


I multi processori

Prof. Alberto Borghese
Dipartimento di Scienze dell'Informazione
borghese@dsi.unimi.it

Università degli Studi di Milano

Patterson, sezione 1.5, 1.6, 2.17, 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.9, 7.10


Sommario

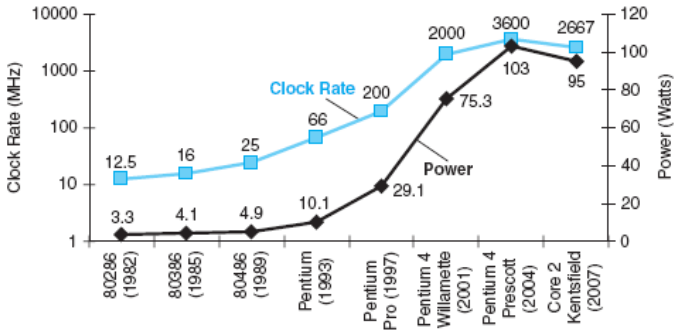
Le architetture multi-processore

Le gerarchie di memoria

Le problematiche

- Parallelizzazione del codice
- Gerarchia di memoria

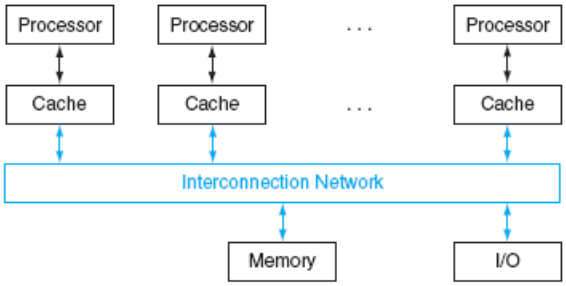
La barriera dell'energia



Power = Capacitive load × Voltage^2 × Frequency switched

E' aumentato di 30 volte — carico delle porte CMOS — È passata da 5V ad 1V — È aumentata di 200 volte in 20 anni

I multi-processori

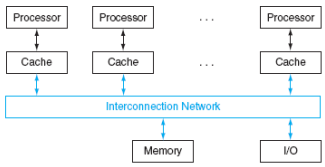


Chiama un parallelismo esplicito (la pipe-line multi-scalare è una forma di parallelismo implicito)

Un programma deve essere:
•Corretto
•Risolvere un problema importante (di grandi dimensioni)
•**Veloce** (altrimenti è inutile parallelizzare)

Esecuzione parallela

Esecuzione parallela richiede un Overhead:
- Scheduling.
- Coordinamento.



Scheduling:
•Analisi del carico di lavoro globale (scheduler).
•Partizionamento del carico sui diversi processori (scheduler).
•Coordinamento nella raccolta dei risultati (e.g. reorder station).

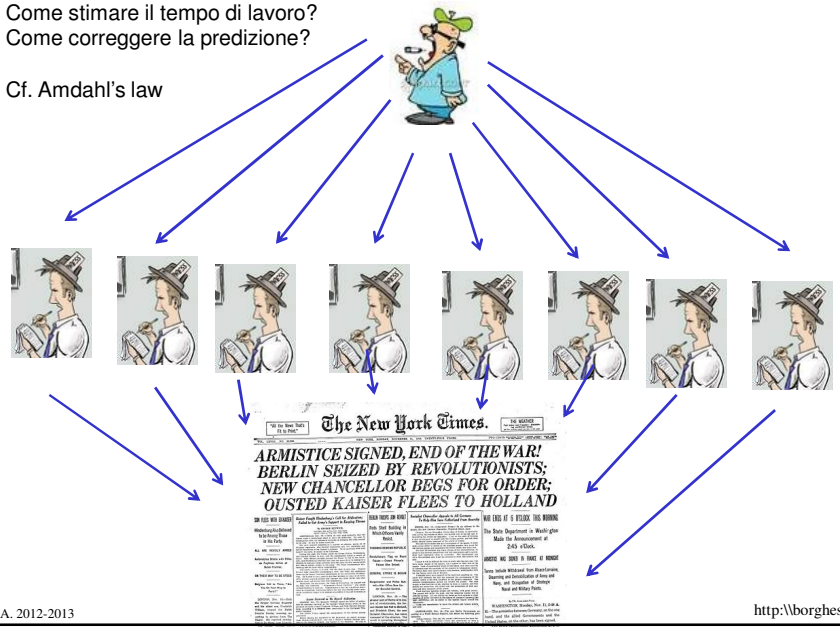
Lo scheduling può essere più (Pentium 4) o meno (CUDA) performante. Sempre più importante è il lavoro del programmatore / compilatore.

