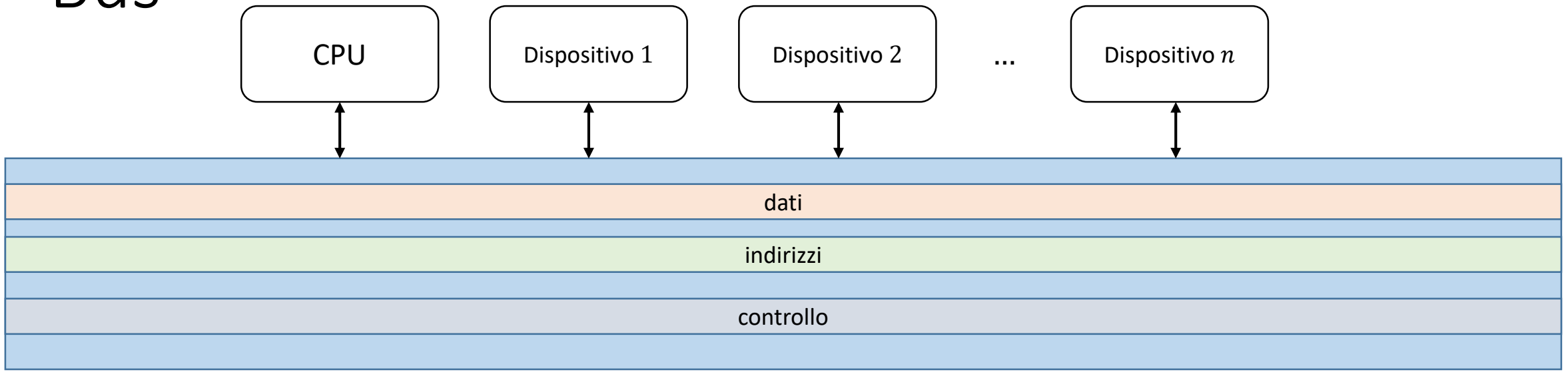
Edizione 2 (Cognomi H-Z), A.A. 2020-2021, Nicola.Basilico@unimi.it

Input/Output

- Nell'elaboratore L'I/O rappresenta l'insieme di metodi e dispositivi con cui trasferire le informazioni tra la CPU e il mondo esterno
- I vari dispositivi di I/O si distinguono per caratteristiche come capacità, velocità, modalità di acquisizione delle informazioni (memorie flash, tastiera, monitor, etc.)
- Per trasferire le informazioni, questi dispositivi sono collegati per via di una canale detto **bus**
- Architetturalmente è un canale condiviso, su cui possono viaggiare le informazioni di dispositivi diversi



Bus



- Internamente il bus ha una architettura finalizzata a supportare trasferimenti di informazione tra le entità che condividono il canale
- Si compone di:
  - **Bus dati**: linee di trasferimento su cui viaggia il contenuto (payload) del trasferimento, il **dato**
  - **Bus indirizzi**: linee di trasferimento su cui la CPU trasmette l'indirizzo di un dato, ad esempio per richiedere accesso in lettura o scrittura ad una parola di memoria
  - **Bus controllo**: informazioni di controllo per lo svolgimento delle operazioni di trasferimento, ad esempio un segnale per consentire la sincronizzazione tra sorgente e destinazione del trasferimento, o segnalazione di stato del trasferimento

Gestione della comunicazione

- Per poter effettuare trasferimenti corretti la comunicazioni tra le parti coinvolte deve rispettare delle regole imposte dall'architettura del bus che prevede una **condivisione** del canale
- **Primo problema**: mittente e destinatario devono coordinarsi, il primo deve sapere quando può scrivere sul bus, il secondo deve sapere quando può leggere dal bus; in caso di errore si ha un trasferimento scorretto
  - Per risolvere questo problema è necessario un meccanismo di **sincronizzazione** tra le parti
- **Secondo problema**: gestire la competizione sul bus
  - Essendo il bus condiviso, in generale, da più di due dispositivi cosa succede se più coppie mittente/destinatario vogliono scambiarsi un dato nello stesso momento?
  - Per risolvere questo problema è necessario un meccanismo di **arbitraggio** del bus, cioè una politica di allocazione della risorsa condivisa (il bus) alle varie parti

