Riassunto di

Architettura dei Calcolatori

Riva 'ZaX' Samuele

28 gennaio 2006

Indice

Capitolo 1

Introduzione

L'architettura di un calcolatore può essere vista da due diversi punti di vista:

Programmatore: modello programmativo, descrizione del linguaggio macchina;

Progettista: modello hardware.

La base di ogni architettura è il paradigma di **Von Neumann** (Fig. 1.1), con la sua variante di Harvard.

1.1 Flusso di Controllo

Lo spazio di indirizzamento in memoria è unico, l'informazione è identificata dal suo indirizzo.
Tramite l'utilizzo di un Program Counter (PC), l'Unità di Controllo (UC) può reperire l'indirizzo della prossima istruzione da eseguire.
L'UC, unica unità *attiva*, decodifica l'istruzione e ne controlla l'esecuzione, simultaneamente calcola l'indirizzo della prossima istruzione da eseguire (PC+1 di solito, a meno che ci siano salti o sottoprogrammi...).
L'esecuzione è **sequenziale** e **seriale**: le istruzioni vengono eseguite in sequenza e una dopo l'altra, senza sovrapposizioni.

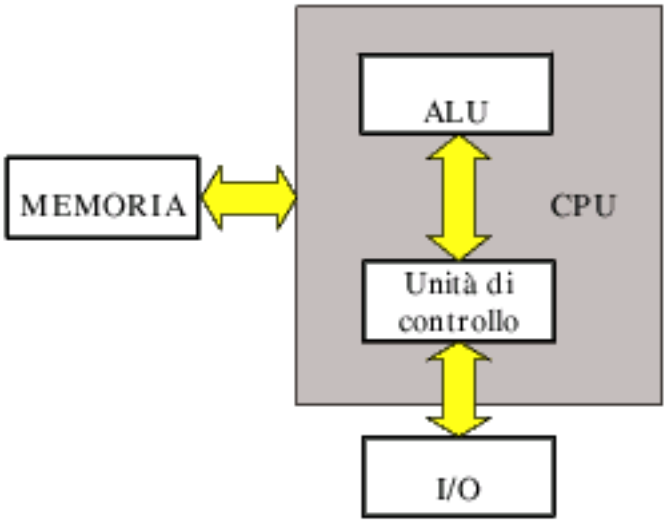


Figura 1.1: `Il paradigma di Von Neumann`

1.2 Prestazioni

Le prestazioni indicano il tempo di risposta/di esecuzione. Minore è il tempo di risposta, migliori sono le prestazioni.

Di fondamentale importanza nello sviluppo di un'architettura è la valutazione delle prestazioni ai diversi livelli (applicazione, Kernel, ...).
Le prestazioni dipendono dal ciclo di clock, dal numero di cicli di clock e altri fattori, come si può vedere dalla Fig. 1.2. I dati di partenza sono il tempo di un ciclo di clock e il numero di istruzioni eseguite.

$$\frac{Instructions}{program} \times \frac{clock\_cycles}{instructions} \times \frac{seconds}{program} = CPUtime$$

Figura 1.2: `Tempo di CPU in un'architettura realistica`

Parametro essenziale molto usato è CPI, una cifra che permette di valutare diversi stili alternativi di implementazione:

$$CPI = \frac{CicliDiClockTotali}{Istruzioni} \tag{1.1}$$

Dove sia cicli di clock che istruzioni si riferiscono al programma di cui esaminare le prestazioni.

I colli di bottiglia che non permettono il miglioramento di performance sono diversi:

- Logic–memory gap: la memoria è più lenta della logica:
- Esecuzione seriale: è necessario attendere la fine di un'istruzione prima di iniziarne una seconda;
- Esecuzione sequenziale: le istruzioni sono eseguite nell'ordine stabilito dal programmatore.

Come migliorare le prestazioni?

- modificare la struttura della memoria: il programmatore deve avere tanta memoria a disposizione, ma la CPU deve poter accedere ad una memoria veloce. → **creo una gerarchia di memorie**;
- modifico il paradigma di esecuzione della sequenza di istruzioni → **parallelismo**.

Questi due punti verranno trattati nei primi capitoli.

1.3 In sintesi...

Macchina di Von Neumann → sequenziale e seriale.
É possibile migliorare le prestazioni in due modi:

1. Gerarchie di memoria
2. Parallelismo

Capitolo 2

Gerarchie di Memorie

2.1 Introduzione

Le applicazioni richiedono memorie sempre più grandi ma anche prestazioni sempre più elevate. La logica però è molto più veloce della memoria, che quindi risulta essere un collo di bottiglia per le performance di un'architettura hardware.
Le memorie veloci esistono ma sono MOLTO più costose, non è possibile sostituire hard disk di 100 giga con cache altrettanto grandi....
Una soluzione che media tra le prestazioni e l'accessibilità a livello di prezzi è quella delle gerarchie di memoria (vedi Fig.2.1), ognuna con la propria tecnologia e i propri meccanismi di accesso. Al livello più elevato le memorie sono molto piccole ma rapidissime.
Durante l'esecuzione inoltre è spesso necessario eseguire dei trasferimenti tra i diversi livelli.

