

High Performance and Quantum Computing

Anno Accademico
2025/26

**Rocco Lo Russo
Agostino D'Amora**

Contents

1 Richiami APC	5
1.1 Richiami Pipelining	5
1.2 Pipeline Hazards	6
1.2.1 Hazards sui dati	7
1.2.2 Hazards di controllo	7
1.3 Problemi nelle architetture superscalari	11
1.4 Memoria	11
1.4.1 Problema del piazzamento di un blocco	12
1.4.2 Problema della ricerca di un blocco	14
1.4.3 Problema della sostituzione di un blocco	17
1.4.4 Problema della strategia di scrittura	18
2 Architetture superscalari dei moderni processori	20
2.1 Scheduling statico	21
2.2 Scheduling dinamico	22
2.2.1 Forward paths	22
2.2.2 Esecuzione out-of-order	23
2.2.3 Scoreboard	25
2.3 L'algoritmo di Tomasulo	27
2.4 Dipendenze di controllo	29
2.4.1 Generalizzazione predittore a 2 bit	30
2.4.2 Gestione delle eccezioni	32
2.5 Multiple issue	33
2.6 Hardware supported multithreading	35
2.6.1 Simultaneous multithreading	36
2.6.2 SMT performance	37
3 Coerenza e consistenza della memoria	39
3.1 Architettura cache	40
3.2 Protocolli di coerenza	41
3.2.1 Protocollo a due stati con Writeback	43
3.2.2 Protocollo a tre stati	44
3.2.3 Lo stato Exclusive	44
3.2.4 Lo stato Owned	46
3.2.5 Recap Stati di coerenza	46
3.2.6 Rimozione delle ipotesi	47
3.2.7 Directory	48

3.2.8 Protocollo Directory MSI	50
3.3 Primitive di sincronizzazione	51
3.4 Cache performance nei sistemi paralleli/multicore	53

Introduzione

In questo documento vengono raccolti appunti del corso *High Performance and Quantum Computing* tenuto nell'anno 2025-26 dal professor Cilardo presso l'università di Napoli Federico II, Dipartimento Ingegneria Elettrica e Tecnologie dell'Informazione.

È assolutamente scoraggiata la vendita di questi appunti, che possono in qualsiasi momento essere scaricati gratuitamente da github. Allo stesso repository è possibile in qualsiasi momento contruire seguendo le istruzioni esposte nel CONTRIBUTING.md.

Per la parte iniziale di ricapitolazione, sono stati utilizzati gli appunti raccolti durante il corso di Architettura e Progetto dei Calcolatori tenuto nell'anno 2024-25 dal professor Mazzocca presso l'università Federico II di Napoli (a cui è possibile accedere sia per consultare che per contribuire, nelle stesse modalità, su github), i libri *Computer Organization and Desing Patterns* (Hannessy-Patterson) e degli stessi autori *Computer Architecture (a quantitative approach)*.

Una buona parte degli esempi a cui si fa riferimento, e di cui per motivi di spazio non verrà riportato l'intero svolgimento, è tratta dai lucidi messi a disposizione dal professore sul canale teams del corso. Ove necessario, sarà fatto esplicito riferimento a quelli.

Chapter 1

Richiami APC

1.1 Richiami Pipelining

Il pipelining è una tecnica di implementazione di processori dove diverse istruzioni si sovrappongono durante l'esecuzione. Questa tecnica, chiave nella progettazione dei moderni processori, trae vantaggio dal parallelismo che esiste intrinsecamente tra le azione necessarie per l'esecuzione di un'istruzione. Nella pipeline, ogni passo necessario all'esecuzione di un'istruzione è svolto da un'unità funzionale autonoma, denominata *pipe segment* o *pipe stage*. Ogni unità è collegata alla successiva, e il throughput di una pipeline è determinato dal *tasso di attraversamento*, ovvero dalla frequenza con la quale un'istruzione esce dalla pipeline. Il tempo richiesto da un'istruzione per procedere allo stage successivo è denominato *processor cycle*, e dato che tutti gli stage devono necessariamente essere pronti a procedere allo stesso tempo, la lunghezza di un processor cycle è determinato dal tempo richiesto dallo stage più lento. In un computer il processor cycle è quasi sempre un colpo di clock.

Se il tempo di ogni stage è perfettamente bilanciato, chiamando T_i il tempo per ogni istruzione sul processore pipelined, T_{np} il tempo per ogni istruzione su un processore non pipelined ed n il numero di stages della pipe, allora vale la relazione: $T_i = \frac{T_{np}}{n}$.

Sotto queste condizioni ideali, lo speedup vale proprio n . Tuttavia, il tempo di ogni stage non è mai perfettamente bilanciato, e in più c'è da considerare il tempo di overhead introdotto dalla complessità del sistema. Una semplice pipeline a cinque stadi presenta i seguenti blocchi funzionali:

- **Instruction fetch:** Viene letto il PC per prelevare l'indirizzo della prossima istruzione; Si accede al registro Instruction Memory e si carica l'istruzione; Si calcola il prossimo valore del PC (+4).
- **Instruction Decode:** L'istruzione viene decodificata, ovvero l'unità di controllo capisce di che tipo di istruzione si tratta; si leggono i valori dei registri sorgente; si verifica l'estensione del segno.
- **Execute:** In questa fase avviene la vera elaborazione.
 - *istruzione aritmetico-logica* → l'ALU esegue l'operazione;
 - *istruzione load/store* → si calcola l'indirizzo effettivo (base + offset);
 - *istruzione branch* → si calcola la condizione e il nuovo PC;
- **Memory Access:** Operazioni di lettura/scrittura in memoria;

- **Write Back:** Scrittura nei registri del processore. Non tutte le istruzioni scrivono un registro.

Da questa presentazione emergono tre osservazioni: innanzitutto, è necessario separare memorie di istruzione e memorie dati, attraverso l'implementazione di diverse caches. Questa soluzione permette di evitare conflitti durante le fasi di IF e MEM; Il register file è usato in due stages, ovvero in ID e WB. Quindi, è necessario in un solo colpo di clock performare due letture e una scrittura, e questo si ottiene effettuando la scrittura nella prima parte di ciclo di clock e la lettura nella seconda parte; infine, è necessario considerare un circuito hardware che incrementi automaticamente il PC nella fase di IF, e un circuito che calcoli un eventuale indirizzo di branch durante lo stadio ID. La condizione del salto viene valutata in fase EXE, con una parte della ALU che confronta due registri.

Per fare in modo che l'output di uno stage non interferisca con lo stage successivo, vengono introdotti dei registri tra uno stage e l'altro, in modo che alla fine di un ciclo di clock, tutti i risultati di uno stage sono conservati in un registro che viene usato con input allo stage successivo al prossimo ciclo di clock. I registri svolgono un ruolo chiave anche nel trasporto di eventuali risultati intermedi dove stage di origine e destinazione non sono immediatamente adiacenti.

Il pipelining incrementa il throughput di istruzioni del processore, ovvero il numero di istruzioni completate per unità di tempo, ma non riduce il tempo di esecuzione di una singola istruzione. Al più infatti il tempo di esecuzione di ogni singola istruzione aumenta a causa dell'overhead introdotto nel controllo della pipeline. Questo fatto pone un limite importante sulla profondità pratica di una pipeline. In particolare, lo sbilanciamento tra i tempi di attraversamento di ciascuno stage implica una riduzione di performance in quanto il clock può essere soltanto veloce quanto il tempo richiesto dallo stage più lento; inoltre, l'overhead introdotto dalla pipeline è causato dalla combinazione di fattori quali il ritardo dei registri della pipeline e il *clock skew*. Il tempo di setup dei registri della pipeline è il tempo necessario affinché l'input che viene dato ad un registro si stabilizzi prima che il clock attivi una scrittura. Il clock skew è il massimo ritardo che segna la ricezione del colpo di clock tra qualsiasi coppia di registri.

1.2 Pipeline Hazards

Vi sono situazioni, denominate *hazards*, nelle quali viene impedito alla prossima istruzione nel flusso di istruzioni della pipeline di venire eseguita nel ciclo di clock designato.

Distinguiamo: hazards **strutturali**, che sorgono a causa di conflitto tra risorse hardware, ovvero quando l'hardware non riesce a supportare tutte le possibili combinazioni di istruzioni simultaneamente in esecuzioni sovrapposte; hazards **sui dati**, che sorgono quando un'istruzione dipende dal risultato di un'istruzione precedente; hazards **di controllo**, che sorgono in casi di istruzioni di branch e in generale di istruzioni che modificano il PC. Spesso, per risolvere un hazard è necessario fermare la pipeline, ovvero forzare una *stall*. Nella situazione in cui un'istruzione viene fermata, tutte le istruzioni successive a questa vengono fermate, mentre quelle precedenti continuano il loro flusso di attraversamento della pipe. Gli hazards strutturali sono poco frequenti in

quanto riguardano unità funzionali hardware (come l'unità di divisione a virgola mobile) non utilizzate frequentemente, perché sia i programmati che i compilatori sono a conoscenza di questo tipo di hazards. Ci concentreremo principalmente sui data hazards e sui control hazards.

1.2.1 Hazards sui dati

La sovrapposizione di istruzioni introdotta dalla pipeline implica un cambiamento dell'ordine di accessi in lettura/scrittura agli operandi rispetto al modello sequenziale di Von Neumann. Un primo passo risolutivo è già stato presentato: quando si cerca di leggere o scrivere da uno stesso registro contemporaneamente, molte architetture prevedono che la scrittura avvenga nella prima metà del ciclo di clock mentre la lettura nella seconda metà. Questo risolve l'hazard quando la scrittura del registro avviene durante lo stesso ciclo di clock della lettura.

In linea di massima, il dato prodotto dall'istruzione da cui dipendono le successive è già pronto durante la fase di execute, quindi basterebbe semplicemente propagare il dato appena è possibile alle unità a cui serve prima che sia disponibile per la lettura dal file register. Ci occuperemo per semplicità solo del caso di propagazione in una fase EX, che può essere la propagazione di un risultato di una operazione ALU o il calcolo di un indirizzo effettivo. Questo significa che quando un'istruzione prova ad usare un registro nella sua fase EX che un'istruzione precedente deve ancora scrivere in fase di WB, può servirsi del valore anticipatamente come input controllato alla ALU.

Un caso in cui il forwarding non può risolvere l'hazard è quando un'istruzione prova a leggere un registro immediatamente dopo un'istruzione di *load* dalla memoria. Il dato risulterebbe ancora in fase di recupero dalla memoria, quindi è necessaria un'unità funzionale che stalli la pipeline. In generale, il compilatore, che è a conoscenza della struttura del processore, in casi di hazard detection di questo tipo si preoccupa di anticipare delle istruzioni indipendenti utilizzando la proprietà commutativa e associativa. Lo stallo è inevitabile quando non è possibile inserire dopo l'istruzione di accesso in memoria un blocco di istruzioni indipendenti. È possibile tuttavia un approccio, denominato **internal forwarding**, che permette di ibernare temporaneamente le istruzioni bloccate, e non stallare la pipeline. Bloccando istruzioni e caricandone altre, tuttavia può accadere che più istruzioni cerchino di operare sullo stesso registro, quindi occorre utilizzare una tecnica di indirizzamento indiretto che permette ai registri fisici del processore (*forwarding registers*) di puntare a dei registri in memoria che contengono valori precedenti assunti dal registro. Le istruzioni vengono ibernate in un'apposita area della memoria, la *ibernation table*.

1.2.2 Hazards di controllo

Quando sopraggiunge un'istruzione di salto, riusciamo a captare che è così solo nella fase di esecuzione dell'istruzione, mentre la pipe ha continuato a caricare le istruzioni in modo sequenziale, che si troveranno rispettivamente nella fase di IF e ID e non avranno quindi ancora modificato lo stato del processore e della memoria. Nel caso in cui il salto non deve essere eseguito, allora la pipe continuerà a funzionare normalmente, mentre se il salto deve essere eseguito, le istruzioni caricate dovranno essere eliminate dalla pipe (*pipe flush*) e dovrà essere caricata l'istruzione a cui punta il salto e le successive.

Il ritardo che segue quest'ultimo caso è detto *branch penalty*. L'obiettivo è quello di confinare la gestione dei salti nelle fasi IF e ID, perchè in quelle fasi le istruzioni non hanno ancora modificato lo stato del processore e della memoria, e quindi la rete di controllo hardware deve riguardare solo queste due fasi. In generale, il problema è risolvibile fondamentalmente mediante due approcci: l'approccio conservativo e l'approccio ottimistico (*branch prediction*). Nel caso dell'**approccio conservativo**, quando il processore interpreta, durante la fase ID, un'istruzione come istruzione di salto, ferma la pipe e disabilita la propagazione dell'istruzione che si trova erroneamente nella fase IF, determina l'istruzione a cui saltare in fase EX e la preleva. Si torna insomma al modello sequenziale di Von Neumann. Il conto è salato se consideriamo che le istruzioni di salto costituiscono il 25% di un programma, e infatti ne consegue un notevole spreco delle risorse di parallelismo fornite dalla pipe. Questo approccio è *leggermente* migliorabile attraverso l'hardware, anticipando la decisione inerente al salto alla fase di ID, in base alla condizione di salto. Ad esempio l'istruzione `JNZ <LABEL>` controlla se il flag Z del SR è alto, e in tal caso non occorre saltare e quindi l'istruzione che si è prelevata nel frattempo è giusta. Molti ritardi possono essere evitati dal programmatore o dal compilatore: il programmatore può contribuire al buon funzionamento del sistema, scrivendo le istruzioni in un ordine tale da minimizzare le probabilità di stallo della pipe. Nel caso di un costrutto if-then-else, ad esempio, conviene inserire nel ramo then l'alternativa più probabile. Il compilatore, da parte sua, può operare vari accorgimenti; ad esempio, siccome un ciclo `for` è sempre tradotto in un if-then-else, esso deve inserire l'alternativa più probabile nel ramo then. In fase di compilazione è possibile evitare l'approccio conservativo. Se immediatamente prima del salto c'è un'istruzione con operandi da cui non dipende l'esito della scelta di saltare o meno, è possibile in fase di compilazione inserire l'istruzione indipendente dopo il branch, in modo da sfruttare lo slot di tempo in cui l'istruzione viene caricata ed entra nella pipe, e quindi non c'è bisogno di pipe flush. Qualora non sia possibile invertire una istruzione if con quella che la precede, il compilatore potrebbe decidere di mettere subito dopo la if una **nop**, istruzione che non ha alcun effetto e quindi non è mai errato inserirla nella pipe. In questo modo si può operare senza considerare alcun tipo di approccio. Possiamo aggiungere che se stiamo considerando un'istruzione di salto in un processore CISC con una condizione di salto elaborata, allora il numero di `nop` che bisogna inserire a seguito dell'istruzione di salto risulta essere superiore a 1, in quanto ricordiamo che la pipe per i CISC è più lunga e la valutazione della condizione coinvolge più fasi oltre le prime due, di caricamento e decodifica. Una soluzione meno conservativa a quella appena presentata è di utilizzare la **branch-prediction** → approccio ottimistico.

La branch prediction è una tecnica che cerca di prevedere a quale ramo un'istruzione di salto condizionato possa saltare. Consideriamo il classico esempio:

```

1 for (int i = 0; i < N; i++) {
2     for (int j = 0; j < N; j++) {
3         op...
4     }
5 }
```

Il controllo prevede l'esecuzione del ramo else una sola volta (guardando il `for` interno) ogni `N` passi. I modi per prevedere il branch possono essere vari, e possono essere descritti mediante degli appositi automi a stati finiti. Un primo approccio molto basilare

è quello di andare a cambiare il branch da caricare successivamente ad ogni errore di decisione e quindi eseguire le operazioni descritte dall'automa [1.1].

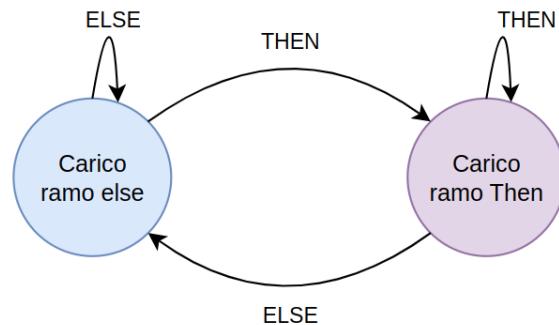


Figure 1.1: Automa della branch prediction base

Questa soluzione non è ottima poiché per quel singolo errore che avviene ad ogni N interazioni la pipeline dovrà assorbire 2 penalties. Per evitare tale condizione, e quindi rendere la persistenza più forte, consideriamo un automa a 4 stati che permette di rendere la condizione di "cambio del branch" più solida, poichè solo in caso di due errori successivi, allora cambia l'indirizzo di salto effettivo. L'automa a cui facciamo riferimento è [1.2].

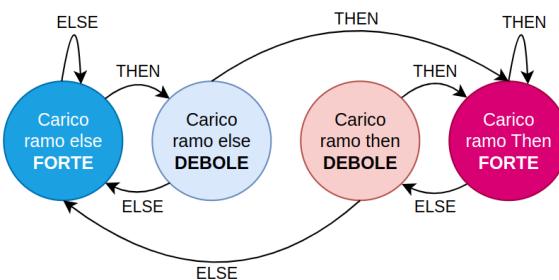


Figure 1.2: Automa della branch prediction avanzato

Per implementare la predizione a livello hardware, il processore utilizza una **tavella di predizione** dei salti. Una possibile struttura prevede i campi (Indirizzo dell'istruzione di salto, indirizzo di destinazione, 2 bit per codificare lo stato).

La tabella viene aggiornata in parallelo durante la fase di Execute (EX), ovvero nel momento in cui il processore verifica se la predizione è corretta. Se c'è stato un errore, si aggiorna lo stato dell'automa (e la corrispondente voce nella tabella). Tale struttura è memorizzata in un'apposita area di memoria, detta **Branch History Table** (BHT).

Con l'introduzione del sistema a pipeline e del meccanismo dell'**internal forwarding**, che altera la sequenzialità delle istruzioni eseguite dal processore, nasce il problema della **gestione delle interruzioni**. Infatti può capitare che alcune istruzioni siano completate in un ordine diverso da quello in cui sono state avviate. In tal caso è possibile che si generino delle *interruzioni non esatte*, cioè situazioni in cui un'istruzione provoca un'eccezione prima che tutte quelle che la precedono siano completate. Ovviamente, per la corretta gestione delle eccezioni stesse, l'obiettivo che ci si pone è quello di creare un

sistema che sia in grado di arrestare opportunamente la pipe in modo da completare le istruzioni che precedono quella che ha generato l'interruzione senza avviare quelle che seguono, così da ottenere un comportamento che rispetti quello descritto dal modello di Von Neumann che viene definito di *gestione precisa* delle interruzioni. Le **interruzioni esterne** sono più facili da gestire: infatti la interrupt service routine relativa all'interruzione sopraggiunta verrà inserita nella pipeline come una normale istruzione, dopo che tutte le istruzioni precedenti verranno correttamente terminate, comprese quelle ibernate. Più critico è il caso delle **interruzioni interne**. Infatti questo genere di interruzioni possono essere causate da diverse istruzioni contemporaneamente all'interno della pipe in diversi stages. Non è dunque garantito il comportamento sequenziale di Von Neumann. Un esempio abbastanza lampante di conflitto è il caso in cui nella stessa pipe un'istruzione cerca di effettuare un accesso illegale in memoria (eccezione scatenata nella fase MEM) mentre un'altra istruzione, successiva, tenta una divisione per zero (eccezione scatenata nella fase ID).