Infatti. Non si può “perdere” troppo tempo nello scheduling e/o nella compilazione.

Un esempio

Come stimare il tempo di lavoro?
Come correggere la predizione?

Cf. Amdahl's law



Esecuzione parallela il quadro generale

		Software	
		Sequential	Concurrent
Hardware	Serial (pipeline)	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel (pipeline)	Matrix Multiply written in MATLAB running on an Intel Xeon e5345 (Clovertown)	Windows Vista Operating System running on an Intel Xeon e5345 (Clovertown)

Alcuni task sono naturalmente paralleli (programmazione concorrente)

Altri task sono seriali e occorre capire come parallelizzarli in modo efficiente (moltiplicazione tra matrici)
Non è neppure facile parallelizzare task concorrenti in modo tale che le prestazioni aumentino con l'aumentare dei core

Esecuzione parallela e codice

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345 (Clovertown)

SIMD (Single Instruction Multiple Data). Array Processor o calcolatori vettoriali. Istruzioni vettoriali: stessa istruzioni su più dati (prodotti di tutti gli elementi di un vettore per uno scalare). Funziona bene nella gestione dei vettori e matrici e con i cicli for. Funziona male quando ci sono branch (switch).

Parallelizzazione dell'esecuzione (multiple-issue) sono architetture MIMD.

Parallelizzazione dell'esecuzione - I

- Somma di 100,000 elementi di un vettore (N=100,000) su un'architettura seriale

```
/* Execute sequentially - 100.000 steps */
sum = 0;
for (i = 0; i < 100000; i++)
        sum = sum + A[i]; /* sum the assigned areas*/
```

- Identifichiamo P lotti (batch) che possono essere elaborati in parallelo (non hanno dipendenze)

```
/* Execute sequentially - 100.000 steps = M * P = 1000 * 100  */
for (k=0; k < 99; k++) // for each of the P batches
{    sum[k] = 0;
     for (i = k*1000;i<(k+1)*1000;i=i+1) // for each of the M values
                                         // inside one batch
     {   sum[k] = sum[k] + A[i];         // sum the assigned areas
     }
}
```

Il numero di passi di esecuzione non cambia, ma possiamo parallelizzare l'esecuzione

Parallelizzazione dell'esecuzione: divide - II

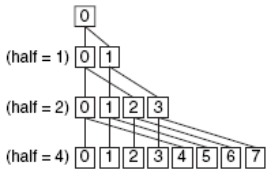
- Somma di 100,000 numeri (N=100,000) su un'architettura 100-core (P=100).
- Sommo N/P (=1,000) numeri su ciascun processore
  - Partizionamento dei dati in ingresso
  - Stessa memoria fisica. L'accesso dei diversi processori è su blocchi diversi di memoria fisica.

```
/* Execute in parallel on each Pn processor */
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i = i + 1)
       sum[Pn] = sum[Pn] + A[i]; /* sum the assigned areas*/
```

- Posso eseguire le somme parziali in 1000 passi invece che in 1000 * 100 = 100,000 passi: il problema scala con il numero dei processori.
- Ottengo P = 100 somme parziali. Per ottenere la somma finale devo sommare tra loro le somme parziali (**riduzione**). Come?