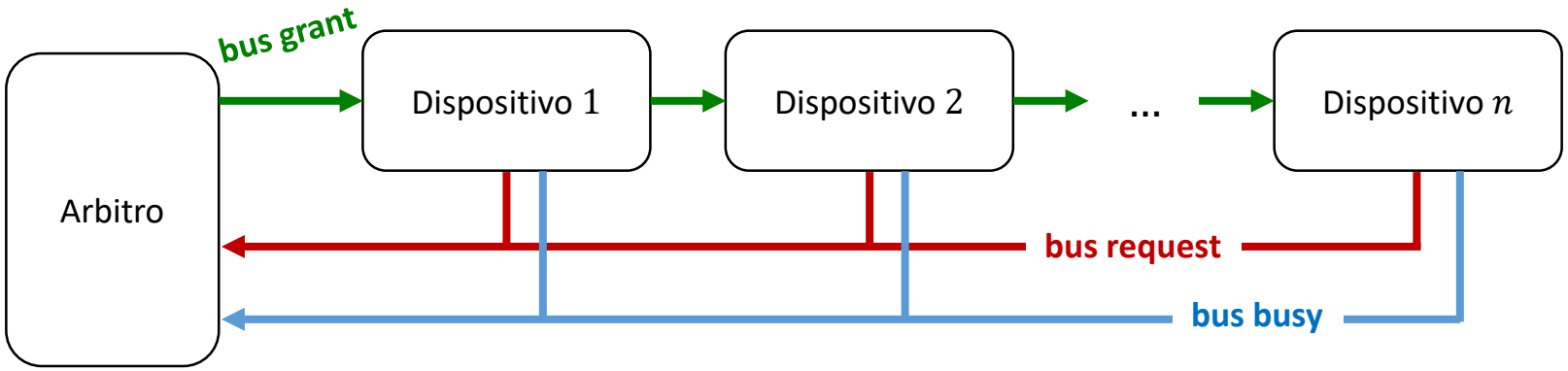
Sincronizzazione del bus

- Prima soluzione: **bus sincrono**
  - il bus contiene un segnale di clock proprio e globale (su una linea di controllo)
  - Ogni dispositivo coinvolto in un trasferimento si coordina implicitamente usando il bus clock come riferimento globale
- Vantaggi:
  - bus molto veloci
- Svantaggi:
  - complessità elevata
  - vincoli di frequenza (il bus clock impone una frequenza di trasferimento ad ogni dispositivo)
  - bus «corto» per evitare grossi ritardi di propagazione del segnale di clock

- Seconda soluzione: **bus asincrono**
  - non esiste un bus clock globale
  - I dispositivi devono sincronizzarsi attuando un protocollo di **handshaking**: uno scambio di messaggi con i quali mittente e destinatario si coordinano
  - **Esempio:** master/slave con messaggi di *transfer ready* e *channel release*
- Vantaggi:
  - bus «lunghi» e con possibilità di avere molti dispositivi collegati
- Svantaggi:
  - limitata velocità di trasferimento