Il principio alla base delle gerarchie di memoria è quello di **località**:

temporale: quando si fa riferimento ad un elemento in memoria, è probabile che si faccia ancora riferimento ad esso entro breve tempo (cicli for, while...);

spaziale: quando si fa riferimento ad un elemento in memoria, è probabile che sia necessario accedere anche ad altri elementi fisicamente vicini (valori in matrici, istruzioni di un processo).

Non sempre questi principi vengono rispettati e a volte diventa necessario modificare l'uso delle gerarchie a seconda dell'uso dell'architettura (video

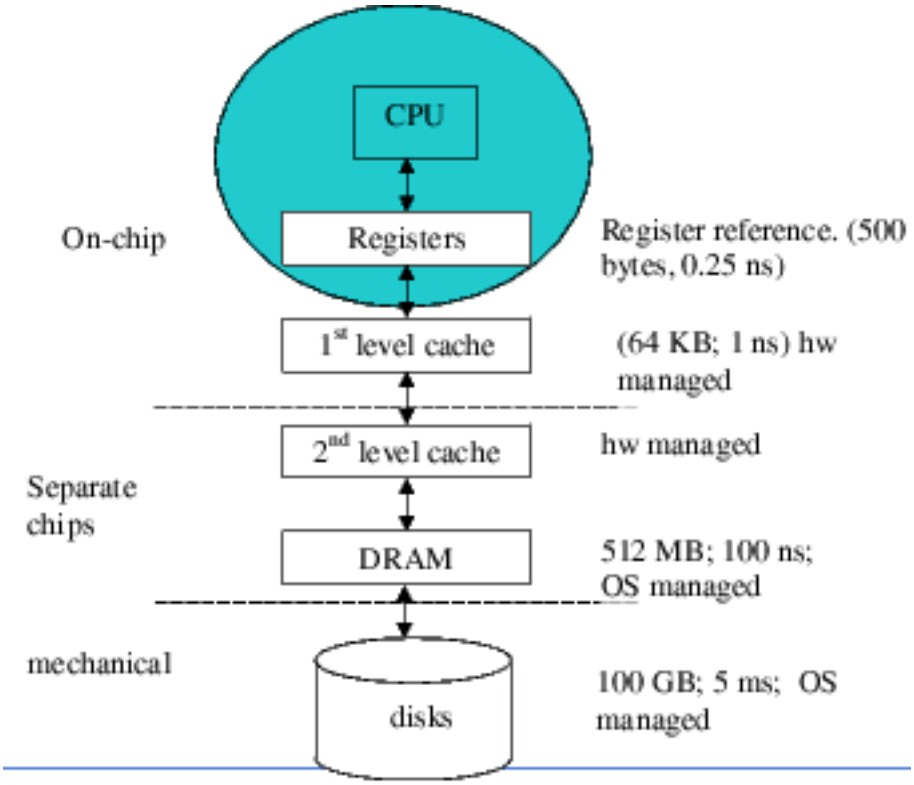


Figura 2.1: `I livelli di memoria in un sistema tipico.`

streaming non ha località temporale per esempio).

La gerarchia più comune è composta di questi elementi:

Registri : fanno intrinsecamente parte della CPU. Sono molto piccoli (64–256 parole), ma estremamente veloci (basta 1 ciclo di clock per accedervi).

Cache di primo livello : di solito è inserita nello stesso chip della CPU, con tecnologia SRAM. Può essere unificata o (ed è meglio) divisa in Instruction–Cache e Data–Cache separate, poichè cache separate possono essere ottimizzate individualmente e con migliori risultati.

Cache di secondo livello : può risiedere su un chip separato dalla CPU, usa tecnologia SRAM. Possono esserci anche altri livelli (tipicamente esiste un terzo livello).

Livello	1	2	3	4
nome	Registri	Cache	RAM	Disk
Dimens. tipica	< 1KB	< 8 MB	< 4 GB	> 30 GB
Tempo di access (ns)	0.25-0.5	0.5-25	150-250	5.000.000
Banda (MB/sec)	20.000-100.000	5.000-10.000	1.000-5.000	20-40
Gestita da	Compiler	HW	OS	OS/ Operator
Memoria retrostante	Cache	RAM	Disk	CD/Tape

Figura 2.2: Caratteristiche delle gerarchie di memoria.