Alcuni processori rinunciano del tutto ad avere interruzioni precise. In questo caso è compito del software assicurare che nella stessa pipeline non ci siano istruzioni le cui possibili eccezioni possano causare questo tipo di conflitto. Considerando il tasso di istruzioni che generano eccezione in un generico codice, questa soluzione può essere più o meno accettabile. Altri processori invece riescono a ricostruire interruzioni precise, utilizzando tecniche di tipo conservativo o ottimistico. Nelle tecniche di tipo **conservativo**, il processore fa procedere nella pipe un'istruzione finché è sicuro che non possa essere fonte di interruzione, e solo a questo punto ne avvia un'altra. Il parallelismo intrinseco della pipe è dunque limitato, a vantaggio della gestione precisa delle eccezioni.

Nel caso di tecniche di tipo **ottimistico**, vengono inserite comunque istruzioni nella pipeline indipendentemente dalla possibilità di generare o meno eccezioni. Se emergono conflitti il sistema deve essere in grado di effettuare un *rollback*, e quindi è necessaria un'immagine dello stato del sistema. Il rollback può essere effettuato mediante due tecniche:

- **Check point:** Vengono effettuate periodici snapshot dello stato dei registri del processore, compreso il PC, conservate in una opportuna area di memoria. Nel momento in cui si verifica un'eccezione, vengono ricopiate i valori memorizzati nei registri del processore e si riprende l'esecuzione *sequenziale* del programma, quindi un'istruzione per volta, a partire dall'ultimo valore di PC riportato nello snapshot. Occorre in questo caso scegliere con attenzione la frequenza di cattura dello snapshot.
- **History Buffer:** Ogni istruzione eseguita viene memorizzata in un'area di memoria insieme ai valori dei propri operandi. Ogni istruzione memorizzata può essere cancellata dal buffer solo quando tutte quelle che la precedono si sono concluse senza generare interruzione. È possibile definire in questo modo una sequenza di operazioni UNDO da effettuare nel caso un'istruzione ibernata precedentemente causi un'eccezione. Questa tecnica è anche utile per garantire la gestione precisa delle interruzioni, facendo in modo che un'eccezione non venga gestita se prima non sono terminate correttamente (quindi senza causare eccezioni) le istruzioni ibernate.

1.3 Problemi nelle architetture superscalari

Per migliorare le prestazioni di un processore, si può pensare di realizzare un'architettura che presenti più pipe che eseguono diverse istruzioni in parallelo. Questo tipo di architettura viene definito **superscalare**. Questa soluzione introduce diverse problematiche. Una prima problematica riguarda il prelievo delle istruzioni in memoria. Le istruzioni vengono prelevate in maniera sequenziale dalla memoria condivisa, quindi non è possibile che due pipeline inseriscano contemporaneamente in esecuzione due istruzioni diverse. Se le pipeline sono completamente disgiunte, ovvero non condividono alcuna unità funzionale, allora l'unico altro problema è quello relativo alla *branch prediction*. I dispositivi per la predizione del salto devono essere, in questo caso, duplicati. Invece se le pipe condividono alcuni stages, la cooperazione e la competizione per le risorse condivise deve essere gestita in maniera più complessa via hardware. Quindi non c'è una vera e propria alimentazione in parallelo. Questa condizione di funzionamento, unita alla considerazione che le pipe condividono alcune unità funzionali, ci consente di affermare che il sistema nel suo complesso può essere visto come costituito da una sola pipe con più unità specializzate e con altre che sono in comune. Infatti è facile notare che nelle architetture superscalari ciascuna pipe è specializzata nell'esecuzione di particolari operazioni. Inoltre, per distribuire le istruzioni tra le pipe è possibile pensare di introdurre a monte della fase di EX di ciascuna pipe (dopo le fasi IF e ID) un'unità hardware, detta **Dispatch**, che analizza più istruzioni per volta e decide quali inserire in quale pipe. Ogni fase di ciascuna pipe, in generale, può essere utilizzata da più pipe per più cicli, quindi risulta particolarmente interessante l'idea di affidare al Dispatch anche il compito di riconoscere le caratteristiche di ciascuna operazione per evitare collisioni tra le pipe. Per le sue caratteristiche, la fase di dispatch non può essere realizzata in software, poiché deve essere molto veloce ed efficiente, quindi deve essere necessariamente realizzata in hardware. Strumento fondamentale sia per il compilatore che per il Dispatcher è il *collision vector*, foriero di informazioni utili riguardo la compatibilità temporale di più operazioni che necessitano di condividere la stessa unità funzionale. Per ulteriori dettagli, consultare gli appunti di APC.

1.4 Memoria

Il collo di bottiglia principale per le prestazioni di un calcolatore è sicuramente costituito dalla memoria. Distinguiamo DRAM (Dynamic Random Access Memory), poco costose ma lente e SRAM (Static Random Access Memory), molto più costose a causa dell'elevato numero di transistori ma molto veloci. Viste le differenze di costo e di velocità, diventa conveniente costruire una gerarchia di livelli di memoria, con la memoria più veloce posta più vicino al processore e quella più lenta, meno costosa, posta più distante. Sfruttando il **principio di località**, l'architettura della memoria in un calcolatore moderno viene implementata come una *gerarchia multilivello*, costituita da un insieme di memorie caratterizzate da differenti **tempi di accesso**, **capacità** e **costo per bit**. A parità di capacità, le memorie con prestazioni più elevate risultano sensibilmente più costose rispetto a quelle più lente; di conseguenza, esse vengono realizzate con dimensioni ridotte e collocate nei livelli più alti della gerarchia, ossia in prossimità della CPU. Al contrario, le memorie di capacità maggiore, ma con latenze più elevate e costo per bit

inferiore, sono collocate nei livelli più bassi, fungendo da *storage* di supporto.

L'obiettivo principale di tale organizzazione è quello di fornire al programmatore l'illusione di disporre di un'unica memoria avente contemporaneamente la **velocità** tipica dei livelli più alti (ad esempio le *cache*) e la **capacità** tipica dei livelli più bassi (come la *memoria principale* e la *memoria secondaria*). In tal modo, la gerarchia di memoria realizza un compromesso efficace tra *prestazioni*, *costo* e *scalabilità*, nascondendo la complessità dell'accesso ai diversi livelli e ottimizzando l'utilizzo delle risorse hardware disponibili. Tecnicamente, il livello più elevato della gerarchia di memoria dovrebbe essere identificato con il **register file** interno alla **CPU**. Tuttavia, la sua gestione risulta peculiare ed estremamente semplificata, poiché la maggior parte delle problematiche legate al suo utilizzo viene affrontata direttamente dal **compilatore**. Per tale motivo, esso non verrà considerato nell'analisi seguente. L'attenzione sarà invece rivolta ai livelli immediatamente successivi, ovvero alle cosiddette *memorie cache*, che costituiscono le gerarchie più alte della *memoria di lavoro*.

Nei calcolatori moderni è sempre presente almeno un livello di **cache**, e molto spesso ne sono disponibili due o più, organizzati gerarchicamente come **L1**, **L2** e, in alcuni casi, **L3**. La cache è strutturata in unità dette *blocchi* o *linee di cache* (*cache lines*), che rappresentano la minima quantità di informazione trasferibile fra due livelli adiacenti della gerarchia.

Quando la CPU richiede un dato e questo risulta già contenuto in uno dei blocchi presenti nella cache di livello superiore, si verifica un **cache hit**, ossia un accesso corretto ed immediato ai dati richiesti. Al contrario, se il dato non è disponibile nella cache, si verifica un **cache miss**; in tal caso è necessario accedere al livello inferiore della gerarchia, individuare il blocco che contiene il dato richiesto e trasferirlo nel livello superiore, aggiornando la cache. Questo meccanismo consente di ridurre in maniera significativa la latenza media di accesso alla memoria, pur introducendo problematiche di *gestione della sostituzione dei blocchi* e di *politiche di coerenza*, che rappresentano aspetti cruciali nell'architettura delle memorie moderne.

1.4.1 Problema del piazzamento di un blocco

Consideriamo innanzitutto il problema del **piazzamento di un blocco** nella memoria cache. Durante l'esecuzione di un programma, la **CPU** può tentare di accedere, in linea di principio, a qualunque parola all'interno dello *spazio totale di indirizzamento*. Tale spazio può essere idealmente assimilato all'intera **memoria RAM**, che rappresenta il livello più basso della gerarchia di memoria di lavoro. Tuttavia, la capacità della cache è di gran lunga inferiore rispetto a quella della memoria principale; ne consegue che non è possibile mantenere simultaneamente in cache tutte le parole potenzialmente indirizzabili. Diventa quindi necessario stabilire un meccanismo che definisca come ciascuna parola della memoria indirizzabile possa essere, quando richiesto, mappata in una specifica posizione della cache. In altri termini, occorre definire una **corrispondenza** tra l'*indirizzo di memoria* di una parola e la *locazione cache* che potrà contenerla. A questo scopo, l'architettura dei sistemi di memoria ha introdotto tre principali modalità di organizzazione:

- **Cache a indirizzamento diretto:** ogni blocco di memoria principale può essere memorizzato in una sola posizione predeterminata della cache, stabilita in base

a una funzione di mappatura (tipicamente, una parte dell'indirizzo). Questa soluzione è semplice ed efficiente in termini di hardware, ma può portare a numerosi *conflict miss* quando più blocchi competono per la stessa posizione.

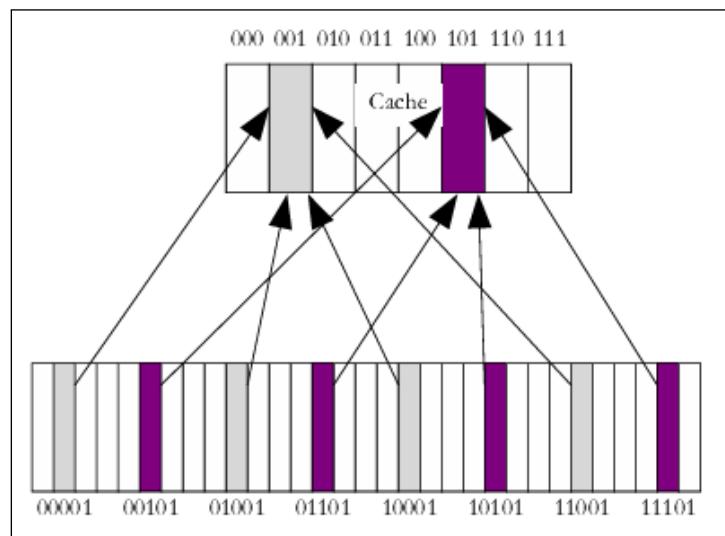
- **Cache set-associativa a n vie:** la cache è suddivisa in insiemi (*sets*), ciascuno composto da n linee. Un blocco può essere memorizzato in qualunque linea del set corrispondente. Questa soluzione rappresenta un compromesso tra flessibilità e complessità hardware, riducendo sensibilmente i conflitti rispetto al caso diretto.
- **Cache completamente associativa:** un blocco può essere collocato in qualunque posizione della cache, senza vincoli. Questo approccio minimizza i *miss da conflitto*, ma richiede circuiteria di ricerca più complessa e costosa, in quanto ogni accesso implica il confronto parallelo dell'indirizzo con tutte le linee della cache.

Ciascuna di queste soluzioni presenta vantaggi e svantaggi in termini di **latenza di accesso**, **costo hardware** e **frequenza dei miss**, rendendo la scelta del tipo di mappatura un elemento cruciale nella progettazione delle architetture di memoria.

Nel caso di **cache a indirizzamento diretto** ogni locazione di memoria corrisponde esattamente ad una posizione nella cache. Chiamando I_c l'indirizzo del blocco nella cache, I_m l'indirizzo del blocco in memoria e N il numero di blocchi contenuti dalla cache la corrispondenza è data da:

$$I_c = I_m \bmod n \quad (1.1)$$

Poiché il numero degli elementi della cache è una potenza di 2, l'operazione modulo può essere fatta considerando i $\log_2 N$ bit meno significativi, che sono usati come indice della cache.



Nel caso di **cache completamente associativa**, ogni blocco di memoria principale può essere collocato in una qualunque posizione della memoria cache, senza vincoli deterministici sul piazzamento. La corrispondenza tra un indirizzo di memoria in **RAM** e una posizione della cache non è quindi stabilita a priori, ma viene gestita mediante la ricerca del blocco desiderato all'interno dell'intera cache. Poiché un blocco può risiedere in qualunque posizione, al momento dell'accesso è necessario confrontare l'*indirizzo* richiesto dalla CPU con tutti gli identificativi (i cosiddetti **tag**) dei blocchi memorizzati

in cache. Questo richiede la presenza di un meccanismo hardware di confronto parallelo (*associative search*), che rende la soluzione estremamente flessibile ma costosa dal punto di vista circuitale.

Una **cache set-associativa a n vie** è organizzata in un numero finito di **insiemi** (*sets*), ciascuno dei quali contiene n linee o blocchi. La memoria principale (RAM) è anch'essa concettualmente suddivisa in blocchi; ciascun blocco della RAM viene mappato in modo deterministico su un solo insieme della cache, ma all'interno di tale insieme può essere collocato in una qualunque delle n vie disponibili.

In termini formali, l'indice dell'insieme si ottiene mediante l'operazione:

$$(\text{Indirizzo blocco})_{\text{set}} = (\text{Indirizzo blocco})_{\text{mem}} \bmod (\#\text{insiemi})$$

dove il numero degli insiemi è pari al rapporto tra il numero totale di linee della cache e il numero di vie n .

Esempio d'uso: Si consideri una cache composta da 64 linee, organizzata come **4-vie set-associativa** ($n = 4$). In questo caso:

$$\#\text{insiemi} = \frac{64}{4} = 16$$

Sono quindi necessari $\log_2 16 = 4$ bit dell'indirizzo per individuare l'insieme. Supponiamo che la CPU richieda il blocco di memoria con indirizzo decimale 45. L'indice dell'insieme risulta:

$$45 \bmod 16 = 13$$

Pertanto, il blocco 45 può essere collocato in qualunque delle 4 vie appartenenti all'insieme 13. Se in seguito la CPU richiede il blocco con indirizzo 61, avremo:

$$61 \bmod 16 = 13$$

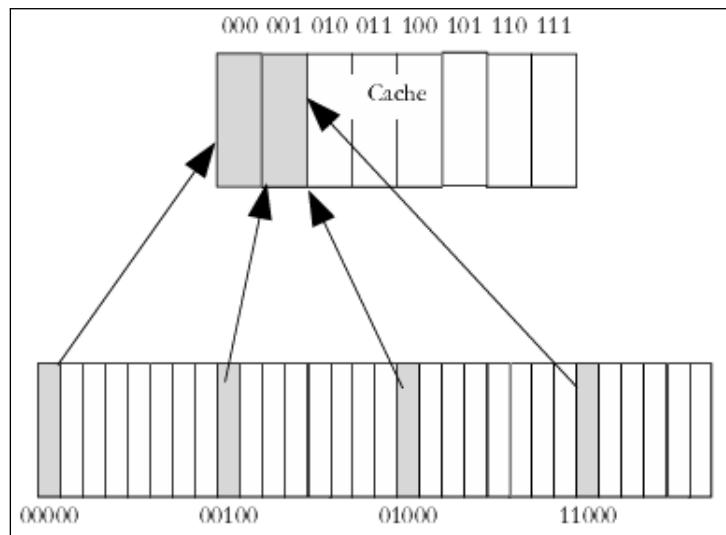
Anche questo blocco si mapperà nello stesso insieme 13, potendo occupare una delle 4 vie disponibili. Se tutte le vie dell'insieme risultassero già occupate, uno dei blocchi dovrebbe essere sostituito secondo una *politica di rimpiazzo*.

Questo schema rappresenta un *compromesso intermedio* tra la cache a indirizzamento diretto (che vincola rigidamente la posizione) e quella completamente associativa (che lascia piena libertà di collocamento). In una cache set-associativa, la mappatura dal blocco RAM all'insieme avviene tramite indirizzamento diretto, mentre la ricerca del blocco avviene confrontando i **tag** di tutte le vie dell'insieme selezionato.

Ogni politica di piazzamento può in realtà essere considerata una variazione di quella set associativa: una cache a indirizzamento diretto è una cache set-associativa a 1 vie, in cui ogni elemento della cache contiene un blocco e forma un insieme di un solo elemento. Una cache di m elementi completamente associativa è una cache set-associativa a m vie: c'è un solo insieme di m blocchi e un elemento può trovarsi in uno qualsiasi dei blocchi dell'insieme.

1.4.2 Problema della ricerca di un blocco

Poiché ogni linea della cache può contenere blocchi provenienti da differenti locazioni della **memoria principale**, è necessario disporre di un meccanismo che consenta di



stabilire se il dato richiesto dalla CPU sia effettivamente presente nella cache. A tale scopo, ad ogni elemento della cache viene associato un insieme di **etichette** (tag), che contengono le informazioni necessarie per identificare la corrispondenza tra l'indirizzo richiesto e quello del blocco memorizzato nella linea.

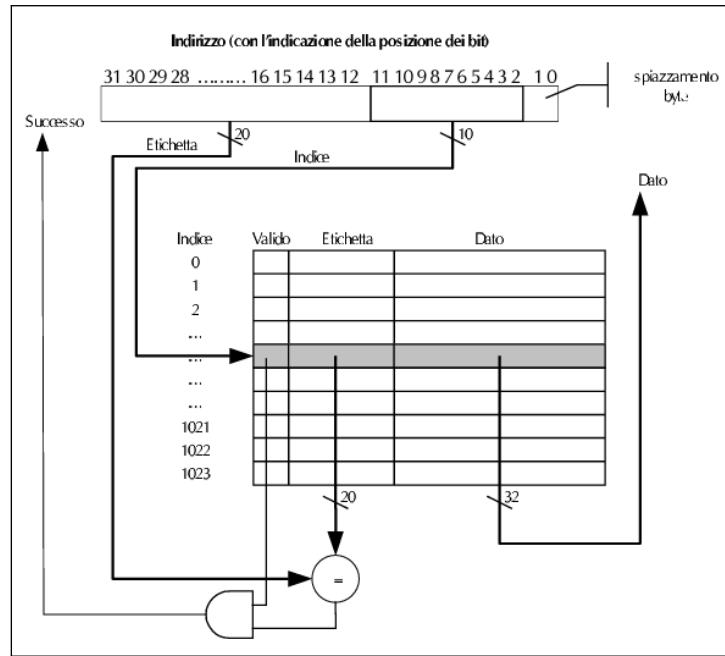
In altre parole, durante un accesso, l'indice della cache individua una o più linee candidate (a seconda della politica di piazzamento: indirizzamento diretto, set-associativa o completamente associativa), e i tag vengono confrontati con la parte più significativa dell'indirizzo in memoria. Solo in caso di uguaglianza tra il tag memorizzato e quello dell'indirizzo richiesto si verifica un **cache hit**; in caso contrario si ha un **cache miss**.