Parallelizzazione dell'esecuzione: reduction - III

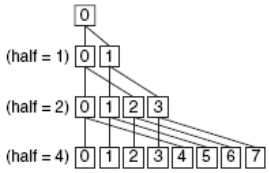
- Sommo i numeri a due a due in modo ricorsivo e gerarchico (**divide and conquer**)




```
half = 128; /* 128 processors, Pn, in multiprocessor*/
repeat
   synch(); /* wait for partial sum completion */
            /* Conditional sum needed when half is even */
   half = half/2; /* dividing line on who sums */
   if (Pn < half)
       sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1); /* exit with final sum in Sum[0] */
```

Osservazione

- **Quanto si guadagna?**

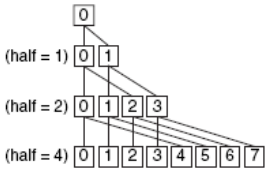


- La riduzione sequenziale costa M passi, dove M è il numero di somme parziali.
- La riduzione parallela costa log2(M) nel caso in cui ad ogni processore possa essere assegnato una somma parziale.

- Per M = 128 abbiamo:
  - 128 passi per la somma sequenziale
  - 7 passi per la riduzione parallela.

Parallelizzazione dell'esecuzione – reduction - IV

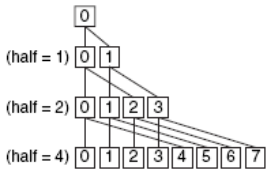
- Sommo i numeri a due a due in modo ricorsivo e gerarchico (**divide and conquer**)




```
half = 128; /* 100 processors, Pn, in multiprocessor*/
repeat
   synch(); /* wait for partial sum completion */
if (half%2 != 0 && Pn == 0)
       /* Test che deve essere eseguito quando half è
       /* dispari: Processor0 gets missing element */
       sum[0] = sum[0] + sum[half-1];
   half = half/2; /* dividing line on who sums */
   if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1); /* exit with final sum in Sum[0] */
```

Parallelizzazione dell'esecuzione - reduction

- Sommo i numeri a due a due in modo ricorsivo e gerarchico (**divide and conquer**)




```
half = 128; /* 128 processors, Pn, in multiprocessor*/
repeat
   synch(); /* wait for partial sum completion */
       /* Conditional sum needed when half is even */
       /* Processor0 gets missing element */
   if (half%2 != 0 && Pn == 0)
       sum[0] = sum[0] + sum[half-1];
   half = half/2; /* dividing line on who sums */
   if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1); /* exit with final sum in Sum[0] */
```


Osservazioni

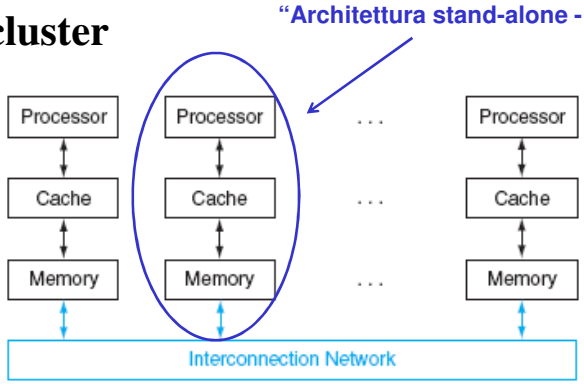
`half = 100;` consente di scalare con il numero di processori

`if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];`
Assegna ai diversi processori il ruolo (accumulatore o semplice memoria)

`synch(); /* wait for partial sum completion */`
Sincronizzazione esplicita alla fine di ogni livello di somme parziali

Distribuzione e sincronizzazione sono problematiche già viste nelle pipe-line superscalari dove venivano risolte dall'HW e/o dal compilatore. Qui distribuzione e sincronizzazione vengono eseguiti a livello di codice. Potrebbero essere eseguiti dai compilatori. Non sono ancora così "smart" per sfruttare appieno il parallelismo ...

I cluster



Synchronization through **message passing** multiprocessors (sender – receiver).

Modalità tipica delle architetture SW concorrenti (Robot: AIBO Sony, File servers)
Ogni architettura ha la sua memoria, il suo SO. La rete di interconnessione non può essere così veloce come quella dei multi-processori.

E' un'architettura molto più robusta ai guasti, facile da espandere.

