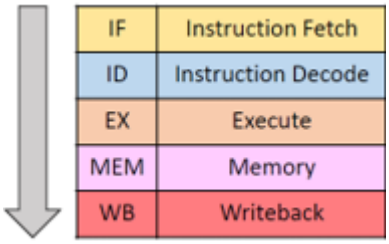
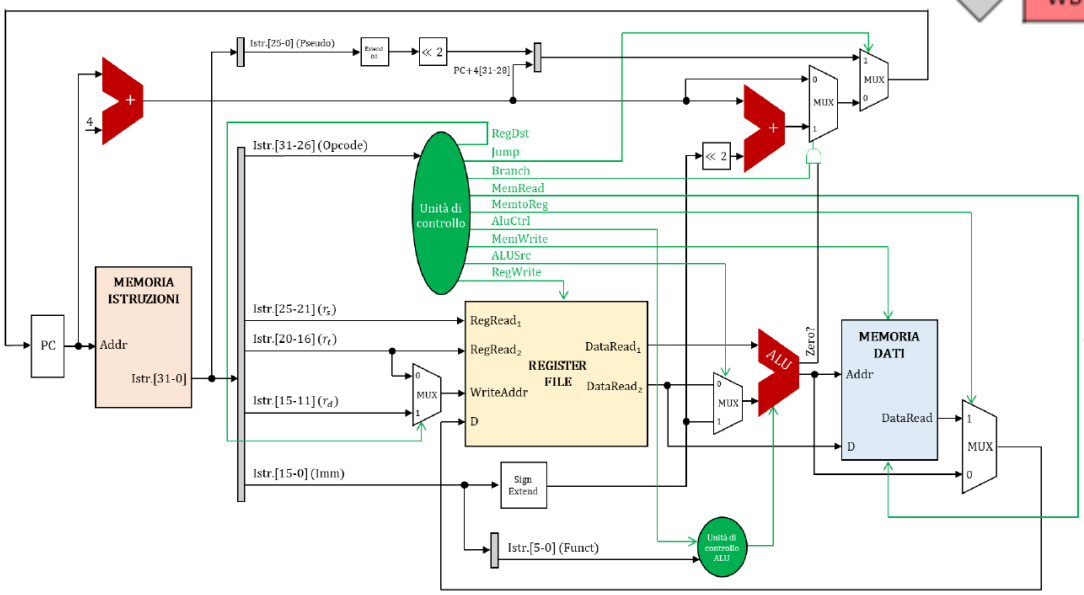
ARCHITETTURA DEGLI ELABORATORI II

CPU A SINGOLO CICLO (archi I)

- IF/Fetch -> scritto in program counter nuovo indirizzo della nuova istruzione (32-bit)
- ID/Decode -> lettura degli indirizzi necessari (lettura in register file)
- EX/Execute -> esecuzione di operazione aritmetico logica da parte dell'ALU
- MEM/Memory -> preleviamo o scriviamo un dato delle memorie
  (Operazioni load_word/store_word)
- WB/Writeback -> scrittura in un registro del register_File (quando necessario)





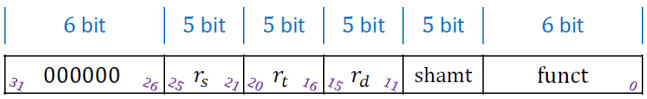
implementazione semplice
- l'esecuzione avviene in un singolo ciclo di clock (1 ciclo di clock, una istruzione)
- l'unità di controllo è una funzione combinatoria (quindi meno complesso)

ma inefficiente
- ogni istruzione richiede un tempo (il clock va dimensionato su istruzione più lenta)
- replicazione dei componenti, perché per ogni istruzione si può fare un ciclo solo
  - se una istruzione ha bisogno di fare due operazioni aritmetiche (es. beq con 2 ALU)
  - due unità di memoria (1 per il fetch dell'istruzione 2 per la lettura/scrittura dati)

Supporta 4 tipi di operazioni: (tipo di istruzione != formato istruzione)
- istruzioni aritmetico-logiche (A/L) (add, or, not, +, -, etc.)
- istruzioni di accesso alla memoria dati (lw, sw)
- salti condizionati (beq, solitamente sempre di uguaglianza)
- salti non condizionati (j)

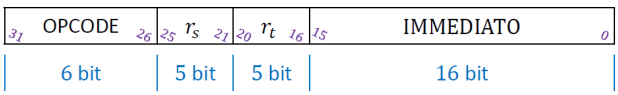
3 tipi di formati istruzione:

R-type



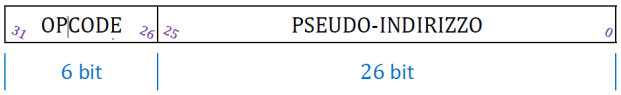
"Somma i valori presenti in rs e rt, inserisci in rd"
rs = registrer_Source (è un puntatore ad un registro)
rt = valore da sommare (perché t viene dopo la s)
rd = register_Destination (puntatore ad un registro)
funct = quale operazione aritmetico logica svolgere

I-type



Rs = primo valore (è un puntatore ad un registro)
Rt = secondo valore (è un puntatore ad un registro)
Immediato = valore immediato

J-type

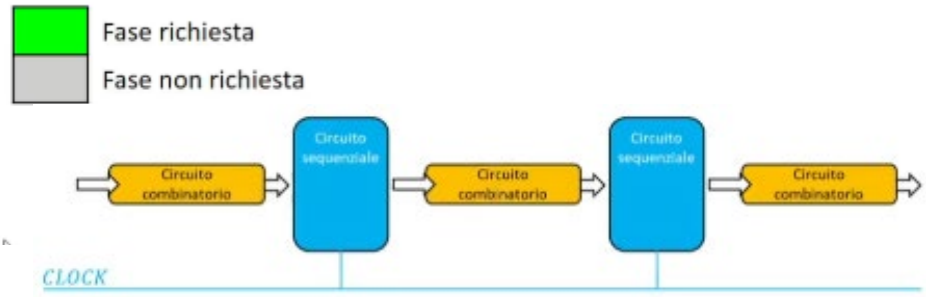


A partire dallo pseudo-indirizzo si arriva all'indirizzo vero e proprio
26 -> 32 gli ultimi 2 a destra sono sempre 0, gli altri 4 vengono presi dal program_Counter

CPU A CICLO MULTIPLO

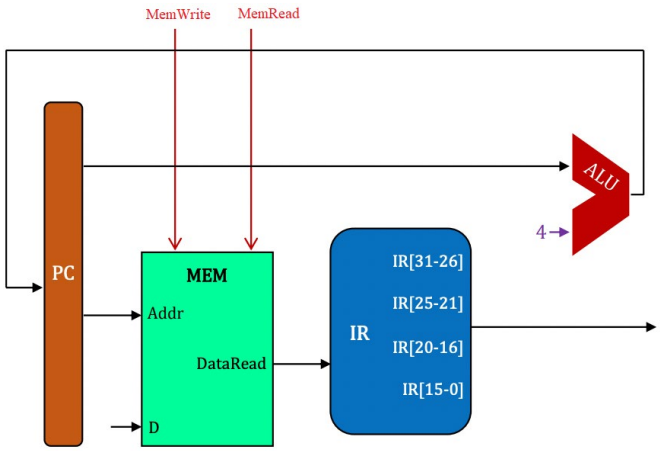
- l'esecuzione di una istruzione è organizzata in 5 sotto-fasi (ad ogni clock una singola sotto-fase)
  -> consentiamo a diverse istruzioni di impegnare la CPU per un numero diverso di cicli di clock

	IF	ID	EX	MEM	WB
lw					
sw					
R					
beq					
j					



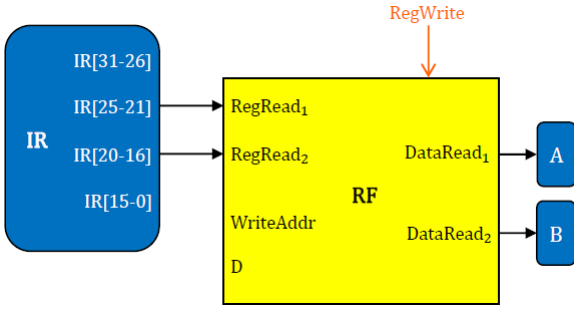
-> L'unità di controllo deve essere in grado di sapere a quale punto dell'esecuzione si trova di ogni istruzione
-> Il datapath deve contenere registri di memoria per il passaggio di informazioni tra sotto-fasi

Fase di IF (Instruction Fetch)



- PC (Program counter): contiene l'indirizzo dell'istruzione corrente
- MEM (Memory): unica unità di memoria con dati e istruzioni.
è dotata di bit di enable/disable scrittura (D):
segnale MemRead posto ad 1 nella fase di IF, per dire che il componente lavora in modalità lettura di dato (per i 32 bit che codificano l'istruzione del PC).
segnale MemWrite, abilita la scrittura nel componente. il componente non può abilitare scrittura e lettura nello stesso ciclo.
il contenuto dell'indirizzo dell'istruzione corrente prende nome di DataRead.
- IR (Instruction Register): contiene l'istruzione corrente trasferita dalla memoria. È il registro di transizione per le sotto-fasi successive (contiene i dati necessari $rs, $rd etc)
- ALU: non è più un sommatore unicamente dedicato al calcolo del prossimo valore del program_Counter, ma viene utilizzato anche dagli altri componenti. (es. nella fase di EX)

Fase di ID (instruction decode)

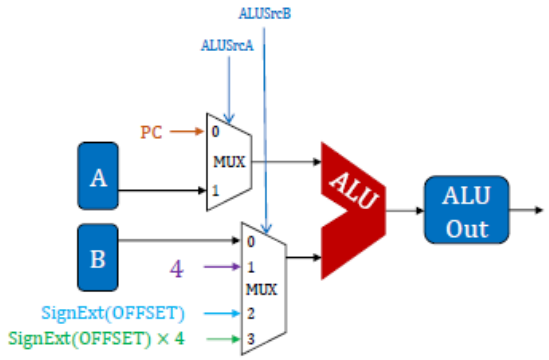


La CPU presuppone sempre che l'istruzione sia già stata prelevata e pronta ad essere decodificata
- in IR si hanno i 32 bit di istruzione da decodificare
- gli indirizzi rt(25-21) e rs(20-16) vanno direttamente in output in DataRead1 e DataRead2, quindi escono per andare nei due registri di transizione A e B

- Le istruzioni R e BEQ → utilizzano il contenuto di A e B come operandi della ALU.
- Istruzioni di accesso a memoria (lw/sw) → utilizzano solo il contenuto di A (rs), il secondo operando dell'ALU (offset) viene ricavato dall'IR
- l'istruzione di JUMP non utilizza A e B

Fase EX/ALU

- nei registri A e B sono contenuti i registri rs e rt (esistenti o no a seconda dell'istruzione)
- fa sempre un'operazione aritmetica/logica. Il risultato sarà memorizzato nel registro di transizione ALU Out
- Fasi in cui viene utilizzata la ALU:
  - durante la fase IF per fare PC+4
  - durante le fasi ex A/L
  - durante LW E SW (rs + SignExt(OFFSET)
  - durante fase di jump la ALU non è usata

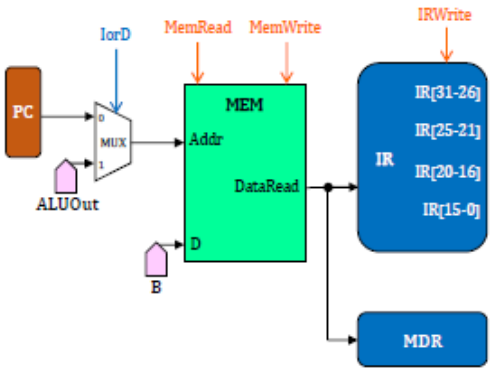


Rimane però da gestire la BEQ, che richiede l'utilizzo dell'ALU 2 volte:
1. Per calcolare l'indirizzo di salto $PC + SignExt(OFFSET)\ x\ 4$ (offset è un IMMEDIATE, in IR)
2. Per verificare la condizione $(rs) - (rt) == 0$

	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

Fase di MEM (Memory)

- Il modulo di memoria lavora in modalità lettura (MemRead = 1), scrittura (MemWrite = 1) e riposo (MemRead = 0, MemWrite = 0; i valori non possono essere entrambi = 1)
- il MUX prima della MEM permette di selezionare l'indirizzo:
  - instruction (indirizzo di una istruzione tramite Program Counter)
  - data (indirizzo di una word tramite ALUOut)

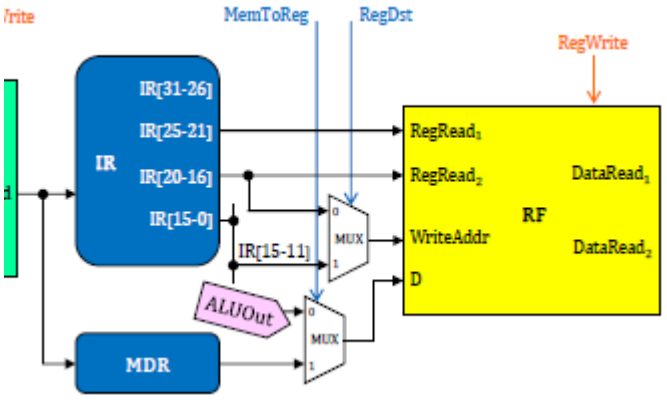


- Il segnale di selezione IorD (Instruction or Data):
  - Nell'istruzione LW
    - IorD = 1 (quindi con l'indirizzo ALUOut)
    - MemRead = 1 (memoria impostata in modalità lettura)
    - il valore viene inserito nel registro di transizione MDR (Memory Data Register)
  - Nell'istruzione di SW
    - IorD = 1 (quindi con l'indirizzo ALUOut)
    - MemWrite = 1 (memoria impostata in modalità di scrittura)
    - il dato viene inserito nel registro di transizione B ((rt))

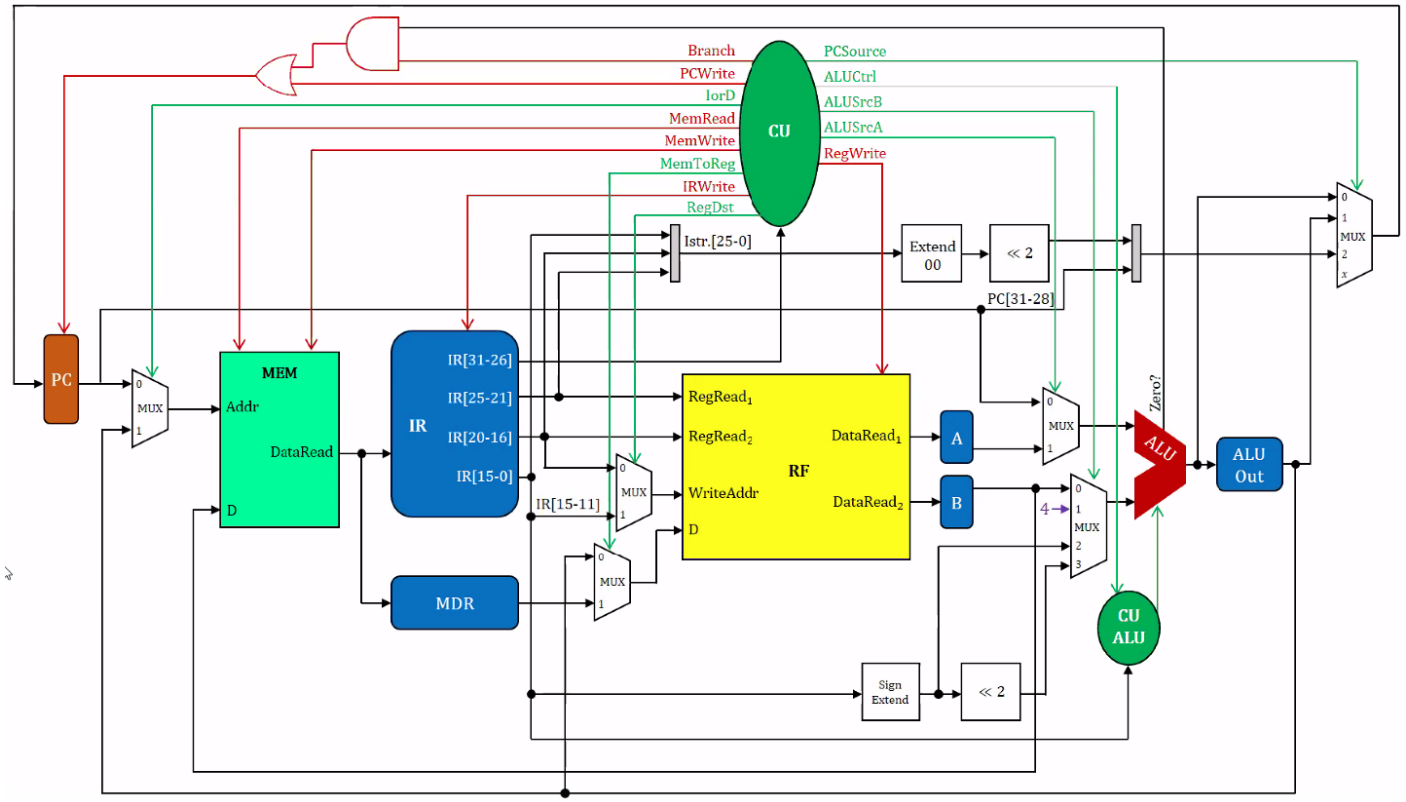
Fase WB (Write Back)

- avviene solitamente dopo le istruzioni A/L o le istruzioni di Mem

durante le istruzioni A/L:
- il dato è in ALUOut
- MemToReg = 0 (per selezionare nel multiplexer ALUOut)
- l'indirizzo del registro in cui devo scriverlo è in IR[15-11] (rd)
- RegDst = 1 (per mandare indirizzo in WriteAddr)



durante le istruzioni di LW:
- MemToReg = 1 (quindi prendiamo il dato in MDR)
- RegDst = 0 l'indirizzo del registor in cui devo scriverlo è in IR[20-16](rt)



PCSource

il valore del PC è determinato dal MUX (3 possibilità, il quarto è vuoto perché non esiste MUX con numero dispari di entrate)
I 3 valori possibili sono:

1) MUX = 0
la prossima istruzione è la successiva => PC + 4 (calcolo svolto con l'ALU)

2) MUX = 1 (nei casi di salto condizionato)
devo inserire nel program counter BTR (presente in ALUOut che è stato presente nella fase di decode precedente)

3) MUX = 2 (nel caso di salto non condizionato)
devo inserire nel PC l'indirizzo nella jump
formo l'indirizzo di salto indicato nell'istruzione di salto: sfrutto il valore immediate, estendo con "00", moltiplico per 4 e appendo i 4 bit più significativi del PC

