

Emiddio Ingenito - Architettura degli elaboratori II

La CPU a singolo ciclo:

CPU a ciclo multiplo

Implementazione HW:

Unità di controllo:

Unità di controllo CPU (Control Unit)

Stati associati ai tipi di istruzione:

CPU a pipeline:

Implementazione datapath per l'esecuzione a pipeline:

Datapath della CPU pipeline:

Esempio di esecuzione su CPU a pipeline:

Unità di controllo:

Hazard:

Data hazard:

Soluzioni (SW e HW):

Hazard LW

Control Hazards:

Soluzione software mediante Delayed Branch slot:

Anticipazione dei salti:

Dynamic branch prediction

Pregi e difetti di questa soluzione:

Estensione predizione, per tenere in considerazione le **due** esecuzioni precedenti:

Eccezioni

Riconoscere la presenza di un'eccezione

Fasi gestione eccezione tramite registro causa:

Fasi gestione eccezione tramite Interrupt Vector Table (IVT):

Memorie

Register file:

Memoria principale (Comunemente ed erroneamente detta "RAM")

Memory-Bottleneck

Gerarchie di memoria:

Cache:

Mappatura diretta

Cache associative

Cache fully-associative

Cache set-associative

Grado di associatività

Accessi direct-mapped

Accessi set-associative

Accessi fully-associative

Associazione del "posto" all'interno di un set.

Implementazione di LRU (N=2, 2 posti per ogni set)

Pseudo-LRU (BST)

Gestione della cache:

Sistemi multi-processore (Multi-Core)

Interazioni con la memoria di massa

Integrità del dato.

Bit di parità

Codice a ripetizione

Hamming code:

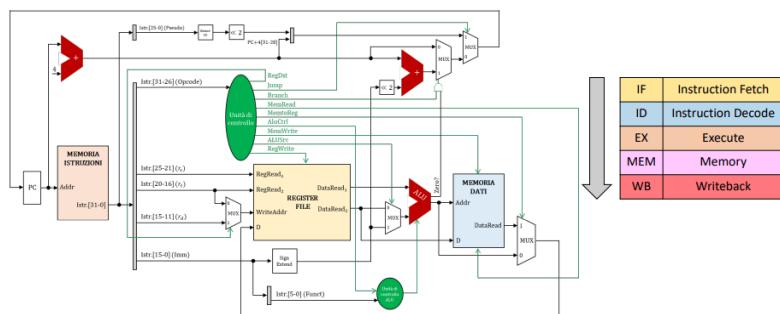
Regola di copertura:

Esercizi:

Bit aggiuntivo di parità:

La CPU a singolo ciclo:

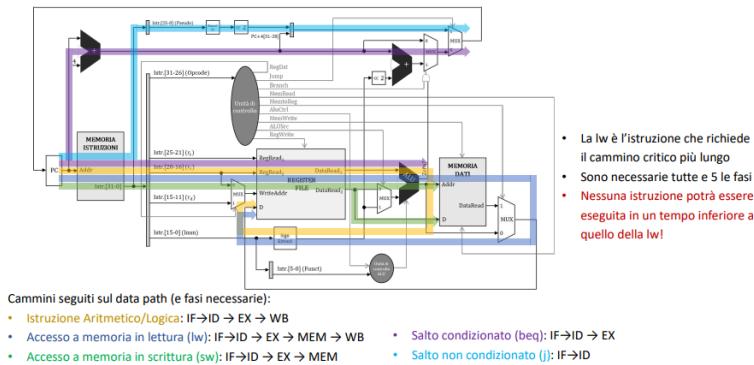
La nostra precedente CPU (singolo ciclo)



Prevede un'implementazione molto semplice, in cui ogni istruzione viene eseguita (in tutte le sue sottostadi, IF, ID, EX, MEM, WB), in un solo ciclo di clock, e la UC manovra il data-path in base al tipo di istruzione da eseguire.

Quindi ogni istruzione, viene eseguita in un tempo T pari alla somma dei tempi di esecuzione di ogni sottofase, dimensionato per far sì che anche l'istruzione più lenta venga eseguita, ma sappiamo che ci sono istruzioni che non richiedono tutte le sottofasi.

Prestazioni della CPU a singolo ciclo



Un'ulteriore inefficienza è data dalla replicazione dei componenti (notiamo la presenza di più ALU e più unità di memoria, istruzioni e dati)

Prestazioni della CPU a singolo ciclo

- Assumiamo che queste operazioni abbiano la seguente durata
 - Accesso a memoria (lettura o scrittura): 2 ns
 - Operazioni A/L: 2 ns
 - Accesso al Register File: 1 ns
 - Le altre operazioni assumiamo che siano trascurabili
- Durate minime:**
- Istruzione Aritmetico/Logica:** $2 + 1 + 2 + 1 = 6$ (IF + ID + EX + WB)
 - Accesso a memoria in lettura (lw):** $2 + 1 + 2 + 2 + 1 = 8$ (IF + ID + EX + MEM + WB)
 - Accesso a memoria in scrittura (sw):** $2 + 1 + 2 + 2 = 7$ (IF + ID + EX + MEM)
 - Salto condizionato (beq):** $2 + 1 + 2 = 5$ (IF + ID + EX)
 - Salto non condizionato (j):** $2 + 1 = 3$ (IF + ID)

Peta	10^{15}
Tera	10^{12}
Giga	10^9
Mega	10^6
Chilo	10^3
Milli	10^{-3}
Micro	10^{-6}
Nano	10^{-9}
Pico	10^{-12}
Femto	10^{-15}
Kibi	2^{10}
Mebi	2^{20}
Gibi	2^{30}
Tebi	2^{40}
Pebi	2^{50}
Exbi	2^{60}
Zebi	2^{70}
Yobi	2^{80}

Prestazioni della CPU a singolo ciclo

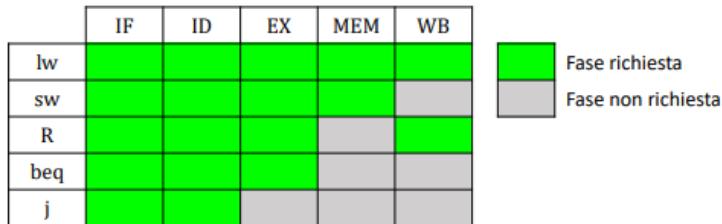
	A/L	lw	sw	beq	j	Durata media istruzione in CPU singolo ciclo	Durata media istruzione in CPU ciclo variabile
Durata minima	6	8	7	5	3		
Programma 1	30%	5%	20%	30%	15%	8	5.55
Programma 2	50%	40%	5%	2%	3%	8	6.74

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

CPU a ciclo multiplo

Introduciamo un nuovo modello di CPU, in cui l'esecuzione è suddivisa su più cicli di clock (uno per ogni singola sotto-fase):

Così, permetto a diverse istruzioni, di poter "impegnare" la CPU per un numero variabile di cicli di clock.



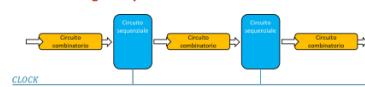
Per fare ciò, ho bisogno di tener traccia dell'esecuzione, per capire quali sottofasi vanno eseguite e quali ancora sono da eseguire.

Inoltre, tra due sottofasi dell'esecuzione, potrebbe essere necessario trasferire dei dati da una sottofase (ciclo di clock) all'altra.

In poche parole, serve della memoria (sappiamo già che sono implementate tramite circuiti sequenziali)

- Fasi diverse avvengono, per definizione, in **cicli di clock diversi**: il passaggio di informazioni tra sotto-fasi necessita di una **memoria**

- Serve una logica sequenziale!



Conseguenze progettuali:

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

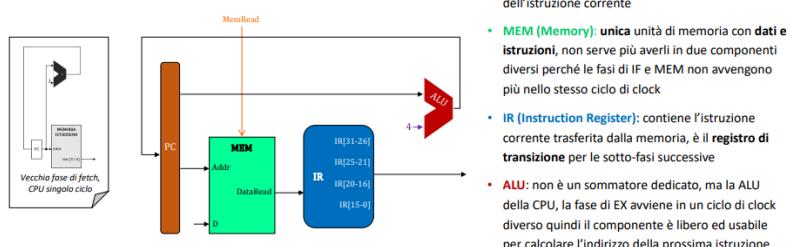
- Ricorda! CPU = Datapath (percorsi possibili) + Logica di controllo (manovrare gli scambi su quei percorsi)

Implementazione HW:

(Prima avevo vincolo che ogni componente era utilizzabile “una volta sola” in quanto il ciclo di clock era unico, ora non più).

CPU a ciclo multiplo: IF

- Ridefiniamo il sotto-circuito per la fase di fetch

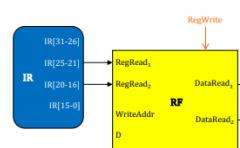


- **PC (Program Counter):** contiene l’indirizzo dell’istruzione corrente
- **MEM (Memory):** unica unità di memoria con dati e istruzioni, non serve più averla in due componenti diversi perché le fasi di IF e MEM non avvengono più nello stesso ciclo di clock
- **IR (Instruction Register):** contiene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

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

CPU a ciclo multiplo: ID

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



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

CPU a ciclo multiplo: EX/ALU

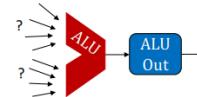
- La fase di EX avviene nel ciclo di clock successivo a ID
- La CPU assume di avere nei registri A e B (o in A e in parte di IR) i due operandi
- In questa fase la ALU fa **sempre** (anche quando non serve!) una operazione aritmetico/logica con due operandi, il risultato sarà memorizzato nel **registro di transizione ALUOut**, potrà essere usato in una fase successiva



CPU a ciclo multiplo: EX/ALU

Chi sono gli operandi della ALU?

- Come nella CPU a singolo ciclo cambieranno (tramite segnali di controllo e MUX posti sugli ingressi) a seconda di quale istruzione sta svolgendo la CPU (diversi datapath)
- Aspetto chiave: nella CPU multi-ciclo la ALU verrà sempre usata nella fase di EX, ma può essere usata anche in fasi diverse poiché è disponibile
- Sappiamo già che nella fase di IF la usiamo per fare $PC + 4$, quindi in questo caso i suoi operandi saranno il **contenuto di PC** e la costante **4**
- Durante la fase di EX di una istruzione A/L i due operandi saranno il **contenuto del registro A** e del **registro B** (il contenuto di, rispettivamente, rs e rt estratti nella fase di ID)
- Durante la fase di EX di una istruzione di accesso a memoria lw/sw i due operandi saranno il **contenuto del registro A** e l'**offset esteso di segno**, la ALU qui è usata per il calcolo dell'indirizzo (rs) + $SignExt(OFFSET) = A + SignExt(IR[0 - 16])$
- Durante la jump la ALU non è usata



CPU a ciclo multiplo: anticipazione BTA

- La beq pone delle difficoltà aggiuntive! Richiede di usare la ALU **due volte**
 1. per calcolare l'indirizzo di salto $PC + SignExt(IR[0 - 16]) \times 4$
 2. per verificare la condizione $(rs) - (rt) == 0$

Come fare?

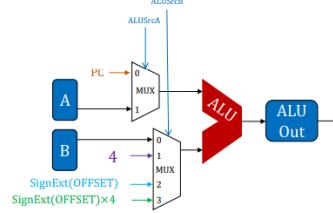
Idea:

- Nella fase di ID nessuno usa la ALU. Sfrutto la disponibilità: le faccio fare $PC + SignExt(IR[0 - 16]) \times 4$ (tutti dati già disponibili). Il risultato è il Branch Target Address (BTA) che viene salvato in ALUOut. Chiamiamo questa operazione **anticipazione del BTA**
- Quando passo alla successiva fase di EX
 - se non sto eseguendo la beq sovrascrivo ALUOut con il risultato richiesto dall'istruzione che devo eseguire (un valore se A/L o un indirizzo se lw/sw)
 - Se invece sto eseguendo una beq, faccio fare alla ALU $(rs) - (rt)$ e se il bit di zero risulta 1 allora il valore in ALUOut (precedentemente settato al Branch Target Address) verrà scritto nel PC

CPU a ciclo multiplo: EX/ALU

- Uso della ALU nella CPU a ciclo multiplo

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



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

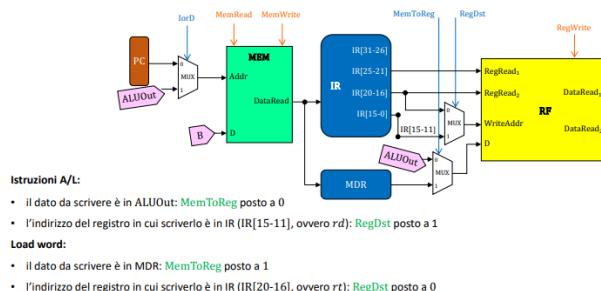
CPU a ciclo multiplo: MEM

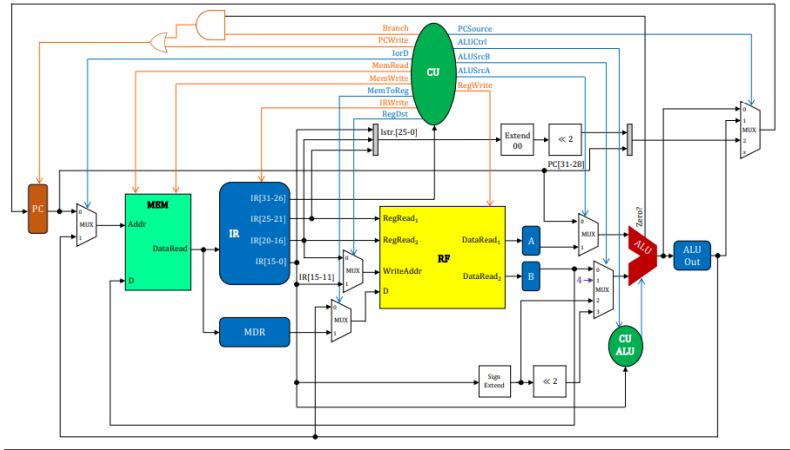
- Quando la CPU esegue una fase MEM assume che in ALUOut vi sia un indirizzo di memoria, calcolato nella precedente fase EX come $(rs) + \text{SignExt}(IR[0 - 16])$
- ALUOut, deve quindi poter essere presentato alla porta Addr del modulo di memoria



CPU a ciclo multiplo: WB

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





CPU a ciclo multiplo: segnali di controllo

- Segnali di selezione

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

Ricorda:

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

CPU a ciclo multiplo: segnali di controllo

- Segnali di comando

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

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

Unità di controllo:

La UC, è una macchina a stati finiti di Moore, che dati in input la sottofase di esecuzione da eseguire, e il tipo di istruzione da eseguire, da in output i segnali di

controllo associati:

Unità di controllo CPU (CU)



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

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



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

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



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

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

Unità di controllo CPU (CU)



Execute di beq

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



Execute di una qualsiasi istruzione A/L

- Grazie alla precedente fase ID-1, in A e B ci sono i contenuti di rs e rt
- I segnali di controllo sono settati in modo che:
 - La ALU svolga $(rs) op (rt)$ (dove op è l'operazione indicata nel campo funct)
 - Il risultato della ALU venga scritto in ALUOut



Execute di lw o sw

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

Unità di controllo CPU (CU)



Memory di sw

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



Memory di lw

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

Unità di controllo CPU (CU)

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

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

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

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

Unità di controllo CPU (Control Unit)

Permette di cambiare il data path, impostando appositi valori sui segnali di controllo, e fa svolgere alla CPU una delle sottostadi dell'esecuzione (IF, ID, EX, WB, MEM).

Per fare ciò, la CU ha bisogno di tenere traccia della sottostadio che la CPU sta eseguendo, e del tipo di istruzione che sta eseguendo.

La CU è una FSM di Moore (FSM in cui output dipende solo dallo stato corrente e non dall'input), in cui lo stato è definito da Sottostadio + Tipo istruzione corrente, e che da in uscita, lo stato prossimo, cioè la prossima fase da eseguire, e il valore dei segnali di controllo associati a quella sottostadio.

Stati associati ai tipi di istruzione:

- Instruction Fetch - Stato iniziale della CU, viene eseguita indipendentemente dal tipo di istruzione (giustamente, l'istruzione non è ancora nemmeno stata decodificata).

In essa, l'istruzione indirizzata da PC viene letta dalla memoria e scritta nell'instruction register.

La ALU svolge PC+4, e lo scrive nel PC.

- Instruction Decode: Anche questa viene eseguita indipendentemente dal tipo di istruzione.

La CPU assume che l'istruzione sia presente nell'IR (data la precedente esecuzione della fase IF).

Analizza l'istruzione, la divide in campi, e indica al Register File di leggere il contenuto di registri, e li carica in A e B (che sono due memorie).

Viene svolta l'anticipazione del BTA (Branch Target Address), cioè l'ALU (che al momento è inutilizzata), viene occupata per svolgere $PC + \text{SignExt}(\text{OFFSET}) \times 4$.

Il Risultato, viene scritto in ALUout.

Caso particolare, e la fase di Decode dell'istruzione della Jump, in cui l'istruzione si conclude, senza passare alla fase di Execute. (segnali di controllo posti in modo che nel PC venga scritto l'indirizzo di salto, e non il BTA).

- Execute

- Nel caso della BEQ, assumo che in ALUout ci sia il BTA.

La ALU esegue rs-rt (questa operazione NON sovrascrive ALUout), e se il bit di zero è a 1, il BTA (ALUout), viene scritto nel PC.

- Istruzioni Aritmetico Logiche (generiche):

Si assume che in n A e B ho i contenuti di rs e rt, la ALU svolge l'operazione indicata da funct, applicata a rs e rt.

Il risultato viene scritto in ALUout.

- LW e SW (Load e Store Word, intese come "dalla RAM"):

Si assume che in A ci sia RS (Base Address), i segnali sono settati in modo che la ALU svolga $rs + \text{SignExt}(\text{OFFSET})$, e scritto in ALUout.

- Memory

- Nel caso di Store Word: in B c'è il contenuto di rt (ciò che voglio scrivere in memoria, collegato direttamente all'ingresso dati del modulo di memoria).

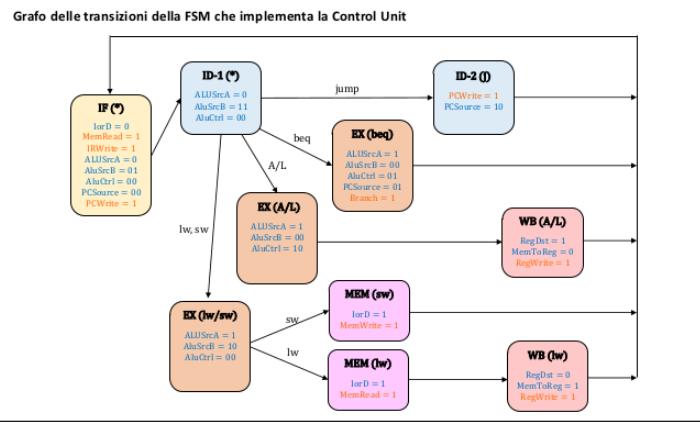
L'indirizzo in cui voglio scrivere, è scritto in ALUout.

- Nel caso di Load Word: ALUout contiene l'indirizzo della parola da cui leggere, e il contenuto della parola di indirizzo ALUout è letta dalla RAM e scritta in MDR.
- Write Back

Per ogni istruzione, ALUout contiene il risultato dell'operazione aritmetico logica.

I segnali di controllo sono settati in modo che ALUout sia scritto nel registro rd.

Se l'istruzione è una load word, il dato letto dalla memoria contenuto in MDR, e i segnali di controllo sono settati i modo da scrivere su rt.



CPU a pipeline:

Pipeline: Catena di montaggio è un metodo di divisione dei compiti in modo efficiente, in sottofasi.

La pipeline

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



Escludendo il primo invitato, nel secondo scenario, dalla coda esce un invitato per ogni unità di tempo.

Il secondo scenario, "costa di più", devo pagare 5 camerieri e non uno, ma è molto più efficiente!