RAM : memoria volatile di grandi dimensioni (fino a 4GB), usa tecnologia (S–)DRAM.

2.2 Cache

La cache è strutturata in blocchi.
Un blocco è la minima quantità di informazione trasferibile tra due livelli di memoria; la sua dimensione è fondamentale per l'efficienza (più i blocchi sono larghi, minori sono i trasferimenti tra livelli, ma l'efficienza potrebbe risentirne e il bus sarebbe più usato).

Gli indirizzi dei byte presenti in cache sono solitamente indirizzi fisici (non serve traduzione rispetto agli indirizzi della RAM), ma non sempre è così. Se sono utilizzati indirizzi virtuali è necessario inserire un meccanismo di traduzione.

2.2.1 Definizioni

Hit : elemento presente in cache;

Miss : elemento NON presente in cache;

Hit rate : frazione degli accessi alla memoria che hanno come risposta un Hit.

Miss rate : frazione degli accessi alla memoria che hanno come risposta un Miss (= 1 – Hit rate).

Hit time : tempo di accesso alla cache in caso di successo.

Miss penalty : tempo di sostituzione di un blocco dal livello inferiore verso la cache.

Miss time : miss penalty + hit time. Tempo necessario per ottenere l'elemento richiesto in caso di miss.

Se aggiungiamo una nuova variabile "cicli di stallo", che indica il numero di cicli in cui la CPU è bloccata in attesa di accesso alla memoria, è possibile calcolare il tempo di esecuzione:

TempoDiEsecuzione = $(CicliCPU + CicliDiStallo) * TempoDiClock$

2.2.2 Problemi essenziali

1. dove situare un blocco (block placement);
2. come trovare un blocco nel livello gerarchico più alto (block identification);
3. quale blocco è necessario sostituire in caso di miss (block replacement);
4. come gestire le scritture, indipendentemente dall'architettura della cache (write policy).

Questi problemi sono risolti in maniera differente a seconda delle diverse tipologie di cache. Ne vediamo ora gli esempi più comuni.

2.2.3 Tipologie

Cache Direct–mapped

Ogni blocco nello spazio degli indirizzi trova il suo corrispondente in un blocco della cache prefissato, più precisamente:

indirizzo in cache = | indirizzo in RAM | $\mathrm{mod}_{\textit{Blocchi nella cache}}$

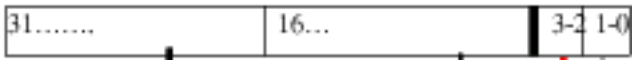


Figura 2.3: Indirizzo usato nelle cache direct–mapped

Nell'esempio della Figura 2.3, vediamo che i primi 2 bit indicano a quale byte della parola ci si riferisce (su 4), i bit 2–3 indicizzano la parola (4 per blocco), i successivi selezionano il blocco (16–3 bit, ho 2^{13} blocchi), gli ultimi 15 indicano l'etichetta (TAG).
Ovviamente il numero di bit delle diverse zone dell'indirizzo può variare a seconda del numero di byte, di parole o di blocchi contenuti nella cache. Tutti i blocchi della RAM che hanno i bit di indirizzamento del blocco uguali (i 13 bit dell'esempio), vengono trasferiti sullo stesso blocco della cache.

Il problema della sostituzione è risolto in maniera semplice ma poco efficiente: non viene tenuto conto della località temporale (un blocco viene sostituito in base ai bit dell'indirizzo senza tenere conto di quanto tempo è stato inserito in cache).
Questo metodo è di facile implementazione, veloce ma non usa tutta la cache a disposizione (in caso di collisioni).

Cache totalmente associativa

É il caso opposto al precedente, poichè non ci sono vincoli sulla sua posizione in cache. Il contenuto del blocco in cache è identificato mediante l'indirizzo completo in memoria (TAG = indirizzo completo).
La ricerca è quindi effettuata mediante confronto parallelo dell'indirizzo ricercato con tutti i TAGS della cache.

In questo caso viene usato ottimamente lo spazio della cache, ma è necessario confrontare i TAGS in parallelo, cosa complicata al crescere delle dimensioni della memoria.

Cache set–associativa a N–vie

Una volta selezionato staticamente il set corretto, il blocco viene inserito in modo dinamico come in una cache totalmente associativa, consentendo un buon sfruttamento dello spazio senza costringere a confrontare i TAG di tutti i blocchi presenti in cache (diventa necessario confrontare solo quelli nel set selezionato).