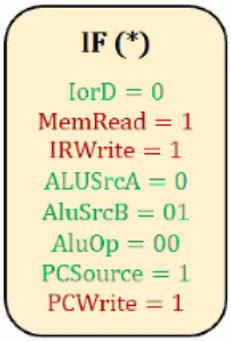
SEGNALI DI CONTROLLO			
SEGNALI DI SELEZIONE		**SEGNALI DI COMANDO**	
ALUSrcA	Selezione del primo operando ALU: PC(0), registro A (1)	**PCWrite**	Scrittura del Program Counter
ALUSrcB	Selezione del secondo operando ALU: Registro B (00), costante 4 (01), SignExt(OFFSET) (10), SignExt(OFFSET)x4 (11)	**Branch**	Se posto a 1 abilita la scrittura del PC quando il bit di zero della ALU vale 1, da usare per i salti condizionati
IorD	Selezione per il campo Addr nel modulo memoria: Indirizzo istruzione (0), Indirizzo dato (1)	**IRWrite**	Scrittura dell'Instruction Register
PCSrc	Selezione di quale indirizzo mandare al PC: risultato della ALU (00), contenuto di ALUOut (01), indirizzo della jump (10), (11 non usato)	**RegWrite**	Scrittura del Register File
RegDest	Selezione per il campo WriteAddr del RF: rt(0) rd(1)	**MemWrite**	Scrittura della memoria
MemToReg	Selezione del dato presentato in scrittura al RF: contenuto di ALUOut (01), contenuto di MDR (1)	**MemRead**	Lettura della memoria
ALUCtrl	Selezione della modalità di operazione della ALU (analogo a CPU singolo ciclo)		

CONTROL UNIT (MACCHINA DI MOORE FSM)

lo stato dipende da che sotto-fase + tipo-istruzione-corrente
- in input: il tipo dell'istruzione (opcode)
- in output: il valore dei segnali di controllo

Instruction Fetch || IF(*) (STATO INIZIALE)

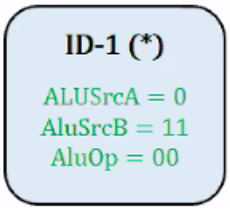
Segnali di controllo per garantire che:
- L'istruzione indirizzata da PC venga letta da memoria e scritta in IR
- L'ALU svolga PC+4 (scritto in ALUOut, ma superfluo)
- Il risultato della ALU viene scritto nel PC



IorD = 0 -> memoria utilizzata per prelevare un'istruzione
MemRead = 1 -> devo leggere la memoria
IRWrite =1 -> devo scrivere nell'Istruction Register
ALUSrcA = 0 (così entra PC)
ALUSrcB (che vale 4)
ALUOp (== ALUControl)

Instruction Decode 1 || ID-1(*)

- IR contiene i 32 bit dell'istruzione i cui campi vanno in ingresso al RF.
- L'RF legge il contenuto dei registri e li carica in A e B

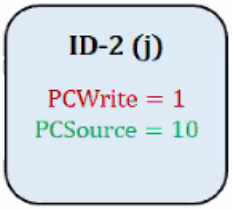


I segnali di controllo sono settati per garantire l'anticipazione del BTA:
- l'ALU svolge il $PC + SignExt(OFFSET) * 4$
- il risultato dell'ALU viene inserito in ALUOUT

ALUSrcA = 0; ALUSrcB = 11 per l'offset

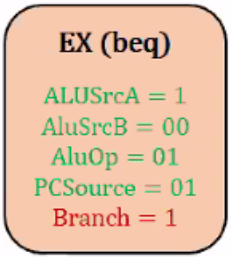
Instruction Decode 2 (per JUMP) || ID-2(J)

Secondo momento della decode (la CPU sa già che è un'istruzione di jump)
I segnali di controllo sono settati in modo che venga sovrascritto nel PC l'indirizzo di salto



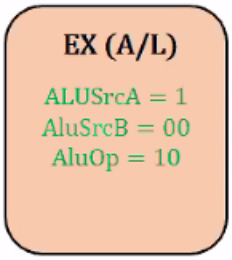
Execute di BEQ || EX(beq)

Grazie alla precedente fase di ID-1, in ALUOut abbiamo il BTA
I segnali di controllo sono settati perché:
- ALU svolga $(rs) - (rt)$, che verrà sovrascritto in ALUOut
- se il bit di zero = 1; il contenuto di ALUOut è scritto in PC



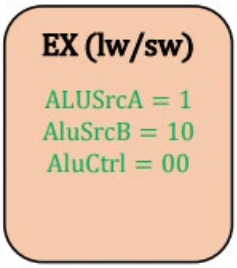
Execute di una qualsiasi operazione A/L || EX(A/L)

Grazie alla precedente fase ID-1, in A e B ci sono rs e rt
I segnali di controllo sono settati perché
- L'ALU svolga $(rs)\ op\ (rt)$
- il risultato dell'ALU viene scritto in ALUOut



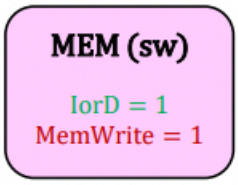
Execute di lw o sw || EX(lw/sw)

Grazie alla precedente fase di ID-1, in A c'è rs
I segnali di controllo sono segnati perché:
- l'ALU svolga $(rs) + SignExt(OFFSET)$ per il calcolo dell'indirizzo
- Il risultato dell'ALU venga scritto in ALUOUT



Memory di SW || MEM(sw)

Grazie alla precedente fase di ID-1, in B abbiamo il contenuto di rt.
B è collegato direttamente all'ingresso dati del modulo di memoria
I segnali di controllo sono settati in modo tale che rt venga scritto in memoria all'indirizzo contenuto in ALUOUT



Memory di LW || MEM(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

MEM (lw)
IorD = 1
MemRead = 1

Writeback di una qualsiasi istruzione A/L || WB(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 rs

WB (A/L)
RegDst = 1
MemToReg = 0
RegWrite = 1

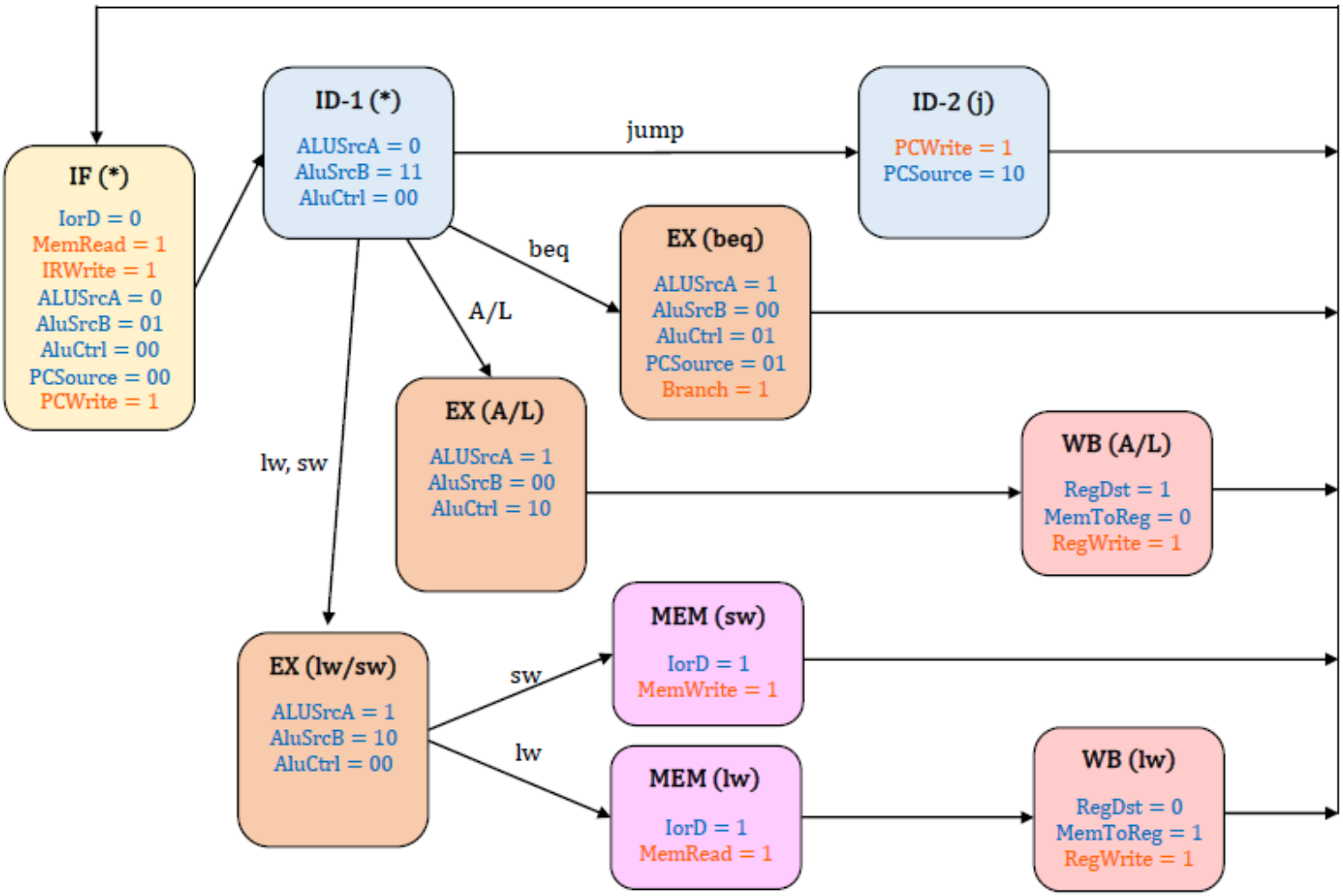
Writeback di LW || WB(lw)

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

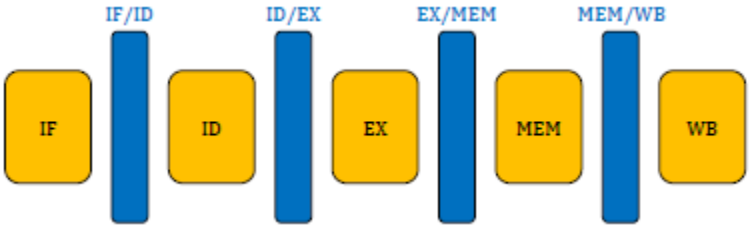
WB (lw)
RegDst = 0
MemToReg = 1
RegWrite = 1

Grafo delle transizioni della FSM che implementa la CU

Gli archi identificati da un'istruzione, dipendono da che tipo di Opcode riceve in input; dove gli archi non sono identificati -> la transizione è spontanea (qualsiasi input si muove)



CPU A PIPELINE



- implementazione di una catena di montaggio
- le fasi lavorano in parallelo (In un ciclo di clock, possono esserci 5 istruzioni contemporaneamente ma in fasi diverse)
- N volte (numero di fasi) più veloce della macchina a ciclo singolo
- suddividere il datapath in modo che ogni fase possa lavorare indipendentemente
- servono dei registri di transizione per tenere il risultato parziale per la fase successiva

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 $T_{ck} \geq t$

$$T_P \leq T_M \leq T_S$$

CPU pipeline anche nei casi peggiori è più veloce della CPU a multiciclo

CRITICITÀ STRUTTURALE (STRUCTURAL HAZARD)

- se manteniamo una singola ALU e una memoria, le istruzioni possono andare in conflitto d'utilizzo di una risorsa (ex. *sw e sub*, vanno in conflitto sull'ALU).

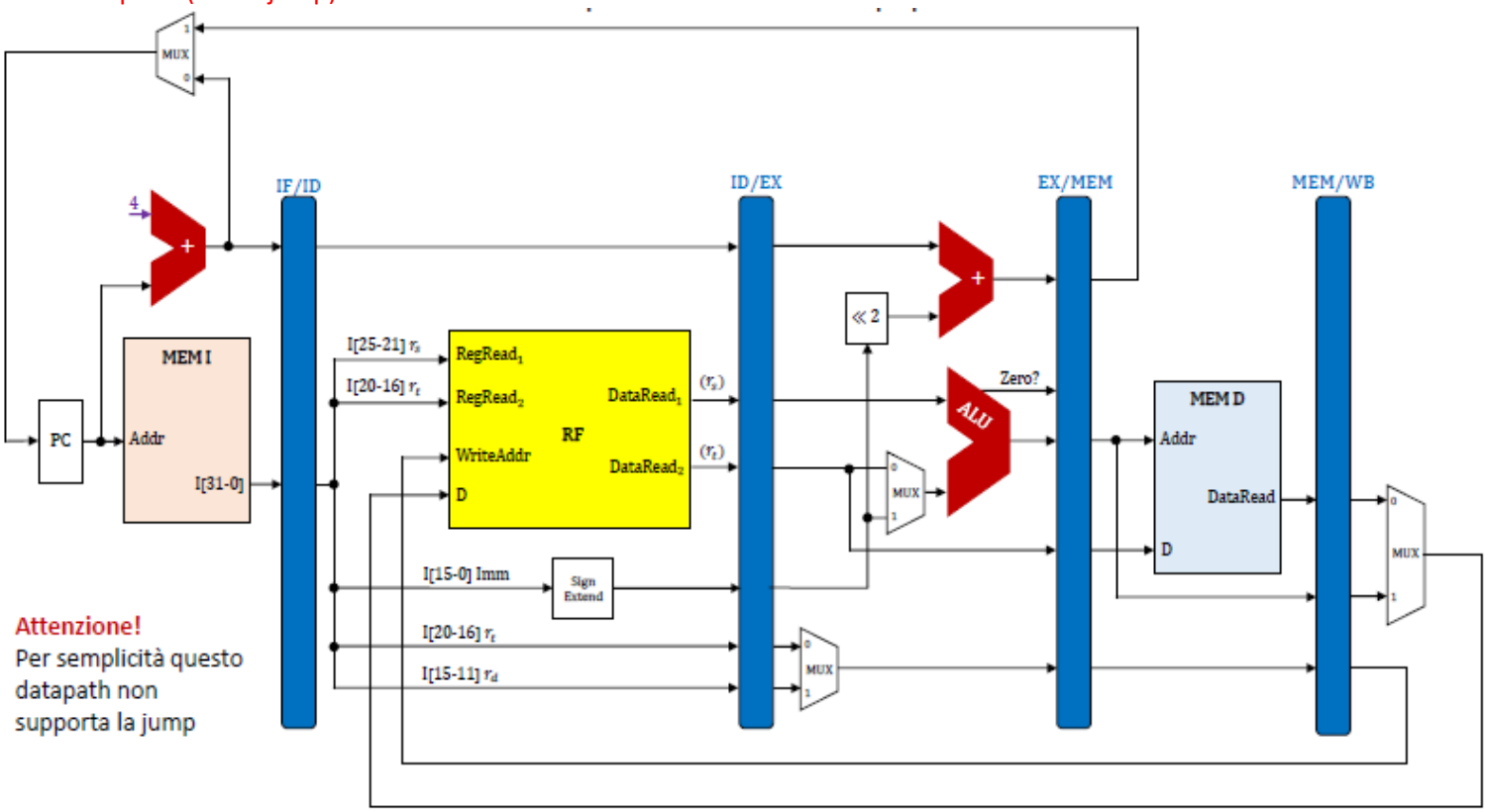
- La memory può svolgere fino a 2 operazioni nello stesso ciclo di clock, a patto che siano opposte (read + write) (o read o write).

Torniamo a duplicare i componenti:
- 1 RF, perché non c'è criticità (Sono sempre in modalità diverse read o write)
- 3 ALU, per IF ID EX
- 2 MEM, per IF e MEM

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

(PAGINA 13 DEL PDF CPU PIPELINE PER TROVARE UN ESEMPIO)

CPU completa (senza jump)



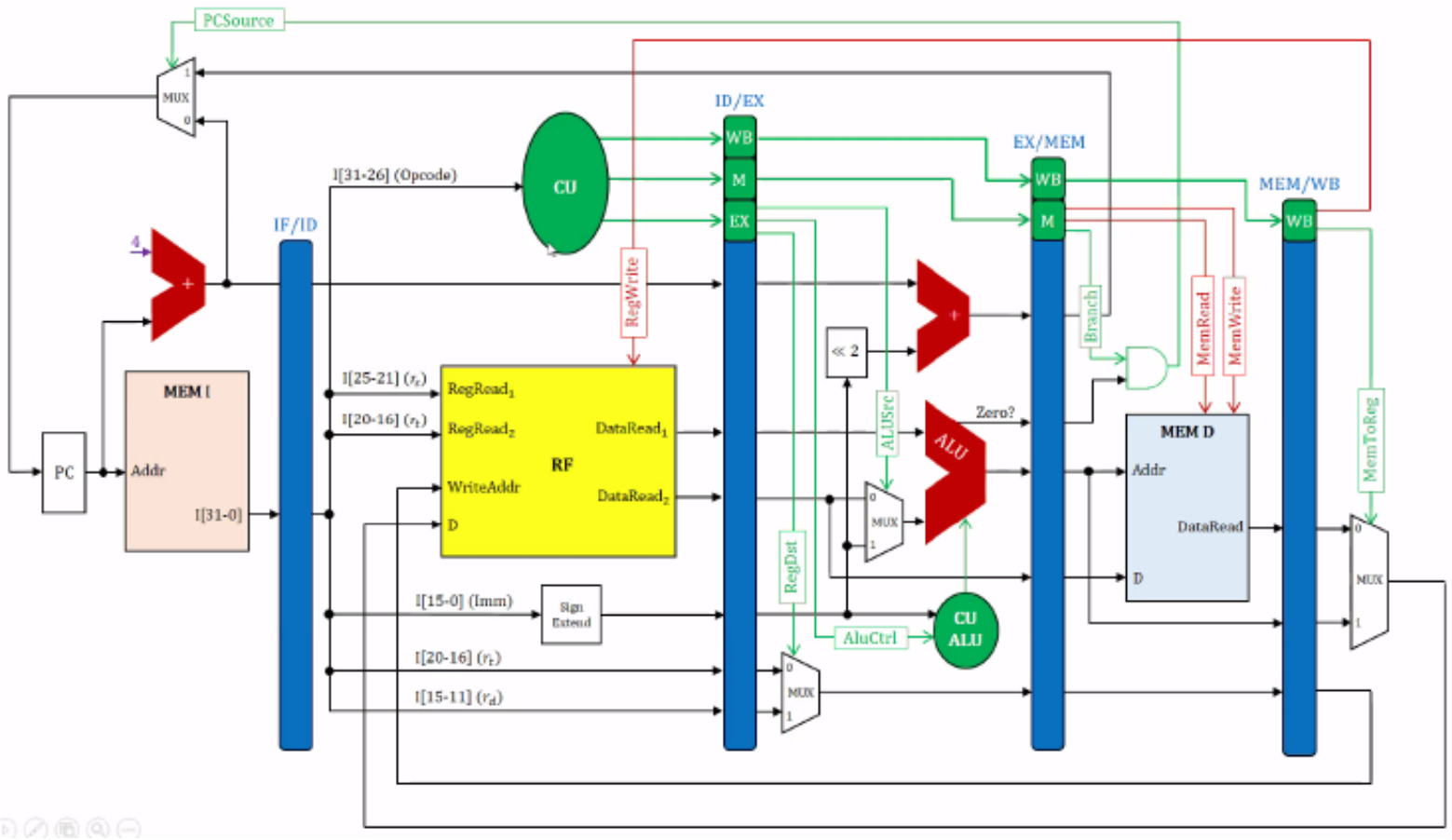
SEGNALI DI CONTROLLO			
SEGNALI DI SELEZIONE		SEGNALI DI COMANDO	
ALUSrc	Selezione del secondo operando ALU: (rt) (0) immediato esteso di segno (1)	RegWrite	Scrittura del Register File
RegDest	Selezione campo WriteAddr del RF: rt (0), rd (1)	MemWrite	Scrittura della memoria
MemToReg	Selezione del dato presentato in scrittura al RF: risultato ALU (0), contenuto di dato letto da MEM (1)	MemRead	Lettura della memoria
AluCtrl	Selezione della modalità di operazione della ALU (uguale a CPU singolo ciclo)	***il registro di memoria non ha segnali di controllo perché legge ad ogni ciclo di clock**	
Branch	Indica se istruzione corrente è una beq		
PCSrc	Selezione di quale indirizzo mandare al PC: PC+4 (0), BTA (1)		