Un'ulteriore informazione fondamentale è il **bit di validità** (validity bit), che indica se la linea della cache contiene dati effettivamente validi. Infatti, al momento dell'accensione del processore, la cache risulta inizialmente vuota, e le etichette non hanno alcun significato: tutte le linee vengono quindi marcate come *non valide*. Il bit di validità è del tutto indipendente dalla filosofia di piazzamento scelta ed è quindi presente in qualsiasi schema di organizzazione della cache.

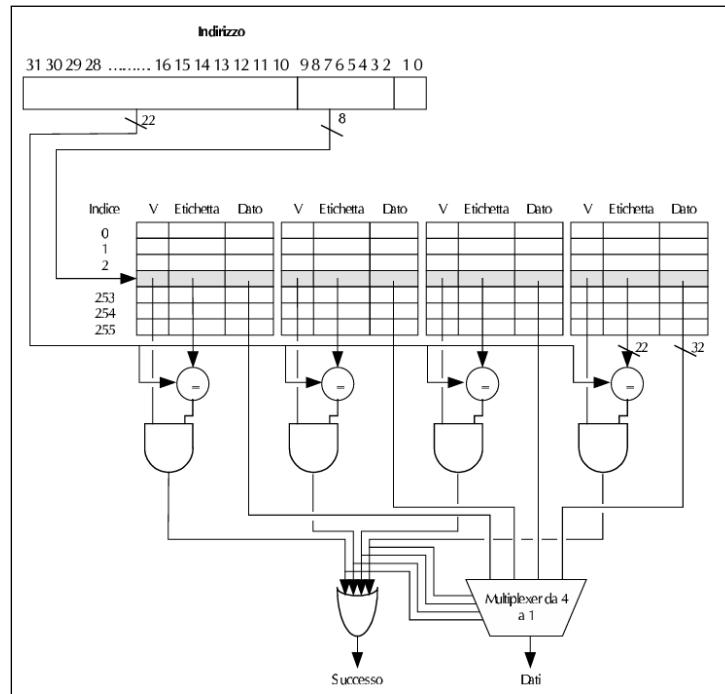
A titolo esemplificativo, consideriamo una cache a indirizzamento diretto da 4 Kbyte, blocco corrispondente ad una sola parola di 32 bit; Questa situazione viene schematizzata come illustrato nella seguente figura.

Lo schema descritto finora non sfrutta il principio di località spaziale, in quanto ogni parola corrisponde ad un blocco. Per trarre vantaggio dalla località spaziale è necessario che la dimensione del blocco della cache sia maggiore della dimensione della parola in memoria, in modo che il blocco contenga più di una parola. Serve quindi un nuovo campo in cui suddividere l'indirizzo che corrisponda allo spiazzamento della parola nel blocco. In caso di *miss*, dalla memoria centrale vengono prelevate più parole adiacenti che hanno un'elevata probabilità di essere richiesta a breve. Il numero totale di etichette è inferiore in questo caso, perché ogni etichetta è utilizzata per più parole.

Consideriamo il caso più interessante della ricerca di un blocco in una cache associativa a n vie. Ogni blocco della cache comprende ancora un'etichetta che permette di individuare l'indirizzo del blocco in memoria principale. Il valore dell'indice serve a selezionare l'insieme che contiene l'indirizzo desiderato. Se la CPU produce una richiesta ad un determinato indirizzo, viene selezionato un gruppo di blocchi corrispondenti all'insieme specificato, e ne vengono controllate *in parallelo* le etichette. Vediamo a titolo di esempio



lo schema generale di una cache set-associativa a 4 vie da 4 Kbye e blocco da 32 bit:



Se si mantiene costante la dimensione totale, al crescere dell'associatività aumenta anche il numero dei blocchi compresi nell'insieme, e questo corrisponde al numero dei confronti associativi che devono essere effettuati per realizzare una ricerca in parallelo. Ogni volta che si raddoppia il grado di associatività si raddoppia il numero di blocchi compresi in un insieme e si dimezza il numero degli insiemi. Al tempo stesso, ogni incremento dell'associatività di un fattore due fa diminuire di un bit la dimensione dell'indice e aumentare di un bit la dimensione dell'etichetta. Vale la pena osservare che aumentando l'associatività aumenta il numero dei comparatori e aumenta anche la dimensione di ogni singolo comparatore, e ciò corrisponde ad una maggiore complessità circuitale.

In una cache completamente associativa c'è un unico insieme e tutti i blocchi devono essere esaminati in parallelo: di conseguenza l'intero indirizzo, a parte lo spiazzamento nel blocco, viene confrontato con l'etichetta di ogni blocco: occorrono quindi tanti comparatori quanti sono i blocchi. (Dualmente, in una cache a indirizzamento diretto è necessario un solo comparatore, dato che l'elemento può essere in una sola posizione). In una cache set-associativa a n vie, sono necessari n comparatori, oltre a un multiplexer da n a 1 per scegliere tra gli n possibili blocchi dell'insieme selezionato. I comparatori individuano quale elemento dell'insieme corrisponde all'etichetta e forniscono quindi gli ingressi di selezione del multiplexer, in modo da avviare all'uscita uno solo degli n blocchi dell'insieme selezionato. L'accesso alla cache si effettua utilizzando l'indice per individuare l'insieme e poi esaminando in parallelo tutti gli n blocchi dell'insieme. Oltre al costo correlato ai comparatori aggiunti occorre tenere conto dei ritardi imposti dalla necessità di confrontare e selezionare l'elemento desiderato tra quelli dell'insieme. D'altronde, è chiaro che la soluzione completamente associativa permette uno sfruttamento migliore dello spazio disponibile in cache, dato che, ad esempio, in fase di scrittura, è possibile trasferire un blocco dalla RAM a un qualsiasi blocco della cache. In ogni gerarchia di memoria, la scelta tra lo schema a indirizzamento diretto, quello set-associativo e quello completamente associativo dipende dal confronto tra il costo di un fallimento e quello di realizzazione dell'associatività, sia dal punto di vista del tempo sia da quello della circuiteria aggiuntiva.

1.4.3 Problema della sostituzione di un blocco

Si consideri ora il problema della **sostituzione di un blocco** nella cache. Quando si verifica un *cache miss*, è necessario decidere quale linea della cache liberare per fare spazio al nuovo blocco proveniente dalla memoria principale.

Nel caso di una **cache a indirizzamento diretto**, il problema non si pone: l'indirizzo di memoria individua un'unica linea di cache e il blocco presente in quella posizione viene automaticamente rimpiazzato. Al contrario, in una **cache completamente associativa**, ogni blocco della cache è un potenziale candidato per la sostituzione, e diventa quindi necessario adottare una *politica di rimpiazzo*. Nelle **cache set-associative**, l'insieme di appartenenza del blocco è determinato in maniera univoca dall'indirizzo, ma la scelta deve comunque essere effettuata tra le n vie appartenenti all'insieme selezionato.

Le principali strategie di sostituzione comunemente adottate sono tre:

- **Sostituzione casuale (Random)**: la linea da rimpiazzare viene scelta in maniera casuale tra le candidate. Tale metodo è semplice da implementare, eventualmente con l'ausilio di componenti hardware per la generazione pseudo-casuale, ma non garantisce prestazioni ottimali.
- **Least Recently Used (LRU)**: viene sostituito il blocco che non è stato utilizzato da più tempo. Una possibile implementazione elementare associa ad ogni blocco un contatore: quando un blocco viene caricato, il contatore viene inizializzato al valore massimo; ad ogni accesso a un blocco diverso, i contatori vengono decrementati. Al momento della sostituzione, viene scelto il blocco con contatore minimo. Sebbene LRU sia molto efficace nel ridurre i *miss*, la sua implementazione hardware può risultare costosa nelle cache di grandi dimensioni.

- **First In First Out (FIFO)**: viene sostituito il blocco caricato da più tempo, indipendentemente dal suo utilizzo negli accessi recenti. Questo schema è semplice da implementare mediante una coda circolare, ma può risultare meno performante rispetto a LRU in presenza di dati con elevata riutilizzabilità.

1.4.4 Problema della strategia di scrittura

Infine, consideriamo il problema della **strategia di scrittura**. Esso nasce dal fatto che, quando il processore deve memorizzare il risultato di un'operazione, è desiderabile che l'istruzione di scrittura venga eseguita con la massima velocità possibile (quindi accedendo alla **cache**), ma al tempo stesso è necessario che i dati contenuti nella cache siano **coerenti** con quelli presenti nella memoria principale.

Le principali strategie adottate sono due: **write-through** e **write-back**. Nell'approccio **write-through**, ogni operazione di scrittura aggiorna simultaneamente sia il blocco presente nella cache sia quello corrispondente in memoria principale. In questo modo la coerenza è sempre garantita, ma al prezzo di un tempo di scrittura maggiore, poiché ogni operazione implica un accesso anche alla memoria più lenta.

Nell'approccio **write-back** (o *copy-back*), invece, i dati vengono scritti inizialmente solo nella cache. La copia nel livello inferiore della gerarchia avviene soltanto quando il blocco modificato deve essere sostituito. Di conseguenza, dopo un'operazione di scrittura, la cache può contenere valori diversi rispetto alla RAM. In questo caso si parla di **incoerenza**: il blocco può essere *clean* (non modificato) oppure *dirty* (modificato). Per gestire questa condizione, ogni linea della cache è dotata di un **dirty bit**, che segnala se il contenuto è stato modificato rispetto alla memoria principale.

Poiché le prestazioni delle CPU crescono a un ritmo molto più elevato rispetto a quelle delle memorie DRAM, la frequenza delle scritture generate dal processore tende a superare quella sostenibile dalla memoria principale. Inoltre, secondo il **principio di località**, se un blocco viene scritto una volta, è altamente probabile che venga riscritto più volte prima di essere sostituito. Per questi motivi, la strategia **write-back** è destinata a diventare sempre più diffusa nei sistemi moderni. Entrambe le soluzioni presentano vantaggi e svantaggi.

I principali **vantaggi della write-back** sono:

- le singole parole possono essere scritte alla frequenza accettata dalla cache, senza attendere i tempi della memoria principale;
- più scritture all'interno dello stesso blocco richiedono una sola operazione di scrittura nel livello inferiore al momento della sostituzione;
- quando i blocchi vengono trasferiti, l'uso di un bus ampio permette di sfruttare la scrittura a livello di intero blocco, migliorando anche la gestione dei fallimenti in lettura.

I principali **vantaggi della write-through** sono invece:

- i fallimenti in lettura risultano meno costosi, poiché non richiedono mai la scrittura preventiva di un blocco nel livello inferiore (al contrario, con write-back un blocco *dirty* deve essere scritto prima di essere sostituito);
- lo schema è più semplice da implementare rispetto al write-back; tuttavia, per essere efficace in un sistema ad alte prestazioni, una cache **write-through** deve essere

dotata di un **write buffer**, così da non costringere la CPU ad attendere i tempi di accesso della memoria principale.

Chapter 2

Architetture superscalari dei moderni processori

Consideriamo una semplice architettura pipeline a cinque stages, costituita da *instruction fetch IF* → *instruction decode e operand assembly ID* → *Execute EXE* → *Memory Load/Store MEM* → *Writeback WB*. Ognuno di questi stages logici ha una mappatura diretta con una macchina combinatoriale dell'architettura del processore.



Info: Una macchina combinatoriale (o circuito logico combinatorio) è un sistema logico digitale in cui le uscite dipendono esclusivamente dai valori presenti agli ingressi in un dato istante, senza alcuna memoria degli stati precedenti.

Diverse fasi di diverse istruzioni, come osservato nel capitolo precedente, possono sovrapporsi nel tempo, e il risultato è un incremento del throughput (istruzioni completate/unità di tempo). È immediatamente possibile osservare che il tempo di completamento di un'istruzione (in letteratura *latency*) non migliora, anzi al più peggiora, a causa del già discusso overhead introdotto nel controllo della pipeline (Capitolo 1). Il guadagno in termini di throughput idealmente è pari al numero di stadi, ma nella pratica è limitato dall'overhead di pipeline e dagli *hazards*.

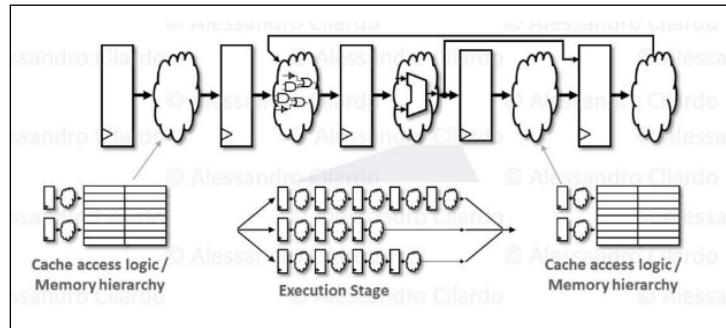


Warning: Un **ciclo di clock** è l'unità di tempo fondamentale dettata dal segnale di clock. Tutte le operazioni elementari avvengono in corrispondenza dei fronti di salita del segnale di clock. Un **ciclo del processore** è il tempo necessario al processore per completare un'operazione base del set di istruzioni. Nelle architetture RISC un ciclo di processore coincide con pochi cicli di clock. Questo tempo viene misurato in **IPC** (instruction per clock).

Quindi in una pipeline il tempo di attraversamento di un'istruzione è leggermente peggiorato principalmente a causa del fatto che la sequenzialità combinatoriale della macchina viene frammentata da registri che hanno bisogno di un tempo di *setup*. Il parametro che ne beneficia è il *clock cycle*, che rispetto ad una macchina sequenziale, è ridotto a $\frac{1}{N}$ dove N è il numero di fasi in cui viene frammentata la macchina combinatoriale. In realtà il clock non può essere più veloce del più lento stadio combinatoriale, che

ne scandisce un limite superiore. La tecnica della pipeline dunque non è facilmente scalabile: I *path* combinatoriali non possono essere arbitrariamente frammentati; inoltre, superata una certa granularità, gli overhead limitano i benefici.

Nel caso reale, nei diversi stages la latenza è variabile. Questo è dovuto al fatto che l'infrastruttura della memoria è complessa e nel caso peggiore potrebbero verificarsi dei *cache miss*, così come il processore potrebbe avere diverse unità funzionali dedicate all'esecuzione, ognuna con la propria latenza (alcune operazioni sono più lunghe e complesse di altre).



Per migliorare l'IPC, si può ricorrere alla moltiplicazione delle istanze di unità funzionali che lavorano in parallelo su più istruzioni. Se $\text{IPC} > 1$ si parla di architetture **super-scalari**.

Parlando sempre di pipeline semplice a cinque stadi, possiamo invece agire sull'overhead introdotto dal controllo della pipeline, in particolare risolvere quei conflitti che provocherebbero il blocco (stall) della pipe. Questo si può fare con l'esecuzione delle istruzioni *out of order*, ovvero con un problema di scheduling vincolato. La pipeline può essere vista nella prospettiva di schedulatore di micro operazioni soggette a vincoli strutturali e dipendenze. I vincoli strutturali sono dettati dell'hardware e scandiscono cosa si può fare in un particolare istante di tempo. I vincoli sulle dipendenze riguardano le relazioni di precedenza tra micro operazioni che devono necessariamente essere soddisfatte nel momento in cui vengono processate dalle unità hardware. Rischedulare le micro operazioni può violare i vincoli sulle dipendenze:

- **READ AFTER WRITE (RAW)**: l'operazione di lettura operando di un' istruzione successiva è performata prima dell'operazione di WB sullo stesso registro di un'istruzione precedente;
- **WRITE AFTER WRITE (WAW)**: due istruzioni scrivono lo stesso registro, ma l'ordine di WB è invertito e dunque nel registro si trova un dato vecchio;
- **WRITE AFTER READ (WAR)**: l'operazione di lettura operando di un'istruzione precedente viene ritardata fino al momento in cui l'operazione di WB di un'istruzione successiva è già avvenuta, distruggendo il contenuto del registro prima che sia letto dall'istruzione precedente.

2.1 Scheduling statico

Lo scheduling statico di una pipeline è una tecnica di ottimizzazione usata nei processori con pipelining (soprattutto nelle architetture RISC e VLIW) per ridurre o eliminare i

hazard (conflitti di dati, di controllo o strutturali) senza dover ricorrere a meccanismi di risoluzione hardware dinamici. Statico significa che lo scheduling viene fatto dal compilatore, in fase di compilazione, non dal processore a runtime. L'idea è che il compilatore riordini le istruzioni in modo da minimizzare i cicli di stallo (stall) e sfruttare al massimo la pipeline.

2.2 Scheduling dinamico

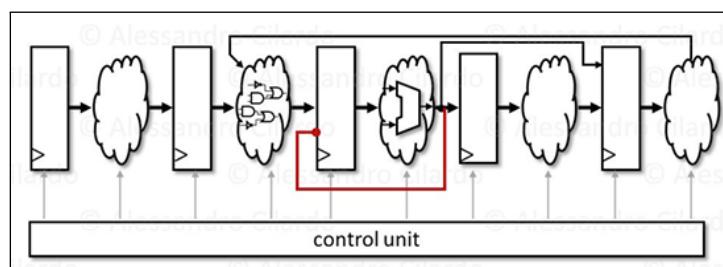
Con scheduling dinamico intendiamo tecniche hardware che *on the fly*, durante l'esecuzione, cambiano l'ordine di processo di determinate operazioni per risolvere i conflitti ed evitare quando possibile lo stallo della pipeline. I vantaggi di uno scheduling dinamico sono molteplici:

- Permette a codice compilato per una determinata architettura di essere efficiente anche su un'architettura diversa, eliminando parzialmente la relazione che sussiste tra efficienza e compilazione;
- Permette di gestire casi in cui alcune dipendenze non sono note a tempo di compilazione (riferimenti in memoria e salti dipendenti dai dati);
- Permette al processore di tollerare dei ritardi non prevedibili a tempo di compilazione (cache miss).

Il datapath di una pipeline deve essere gestito da un'opportuna unità di controllo, che risolva potenziali condizioni di *hazard*. Questa unità è responsabile di rilevare il problema e risolverlo bloccando la pipeline a partire da un certo stadio in poi, inserendo delle *bolle*. Per rendere possibile questo controllo è necessario propagare dalla fase ID in poi gli indici degli operandi, in modo tale che l'unità di controllo possa rilevare possibili conflitti.

2.2.1 Forward paths

Il vincolo sulla dipendenza RAW riguarda *vere dipendenze*, ovvero dipendenze dettate dal problema produttore-consumatore tra le istruzioni, e sono indipendenti dall'architettura (riguardano in senso logico il flusso di esecuzione del codice). Inserire un path diretto [output ALU → registro operandi] permette di rilassare il vincolo di dipendenza e fare in modo che l'unità di controllo possa risolvere il potenziale conflitto. In altre parole, si cambia il *grafo delle dipendenze* tra le operazioni.





Info: Grafo delle dipendenze: grafo orientato in cui le istruzioni costituiscono i nodi, gli archi invece rappresentano le dipendenze tra queste.