Arbitraggio del bus: daisy chain

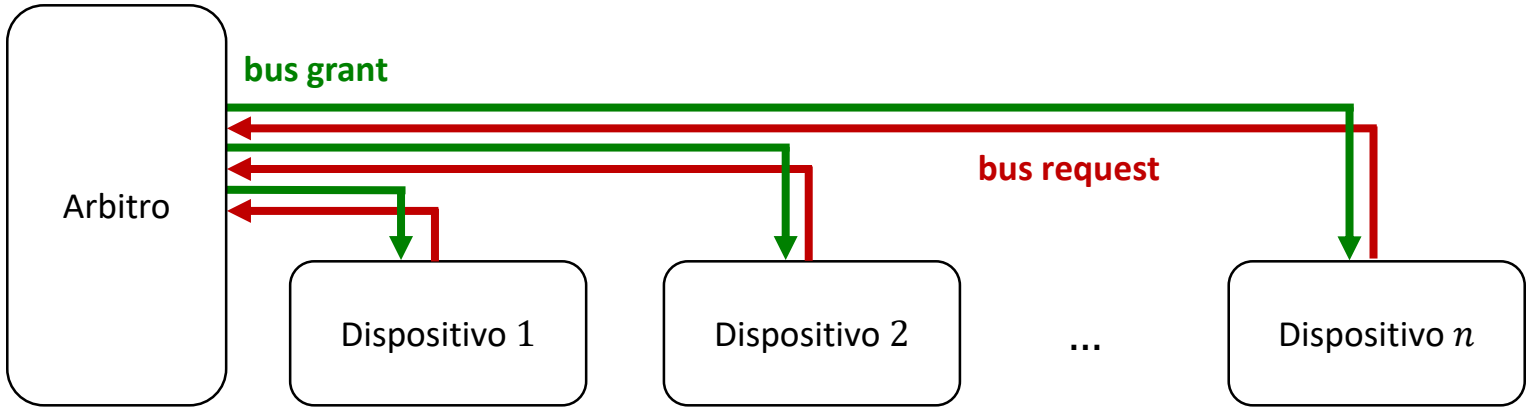
- I dispositivi sono in ordine di priorità, per distanza crescente dal modulo di arbitraggio (CPU)
- Il dispositivo i fa una richiesta del bus (bus request)
- L'arbitro attiva il segnale di bus grant che si propaga fino al **primo dispositivo in ordine di priorità** che ha fatto una bus request (potrebbe essere i o un dispositivo a priorità più alta)
- Il segnale di bus grant viene bloccato dal dispositivo che lo ha ottenuto e non procede in avanti nella catena, tale dispositivo occupa il bus e attiva il segnale di bus busy



- Metodo semplice ma soffre di alcuni problemi:
  - ritardi di propagazione, non è possibile aggiungere troppi dispositivi
  - single point of failure: se un dispositivo si blocca tutta la catena si blocca
  - **unfair**: la risorsa non è ripartita equamente, può succedere che uno o più dispositivi a priorità bassa non accedano mai al bus

Arbitraggio del bus: richieste indipendenti

- Approccio opposto: ogni dispositivo ha la sua linea dedicata per fare bus request e per ricevere il grant