- anche i segnali di controllo sono trasportati e fanno tutte le varie fasi della CPU
- Nella fase IF non esistono segnali di controllo
- Nella fase di ID non dipende dai segnali di controllo che in questa fase vengono tutti generati a partire dall'istruzione

A seconda del tipo di istruzione ➔ segnali diversi

	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

- Estendiamo i registri pipeline ed il datapath per trasportare i segnali di controllo dopo ogni stadio, in modo solidale con l'istruzione
- Ogni stadio opera in maniera parallelo all'istruzione in lavorazione al suo interno, leggendo i risultati parziali e i segnali di controllo presenti nella sua sinistra

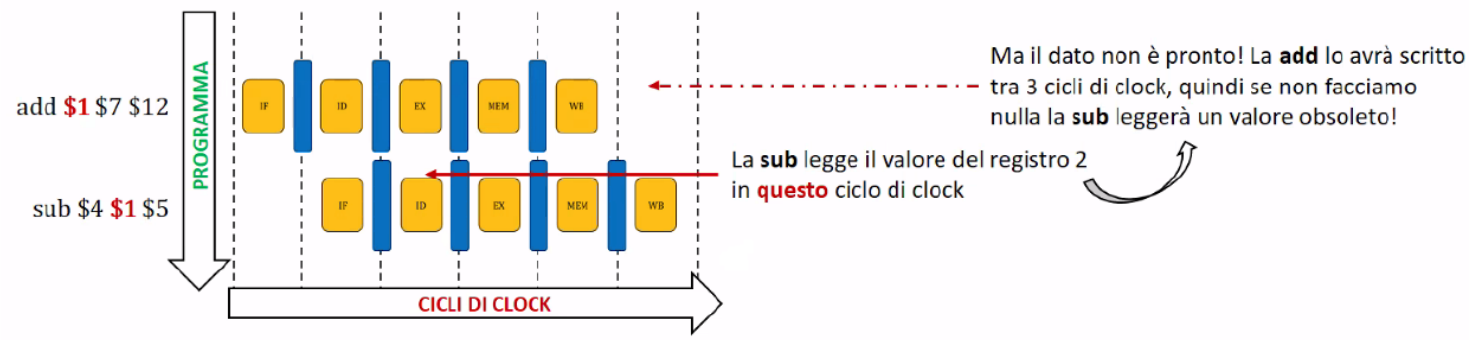


STRUCTURAL HAZARDS (criticità strutturali)

Risolte - ridefinizione del datapath con duplicazione delle risorse
- estensione del datapath e dei registri di transizione

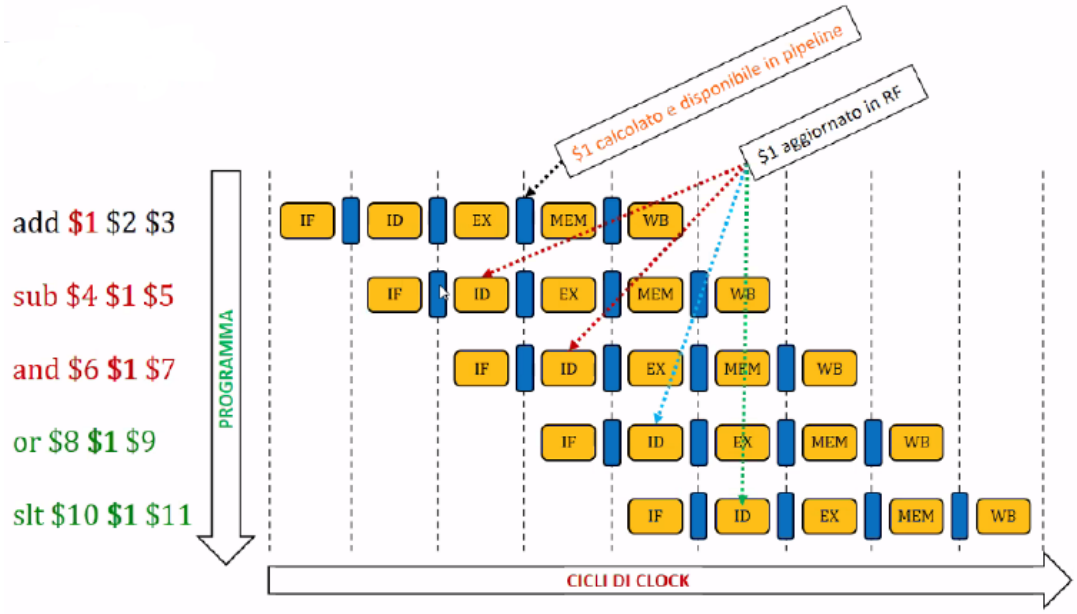
DATA HAZARDS (criticità di dato)

-le istruzioni non sono indipendenti l'una dall'altra
Ex1 Una sub potrebbe usare un operando che è risultato di una add precedente, che si non si trova ancora nella fase di WB
Ex2 Una jump potrebbe non essere eseguita se viene svolta un beq prima



HAZARD A/L

quando il dato è disponibile VS quando il dato è utilizzato



- le frecce rosse sono gli hazard critici, istruzioni che utilizzano valori sbagliati
(l'OR non causa hazard di dato: grazie ai register file che hanno la scrittura sul secondo fronte di clock (discesa), la or riesce a salvarsi e ricevere il valore della add)

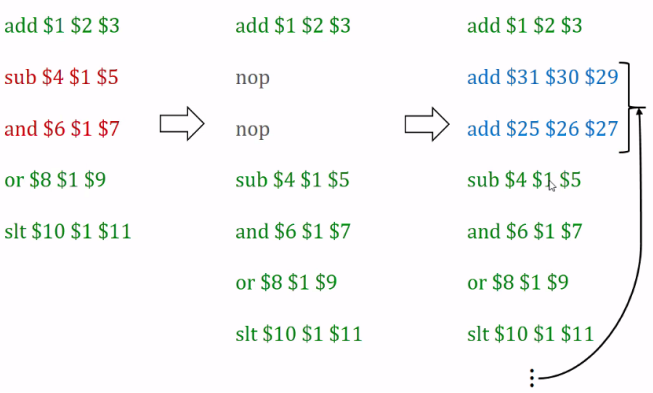
1) Prima soluzione: intervenire sul codice (operazione implementata dal compilatore)

A

- Introduco un'istruzione speciale *nop* => istruzione speciale che non provoca effetti sulla esecuzione (tipo add $0 $0 $0)
- approccio è piuttosto inefficiente (perdiamo un sacco cicli di clock perché sarebbe molto frequente)

B

- Provare a riorganizzare il codice, spostando operazioni che non introducono hazard nelle posizioni di nop;
- è possibile che non riusciamo a riempire tutte le nop



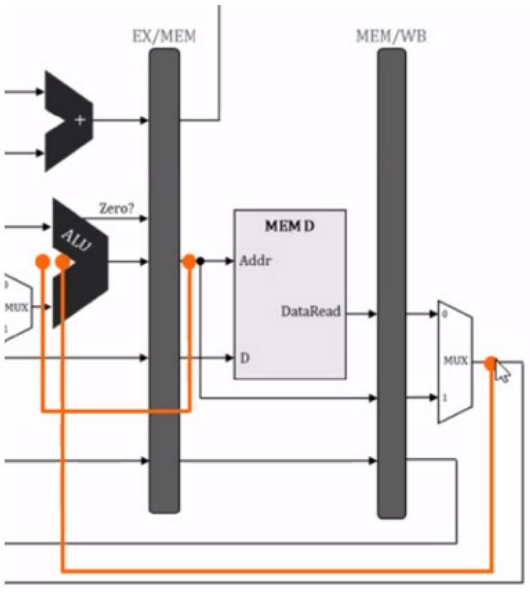
2) seconda soluzione: intervenire sull'hardware

Ex. Quanto l'AND entra in EX, nella fase ID/EX c'è il valore obsoleto di $s1, ma il valore è già presente nel registro MEM/WB

- posso implementare una anticipazione del valore, saltando la scrittura in MEM, e ricavando il valore direttamente dal registro

Introduciamo il forwarding (anticipazione)

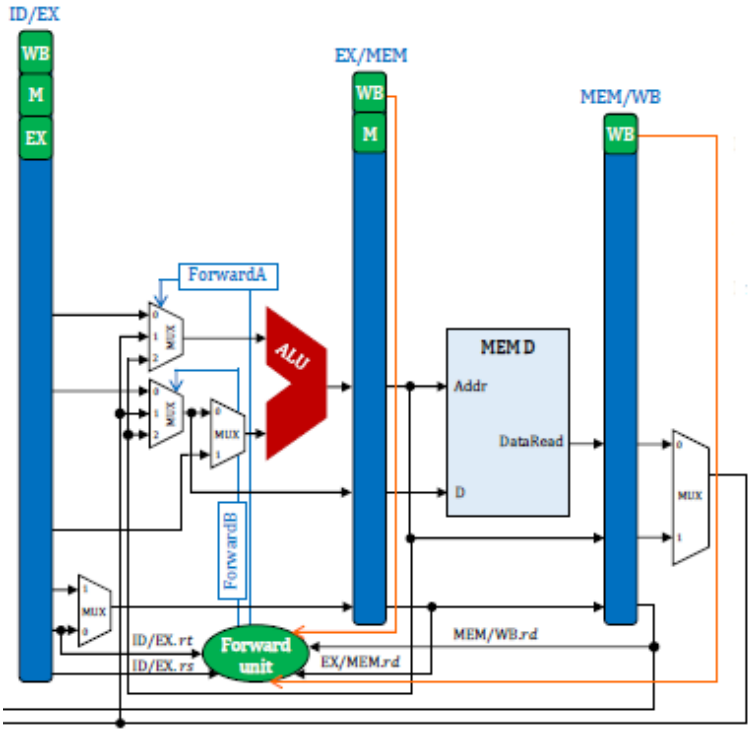
- colleghiamo EX/MEM.ALUOut alla ALU — EX/MEM -> EX
- colleghiamo il dato in uscita della fase di WB alla ALU — MEM/WB -> EX



- aggiunta di ulteriori operandi in input -> servono MUX con ulteriori segnali di controllo, comandati dalla CU

Come primo operando della ALU, possiamo avere 3 diversi dati:
1. valore di rs, quando non c'è anticipazione
2.le due possibili anticipazioni di rs:
- il dato da WB
- il dato dal registro di transizione EX/MEM.ALUOut

Come secondo operando della ALU abbiamo bisogno di una cascata di MUX, per gestire 4 possibili input:
- il valore immediate
- il valore rt e le anticipazioni:
1) il valore del registro rt
2) il dato da WB
3) il dato dal registro di transizione EX/MEM.ALUOut



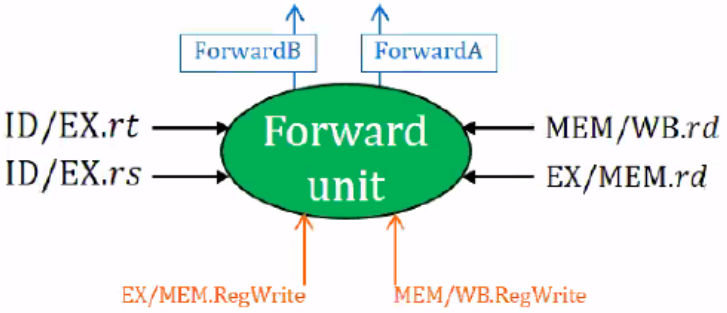
ci servono nuovi segnali di controllo:
forward unit -> manda segnali di selezione forward_A al primo multiplexer, e forward_B al secondo multiplexer

FORWARD UNIT
2 output segnali di selezione - forward_A
- forward_B

La forward unit viene implementata con una funzione logica combinatoria, con 4 possibilità, in AND fra 3 condizioni:

Conflitti tra istruzione attuale e la precedente
1) Il forward di A
- Se ID/EX.rs è uguale a EX/MEM.rd
- Se EX/MEM.RegWrite <- dobbiamo scrivere
- Se EX/MEM.rd è diverso da 0

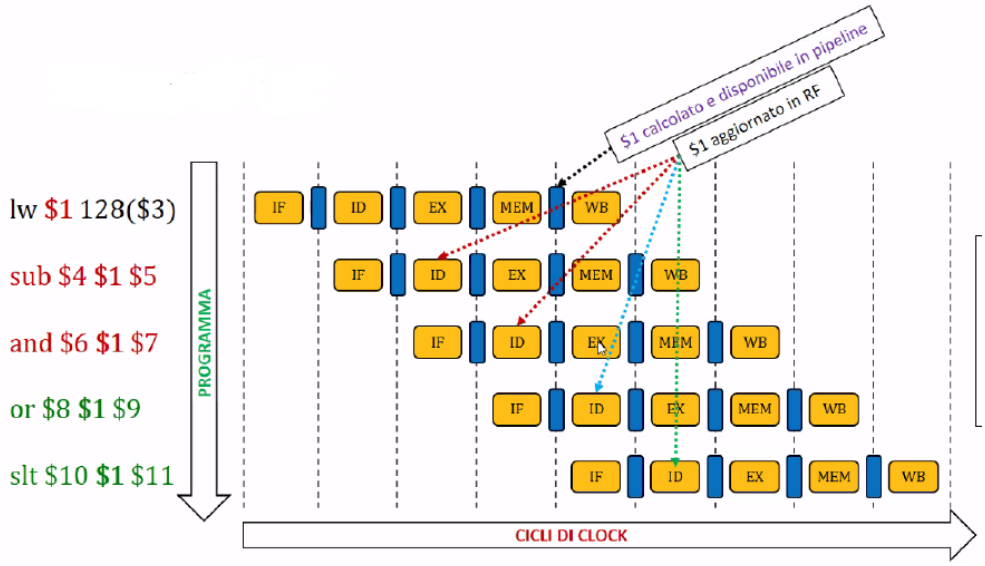
2) Il forward di B
- Se ID/EX.rt è uguale a EX/MEM.rd
- Se EX/MEM.RegWrite
- Se EX/MEM.rd è diverso da 0



Conflitti tra istruzioni attuale e precedente della precedente
3) il Forward di A
- Se ID/EX.rs è uguale a EX/WB.rd
- Se MEM/WB.RegWrite
- Se MEM/WB.rd è diverso da 0
4) il Forward di B
- Se ID/EX.rt è uguale a EX/WB.rd
- Se MEM/WB.RegWrite
- Se MEM/WB.rd è diverso da 0

HAZARD LW

Sempre un hazard sul dato, in questo caso prodotto nella fase di MEM -> l'istruzione di lw



-> quando SUB è in EX, in ID/EX c'è un valore obsoleto di $2, il valore corretto non esiste ancora

-> quando AND è in EX, in ID/EX c'è un valore obsoleto di $2, ma il valore corretto è in pipeline all'interno di MEM/WB

1) hazard sulla seconda istruzione che segue la lw, può essere risolta con un forwarding
Quando AND entra in fase di EX, lw sta svolgendo WB, abbiamo:
- ID/EX.rs = 1 (per AND)
- MEM/WB.rd = 1 (rt di lw)
- MEM/WB.RegWrite = 1 (lw che deve scrivere nella RF)

La Forward Unit risolve il problema → → →

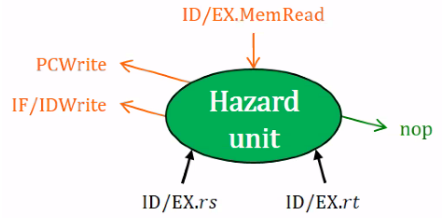
$$\text{if } rs == \text{EX/MEM}.rd \;\&\&\; \text{EX/MEM.RegWrite} \;\&\&\; \text{EX/MEM}.rd \;!= 0$$
$$\text{ForwardA} \leftarrow 10$$
$$\text{if } rt == \text{EX/MEM}.rd \;\&\&\; \text{EX/MEM.RegWrite} \;\&\&\; \text{EX/MEM}.rd \;!= 0$$
$$\text{ForwardB} \leftarrow 10$$

$$\text{if } rs == \text{MEM/WB}.rd \;\&\&\; \text{MEM/WB.RegWrite} \;\&\&\; \text{MEM/WB}.rd \;!= 0$$
$$\text{ForwardA} \leftarrow 01$$
$$\text{if } rt == \text{MEM/WB}.rd \;\&\&\; \text{MEM/WB.RegWrite} \;\&\&\; \text{MEM/WB}.rd \;!= 0$$
$$\text{ForwardB} \leftarrow 01$$

2) L'hazard della SUB non è risolubile, il dato non c'è ancora → si manda in stallo la pipeline, perdendo un ciclo di clock
Simile alla *nop*, ma vogliamo che sia fatto dall'hardware durante l'esecuzione

HAZARD UNIT

1) dobbiamo rilevare che c'è un hazard (nella fase di ID)
2) nello stesso ciclo di clock, devo scrivere nel registro di ID/EX segnali di controllo di una nop, anziché quelli dell'istruzione corrente in ID
3) nel ciclo successivo:
- IF e ID devono essere ripetute come al ciclo precedente
- EX svolge una nop
- MEM e WB continuano senza alterazioni



Se la condizione di IF è vera:
-> nel ciclo precedente è stata codificata una lw, che in questo ciclo dovrebbe fare EX
-> l'istruzione successiva j, che è ora ID, opera sul valore letto della lw (con rs o con rt)

$$\text{If ID/EX.MemRead} \;\&\&\; (\text{ID/EX}.rt == \text{IF/ID}.rs \;||\; \text{ID/EX}.rt == \text{IF/ID}.rt)$$
$$\text{Stallo di IF e ID}$$

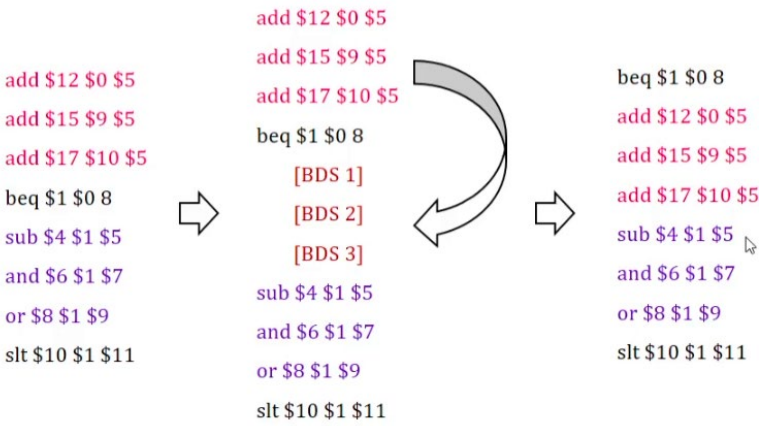
Stalliamo l'IF:
- PCWrite = 0, impedisco la scrittura in questo ciclo di clock del PC+4, nel prossimo ciclo ripete l'IF sulla stessa istruzione
- IF/IDWrite = 0, impedisco che l'istruzione i proceda verso la fase di decode; perché nel decode c'è già la j che deve essere decodificata (tanto la i viene riprelevata nel prossimo ciclo)
- nop = 1, nella fase di ID/EX vengono inseriti tutti i segnali di controllo pari a 0 dell'istruzione che va in avanti, nel prossimo ciclo di clock la CPU fa ex di una nop (tutti segnali di controllo = 0 -> nop) j deve rimanere ferma, mando avanti una copia di j (j rimane in IF/ID, e finisce pure in ID/EX

CONTROL HAZARDS (Criticità di controllo)

ex. - Un'istruzione j determina un salto nel flusso di controllo del programma (branch o jump)
- Un'istruzione i, successiva a j, non deve essere eseguita a causa del salto

-> il problema è che quando j sovrascrive il PC, l'istruzione i è già entrata in pipeline e la CPU la sta già elaborando, stiamo sprecando risorse

-> La beq effettua la sovrascrittura del PC con il BTA durante la fase di MEM
ma mentre questa sovrascrittura avvengono le istruzioni di sub and or



1) Prima soluzione: branch delay slot
Ogni volta che viene effettuata una branch, essa avviene in modo ritardato.
Questo ritardo lo utilizzo per riorganizzare il codice → Inserisco 3 slot vuoti sotto alla branch (BSD1, 2, 3)
In ogni BSD posso inserire altre istruzioni, che non generino conflitti

In questo caso le operazioni di add non generano conflitti dati, e devono essere eseguite comunque, le sposto nello slot delle BSD (Qualora la branch non viene presa, le istruzioni sono comunque eseguite).