Il costo di questi benefici è una notevole complicazione dell'hardware. Osserviamo inoltre che lo scheduling dinamico è utilizzabile insieme allo scheduling statico effettuato dal compilatore: una tecnica non prescinde l'altra.

Osserviamo che in uno schema pipeline a cinque stages di base, dove non contempliamo istruzioni fuori ordine, è verificata sempre la condizione: *una micro-operazione di un'istruzione precedente è sempre eseguita prima della stessa o una successiva micro-operazione di un'istruzione successiva*. In altre parole, se un'istruzione A precede un'istruzione B, l'operazione di *lettura operandi* dell'istruzione A è eseguita prima delle operazioni EX, MEM e WB dell'istruzione B. Di conseguenza, in questo schema semplificato non sono possibili hazards di tipo WAW e WAR. Questo è in generale **falso** per gli approcci basati sul rescheduling delle operazioni.

2.2.2 Esecuzione out-of-order

Un'unità funzionale, come un moltiplicatore o un divisore, può essere internamente organizzata come una pipeline. In questo modo l'unità può iniziare una nuova operazione ad ogni ciclo, pur impiegando diversi cicli per completare un'operazione.



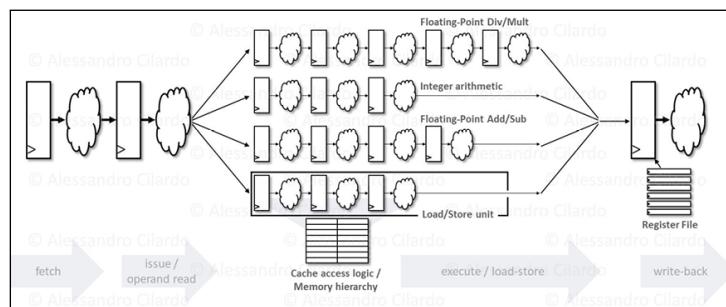
Info: L'**initialization interval** (II) è il numero di cicli di clock che devono passare prima di poter avviare una nuova iterazione della pipeline. Questo limite è imposto dall'hardware.

Se l'II dell'unità funzionale serializzata è minore della latenza totale dell'unità, allora più micro-operazioni possono coesistere nella stessa unità funzionale, pur attraversandola sempre in ordine. Se lo stage EXE contiene più di un'unità funzionale specializzata, ognuna con diverse latenze, è necessario introdurre l'esecuzione *out of order*. Complicando in questo modo l'hardware, è necessario fare delle considerazioni: differenti unità funzionali possono completare l'esecuzione nello stesso ciclo → hazard strutturali sull'insieme dei registri → WAW hazard; se l'hardware supporta molteplici operazioni di lettura operandi concorrenti, sono possibili hazards strutturali sullo stage EXE (?); Sono possibili hazard di tipo WAR, e gli hazards di tipo RAW sono più frequenti. Nelle architetture superscalari il processore può emettere ed eseguire più istruzioni nello stesso ciclo di clock. Questo implica che diverse unità funzionali possano aver bisogno, contemporaneamente, di leggere o scrivere registri. Per esempio, due istruzioni aritmetiche emesse nello stesso ciclo possono richiedere entrambe di leggere due operandi e di scrivere un risultato; se il file dei registri fosse accessibile da una sola porta di lettura e scrittura, ci sarebbe un collo di bottiglia che impedirebbe l'esecuzione parallela. Per questo motivo i registri devono essere multiportati, cioè progettati per permettere più accessi simultanei da locazioni diverse, in modo da garantire che le varie istruzioni in issue nello stesso ciclo possano ottenere i loro operandi o aggiornare i risultati senza conflitti. Rendere i registri multiporting è difficile principalmente per ragioni di complessità circuitale e di costo. Ogni porta aggiuntiva di lettura o scrittura richiede linee di accesso dedicate, logica di decodifica separata e soprattutto un incremento del numero

di connessioni interne, il che fa crescere in modo quasi quadratico l'area del circuito e la capacità parassita. Questo significa che più porte si aggiungono, più aumenta il consumo energetico e più rallenta il tempo di accesso ai registri.

Avere a che fare con molteplici scritture implica scegliere una strategia di rilevazione della dipendenza e di risoluzione: se si sceglie di rilevare il conflitto già in fase di ID, il processore deve subito confrontare i registri di destinazione delle istruzioni decodificate con quelli delle istruzioni più vecchie ancora in pipeline. Per non perdere l'informazione man mano che le istruzioni avanzano, si usa una catena di shift registers che traccia la distanza (in termini di stadi di pipeline) fino a quando l'istruzione precedente eseguirà il WB. In questo modo, fin dall'inizio si sa che una certa istruzione dovrà aspettare un certo numero di cicli prima di poter scrivere in sicurezza. Il vantaggio è che i conflitti sono gestiti in anticipo, evitando bolle tardive. Lo svantaggio è che servono comparatori e logica aggiuntiva in ID, e gli shift register complicano l'hardware. L'alternativa è rilevare il conflitto il più tardi possibile, cioè attendere che le istruzioni arrivino in prossimità della fase di write-back e solo lì controllare se due istruzioni stanno tentando di scrivere lo stesso registro nello stesso ciclo. Questo riduce la complessità logica in ID e alleggerisce la pipeline nei primi stadi, ma aumenta la probabilità di dover bloccare o annullare (stall o flush) istruzioni già avanzate nella pipeline. In altre parole: hardware più semplice, ma più penalità in caso di conflitti. In generale conviene rilevare i conflitti presto (in ID con shift register) quando si ha un'architettura aggressivamente superscalare o molto profonda, dove i conflitti diventano frequenti e lo stall tardivo sarebbe molto costoso. Gli hazard possono essere significativamente ridotti introducendo **isolated type registers**. Ciò significa che, invece di avere un unico file di registri condiviso da tutte le unità funzionali, il processore suddivide i registri in più gruppi separati, ciascuno dedicato a un certo "tipo" di istruzione o unità. Per esempio, ci possono essere registri isolati per le operazioni intere, altri per le operazioni in virgola mobile etc. La separazione riduce drasticamente le probabilità di conflitto, perché istruzioni appartenenti a domini funzionali diversi non competono per lo stesso insieme di registri. Inoltre, avere registri specializzati permette anche di ridurre la complessità del multiporting, dato che ogni file di registri isolato può essere più piccolo e più facile da gestire in parallelo.

L'idea chiave è separare la fase ID in due fasi indipendenti, **instruction issue** e **operand read**. L'instruction issue consiste nell'accettare l'istruzione, riservando risorse per tracciarla in stages successivi, e ciò viene di solito eseguito *in ordine*; l'operand read invece aspetta che gli operandi siano pronti, ed è tipicamente eseguita *out of order*.



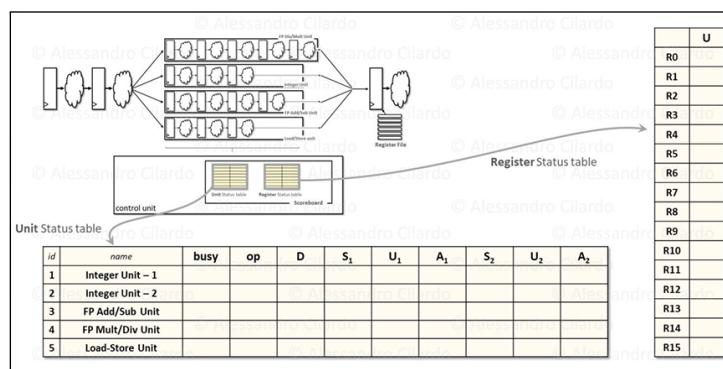
Nella trattazione teorica qui presentata conviene trattare l'unità di load/store (MEM) come un'unità funzionale al pari delle unità funzionali dedite alle operazioni aritmetico-logiche.

2.2.3 Scoreboard

In questa tecnica di implementazione dell'unità di controllo facciamo delle assunzioni:

- L'issue di un'istruzione ovviene in ordine, e in questa fase vengono rilevati hazards strutturali e WAW;
- L'operand read avviene, così come la fase EX e la fase di completamento, out-of-order. Inoltre le dipendenze sui dati sono risolte dinamicamente;
- Non vi sono esplicativi forwarding paths;
- Per ora ci disinteressiamo del modello delle eccezioni/interruzioni.

La logica per le fasi issue e operand read sono gestite attraverso due tabelle (**unit status table** e **register status table**) dall'unità di controllo.



La unit table ha tante righe quante le unità funzionali e i seguenti campi:

<i>id</i>	Intero che identifica l'unità funzionale
<i>name</i>	Nome dell'unità funzionale
<i>busy</i>	Flag che indica se l'unità è occupata
<i>op</i>	Specifica operazione richiesta all'unità funzionale
<i>D</i>	Indice del registro destinazione
<i>S_i</i>	Indice dell' <i>i</i> -esimo registro sorgente
<i>U_i</i>	Id dell'unità che eventualmente sta processando il valore che sarà scritto in <i>S_i</i> (=0 se è già disponibile)
<i>A_i</i>	Flag che verifica se il valore nel registro è già disponibile per la lettura

Per quanto riguarda la Register Status Table, questa ha tante righe quanti sono i registri e l'unico campo *U* →, che contiene l'id dell'unità che eventualmente sta processando il valore che verrà scritto nel registro (=0 se già disponibile).

Le istruzioni sono recuperate dalla pipeline in ordine e poi possono trovarsi in una delle seguenti fasi: Issue, Operand Read, Execute o Write Back. Le regole per aggiornare le tabelle sono queste:

- Un'istruzione viene *accettata* (issued) solo se:
 - L'unità funzionale richiesta è libera (*busy* = 0);

- Il registro di destinazione non è già segnato ($U=0$ nella register status table);
- L'Operand Read è performata solo quando entrambi i registri sorgente sono disponibili;
- Quando viene performata Operand Read, i flag A_1 e A_2 vengono resettati;
- L'esecuzione procede secondo la pipeline dell'unità funzionale;
- Il WriteBack avviene solo se non ci sono Operand Read sospese su tutte le unità funzionali che aspettano il registro che si sta cercando di scrivere; Una volta effettuato il WB, le entry della tabella relative all'istruzione completata vengono liberate.

Una volta che un'istruzione viene accettata, le tabelle di scoreboard vengono aggiornate per tenere traccia delle unità e dei regeistri coinvolti. Formalmente:

next step	stalled if:	action performed when unstalled:
Issue	$(US(id).busy) \vee (RS(D).U \neq 0)$	set the .busy, .op, .D, S_1 , S_2 fields properly in US $US(id).U_1 \leftarrow RS(S_1).U$, $US(id).A_1 \leftarrow \text{not}(US(id).U_1)$ $US(id).U_2 \leftarrow RS(S_2).U$, $US(id).A_2 \leftarrow \text{not}(US(id).U_2)$ $RS(D).U \leftarrow id$
Operan Read	$A_1 = 0 \vee A_2 = 0$	$US(id).A_1 \leftarrow 0$ $US(id).A_2 \leftarrow 0$
Execute	(depends on the internal FU behavior)	-
Write-Back	$\exists j : (US(j).S_1=D \wedge A_1=1)$ $\vee \exists j : (US(j).S_2=D \wedge A_2=1)$	$\forall j : \text{if } US(j).U_1=id \text{ } US(j).A_1 \leftarrow 1$ $\forall j : \text{if } US(j).U_2=id \text{ } US(j).A_2 \leftarrow 1$ $US(id).busy = 0$, $RS(D).U \leftarrow 0$

Si consultino le slides del corso per un esempio sulla tecnica dello scoreboard. In sintesi, questa tecnica di gestione previene gli hazards in questo modo: i RAW sono risolti tenendo traccia delle dipendenze tra le istruzioni che sono già state accettate; i WAR sono risolti ritardando il WB finchè i registri scrivendi non vengono letti, e ritardando la lettura dei registri alla fase Read Operands; i WAW sono risolti a monte ritardando l'issue di un'istruzione finchè l'istruzione precedente non completa la scrittura del registro destinazione. Con questa soluzione, comunque l'IPC massimo resta sempre 1. Osserviamo che i dati sono gestiti da registri dell'architettura, e questa è una limitazione chiave che vorremmo superare, poichè è causa di molti hazards di tipo WAR e WAW.



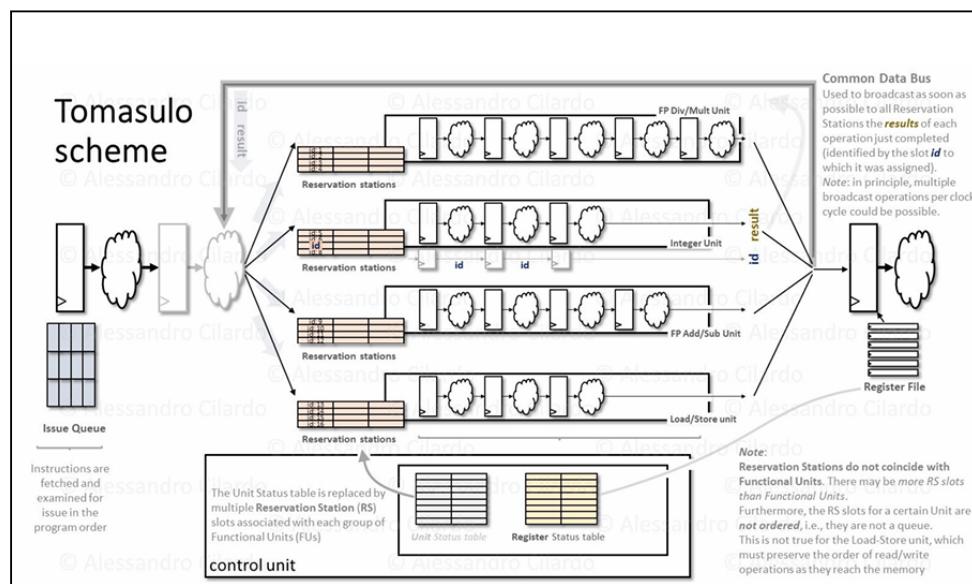
Warning: Gli hazards RAW sono causati da dipendenze intrinseche delle istruzioni. Gli hazards WAR e WAW sono causate da dettagli implementativi (i registri), e pongono solo dei vincoli sullo scheduling.

Il limite principale di questa tecnologia è sicuramente il fatto che la gestione avviene attraverso i registri. Le false dipendenze sono causate proprio dalla numero limitato di registri. Ciò non è tanto dovuto ai limiti fisici legati allo spazio sul silicio su cui realizzarli, quanto più è legato ai bit riservati nella codifica di un'istruzione per specificare un determinato registro sorgente/destinazione. Questo limite emerge soprattutto con i loop (consultare le slides del corso per il chiaro esempio in cui emerge il calo di performance pur assumendo la perfetta predizione del salto).

2.3 L'algoritmo di Tomasulo

Si tratta di un algoritmo inventato dall'informatico di IBM Robert Tomasulo, passato alla storia per aver *inventato* l'algoritmo che ha permesso l'esistenza dei processori che eseguissero istruzioni non in ordine. Gli ingredienti chiave dell'algoritmo sono le **reservation stations**: Le reservation stations sono dei buffer associati alle unità funzionali di un processore out-of-order. Servono a tenere in sospeso le istruzioni in attesa che i loro operandi diventino disponibili. In pratica, quando un'istruzione viene decodificata non entra subito nell'unità funzionale: viene invece “prenotato” uno slot in una reservation station, che contiene il tipo di operazione da eseguire, i riferimenti agli operandi (o i loro valori, se già disponibili) e il registro destinazione. Così, più istruzioni possono essere emesse in parallelo senza bloccare la pipeline, anche se non hanno ancora tutto ciò che serve per l'esecuzione.

L'idea di base dell'algoritmo è schedulare dinamicamente istruzioni in modo che possano iniziare appena gli operandi diventano disponibili, e duplicare i valori degli operandi (ovvero rinominare i registri) per evitare inutili hazards. Un altro elemento chiave è il Common Data Bus (CDB): quando un'unità funzionale calcola un risultato, lo invia sul bus comune, e tutte le reservation stations che aspettavano quel valore possono aggiornarlo immediatamente. In particolare, il CDB comunica la coppia (id, value) dove id è l'identificativo dello slot della RS a cui è stato assegnato il valore.



Rispetto allo scoreboard, le Unit Status tables vengono rimpiazzate dalle Tomasulo Tables (Reservation Stations RS), che in generale possono essere più delle unità funzionali. Queste tabelle vengono assegnate in rapporto n:1 ad un'unità funzionale, e sono divise in slot, uno per ogni riga della singola tabella. I campi della tabella sono:

id	Intero che identifica univocamente lo slot
name	Nome dell'unità funzionale + slot number
busy	Flag che indica se l'unità funzionale è occupata
op	Specifica operazione richiesta all'unità funzionale

st_i	Indice dello slot della RS che fornirà eventualmente il valore dell'i-esimo operando (=0 se il dato è già disponibile e può essere direttamente copiato dai registri)
V_i	Valore dell'i-esimo operando, copiato
Ad	Usato per mantenere l'eventuale offset in caso di indirizzamento base+offset e poi l'indirizzo effettivo (solo nelle operazioni di load/-store).

<i>id</i>	<i>name</i>	busy	op	St₁	V₁	St₂	V₂	Ad
1	Integer Unit (slot 0)							-
2	Integer Unit (slot 1)							-
3	Integer Unit (slot 2)							-
4	FP Add/Sub Unit (slot 0)							-
5	FP Add/Sub Unit (slot 1)							-
6	FP Mult/Div Unit (slot 0)							-
7	FP Mult/Div Unit (slot 1)							-
8	Load-Store Unit (slot 0)							
9	Load-Store Unit (slot 1)							

La register status table invece contiene i campi nome registro e St, ovvero l'indice dello slot della reservation station che fornirà il valore (=0 se già disponibile). Occorre fare un'osservazione sull'unità di load/store (LSU): La RS associata (chiamata anche Load-/Store buffer) mantiene anche i dati da archiviare e l'indirizzo di memoria target. In un primo momento, in caso di indirizzamento base+offset, nel campo aggiuntivo verrà conservato l'offset, e in un secondo momento verrà conservato l'indirizzo effettivo. Osserviamo inoltre che la LSU garantisce l'ordinamento degli accessi in memoria. Questo è cruciale per sequenze di operazioni verso lo stesso indirizzo che includono almeno una Store. La LSU garantisce che le operazioni di lettura/scrittura allo stesso indirizzo accedano in memoria in ordine. Possono essere implementati all'interno della stessa tabella dei *forward paths* nel campo valore, dove nel caso di letture che seguono una scrittura il valore può essere direttamente passato, comparando il campo *effective address*.

Vediamo nel dettaglio come funziona lo schema di Tomasulo:

- Le istruzioni sono recuperate in ordine, poi seguono i passi Issue, Execute, Write Back;
- Un'istruzione viene accettata se un appropriato slot è disponibile nella RS (anche se le unità funzionali sono occupate);
- Una volta accettata, se possibile vengono copiati i dati necessari, oppure si ritiene traccia delle **operazioni** sospese che forniranno il dato più tardi;
- L'esecuzione avviene quando tutti i dati sono disponibili e copiati nei campi V_i ed alla RS;
- Il WB avviene quando il CDB è disponibile, in modo che il risultato può essere trasmesso in broadcast con l'id dello slot che ha generato quel risultato. In questo modo, tutte gli slot in attesa del risultato da quell' ID possano ricevere contemporaneamente il valore.