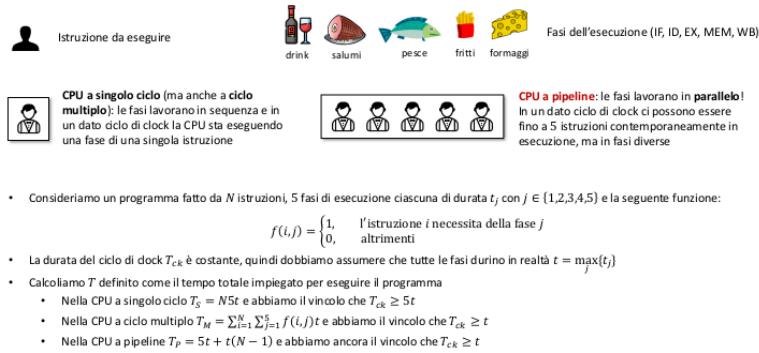
Se considero un invitato come un istruzione, essa sarà eseguita quando è passata per tutte e 5 le isole (drink, salumi...), che per noi sono le 5 fasi dell'esecuzione (IF, ID, EX, MEM, WB).

Devo introdurre quindi più camerieri, dividere l'HW in porzioni specializzate a svolgere una determinata sottofase di esecuzione.

What definition are you using for pipelining? Under the one I use pipelining is essentially a single-core phenomenon. If you have more cores it's parallelism, which is a different thing. Pipelined instructions are something that was invented in the late 1970s: let's say a multiplication takes 4 cycles, then a processor can have 4 separate pieces of hardware for the stages, and so you can feed a sequence of mults through your

processor: one mult takes 4 cycles, two takes 4+1 cycles, three take 4+2, et cetera. Asymptotically, you get one mult per cycle. Note that this is a single processor/core story.

La pipeline



Dati N invitati, e dato l'insieme dei tempi per eseguire le sottofasi (t_{Max} , t_2, t_3, t_4 , t_{Min} (5 fasi, ci sarà sempre una fase più lenta e più veloce)).

Nella CPU a singolo ciclo, il clock dimensionato per eseguire UNA istruzione (che è eseguita tutta in un ciclo di clock), è $TsCLK = t_{Max} + t_2 + t_3 + t_4 + t_{Min}$.

Nella CPU a ciclo multiplo, il clock per eseguire UNA sottoistruzione, è $TmCLK = t_{Max}$ (devo dare tempo alla fase più onerosa, ogni sottofase è eseguita in un ciclo di clock diverso, POSSO SALTARE SOTTOFASI!!)

TsCLK > TmCLK. (Clock singolo ciclo vs clock ciclo multiplo)

Se ogni istruzione ci mette $TsCLK$ per eseguire, nella CPU a singolo ciclo, il tempo per eseguire N istruzioni sarà $NTsCLK$, cioè $N*(t_{Max} + t_2 + t_3 + t_4 + t_{Min})$.

Se assumo che ogni sottoistruzione sia eseguita in uno stesso tempo T , il tempo per eseguire N istruzioni diventa $N5T$ (5 sottoistruzioni, *N)

Nella CPU a ciclo multiplo invece, io ho un clock di TmCLK pari a tMax.

Ogni istruzione però ha "tempo di esecuzione" variabile, perchè potrei saltare delle fasi di esecuzione.

Quindi il numero di cicli di clock necessari a eseguire N istruzioni è:

$$T_M = \sum_{i=1}^N \sum_{j=1}^5 f(i,j) t \quad f(i,j) = \begin{cases} 1, & \text{l'istruzione } i \text{ necessita della fase } j \\ 0, & \text{altrimenti} \end{cases}$$

La somma di N tempi di clock (Prima sommatoria), in cui ogni tempo di clock, è calcolato come sommatoria da 1 a 5 (le 5 fasi di esecuzione), in cui se l'istruzione richiede quella sottofase, viene sommato tMax, altrimenti no.

Nella CPU a pipeline invece, dovendo servire 5 invitati, assumo che ogni invitato passi per tutte e 5 le isole, e che in ogni isola il cameriere serva in un tempo tMax:

Tempo per eseguire N invitati = $5t_{\text{Max}} + (N-1)t_{\text{Max}}$ (5 è il numero di isole, numero delle nostre sottofasi).

Assumo che ogni cameriere serva allo stesso tempo, che sia pari a quello del più lento, e che ogni invitato passi obbligatoriamente per tutte e 5 le isole, anche se non vuole quel cibo (non posso saltare sottofasi, passo comunque per tutte e 5 IF, ID, EX, MEM, WB).

Metodo standard per la misurazione delle performance: Calcolo del Throughput = Lavoro da eseguire / Tempo di esecuzione = Numero di istruzioni da eseguire / Tempo necessario per eseguire una singola istruzione (nelle 5 sottofasi).

Dovendo scegliere un numero di istruzioni da eseguire, io voglio sapere come performa la mia CPU nel peggiore dei casi, quindi definisco il Throughput come:

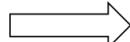
Throughput (tasso di produzione), definizione: $\rho = \lim_{Lavoro \rightarrow \infty} \frac{Lavoro}{Tempo speso}$

La pipeline

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

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

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



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

Il risultato si generalizza rispetto al numero di stadi:

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

- E la CPU a ciclo multiplo?

- Worst case**: tutte le istruzioni richiedono 5 fasi: $\rho_M = \frac{1}{5t}$ (come CPU a singolo ciclo)
- Best case**: tutte le istruzioni richiedono 2 fasi: $\rho_M = \frac{1}{2t}$

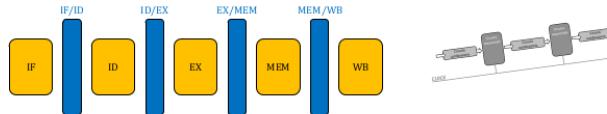
- In ogni caso più lenta della CPU a pipeline, nonostante con la pipeline richiediamo ad ogni istruzione di attraversare tutte le 5 fasi, anche quando non serve!

Implementazione datapath per l'esecuzione a pipeline:

Il Concetto chiave della pipeline, è l'esecuzione in sequenza e in parallelo (perchè in uno stesso momento, per ogni sottofase di esecuzione, quindi per ogni sottoporzione specializzata di HW, succede qualcosa).

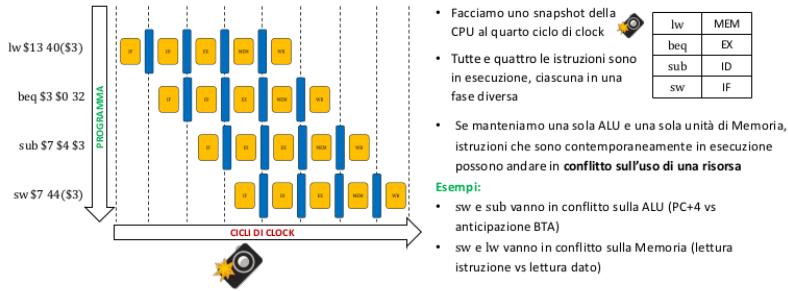
Perchè si possa lavorare in parallelo, cioè che possano essere eseguire più sottofasi di esecuzioni allo stesso tempo, devo suddividere ogni sottofase in un sotto-circuito (regione del datapath) dedicata.

- Perchè possano lavorare in parallelo serve che ogni fase venga eseguita in una **regione** (sotto-circuito) dedicata del datapath
- Perchè possano essere applicate in sequenza serve un modo per rendere accessibile il risultato parziale di una fase alla successiva: **registri di transizione**



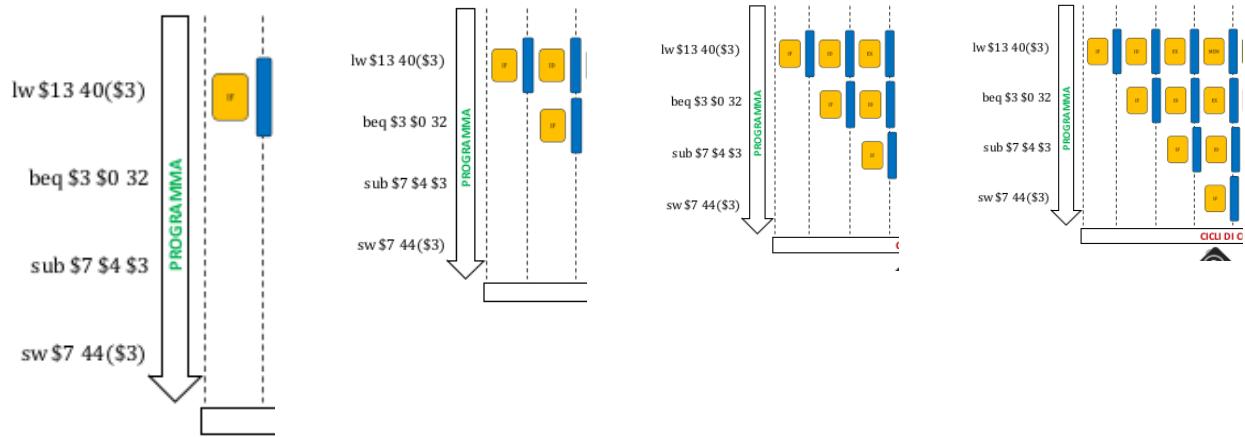
- I registri di transizione servono per trasferire nella fase $i + 1$ il risultato dell'elaborazione svolta nella fase i

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



Le prossime istruzioni, verranno scritte "sotto la store word", il grafico va interpretato come se ad ogni ciclo di clock venisse "aggiunta" una sbarretta verticale.

Esempio:



Primo, secondo, terzo, quarto ciclo di clock.

Al terzo ciclo di clock per esempio, la LW sta passando per la fase di EX, la BEQ per la fase di ID, la SUB per la fase di IF.

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

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

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

Le criticità evidenziate, mi dicono anche quante sottofasi di esecuzione entrano in conflitto contemporaneamente,

Capisco quindi, che dovrò ridondare la ALU in 3 unità (perchè dalla tabella vedo che le fasi IF, ID e EX entrano in conflitto per utilizzare la ALU), e la memoria in 2 unità (fase di IF e MEM in conflitto)

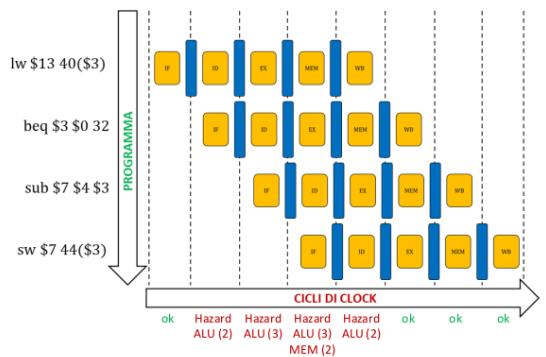


Devi guardare i COLORI diversi, non le funzioni da eseguire, perchè per esempio, nella fase MEM, o eseguo la lw o la sw, nella fase di EX, o eseguo la BEQ o la Jump, quindi non entrano in conflitto. (In ogni sottofase eseguo una singola funzione).

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

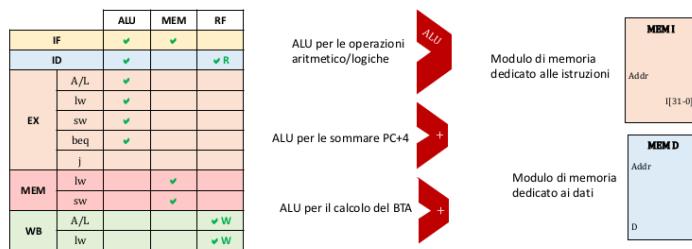
- Quanti RF servono?
 - Uno solo, il RF non presenta criticità
- Quante ALU servono?
 - Su quanti sfondi diversiabbiamo la ✓ nella colonna ALU? Tre! Servono 3 ALU
- Quante unità di memoria servono?
 - Su quanti sfondi diversiabbiamo la ✓ nella colonna MEM? Due! Servono 2 MEM

- Tutte le criticità strutturali dall'esempio precedente



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

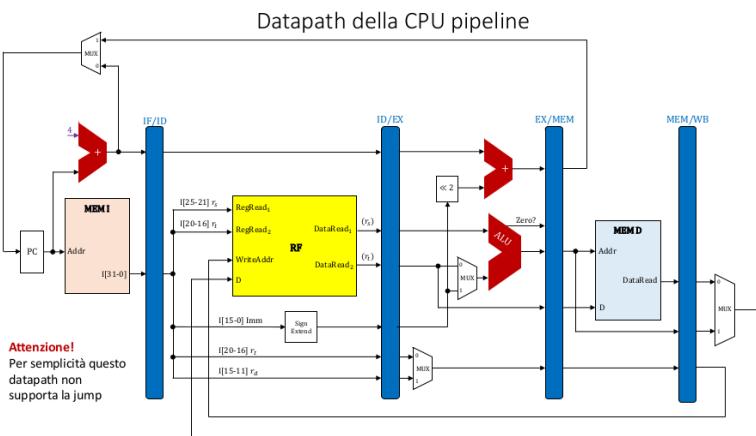
- Dupliciamo le risorse ritornando alla configurazione che avevamo nella CPU a singolo ciclo



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

La concorrenza sulla Memoria, mi induce a dividere di nuovo la memoria in Istruzioni e dati (visto che la IF richiede di scrivere sulla parte "dedicata" alle istruzioni, e la MEM di scrivere sulla memoria dati).

Datapath della CPU pipeline:



(rt) → Contenuto del registro rt.

I [20-16] rt → Indirizzo (Identificativo) del registro rt

Dalla ALU nella fase di execute, esce un segnale che potrebbe rappresentare o l'indirizzo di memoria su cui scrivere (ADDR), oppure il risultato di un'operazione aritmetico logica, da scrivere sul banco dei registri.

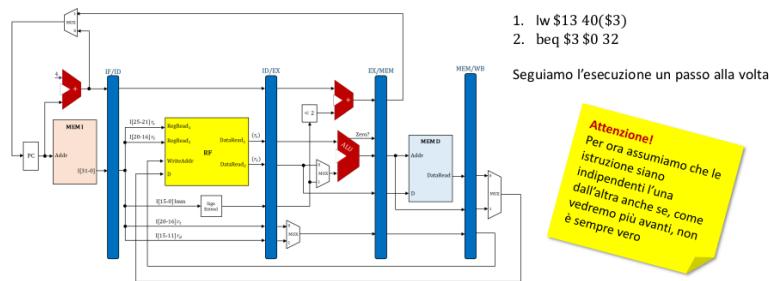
Come si vede, lo stesso dato si presenta sia alla porta ADDR del modulo di memoria dati, sia al multiplexer in cui si decide se sul banco dei registri va scritto il

risultato della ALU oppure il contenuto della RAM.

Lo stesso dato però, non può rappresentare contemporaneamente sia un indirizzo che il risultato di un A/L, pertanto sarà la UC a indicare alla memoria dati, tramite i segnali di controllo di letture e scrittura, se accettare o meno l'indirizzo che si presenta all'ingresso ADDR.

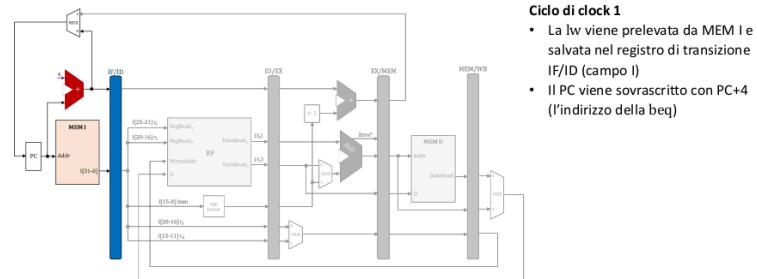
Esempio di esecuzione su CPU a pipeline:

Esempio di esecuzione



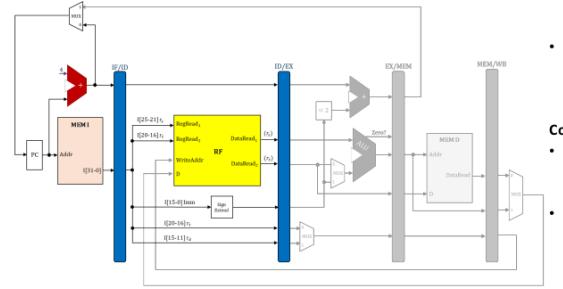
Esempio di esecuzione (ciclo di clock 1)

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



Esempio di esecuzione (ciclo di clock 2)

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

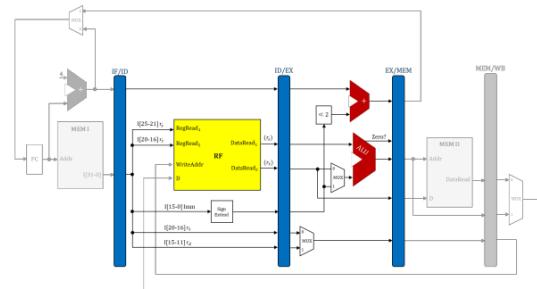


Ciclo di clock 2

- Il contenuto dei registri 3 e 13, (numeri letti dal campo I di IF/ID) viene prelevato dal RF e caricato in ID/EX (campi A e B)
- Gli altri campi di I che serviranno ad una fase successiva vengono copiati in ID/EX (**forwarding di informazioni**)
- Contemporaneamente
 - La beq viene prelevata da **MEM I** e salvata nel registro di transizione **IF/ID** (campo I)
 - Il PC viene sovrascritto con **PC+4** (l'indirizzo della prossima istruzione)

Esempio di esecuzione (ciclo di clock 3)

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

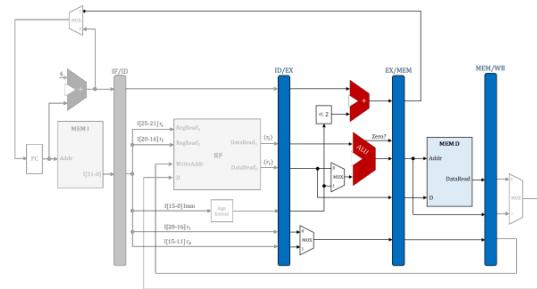


Ciclo di clock 3

- La ALU calcola l'indirizzo a cui accederà la lw per recuperare il dato, viene salvato in EX/MEM
- Il sommatore, parallelo, calcola un BTA, viene salvato in EX/MEM, è un scarto residuo
- Contemporaneamente
 - Il contenuto dei registri 3 e 0, (numeri letti dal campo I di IF/ID) viene prelevato dal RF e caricato in ID/EX
 - Gli altri campi di I che serviranno ad una fase successiva , tra cui l'offset per il calcolo del BTA, vengono copiati in ID/EX

Esempio di esecuzione (ciclo di clock 4)

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

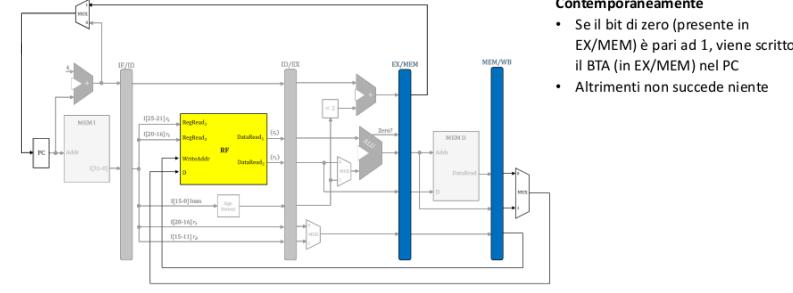


Ciclo di clock 4

- La **MEM D** recupera il dato per la **lw**, l'indirizzo di accesso è letto da **EX/MEM**, il dato è scritto in **MEM/WB**
- Contemporaneamente
 - La ALU fa la differenza tra i contenuti dei registri 3 e 0, il risultato e il bit di zero sono salvati in **EX/MEM**
 - Il sommatore, in parallelo, calcola il BTA che viene salvato in **EX/MEM**

Esempio di esecuzione (ciclo di clock 5)

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

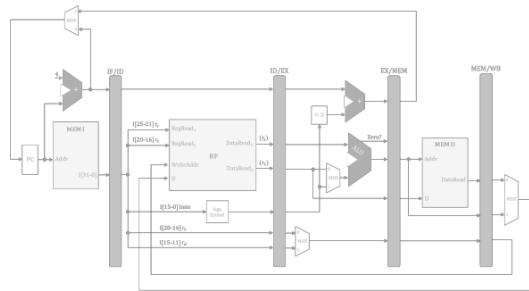


Ciclo di clock 5

- Il dato recuperato viene scritto nel RF nel registro 13
- Contemporaneamente
 - Se il bit di zero (presente in EX/MEM) è pari ad 1, viene scritto il BTA (in EX/MEM) nel PC
 - Altrimenti non succede niente

Esempio di esecuzione (ciclo di clock 6)

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

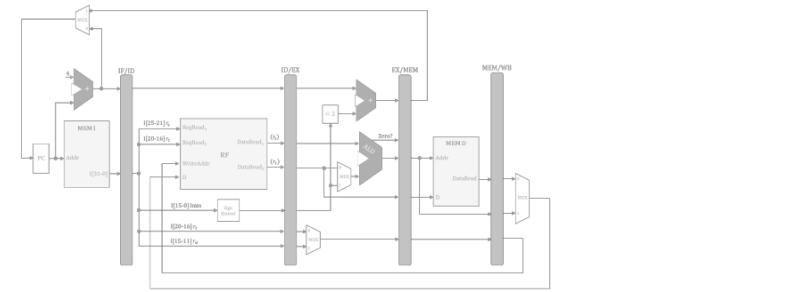


Ciclo di clock 6

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

Esempio di esecuzione (ciclo di clock 7)

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

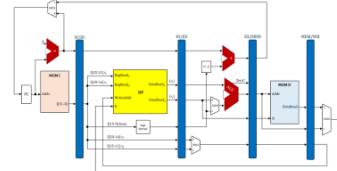


Unità di controllo:

CPU a pipeline: segnali di controllo

- Segnali di selezione

ALUSrc	Selezione del secondo operando ALU: rt (0) immediato esteso di segno (1)
RegDest	Selezione campo WriteAddr del RF: rt (0), rd (1)
MemToReg	Selezione del dato presentato in scrittura al RF: risultato ALU (0), contenuto di dato letto da MEM (1)
AluCtrl	Selezione della modalità di operazione della ALU (analogico a CPU singolo ciclo)
Branch	Indica se istruzione corrente è una beq
PCSrc	Selezione di quale indirizzo mandare al PC: $PC+4$ (0), BTA (1)



Ricorda:

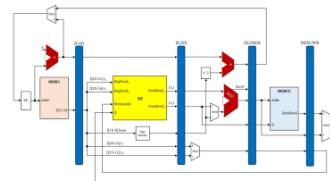
Opcode	AluCtrl
Tipo R, rimanda a funct	10
lw, somma	00
sw, somma	00
beq, sottrazione	01

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

CPU a pipeline: segnali di controllo

- Segnali di comando

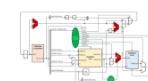
RegWrite	Scrittura del Register File
MemWrite	Scrittura della memoria
MemRead	Lettura della memoria



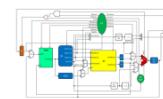
- La scrittura in tutti gli altri registri non menzionati è data dal clock, vengono scritti ad ogni ciclo di clock (fronte di discesa)
- **Problema:** come organizziamo i segnali di controllo nel datapath?