Ci sono diversi approcci, in base a dove vengono estratte le operazioni:
- da prima del salto | from before (la prima utilizzata)
- dalla zona del salto | from target (scommettendo sul branch taken)
Anticipiamo le operazioni subito dopo la zona di salto, scommetto che prendo il salto e mi porto avanti
- dalla zona tra il beq ed il salto | from fall-through
Istruzioni che stanno tra la beq ed il salto, istruzioni prese dalla zona direttamente successiva scommettiamo sul fatto che la branch non viene presa

Sotto al branch, solitamente, il programmatore deve inserire istruzioni che non possono generare hazards

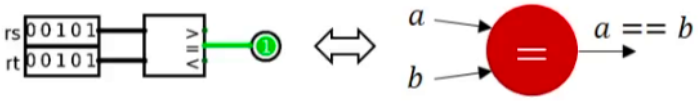
2) Seconda soluzione: approccio HW
2.a.) La CPU scommette su branch not taken
- branch not taken: non ci sono hazard
- branch taken: ho 3 istruzioni che non devono esserci, e devono essere cancellate

- Per cancellare una istruzione nella pipeline, settiamo tutti i segnali di controllo, delle istruzioni precedenti, a 0 (*nop*) trasformando i suoi segnali di controllo in nop

- Spostiamo l'esecuzione della branch nella fase di ID -> dobbiamo rilevare subito la criticità per prevenire stalli
la branch deve essere svolta in 1 solo ciclo (fase ID)

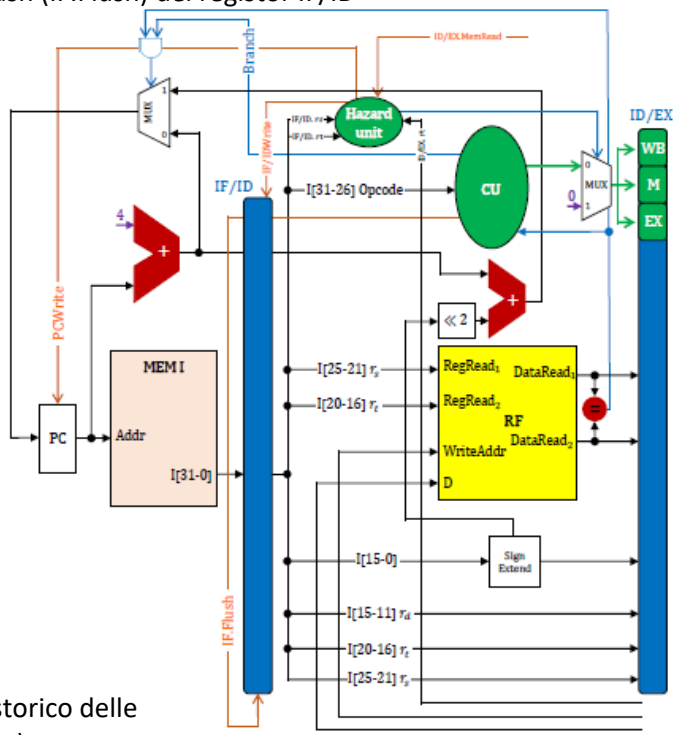
Ci serve una logica combinatoria per verificare condizione, e calcolare BTA
- un sommatore per calcolare il BTA, lo spostiamo nello stadio ID
- un comparatore dedicato alla verifica di uguaglianza (disegnato come un pallino, ma è il comparatore a 32 bit)

Qualora il branch debba essere preso, il comparatore attiva il multiplexer che seleziona il dato dal BTA, e il Control_Unit accende la linea di flush



Qualora il salto viene effettuato, devono svolgere anche una flush (IF.Flush) del registor IF/ID
-> sovrascriviamo una nop dell'istruzione successiva alla beq che l'istruzione sta prelevando

-> se indovina l'esecuzione prosegue normalmente
-> se sbaglia, paga con un flush (un ciclo di clock)



2.b.) La CPU scommette su Dynamic branch prediction
Branch Target Buffer -> memoria aggiuntiva che contiene uno storico delle branch più recenti (elenco di tutte le branch con l'esito del salto)

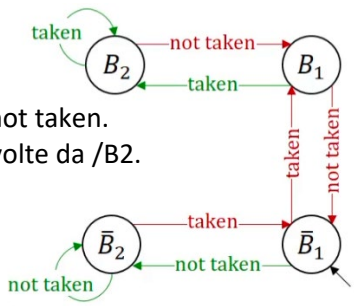
- Schema di predizione a 1 bit: 0 branch not taken, 1 branch taken
- Ogni branch è indicizzata in base la parte bassa dell'istruzione
Nelle istruzioni gli ultimi 3 bit sono quelli salvati nelle branch, ci bastano quelle per identificare le branch.
- Target address è l'indirizzo di salto, calcolato quando viene presa la branch
- Il BTB riceve in input il PC, e in base al valore presente nella memoria, e scommette in base ai valori di verità presenti nella memoria
- Il bit di salto è la predizione, la CPU scommette che l'esito del salto sarà uguale allo stesso dell'ultima volta che la branch è stata eseguita
Questo metodo ci permette di migliorare l'esecuzione dei loop (while, for, etc)

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

Ex.
La branch viene eseguita 15 volte ➔ Per 14 istruzioni, la CPU scommette che la branch è not taken, e per la 15 effettuiamo un flush.
Problema ➔ Cosa c'è di più frequente di un loop? Doverla ripetere più volte.
loop annidati ➔ Ogni ciclo mi costa 2 flush, che è tanto

2.c.) dynamic branch prediction a 2 bit
Si arriva ad avere solo 1, e non 2, flush per istruzione

- inseriamo lo stato (al posto del salto) → stato di una macchina di Moore, per ogni branch salvo una piccola MSF
Per gli stati negati, prendiamo una prediction con branch not taken
per gli stati veri predico un branch taken



Se mi trovo nello stato /B1 (not taken), se è vero, mi metto nello stato /B2, finché è not taken.
Quando invece sbaglio, mi sposto nello stato /B1, per arrivare a B1 devo sbagliare 2 volte da /B2.

Quindi Ogni ciclo mi costa 1, e non più 2

ECCEZIONI

Le eccezioni sono deviazione di flusso di controllo, dovuto da eventi non previsti, che può verificarsi internamente o esternamente alla CPU
È diverso dall'interrupt che è un'eccezione la cui causa si è verificata esternamente alla CPU (interrupt evento esterno)

Esempi di eccezioni →

EVENTO	ORIGINE	TERMINOLOGIA MIPS
Istruzione non riconosciuta: è stato svolto il Fetch di un'istruzione che la CPU non riesce a decodificare nella fase di ID	Interna	Eccezione
Overflow aritmetico: Un'istruzione aritmetico nella fase di EX ha generato un 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

Overflow aritmetico
L'hardware può riconosce quando questi sono avvenuti, e gestiti.
1. Quando in ID la CU legge un opcode non valido -> flag con istruzione non riconosciuta
2. Nella fase di EX quando la ALU svolge l'operazione si attiva il bit di overflow

GESTIRE LE ECCEZIONI

- trovare la causa dell'eccezione
- trovare l'indirizzo dell'istruzione colpevole che ha generato l'eccezione (offending instruction)
- raccolte le informazioni, trasferiamo il controllo ad un blocco di istruzioni che gestisce l'evento, programma chiamato dell'exception handler, che fa parte nel sistema operativo

L'exception handler gestisce la situazione, cercando di ripristinare l'esecuzione del programma:
- stampare un messaggio di errore
- terminare l'esecuzione del programma
- ripristinare l'esecuzione del programma

- rilevata l'eccezione, l'indirizzo dell'istruzione successiva alla offending instruction all'interno di un registro speciale chiamato EPC (Exception Program Counter)
Nel ciclo di clock in cui avviene l'eccezione, questo indirizzo PC + 4, calcolato al ciclo precedente, che nel ciclo di clock corrente è all'interno al registro IF/ID

Possiamo ora gestirla in 2 modi:

1) gestione basata sul registro di stato (registro causa in MIPS)
- Dopo aver rilevato l'eccezione, scriviamo all'interno del registro causa un codice speciale
codice 10 (Reserved Instruction) usata per estendere l'ISA con nuove operazioni non supportate in modo nativo

- Viene quindi fatto un salto non condizionato all'indirizzo 0x800000180 dove risiede la prima istruzione dell'exception handler (chiamata a procedura)
L'handler legge il codice dell'eccezione dal registro causa, e agisce di conseguenza

Eccezione	Codice
Istruzione non riconosciuta	10
Overflow aritmetico	12

2) gestione basata su interrupt vector table (IVT)

l'hardware ad ogni eccezione effettua un salto ad un indirizzo diverso a seconda dell'eccezione

È una tabella dove data una causa di eccezione, si ottiene l'indirizzo a cui saltare per cedere il controllo all'handler (non usato in MIPS)

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

Una volta che l'handler ha gestito l'eccezione:
- il programma può essere terminato
- l'esecuzione del programma viene ripristinato

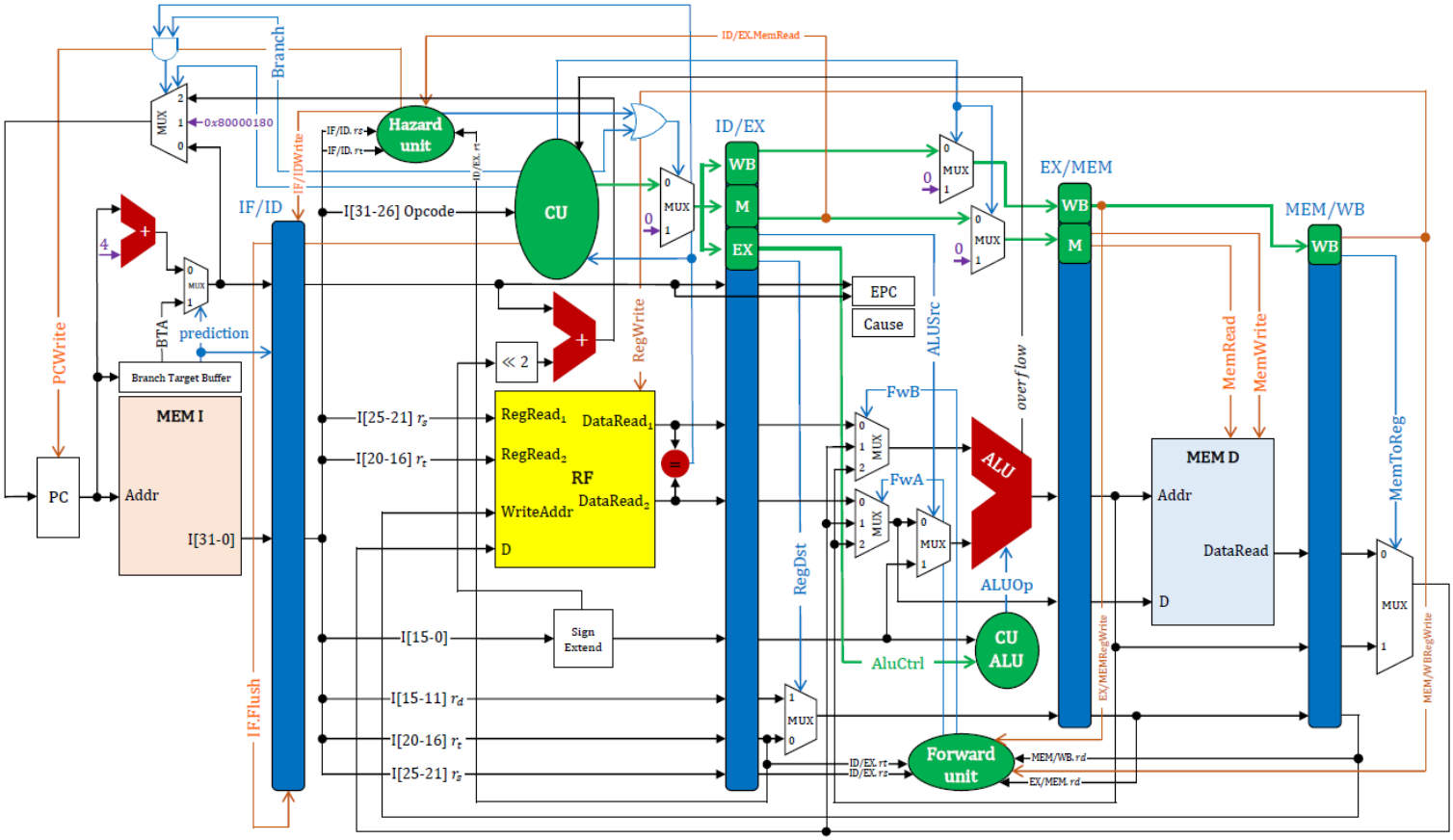
1. aggiungiamo i registri EPC e Cause

2. Estendiamo la CU in modo che:

a) se durante la fase di ID l'opcode è diverso da quelli riconosciuti, scriva in IF/ID, EPC e il codice 10 in cause
la CU deve scartare la offending instruction dalla fase ID e la successiva che ha completato la fase di IF

b) se durante la fase di EX la ALU attiva il bit di overflow -> scriviamo in ID/EX, EPC e il codice 12 in cause
la CU deve scartare la offending instruction dalla fase di EX e le 2 successive che hanno completato la fase di ID e IF

3. Aggiungiamo 0x800000180 tra i segnali da poter scrivere nel program_Counter -> salto condizionato speciale, non condizionato



...

0x40 add $12 $12 $13

0x44 add $12 $12 $13

0x48 add $12 $12 $13 → Overflow

0x4C and $1 $2 $4

0x50 beq $12 $0 $1

Valori di questi registri?

PC	EPC	0x4C
0x54	Cause	12

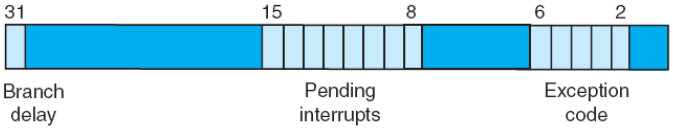
La scelta più logica sarebbe quella di non fare nessun flush, ma eseguire l'interrupt handler e ripristinare dall'indirizzo nell'istruzione che era contenuto nel PC prima che iniziasse la successiva fase di IF -> dipende dall'architettura

COPROCESSORE 0

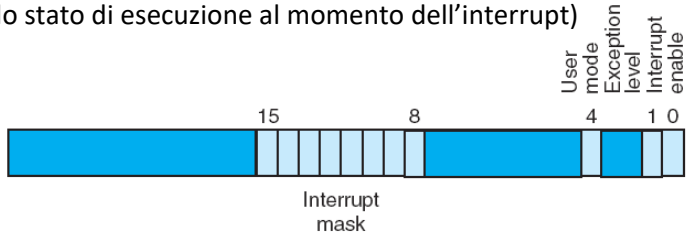
Nel MIPS, un coprocessore 0 è l'unità dedicata alla gestione delle eccezioni

Registers | Coproc 1 | Coproc 0

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 l'exception code (unsigned int, bit 2-6). Indica anche gli interrupt che al momento dell'eccezione erano ancora da servire
- registro EPC: Exception Program Counter
- registro BadVadd: indirizzo errato nel caso di una address exception store che cerca di entrare in un indirizzo malformato
- registro status: interrupt mask e bit di controllo (32 bit che indicano lo stato di esecuzione al momento dell'interrupt)



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 A LANCIO MULTIPLO

Aumentare l'efficienza ulteriormente -> aumentare il parallelismo a livello di istruzione

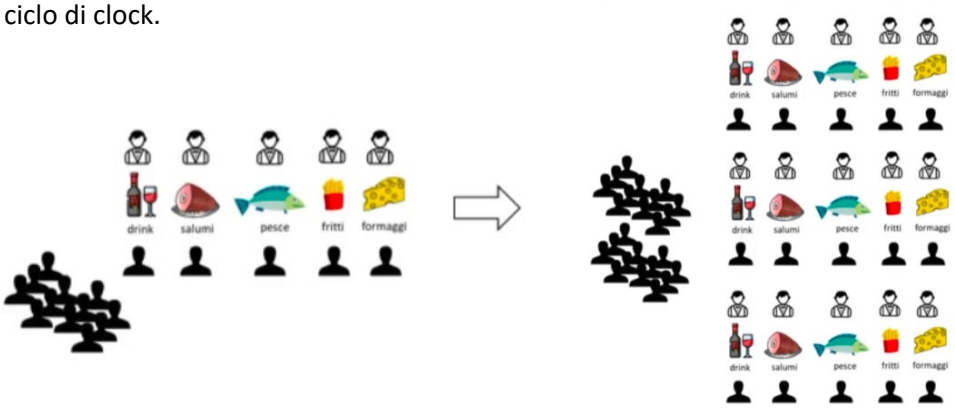
PARALLELISMO A LIVELLO DI ISTRUZIONE

Introduciamo ILP = instruction level parallelism (numero di istruzioni in esecuzione contemporanea, nel nostro caso 5 stadi) unità di misura che caratterizza la performance della pipeline

Vogliamo aumentare l'ILP aumentando il numero massimo di numero di istruzioni in esecuzione 2 approcci:

1) Approccio orizzontale => aumentare gli stadi della pipeline
   - Ogni stadio svolge micro-operazioni più elementari, la durata del diminuisce ed il throghput aumenta
   - Non è proprio applicabile nel nostro caso, nelle CPU con ISA più complesse è applicabile.
2) Approccio verticale => duplichiamo le risorse hardware per permettere a più istruzioni entrare nella pipeline
   - si chiama una pipeline multi-issue (a lancio multiplo) a k vie
   - Il numero delle vie è anche chiamato IPC (Instruction per clock-cycle), numero massimo di istruzione che la pipeline è in grado di lanciare in un ciclo di clock.

Partendo dall'esempio dei camerieri ed i banconi, aggiungiamo più code.