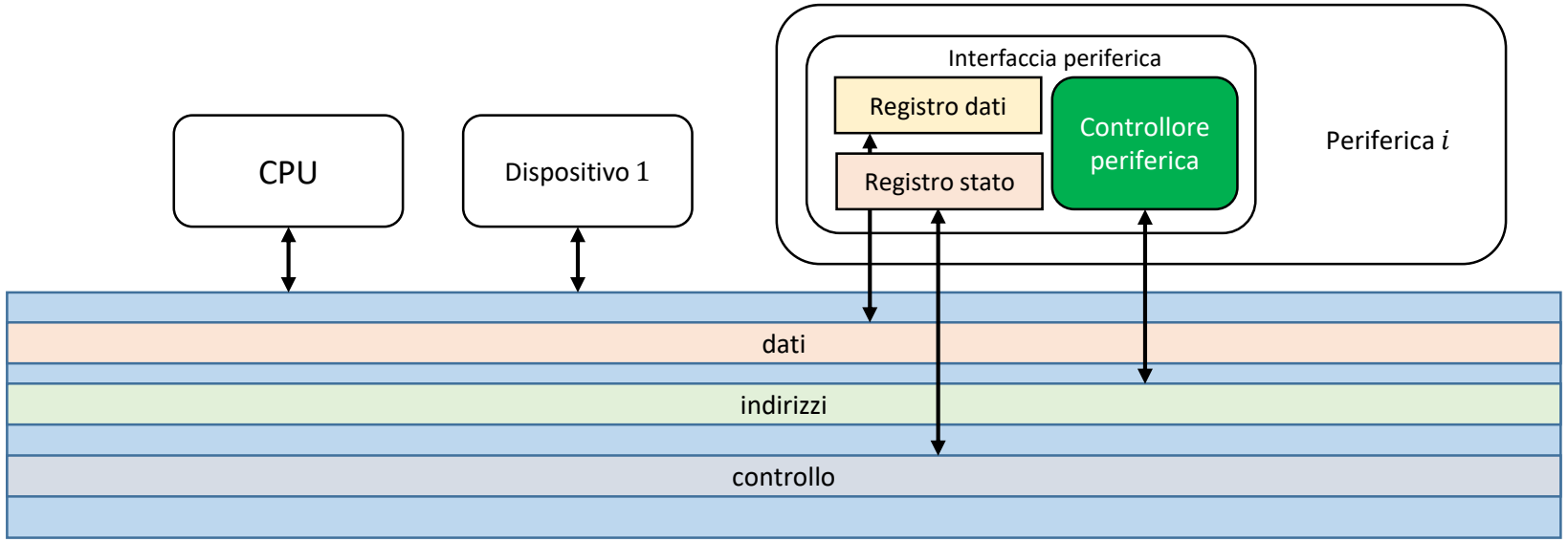


- Possibilità di aumentare la fairness: l'arbitro può implementare una policy che garantisca uno share minimo della risorsa ad ogni dispositivo
- L'hardware di arbitraggio costa di più ed è più complesso

Arbitraggio del bus: altri approcci

- Approcci intermedi tra daisy chain e richieste indipendenti
- Gruppi di dispositivi, all'interno di ogni gruppo gestione daisy chain, tra gruppi diversi invece richieste indipendenti
- Approcci decentralizzati: non c'è un arbitro, solo i dispositivi!
  - Il coordinamento avviene tramite un protocollo distribuito tramite il quale i dispositivi raggiungono un agreement su chi ha diritto ad accedere al bus
  - Approccio più semplice: priorità statica fissata a priori tramite un ID

Accesso alle periferiche



- **Controllore**: logica di controllo per accedere al bus e trasferire dati (simile al driver, ma ad un livello diverso)
- **Registro dati**: buffer temporaneo per i dati da scrivere/leggere sul bus
- **Registro di stato**: codici che definiscono lo stato attuale della periferica

Accesso alla periferica da programma

- memory-mapped:
  - l'accesso alla periferica è mascherato da accesso alla memoria principale, si definiscono indirizzi speciali di I/O (tipicamente in area kernel) che, anziché puntare a celle di memoria, identificano un buffer di una periferica
  - Il controller di ogni periferica «ascolta» il bus indirizzi, quando riconosce l'indirizzo associato si attiva
- istruzioni speciali I/O:
  - istruzioni macchina definita dall'ISA per l'accesso specifico ad una data periferica