CPU a pipeline: logica di controllo

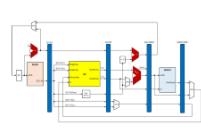
- La logica di controllo di una CPU a pipeline presenta alcuni aspetti peculiari, dovuti a questa specifica architettura
- Data una istruzione i l'unità di controllo calcola tutti i segnali di selezione e comando



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



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



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

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

Hazard:

Per Hazard (**conflitto**), si intendono:

-**Structural hazards:** Le risorse HW non supportano alcune combinazioni di

istruzioni

-**Data hazards:** Un'istruzione dipende dal risultato di una istruzione che è ancora nella pipeline

Gli hazard strutturali, cioè i problemi di conflitto sull'HW, li abbiamo già risolti, ridondando ALU e Memoria (tornando a suddividere in memoria dati e istruzioni).

Conflitti più interessanti, sono i data hazards.

Data hazard:

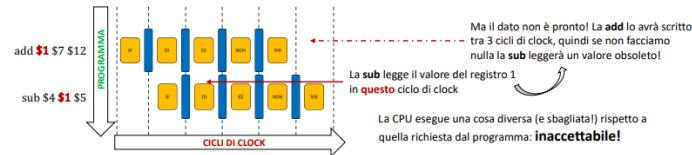
Un'istruzione dipende dal risultato di una istruzione che è ancora nella pipeline.

- Esempi

- Una add potrebbe dover usare un operando che è il risultato di una sub precedente, la add potrebbe trovarsi nella fase di EX quando la sub non ha ancora concluso la fase di WB!
- Una j potrebbe non dover essere eseguita se viene svolto un salto condizionato di una beq che la precede, ma la sua esecuzione potrebbe essere iniziata prima di sapere l'esito della condizione nella branch!
- La CPU a pipeline che abbiamo progettato non è ancora in grado di prevenire/gestire questi eventi che possono compromettere la correttezza dell'esecuzione di un programma

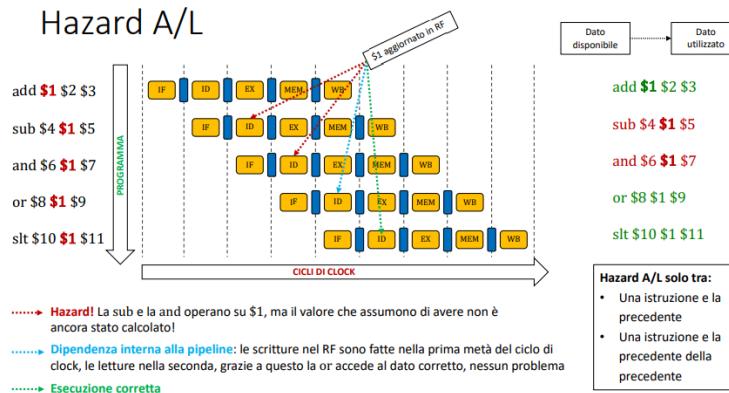
- Criticità di dato (data hazards)

- Un'istruzione *a* deve scrivere un valore nel registro *r*
- Un'istruzione *b*, successiva ad *a*, deve usare il contenuto di *r*
- Quando *b* recupera (*r'*) dal RF (fase ID) *a* è ancora in pipeline e non ha svolto la WB



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

Hazard A/L



Soluzioni (SW e HW):

Hazard A/L (approccio SW)

- Prima soluzione: intervenire sul software

add \$1 \$2 \$3	add \$1 \$2 \$3	• nop : istruzione speciale che svolge un'operazione fittizia che non provoca effetti sulla normale esecuzione (no operation)
sub \$4 \$1 \$5	nop	• Esempio add \$0 \$0 \$0 (in MIPS il registro \$0 contiene sempre la costante \$0 per convenzione)
and \$6 \$1 \$7	nop	• Effetto di una nop: la CPU aspetta per 1 ciclo di clock senza fare niente, come se inserissimo una bolla nella pipeline
or \$8 \$1 \$9	sub \$4 \$1 \$5	• Chi dovrebbe svolgere questo lavoro sul codice? Idealmente il compilatore
slt \$10 \$1 \$11	and \$6 \$1 \$7	• Pro: con questo approccio risolviamo i data hazard
	or \$8 \$1 \$9	• Contro: molto inefficiente, si perdono un sacco di cicli di clock
	slt \$10 \$1 \$11	

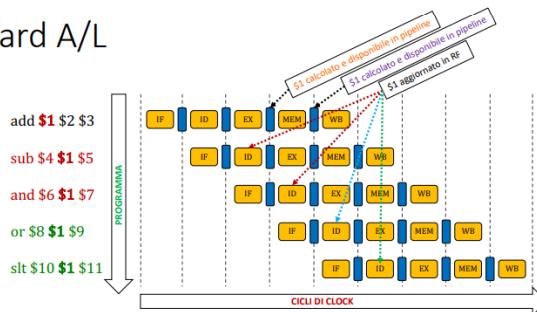
Inserisco due bolle se l'istruzione dopo deve usare il risultato dell'istruzione precedente (cioè se lo stesso registro compare come destinazione e utilizzato in due istruzioni adiacenti)

Hazard A/L (approccio SW)

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

add \$1 \$2 \$3	add \$1 \$2 \$3	add \$1 \$2 \$3	• Queste sono istruzioni che sarebbero venute dopo, ma la cui esecuzione può essere anticipata senza compromettere la correttezza del programma: sarebbero comunque state eseguite, non generano data hazard
sub \$4 \$1 \$5	nop	add \$31 \$30 \$29	• Fare questo genere di ottimizzazioni sul codice è molto complesso e comunque non vi è garanzia di riuscire a riempire tutte le «bolle»
and \$6 \$1 \$7	nop	add \$25 \$26 \$27	
or \$8 \$1 \$9	sub \$4 \$1 \$5	sub \$4 \$1 \$5	
slt \$10 \$1 \$11	and \$6 \$1 \$7	and \$6 \$1 \$7	
	or \$8 \$1 \$9	or \$8 \$1 \$9	
	slt \$10 \$1 \$11	slt \$10 \$1 \$11	
		:	

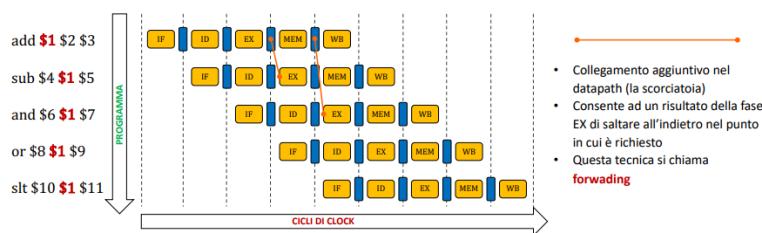
Hazard A/L



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

Hazard A/L (approccio HW)

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



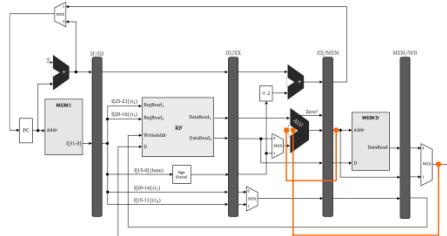
Devo fare forwarding da due registri diversi, perchè altrimenti se facessi collegamenti partendo da EX/MEM per entrambi, questo verrebbe sovrascritto al momento di dover fare forwarding alla AND.



Ricorda! I Registri in blu che vedi, sono gli stessi per tutte le istruzioni, tra il ciclo di clock e l'altro, viene effettuata una sovrascrittura dei dati in esso, per questo alla AND va dato il valore della ADD preso dal registro MEM/WB.

Hazard A/L: forwarding

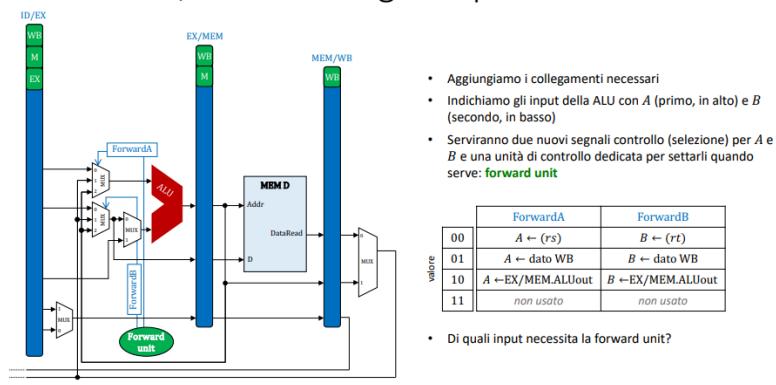
- Dobbiamo modificare il datapath



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

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

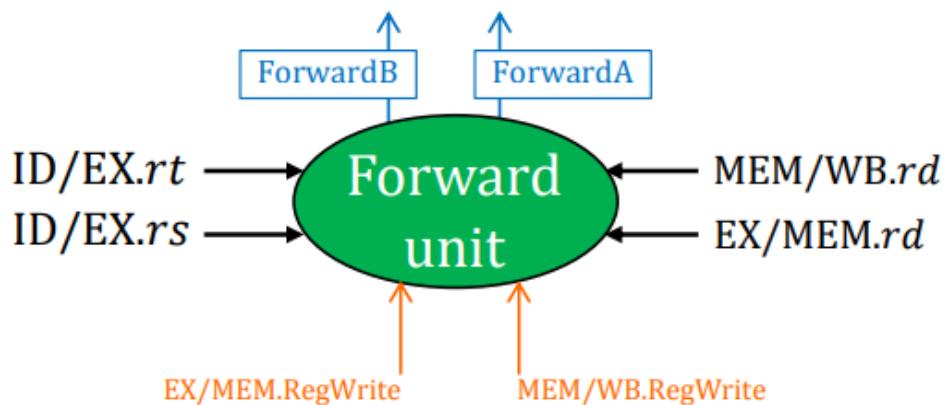
Hazard A/L: forwarding datapath



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

- Di quali input necessita la forward unit?

La Forward unit, comanderà il valore dei segnali di controllo, in base ai seguenti segnali in input:

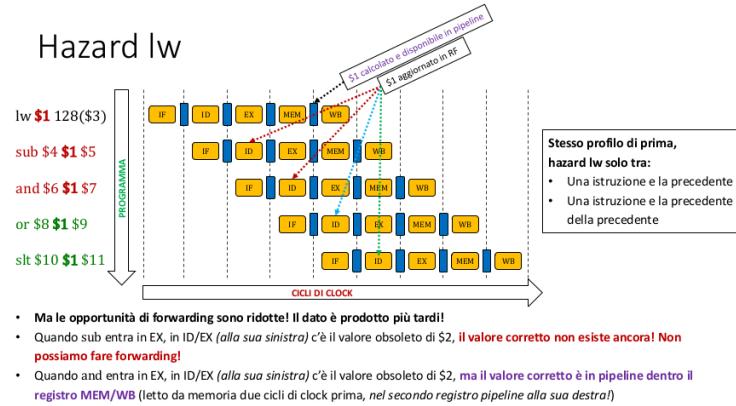


Hazard LW

Negli Hazard precedentemente analizzati, il dato era prodotto nella fase **EX**.

Abbiamo risolto ciò con la tecnica del forwarding HW, e l'implementazione di una **"Forwarding Unit"**.

Potrebbe però generarsi un'altro tipo di hazard, nel caso della **Load Word**, in cui il dato corretto, sarà reso disponibile solo al termine della fase **MEM** (Carico DALLA memoria)



L'Hazard sulla AND può quindi essere risolto tramite forwarding, perchè il dato, nel ciclo di clock in cui la AND va in EX, è disponibile in pipeline.

Quindi a risolvere quell'hazard, ci pensa già la "Forward Unit" precedentemente implementata.

Per l'Hazard sulla SUB, devo necessariamente implementare uno stallo sulla IF, ID e EX , di un ciclo di clock (devo attendere la MEM della LW, poi per forwarding e dipendenza interna sarà tutto ok).

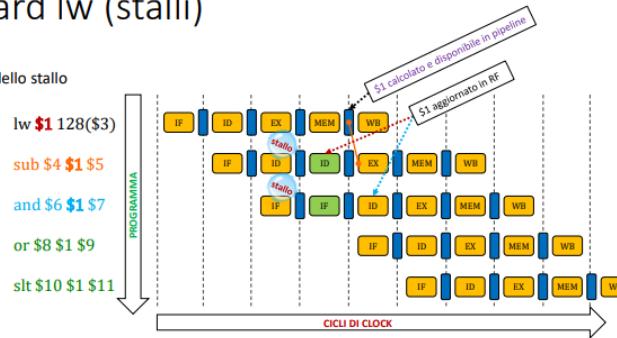
Hazard lw

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



Hazard lw (stalli)

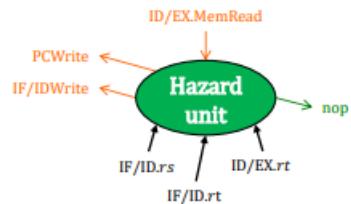
- Effetti dello stallo



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

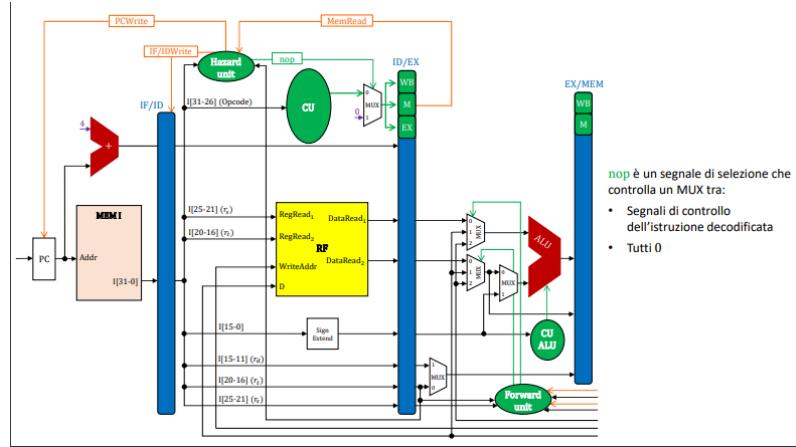
Hazard lw (stalli)

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



- La Hazard Unit lavora nella fase di ID, assumiamo di essere all'inizio di quel ciclo di clock
- Se la condizione dell' If è vera allora:
 - Nel ciclo prima era stata codificata una **lw**, che in questo ciclo fa la EX
 - L'istruzione successiva **j**, che è ora in ID, opera sul valore letto dalla **lw** (con **rs** o con **rt**)
- Stallo di IF:
 - $PCWrite \leftarrow 0$ impedisce, per questo ciclo di clock, la scrittura di $PC+4$ nel PC, nel prossimo ciclo di clock la CPU ripeterà IF di **i** che aveva svolto in questo ciclo
 - $IF/IDWrite \leftarrow 0$ impedisce, per questo ciclo di clock, l'aggiornamento di IF/ID con l'istruzione **i**, nel prossimo ciclo di clock la CPU ripeterà quindi ID di **j** che aveva svolto in questo ciclo
 - $nop \leftarrow 1$ nel registro ID/EX vengono scritti tutti i segnali di controllo pari a 0 anziché i segnali di controllo della **j**, nel prossimo ciclo di clock la CPU fa EX di una **nop** (imporre tutti i segnali di controllo a 0 è il modo più semplice che usiamo noi per implementare la **nop**)





Con questa soluzione, ho risolto tutti i data hazard.

Control Hazards:

Dopo aver gestito hazard strutturali (ridondando risorse) e gli hazard di dati (tramite la tecnica del forwarding e dello stallo), analizziamo gli hazard di controllo.

Se un'istruzione "a", implica un salto (è una jump o una beq), il pc andrà sovrascritto con l'indirizzo di salto, e alcune delle istruzioni successive, non vanno elaborate.

Prima che però il salto venga effettuato, bisogna aspettare che "a", arrivi nella fase MEM! In cui avviene la sovrascrittura del PC con il BTA, ma nel frattempo, altre 3 istruzioni sono entrate in pipeline (MEM è la quarta sottofase di esecuzione, nel frattempo in pipeline ci sono altre 3 istruzioni rispettivamente in Execute, Decode e Fetch)

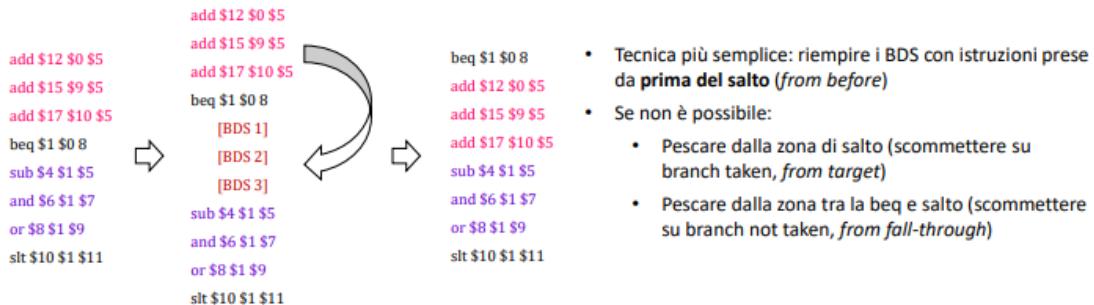
Criticità di controllo

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



Soluzione software mediante Delayed Branch slot:

- **Delayed branch slot:** ogni branch avviene di fatto in ritardo rispetto a quando entra nella CPU
- Vedo questo ritardo come slot di esecuzione su cui riorganizzare il codice (analogamente a quando abbiamo fatto per la gestione software dei data hazard)



Branch Delay Slot → Se l'istruzione è una branch o una jump, le istruzioni dopo la jump NON DOVREBBERO essere eseguite, ma tempo che la branch scrive su PC il BTA (Nella fase MEM), altre tre istruzioni sono entrate nella pipeline, e verranno elaborate.