- Il lancio multiplo implica la necessità di considerare nuovamente problemi legati a data e control hazards
- L'issue slot è "la rampa di lancio delle istruzioni", in una pipeline a k vie ad ogni ciclo di clock dobbiamo riempire uno slot di k istruzioni che verranno messe in esecuzione nella pipeline contemporaneamente.
- quando non è possibile riempire completamente lo slot con k istruzioni, queste verranno lasciate vuote o occupate da nop

L'organizzazione ad ogni ciclo di clock, dei lanci multipli, evitando conflitti ed errori è chiamato scheduling, che può essere:

3) software => delegato al compilatore, multi-issue statica (SW)
   - Il compilatore è consapevole dell'architettura di cui si occupa, ed organizza le istruzioni opportunamente
   - Statica perché non è facilmente modificabile
4) approccio hardware => delegato alla CPU, multi-issue dinamica (HW)
   - Questi approcci non sono mutualmente esclusivi, sono solitamente presenti entrambi

NAMING CONVENTION IN 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

problema nello scheduling -> non posso sapere a priori gli effetti di un'istruzione prima dell'esecuzione

SPECULAZIONE

agire sotto assunzioni di cui non si ha certezza di realizzazione: -> qualora le assunzioni si realizzino, abbiamo dei vantaggi
-> qualora le assunzioni non si realizzino, abbiamo dei costi

Questo aspetto si traduce in due requisiti operativi per gli approcci SW o HW:
1) dobbiamo controllare se la speculazione è andata a buon fine
2) se non è andata buon fine -> dobbiamo attuare una procedura di recovery (rollback)

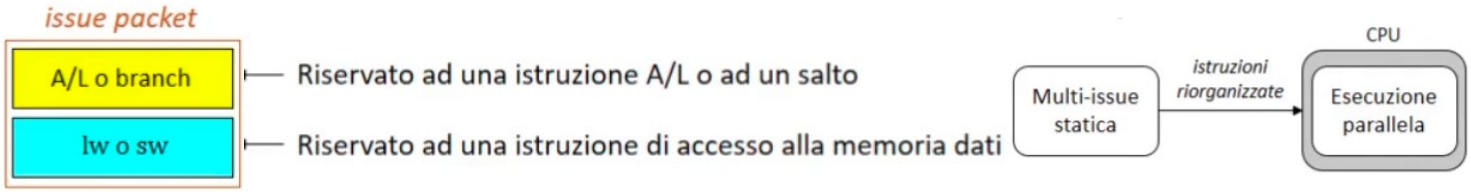
- per la cpu: i risultati delle istruzioni speculate vengono temporaneamente mantenute in un buffer, fino a quando:
  - esito positivo, scriviamo i risultati in RF o memoria, in un'operazione chiamata commit
  - esito negativo, scartiamo i risultati e rimandiamo in esecuzione con le istruzioni corrette
- per il compilatore: bisogna implementare delle istruzioni speciali che controllino l'esito della speculazione, e in caso negativo, riparare gli errori

Se un'istruzione genera un'eccezione aspettiamo a servire l'eccezione fino a che la speculazione non sia risolta

MULTI-ISSUE STATICA | SW

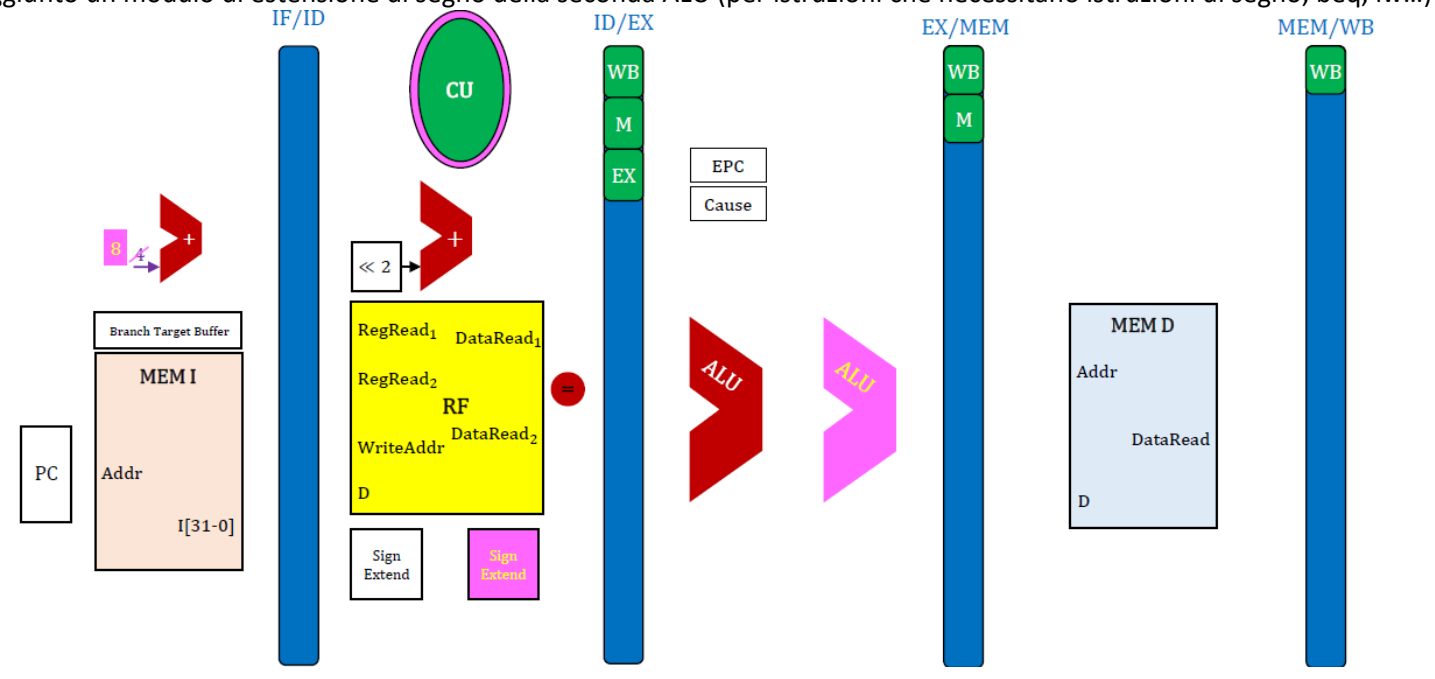
Consideriamo una CPU con pipeline multi-issue a 2 vie

l'issue packet non può essere arbitrario, vengono imposti vincoli in modo da semplificare la gestione degli hazards



Assumendo che la CPU esegua dei fetch da 64 bit (2 istruzioni) Il compilatore deve riordinare il codice in modo che:
- garantisca che non ci siano hazard tra le istruzioni all'interno dell'issue packet
- cercare di minimizzare gli hazard tra issue packets diversi (nel caso la CPU rileva e risolve hazard tra issue packets diversi)

Andiamo a duplicare solo i componenti che veramente ci servono:
- Il program_Counter fa il fetch del $PC + 8$
- I registri pipeline per poter contenere i risultati parziali di 2 istruzioni
- La CU deve decodificare 2 istruzioni alla volta
- due ALU, nel caso vengano effettuate 2 EX contemporaneamente
- aggiunto un modulo di estensione di segno della seconda ALU (per istruzioni che necessitano istruzioni di segno, beq, lw…)



MULTI-ISSUE STATICA (SCHEDULING)

lw $t0 0($t1)
addi $t1 $t0 4
sw $t2 0($t0)
issue packet

← la lw impone uno stallo al ciclo di clock successivo che però impatta due istruzioni potrei metterlo in stallo entrambi

la lw non può usare il risultato della add, se si trova accodata con lei nello stesso packet -> dobbiamo trovare un sostituto da inserire al posto della lw →

addi $t1 $t0 4
lw $t2 0($t1)
issue packet

EX. → → →

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

1) Primo packet, il primo spazio è giallo quindi ci serve una aritmetico logica:
- add non va bene, usa il risultato della lw
- subi causerebbe un errore nella sw (perché ha bisogno di $s1, rompiamo la sequenza di codice)
- bne usa il risultato di sub

dobbiamo utilizzare una nop. Nel secondo spazio (azzurro) si inserisce load word

nop
lw $t0 0($t1)

add $t0 $t0 $s2
nop

2) Secondo packet: iniziamo inserendo una add, l'altra operazione possibile è una operazione dati, tuttavia la sw usa il risultato di add, inseriamo una nop.

3) Terzo packet: possiamo inserire la sw perché $t0 è pronto, e la subi non ci crea problemi. Entrambi utilizzano $s1, tuttavia le istruzioni non hanno bisogno di attendere l'esecuzione dell'altra

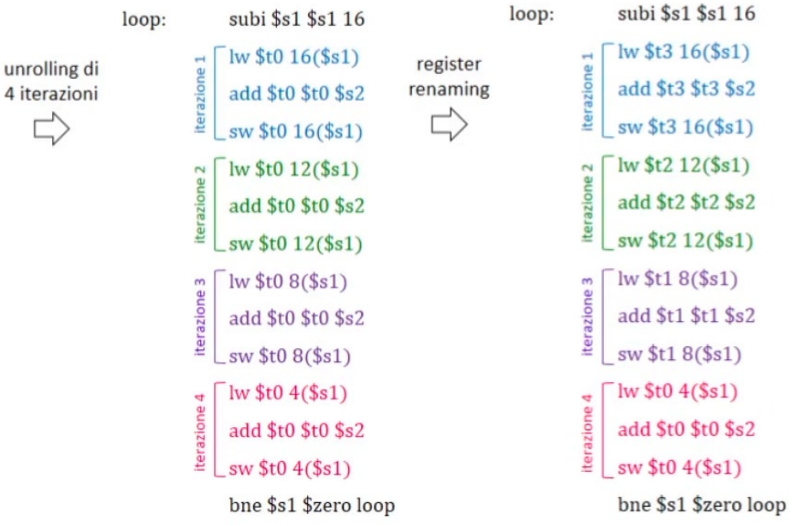
subi $s1 $s1 4
sw $t0 0($s1)

bne $s1 $zero loop
nop

4) Quarto packet: inseriamo la bne

NO BUONO → ABBIAMO SFRUTTATO IL VANTAGGIO SOLO 1 SU 4 VOLTE

MULTI-ISSUE STATICA (LOOP UNROLLING)



Supponiamo che analizzando il codice, il compilatore sia in grado di scoprire che il ciclo deve essere eseguito almeno 4 volte.

Loop unrolling, stiamo esplicitando le prime iterazioni del ciclo, con un codice che abbia lo stesso effetto; le istruzioni successive non cambiano

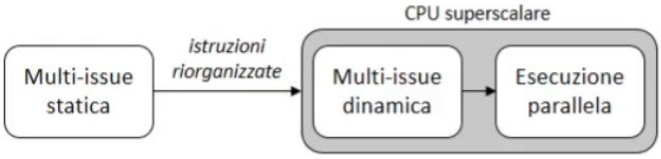
Register renaming elimino le dipendenze che dipendono all'uso dello stesso registro ma non dello stesso dato

Con loop unrolling

A/L o branch	Accesso a memoria
subi $s1 $s1 16	lw $t3 **0**($s1)
	lw $t2 12($s1)
add $t3 $t3 $s2	lw $t1 8($s1)
add $t2 $t2 $s2	lw $t0 4($s1)
add $t1 $t1 $s2	sw $t3 16($s1)
add $t0 $t0 $s2	sw $t2 12($s1)
	sw $t1 8($s1)
bne $s1 $zero loop	sw $t0 4($s1)

MULTI-ISSUE DINAMICA | HW

Le decisioni su come riempire gli slot sono prese dall'hardware (detti processori superscalari)



- APPROCCIO BASE:
  - Decido di volta in volta quante istruzioni entrano
  - le istruzioni vengono lanciate nello stesso ordine in cui sono allocate nel segmento testo.
  - La cpu, per ogni ciclo, decide se lanciarne nessuna, lanciarne una o lanciarne più di una
    -> l'hardware decide la sequenza, una dopo l'altra, e la cpu decide quante eseguirne

- la decisione di scheduling dipende dai vincoli di packaging delle istruzioni.
- Hazard di tipo diverso possono limitare la parallelizzazione.
- La riorganizzazione del codice è fortemente legata allo specifico processore, e non all'ISA.
  -> cambiato il processore, va ricompilato il codice.

- Il compilatore supporta il lavoro riorganizzando le istruzioni in modo da diminuire gli hazard.
- La CPU garantisce sempre la correttezza di esecuzione, anche quando la riorganizzazione del compilatore genererebbe errori

- APPROCCIO CON SCHEDULING DINAMICO:

L'approccio base viene arricchito lasciando la possibilità all'hardware di riordinare le istruzioni: scheduling dinamica della pipeline.
L'hardware riordina le istruzioni attraverso l'architettura -> quando entrano le istruzioni possono essere riorganizzate opportunamente, garantendo la correttezza dell'esecuzione, migliorando le prestazioni e prevenendo gli stalli:

Prelievo e decodifica (in-order)

- Le istruzioni vengono prelevate una alla volta, nell'ordine in cui sono presenti nel segmento testo.
- Queste vengono decodificate, per identificare eventuali hazard

Esecuzione (out-of-order)

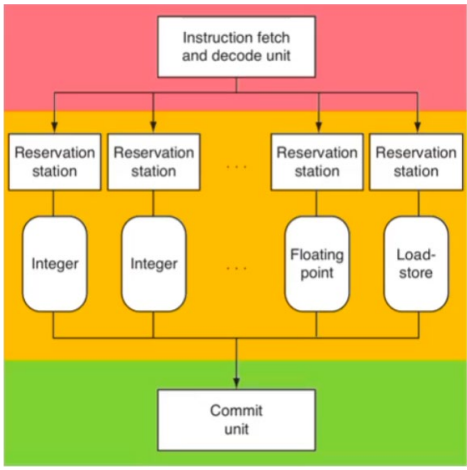
- una volta decodifica l'istruzione, viene inviato verso il banco di lavoro.
- Il banco di lavoro è composto da n unità funzionali:
  - ogni unità funzionale svolge una particolare tipo di istruzione, lavorando in parallelo con le altre
  - siccome possono terminare in tempi diversi, può accadere un riordinamento delle istruzioni

Commit (in-order)

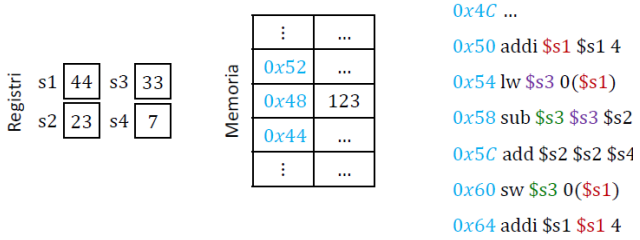
- buffer per I risultati delle istruzioni, che verranno scritti nel RF o in memoria nell'ordine corretto
- oppure viene inviato a tutte le altre reservation station che le richiedevano

- Ogni unità funzionale ha un buffer d'ingresso, chiamato reservation station, che contiene l'operazione che l'istruzione vuole svolgere, e il valore degli operandi su cui deve svolgere l'operazione

- Se per via di uno o più hazard gli operandi non sono disponibili nell'RF, il valore dell'operando non è copiato e l'ingresso nell'unità funzionale è posto in attesa.
- Se uno o più operandi non sono disponibili, l'istruzione si mette in attesa, aspettando che essa vengano caricate.

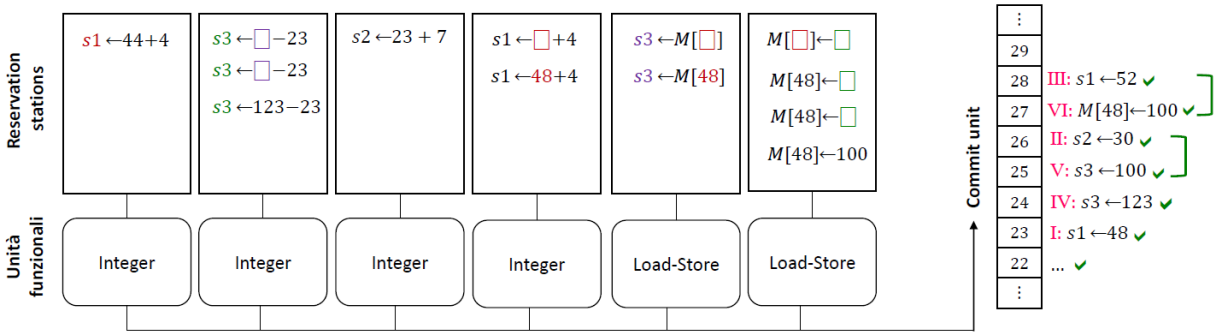


EX.



Dati i seguenti valori in registri ed in memoria; e le seguenti istruzioni:

esecuzione attraverso la multi-issue dinamica:



La multi-issue dinamica è in grado di gestire fino a 6 istruzioni contemporaneamente (4 con operazioni tra operandi intereger, 2 operazioni di load-store)

Le istruzioni vengono inserite tutte contemporaneamente, ma vengono suddivise in operazioni:
- che possono essere eseguite direttamente ($0x50$ *e* $0x5c$)
- che hanno dipendenze da operazioni precedenti (tutte le altre)

La commit unit contiene i risultati delle operazioni, ma non le finalizza, le tiene solo in un buffer, effettuando il commit quando possibile -> non ha dipendenze future, è nell'ordine giusto.

Una volta eseguito il primo ciclo, possono essere eseguite le istruzioni $0x54$, $0x60$ *e* $0x64$
Una volta eseguite le istruzioni, la commit unit effettua uno scan per verificare se ci sono operazioni committabili -> effettua il commit (commit-unit[23])

I vantaggi dello scheduling dinamico
- ci permette di fare scheduling a runtime => possiamo svolgere lo scheduling utilizzando informazioni sullo stato dell'esecuzione del programma, a differenza di farlo in fase di compilazione
- la branch-prediction dinamica (prediction in base agli ultimi salti) viene fatta a runtime. Non è quindi possibile stabilire un ordine delle istruzioni a compile time, dato che non si conosce la predizione.
- lo scheduling dinamico ci permette di concentrarsi solo sull'ISA, ma anche con implementazione diversa. È in grado di supportare architetture diverse, con stesso ISA senza generare errori.

Ma ci sono anche dei limiti
Ci sono casi che possono limitare l'ILP in una pipeline multi-issue, per esempio l'aliasing tramite puntatori


ARM Cortex A8


Intel Core i7 920

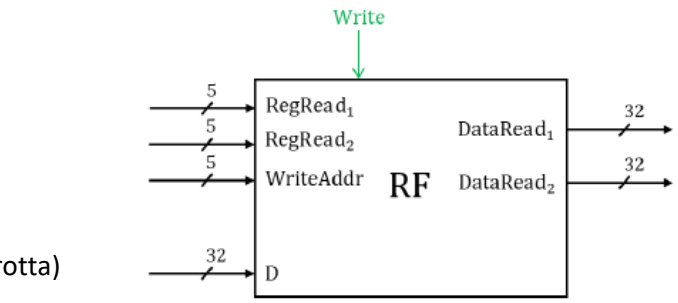
Target	Dispositivi mobili	Server, Cloud, Workstation
TDP*	2 W	130 W
Ciclo di clock / Frequenza	1 ns / 1 GHz	0,37 ns / 2,66 GHz
Multi-issue?	Sì, dinamica	Sì, dinamica
IPC	2	4
Stadi pipeline	14	14
Scheduling pipeline	Statico (esecuzione in-order)	Dinamico (esecuzione out-of-order) con speculazione

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

MEMORIA

-> è un componente fondamentale, ed ogni accesso “costa”

Le 3 memorie principali che abbiamo analizzato sono:
- MEM I: la memoria che contiene le istruzioni
- MEM D: memoria che contiene i dati su cui lavorano le istruzioni
- Register File: banco di lavoro della CPU

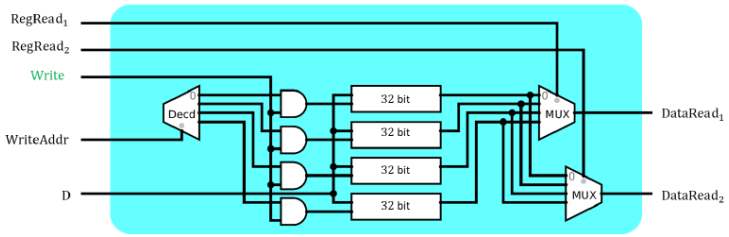


REGISTER FILE (su tecnologia SRAM)

- Memoria di lavoro della CPU
- Memoria volatile (dati svaniscono quando l’alimentazione è interrotta)

Il tempo di accesso dipende da:
- in lettura, dal cammino critico del decoder
- in scrittura, dal cammino critico del multiplexer

Più aumenta il numero di registri più aumenta il cammino critico



SRAM (Static Random Access Memory)

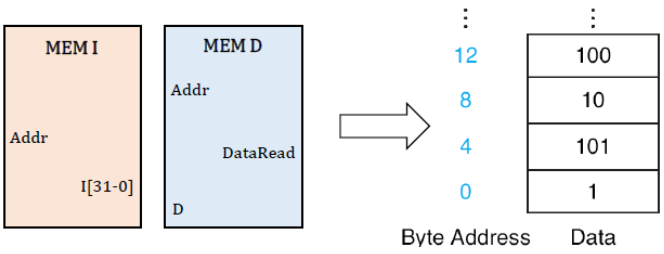
Random Access => lettura e scrittura hanno costi simili (qualsiasi posizioni accedi, il costo è sempre lo stesso)
Static => non è necessario aggiornare le celle di memorie per mantenere il dato
-> dato scritto, memoria accesa -> il dato non degrada/decade, rimane

Solitamente per memorizzare 1 bit si utilizzano da 6 a 8 transistors (per garantire una robustezza dei dati).
Le performance risultano alte, ad un costo elevato

Main Memory (su tecnologia DRAM)

- Composta dalla memoria istruzioni e la memoria dati (gestita dal sistema operativo)
- Memoria volatile.
- Modellata come un array unidimensionale, dove ogni elemento è una parola di memoria, a cui è associata un indirizzo.