Notiamo subito che gli hazards strutturali sono ancora possibili nel caso che le tabelle vadano in overflow. Ma costruire RS più capienti è meno costoso di aggiungere unità funzionali.

Il miglioramento di performance, una volta evitati i conflitti dovuti alle false dipendenze, è palese ed è visibile nell'esempio presentato nelle slides del corso.

next step	stalled if:	action performed when unstalled:
Issue	\forall suitable slots j $ST(j).busy=1$	set $ST(id).busy \leftarrow 1$ and $ST(id).op$ properly; $RS(D).St \leftarrow id$ $ST(id).St_1 \leftarrow RS(S_1).St$, if $RS(S_1).St=0$ $ST(id).V_1 \leftarrow Regs[S_1]$ (...the same for S_2, V_2). For Load/Store: $ST(id).Ad \leftarrow imm$ (Note: Loads do not use St_2, V_2 ; Stores do not set $RS(D)$)
Execute	$ST(id).St_1 \neq 0 \vee ST(id).St_2 \neq 0$ (Load/Store: only $ST(id).St_1 \neq 0$) (after that, execution depends on the FU)	(Load/Store: address .Ad is updated from V_1 , if needed) (Load: memory read is performed, no result is generated) Upon completion: keep <i>result</i> before broadcasting until CDB is available
Write-Back	Common Data Bus unavailable (Store: $ST(id).St_2 \neq 0$)	$\forall j: if RS(j).St=id \{ RS(j).St \leftarrow 0, Regs[i] \leftarrow result \}$ $\forall i: if ST(i).St_1=id \{ ST(i).St \leftarrow 0, ST(i).V_1 \leftarrow result \}$ (same for $.St_2, .V_2$) $ST(id).busy \leftarrow 0$ (Store: $MEM[ST(id).Ad] \leftarrow ST(id).V_2$)



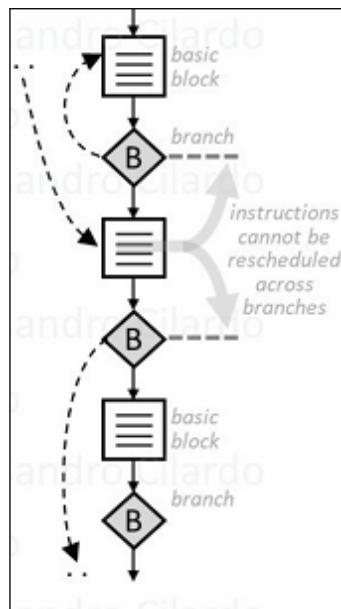
Il miglioramento è ravvisabile soprattutto nei loop ristretti, dove vengono usati in rapida successione gli stessi registri. Qui è lampante la potenza del *renaming* dei registri.



Info: Un'interessante considerazione riguarda il *software loop unrolling*, ovvero una naturale ottimizzazione a livello software della pipeline e dello scheduling dinamico. Di fatto si srotola parzialmente il loop, in modo da rendere i branch meno frequenti ed utilizzare più registri in maniera esplicita. Questa operazione viene fatta dal programmatore o dal compilatore. Questa scelta potrebbe causare la crescita del codice, e di conseguenza potrebbero verificarsi più cache miss in fase di IF (riduzione della località spaziale), e per codici particolarmente complessi potrebbe risultare oneroso rilevare parallelismo, analizzare le dipendenze riguardanti la memoria e risistemare le condizioni di fine loop.

2.4 Dipendenze di controllo

Per massimizzare l'efficienza di un processore devono verificarsi meno *pipe flush* possibili e ridurre al minimo l'inserimento di bolle. Le dipendenze di controllo si riferiscono a situazioni in cui nel codice è presente un salto. Ciò accade molto frequentemente, infatti circa una ogni quattro istruzioni prevede un salto. Questo è problematico perché idealmente il grafo di esecuzione si sdoppia, ed è praticamente impossibile sfruttare il parallelismo intrinseco delle istruzioni non conoscendo a priori quali istruzioni caricare. In sintesi, i salti rendono il flusso dati *dinamico*: le assegnazioni sono condizionali. Idealmente, le dipendenze di controllo possono essere eliminate linearizzando il flusso di esecuzione del codice. Questo si può fare solo sotto l'ipotesi di predizione del salto perfetta e sotto l'ipotesi che tra un branch e l'altro ci siano sufficienti istruzioni.



La predizione del branch a cui saltare però nella pratica non è mai perfetta, e avviene tramite meccanismi hardware (quelli di base sono descritti nel paragrafo [1.2.2]) e software, come l'inserimento di un bit di indizio staticamente inserito nell'istruzione. Osserviamo che poichè il branch predictor *scommette* sulla prossima istruzione da caricare, non fermando mai la pipeline, sono necessari meccanismi di **commit** o **rollback** per non comporomettere in maniera irreversibile lo stato del processore o peggio ancora della memoria.

2.4.1 Generalizzazione predittore a 2 bit

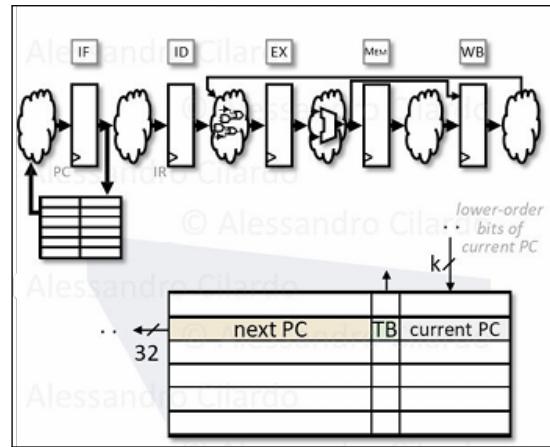
Un predittore dinamico (hardware) a due bit è posizionato a monte della fase issue. Il suo scopo è associare ad ogni istruzione di salto una predizione, ovvero un indirizzo target. Questo avviene consultando una memoria associativa, la cui chiave è un sottoinsieme dei bit dell'istruzione di salto.



Warning: di solito per indicizzare la branch history table si usano i bit meno significativi dell'istruzione di salto. Questo perchè si vuole mantenere contenuta la dimensione spaziale della tabella senza implementare meccanismi di decodifica o di hashing. Ciò significa eventualmente accettare collisioni, ma questo è un problema solo di performance, risolvibile e tollerabile se sufficientemente raro. Infatti i cambiamenti allo stato della memoria e del processore diventano effettivi solo quando la condizione del salto viene calcolata in fasi più avanzate della pipeline.

I predittori a due livelli introducono un'inerzia nel cambio di decisione per massimizzare l'efficienza nel caso di due loop innestati. Questa tecnica può essere generalizzata. Infatti è possibile costruire predittori a n bit, rendendo disponibili 2^n stati per la decisione e potendo arbitrariamente scegliere una tra le transizioni di stato che generano il cambio di decisione.

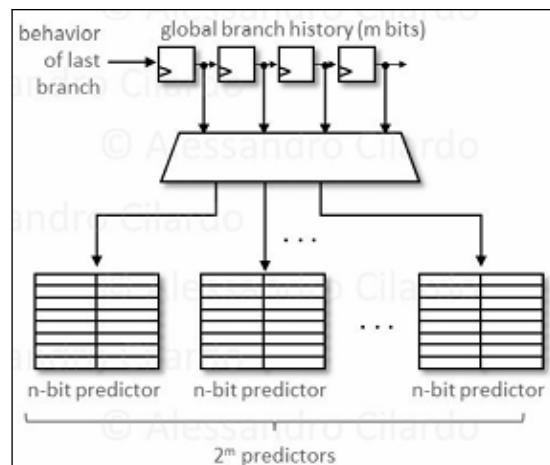
Succede spesso che il comportamento di un branch sia correlato a quello di diversi altri branch presenti nel codice. L'informazione sull'esito dei branch correlati può essere utile



nel determinare la predizione. Si può pensare quindi di rendere la predizione locale dipendente dalla storia *globale* dei salti, ovvero dal comportamento degli ultimi m salti, indipendentemente da dove si trovavano nel codice (per questo globale). È naturale quindi implementare questa *storia* come uno shift register a m bit. Così si costruiscono meccanismi hardware, denominati (m,n) -predittori, che in base al comportamento globale scelgono una delle 2^m branch history tables da consultare. Questo meccanismo è veloce ed efficace, ma richiede diverse iterazioni per andare a regime. Quindi è ragionevole usarlo in caso di molte iterazioni che coinvolgono diversi salti. Osserviamo che questo meccanismo approssima una tecnica di machine learning di *addestramento* per una predizione.



Warning: Il numero di branch history tables da consultare cresce esponenzialmente con m , quindi bisogna cercare un trade off tra efficienza e consumo di risorse.



Alcune tecnologie mirano all'implementazione di un predittore di indirizzo, oltre che ad un predittore di esito del salto. Questo è utile in casi come istruzioni di ritorno da subroutine, in cui il campo *next PC* non è noto staticamente, ma deve essere recuperato nello stack. In alcuni processori moderni viene contemplata l'esistenza di una stack utilizzata soltanto per contenere indirizzi di ritorno, in modo da semplificare e velocizzare il recupero (questa memoria è alla base di molti attacchi di tipo *code injection*).



Info: i problemi di indirizzo determinato dinamicamente possono essere risolti staticamente da software, attraverso le espansioni *inline* delle funzioni, in modo da determinare a tempo di compilazione (sfruttando meccanismi di memoria virtuale) l'indirizzo a cui saltare.

2.4.2 Gestione delle eccezioni

Nel modello di esecuzione out of order, le eccezioni possono occorrere in uno stage qualsiasi della pipeline. Per un richiamo su come vengono gestite le eccezioni in una pipeline semplice a cinque stadi, consultare il paragrafo [1.2.2].

Per quanto riguarda l'approccio speculativo, è possibile permettere alle istruzioni di procedere fino alla fase di WB, anche quando non è sicuro che debbano essere eseguite. Chiaramente serve un meccanismo di ripristino qualora la speculazione fosse sbagliata. La speculazione hardware ha quindi bisogno di una fase *commit* aggiuntiva. Il workflow diventa **issue** (in order) → **OpRead, EXE, WB** (out of order) → **commit** (in order). Le istruzioni possono leggere risultati ancora non committati da istruzioni precedenti, attraverso un buffer temporaneo, non direttamente dai registri.

Una particolare tecnica utilizzata per la speculazione hardware è il **Re-Order Buffer** (ROB). Questo buffer traccia le istruzioni complete ma non committate e mantiene gli operandi di queste istruzioni tra il completamento e il commit. Usato in combinazione con lo schema di Tomasulo, i dati sono bufferizzati anche all'interno delle Reservation stations tra la fase di issue ed execute.

rob #	busy	type	state	V	D
1					
2					
3					
4					
5					
6					
7					
8					

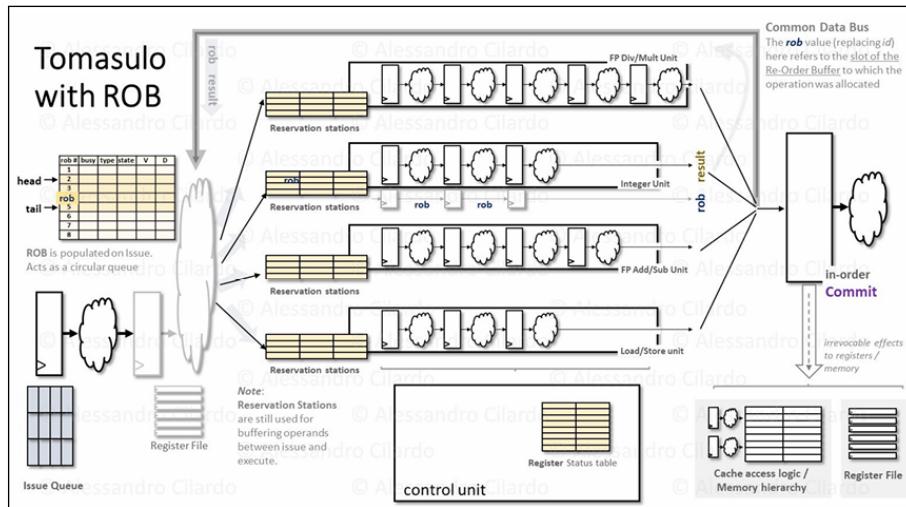
Re-Order Buffer (ROB)

ROB è gestito come una coda circolare, con puntatori per gestire la testa e la coda. I campi del ROB hanno il seguente significato:

rob #	Identifica le operazioni
type	usato per distinguere le operazioni di store e tracciare i salti
state	Flag che indica se il campo V è valido
D	Registro target in scrittura
V	Valore da scrivere nel registro

Le Reservation Station subiscono delle modifiche, in particolare i campi S_i vengono sostituiti dalle entry rob_i , mentre il campo D viene sostituito da D_{rob} . Analogamente la

status register table conterrà due campi per ogni registro (busy, rob) in cui viene indicato se una particolare operazione aspetta di effettuare il commit di una scrittura su quel particolare registro. Il funzionamento dello schema di Tomasulo con ROB è il seguente: Le istruzioni vengono recuperate in ordine, e seguono gli steps Issue, Execute, Write-Back e Commit. Un'istruzione viene accettata se è disponibile un appropriato slot delle Reservation Stations ed è disponibile una ROB entry. Una volta accettata, l'operazione viene inserita nella ROB e viene aggiornata la RS e la Register Status Table. L'esecuzione avviene appena tutti i dati sono disponibili e copiati nei campi V_i . Nella fase WriteBack, attraverso il common data bus il ROB Value viene trasmesso in broadcast, in modo che le RS possano eventualmente accettarlo. Infine, la fase di commit esamina la coda ROB in testa, e se è il caso di predizione errata si effettua il flush della tabella fino alla coda. Altrimenti, si procede con il commit e si fa avanzare il puntatore coda. Osserviamo che sono ancora possibili hazards strutturali, ma costruire RSs e ROB più larghi è meno costoso che implementare più unità funzionali.

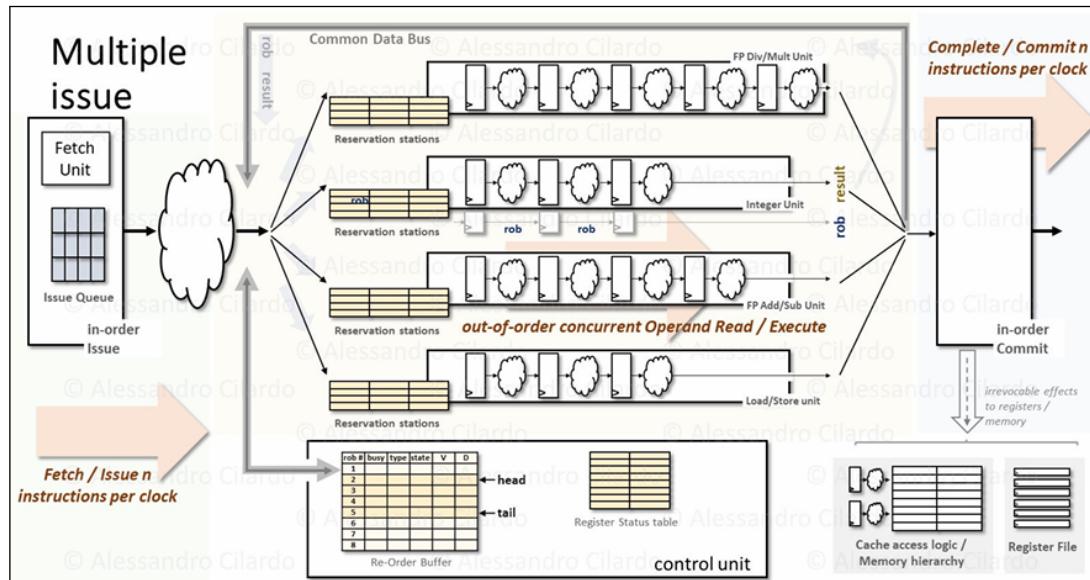


Grazie a questo schema è possibile committare e quindi scrivere più istruzioni per colpo di clock, superando uno dei bottleneck della pipe. Si può pensare di superare anche il bottleneck dovuto al fatto che la pipeline esegue una sola issue alla volta.

2.5 Multiple issue

Il Multiple Issue è la capacità fondamentale delle moderne CPU superscalari di prelevare, decodificare ed accettare più istruzioni contemporaneamente nello stesso ciclo di clock. Per sostenere questo parallelismo, l'architettura interna necessita di un potenziamento significativo: bisogna allargare il bus degli operandi per gestire il trasferimento simultaneo di dati e potenziare la logica di emissione e scheduling delle istruzioni. Il processo è estremamente rapido e prevede l'assegnazione preventiva delle risorse necessarie e una meticolosa analisi delle dipendenze tra tutte le istruzioni candidate all'emissione, al fine di evitare conflitti di dati (RAW, WAW, WAR). L'implementazione hardware è complessa perché le tabelle di stato interne che tracciano l'uso delle risorse (come i registri e le unità funzionali) devono essere aggiornate in parallelo per l'intero bundle di istruzioni e completare il tutto in un solo ciclo di clock. Inoltre, il controllo delle dipendenze tra

più istruzioni richiede un numero massivo di operazioni di confronto eseguite in parallelo, generando un notevole overhead logico (in generale per un blocco di n istruzioni bisogna considerare $O(2^n)$ comparazioni per il controllo dipendenze). È difficile superare le quattro istruzioni emesse per ciclo perché l'aumento della complessità logica e il conseguente ritardo sul clock spesso superano i guadagni di prestazione. Questa soglia è critica anche perché la speculazione (l'esecuzione predittiva) deve continuare a funzionare in modo efficiente; gestire troppe istruzioni speculative in contemporanea aumenta drasticamente il costo di un errore di predizione. Per migliorare l'efficienza, una strategia comune è separare le istruzioni intere da quelle in virgola mobile, permettendo un scheduling più flessibile e l'uso di unità funzionali dedicate. Altre unità di supporto giocano un ruolo cruciale: un'unità di prelievo (fetch) efficiente con un'unità branch integrata per la predizione dei salti e la disponibilità di più linee di cache L1 aiutano a mantenere il flusso di dati e istruzioni. Le sfide aggiuntive includono l'elevato consumo energetico derivante dall'attivazione parallela di molteplici componenti, la penalizzazione causata dai mancati accessi alla cache che possono bloccare interi bundle di istruzioni, e la complessità di gestire più istruzioni di salto speculative nello stesso ciclo di emissione.



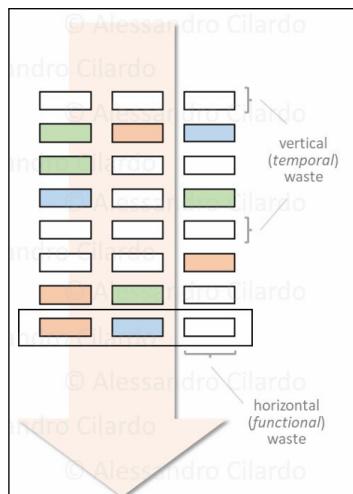
L'implementazione del ROB implica un overhead non indifferente, e in alcune situazioni può risultare opportuno rinunciare ad una gestione *precisa* delle eccezioni. Infatti alcuni processori supportano il modello di esecuzione floating point non preciso, ovvero il comportamento della pipeline non è trasparente alla sequenzialità. Questo modello non preciso ha senso quando effettivamente il calcolo produce NaN, e ciò implica che il calcolo non è andato a buon fine e può essere abortito completamente e rifatto.