Devo riempire quindi questi 3 slot di istruzioni, con delle istruzioni che verranno SICURAMENTE eseguite, per evitare di sbagliare (prendendole da prima del salto).

Se non posso prendere istruzioni da prima del salto, si aprono due scenari:

- Prendo queste tre istruzioni da dopo la zona di salto (dopo la slt, from target), e se il salto viene eseguito, sono già avanti (le istruzioni dopo la zona di salto sono già in pipeline), altrimenti (se il salto non viene effettuato), devo fare il flush delle tre istruzioni "erroneamente" caricate (flush dei 3 registri di transizione delle fasi IF, ID, EX).

- Prendo queste tre istruzioni dalla zona tra la branch/jump e il punto di salto (scommettendo sul fatto che il salto NON venga eseguito).

Stessa situazione, se il salto non viene effettuato sto già un passo avanti, altrimenti, flush dei 3 registri di transizione tra le sottofasi.

Criticità di controllo (gestione HW)

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



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

In questa configurazione, la CPU “scommette” sempre sul fatto che il salto non venga effettuato, quindi prende le prossime istruzioni da eseguire, dalla zona tra la branch/jump, e la zona di salto.

Anticipazione dei salti:

Data l'architettura del datapath, prima che il salto venga effettuato, passano 4 cicli di clock, quindi in caso di “errore” ci sono in pipeline 3 istruzioni che non avrebbero dovuto entrare, devo effettuare il flush di 3 registri, “pagando” con la perdita di 3 cicli di clock, uno per ogni flush.

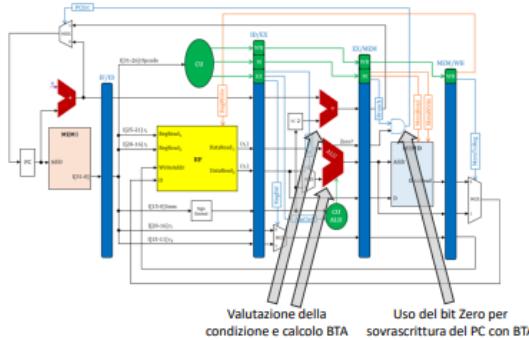
Per ridurre questo “costo”, devo cercare di prevedere il prima possibile se il salto viene effettuato o no.

Il prima possibile, vuol dire nella fase ID, in cui capisco se l'istruzione è una branch/jump o no.

Se mi accorgo già in ID del salto, se sbaglio, devo effettuare il flush solo del registro tra IF e ID, quindi perdo un solo ciclo di clock!

Anticipazione dei salti

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



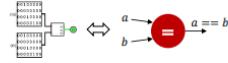
- Dagli hazard Iw abbiamo imparato una regola fondamentale: **per prevenire gli stalli bisogna rilevare la criticità il prima possibile e cioè nella fase ID**
- Come posiamo rilevare questo tipo di criticità nella fase di ID?
- Anticipazione del salto:** modifico il datapath in modo che sia il calcolo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

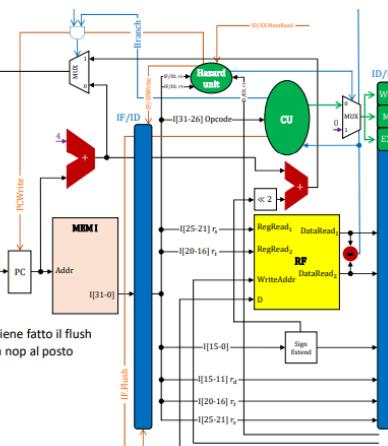
Effettuo una modifica, per far sì che il BTA venga scritto nel PC nella fase ID, e non più nella fase MEM,(Comparatore dedicato nella fase ID).

Un'altra modifica, deve permettere alla fase ID di avere già disponibile il BTA (sennò come lo scrivo..).

Prima il BTA era calcolato nella fase EX, ora deve essere per forza calcolato nella fase ID.

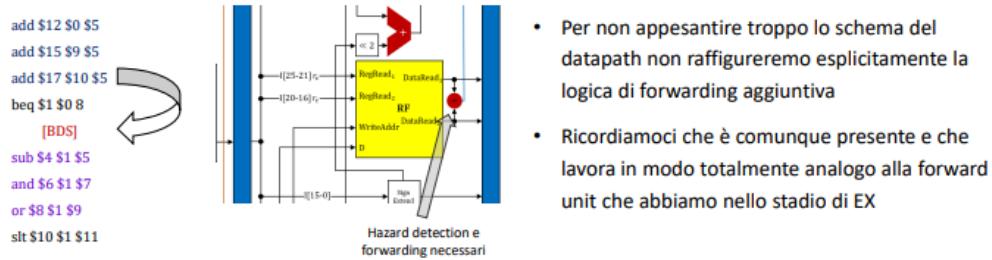
Anticipazione dei salti

- Il sommatore per il calcolo del BTA ritorna nello stadio ID (come nella CPU a ciclo multiplo)
- Va aggiunto un comparatore dedicato per la verifica della condizione di uguaglianza
 
- Il comparatore è collegato direttamente alle uscite del RF per confrontare (r_s) e (r_t)
- La control unit viene aggiornata: in caso di salto effettuato viene fatto il flush (comando **IFLush**) del registro IF/ID, viene cioè sovrascritta nop al posto dell'istruzione successiva alla beq che la CPU sta prelevando
- La CPU scommette sempre su **branch not taken**
 - Se indovina l'esecuzione prosegue a pieno regime
 - Se sbaglia paga con un flush (un ciclo di clock)



Anticipazione dei salti

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

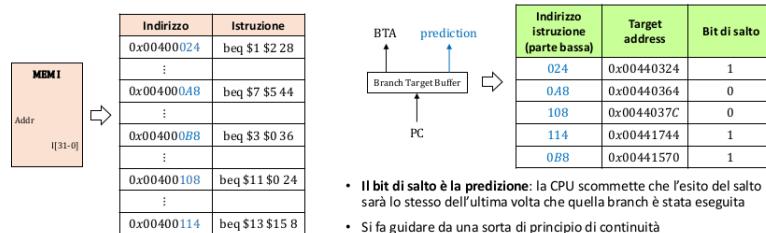


Dynamic branch prediction

La CPU cerca di prevedere lo stato della branch (salto), attraverso un'analisi dello storico delle istruzioni branch (taken or not taken).

Questo storico, si chiama branch target buffer, una memoria associativa (che associa dei metadati ai dati, come fosse un dizionario).

Ogni branch è indicizzata (metadati) attraverso la parte bassa **del proprio indirizzo**.



Pregi e difetti di questa soluzione:

- **Argomento a favore:** un programma spesso ha dei loop (while, for, etc.) nei loop una stessa branch viene eseguita ripetutamente

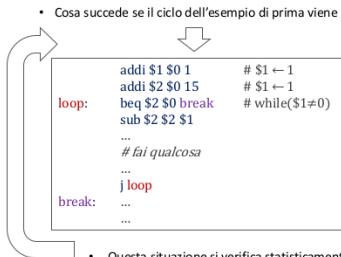
```
loop:    addi $1 $0 1      # $1 ← 1
          addi $2 $0 15     # $2 ← 15
          beq $2 $0 break   # while($1≠0)
          sub $2 $2 $1
          ...
          # fai qualcosa
          ...
          j loop
break:   ...
          ...
```

- Quante volte viene eseguita la branch?
 - 15
- Quante volte indoviniamo il salto con dynamic branch prediction?
 1. Scommetto not taken (initial guess non informato), **indovinato!**
 2. Scommetto not taken, **indovinato!**
 3. Scommetto not taken, **indovinato!**
 - ⋮
 15. Scommetto not taken, **sbagliato!**



Limite: La scommessa si basa sul solo stato dell'ultima esecuzione della branch, e non sull'intero storico! (che è già a disposizione).

Esempio limite basandosi solo sullo stato dell'ultima esecuzione:



- Cosa succede se il ciclo dell'esempio di prima viene ripetuto una seconda volta dopo poco tempo?

- Quante volte viene eseguita la branch?
 - 15
- Quante volte indoviniamo il salto con dynamic branch prediction?
 1. Scommetto taken, **sbagliato!**
 2. Scommetto not taken, **indovinato!**
 3. Scommetto not taken, **indovinato!**
 - ⋮
 15. Scommetto not taken, **sbagliato!**



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

Alla fine della prima esecuzione del primo ciclo, l'ultima istruzione la branch è stata **taken**, pertanto la prima istruzione del secondo ciclo (basandosi solo sull'ultima esecuzione della branch), sarà prevista come **taken, SBAGLIATO!**

Al secondo ciclo quindi, avrò due flush su 15 esecuzioni, e non più uno.

I cicli annidati, vengono eseguiti statisticamente così tanto spesso, che di fatto ogni ciclo costa 2.

Estensione predizione, per tenere in considerazione le due esecuzioni precedenti:

Per tenere in considerazione lo stato dell'esecuzione di una branch, necessito di un bit (devo memorizzare il solo bit di salto).

Estendendo la memorizzazione dello stato su due bit, posso memorizzare le due esecuzioni precedenti di una stessa branch.

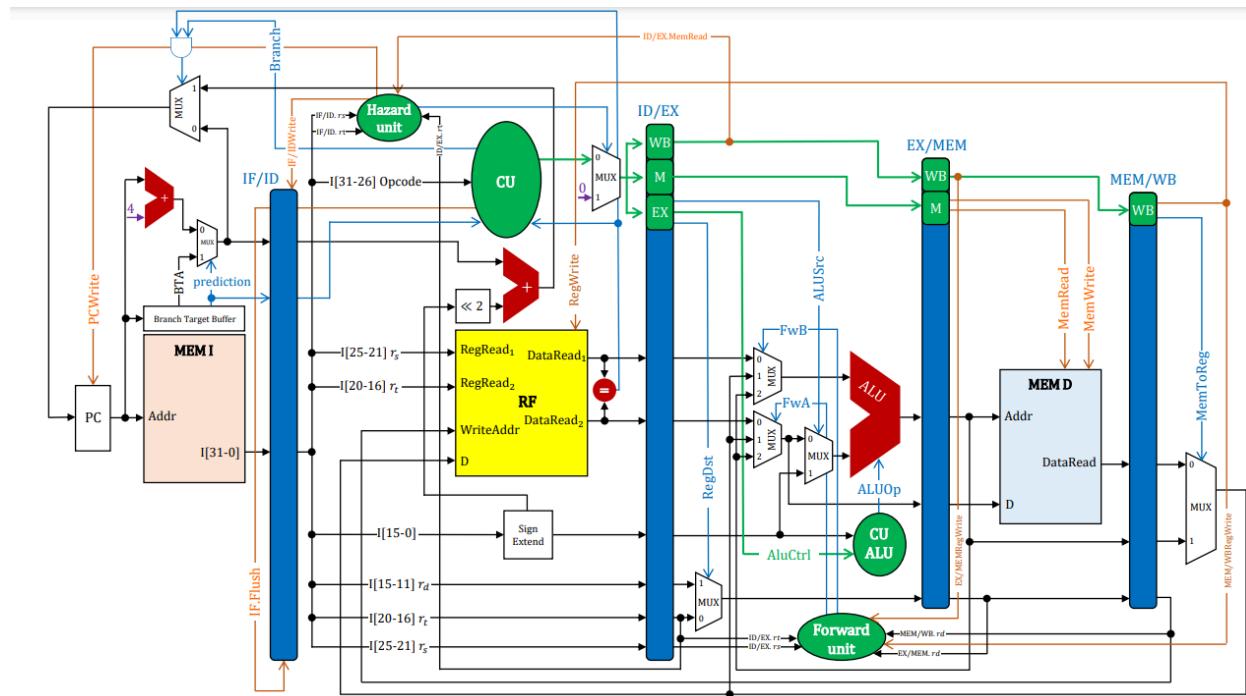
- Estendiamo a 2 bit: **2-bit prediction**
- I due bit di salto ora rappresentano lo stato corrente di una FMS (Moore), **una per ogni branch** (riga della tabella)
- **L'uscita** è il bit di salto usato per la predizione, il modello di transizione
- **Modello di transizione:** cambio la predizione solo dopo due errori consecutivi (con 1 bit era dopo ogni errore)



La transizione della predizione, avviene secondo la tabella a destra.

Si parte di default a 00, se la branch non è ancora parte dello storico.

Da ogni stato, esce un valore (quella branch è stata eseguita! si scopre se ha seguito la predizione o meno)



Eccezioni

Le **branch** e le **jump** causano deviazioni strutturate del flusso di controllo: sono appositamente progettate dal programmatore affinché il programma svolga il compito desiderato, sono previste e fondamentali alle funzioni svolte

Potrebbe capitare però, che si generino delle deviazioni del flusso di controllo impreviste.

Queste deviazioni, sono dette **eccezioni**.

- Eccezione: una deviazione imprevista del flusso di controllo la cui causa può genericamente verificarsi internamente o esternamente alla CPU
- Interrupt: un'eccezione la cui causa si verifica esternamente alla CPU

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

Istruzione non riconosciuta (Reserved instruction): esempio, ho un programma compilato per ISA MIPS 2, lo voglio eseguire su MIPS 1, MIPS 2 prevede più istruzioni di MIPS 1, potrebbe capitare che prelevo un'istruzione non riconosciuta dall'ISA MIPS 1.

Si chiama reserved instruction perché quel particolare opcode o funct, identifica una combinazione non utilizzata!

Syscall: Richiesta di una risorsa al gestore dell'esecuzione (il sistema operativo).
Syscall è l'abbreviazione di "chiamata al sistema operativo", cioè chiedo al sistema operativo, tramite appositi driver, di gestire una particolare operazione.

Riconoscere la presenza di un'eccezione

La nostra CPU sarà in grado di riconoscere (e risolvere) le seguenti eccezioni:

- Istruzione non riconosciuta: Nella fase di Decode, la CU riconosce un opcode non valido (oppure nel caso di un'operazione in formato R, l'opcode è 000000, ma il campo funct non è valido)
- Overflow aritmetico: Abbiamo già un bit di overflow incaricato a verificarne la presenza!

Per la gestione dell'eccezione poi, ho bisogno di altre due informazioni:

- La causa dell'eccezione (istruzione non riconosciuta o overflow?)
- L'indirizzo della offending instruction e cioè di quella istruzione che ha causato l'eccezione.

Un componente software del sistema operativo poi (un programma, l'exception handler), riceve queste due informazioni, e agisce in base al tipo di eccezione:

- Stampando un messaggio di errore
- Terminando l'esecuzione del programma (come nel caso dell'overflow, errore critico che non permette il ripristino dell'esecuzione)
- Ripristinare l'esecuzione del programma (In questo secondo caso l'exception handler farà uso dell'informazione contenuta in EPC, un particolare registro contenente l'indirizzo dell'istruzione successiva a quella che ha causato l'eccezione e che ora è stata gestita)

Fasi gestione eccezione tramite registro causa:

1. L'Eccezione viene rilevata dall'hardware.
2. La CPU scrive in un apposito registro detto EPC (Exception Program Counter), l'indirizzo dell'istruzione successiva all'offending instruction (nel ciclo di clock in cui avviene l'eccezione questo indirizzo è $PC + 4$ calcolato al ciclo precedente e che nel ciclo corrente è stato scritto nel registro IF/ID)
3. La CPU scrive in un'altro registro, detto "registro causa", il codice dell'eccezione rilevata.

Eccezione	Codice	
Istruzione non riconosciuta	10	
Overflow aritmetico	12	

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

La tabella che associa i codici delle eccezioni, è definita dal produttore del processore.

4. Viene effettuato un salto non condizionato, all'indirizzo (ben definito), dove risiede la prima istruzione dell'exception handler, che leggendo dai registri causa e EPC, agirà di conseguenza.

Fasi gestione eccezione tramite Interrupt Vector Table (IVT):

1. L'Eccezione viene rilevata dall'hardware.
2. La CPU scrive in un apposito registro detto EPC (Exception Program Counter), l'indirizzo dell'istruzione successiva all'offending instruction (nel ciclo di clock in cui avviene l'eccezione questo indirizzo è $PC + 4$ calcolato al ciclo precedente e che nel ciclo corrente è stato scritto nel registro IF/ID)
3. L'Hardware, in base al tipo di eccezione generata, effettua un salto ad un diverso handler specifico di quella specifica eccezione.

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

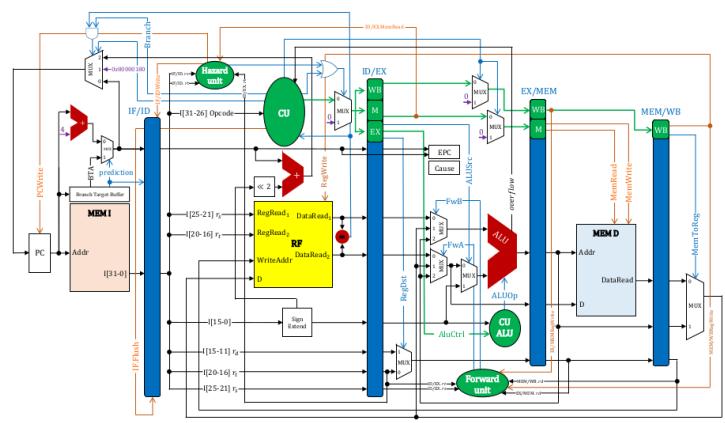
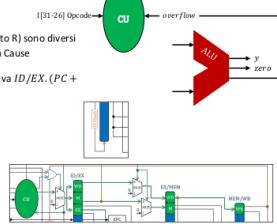
- La corrispondenza tra eccezione e indirizzo a cui saltare è memorizzata dentro una struttura dati chiamata **Interrupt Vector Table**: è una tabella dove data una causa di eccezione si ottiene l'indirizzo a cui saltare per cedere il controllo all'handler specifico di quella eccezione
- MIPS non usa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

Adattare il datapath per la gestione delle eccezioni:

Datapath per le eccezioni

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

- Aggiungiamo i registri EPC e Cause
- Estendiamo la CU in modo che:
 - Se durante la fase di ID l'opcode (o il campo funct in caso di formato R) sono diversi da quelli conosciuti scriva IF/ID, (PC + 4) in EPC e il codice 10 in Cause
 - Se durante la fase di EX la ALU attiva in uscita il bit di overflow scriva ID/EX, (PC + 4) in EPC e il codice 12 in Cause
- In caso si sia verificato a), la CU deve scartare la offending instruction dalla fase di ID e la successiva che in quello stesso ciclo di clock ha completato la fase IF
- In caso si sia verificato b), la CU deve scartare la offending instruction dalla fase di EX e le due successive che nello stesso ciclo di clock hanno completato, rispettivamente, le fasi di ID e IF
- Aggiungere 0x80000180 tra i segnali da poter scrivere nel PC (con logica di selezione)



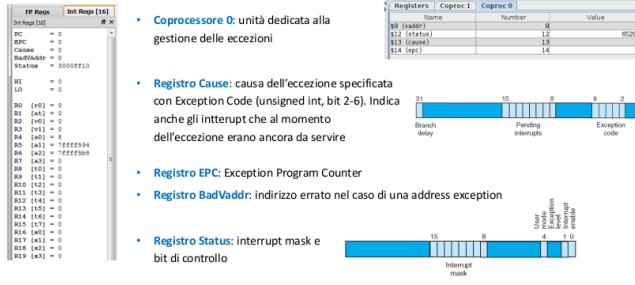
Eccezioni in MIPS:

Eccezioni in MIPS

- Nella pratica i tipi di eccezioni riconosciute da una architettura sono più di due e includono anche interrupt causati da eventi esterni, nella nostra architettura pipeline semplificata non le abbiamo considerate
- Se ammettiamo anche cause esterne (ad esempio la richiesta di una periferica di I/O) il concetto di offending instruction non è sufficiente a descrivere l'informazione che viene salvata in EPC
- Chi è la offending instruction in caso di interrupt? La risposta non è ovvia e richiede delle scelte architettoniche su dove interrompere la pipeline
- Scelta più logica:** non fare flush di nessuna istruzione, eseguire l'interrupt handler e ripristinare l'esecuzione dall'istruzione all'indirizzo che era contenuto in PC prima che iniziasse la successiva fase di IF

Number	Name	Cause of exception
0	Int	internal hardware
1	ADR	address error exception (load or instruction fetch)
5	ADRIS	address error exception (store)
6	BE	bus error on instruction fetch
7	BEW	bus error on memory read or store
8	Sys	syscall exception
10	BP	breakpoint exception
11	CPL	coprocessor unimplemented
12	DF	divide by zero exception
13	Tr	trap
15	FPE	floating point

Eccezioni in MIPS



Aumentare profondità pipeline:

Profondità della pipeline

La pipeline che abbiamo analizzato fin'ora, si basa su 5 stadi (ha ILP pari a 5, cioè si basa su parallelismo a 5 stadi).