In MIPS gli indirizzi di 32 bit sono codificati sui singoli byte, parole di 4 byte, con allineamento su multipli di 4.



DRAM (Dynamic Random Access Memory)

Random access => letture e scritture hanno costi simili, non dipendono da dove il dato è fisicamente immagazzinato
Dynamic => è continuamente necessario aggiornare le celle di memoria, per mantenere il dato
il refresh di una cella di memoria avviene con: 1) lettura del valore della cella
2) riscrittura del valore letto

Tipicamente avviene ogni 65 ms, e nel caso il dato debba cambiare nella fase 2) subentra il nuovo valore al posto di quello vecchio.

Per memorizzare 1 bit viene utilizzato 1 transistor -> abbiamo performance più basse, ma si abbassano anche i costi.

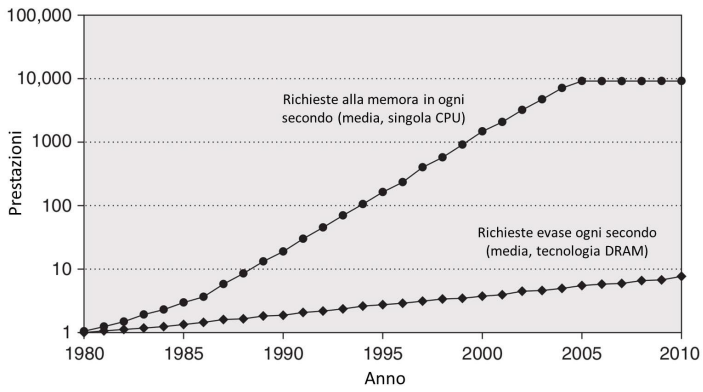
SDRAM

- DRAM con integrato un clock per sincronizzare automaticamente la memoria con la CPU

DDR SDRAM

- DDR (Double Data Rate), in grado di svolgere le operazioni su entrambi i fronti di clock

PROCESSOR-MEMORY BOTTLENECK



Sviluppo tecnologico nei processori MAGGIORE rispetto a quelli generati nel campo delle memorie.

Accessi alla memoria consistono l'attività principale delle CPU (90% del tempo)

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)

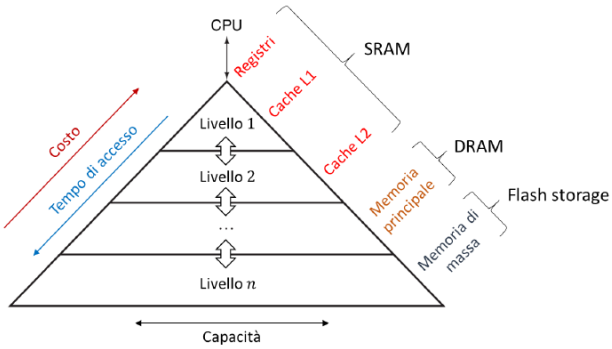
Nome	Abbreviazione	Fattore
KibiByte	KiB	2^{10}
MebiByte	MiB	2^{20}
GibiByte	GiB	2^{30}
TebiByte	TiB	2^{40}
PebiByte	PiB	2^{50}
ExbiByte	EiB	2^{60}
ZebiByte	ZiB	2^{70}
YobiByte	YiB	2^{80}

strutturare il sistema di memoria in modo che aumenti l'efficienza del lavoro sfruttando due principi:
- Principio di località temporale: molto spesso le operazioni vengono svolte più volte sulle stesse cose (aumentare il valore di una variabile)
- Principio di località spaziale: molto spesso le operazioni vengono svolte su dati che sono fisicamente vicini

GERARCHIA A MEMORIA

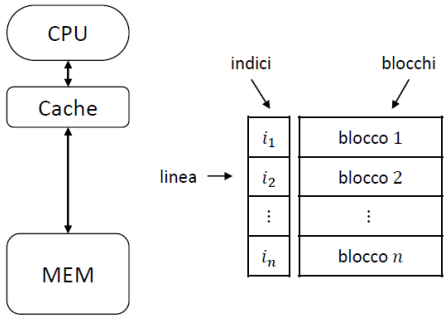
- La memoria è progettata secondo una gerarchia a livelli
- Il trasferimento di dati può avvenire solo tra livelli adiacenti

Le richieste di accesso sono gestite attraverso la gerarchia
- la richiesta di un dato dal livello i, il dato è contenuto al livello i?
  - se sì, richiesta è soddisfatta
  - se no, la richiesta è inoltrata al livello i + 1 (in basso)



CACHE

La cache è modellabile come un array monodimensionale, dove ogni elemento è una linea indicizzata da un indice che contiene un blocco 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

Nella memoria	Nella cache
ogni parola è identificata dall'indirizzo, mappato su byte (ogni 4)	ogni parola è indicata dall'indice di linea

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

MAPPATURA DIRETTA

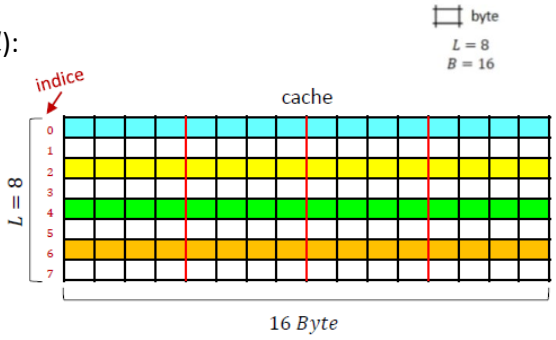
Tipo di cache dove il blocco di memoria è assegnato in modo univoco per ogni linea di cache
- competizione sulla linea (più blocchi su una linea)
- accesso al dato

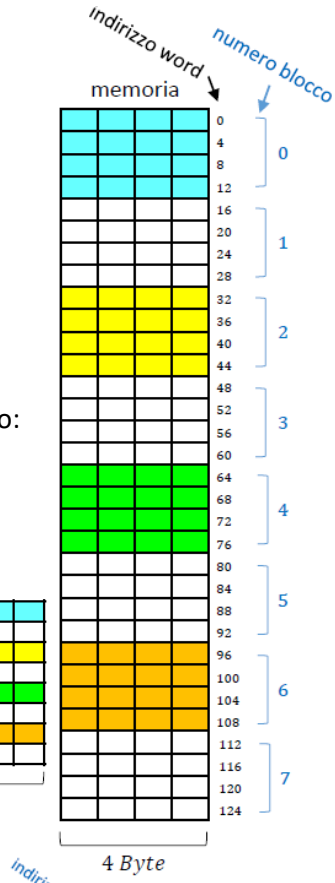
come andiamo a determinare l'indice di linea I(d) per un dato dato d?

Ad ogni blocco in memoria principale associamo un indice univoco della cache
Bisogna tuttavia notare che la corrispondenza non è biunivoca, il numero di blocchi in memoria è maggiore del numero di linee della cache, ogni indice di linea è condiviso da più blocchi in memoria centrale.

La soluzione è quello di determinare il numero di blocco in memoria, partendo dall'indirizzo di d ovvero:
$N(d) = div(M(d), B)$

Determinare l'indice di cache a cui è assegnato il blocco $N(d)$:
$Id{=}mod(N(d), L)$





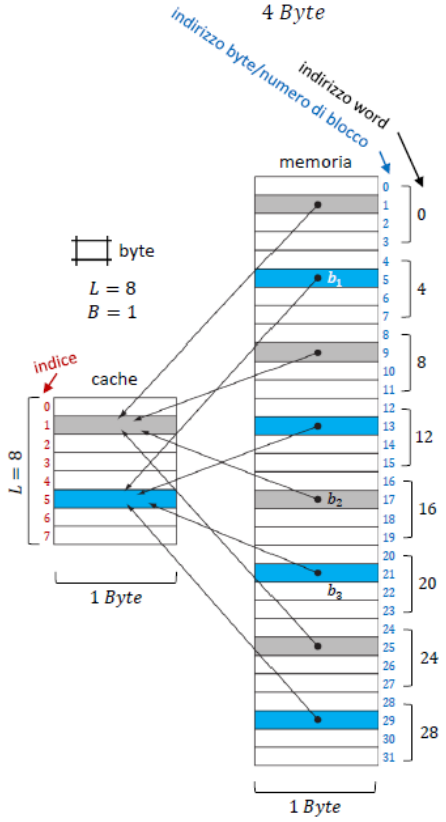
Ex1.
- numero di blocco dell'indirizzo 20?
N(d) = div(M(d), B) -> N(d) = div(20, 16) -> N(d) = 1

- numero di blocco dell'indirizzo 92?
N(d) = div(92, 16) -> N(d) = 5

Ex2. Esercizi di mappatura

a quale indice di linea di cache dovrebbero trovarsi i byte $b\#$, $b\$$, $b\%$?
- abbiamo 8 linee, ed i blocchi sono grandi 1 (solitamente multipli di 4)
- un blocco contiene un byte -> un indirizzo in memoria coincide col numero di blocco.

La posizione in memoria di $b1$ è 5 - $M(b1)$ = 5 - $N(b1 = div(5,1)$ = 5 - $I(b1) = mod(5,8)$ = 5 Il byte 5 finisce nella riga 5	La posizione in memoria di $b2$ è 17 - $M(b2)$ = 17 - $N(b2) = div(17,1)$ = 17 - $I(b2) = mod(17,8)$ = 1 Il byte 17 finisce nella riga 1
La posizione in memoria di $b3$ è 22 - $M(b3)$ = 22 - $N(b3) = div(22,1)$ = 22 - $I(b3) = mod(22,8)$ = 6 Il byte 17 finisce nella riga 6	



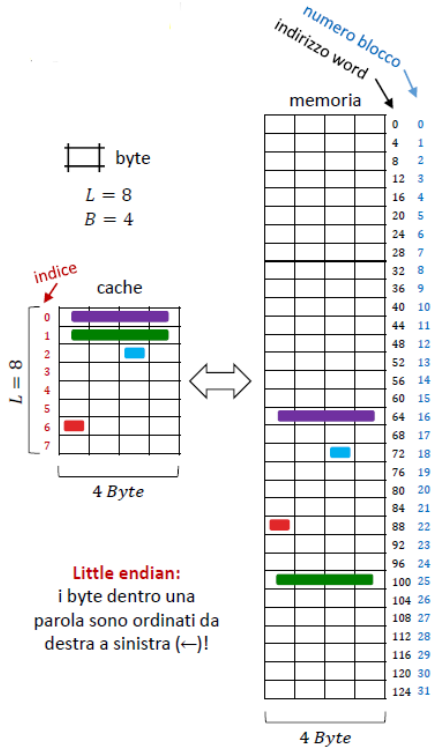
Ex3. Esercizio di mappatura diretta

i blocchi sono di 4 byte -> ci troviamo in situazione di little endian

A quale indice si trova la parola con indirizzo 100?
- $Indirizzo\ del\ dato\ M(d)$ = 100
- $numero\ di\ blocco$: $N(d)$ = div(100,4) = 25

In questo caos il resto è zero, il resto rappresenta l'offset di byte
all'interno del blocco.
Dato che è 0, esso rientra nel primo byte al blocco 25
- $Indice\ di\ cache\ I(d)$ = mod(25,8) = 1

Il byte 100 finisce nella riga 1



A quale indice si trova la parola con indirizzo 64?
- $M(d)$ = 64
- $numero\ di\ blocco$: $N(d)$ = div(64, 4) = 16

In questo caos il resto è zero, il resto rappresenta l'offset di byte
all'interno del blocco.
Dato che è 0, esso rientra nel primo byte al blocco 16
- $I(d)$ = mod(16,8) = 0

Il byte 64 finisce nella riga 0

A quale indice si trova il byte con indirizzo 73?
- $M(d)$ = 73
- $numero\ di\ blocco$: $N(d)$ = div(73,4) = 18

Il resto è 1, quindi rientra nel secondo byte dentro al blocco 18.
Ricordiamo che dato che è little endian, contiamo da destra
- $I(d)$ = mod(18,8) = 2

Il byte 73 finisce nella riga 2

A quale indice si trova il byte con indirizzo 91?
- $M(d)$ = 91
- $numero\ di\ blocco$: $N(d)$ = div(91,4) = 22

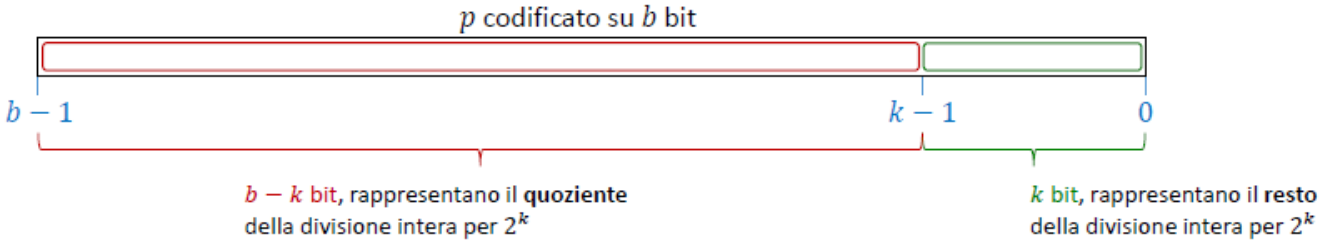
Ha un resto di 3, l'offset dentro al blocco 22 è 3
- $I(d)$ = mod(22,8) = 6

Il byte 22 finisce nella riga 6

DIVISIONE INTERA PER POTENZA DELLA BASE

Le cache non hanno mai dimensioni arbitrarie
- il numero delle linee sarà sempre una potenza di 2: L = 2"
- la dimensione del blocco è definita da un numero intero di parole (se ci sono 2^m parole, allora B = 2^{m+2})
- I calcoli, quando svolti dalla CPU, sono divisioni intere tra un numero intero binario senza segno, e una potenza di 2.

Date queste presupposizioni possiamo dire che:
Dato un numero 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
- il resto della divisione è dato dalla k cifre meno significative di p

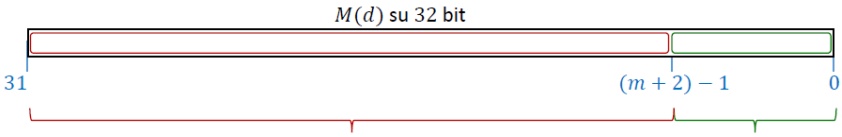


Lo shift a destra rappresenta la divisione per la base (in questo specifico caso)

ACCESSO ALLA CACHE

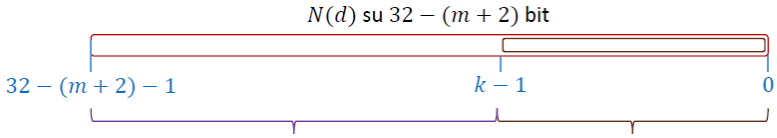
Sfruttando questo principio possiamo formalizzare l'accesso alla cache in maniera molto più semplice:
Consideriamo una cache con $L = 2^k$ e $B = 2^{m+2}$ come calcoliamo l'indice di linea di un dato d?

- Calcoliamo il numero del 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!**

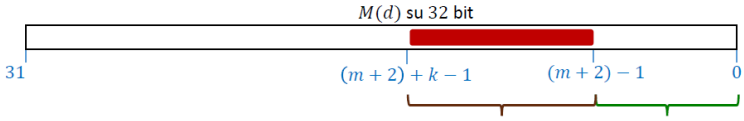


- Calcoliamo l'indice di cache

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

La CPU per ricavare la posizione in cache quindi ha solo bisogno di estrarre dei bit

Data una cache con L linee ciascuna contenenti blocchi da B byte e dato M(d) per calcolare la posizione in cache



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

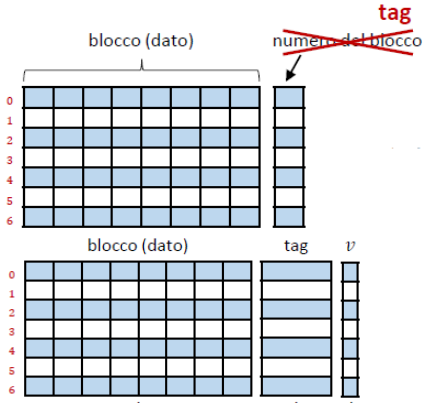
PROPRIETÀ MAPPATURA DIRETTA

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*)
Quindi tutti gli indirizzi che condividono la porzione rossa competono per la stessa posizione in cache

Con queste 3 informazioni ($indice$, $offset\ di\ parola$, $offset\ di\ byte$) costituiscono una singola richiesta alla cache

dentro ad una linea di cache, come possiamo stabilire se il blocco che abbiamo richiesto è effettivamente giusto, visto che può essere condiviso da più blocchi di memoria? Oppure il blocco potrebbe non essere valido (non inizializzato, obsoleto …)

- nella linea di cache aggiungo un'informazione aggiuntiva chiamato tag
- aggiungo un bit di validità



Possiamo quindi quantificare il costo totale della cache.
Quanta memoria in byte totale ci serve, per una cache con L linee e dimensione di blocco B?