Massive parallelism -> data center -> Grid computing (SETI@home, 257 TeraFLOPS-> Cloud computing.

Sommario

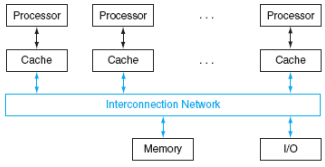
Le architetture multi-processore

Le gerarchie di memoria

Esecuzione parallela con una memoria condivisa

La memoria viene suddivisa in memoria condivisibili e memoria non condivisibile o privata del singolo processo. In questo secondo caso la parte corrispondente della memoria principale non può essere utilizzata da altri processi.

- Competizione sui dati (data race).
- Occorre una coordinazione tra i diversi processi nell'accesso alla memoria (sincronizzazione) => **cache coherence**.
- Viene introdotta un'operazione atomica di scambio dei dati tra un registro ed una cella di memoria: nessun altro processsore o processo può inserirsi fino a quando l'operazione atomica non è terminata.
- Viene inserito un meccanismo hardware di blocco di una cella di memoria (lock o lucchetto).
- Viene gestito dal sotto-sistema di controllo della memoria dietro istruzioni della CPU.



Come funziona la scrittura in un singolo core?

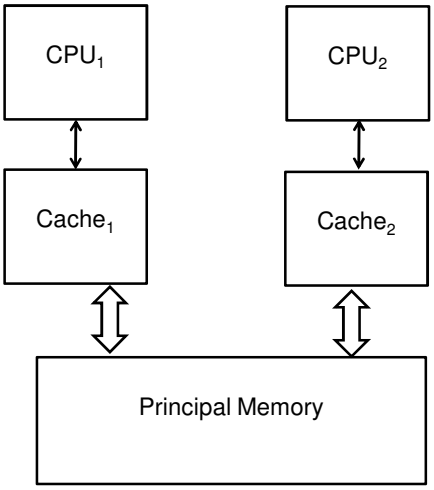
Quando c'è register spilling il dato viene scritto in cache e si verifica un **disallineamento** della memoria principale e della cache.

Write-through. Scrittura in cache e contemporaneamente in RAM.
Write_buffer per liberare la CPU (DEC 3100)
Chi libera il write buffer?
Cosa succede se il write buffer è pieno?
Sincronizzazione tra contenuto della Memoria Principale (che può essere letto anche da I/O e da altri processori) e Cache.
Svantaggio: traffico intenso sul bus per trasferimenti di dati in memoria.

Write-back. Scrittura ritardata. Scrivo quando devo scaricare il blocco di cache.
Utilizzo un bit di flag: UPDATE, che viene settato quando altero il contenuto del blocco. Questo flag si chiama anche "dirty bit".
Vantaggiosa con cache n-associative.
Alla Memoria Principale trasferisco il blocco quando devo scrivere da CPU a cache (è equivalente al register spilling).

Contenuto della memoria principale e della cache può non essere allineato.

Cache in un'architettura dual-core



Le cache si parlano attraverso la memoria principale

Più cache ed un'unica memoria principale: **"the view of memory held by two different processors is through their individual caches"**

Coerenza e consistenza

Coerenza: determina se il dato letto dalla cache è lo stesso di quello contenuto nella memoria principale.
Consistenza: quando un dato può essere letto dopo una scrittura?

Esempio su una memoria Write-through

Time step	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A stores 1 into X	1	0	1

Cosa succederebbe se la cache fosse "Write-back"?

Consistenza: Need for a serialization of the writes (cf. Commit unit in the CPU)

Bus snooping

Mantenimento dell'informazione di cache coerente tra varie cache (sistemi multi-processori).

Elemento chiave è il protocollo per il **tracking dello stato** di ciascuna linea di ciascuna cache.

In cache, oltre al TAG e al bit di validità viene memorizzato lo stato della linea.

Ogni trasferimento dalla Memoria Principale viene monitorato da tutte le cache.

Il controller della cache monitora il bus indirizzi + segnale di controllo read della memoria e legge l'indirizzo della memoria principale delle richieste di tutte le altre cache.