Esistono due modi per aumentare l'ILP (Instruction level parallelism), ovvero il numero di istruzioni che può eseguire contemporaneamente la CPU, possiamo seguire due approcci:

1. Scalabilità Orizzontale: aumentare gli stadi della pipeline, ogni stadio esegue micro-operazioni elementari, la durata del ciclo di clock diminuisce e il throughput aumenta
2. Scalabilità Verticale: replicare le risorse hardware in modo che possano entrare in pipeline più istruzioni alla volta, nello stesso ciclo di clock. Una pipeline di questo tipo si chiama **multi-issue** (a lancio multiplo) a k vie. Il numero delle vie è chiamato IPC (Instruction per clock cycle) ovvero il numero massimo di istruzioni che la pipeline è in grado di lanciare in un singolo ciclo di clock. Questo implica che bisogna rivedere la gestione degli hazards.

Il problema principale della pipeline multi-issue è come riempire gli issue slot.

Issue slot: in una pipeline con k vie ad ogni ciclo di clock dobbiamo riempire uno slot con k istruzioni che verranno lanciate nella pipeline

contemporaneamente.

Non sempre è possibile riempire completamente uno slot con k istruzioni, in questi casi i posti vuoti verranno lasciati tali o occupati dalle nop.

Scheduling: è l'operazione che organizza ad ogni ciclo di clock i lanci multipli.

- **Scheduling statico (CPU Pipeline Multi-Issue Statica):** il compilatore, effettua un'analisi statica del codice, capisce quali istruzioni possono essere eseguite in parallelo, e le riorganizza in "issue packets" già composti. La CPU si limita solo ad eseguire le istruzioni così come sono organizzate.

Esempio:

- Un compilatore potrebbe vedere che due istruzioni non hanno dipendenze tra loro (ad esempio, una istruzione che carica un dato dalla memoria e un'altra che somma due registri) e potrebbe programmare queste istruzioni per essere eseguite in parallelo su diverse unità funzionali del processore.
- **Scheduling dinamico (CPU Pipeline Multi-Issue Dinamica):** la CPU, a runtime, può effettuare dinamicamente ulteriori ottimizzazioni, sul codice già precedentemente riorganizzato dal compilatore, tramite tecniche cui l'esecuzione speculativa.

Il Processore ha quindi una logica interna, con cui può analizzare le dipendenze tra le istruzioni in tempo reale, effettuare ulteriori riorganizzazioni, e speculare sull'esecuzione di queste, in modo da riempire le bolle lasciate nel codice fornito dal compilatore (che effettua un'analisi solo di tipo statica).

Speculazione:

Speculazione

- Sia con multi-issue statica (SW) che dinamica (HW) la maggiore difficoltà nel fare scheduling è quella di **non conoscere gli effetti delle istruzioni prima di averle eseguite**
 - **Ma è proprio da quegli effetti che dipende la correttezza dello scheduling**
 - **Esempio:** come posso riempire il posto libero nell'issue packet che sta per essere lanciato?
 - add \$t0 \$2 \$t3
sw \$t0 0(\$\$s1)
#: # no accessi a mem.,
#: # no uso di \$t1
 - issue packet
 - add \$t0 \$2 \$t3
lw \$t1 0(\$\$s3)
 - alla pipeline multi-issue a 2 vie
 - Posso inserire la sw?
 - **No!** Le istruzioni che vengono lanciate insieme hanno un parallelismo totalmente sincronizzato sulle sotto-fasi (quando una è in una certa fase X anche l'altra lo è!) e la sw ha bisogno del risultato della add, il forwarding è impossibile
 - Posso inserire la lw?
 - **Forse!** Se l'indirizzo a cui accede è diverso da quello della sw allora si potrebbe! Ma questo, in generale, non è deducibile dall'analisi del codice
- Esempio di una **speculazione**: scommettere che l'indirizzo di accesso della lw sarà diverso e inserire l'istruzione nello issue packet

Nell'esempio, la pipeline è multi-issue a 2 vie, quindi in un ciclo di clock, entrano due istruzioni, in due pipeline diverse.

La prima istruzione inserita è la ADD, la seconda, NON può essere la SW, perchè la SW necessita di operare su un dato che non è ancora stato prodotto (spostare contenuto di \$t0 in RAM), e che verrà prodotto il parallelo alla SW (perchè sono in due pipeline diverse, la ADD e la SW si troveranno in ogni ciclo di clock nello stesso stadio).

Posso inserire la LW? Forse!, se \$\$s1 e \$\$s3 contenessero lo stesso indirizzo? Eseguendo prima la LW, preleverei dalla RAM un dato sbagliato, quando mi servirebbe il dato "aggiornato" dalla SW.

La Speculazione è questo, scommettere che la LW non operi sulla stessa cella di memoria della SW.

Devo introdurre dei meccanismi di controllo della speculazione: se "ci ha azzeccato", ottengo dei vantaggi, altrimenti, devo effettuare un'operazione di roll back, e devo annullare gli effetti dell'istruzione che è stata erroneamente eseguita (la LW in questo caso).

- **Per il compilatore:** prevedere delle istruzioni speciali che controllino l'esito della speculazione e, nel caso, riparino gli errori
- **Per la CPU:** il risultato delle istruzioni speculative vengono temporaneamente mantenute in un buffer fino a quando non è possibile stabilire l'esito della speculazione
 - Se l'esito è positivo i risultati vengono scritti nel RF o in memoria, questa operazione spesso si chiama **commit**
 - Se l'esito è negativo i risultati vengono scartati e si mandano in esecuzione le istruzioni corrette

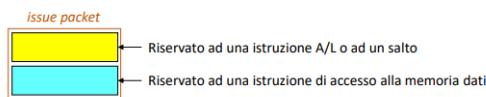
E se un'operazione speculata genera eccezione? Si "aspetta" prima di controllare che la speculazione sia corretta, e in caso risolvere l'eccezione (se

la speculazione non è corretta, è inutile risolvere un'eccezione di un'istruzione che non sarebbe dovuta esser eseguita.

Multi-issue statica:

Lancio multiplo, basata su analisi statica del codice, effettuata dal compilatore, a compile-time.

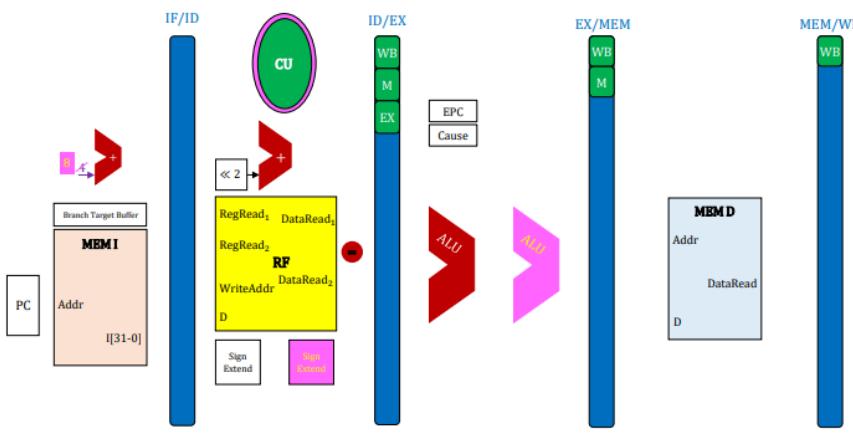
- Consideriamo una CPU con pipeline multi-issue a 2 vie
- Primo aspetto implementativo: lo issue packet non può essere arbitrario, vengono imposti dei vincoli in modo da semplificare la gestione degli hazard



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

Implementazione Hardware:

Multi-issue statica (data path)



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

Per far si che ad ogni ciclo di clock vengano prelevate due istruzioni (64 bit alla volta), cambio l'immediato in ADD con PC in 8.

Estendo i registri intermedi pipeline, per far si che possano contenere i risultati parziali di due istruzioni, e in generale devo arricchire/duplicare ogni

componente.

Dalla memoria istruzioni, usciranno 64 bit, due istruzioni.

Dal register file, usciranno 4 dati, duplico le ALU ecc...

Scheduling statico:

Multi-issue statica (scheduling)

- Scheduling di questa sequenza di istruzioni

loop: `lw $t0 0($t1)`

`add $t0 $t0 $s2`

`sw $t0 0($s1)`

`subi $s1 $s1 4`

`bne $s1 $zero loop`



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



Mettendo la LW nel primo issue packet, mi accorgo che non ci sono operazioni AL che non dipendono da T0, devo necessariamente riempire con un NOP.

Per il secondo issue packet, riempio lo slot con la ADD, mi accorgo che la SW però ha bisogno di aspettare il risultato della ADD, devo riempire con NOP.

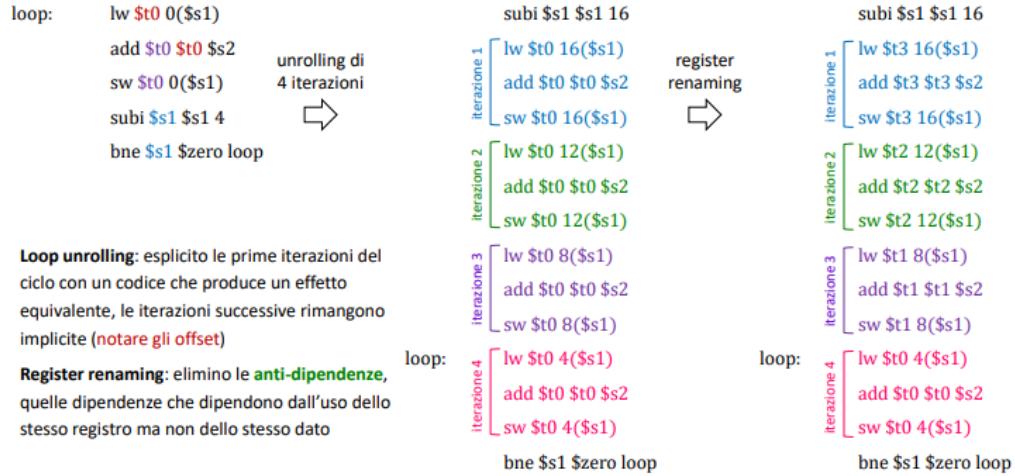
Terzo issue packet, riempio issue slot con SW, che ha ricevuto il dato aggiornato \$t0 alla ADD, in parallelo, subi può comunque eseguire, perchè la SW ha bisogno del dato "vecchio", non ha bisogno dell'aggiornamento. (SW e SUBI avvengono INSIEME, non prima una poi l'altra, quindi SW ottiene il dato giusto, e nel frattempo con questo dato la SUBI ci lavora, quindi tutto ok).

Loop unrolling:

Il Compilatore, tramite una più approfondita analisi statica del codice, può effettuare alcune ottimizzazioni nello scheduling delle operazioni:

Multi-issue statica (loop unrolling)

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



Multi-issue statica (loop unrolling)

loop:	Con loop unrolling		Senza loop unrolling	
	A/L o branch	Accesso a memoria	A/L o branch	Accesso a memoria
subi \$s1 \$s1 16	subi \$s1 \$s1 16	lw \$t3 0(\$s1)		lw \$t0 0(\$s1)
iterazione 1	lw \$t3 16(\$s1)	lw \$t2 12(\$s1)		
	add \$t3 \$t3 \$s2	lw \$t1 8(\$s1)		
	sw \$t3 16(\$s1)	add \$t2 \$t2 \$s2	lw \$t0 4(\$s1)	
iterazione 2	lw \$t2 12(\$s1)	add \$t1 \$t1 \$s2	sw \$t3 16(\$s1)	
	add \$t2 \$t2 \$s2	add \$t0 \$t0 \$s2	sw \$t2 12(\$s1)	
	sw \$t2 12(\$s1)	sw \$t1 8(\$s1)		
iterazione 3	lw \$t1 8(\$s1)	bne \$s1 \$zero loop	sw \$t1 8(\$s1)	
	add \$t1 \$t1 \$s2		sw \$t0 4(\$s1)	
	sw \$t1 8(\$s1)			
iterazione 4	lw \$t0 4(\$s1)			
	add \$t0 \$t0 \$s2			
	sw \$t0 4(\$s1)			
bne \$s1 \$zero loop				

Annotations indicate "iterazione 1", "iterazione 2", "iterazione 3", and "iterazione 4".

IPC calculations:

IPC_e = $\frac{5}{4} = 1,25$

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

IPC empirico: $IPC_e = \frac{\text{num.di istruzioni}}{\text{num.cicli di clock}} = \frac{14}{8} = 1,75$

percentuale di efficienza: $\rho = \frac{IPC_e - IPC_{min}}{IPC_{max} - IPC_{min}} = \frac{1,75 - 1}{1,75 - 1} = 0,75$

• Applicando il loop unrolling otteniamo un miglioramento del 50%

IPC = Instructions per cycle, in una pipeline a N vie, l'IPC è sempre compreso tra 1 e N (dipende dalla composizione del packet lanciato)

IPC Empirico, va interpretato in base ad IPC_{max}, indica "quanto bene è sfruttato" il parallelismo della nostra pipeline.

Multi-issue Dinamica:

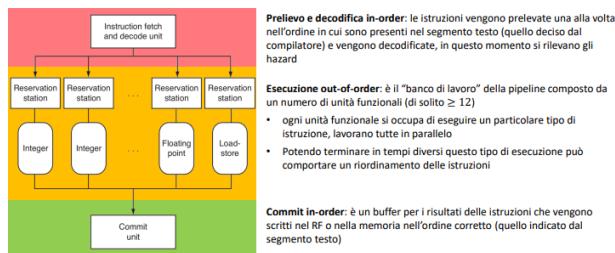
Una CPU in grado di decidere a run-time, a livello HW, come riempire gli issue-slot, è detta **superscalare**.

La CPU Pipeline Multi-Issue Dinamica, è in grado di prendere decisioni a run-time, su come riempire gli slot di lancio, in modo ancora più efficiente.

Scheduling dinamico:

La CPU superscalare, è in grado quindi a sua volta di effettuare una riorganizzazione delle istruzioni, sul prodotto del compilatore.

Questa operazione, è detta **scheduling dinamico**.



Bisogna completamente ripensare alla struttura della CPU:

C'è un macro-stadio "rosso", che si occupa del prelievo e della decodifica delle istruzioni, nel modo in cui sono state scritte nel segmento testo (ordine deciso dal compilatore).

Al momento del loro ingresso, le istruzioni vengono decodificate, e inviate (una alla volta, in base al tipo di istruzione) ad un reparto di esecuzione specializzato.

Qui nello stadio rosso, identifico anche eventuali hazard.

Nell'area gialla, ogni "blocco" di esecuzione specializzato (detto unità funzionale, ognuna con HW specifico per quel tipo di istruzione), ha un proprio buffer (Reservation Station), che è un'altra QUEUE, in cui ci sono SOLO

istruzioni di quello specifico tipo, assieme ai dati che servono per eseguire quella specifica istruzione.

Nella reservation station, le operazioni vengono preparate per l'esecuzione, quindi vengono anche prelevati i dati su cui operare, dal RF o dalla memoria.

Potrebbe capitare però, che un dato necessario, non sia ancora stato calcolato! La CPU si è già accorta degli hazard, nello stadio rosso, quindi mette in attesa l'unità funzionale dedicata, in attesa del dato.

Questo dato, sarà calcolato in un'altra unità funzionale, e appena sarà calcolato, verrà inviato alla reservation station dov'è stato richiesto, senza prima dover passare dal RF! Risparmio molto tempo!

Ora che ho il/i dati richiesti, posso riprendere l'esecuzione nell'unità funzionale.

Qui in quest'area "gialla", le istruzioni possono essere eseguite in tempistiche diverse (in parallelo), pertanto si dice che l'esecuzione è "out-of-order".

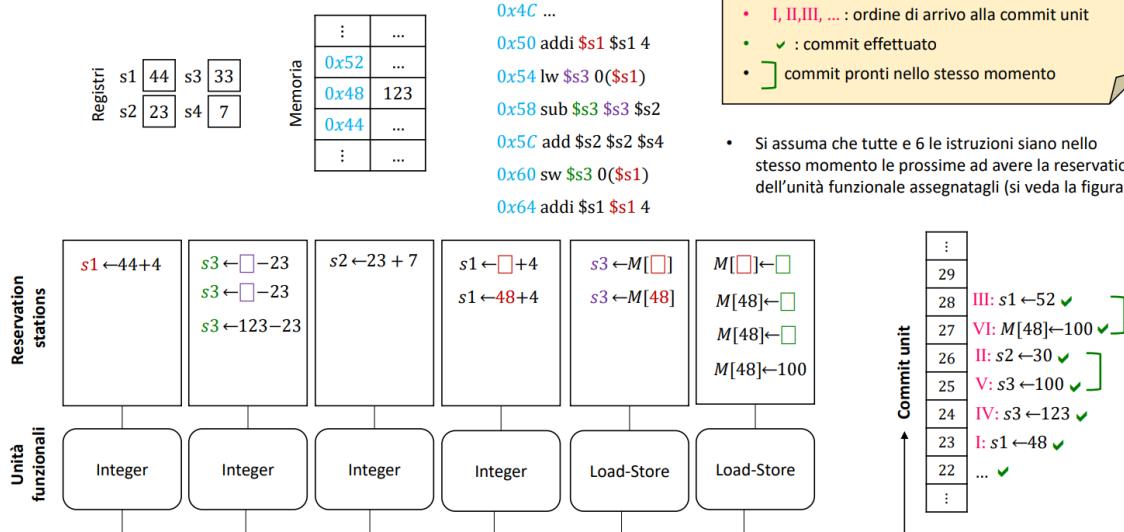
Dopo aver eseguito l'istruzione (aver calcolato qualcosa), le istruzioni assieme al dato che hanno chiesto di produrre, vanno verso la commit unit, un'altro stadio in cui si fanno solo ed esclusivamente scritture su memoria.

ES: una ADD \$7, 2, 3. Nell'area gialla, viene fatto $2+3=5$, poi all'area verde arriva la ADD con scritto "io ho il dato 5, e va scritto in \$7".

L'Area verde, si occupa di scrivere in memoria/sui registri, tutte le conseguenze delle operazioni eseguite. Qui è importante che la scrittura avvenga in-order, altrimenti genererei degli hazard.

A Esecuzioni terminate, la situazione nella commit-unit è questa:

Scheduling dinamico

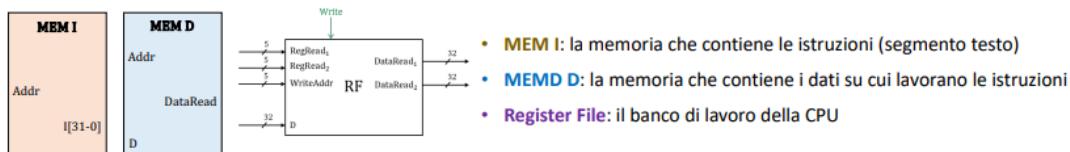


Memorie

La memoria, fin'ora è stata trattata come una "black-box", che opera tramite operazioni di lettura e di scrittura.

Siccome la CPU lavora spesso con essa, è necessario studiarne a fondo la struttura, e migliorarne al meglio le performance.

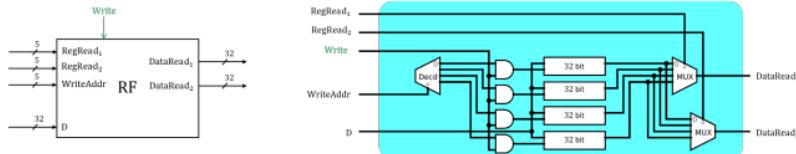
- Fino ad ora la memoria è stata rappresentata come un componente nel datapath che abbiamo utilizzato tramite la sua interfaccia: data una richiesta fatta dalla CPU (lettura o scrittura) il componente la eseguiva



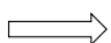
- La memoria rappresenta un punto critico legato alla performance di una CPU che necessita di essere trattato **a livello architettonico** (non solo migliorando l'efficienza della tecnologia)

Register file:

Il register file



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



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

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



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

Per memorizzare un bit, ho bisogno di un Flip Flop. Un Flip Flop è composto da due transistor in retroazione, quindi ogni dato (bit) è clonato 3/4 volte, per cercare di prevenire fenomeni di degradazione.