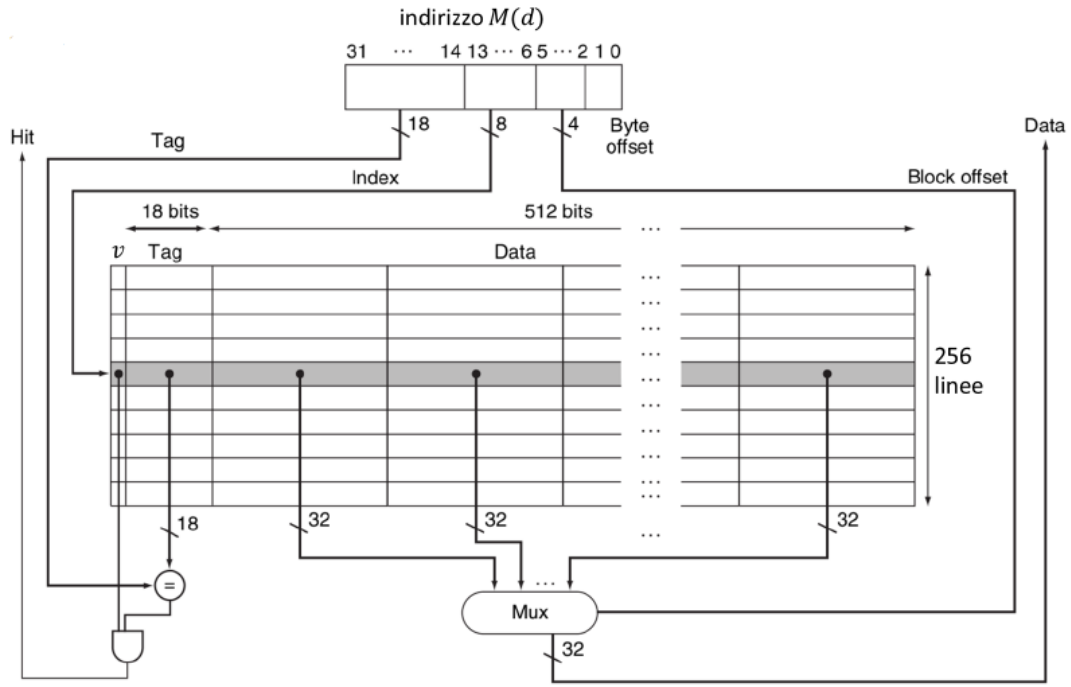
$$D_{tot} = L \times \frac{(8B + (32 - \log_2 B - \log_2 L) + 1)}{8}$$

Ex.
dimensionare e implementare una cache con D_{data} = 16 KiB, dove ogni blocco contiene 16 parole di memoria

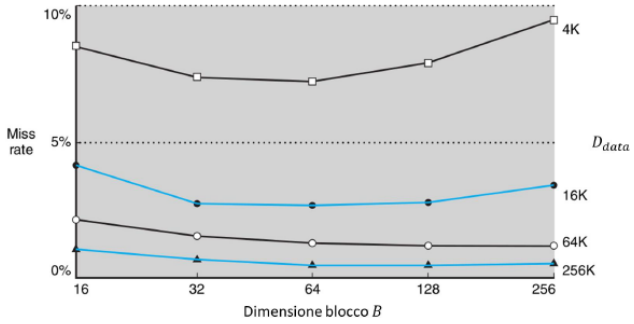
Formalizziamo i dati:

Trasformiamolo in una base 2	$D_{data} = 16KiB = (16 * 2^{10})B = 2^{14}B$
Siccome contiene 16 parole	$B = (16 * 4)B = 64B = 2^6B$
Linee di cache	$L = \frac{D_{data}}{B} = \frac{2^{14}B}{2^6B} = 2^8 = 256$
quanti bit per indice di linea?	$k = \log_2 L = \log_2 2^8 = 8$
quanti bit per l'offset dentro al blocco?	$B = 2^{m+2}$ $m = \log_2 B - 2 = 6 - 2 = 4$
quanti bit per il tag?	$32 - (m+2) - k = 32 - 6 - 8 = 18$
Dimensione totale	$D_{tot} = 256 * \frac{8 * 64 + (32-6-8) + 1}{8} = 16992B \rightarrow 16{,}6\ KiB$

Servono quindi 608 B in aggiunta alla capacità dati



Nella realtà si dimensiona la cache aumentando la dimensione del blocco, sfruttiamo meglio la località spaziale, tuttavia se il blocco è troppo grande rispetto alla dimensione della cache, ci sarà molta competizione per le linee di cache tra i vari blocchi.



Blocco più grande > abbiamo tanti bit, ma se c'è una miss la paghiamo di più

GESTIONE DELLA MISS (CACHE-MISS E ACCESSI IN SCRITTURA)

Quando c'è una hit, il dato viene prelevato dalla cache -> cosa succede quando c'è una miss (dato corretto e valido)?

Assumiamo che un'istruzione richieda l'accesso in lettura ad un dato d
1. Accesso alla cache in linea $I(d)$
2. Check sul bit di validità, confronto tag con parte alta di $M(d)$ → $cache\ miss$
3. 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. CPU riparte, cache hit

Le fasi da 3 a 7 (compresi) vengono chiamate miss-penalty, operazioni spese dovute a una miss

ACCESSI IN SCRITTURA

Assumiamo che un'istruzione richieda accesso in scrittura ad un dato d
- Il primo step è il medesimo, è necessario verificare che $N(d)$ è disponibile in cache

1) Se è disponibile abbiamo un write hit, l'operazione di scrittura viene svolta sulla copia del dato nella cache
2) se il dato non è disponibile abbiamo un write miss, che gestiamo con la procedura precedente, così da avere un write-hit

Problema -> coerenza della cache -> dopo l'operazione di scrittura il dato in cache è aggiornato, ma la sua versione corrispondente in memoria è obsoleta
-> dobbiamo andare a ripristinare la coerenza tra cache e memoria principale.

WRITE-THROUGH	WRITE-BACK
Ogni volta che un dato viene scritto in cache, viene aggiornato subito dopo in memoria La coerenza è forzata ad essere sempre rispettata Questo tuttavia produce un problema di tempo, tutto il punto della cache è di parlare solo con la cache, se propago in questo modo è come se parlassi con la memoria Posso implementare un write-buffer, che propaga gli aggiornamenti alla memoria	Le operazioni di scrittura avvengono solo in cache (e non in memoria) La coerenza è ristabilita qualora il dato debba lasciare la cache; quando il blocco viene rimpiazzato, prima della sovrascrittura si aggiorna la sua versione in memoria

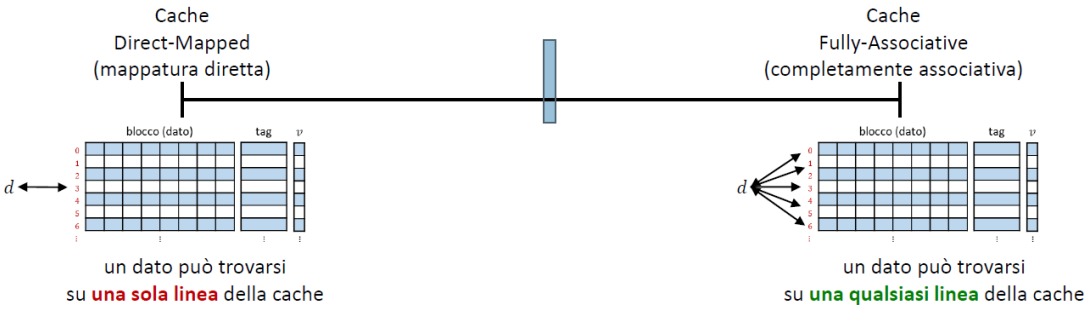
CACHE ASSOCIATIVE

Nelle cache a mappatura diretta, ogni dato ha 1 solo possibile punto assegnabile, definito dall'informazione presente nella meta-informazione.

Ma se gli indirizzi possibili sono molteplici?
- L'accesso non avviene tramite un indirizzo, ma utilizzando una parte del contenuto del dato che lo identifica univocamente chiamato chiave.

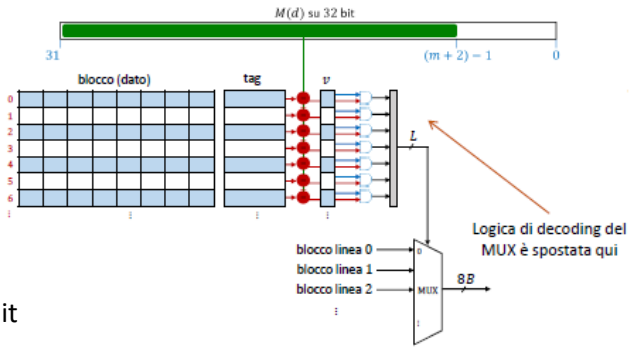
Nella memoria il contenuto di una linea di cache che identifica univocamente il blocco è il tag

N.B.
- l'indice di linea non c'è più, è come se ci fosse una sola linea $L = 1$
- Il tag è dato da tutti $32 - (m + 2)$ bit del numero di blocco



FULLY ASSOCIATIVE CACHE

Dato un $M(d)$ su 32 bit in entrata, il tag viene estratto => i bit $(m + 2) - 1$



- Il tag viene confrontato con tutte le linee di cache, e viene dato un risultato positivo su al più una linea.
- Il bit di test di uguaglianza va in AND con il bit di validità della linea corrispondente -> a partire del tag accedo a solo 1 linea
    - Se il dato non è presente, nessun comparatore emette hit
    - se il dato è presente, un solo comparatore emette hit
- Il MUX seleziona la porta di lettura con la linea di cache su cui c'è stata la hit

$linea\ di\ cache$ = $bit\ di\ validit$à + tag

$linea\ di\ cache \rightarrow MUX$

Approccio efficiente (il cache miss c'è solo quando è pieno), veloce e versatile
Però architettura costosa, in quanto ci serve una logica di test, e ogni linea di cache deve contenere più informazioni.

Calcolo dei dati con la memoria cache fully-associative

Quanti dati può contenere D_{data} una cache completamente associativa dove il blocco contiene
- 16 parole di memoria, e D+2+ = 33.6875 KiB

L = 1
m + 2 = $\log_2(4 \text{ x } 16) = 6$ (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 x 16 x 8 = 512 per il dato
- 26 bit per il tag
- 1 bit di validità

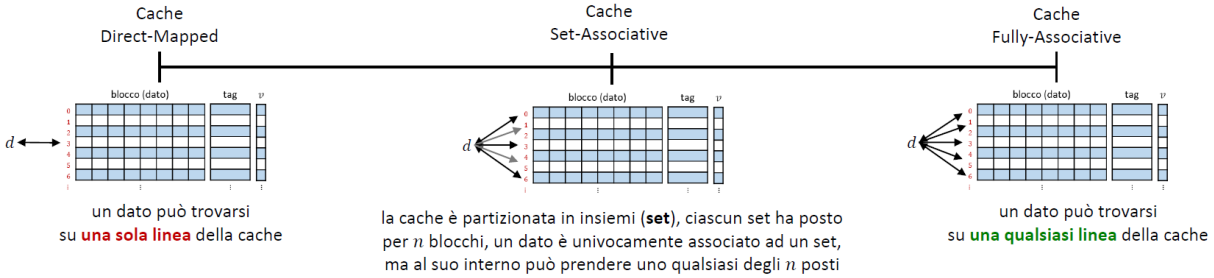
Capacità di numero di blocchi: $\frac{33.6875 \times 2^{10+3}}{539} = 512$

CACHE A SET ASSOCIATIVA

Via di mezzo tra la cache direttamente mappata e cache completamente associativa.

- determinazione del set (linea): avviene col metodo direct-mapped col calcolo dell'indice
- determinazione del posto (banco): metodo fully-associative con test di uguaglianza del tag, check sul bit di validità

La cache è partizionata in insiemi chiamati set, in cui ciascuno ha posto per n blocchi
Un dato è univocamente associato ad un set, ma al suo interno può posizionarsi in uno qualsiasi degli n posti.



I set sono gruppi di posizioni, che messi insieme formano la cache

- l'insieme dei set devono formare la cache
- all'interno del set è completamente associativa
- la selezione del set è direttamente mappata

Ma come lo calcoliamo L?

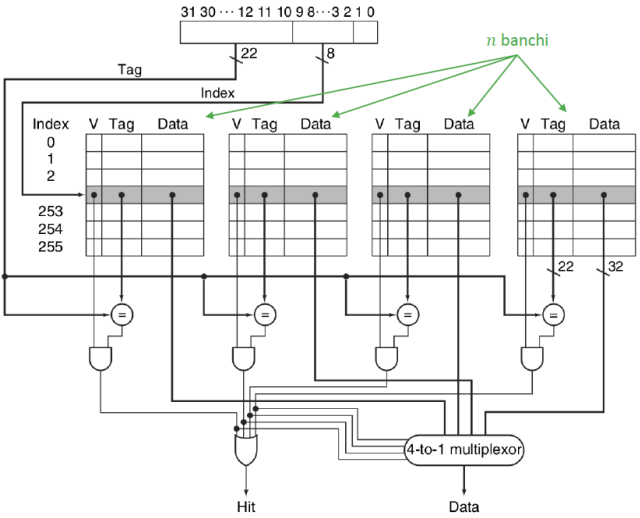
Direct-mapped: L = $\frac{D_{data}}{B}$ set (linee) da n = 1 posto, chi è assegnato a quel set (linea) può occupare solo quel posto

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 interamente associativa
- se invece n = 1 allora è direct-mapped)

Possiamo quindi generalizzare la formula $D_{\text{data}} = L * n * B$

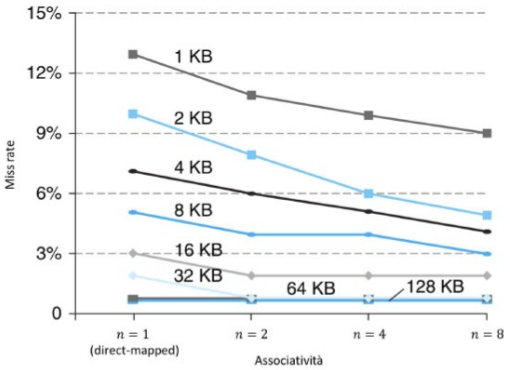


Ex.
una cache associativa a 4 vie, da KiB, e blocchi da 1 parola
Le posizioni colorate in grigio sono il set, ed è composto da 4 posti, ed è in posizione 3.
questi 4 posti sono le posizioni in cui il dato può essere presente

- qual è il numero dei set → $L = \frac{4 + 2^{10}}{4 + 4}$ = 256 quindi k = $\log_{\$}$ 256 → 8 bit di indice
- tag → su 22 bit 32 − 2 − 8 = 22
- m + 2 = $\log_2 B$ = 2 → m= 0 (non c'è un offset, c'è solo l'offset di byte dato che ogni blocco coincide con una parola)

Quale cache conviene di più?

- L'aumentare delle vie (n), comporta in generale un abbassamento del missrate
- i benefici sono man mano decrescenti
- se la missrate è già bassa, i benefici sono Limitati



Ex1. (directly-mapped)

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

cache da 16 B (D_{data}), directly-mapped con L = 4, B = 4
k = 2 |
m = 0 | -> con questi dati possiamo calcolare la posizione in cache dei dati presenti in 48, 28, 16
tag su 28 bit |

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)

$M(d)_2$ dovrebbe essere su 32 bit, tuttavia ne scriviamo solo 8 perché oltre sono tutti 0
Le istruzioni che vogliamo svolgere sono le seguenti:

Alla prima esecuzione dell'istruzione
- lw $s0 16($zero$) -> abbiamo una miss, perché il bit di validità è 0 -> viene aggiornato il dato nella tabella

- $lw$$s1 48($zero$) -> abbiamo una miss, perché il campo tag è errato
dovrebbe valere 3, infatti tag_2(48) è 0011
Viene aggiornato il dato nella tabella

- lw $s4 16($zero$) -> abbiamo un'altra miss, il tag è di nuovo errato
dovrebbe valere 1 ma vale 3, viene riaggiornata la tabella

Siamo in una situazione di forte competizione della prima linea della tabella di cache

- lw $s3 28($zero$) -> abbiamo un'altra miss, il bit di validità è 0
Non è ancora stato inizializzato, lo inizializziamo

- lw $s1 48($zero$) -> abbiamo ancora una miss
la linea è occupata, dobbiamo riaggiornarla

Abbiamo avuto quindi 5 operazioni, con 5 miss. Avevamo una forte competizione per la prima linea

cerchiamo di risolvere il problema introducendo gli accessi "set-associative"

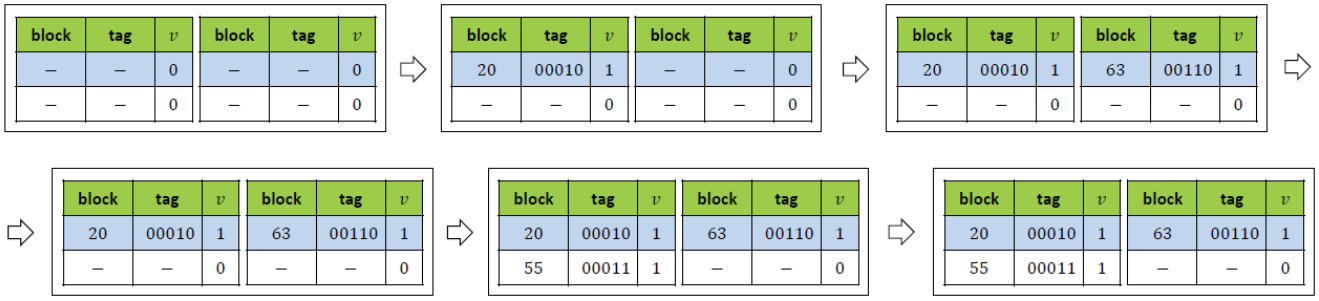
Ex2. Set-associative

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!



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

- lw $s0 16($zero) e lw $s1 48($zero) sono entrambe miss, ma se assegniamo in maniera intelligente i dati nelle tabelle, possiamo diminuire le miss-rate
- lw $s4 16($zero) finalmente risulta una hit, il dato è presente nella tabella
- lw $s3 28($zero) è ancora una miss, ma lw $s1 48($zero) è una hit

Abbiamo migliorato da 5 miss, a 2 hit e 3 miss

Ex3. Accessi fully-associative

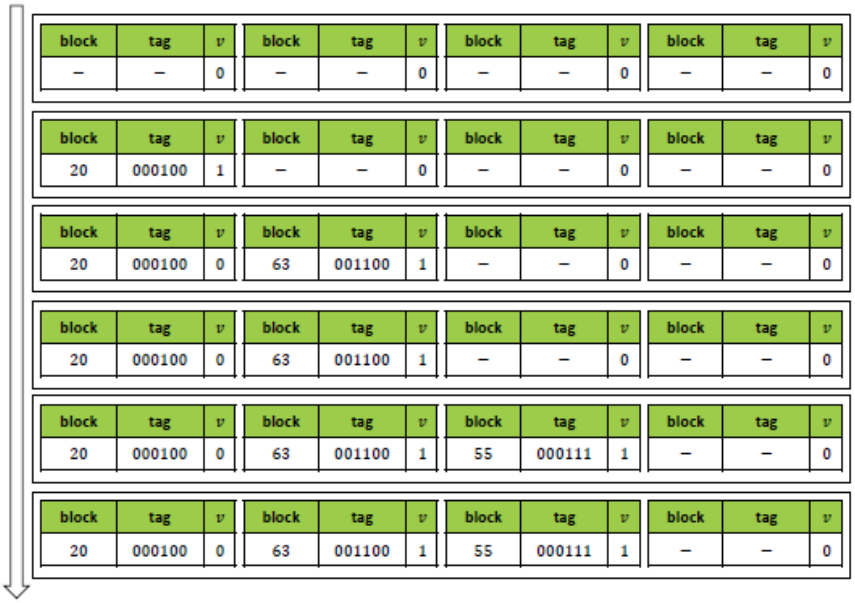
3) Cache da 16B, fully associative, L = 1, B = 4 (un unico set da 4 posti)
k = 0
m = 0
tag da 30 bit

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

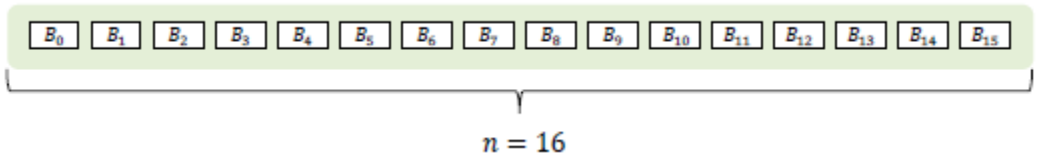


Riusciamo a vedere che le miss iniziali, dovute al cold-start non possono essere evitate.
È una deficienza strutturale, dovuto al design della cache, non possiamo evitarla

SOSTITUZIONE DEI BLOCCHI

Quando un blocco è trasferito in un set, quale dei set presenti viene sovrascritto?

consideriamo una cache con 16 vie, consideriamo un singolo set



Se tutti i 16 blocchi sono occupati, e arriva un blocco in entrata, quale dei 16 viene sovrascritto?