Fin qui abbiamo visto che è possibile aumentare l'efficienza di un processore pipelined attraverso l'incremento della frequenza del clock e attraverso la miniaturizzazione delle risorse che ne permette la n-uplicazione. Ma se un numero di transistori (nell'ordine delle decine di miliardi) commutano insieme, dobbiamo considerare problemi in termini di *densità di potenza dissipata*, ovvero il rapporto tra potenza dissipata e superficie su cui si trovano i transistori. La temperatura del dispositivo rappresenta di fatto un limite. Per risolvere questo problema, ci sono due soluzioni: si spengono alcune parti della CPU (problema del Dark Silicon, incompatibile con le architetture superscalari);

oppure si mantiene più bassa la frequenza, in modo che la dissipazione resti costante e il sistema si trovi in un punto di lavoro prossimo al punto critico. Il parallelismo interno è possibile grazie alle tecniche hardware e software fin qui presentate; viene demandata al programmatore, tramite il supporto hardware, l'espressione di ulteriore parallelismo.

2.6 Hardware supported multithreading

I threads hanno PC, stack e registri del processore privati ma condividono la memoria (a differenza dei processi). Nella gestione dei thread non serve duplicare gran parte del processore, perchè le unità funzionali possono essere condivise. Ciò significa che i threads competono realmente per la stessa unità funzionale, e la memoria virtuale tramite i meccanismi di paging on demand fornisce un supporto enorme per gestire la memoria condivisa in termini di efficienza di spazio. Il cambiamento di contesto tra più threads è quasi immediato. Questo permette rapidi cambi di contesto in caso di eventi di stallo (come cache misses) per nascondere lunghe latenze. In questo caso stiamo parlando di Thread level parallelism (TLP), ed è gestito esplicitamente dal programmatore o dal compilatore. Possiamo identificare, in termini di sprechi di risorse del processore, *sprechi verticali* e *sprechi orizzontali*.



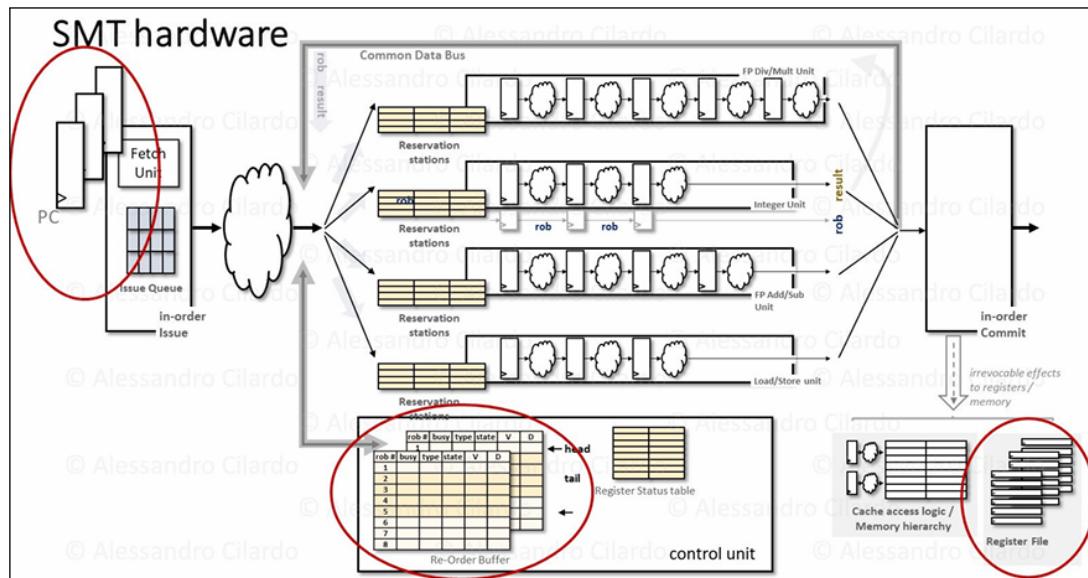
Il meccanismo del multithreading simultaneo conviene perchè è possibile aumentare l'ampiezza della finestra di istruzioni recuperata (poichè i thread utilizzano istruzioni indipendenti) invece che la profondità. Si possono recuperare e accettare n istruzioni per colpo di clock, l'esecuzione è out of order e poi si possono committare fino a n istruzioni per clock. Questo risolve il problema della multiple issue. Vediamo con ordine le varie tipologie di hardware multithreading:

- **Coarse-grain:** viene effettuato il cambio di contesto solo su eventi di attesa prolungata (come cache miss), e in generale su eventi che provocano sprechi verticali. Questa soluzione risulta inefficace per nascondere brevi latenze;
- **Fine-grain:** ad ogni colpo di clock, secondo una politica Round Robin, viene effettuato il cambio di contesto. Questa scelta è ottima perchè nasconde anche le latenze brevi e rilassa gli hazards RAW (basti pensare al fatto che vengono interacciate istruzioni indipendenti per definizione). Questo approccio è utilizzato nelle GPU e in processori con elevato numero di thread per core;

- **Simultaneous multithreading:** tecnica che combina fine grain multithreading con issue multipla e scheduling dinamico. Le tecniche di *register renaming* permettono a istruzioni di thread indipendenti di essere eseguito in maniera simultanea.

2.6.1 Simultaneous multithreading

Il supporto al SMT può essere implementato *on top* all'architettura pipeline vista in questo capitolo, con alcune accortezze. Il supporto al multithreading in un processore out-of-order si ottiene mediante la replicazione di alcune strutture architettoniche fondamentali, così da garantire che ciascun thread disponga di un contesto di esecuzione autonomo. Ogni thread possiede una propria tabella di rinominazione, che costituisce il meccanismo essenziale per risolvere le dipendenze sui registri architettonici. Tale tabella associa dinamicamente i registri logici, visibili al programmatore, a registri fisici interni al processore, permettendo a thread distinti di utilizzare gli stessi identificatori senza generare conflitti. In parallelo, ogni thread mantiene un proprio Program Counter, ossia il registro che memorizza l'indirizzo dell'istruzione successiva da eseguire, il che consente al processore di prelevare istruzioni da flussi di controllo differenti in modo indipendente. Grazie a questa duplicazione delle strutture di contesto, il processore è in grado di mantenere contemporaneamente più sequenze di istruzioni pronte per l'esecuzione. L'unità di fetch può selezionare istruzioni provenienti da thread distinti e inserirle nello stesso flusso di pipeline, mentre lo scheduling out-of-order consente di collocarle dinamicamente sulle unità funzionali disponibili, sfruttando eventuali slot altrimenti inutilizzati. In tal modo, l'esecuzione multithreaded non si riduce a una semplice alternanza temporale tra thread, ma assume la forma di una coesistenza effettiva all'interno della microarchitettura, con istruzioni appartenenti a thread diversi che avanzano in parallelo nel pipeline. L'obiettivo è incrementare il throughput globale del processore e ridurre le bolle nella pipeline, migliorando l'utilizzo delle risorse computazionali senza richiedere un incremento proporzionale del numero di core fisici.



Questa tecnologia hardware necessita di *estendere* l'insieme di registri del processore. Infatti in un contesto multithread dobbiamo considerare che ogni thread necessita di un

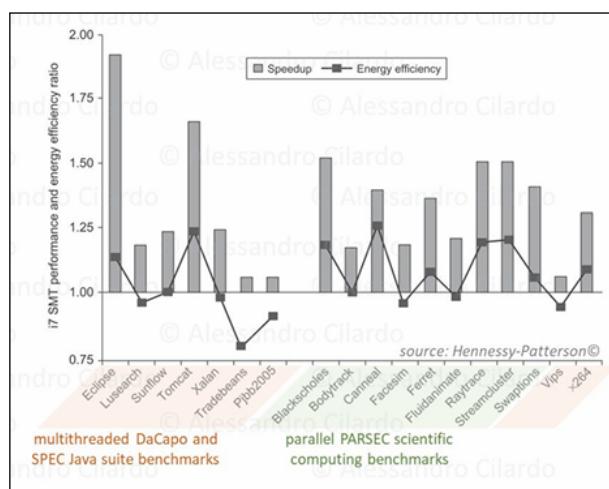
certo numero di registri per funzionare. In fase di esecuzione sono presenti *operazioni* proveniente da istruzioni di più threads, i cui operandi sono letti e scritti da registri del processore. Questo implica una n-uplicazione dei registri del register file, e aggiungendo dei bit in più alla chiave della tabella che mantiene l'associazione nome registro → indirizzo, in modo da scriverci, in fase di issue quindi in modo totalmente trasparente all'esecuzione, l'ID del thread che sta lavorando su quel registro, e in modo che l'indirizzo del registro fisico sia univoco, nonostante più thread possano usare lo stesso registro logico.

Quindi, anche se avviene un context switch “software” (OS), non è necessario che il processore hardware ricrei da zero ogni volta tutto il contesto dei registri se il design architetturale ha già riservato contesti separati per ogni thread logico attivo. Il vantaggio è che il passaggio fra thread hardware (SMT) può avvenire con latenza minima, senza il costo elevato di salvataggio su memoria e ricaricamento per ogni piccola alternanza tra thread. Solo quando un thread viene sospeso a livello di OS (non più residente nell'hardware) si fa il salvataggio completo dello stato. Una delle principali complessità nella realizzazione del multithreading simultaneo riguarda la fase di commit delle istruzioni. Affinché ciascun thread possa completare correttamente e in modo indipendente la propria sequenza di istruzioni, è necessario che il processore mantenga strutture di riordino separate, spesso implementate come Reorder Buffer distinti per ogni contesto hardware. In questo modo, il ritiro delle istruzioni avviene in ordine per ciascun thread, evitando che un errore o un'eccezione in un thread influenzi la correttezza di un altro. Un'ulteriore difficoltà deriva dalla gestione della memoria condivisa: la coesistenza di thread multipli aumenta la probabilità di conflitti di cache e di fenomeni di thrashing, riducendo l'efficacia della gerarchia di memoria e degradando le prestazioni soprattutto nelle applicazioni caratterizzate da accessi intensivi e poco localizzati.

2.6.2 SMT performance

La tecnologia analizzata finora permette all'utente (di solito il sistema operativo) di vedere più **core virtuali** (o logici) per ogni core fisico. Nelle attuali implementazioni, l'hardware di supporto al SMT è in grado di mantenere residenti, contemporaneamente, i contesti architetturali di quattro thread diversi, ciascuno con il proprio Program Counter, i propri registri architetturali e le proprie strutture di rinominazione. In altre parole, il processore può tenere “attivi” più flussi di istruzioni, pronti a essere eseguiti. Tuttavia, il processore può prelevare e inviare istruzioni all'esecuzione solo da un thread alla volta. Ciò implica che, pur avendo più thread residenti, la fase di fetch e dispatch è monopolizzata da un singolo thread in ciascun ciclo di clock. L'unità di fetch, il decodificatore e il meccanismo di issue non combinano istruzioni provenienti da thread differenti nello stesso ciclo, ma scelgono uno dei thread disponibili e prelevano le istruzioni esclusivamente da quello. Questa è una limitazione rispetto all'SMT più “aggressivo”, dove il processore è in grado di mescolare nello stesso ciclo istruzioni provenienti da thread diversi, riempiendo le unità funzionali libere con qualunque istruzione pronta, indipendentemente dal thread di origine.

Dal grafico emerge che l'SMT può migliorare le prestazioni in alcuni casi, ma l'efficienza in termini energetici varia a seconda dell'applicazione. L'idea chiave è che SMT funziona meglio per alcune applicazioni rispetto ad altre, e l'efficienza energetica può essere un fattore determinante.



Chapter 3

Coerenza e consistenza della memoria

I limiti espressi nel capitolo precedente in termini di efficienza energetica hanno, nel tempo, imposto la tendenza dell'industria produttrice di processori la tendenza a progettare sistemi multi/many core invece che singoli processori ad alte performance. I sistemi **multicore** sono composti da 2 fino 8 nuclei di fascia medio-alta, mentre i sistemi **manycore** sono composti da 8 fino a 16 nuclei di fascia medio-bassa. Nelle architetture manycore l'enfasi è posta sulle *interconnessioni* e sulle *architetture di memoria*. Gli accessi in memoria possono **uniformi** (UMA), come ad esempio in RAM dove il tempo in cui viene effettuato l'accesso al dato non dipende dal punto in cui vi si accede, oppure **non uniformi** (NUMA), durante il quale il tempo di accesso ad uno spazio di indirizzamento logicamente unico dipende però dal punto in cui si accede.

Gli accessi in memoria possono essere indipendenti gli uni dagli altri, e può succedere che il processore o la Load/Store unit invertano, per massimizzare l'efficienza, l'ordine di queste operazioni. In particolare queste operazioni possono essere invertite nel caso operino su indirizzi diversi, in modo da non generare interferenze. Questo funziona su un singolo core, mentre nelle architetture manycore che sfruttando la gerarchizzazione della memoria possono verificarsi delle complicazioni. Lo scheduling dinamico, a livello core, può cambiare l'ordine di operazioni di Load/Store che sono viste indipendenti *localmente*, mentre potrebbero non esserlo a globalmente. Un esempio di problema di coerenza potrebbe essere il caso di due processi, produttore e consumatore, in esecuzione su due core diversi, che usano la primitiva forma di sincronizzazione di set/lettura di un flag condiviso. Il produttore performa in memoria le operazioni di scrittura dato e scrittura flag, mentre il consumatore performa in memoria le operazioni di lettura flag e lettura dato; se l'ordine di una di queste operazioni fosse invertito, si violerebbe la consistenza della memoria, e ciò condurrebbe ad una violazione del principio di causa/-effetto.

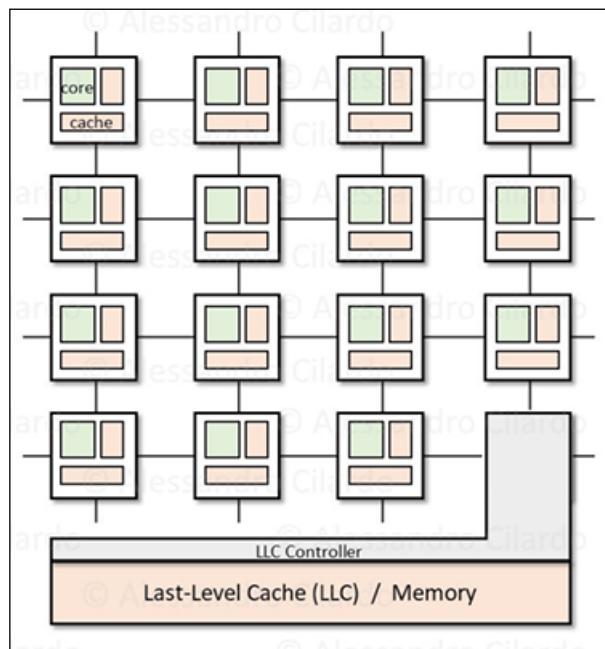
Per quanto riguarda l'unità di Load/Store, abbiamo visto che è possibile applicare la tecnica del *bypass locale*, che permette ad operazioni di store di alimentare operazioni di load conseguenti a livello hardware. È possibile applicare tecniche come **Address Alias Prediction**, tramite la quale predire se due operazioni con indirizzi effettivi ancora non risolti possano andare in conflitto, e riordinare speculativamente le operazioni, o tecniche come il **Memory coalescing**, tramite il quale compattare più operazioni di Load/Store in una singola operazione fisica, accedendo in una volta sola a più blocchi di memoria.

Un'ulteriore fonte di complessità deriva dal modello architetturale di memorie distribuite

ed interconnesse. L'interconnessione avviene tramite protocolli bus più o meno complessi: **bus atomici** consentono la soluzione più semplice, rispettando l'ordine stretto delle transazioni e non consentendo la sovrapposizione; **bus pipelined** consentono la sovrapposizione delle transazioni (*richieste*), garantendo però che le *risposte* arrivino nell'ordine giusto; **bus split-transaction**, consentono di invertire l'ordine delle risposte, mantenendo però l'informazione riguardo l'ID della richiesta. Le complicazioni possono aumentare nel caso di interconnessioni più sofisticate (*networks on chip*).

3.1 Architettura cache

La gerarchia di memoria consiste in cache L1 ed L2 collocate in prossimità del core e controllate localmente dal core. La cache L2 può essere privata, o condivisa tra più nodi. L'ultimo livello di cache (LLC) ha un controllore integrato, che funge da interfaccia con gli elementi di ordine superiore. La memoria principale si trova oltre l'LLC.

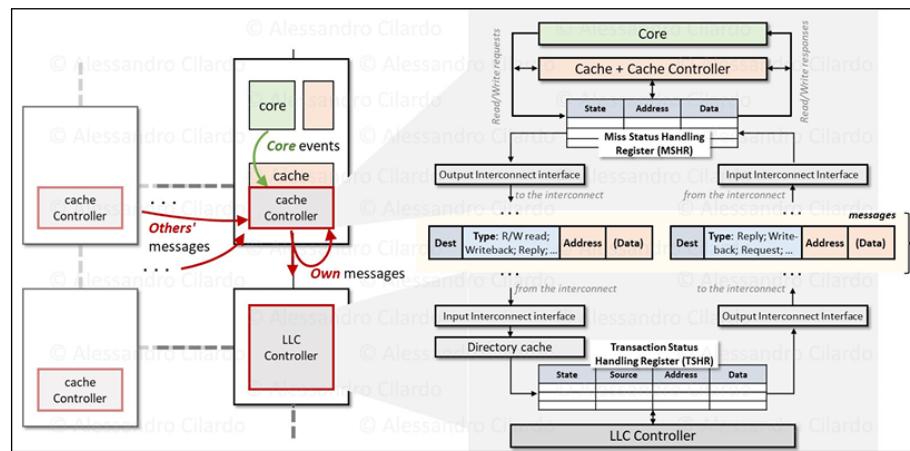


Info: Il memory controller che interfaccia LLC e archittettura di livello superiore (cache più veloci) si può trovare in un nodo separato oppure in uno dei nodi core. LLC è l'ultimo livello in cui ci si deve preoccupare della coerenza, perché è l'unico componente che dialoga con la memoria off-chip.

D'ora in avanti faremo riferimento alla terminologia e alle architetture presentate nella sezione [1.4]. Ogni blocco di memoria in cache è associato ad uno stato, che va opportunamente gestito. Nelle caches e LLC, si estende lo stato associato ad un blocco di qualche bit, e di solito è necessario gestire solo gli stati *stabili* (gli stati *transienti* sono associati a transizioni sospese); Nella memoria principale, notiamo che l'LLC è visto come un insieme di caches locali, e quindi la memoria principale non ha bisogno di stati

esplicativi: se un blocco non è nell'LLC, il suo stato in memoria è automaticamente *invalido*, ovvero nessuna cache attualmente lo mantiene.