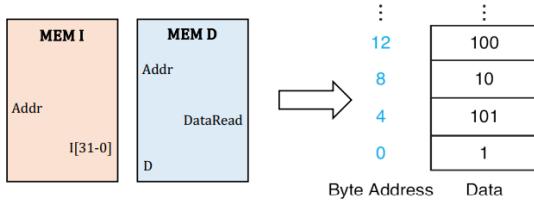


*The main difference between SRAM (Static Random-Access Memory) and DRAM (Dynamic Random-Access Memory) is that SRAM stores data using flip-flop circuits, which are faster and require continuous power to retain data, while DRAM uses capacitors (**condensatori**) to store data, making it slower but more power-efficient as it only needs periodic refreshing.*

Le Prestazioni della SRAM, sono maggiori rispetto a quelle di una DRAM, ma sono anche più costose, ecco perchè le SRAM sono utilizzate principalmente per le cache, che sono più piccole.

Memoria principale (Comunemente ed erroneamente detta "RAM")

La memoria principale (Main Memory)



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

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

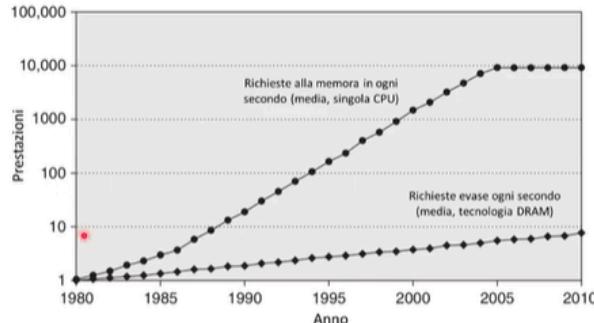
Dynamic → Necessita di refreshare le celle per mantenere tale il dato, perchè la memorizzazione si basa su condensatori, e non su flip flop.

I condensatori, se non vengono refreshati, si "scaricano", e perdono l'informazione.

DDR SDRAM → Posso effettuare operazioni di lettura/scrittura su entrambi i fronti del ciclo di clock, raddoppiando quindi l'efficienza della memoria.

Memory-Bottleneck

Processor-Memory bottleneck



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

Per migliorare le prestazioni, devo riprogettare anche l'architettura della memoria.

Gerarchie di memoria:

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



Approccio 1

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



Approccio 2

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

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

- **Primo approccio:** Prendo un libro, lo leggo, torno in biblioteca, cambio libro, torno a casa, lo leggo, torno in biblioteca...
- **Secondo approccio:** Prendo il massimo dei libri che la biblioteca mi consente di prendere, torno a casa, li leggo tutti, ed EVENTUALMENTE torno in biblioteca a prenderne altri.

L'Esempio viene applicato alla memoria, ponendo un parallelismo tra il numero di libri presi, e il numero di word prelevate dalla memoria, e il tragitto casa-biblioteca, e un'operazione di accesso a memoria (che ha un costo).

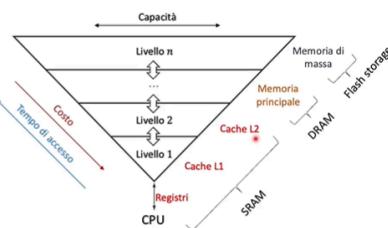
Il Secondo approccio, risulta migliore, e si giustifica per due principi ben precisi, che sono quelli di località spazio-temporale.

- **Principio di località temporale:** se dopo un tempo T ho la necessità di accedere ad un dato, è probabile che dopo un tempo T + Epsilon (da lì a breve), dovrò di nuovo accedere allo stesso dato.
Esempio: se sto facendo la ricerca sul risorgimento, io sono su una pagina specifica, leggo un paragrafo, scrivo, e poi dopo poco rileggerò la stessa pagina.
- **Principio di località spaziale:** se dopo un tempo T ho la necessità di accedere ad un dato, è probabile che dopo un tempo T + Epsilon (da lì a breve), avrò necessità di accedere ad un dato molto fisicamente vicino a quello già letto
Esempio: leggo un paragrafo, scrivo, poi leggerò il paragrafo dopo (fisicamente vicino).

Confidando in questi due principi, posso costruire una memoria che implementa il secondo approccio della biblioteca, cioè prelevo non uno, ma più libri alla volta (più word alla volta).

Gerarchia di memoria

- Il sistema di memoria è progettato come una gerarchia a livelli
- I trasferimenti possono avvenire solo tra livelli adiacenti
- Anche per i dati viene mantenuta una organizzazione gerarchica: il livello i contiene un sottoinsieme dei dati immagazzinati al livello $i + 1$
- I livelli che stanno tra il banco registri e la memoria principale vengono chiamati **cache** (il *nascondiglio*, una memoria nascosta)
- **Livelli bassi:** alte performance, ma costi alti quindi bassa capacità
- **Livelli alti:** basse performance, ma costi bassi quindi alta capacità
- Le richieste di accesso (lettura o scrittura) sono gestite seguendo la gerarchia
- Richiesta di un dato d al livello i , il dato è contenuto nel livello i ?
 - **Sì:** la richiesta viene evasa (lettura o scrittura)
 - **No:** la richiesta viene inoltrata al livello $i + 1$ sottostante



La CPU è a capo di questa gerarchia, e si interfaccia direttamente (di norma, a meno che il software non bypassi appositamente la memoria cache) SOLO coi registri (velocissimi, bassati su tecnologia SRAM), non interagisce direttamente con la memoria centrale.

I Registri, interagiscono con la cache di primo livello. La cache di primo livello, fornisce supporto ai registri, e riceve supporto dalla cache di secondo livello, e così via fino alla memoria di massa.

La forma a triangolo rovesciato, è data dal fatto che la memoria di massa, è più capiente della memoria principale, della cache ecc..

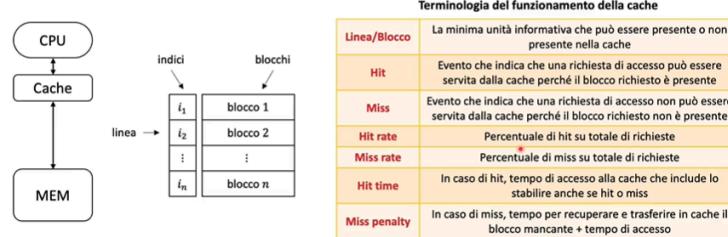
Allo stesso tempo, aumentano i costi (Cache e registri si basano su SRAM, più costosa della DRAM e della tecnologia flash) e le velocità andando verso la CPU.

La CPU, quando deve eseguire un'istruzione, chiede al register file i dati di cui ha bisogno, se nel register file ci sono, bene, la richiesta viene evasa, altrimenti il register file chiede alla cache L1 se li ci sono i dati di cui ha bisogno, se si, dalla cache L1 il dato viene trasferito nel register file, e dato alla CPU, altrimenti si va nella cache L2, L3, fino a casi estremi, in cui si può anche arrivare alla memoria di massa (in caso di swap per esempio, cioè la memoria centrale è finita, e ho bisogno di trasferire dei dati nella memoria di massa).

Cache:

Modello della cache

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



La Cache lavora per letture e scrittura, non più per word, ma per linee/blocchi (in cache la minima unità informativa si chiama così, una linea o blocco può essere costituito da più word).

Cache hit → chiedo un dato nella cache, ed è presente in esso

Cache miss → il dato non è presente in cache, la richiesta passa agli altri livelli di cache/alla memoria centrale.

Hit e miss rate, sono le percentuali di hit e miss sul totale di richieste di dati alla cache.

Hit time e Miss penalty, sono le tempistiche che servono per recuperare un dato, in caso di hit o di miss (miss richiede tempistiche maggiori, ecco perchè si chiama penalty).

Mappatura diretta

Mappatura diretta

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

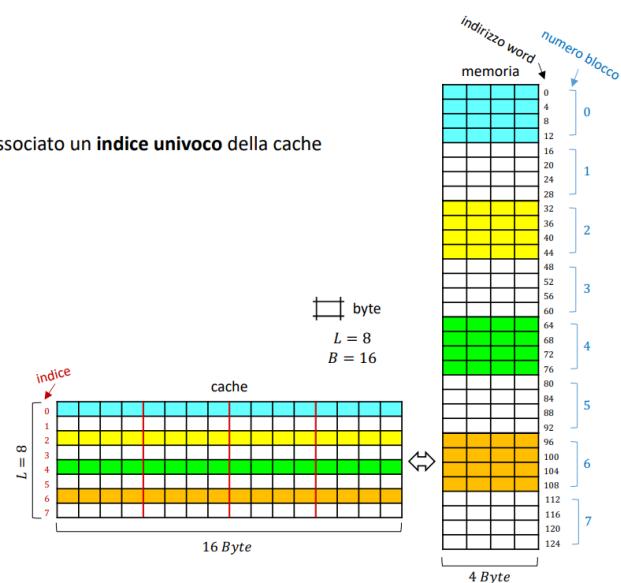
Soluzione

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

$$N(d) = \text{div}(M(d), B)$$

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

$$I(d) = \text{mod}(N(d), L)$$



La memoria centrale, è ovviamente più grande della cache, quindi si crea una concorrenza per decidere quali "blocchi" di ram vengono trasferiti in cache (cioè quali gruppi di 4-64 (dipende dall'implementazione) word vanno trasferiti).

Dato un blocco di memoria centrale, nell'esempio formato da 4 word = 16 bytes, come scelgo quale indirizzo della cache associare ad un determinato blocco?

Questo non significa che ogni blocco della memoria centrale viene copiato nella cache, ma a livello ipotetico, dovrei assegnare per ogni blocco della memoria centrale, un indirizzo.

Esempio: a livello ipotetico, se dovessi trasferire questo X blocco di memoria nella cache, a che indirizzo andrebbe? Capiamo quindi che con soli 8 linee (in esempio), devo associare tutta la memoria centrale, quindi blocchi diversi della memoria centrale avranno indirizzo in cache uguale.

Dato l'indirizzo di una word (nella memoria centrale), in che blocco della memoria centrale si trova?

$N(d) = \text{div}(M(d), B)$, cioè la parte intera della divisione dell'indirizzo della word, per il numero di bytes contenuti in un blocco della memoria.

Quindi, dato l'indirizzo del dato X, se so che ogni blocco contiene Y bytes, il dato X si troverà nel blocco X/Y (parte intera)

Esempio: dalla slide, in che blocco si trova la word di indirizzo 36? $36/16 = 2$ resto 4, e infatti si trova nel blocco 2.



I blocchi della memoria centrale, sono più di 8! Non come in slide! Potrei avere anche come blocco associato ad una word il numero 3000.

Ora, in memoria centrale, ho più blocchi di quanti ne posso mettere in cache, come determino il blocco della memoria centrale XX in che indirizzo della cache va? Io divido in modulo per il numero di blocchi della cache che ho a disposizione.

Esempio: il blocco 4080 della memoria centrale, se dovessi metterlo nella cache, in che indirizzo andrebbe? $\text{MOD}(4080, 8)$ (in cache ho 8 Linee) = 1, sarebbe messo nella linea 1



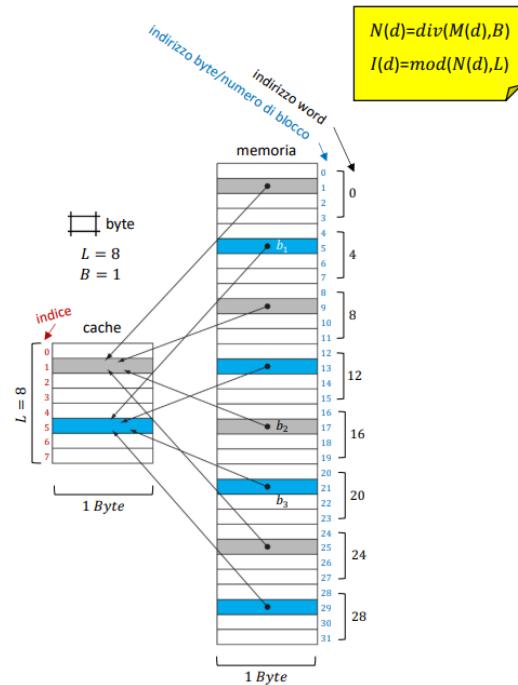
Reminder, in memoria centrale, ci sono più blocchi di quanti ne posso mettere in cache, quindi più blocchi "competono" per lo stesso posto in cache.

Mappatura diretta

Esempio:

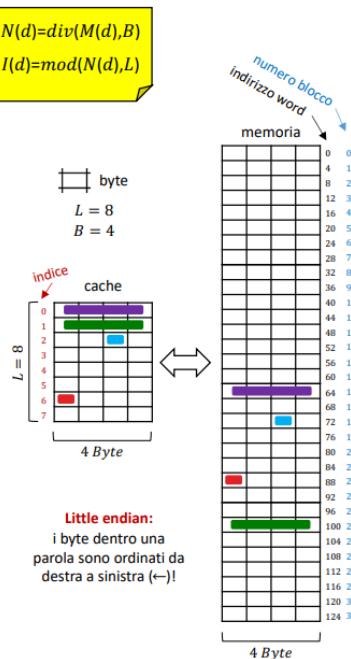
- A quale indice di linea di cache dovrebbero trovarsi i byte b_1, b_2 e b_3 ?
- Attenzione!** In questo esempio un blocco contiene un byte (l'elemento indirizzato in memoria), quindi l'indirizzo in memoria coincide con il numero di blocco!

- $M(b_1) = 5$
- $N(b_1) = \text{div}(5,1) = 5$
- $I(b_1) = \text{mod}(5,8) = 5$
- $M(b_2) = 17$
- $N(b_2) = \text{div}(17,1) = 17$
- $I(b_2) = \text{mod}(17,8) = 1$
- $M(b_3) = 22$
- $N(b_3) = \text{div}(22,1) = 22$
- $I(b_3) = \text{mod}(22,8) = 6$



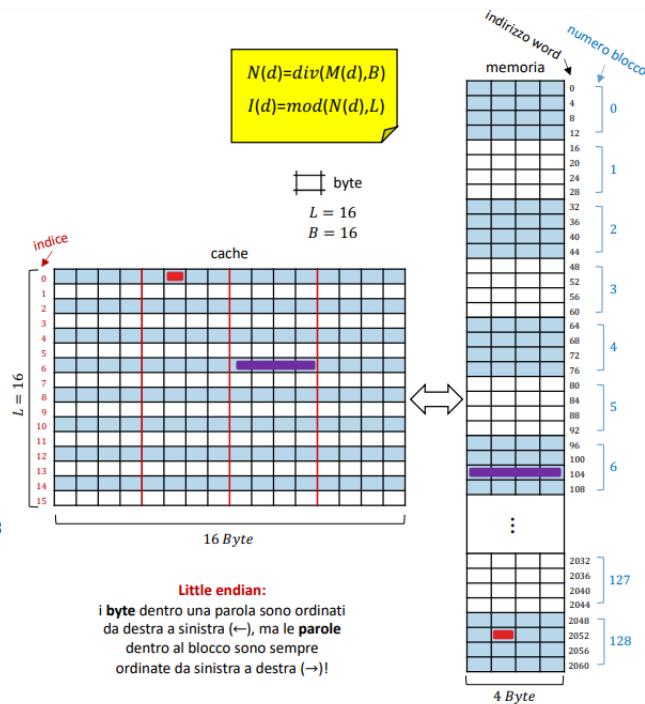
Mappatura diretta

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



Mappatura diretta

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



Divisione intera per potenza della base

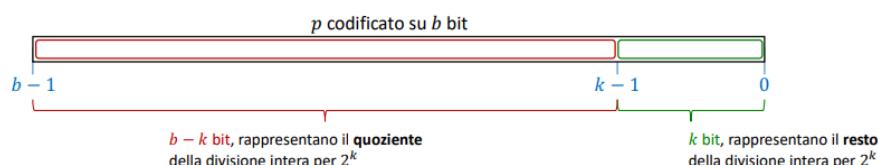
- Le cache che consideriamo non hanno dimensioni arbitrarie
 - Il numero di linee è sempre una potenza di 2: $L = 2^k$
 - La dimensione di un blocco è tipicamente definita da un numero intero di parole, se nel blocco ci sono 2^m parole allora $B = 2^{m+2}$, quindi anche B è una potenza di 2
- I calcoli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

In base dieci:

- $\text{div}(104,16) = \text{div}(104,2^4) = 6$
- $\text{mod}(104,16) = \text{mod}(104,2^4) = 8$
- Dato un intero binario senza segno p codificato su b bit, dalla divisione intera tra p e la k -esima potenza di due 2^k si ha che:
 - Il quoziente della divisione è dato dalle $b - k$ cifre **più significative** di p (estraibili con uno shift a destra di k posizioni applicato a p)
 - Il resto della divisione è dato dalle k cifre **meno significative** di p

Rappresentando il dividendo in base due:

- $\text{div}(1101000,2^4) = 1101000$
- $\text{mod}(1101000,2^4) = 1101000$

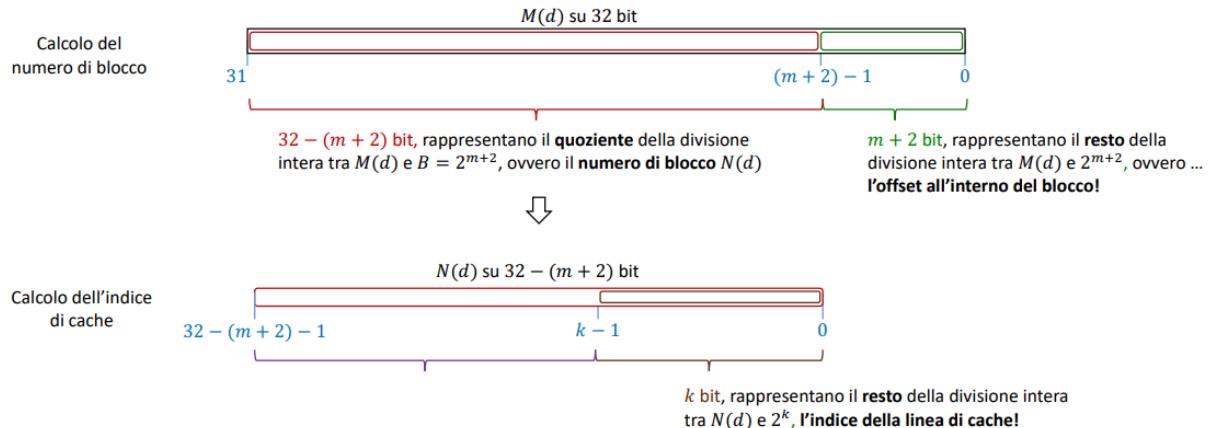


$$N(d) = \text{div}(M(d), B)$$

$$I(d) = \text{mod}(N(d), L)$$

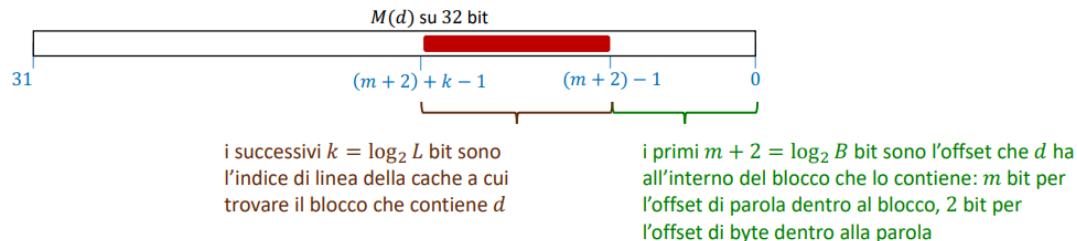
Accesso alla cache

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



Mappatura diretta

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



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

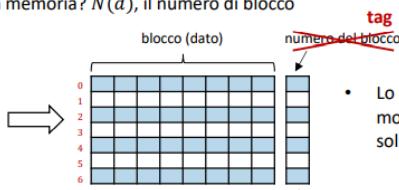
Mappatura diretta

- **Secondo problema:** quando si accede ad un blocco immagazzinato dentro ad una linea di cache, come si può stabilire se è quello effettivamente richiesto visto che quella stessa linea può essere condivisa da diversi blocchi della memoria?