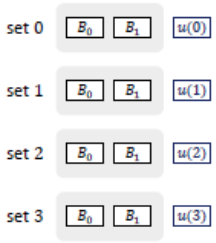
- a random = soluzione semplice da implementare, e con alti gradi di associatività (con tante vie) può funzionare bene
- LRU (Least Recently Used) = sostituiamo il blocco che non viene utilizzato da maggior tempo
- Pseudo-LRU = LRU approssimato

Non viene scelto sempre quello meno utilizzato, ma sceglie uno prossimo ad esso (penultimo, terzultimo etc)

LRU

Abbiamo bisogno di tener traccia in ogni set di quale blocco non è stato usato da più tempo

-> è facile da implementare se abbiamo una bassa associatività, ma diventa esponenzialmente più difficile quando aumentano le vie



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

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

- Possiamo utilizzare un approccio basato su "use bits", con n = 2, ogni set s ha un singolo use bit $u(s)$

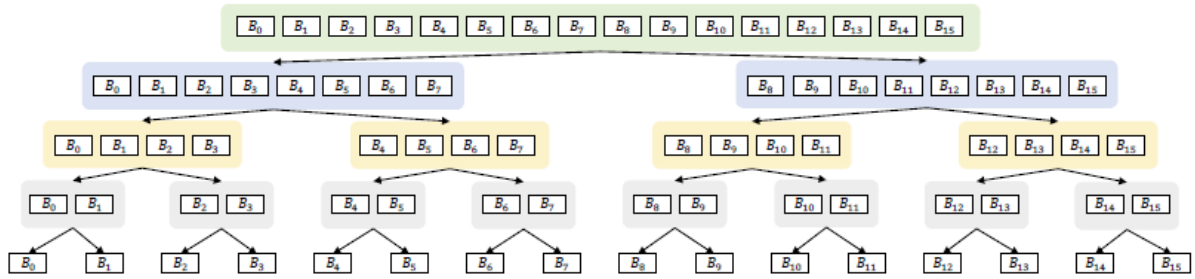
Questo approccio costa 1 bit per ogni set di cache

-> l'ultimo accesso implicitamente localizza la prossima scrittura; conoscere il blocco acceduto più recentemente equivale a conoscere anche quello non acceduto da più tempo.

Cosa succede se ci sono più di 2 blocchi? Non mi bastano 2 bit

PSEUDO-LRU

basato su un albero binario di ricerca



Allochiamo 1 bit su ogni biforcazione, 0 via sinistra, 1 via destra (con n vie ci servono n-1 bit)

- inizialmente tutte le biforcazioni sono a 0 -> ad ogni nodo lungo il percorso dobbiamo scegliere destra o sinistra?
- ad ogni accesso, per ogni biforcazione attraversata da quell'accesso:
  - se siamo andati a sinistra, settiamo il bit a 1 (per l'LRU bisogna andare a destra)
  - se siamo andati a destra, settare il bit a 0 (per l'LRU bisogna andare a sinistra)

Questo bit settato viene utilizzato dal successivo, che effettua l'operazione medesima

non abbiamo un LRU preciso, però ci permette di bilanciare gli accessi

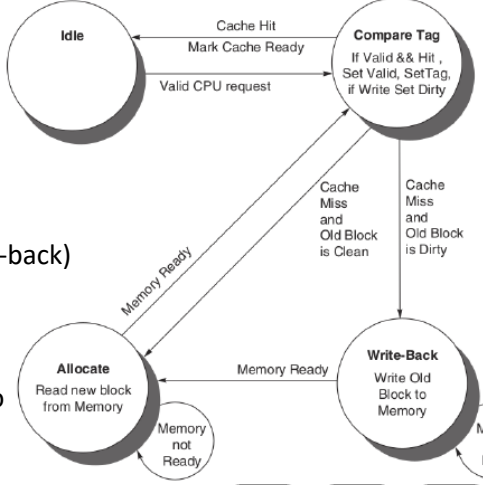
PSEUDO-LRU BASATO SU MRU (Most Recentrly Used)

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

Con $n$vie servono $n$bit, per ogni set

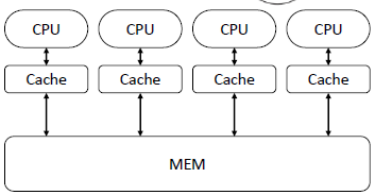
GESTIONE DELLA CACHE

- Come andiamo ad implementare la gestione della cache?
Utilizzo una FSM con 4 stati (come con la Control Unit):
  - stato di attesa (idle)
  - stato di verifica di hit/miss (compare tag)
  - scrittura di un blocco della cache alla memoria (coerenza, write-back)
  - trasferimento di un blocco della memoria alla cache (allocate)



MIGLIORAMENTO DELLE MISS PENALTY (multi-processore)

I sistemi multiprocessore hanno più CPU (cores) che lavorano in parallelo
Sono più cores che condividono la memoria centrale

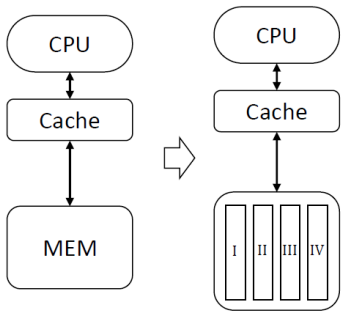


Ex. dato un dato presente nella $cache_1$
se questo dato è presente anche in un'altra $cache_k$, se il dato nella $cache_1$ viene sovrascritto, sorgono problemi di incongruenza tra $cache_1$ *e* $cache_k$ e la memoria centrale

- Implementare una serie di politiche che vanno a riallineare i dati tra memoria centrale, e le varie cache:
  - **Snooping:** le cache lavorano sempre in write-through (appena i dati vengono modificati, vengono inoltrati direttamente in memoria centrale).
    Ognuna cache monitora costantemente il bus indirizzi, e quando osservano un write su un dato che hanno in cache lo invalidano (write-invalidate)
    quando viene invalidata la copia del dato, un circuito dedicato porta il dato corretto all'interno
  - **Hardware transparency:** quando un dato viene scritto, un circuito dedicato aggiorna il corrispondente in ogni cache
  - **Non-cacheable memory:** viene definita una regione della memoria come non allocabile in cache

MIGLIORAMENTO DELLA MISS-PENALTY (interleaving)

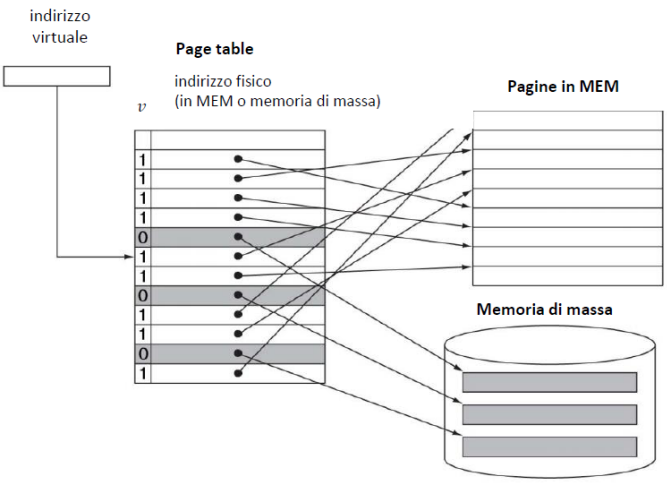
Aumentando la dimensione del blocco diminuisce la missrate, ma la miss penalty, anche se pagata meno frequentemente costa di più.
Questo avviene perché blocchi più grandi richiedono il trasferimento di un maggior numero di informazioni tra memoria e cache.



Andiamo a suddividere la memoria in banchi, che possono lavorare in parallelo
- possiamo farli lavorare in maniera analoga ad una pipeline, il throughput delle richieste id accesso aumenta
- possiamo aumentare l'ampiezza del bus e permettere il trasferimento di più parole per volta

MEMORIA VIRTUALE

Quando ragioniamo secondo la logica della gerarchia delle memorie, andiamo a focalizzarci tra i livelli di memoria principale e memoria di massa



- vogliamo andare a consentire una condivisione efficiente della memoria tra diversi programmi
- vogliamo andare a rimuovere il vincolo di uno spazio ridotto ad ogni programma

Come funziona?
- La memoria fisica viene suddivisa in pagine (i blocchi della cache)
- ogni programma lavora su uno spazio di indirizzamento virtuale, che non tiene conto della presenza degli altri programmi
- la memoria principale lavora come una cache, e al posto del blocco ci sono le pagine di memoria (page hit, page fault)
- In una page table (contenuta in memoria centrale), ogni programma ha le proprie corrispondenze tra indirizzi virtuali ed indirizzi fisici per ogni programma

1. la memoria riceve una richiesta di accesso tramite un indirizzo virtuale
2. viene controllata la page table
   - **Se la corrispondenza dell'indirizzo virtuale è presente e valida:** page hit accediamo all'indirizzo fisico corrispondente, e recuperiamo il dato
   - **se la corrispondenza non è valida/non presente:** page fault dobbiamo trasferire in memoria della memoria di massa, attraverso lo swap

in ogni caso abbiamo bisogno di 2 accessi a memoria

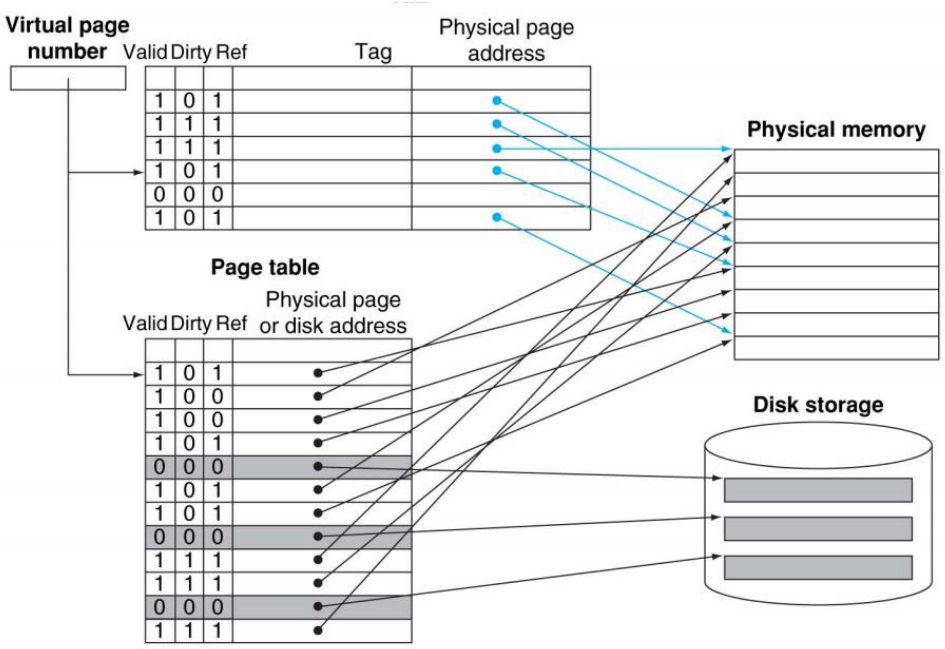
TRANSLATION LOOK-ASIDE BUFFER

È una cache per gli indirizzi delle pagine

- La memoria riceve una richiesta di accesso, tramite un indirizzo virtuale

Viene controllato TBL:

- **se c'è una hit:** si accede al corrispondente indirizzo fisico in MEM e il dato è recuperato
- **se c'è una miss:** bisogna trasferire il dato in MEM dalla memoria di massa, swap e aggiornare TBL



RILEVAMENTO E CORREZIONE DEI DATI

Fino ad ora abbiamo assunto che un dato presente in memoria, in qualsiasi livello della gerarchia esso sia, fosse integro e privo di errori.
Un errore può essere per esempio un bit di memoria errato, dovuto ad un malfunzionamento hardware; il motivo di questo funzionamento può essere casuale, imprevedibile ed i suoi effetti possono essere molto gravi.

Vogliamo andare a garantire l'integrità del dato
- costruire delle memorie più affidabili
- replicare i dati (ridondanza)
- utilizziamo codici di rilevamento/correzione

Aggiungiamo al dato delle informazioni che ci permettono di rilevare eventuali corruzioni
Se un codice è di correzione, oltre a rilevare un errore, ci permette anche di correggerlo

rilevamento -> solo rilevazione | correzione -> rilevamento + correzione

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
Ex1. quanto è la distanza di Hamming tra 11111010 e 11111110? (Risposta = 1)
Ex2. date 2 stringhe identiche = distanza Hamming è 0
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!

Probl. 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)?

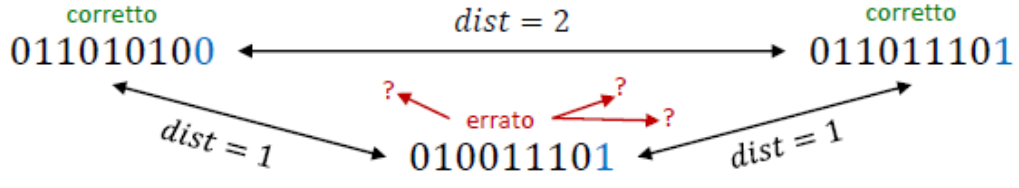
IL BIT DI PARITÀ

Primo approccio per garantire l'integrità del dato che andiamo vedere, basato sulla ridondanza

- Supponiamo di avere un dato d = 01101010, e di doverlo salvare in memoria.
  Il dato è scritto in un codice qualsiasi (complemento a 2, intero senza segno etc).
- Chiamiamo $\Sigma(d)$ la somma di tutte le cifre da interpretare in base 10; nel nostro caso $\Sigma(d) = 4$
- Aggiungiamo questo valore come bit extra, e lo chiamiamo $p(d)$, che ha valore: $p(d) = \begin{cases} 1, \text{se } \Sigma(d) \text{ è dispari} \\ 0, \text{se } \Sigma(d) \text{ è pari} \end{cases}$
- Questo valore viene salvato insieme al dato -> $(d, p(d))$

Ogni dato salvato sfruttando questo approccio ha un numero pari di 1, se il numero risulta dispari allora siamo sicuro che il dato sarebbe corrotto (XOR)

Probl1. Rileviamo solo un numero dispari di errori -> se si verificano 1, 3, 5… errori riusciamo a rilevarli, se sono pari no
Probl2. Non siamo in grado di correggere gli errori



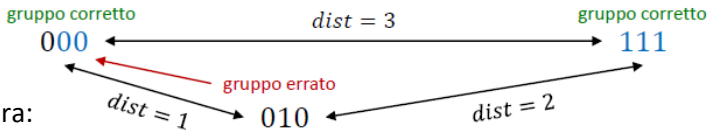
CODICE A RIPETIZIONE

Secondo approccio basato ancora sulla ridondanza => replico ogni singolo bit, n volte

Ex. n = 2, $il\ dato\ d$ = 01101010 diventa 000 111 111 000 111 000 111 000

- Ogni bit viene memorizzato come un gruppo di n + 1 bit

Infatti la distanza di Hamming minima tra due elementi è pari a n + 1



In generale, se ∂ è la distanza di Hamming minima di un codice C, allora:

- C può rilevare $\partial - 1$ errori
- C può correggere $\frac{\partial - 1}{2}$ errori

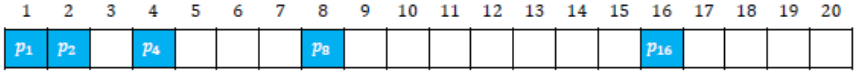
-> maggiore è ∂, migliore è il codice, ma aumenta anche il costo

CODICE DI HAMMING

È un codice a distanza 3, in grado di rilevare e correggere singoli errori, con un costo minore del codice a ripetizione.
Dato un codice di n bit, una porzione di dati sono assegnati a garantire l'integrità, il rimanente vengono utilizzati per rappresentare dati

- numeriamo i bit, da 1 a n, da sinistra a destra (a differenza della codifica binaria, destra a sinistra)
- i bit di parità sono quelli il cui numero è una potenza di 2

Su 20 bit totali, 5 bit sono di parità, e 15 per il dato



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

- Ogni bit di parità ha il compito di proteggere l'integrità di un sottogruppo dei valori.
- Regola di copertura: un bit con indice h è protetto dal bit di parità p_i, se solo se h in binario ha un 1 alla posizione $\log_2 i$

Ex.
bit di parità p1, 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 p2, 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 p4, 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 p8, 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 p16, 16 in binario è 10000, protegge tutti gli indici che, in binario, hanno 1 nel bit in posizione 4: 10000, 10001, 10010, …

Il bit pi 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

pattern di copertura

- Ogni bit di parità copre più bit, incluso se stesso
- 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!

Quando leggo il dato calcolo un codice di errore (sindrome) definito:
ECC = $c_{16}c_8c_4c_2c_1$ è un intero unsigned dove ogni bit cicorrisponde ad uno dei bit di parità p_i

	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}																■	■	■	■	■

c_i è a sua volta un bit di parità calcolato sul gruppo di bit coperti da p_i, 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 kè errato, lo correggiamo!

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 $p_8 \leftarrow 0$

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

- codifica: 011111000101

Bit aggiuntivo di parità

Aggiungendo un ulteriore bit al bit di parità sul pattern possiamo ottenere ∂ = 4, quindi possiamo identificare e correggere errori singoli e anche identificare (ma non correggere!) errori doppi

Nuovo bit di parità p_{nd+1} si applica a tutti gli altri bit (che ora diventano n_{tot-1})
Si possono verificare 4 casi:

1. ECC = 0 e p_{nd+1} corretto -> non ci sono errori
2. ECC > 0 e p_{nd+1} corretto -> doppio errore, sappiamo che c'è stato ma non si può correggere
3. ECC = 0 e pnd+1 errato -> singolo errore proprio in pnd+1, si può correggere
4. ECC > 0 e pnd+1 errato -> singolo errore che si può correggere

codice di Hamming errato, ma a livello globale risulta corretto.
In questo caso significa che l'errore è una combinazione dei due, o uno dei due, ma non siamo in grado di correggerlo.
Equidistanti tra i due possibili casi.

INPUT / OUTPUT

nell'elaborazione l'I/O rappresenta l'insieme dei metodi e dispositivi con cui trasferire le informazioni tra la CPU ed il mondo esterno
Per comunicare con la CPU, le periferiche sono collegate attraverso un canale chiamato bus.
Il bus è un canale condiviso, su cui viaggiano le informazioni di dispositivi diversi
Più precisamente, il bus è formato da 3 tipi di informazioni necessarie per gestire il trasferimento