Concettualmente, ad ogni blocco è associata una macchina a stati finiti. Lo stato deve essere conservato per tutti i blocchi *cached* lato core, e per tutti i blocchi lato memoria. L'informazione conservata coinvolge solo stati permanenti per quei blocchi che non sono affetti da transazioni. Altrimenti, è necessario conservare informazioni sui *transienti*, che registrano il corrente stato di una transazione. Si usa il MSHR (Miss Status Handling Register) lato core e i TSHRs (transaction Status Handling Registers) lato memoria.



Info: Le memorie cache possono anche essere progettate come non coerenti, come nel caso delle **ScratchPad Memories**: Le scratch pad memories sono memorie veloci di piccole dimensioni integrate all'interno di un processore, pensate per fornire un'area di lavoro temporanea molto più rapida rispetto alla RAM principale. Vengono spesso usate per conservare dati intermedi, variabili critiche o risultati parziali di calcolo che devono essere accessibili con latenze minime. A differenza delle cache, non hanno logiche di gestione automatica della coerenza, ma devono essere programmate esplicitamente: è il programmatore o il compilatore a decidere quali dati caricare e scaricare. Questo implica hardware semplice ma software complesso. Infatti queste memorie hanno uno spazio di indirizzamento autonomo e necessitano di istruzioni speciali di load/store.

Per applicazioni general purpose è preferibile utilizzare caching trasparente.

3.2 Protocolli di coerenza

La coerenza può essere definita formalmente tramite un appropriato invarianto, scelto equivalentemente tra:

- **SWMR** (Single Writer Multiple Reader): Durante una data epoca, o c'è un singolo core che può leggere o scrivere la locazione A, oppure un numero di cores che possono solo leggerla.
- **DV** (Data Value): Il valore di A all'inizio di un'epoca è lo stesso valore di A alla fine della sua ultima epoca di lettura/scrittura.

Altre opzioni per i protocolli di coerenza sono operazioni di **Update**, di fatto transazioni che aggiornano tutte le copie nelle altre cache su una scrittura, o **Invalidate**, transazioni che invalidano tutte le altre copie su una scrittura.

Il protocollo determina la macchina a stati finiti nelle cache e nei controller. Gli input alla FSM della cache sono eventi del core (istruzioni di Load/Store, *eviction* di un blocco) e eventi di interconnessione (messaggi *own* e *other's* accettati dall'interconnessione); gli input alla FSM dell'LLC controller sono solo gli eventi di interconnessione. Gli eventi di interconnessione consistono nei messaggi provenienti da altri core diretti al "nostro" core o al memory controller, e nei messaggi *own*: ricevere un messaggio *own* significa che l'interconnessione ha accettato il messaggio (altrimenti il messaggio risulterebbe *pending* sull'interfaccia di interconnessione).

States	Core events		Interconnect events					
	Load/Store	Evict	Own messages			Others' messages		
		OwnGet	OwnDataResp	OwnPut	OtherGet	OtherDataResp	OtherPut	
Transient								
Normal								
	as a convention, darker background will be used for transient states		Note: We will borrow the notation and examples from the textbook:					
			D. J. Sorin, M. D. Hill, and D. A. Wood, <i>A Primer on Memory Consistency and Cache Coherence</i> , Morgan Claypool 2011					

Info: Una macchina a stati finiti funziona descrivendo un sistema che può trovarsi in un insieme limitato di stati e che cambia stato in base agli ingressi ricevuti. In ogni istante la macchina si trova in uno stato ben preciso, detto stato corrente. Quando riceve un input, consulta le regole di transizione per capire in quale nuovo stato deve andare. A seconda del modello, la macchina può anche produrre un output che dipende solo dallo stato (nel caso di una FSM di Moore) oppure dallo stato e dall'input ricevuto (nel caso di una FSM di Mealy). Questo meccanismo rende le FSM particolarmente utili per descrivere sequenze di eventi o sistemi di controllo. Per rappresentarla in una tabella si usa una struttura che mette in relazione gli stati correnti con gli ingressi e specifica quale sarà lo stato successivo e, se previsto, l'output. Una riga della tabella corrisponde a una coppia formata dallo stato corrente e dall'input, mentre le colonne riportano lo stato verso cui si passa e l'eventuale valore di uscita. In questo modo, la tabella diventa una mappa compatta e leggibile che descrive completamente il comportamento della macchina.

Un messaggio di tipo **Get** rappresenta una *richiesta di lettura o allocazione* di un blocco di memoria. Esso viene tipicamente generato da un core quando si verifica un *cache miss*, al fine di occupare una linea della cache con il blocco richiesto. L'informazione relativa al tipo di richiesta è codificata nel campo *type* del messaggio. Quando un core osserva sull'interconnessione un messaggio di tipo Get emesso da un altro core, esso viene interpretato come **other Get**. In questo caso, il controller deve verificare se il blocco richiesto è eventualmente presente nella propria cache, per rispondere o per aggiornare lo stato del blocco secondo il protocollo di coerenza adottato.

Il messaggio di tipo **Put**, invece, è un'operazione di *invio proattivo* di un blocco dalla cache verso il livello inferiore della gerarchia. Ciò avviene tipicamente in caso di *eviction*, ossia

quando una linea di cache deve essere liberata. In modo analogo al caso precedente, se un core osserva sull'interconnessione un messaggio di tipo Put proveniente da un altro core, questo viene classificato come **other Put**.

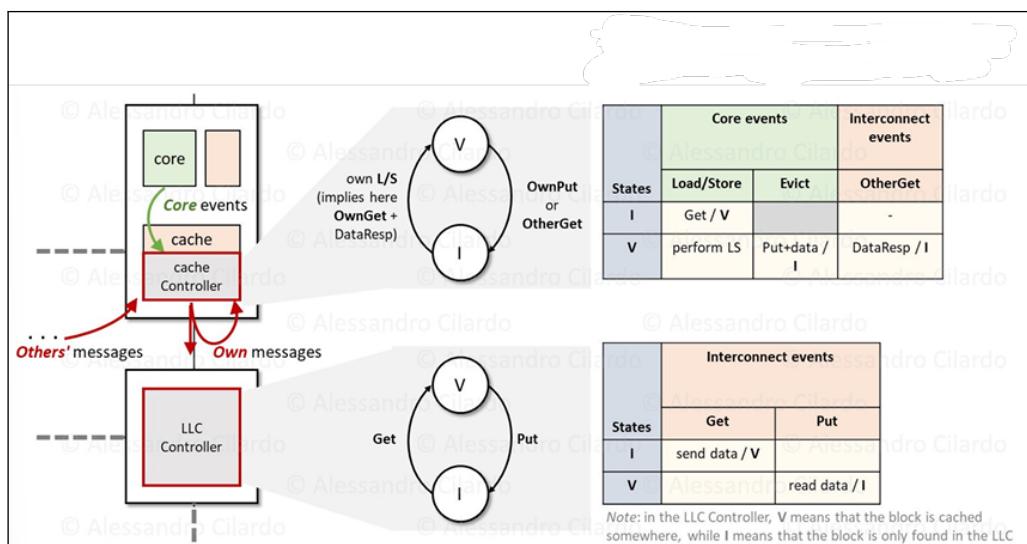
È importante notare che l'emissione di una richiesta da parte di un cache controller non implica necessariamente la sua immediata accettazione: l'interconnessione può infatti trovarsi in stato *busy*. Per questo motivo, il controller mantiene visibilità delle proprie richieste sotto forma di **own messages**, che fungono da meccanismo di *acknowledgment* da parte dell'interconnessione, consentendo di sincronizzare la generazione e l'elaborazione dei messaggi.

3.2.1 Protocollo a due stati con Writeback

In questo protocollo basilare entrambe le FSM hanno due stati, V (Valid) e I (Invalid). Per le FSM delle cache, V significa che il blocco è disponibile nella cache per lettura/scrittura, mentre I significa che il blocco non è presente in cache. Per l'FSM dell'LLC, V significa che il blocco è in qualche cache di livello superiore, I significa che è disponibile solo nell'LLC. Questo protocollo rispetta gli invarianti di coerenza, ma non supporta molteplici repliche dei dati in nodi diversi quando sono usati solo per operazioni di lettura.



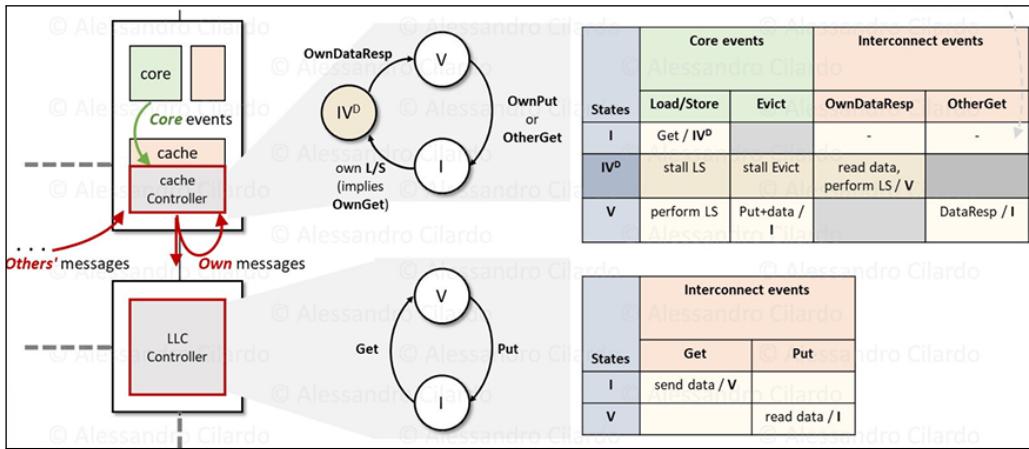
Warning: Lavoriamo sotto l'ipotesi che tutti i messaggi sono inviati in broadcast a tutti i controller e sono visti da tutti nello stesso ordine. Un'ulteriore ipotesi è che gli eventi interni e di interconnessione sono *simultanei* (Load miss → OwnGet + DataResp).



Tipicamente l'interconnessione supporta messaggi di controllo e dato separati, perché i dati hanno bisogno di risorse di interconnessione (bus) dedicati e impiegano molto più tempo rispetto alle informazioni di controllo. Di conseguenza, un'interazione di richiesta/risposta deve necessariamente essere gestita da due messaggi separati: Il messaggio di richiesta risposta e il messaggio contenente i dati. È necessario introdurre degli stati *transienti*, per gestire la situazione in cui un messaggio di controllo è arrivato, ma non sono arrivati i dati.

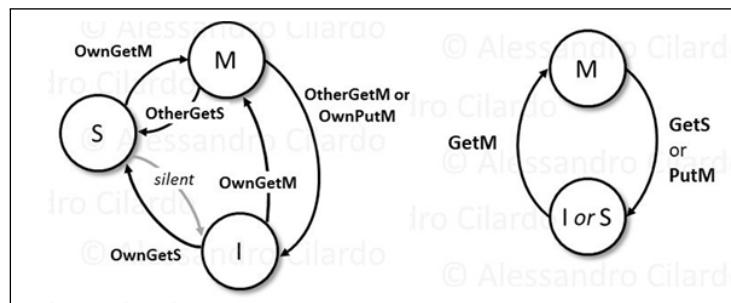


Warning: Per ora assumiamo che se un core sta aspettando i dati in uno stato transiente, non può intervenire una get proveniente da un altro core.



3.2.2 Protocollo a tre stati

Per permettere la lettura simultanea di dati condivisi, mantenendo l'invariante SWMR, introduciamo il protocollo a tre stati **Modified-Shared-Invalid**. Questo protocollo permette di avere repliche dei dati in cache diverse, tranne per il caso in cui un core vuole scrivere il dato. Dal punto di vista dei core, è necessario distinguere richieste di blocchi che saranno solo letti e richiesti di blocchi che invece dovranno essere scritti. Questo implica la distinzione tra messaggi *getS* (Get Shared) e *getM* (Get Modified). Analogamente servirebbe una distinzione tra i messaggi *putM* e *putS*, ma *putS* in questo caso è superfluo in quanto per l'eviction di un blocco in sola lettura non è necessaria comunicazione (*eviction silente*).



Dal punto di vista dell'LLC, è interessante tracciare solo due stati: o una linea può essere modificata da qualche core oppure *indifferentemente* o qualcuno può solo leggerla o ce l'ha solo LLC.

3.2.3 Lo stato Exclusive

È possibile espandere il numero di stati possibili per gestire al meglio determinate situazioni ricorrenti. Lo stato **Exclusive** serve a gestire i casi di Load → Store. Questo stato

States	Core events			Interconnect events							
	Load	Store	Evict	Own messages			Others' messages				
I	GetS / IS ^D	GetM / IM ^D		OwnGetS	OwnGetM	OwnPutM	OwnData	OtherGetS	OtherGetM	OtherPutM	
IS ^D	stall	stall	stall				read data / S	(A)	(A)	(A)	
IM ^D	stall	stall	stall				read data / M	(A)	(A)	(A)	
S	(hit)	GetM / SM ^D	- / I					-	- / I		
SM ^D	(hit)	stall	stall				read data / M	(A)	(A)	(A)	
M	(hit)	(hit)	PutM, send data / I					send Data / S	send Data / I		

Impossibile per l'ipotesi di richieste atomiche: i messaggi own sono visti immediatamente.

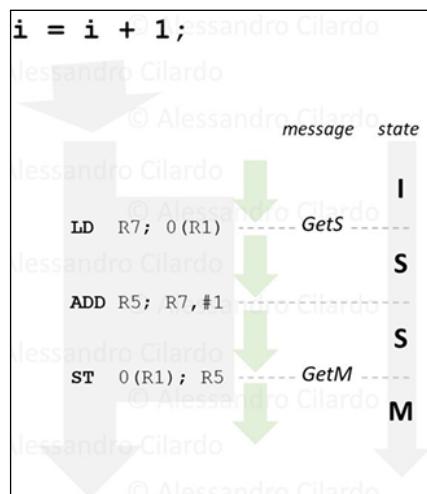
impossibile per l'ipotesi di transazioni atomiche: le transazioni non possono sovrapporsi

States	Interconnect events			
	GetS	GetM	PutM	Data
IorS	send Data / IorS	send Data / M		
IorS ^D	(A)	(A)		read Data / IorS
M	- / IorS ^D		- / IorS ^D	

è usato in quasi tutti i protocolli di coerenza in commercio, in particolare nei processori ARM. Questo stato ottimizza il caso in cui un core inizialmente legge un blocco, e poi lo sovrascrive. Con il protocollo a tre stati questa situazione è descritta dalla sequenza load miss → GetS → Write Miss → GetM. Aggiungendo lo stato Exclusive, il controller può *silenziosamente* aggiornare lo stato della linea da E → M, ma chiaramente solo se è l'unico core ad avere in cache attualmente la linea.



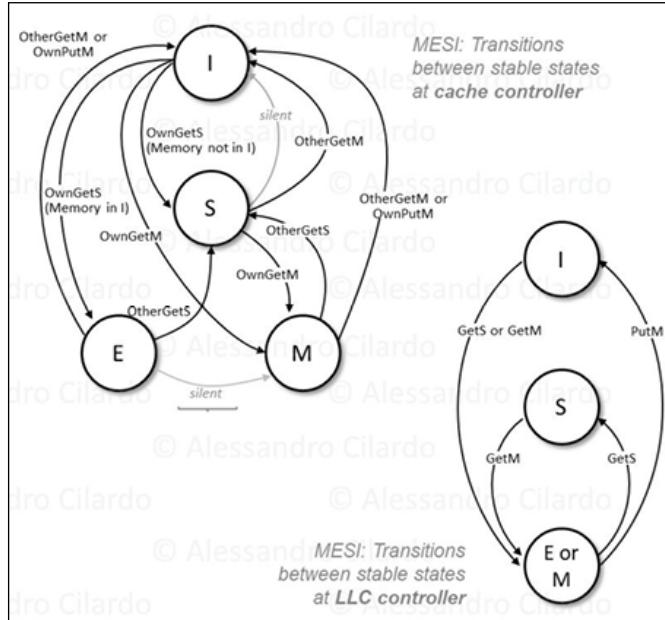
Warning: In realtà al posto di GetM andrebbe usato *Update*, perché è inutile trasportare i dati di nuovo dopo la GetS.



È possibile implementare questo protocollo facendo in modo che il controller LLC abbia traccia di quante caches abbiamo il blocco nello stato S, attraverso un contatore e ricevendo i messaggi PutS, usati dalle cache quando vogliono fare un'operazione di evict

che fino a questo momento era rimasta silente. In questo modo il controller LLC può tenere traccia di quanti *sharers* esistono, ma al contempo questo aumenta notevolmente il traffico di messaggi per la gestione della coerenza.

Esiste una versione *imperfetta* ma meno costosa, che consta di un approccio conservativo. Il controller LLC setta lo stato E di una linea su una prima richiesta di accesso, e nei successivi accessi lo stato muta E → S, e non torna mai più allo stato E, anche quando resta solo uno sharer.



3.2.4 Lo stato Owned

È possibile introdurre lo stato owner: un **Owner** è la cache che dispone della versione più aggiornata di una linea. Una cache in stato O deve rispondere alle richieste di altre cache per quel blocco, sollevando il controller LLC da questo onere. Quando una cache ha un blocco nello stato M oppure E, e riceve una GetS da un altro core, nel protocollo MSI, la cache deve rispondere mandando il dato al richiedente e al controller LLC, delegando la *ownership* all'LLC. Il protocollo che include lo stato owned elimina il passaggio per l'LLC.