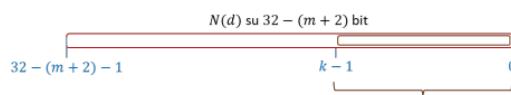
- **Soluzione:** nella linea di cache aggiungo, oltre al blocco stesso, l'informazione per identificarlo

- Cosa identifica in modo univoco un blocco in memoria? $N(d)$, il numero di blocco

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



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



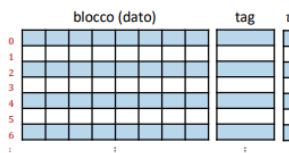
- i k bit meno significativi di $N(d)$ coincidono con l'indice della cache
- Sono implicitamente determinati dalla posizione della linea di cache in cui sta quel blocco! Non serve memorizzarli esplicitamente!
- Per identificare il blocco basta memorizzare i $32 - (m + 2) - k$ bit più significativi del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

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

Mappatura diretta

- **Terzo problema:** il blocco presente in una linea di cache potrebbe non essere valido (per esempio perché la linea non è stata ancora inizializzata oppure il blocco è diventato obsoleto)

- **Soluzione:** nella linea di cache aggiungo, oltre al blocco stesso e al tag, un bit di validità v



Dimensionamento della cache:

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

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

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

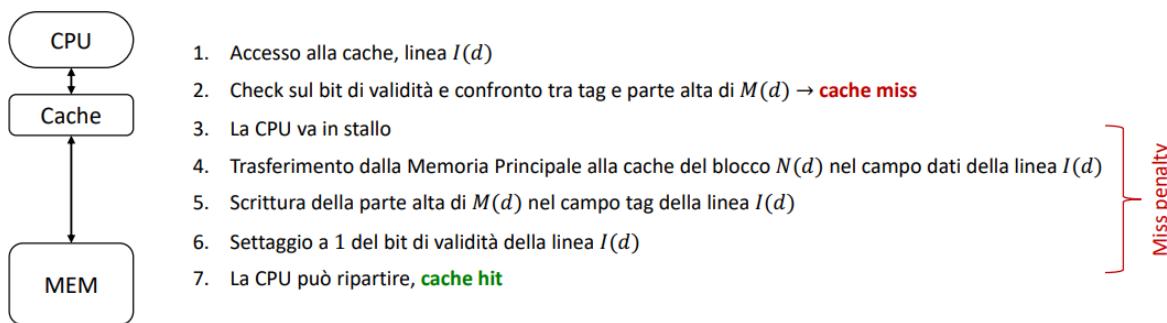
Mappatura diretta

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

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

Gestione delle miss

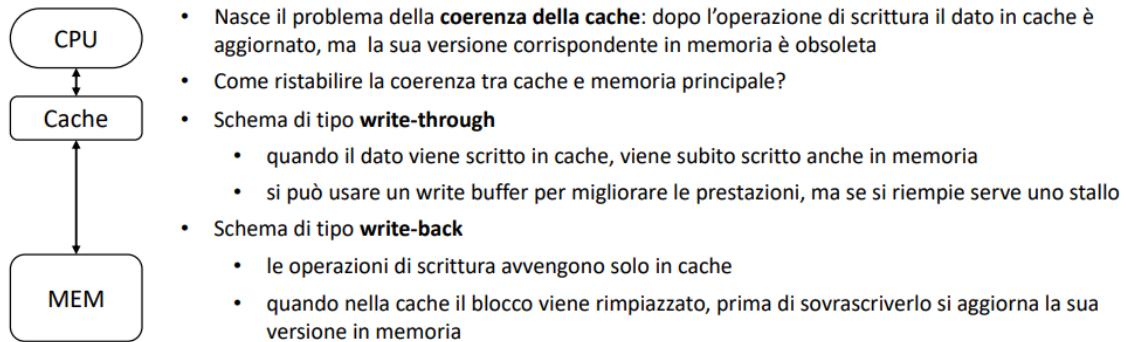
- Dato l'indirizzo di un dato $M(d)$ sappiamo dove cercarlo in cache e come stabilire se è il dato corretto e se è valido
- Cosa succede quando una delle due condizioni non si verifica? Si assuma che un'istruzione (es. una `lw`) richieda accesso in lettura ad un dato d



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

Accessi in scrittura

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



Cache associative

Nella cache che abbiamo analizzato fino ad ora, vi è una "mappatura diretta" tra i dati e le linee della cache, cioè un dato in memoria, può essere memorizzato unicamente in una determinata linea, pre-calcolabile con le formule viste precedentemente.

Questo approccio rende semplice l'operazione di accesso (non devo "cercare" il dato, so già dov'è), ma rende allo stesso tempo, poco efficiente l'uso della memoria.



Agli estremi delle soluzioni di progettazione della cache, ho cache a mappatura diretta (un dato può essere memorizzato solo in una specifica linea), e la cache completamente associativa, cioè in cui un dato, potrebbe trovarsi su una qualsiasi linea della cache.

Cache fully-associative

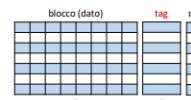
Per strutturare una cache fully associative, in cui un dato potrebbe ipoteticamente trovarsi su ogni linea, devo ripensare alla modalità di accesso ai dati.

Per cercare un dato in una cache a mappatura diretta, io specificavo alla cache, il numero di linea di cui volevo che mi fosse restituito il dato.

Nella cache fully-associative invece, io non so dove sta il dato, non posso dire "dammi il dato alla linea 4", non so il dato dov'è.

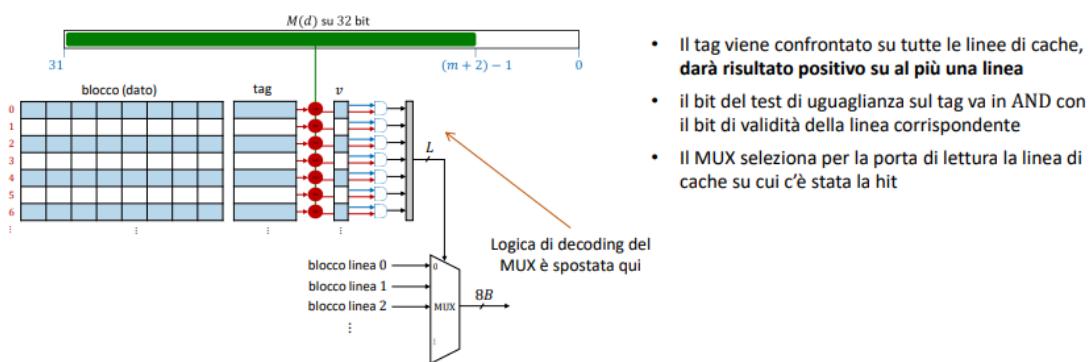
Quindi la cache fully-associative, ha un unico input, su cui viaggia un TAG, che identifica un dato univocamente (come fosse un dizionario, chiave-valore).

- **Memoria associativa:** l'accesso non avviene tramite un indirizzo, ma con una parte del contenuto del dato che lo identifica univocamente (un valore detto **chiave**)
- **Domanda:** quale parte del **contenuto** di una linea di cache identifica univocamente il blocco memorizzato in quella linea?
- **Risposta:** il **tag**
- **Attenzione**
 - non esiste l'indice di linea, è come se ci fosse un'unica linea, $L = 1$, quindi $k \leftarrow 0$
 - il tag è quindi dato da tutti i $32 - (m + 2)$ bit del numero di blocco



L'indice di linea, era prima implicito, ora va incluso nel tag! Il campo tag passa da $32 - (m+2) - k$ bits, a $32 - (m+2)$. Corrisponde al numero del blocco in RAM.

Cache fully-associative



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

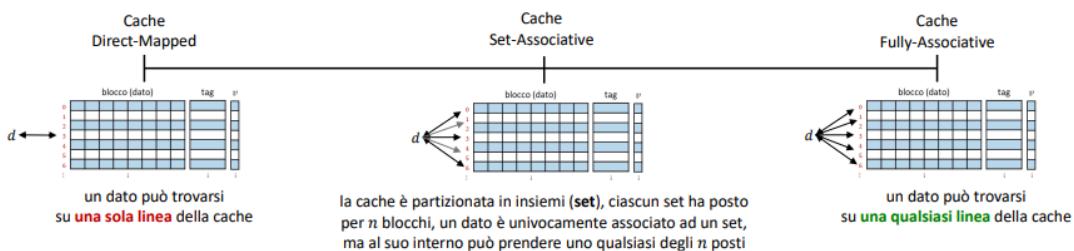
Quei "cerchiolini" rossi, vicino ad ogni linea della cache, sono dei comparatori, che comparano il tag di quella linea, con quello in input nella cache (il tag da cercare entra direttamente nei comparatori, come da linea verde nel disegno).

Il risultato di questi comparatori può essere o tutti zero (nessuna linea ha il dato richiesto, cache miss), oppure al più una linea restituisce uno (cache hit).

Una volta identificato il dato, il suo tag va in AND col bit di validità, perchè in caso la cache avesse il dato cercato, ma non fosse valido, non deve essere restituito!

All'entrata del MUX, devo immaginare che vadano in ingresso le linee dei blocchi dati, da SX.

Cache set-associative



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

Per quanto riguarda la modalità d'accesso, la soluzione è ibrida tra la cache a mappatura diretta, e la cache fully associative.

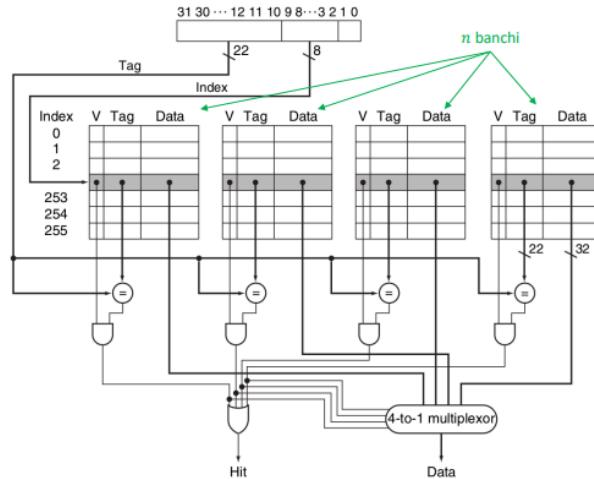
Cioè io alla cache dico, "vai a cercare il dato nel SET X" (Un set è una riga condivisa in questo caso tra 4 chip di memoria diversi), e in quel set poi, la ricerca si svolgerà tramite TAG, come nella fully-associative.

Cache set-associative

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

Esempio:

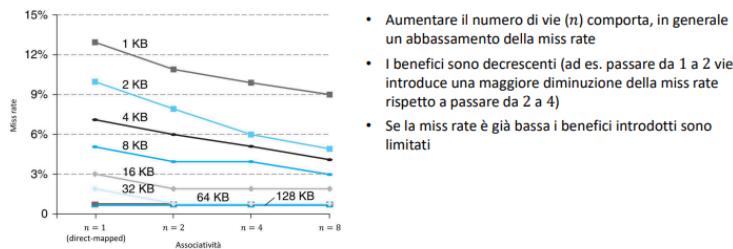
- Cache associativa a 4 vie da 4 KiB e blocchi da 1 parola
- numero di set $L = \frac{4 \times 2^{10}}{4 \times 4} = 256$, quindi $k = \log_2 256 = 8$ bit di indice
- $m + 2 = \log_2 B = 2$, quindi $m = 0$ (c'è solo offset di byte perché un blocco coincide con una parola)
- Tag su 32 – 2 – 8 = 22 bit



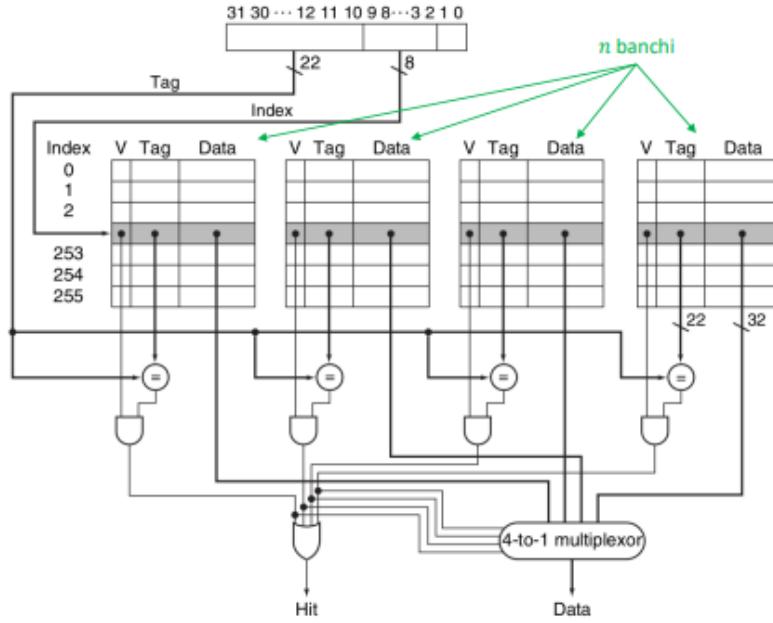
Ho quindi per ogni set, un "metadato", e poi per ogni banco del set, un TAG associato.

Grado di associatività

Grado di associatività



Aumentare il numero di vie di una cache, vuol dire riprogettarla, aggiungendo più banchi di memoria (ingrandirla).



Accessi direct-mapped

Cache a 4 linee (in verticale, 4 righe della tabella)

Accessi direct-mapped

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

Memoria	48		63	

28	55	16	00010000	$M(d)$
...	...	28	00011100	$M(d)_2$
16	20	48	00110000	$N(d)$
				$I(d)$
				tag_2

block	tag	v															
-	-	0	20	1	1	63	3	1	20	1	1	20	1	1	63	3	1
-	-	0	-	-	0	-	-	0	-	-	0	-	-	0	-	-	0
-	-	0	-	-	0	-	-	0	-	-	0	-	-	0	-	-	0
-	-	0	-	-	0	-	-	0	-	-	0	-	-	0	-	-	0

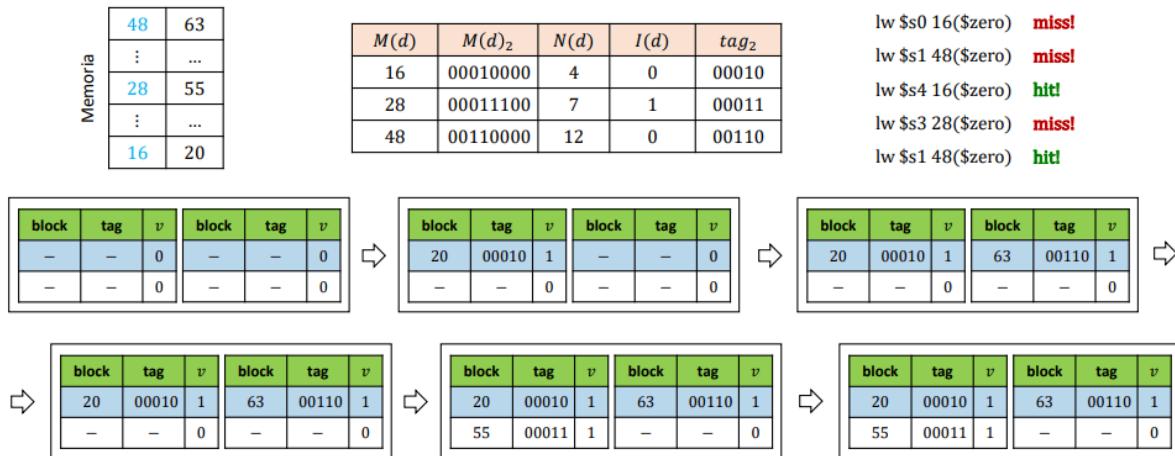
i bit altri non riportati nelle codifiche binarie di indirizzi e tag sono da considerarsi implicitamente pari a 0 (vale in questa slide e nelle due successive)

Accessi set-associative

Stessa cache a 16 bytes, 4 linee fisiche, ma divise su due banchi (due linee logiche, ognuna mappata su altre due linee).

Accessi set-associative

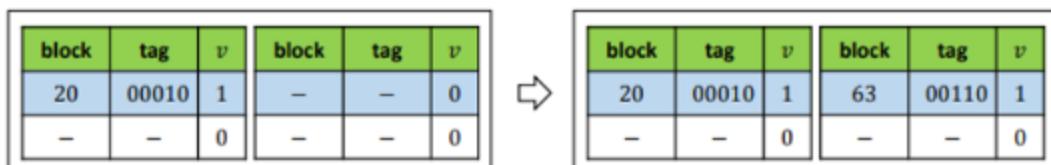
- Cache da $16 B$, set-associativa a 2 vie, $L = 2, B = 4$ (due set da 2 posti)
- $k = 1, m = 0$, tag su 29 bit



L'Indice di linea ora è a un bit solo (ho due linee logiche su cui un dato può essere mappato)

Il dato di indirizzo di memoria 48 e quello di indirizzo 16, hanno lo stesso numero di linea, ma sappiamo che ogni linea, ha 2 posti disponibili! (ogni linea è mappata su altre due linee).

La mia cache è divisa quindi in due tabelle, due banchi da due posti ciascuno.

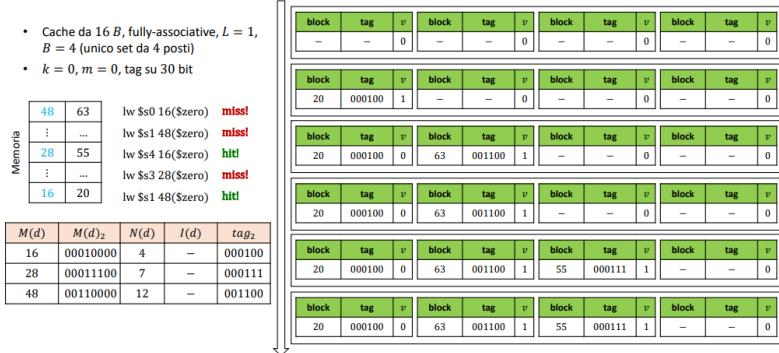


La competizione tra questi due dati è stata risolta: stessa linea (0), posti diversi.

Quando andrò a cercare il dato, andrò a identificare la linea 0 tramite indicizzazione, poi troverò il dato corretto analizzando il tag, tra i dati sulla stessa linea (che in questo caso sono solo due).

Accessi fully-associative

Accessi fully-associative



In questa cache, un dato può trovarsi in una qualsiasi cella di memoria nella cache.

La ricerca è effettuata unicamente tramite comparazione del tag, che viene esteso su 30 bit per poter indicizzare tutta la cache.

Ho una linea unica, divisa in 4 posti.

Associazione del “posto” all’interno di un set.

Nella cache fully associative, ho un unico set, con N posti.

All’interno di quel set, ho deciso di memorizzare i dati da sinistra verso destra (per semplicità), ma ci sono delle strategie ben definite:

- **Random:** soluzione semplice da implementare, a costo zero (si dice strategia “non informata”, cioè non ha bisogno di nessun prerequisito per essere implementata).

Con alti gradi di associatività può funzionare bene

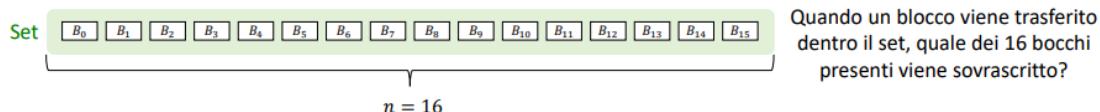
- **Least-Recently Used (LRU):** si sostituisce il blocco che non viene acceduto dal maggior tempo, tra quelli correntemente nel set

- **Pseudo-LRU:** LRU approssimato (Tengo traccia del blocco meno utilizzato, ma in modo approssimativo).

Per implementare LRU, ho bisogno di tener traccia di uno storico degli accessi dei dati, per definire qual'è quello meno utilizzato.

Sostituzione dei blocchi

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



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

N = Numero di "posti" in un set della cache.

Ipotizzando di avere cache piena, in caso di cache miss, la macchina ha bisogno di tenere traccia del blocco LRU, per poterlo sostituire.

Al crescere di N, diventa più oneroso identificare il blocco LRU.

Inoltre, ad ogni richiesta di lettura/scrittura della cache, il blocco LRU potrebbe cambiare, quindi va aggiornato costantemente.