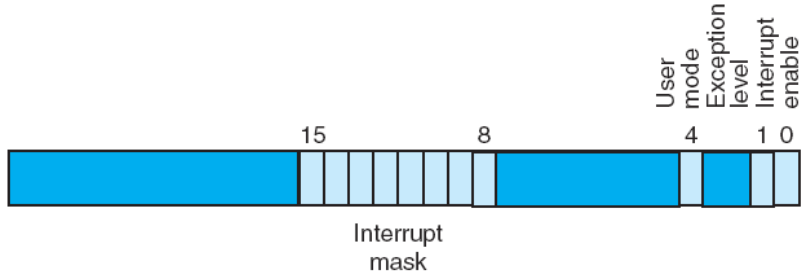
Controllo I/O

- Come coordinare l'attività della CPU con quella di una periferica all'interno di un programma? Come viene scambiato il controllo di esecuzione?
- Primo metodo: **controllo a programma**
- Fa tutto la CPU:
    1. Comunica al controllore della periferica l'esecuzione di un'operazione i I/O
    2. Resta in ascolto sul registro di stato in attesa che l'operazione sia conclusa
    3. Quando l'operazione si è conclusa continua con l'esecuzione del programma
- Lo step 2 può causare stallo e quindi, con periferiche lente, grandi tempi di attesa
- Una soluzione per mitigare il problema è il **polling**: anziché andare in stallo la CPU controlla periodicamente (con una frequenza prestabilita) il registro di stato

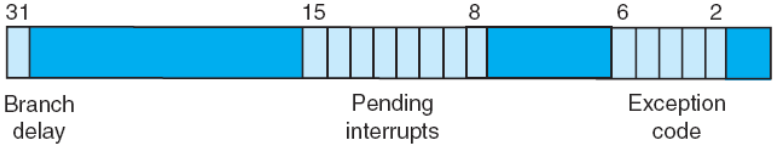
Controllo I/O

- Secondo metodo: **interrupt**
- La periferica prende l'iniziativa inviando un segnale alla CPU
- Una volta ricevuto, il segnale interrompe la normale esecuzione e viene gestito in modo analogo ad un'eccezione
- Interrupt multipli a livelli di priorità diversi
- Interrupt mask: definisce il livello (priorità) corrente, all'interno dello stesso livello gli interrupt seguono una logica FIFO, se bit della maschera pari a 1 gli interrupt a quel livello sono disabilitati

Registro di stato



Registro di causa



Valutazione delle prestazioni

- CPI_i Clock cycles Per Instruction, quanti cicli di clock servono per eseguire un'istruzione i (può essere un valore atteso) , da non confondere con **IPC** (Instructions Per clock Cycle): il numero massimo di istruzioni che la pipeline è in grado lanciare in un singolo ciclo di clock
- Dato un programma p (ad esempio un benchmark) chiamiamo n_i il numero di istruzioni di tipo i in p, e chiamiamo n il totale delle istruzioni, assumendo che ci siano N tipi di istruzioni allora $n = \sum_{i=1}^{N} n_i$
- Frequenza di una istruzione di tipo i nel programma p: $f_i = \frac{n_i}{n}$
- $\overline{CPI}$ medio del programma p

$$\overline{CPI} = \sum_{i=1}^{N} CPI_i \times f_i$$

- Tempo atteso di esecuzione del programma p

$$\bar{T} = \overline{CPI} \times n \times T_{ck}$$

- **Quale è il limite fondamentale di questo modello?**

Valutazione delle prestazioni

- Esempio
- Profilo del programma p riportato in tabella, si assuma una CPU da $500MHz$
- $N = 5, n = \sum_{i=1}^{N} n_i = 250$

Tipo istruzione	CPI_i	n_i	f_i
ALU	1	156	.624
Load	4	37	.148
Store	4	35	.14
Branch	2	15	.06
Jump	2	7	.028

$$\overline{CPI} = \sum_{i=1}^{N} CPI_i \times f_i = .624 + 4 \times .148 + 4 \times .14 + 2 \times .06 + 2 \times .028 = 1.952$$

$$\bar{T} = \overline{CPI} \times n \times T_{ck} = 1.952 \times 250 \times \frac{1}{5 \times 10^8} = .976\mu s$$

Valutazione delle prestazioni

- Metriche spesso usate:
- MIPS: Million Instructions Per Second, attenti al termine *peak*!
- MIPS relativo, si misura rispetto ad una CPU di riferimento: $\frac{T_{ref}}{T} \times MIPS_{ref}$
- MFLOPS: Million FLoating-point Operations Per Second