3.2.5 Recap Stati di coerenza

Dei protocolli visti finora possiamo sintetizzare i vari stati:

- **Modified:** Il blocco è valido, exclusive, owned, e potenzialmente sporco;
- **Shared:** Il blocco è valido ma non esclusivo, pulito e non owned;
- **Invalid:** Il blocco è invalido;
- **Owned:** Il blocco è valido, owned, potenzialmente sporco ma non exclusive;
- **Exclusive:** Il blocco è valido, exclusive e pulito;

Transaction	Goal of Requestor
GetShared (GetS)	obtain block in Shared (read-only) state
GetModified (GetM)	obtain block in Modified (read-write) state
Upgrade (Upg)	upgrade block state from read-only (Shared or Owned) to read-write (Modified); Upg (unlike GetM) does not require data to be sent to requestor
PutShared (PutS)	evict block in Shared state
PutExclusive (PutE)	evict block in Exclusive state
PutOwned (PutO)	evict block in Owned state
PutModified (PutM)	evict block in Modified state
Load	if cache hit, respond with data from cache; else initiate GetS transaction
Store	if cache hit in state E or M , write data into cache; else initiate GetM or Upg transaction
Atomic read-modify-write	if cache hit in state E or M , atomically execute read-modify-write semantics; else initiate GetM or Upg transaction
Instruction fetch	if cache hit (in L-cache), respond with instruction from cache; else initiate GetS transaction
Read-only prefetch	if cache hit, ignore; else may optionally initiate GetS transaction
Read-write prefetch	if cache hit in state M , ignore; else may optionally initiate GetM or Upg transactions
Replacement	depending on state of block, initiate PutS , PutE , PutO , or PutM transaction

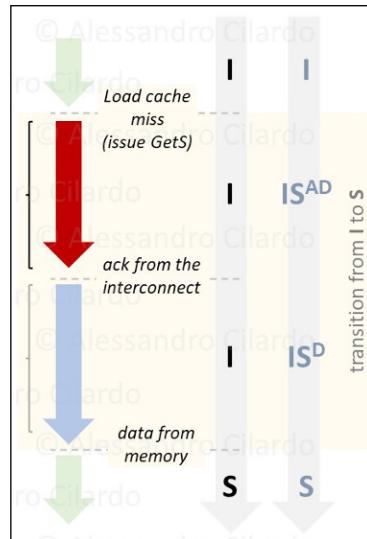
3.2.6 Rimozione delle ipotesi

Nella trattazione fin qui presentata, sono state fatte delle ipotesi semplificative, che possono essere rimosse complicando il protocollo. Le ipotesi finora assunte sono:

- **Richieste atomiche:** Ogni richiesta è accettata dall’interconnessione e notificata al sistema appena viene emessa dal core, non sussiste nessun intervallo di tempo tra l’evento interno e la comparsa del corrispondente messaggio sull’interconnessione;
- **Transazioni atomiche:** Le transazioni non possono sovrapporsi. Una volta che un messaggio di inizializzazione transazione viene accettato, nessun altro messaggio di transazione può essere accettato prima del trasferimento dei dati che completa la precedente transazione. La transazione non accettata risulta in uno stallo dell’evento nel core che l’ha generata;
- **Arbitraggio centralizzato:** Tutti i messaggi sono trasmessi in broadcast a tutti i controller e sono visti *nello stesso ordine*. Questo ordine è garantito dell’interconnessione, che agisce da arbitro centralizzato e determina la consistente sequenza di eventi vista da tutti i controllori.

La rimozione di queste ipotesi introduce protocolli più sofisticati che contemplano più stati transienti. Iniziamo rimuovendo l’ipotesi di **Richieste atomiche**. È necessario adesso considerare una coda di messaggi tra il controller della cache e il bus, in modo che ci sia un delay temporale tra il momento in cui una richiesta è emessa dal controller e il momento in cui viene *ordinata*. Questo comporta un gran numero di stati transienti aggiuntivi. Ad esempio, è necessario introdurre un nuovo stato che contempi l’attesa dell’ACK per una richiesta emessa da parte dell’interconnessione, secondo la traiula $I \rightarrow IS^{AD} \rightarrow IS^D \rightarrow S$. Il passaggio tra il penultimo stato e l’ultimo stato è semplice in quanto stiamo ancora facendo l’ipotesi di transazioni atomiche.

È interessante osservare che nella notazione introdotta per gli stati, per gli stati transienti il primo stato a comparire della coppia è lo stato in cui *effettivamente* il blocco risulta ancora essere nonostante la transizione, e il sistema si comporta di conseguenza. Rimuoviamo ora l’ipotesi di **arbitraggio centralizzato**.



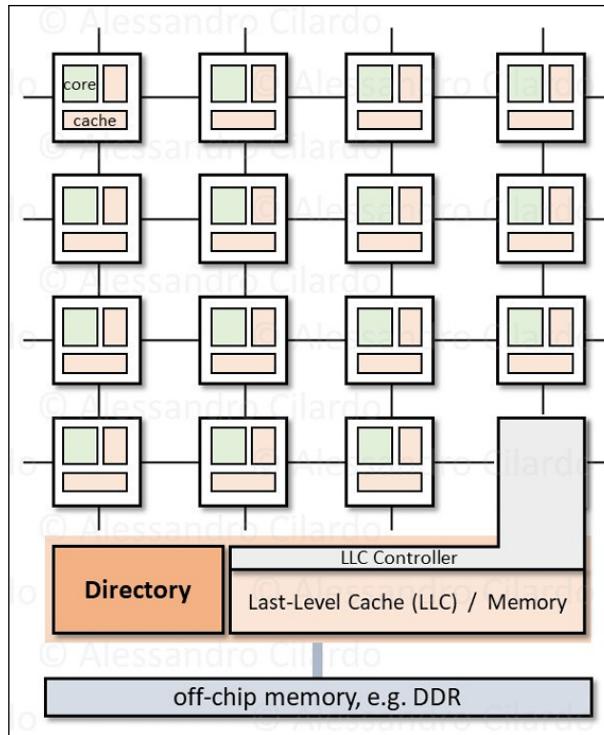
Info: Lo snooping è una tecnica in cui tutte le cache collegate a un bus condiviso monitorano costantemente le transazioni che vi transitano, con lo scopo di mantenere la coerenza dei dati. Ogni volta che un processore esegue un'operazione di lettura o scrittura in memoria, tale operazione viene propagata sul bus e tutte le cache osservano (snoop) l'evento per verificare se possiedono una copia della linea di cache coinvolta. In base allo stato della linea di cache, determinato dal protocollo adottato, ciascuna cache può invalidare la propria copia se un altro processore ha scritto su quella linea, fornire i dati direttamente al richiedente se possiede la copia aggiornata, oppure aggiornare la linea secondo le regole del protocollo. In questo modo, lo snooping assicura che tutte le copie di una stessa cella di memoria distribuite nelle diverse cache rimangano consistenti, evitando che un processore lavori su dati obsoleti.

Nei protocolli basati sullo snooping tutte le richieste da parte di un controller vengono trasmesse in broadcast a tutti gli altri controllori, incluso se stesso. Ipotesi fondamentale per questa modalità è che la rete di interconnessione garantisca un ordinamento completo. Il bus condiviso agisce da *ordinatore*, in particolare per le operazioni di scrittura. Lo scopo è mantenere l'invariante SWMR. I vincoli dell'ordine si applicano soprattutto alle richieste, poiché è necessario che sia ordinata la richiesta trasmessa in broadcast, mentre la risposta unicast arriva dopo solo al richiedente. I messaggi DATA possono viaggiare su una rete separata. Nonostante ciò, i protocolli basati su *snooping* non scalano bene. Infatti su una macchina composta da N nodi, ci saranno almeno N interazioni per ogni cache miss, e questo implica un traffico non sostenibile a livello di latenza sul bus che agisce da collo di bottiglia per sistemi con molti nodi.

3.2.7 Directory

Soluzione alternativa ai protocoli basati sullo snooping maggiornemente scalabile. La **Directory** è di fatto un componente del sistema che tiene traccia globalmente dello stato di coerenza di tutti i blocchi. Questo registro viene consultato per scoprire informazioni su un blocco nelle altre caches, dove si trova anche la corrente locazione delle copie.

Il grande vantaggio di questa soluzione è che queste informazioni sono scambiante in modalità unicast ovvero point2point. La directory inoltre è logicamente unificata, ma in pratica può essere distribuita su più nodi per aumentare la scalabilità.



Di base il protocollo basato di directory prevede che le singola caches mandino tutte le richieste ad una singola directory, che risponde alla richiesta o *inoltra* il messaggio ad altri controllers per poi rispondere. Inoltre, la directory e talvolta i controller richiedenti, devono conoscere il numero di *sharer* che condividono un blocco.



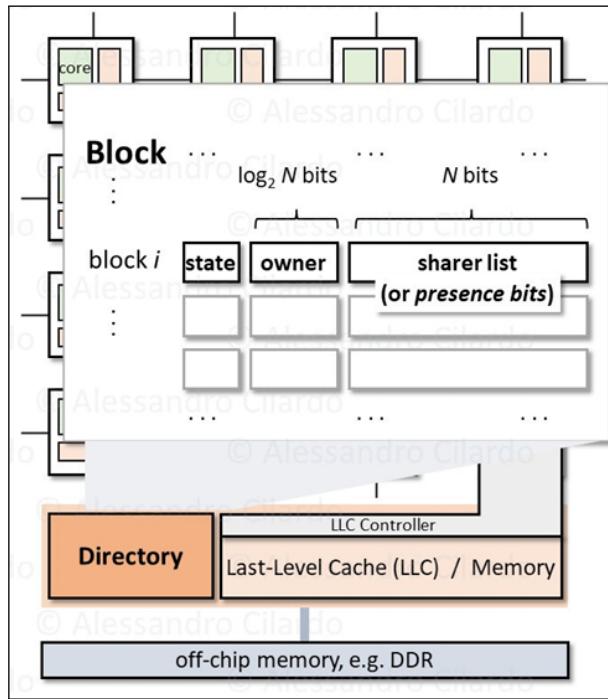
Warning: Il protocollo basato su directory non specifica i dettagli implementativi della rete di interconnessione. Non deve necessariamente essere un bus condiviso, e richieste e risposte possono fluire in diversi canali.

Per ogni blocco possiamo definire dei ruoli per i nodi:

- **Home:** Nodo la cui memoria principale contiene la linea;
- **Dirty:** Nodo che in cache una copia del blocco ma sporca;
- **Owner:** Nodo che al momento mantiene la copia più recente del blocco e deve rispondere alle richieste per quel blocco;
- **Exclusive:** Unico nodo che ha il blocco, sia pulito che sporco;
- **Local:** Nodo che effettua una richiesta per il blocco.

Assumiamo come ipotesi che l'interconnessione garantisca l'ordinamento point2point, ovvero che se un controller A manda due messaggi al controller B, allora i messaggi arrivano a B nello stesso ordine in cui sono partiti da A. Sotto quest'ipotesi la directory agisce la *punto di serializzazione*, ovvero di ordinamento delle richieste, ma, a differenza del protocollo basato su snooping, sono necessari

messaggi di ack dagli sharers in quanto non c'è cognizione di un ordinamento totale o di serializzazione globale. Ipotizziamo inoltre LLC monolitici con un singolo *Directory Controller*.



Assumiamo che il sistema consti di N nodi (cores). Il registro contenuto in Directory tiene traccia di quale cache ospita quale blocco e in che stato, e mantiene lo stato ogni blocco. Per ogni blocco, usa un vettore di N bit denominato **presence bits**, insieme ai bit per indicare lo stato. Le informazioni conservate nella directory includono lo stato **stabile** di coerenza (I,O,M,...), l'identità dell'owner su $\log_2 N$ bits. Le informazioni della directory sono utilizzate e aggiornate in base al particolare protocollo di coerenza.

3.2.8 Protocollo Directory MSI

Quando un core ha bisogno di una linea, la richiede alla Directory. Se lo stato della linea è **I** in Directory (ovvero presente in nessuna cache), allora i dati sono copiati dalla directory al richiedente, e analogamente accade se lo stato è **S**, indipendentemente da quanti sono gli sharer. Se il blocco è **O**, ovvero owned da un altro core e potenzialmente sporco, la Directory richiama il blocco dal proprietario, e ritrasmette i dati al richiedente (si può implementare un meccanismo per trasmettere i dati in parallelo al richiedente e alla Directory), e viene aggiornato lo stato: Il nuovo stato è **S** se arriva una GetS, è **M** se arriva una GetM, e in quest'ultimo caso aggiornare il campo owner con il nuovo proprietario. Prima di completare la transizione a M, la Directory contatta tutte le cache del presence bits individualmente e deve aspettare un ACK da tutte.

Questo protocollo può essere ottimizzato nel seguente modo:

- Dato che non è necessario ordinare le risposte e non è necessario trasferirle in broadcast, queste possono viaggiare su una rete separata che non supporta broadcast e ordinamento;

- Eliminare l'ipotesi di Directory associata ad un singolo LLC: Più Directory indipendenti possono gestire la coerenza di diversi insiemi di blocchi. La Directory per un dato blocco è fissa in un solo posto, ma diversi blocchi possono avere diverse Directories. Ogni blocco ha una *home*, ovvero una directory che ospita i dati e lo stato di directory;
- Riduzione dell'informazione conservata in Directory: Una entry nella sharer list corrisponde ad un gruppo di K caches, e se una o più caches di quel gruppo hanno il blocco potenzialmente è in stato S, allora viene alzato il bit. Questo però è inefficiente in quanto molti dei messaggi di *invalidation*, che comunque devono essere mandati dalle cache, risultano inutili ai fini del mantenimento della bitmap. La soluzione consiste nel tenere traccia degli M blocchi ($M < N$) che hanno 0 o pochi sharer, in modo tale da ridurre i bit a $M \cdot \log_2 N$. Questo potrebbe però richiedere un meccanismo aggiuntivo per gestire il caso, comunque raro, di voler aggiungere un $(M+1)$ -esimo sharer.



Warning: I protocolli di coerenza possono introdurre deadlocks! Infatti la ricezione di un messaggio può scatenare l'invio di un messaggio di tipo *forward*, e questi messaggi possono usare una risorsa comune (linee di rete, buffers o code). Una soluzione è l'uso di *virtual networks*, che separano i messaggi in classi, e vengono utilizzati buffer dedicati per ogni tipologia di messaggio, in modo da evitare cicli nel grafo delle dipendenze a livello di *risorsa di rete*.

3.3 Primitive di sincronizzazione

La sincronizzazione è cruciale nei sistemi a memoria condivisa, ed è intrinsecamente indispensabile il supporto hardware per realizzare delle primitive di sincronizzazione. In particolare è necessaria un'istruzione atomica per leggere e poi scrivere la memoria. È necessaria una nuova classe di istruzioni, ovvero le **atomic RMW** (read-modify-write).

```

1 * ATOMIC EXCHANGE
2 * swap di un valore in memoria con un valore in un registro
3 * se troviamo 0 nel registro il lock preso,
4 * altrimenti continuiamo lo spinning
5
6     EXCH      R5 ,0(R1)
7
8 * ESEMPIO DI SPINNING
9     MOVE      R2 ,#1
10 LOOP    EXCH      R2 ,0(R1)
11     BNE      LOOP
12
13 * TEST AND SET
14 * testa un valore in memoria, e se il valore nel registro
15 * = 0, allora signicia che il lock e preso
16
17     TS       R3 , 0(R1)

```

```

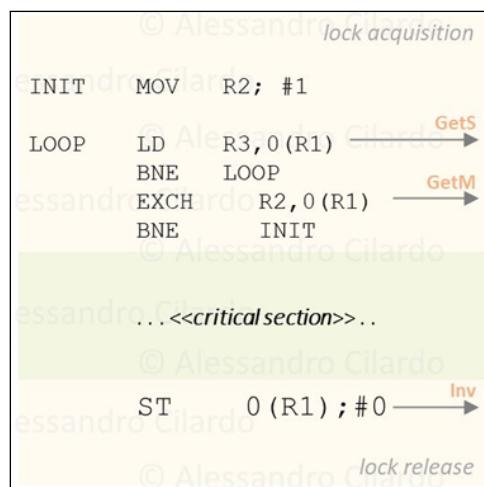
18
19 * FETCH AND INCREMENT
20 * ritorna il valore di una locazione di memoria e
21 * automaticamente lo incrementa
22
23     FI          R8 ,0(R1)

```

Le caches devono necessariamente essere progettate per supportare le istruzioni RMW, ad esempio per supportare un'operazione *Read-Lock*, che recupera un blocco in stato E, invalida gli altri, e impedisce l'accesso al blocco agli altri finché non viene rilasciato, e un'operazione *Write-Unlock*, che fa un update e rilascia la linea.



Info: Quando si effettua lo spin su un lock, tecnicamente si perde tempo in busy wait. Questa cosa può essere evitata sfruttando l'evento di sincronizzazione globale di "sblocco" di una linea, quindi sfruttare un messaggio di gestione della coerenza per la sincronizzazione. Comunque le istruzioni atomiche generano un invalidazione, e di conseguenza un notevole traffico sul bus. Più semplicemente, possiamo iterare in lettura sulla variabile in cache e provare una costosa EXCH solo quando effettivamente il valore cambia.



Fare spin in lettura sulle caches locali riduce il traffico globale, ma sono necessari comunque messaggi di coerenza quando si rilascia e poi si acquisisce il lock. Se k core necessitano del lock, iniziano uno spin lock e ciò comporta che la variabile viene condivisa k volte. Il core che ottiene il lock acquisisce la variabile in stato M, e quando rilascia il lock sono inviati k-1 messaggi di invalidazione. Quindi la variabile viene ricoppiata in memoria, e ricomincia il processo con K-1 cores. In definitiva si hanno $2k + 1$ messaggi di gestione della coerenza per ogni occorrenza di acquisizione/rilascio del lock, e se N cores competono per il lock, allora il numero di messaggi globalmente sarà $2N + 1 + 2(N - 1) + 1 + \dots + 2 + 1 = N^2 + N = O(N^2)$.

Le istruzioni RMW atomiche sono difficile da implementare, perché richiedono due operazioni (di cui una di scrittura) in una sola ininterrompibile operazione. Ciò è difficile da realizzare in quanto la scrittura genera updates invalidanti per gli altri. La soluzione è aggiungere due particolari istruzioni, denominate **Load Linked (LL)** (ritorna il valore

iniziale della variabile lock, è un'operazione di lettura e non richiede traffico globale) e **Store Conditional** (ritorna 1 se ha successo, 0 altrimenti e non genera updates quando non ha successo). Queste istruzioni sono trasparenti in caso di fallimento di SC, e possono essere usate come base per altri meccanismi di sincronizzazione:

```

1 * ATOMIC EXCHANGE
2 LOOP      MOVE    R3 ,R7
3          LL     R5 ,0(R1)
4          SC     0(R1),R3
5          BNE   LOOP
6          MOVE    R7 ,R5
7
8 * FETCH AND INCREMENT
9 LOOP      LL     R5 ,0(R1)
10         ADD   R5 ,R5 ,#1
11         SC     0(R1),R5
12         BNE   LOOP

```

3.4 Cache performance nei sistemi paralleli/multicore

Le performance della cache sono determinate da una combinazione del traffico dovuto ai singoli cache miss e al traffico dovuto alla comunicazione. I cache miss possono essere categorizzati (le 4 c):

- **Conflict Misses**: misses che dipendono dal fatto che la cache non è full associative con LRU replacement;
- **Cold Misses**: Miss in quanto le cache all'avvio del sistema hanno tutti i blocchi invalidi;
- **Capacity Miss**: si verificano quando il *working set* di un programma, ovvero l'insieme dei dati a cui esso accede in un determinato intervallo di tempo, eccede la capacità della cache;
- **Coherence Miss**: sono quei cache miss che non dipendono né dalla capacità della cache né dai conflitti di mappatura, ma dal fatto che in un sistema multiprocessore la coerenza della cache deve essere mantenuta. In pratica, un blocco che era già presente nella cache di un core può essere invalidato perché un altro core ha scritto in quella stessa linea di memoria. Quando poi il primo core torna ad accedere a quel blocco, non lo trova più valido nella sua cache e deve rileggerlo dal livello inferiore (o da un'altra cache): questo genera appunto un coherence miss.

Differenziamo inoltre i True e i False sharing misses: I True sharing misses sorgono dalla comunicazione di dati attraverso il meccanismo di coerenza, e sono dovuti al fatto che uno stesso dato in memoria è condiviso e uno dei thread potrebbe invalidarlo causando un miss; i False sharing sorgono quando un core richiede una word pulita che si trova però in un blocco invalido. In altre parole sono i miss che accadono perché la dimensione del blocco non è una sola word.