Se l'indirizzo corrisponde all'indirizzo dei dati contenuti in una delle linee della cache, viene invalidato il contenuto della linea.

Quando funziona?

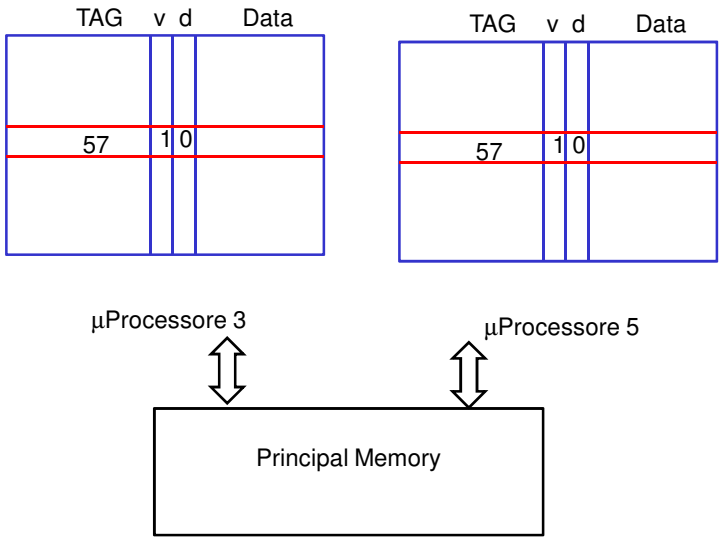
Write invalidate protocol

In questo caso si mira a garantire a ciascuna cache la piena proprietà di un dato. *"Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: all other cached copies of the item are invalidated"*.

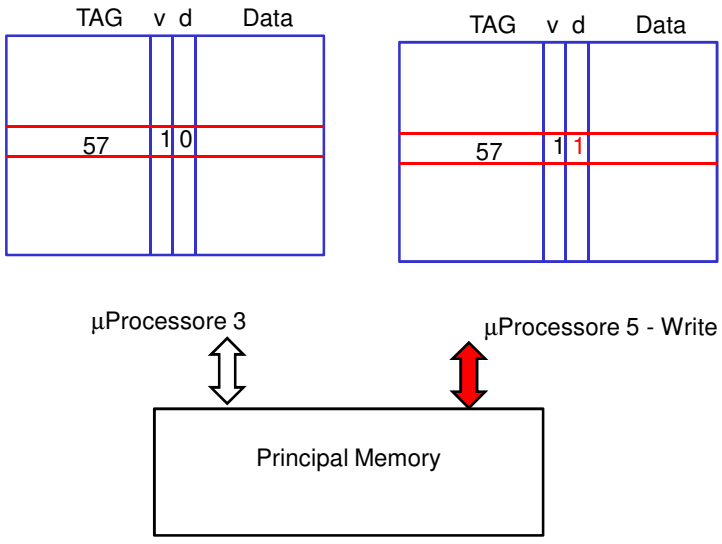
Il controller della memoria monitora il bus indirizzi (**snooping**). Quando si verifica una write, cerca se il dato nell'indirizzo di scrittura è presente nelle altre cache.
Invalida il blocco nelle cache in cui il dato sra stato copiato.
Non c'è un'informazione di "blocco aggiornato" centralizzata, ma è distribuita sulle varie cache.

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

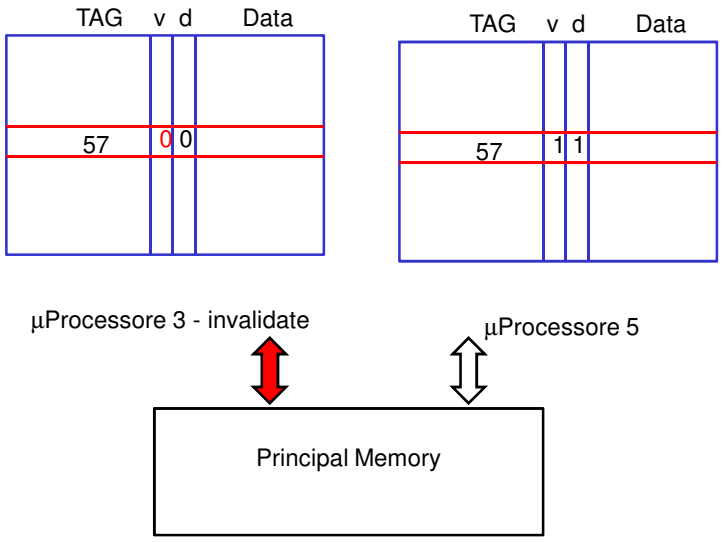
Come agisce il sottosistema di memoria - I