Se volessi associare un ID a ogni blocco, ipotizzando di avere 16 blocchi, mi servirebbero 4 bit ($\log(2, N)$), quindi un ID a 4 bit per ogni blocco nel set, in totale 4*16 bit.

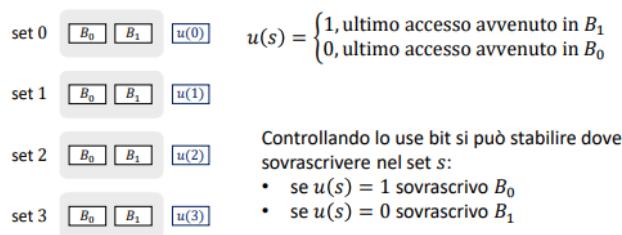
Decido quindi di tenere traccia di un solo ID, quello del blocco più frequentemente usato. (Approccio valido solo su cache con N=2)

Implementazione di LRU (N=2, 2 posti per ogni set)

Memorizzo per ogni set, l'indice del blocco usato durante l'ultima operazione di accesso alla cache. (Indice a 1 bit, per indicizzare due blocchi in un set a 2 blocchi basta un bit).

Se si dovesse verificare una cache miss, controllerò lo "use bit", e sovrascriverò l'altro blocco (che sarà l'LRU).

- Questa operazione è semplice da implementare con bassa associatività (2 o 4), ma diventa significativamente più complicato quando il numero di vie aumenta
- Approccio basato sugli **use bits**
- **Esempio** con $n = 2$, ogni set s ha un singolo use bit $u(s)$

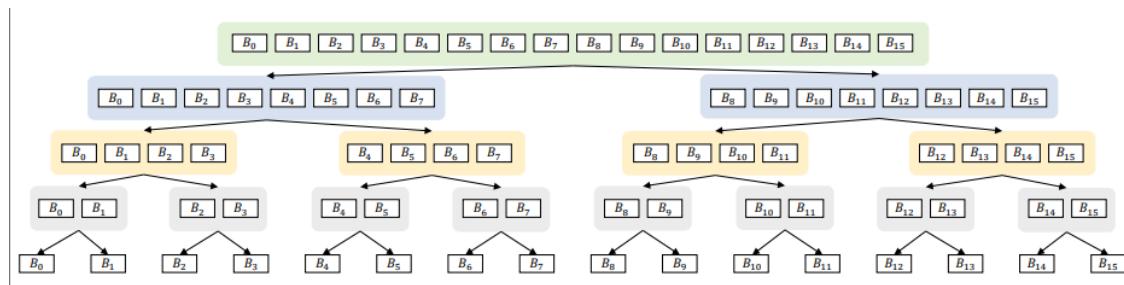


- Questo caso, con $n = 2$, è molto semplice perché, in ogni set, l'ultimo accesso implicitamente localizza la prossima eventuale sovrascrittura
- Conoscere il blocco acceduto più recentemente equivale a conoscere quello non acceduto da più tempo (l'altro)
- Se ci sono più di 2 blocchi in un set le cose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

Questo approccio, diventa molto complesso da implementare all'aumentare di N , pertanto scelgo di implementare una soluzione "approssimativa".

Pseudo-LRU (BST)

Si basa su una struttura dati ad albero binario di ricerca (BST).



Per ogni nodo foglia, esiste un percorso unico che collega la radice dell'albero ad esso.

Esempio: Il percorso che collega la radice al nodo (blocco) B6, è LRRL (Left - Right).

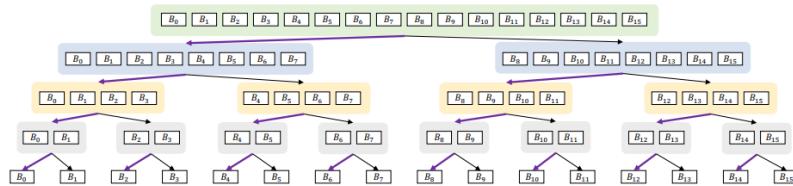
Il Concetto di Left e Right, segue una logica binaria, che posso codificare con 0 e 1 (0 - Left, 1 - Right).

Esempio: B6 → LRRL → 0110.

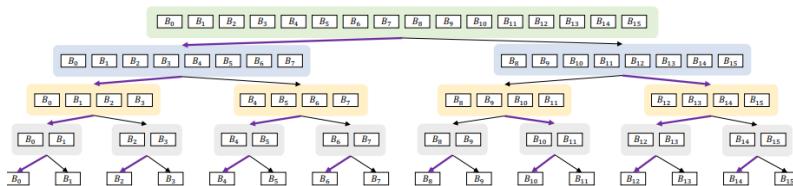
Ogni blocco, necessita ancora di $\log(2, N)$ bit per essere identificato.

Immagino di associare ad ogni biforcazione, un bit che indica la direzione OPPOSTA in cui si trova il blocco più recentemente acceduto.

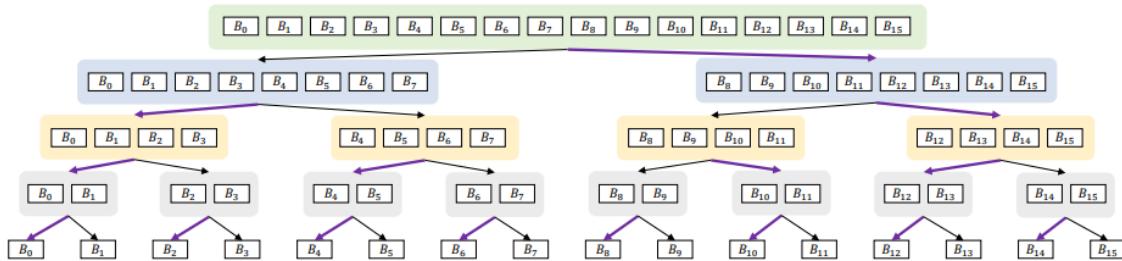
Esempio:



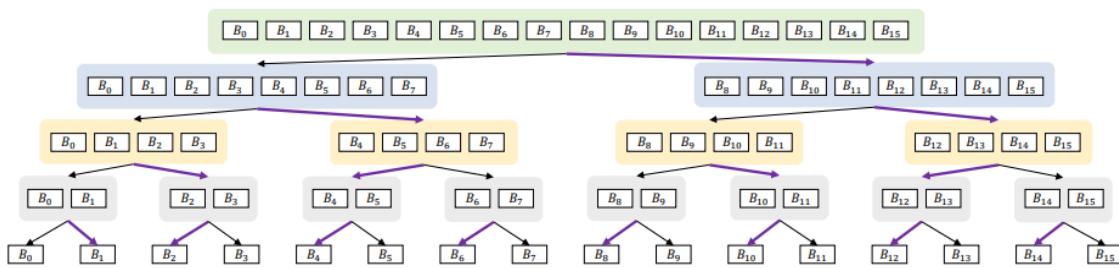
Arriva richiesta di lettura del blocco B9, noto che le frecce dei nodi che ho attraversato per arrivare a B9, si invertono!



Arriva richiesta di lettura del blocco B7:



Arriva richiesta di lettura del blocco B0:



Ora in caso di cache miss, devo sovrascrivere un blocco, chi sovrascrivo? Seguo le frecce settate dalle richieste precedenti, e arrivo a B12.

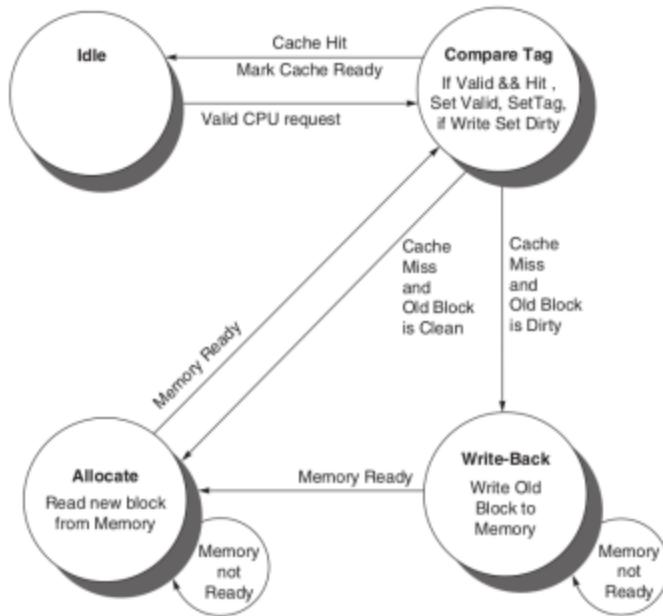


Anche dopo la sovrascrittura di B12, inverto i bit associati a ogni diramazione! Altrimenti in caso di un'altra cache miss, andrei di nuovo a sovrascrivere B12!

Gestione della cache:

Per l'implementazione della logica di gestione della cache, possiamo immaginarla come una FSM, con i seguenti stati:

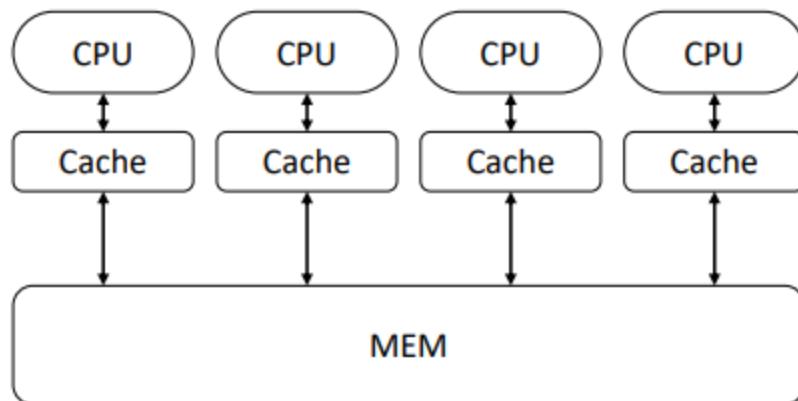
- Idle
- Hit/Miss
- Scrittura sulla cache (allocate, trasferimento di un blocco di memoria nella cache)
- Scrittura sulla memoria (write back, operazione con cui si mantiene la coerenza tra il dato modificato nella cache, e la memoria centrale)



Sistemi multi-processore (Multi-Core)

Più CPU (core), che lavorano in parallelo, ognuno con la propria cache.

La memoria centrale è però solo una: ciò vuol dire che quanto un core effettua una scrittura sulla propria cache, essa si "disallinea" dalla memoria centrale, ma se quello stesso dato è scritto anche nelle cache degli altri cores, si verifica un disallineamento anche rispetto alle altre caches.



Soluzioni per risolvere disallineamento multiplo:

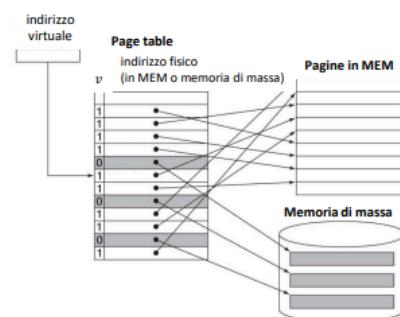
- Snooping: Ogni cache monitora il bus indirizzi: Al momento del write-back (trasferimento di un dato dalla cache alla memoria), le cache che contengono una copia del dato sovrascritto, lo invalideranno (write invalidate, cambiando il valore del bit di validità).
- Hardware transparency: Viene implementato un circuito che aggiorna simultaneamente lo stesso dato in tutte le caches.
- Non cacheable memory: Si definisce una regione della memoria come non allocabile in cache (identifico l'area di memoria che andrebbe copiata in più caches, e la marco come "non copiabile", così che la CPU interagisca direttamente con la memoria, risolvendo l'allineamento, ma riducendo le prestazioni).

Interazioni con la memoria di massa

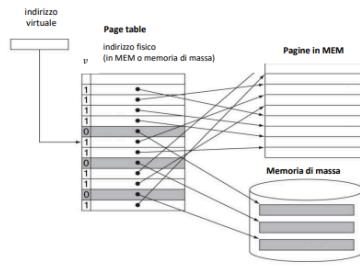
Ad un livello inferiore rispetto alla memoria centrale, abbiamo la memoria di massa.

E' importante che ogni programma in esecuzione, possa riferirsi alla memoria di massa in termini "logici", e non fisici, cioè si riferisce a degli indirizzi logici diversi da quelli mappati fisicamente sul disco.

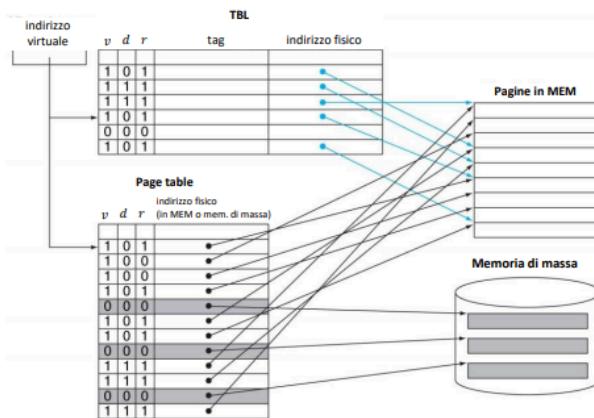
- **Principi di funzionamento**
 - La memoria fisica è suddivisa in pagine (i blocchi della cache)
 - Ogni programma lavora su uno spazio di indirizzamento virtuale, **che non deve tenere conto della presenza di altri programmi**
 - La memoria principale lavora come una cache, dove al posto del blocco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



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



- Soluzione:** Translation Look-aside buffer
 - Una cache per gli indirizzi delle pagine
- La memoria riceve una richiesta di accesso, tramite un indirizzo virtuale
 - Viene controllato TBL
 - hit**: si accede al corrispondente indirizzo fisico in MEM e il dato è recuperato
 - miss**: bisogna trasferire il dato in MEM dalla memoria di massa, **swap e aggiornare TBL**



Ogni processo, non può accedere direttamente ad una periferica: ha sempre bisogno di chiedere l'accesso ad un supervisore dell'elaborazione, che nella maggior parte dei casi è il sistema operativo.

Un processo accede quindi sempre ad uno spazio di memoria "virtuale" assegnatogli dal sistema operativo, che permette al programmatore di utilizzare la memoria come fosse un array contiguo, anche se in realtà i diversi dati del processo in utilizzo, sono scritti in diverse aree della memoria centrale.

Inoltre, dal punto di vista del processo, esso accede ad uno spazio di memoria riservato, "tutto suo", non condiviso con altri processi, e "infinito", cioè il sistema operativo nasconde le operazioni di swap dalla memoria centrale, per il processo, tutti i dati di cui ha bisogno sono già nella memoria centrale.

Integrità del dato.

Fin'ora abbiamo assunto che un dato presente in memoria, sia valido, e privo di qualsiasi alterazione (fisica oppure logica).

Esempio: alcuni bit di una parola di memoria cambiano il proprio valore a causa di un malfunzionamento dell'hardware.

Questo viene garantito, tramite ridondanza e introduzione di codice di error detection ed error correction, cioè aggiungiamo al dato un'ulteriore informazione, che permette di accorgersi se il dato è stato corrotto (ed eventualmente, di correggerlo).

Bit di parità

Aggiungo ad una stringa binaria di dimensione variabile, un ulteriore bit, ottenuto sommando tutte le cifre della stringa, come fossero scritte in decimale (Esempio: da $1011101 \rightarrow 5$), e interpretando questo ulteriore bit come 1 se la somma delle cifre è dispari, 0 se pari.

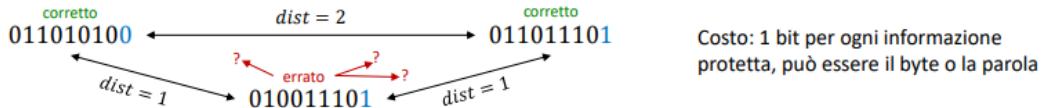
Questo bit, mi dice se il numero di 1 nella stringa di partenza è pari o dispari.

Tramite l'aggiunta di questo bit, garantisco che la stringa formata da stringa di partenza + bit di parità, abbia sempre numero di bit a 1 pari.

Se il numero di bit a 1 nella stringa finale è dispari, si è verificato un errore di trasmissione.



Rileva solo un numero dispari di errori! Esempio, se si sono alterati 2 bit nella mia stringa, essa risulterà ancora valida, perchè il numero di bit a 1 sarà ancora pari.



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

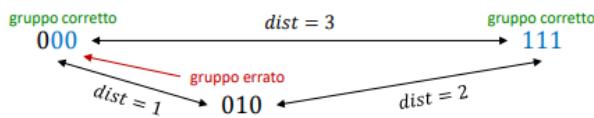
Codice a ripetizione

Ogni bit d'informazione, viene replicato ALTRE N volte.

Ad es. se $n = 2, d = 01101010$ diventa

000 111 111 000 111 000 111 000

In questo codice la distanza di Hamming minima tra due elementi del codice è pari a $n + 1$, nell'esempio 3.



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

In ogni gruppo, posso identificare N errori, correggerne al massimo $N/2$ (parte intera)

Esempio: se il gruppo originario è 000, il gruppo 001 è errato, 011 è errato, ma 111 risulterà corretto (perchè tutti e 3 i valori sono uguali).

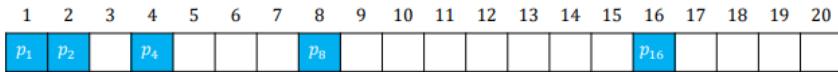
Hamming code:

E' un codice a distanza 3, in grado di rilevare e correggere errori singoli.

Su una stringa a N bit, alcuni sono utilizzati per la rappresentazione del dato, altri sono di parità.

I Bit di parità, saranno quelli di indice (indice va da 1 a N, le posizioni invece vanno da 0 a N) è una potenza di due.

- Esempio con $n = 20$



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

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

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

In altre parole, i bit di parità sono quelli il cui indice, espresso in binario, contiene una sola cifra a uno.

Regola di copertura:

Ogni bit di parità, calcola la parità su uno specifico gruppo di altri bit.

Un bit con indice h è protetto dal bit di parità p_i se e solo se h in binario ha un 1 alla posizione $\log_2 i$



Il Bit di parità indice X, protegge tutti i bit i cui indici (in binario) hanno un 1 in posizione log in base 2 di X

Operativamente:

- bit di parità p_1 , 1 in binario è 00001, protegge tutti gli indici che, in binario, hanno 1 nel bit in posizione 0:
00001, 00011, 000101, ...
- bit di parità p_2 , 2 in binario è 00010, protegge tutti gli indici che, in binario, hanno 1 nel bit in posizione 1:
00010, 00011, 00110, ...
- bit di parità p_4 , 4 in binario è 00100, protegge tutti gli indici che, in binario, hanno 1 nel bit in posizione 2:
00100, 00101, 00110, ...
- bit di parità p_8 , 8 in binario è 01000, protegge tutti gli indici che, in binario, hanno 1 nel bit in posizione 3:
01000, 01001, 01010, ...
- bit di parità p_{16} , 16 in binario è 10000, protegge tutti gli indici che, in binario, hanno 1 nel bit in posizione 4: 10000, 10001, 10010, ...

Il Valore dei bit di parità, è calcolato come già visto, ma solo sul ristretto gruppo di bit che esso copre. (Sommo il valore dei bit coperti come fossero interpretati in decimale: il bit di parità vale 1 se tale somma è dispari, 0 se pari).

Esempio

- $d = 11100101$
- servono in totale 12 bit, 4 bit di parità e 8 bit per il dato
- p_1 parità su $d_1 + d_2 + d_4 + d_5 + d_7 = 2$, quindi $p_1 \leftarrow 0$
- p_2 parità su $d_1 + d_3 + d_4 + d_6 + d_7 = 3$, quindi $p_2 \leftarrow 1$
- p_4 parità su $d_2 + d_3 + d_4 + d_8 = 3$, quindi $p_4 \leftarrow 1$
- p_8 parità su $d_5 + d_6 + d_7 + d_8 = 2$, quindi $p_8 \leftarrow 0$
- codifica: 011111000101

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

	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}																				

Dalla tabella, possiamo chiederci "quali bit di parità proteggono d9? Risposta: p1 e p8".

Notiamo che ogni bit di parità, è protetto da una combinazione univoca di bit di parità.

Con 5 bit di parità (scritti in verticale), posso "proteggere" fino a 32 bit di dato.

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

Esercizi:

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

Bit aggiuntivo di parità:

Bit aggiuntivo di parità

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