Come agisce il sottosistema di memoria - II



Come agisce il sottosistema di memoria - III



Protezione della memoria

Come garantire la piena proprietà di una linea di memoria?

Data race sulla memoria principale ➔ **Lock** di una cella di memoria e unlock.

L'accesso ad una cella di memoria viene riservato ad una certa procedura (e ad un certo processore)

```
try:    add $t0,$zero,$s4 ;copy exchange value
        ll $t1,0($s1) ;load linked
        sc $t0,0($s1) ;store conditional
        beq $t0,$zero,try ;branch store fails
        add $s4,$zero,$t1 ;put load value in $s4
```

Si controlla che la load abbia avuto successo controllando il contenuto di $t0 (sc). Se la procedura non è riuscita a leggere perchè c'era un blocco sulla cella di memoria, riprova.

La condivisione della memoria

Istruzioni corrispondenti al meccanismo hardware del **lock**:

- *load linked* (load collegata, `ll`) + *store conditional* (store condizionata, `sc`) che vengono utilizzate in sequenza.
- Se il contenuto di una locazione di memoria letta dalla load linked venisse alterato prima che la store condizionata abbia salvato in quella cella di memoria il dato, l'istruzione di store condizionata fallisce.
- L'istruzione di store condizionata ha quindi due funzioni: salva il contenuto di un registro in memoria *ed* imposta il contenuto di quel registro a 1 se la scrittura ha avuto successo e a 0 se invece è fallita.
- L'istruzione di load linked restituisce il valore letto e l'istruzione di store condizionata restituisce 1 solamente se la scrittura ha avuto successo.
- Controllo del contenuto del registro target associato all'istruzione di store condizionata.

Altri meccanismi per la cache coherence

Hardware trasparency.
Circuito addizionale attivato ad ogni scrittura della Memoria Principale.
Copia la parola aggiornata in tutte le cache che contengono quella parola.

Noncachable memory.
Viene definita un'area di memoria condivisa, che non deve passare per la cache.

NB Blocchi di grandi dimensioni possono provocare il **false sharing**. Due programmi che stanno girando su due CPU diverse richiedono due variabili diverse ma che ricadono nello stesso blocco della cache (a mappatura diretta). Alla cache il blocco appare condiviso e si innesca il meccanismo di invalidazione.

Soluzione: allocare le due variabili in memoria principale in modo tale che cadano in due blocchi diversi. I compilatori (ed i programmatori) sono incaricati di risolvere questo problema.

ECC

- Errori dovuti a malfunzionamenti HW o SW.
  - Date le dimensioni delle memorie (**10^{10} celle**) la probabilità d'errore non è più trascurabile.
  - Per applicazioni sensibili, è di fondamentale importanza gestirli.
- Codici rivelatori d'errore
  - Es: codice di parità.
  - Consente di individuare errori singoli in una parola.
  - Non consente di individuare su quale bit si e' verificato l'errore.
- Codici correttori d'errore (error-correcting codes – ECC)
  - Consentono anche la correzione degli errori.
  - Richiedono più bit per ogni dato (più ridondanza)
    - Per la correzione di 1 errore per parole e l'individuazione di 2 errori, occorrono 8bit /128 bit.

Codice rilevatore di errore: il codice di parità

- **Es: Bit di parità (even):**
  - aggiungo un bit ad una sequenza in modo da avere un n. pari (*even*) di "1"
    - `0000 1010 0` ← `bit di parità`
    - `0001 1010 1`
  - Un errore su uno dei bit porta ad un n. dispari di "1"
- Prestazioni del codice
  - mi accorgo dell'errore, ma non so dov'è
  - **rivelo ma non correggo errori singoli**
  - **COSTO: 1 bit aggiuntivo ogni 8 → 9/8 = +12,5%**

Codici correttori d'errore

- **Es: Codice a ripetizione**
  - Ripeto ogni singolo bit della sequenza originale per altre 2 volte → triplico ogni bit

    `0 00 1 11 1 11 0 00 1 11 0 00 0 00 1 11 ...`

  - Un errore su un bit di ciascuna terna può essere corretto:

    `000 → 010 → 000`

    `111 → 110 → 111`

- Prestazioni del codice
  - **rivelo e correggo errori singoli**
  - **COSTO: 2 bit aggiuntivi ogni 1 → 3/1 = +200%**

Definizioni

- **Distanza di Hamming**, **d** (tra 2 sequenze di N bit)
  - il numero di cifre differenti, giustapponendole

    `01001000`

    `01000010 → d = 2`

- **Distanza minima** di un codice, d_{MIN}
  - il valor minimo di **d** tra tutte le coppie di sequenze di un codice

- Capacità di rivelazione di un codice: $t = d_{MIN} - 1$
- Capacità di correzione di un codice: $r = (d_{MIN}-1)/2$

- Esempi:
  - Codice a bit di **parità**: $d_{MIN} = 2 \rightarrow$ t=1, r=0
  - Codice a **ripetizione (3,1)**: $d_{MIN} = 3 \rightarrow$ t=2, r=1

Applicazioni nelle memorie

- RAM con controllo di parità
  - Aggiungo un bit di parità ad ogni byte
  - Es: RAM 1 M x 9 bit (8+1)
- RAM con codice correttore di errori (ECC)
  - si usa nelle memorie cache
  - codici ECC evoluti (alta efficienza)
    - Hamming, CCITT-32, Reed Solomon, …

Dimensione di codici ECC

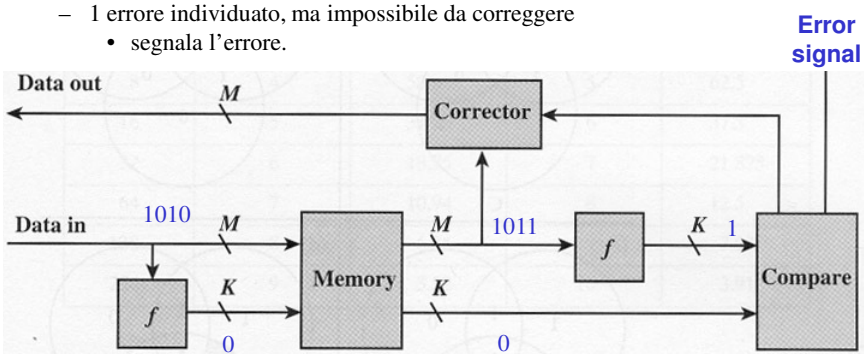
- Conviene applicare ECC a parole più lunghe possibile → aggiungo meno ridondanza → maggiore efficienza del codice
  - A costo di complessità maggiori di codifica/decodifica

	Single-Error Correction		Single-Error Correction/ Double-Error Detection	
Data Bits	Check Bits	% Increase	Check Bits	% Increase
8	4	50	5	62.5
16	5	31.25	6	37.5
32	6	18.75	7	21.875
64	7	10.94	8	12.5
128	8	6.25	9	7.03
256	9	3.52	10	3.91

E' conveniente avere parole di memoria lunghe.

Correzione degli Errori

- **OUT** possibili:
  - No errors detected
    - I dati letti possono essere inviati in uscita così come sono.
  - 1 errore è stato individuato e corretto
    - I bit del dato, più il codice associato vengono inviati al correttore, il quale provvede a correggere il dato.
  - 1 errore individuato, ma impossibile da correggere
    - segnala l'errore.



f può essere ad esempio il codice di parità

Sommario

Le architetture multi-processore

La parallelizzazione del codice