Aumentando l'associatività di un fattore 2, raddoppia il numero di blocchi in un set e quindi vengono dimezzati i set. L'indice è un bit più corto, mentre il TAG aumenta di un bit (raddoppia perciò il numero di confrontatori da inserire nella cache!).

Problema della sostituzione
Nelle cache diverse da quella direct–mapped si pone il problema di quale blocco sostituire in caso di cache miss. Le soluzioni standard sono 3:

- random: scelta casuale;
- LRU: viene sostituito il blocco che non viene utilizzato da maggior tempo (esiste un contatore che viene decrementato ad ogni accesso alla cache. Il blocco del set richiesto che ha il valore minore viene sostituito);
- FIFO: si approssima la soluzione LRU, eliminando il blocco presente da più tempo in cache e non quello meno usato.

Problema della scrittura
Le scritture sono molto meno frequenti delle letture, e hanno anche prestazioni meno buone (è necessario verificare il preciso byte in cui si vuole scrive controllando il TAG e la parola corretta).
Per ottenere una buona velocità e un'adeguata consistenza dei dati (informazione sulla cache deve coincidere con quella in RAM), si usano due strategie:

- Write–through: l'informazione viene scritta simultaneamente in cache e in RAM. Consistenza ok, velocità male, perchè mi adeguo al tempo di accesso alla RAM.
  Per migliorare le prestazioni di questa tecnica spesso viene usato un write–buffer, un buffer a livello cache in cui sono inserite le diverse modifiche. La RAM leggerà e aggiornerà le modifiche quando presenti nel buffer, mentre la CPU prosegue il proprio lavoro.
  Nascono problemi se il buffer viene riempito (bisogna attendere che la RAM lo svuoti) e in caso di read miss (potrei leggere dati non ancora aggiornati dalla RAM!).

- Write–back: i dati sono scritti nella cache, mentre la RAM viene aggiornata solo al momento della sostituzione del blocco modificato (aggiungo un *dirty bit* che indica se il blocco è stato modificato o meno).
  Ottima velocità (spesso si scrive più volte nello stesso blocco, ma con la velocità della cache!), consistenza raggiunta tramite algoritmi sofisticati.

Le scritture possono portare a write–miss (tento di scrivere su un indirizzo non presente in cache), a cui si può rispondere con 2 algoritmi:

1. Write allocate: carico il blocco in cache ed eseguo la scrittura. Metodo usato dalle cache write–back, si ottengono vantaggi solo se avvengono scritture successive sullo stesso blocco.

2. No write allocate: il blocco viene modificato direttamente in RAM. Metodo usato dalle cache write–through.

2.2.4 Migliorare le prestazioni

Un cache miss può avvenire per 3 motivi:

Compulsory : sono i miss alla prima referenza, al primo accesso un blocco non può certo essere in cache.

Capacity : miss dovuti alla sostituzione dei blocchi dalla cache, diminuiscono al crescere delle dimensioni della memoria.

Conflict : miss da collisione, possibili solo con cache direct–mapped e set–associative, causati dalla condivisione di uno stesso set (o di un blocco se direct–mapped) di più blocchi presenti in RAM.

Un fenomeno che crea perdite di prestazioni è il *cache trashing*, il ripetuto trasferimento dalla e alla cache degli stessi blocchi.
Una possibile risposta a questo problema è il *cache locking*, tramite cui è possibile per il programmatore "bloccare" dei blocchi nella cache, impedendone la sostituzione.

Per ridurre la miss penalty (tempo di trasferimento di un blocco verso la cache) si usa una **cache multilivello**, una di primo livello piccola ma veloce e una di secondo livello più grande per "catturare" molti degli accessi che andrebbero verso la RAM.
Ci sono diverse politiche di organizzazione delle cache multilivello, Multilevel inclusion consiste nel mantenere nella cache L2 tutti i dati presenti in L1 (consistenza I/O e cache), al contrario Multilevel exclusion evita che i blocchi in L1 siano presenti in L2, per ottimizzare lo spazio.

Esistono diversi algoritmi per ridurre al massimo la miss penalty:

- Critical word first: in caso di read miss viene immediatamente caricata la parola mancante, e subito eseguita. In seguito viene caricato il resto del blocco.

- Early restart: le parole del blocco vengono caricate in ordine, la parola mancante viene inviata alla CPU appena è disponibile.

- problemi con write buffer: in caso di read miss dovrei attendere che il buffer sia vuoto, diminuendo le prestazioni; meglio controllare dal buffer gli eventuali conflitti e poi servire il read miss con massima priorità.

- problemi con write back: si serve prima la read miss, salvando temporaneamente il blocco dirty nel write buffer, solo in seguito il blocco è copiato nella memoria. Se nel WB esisteva già lo stesso blocco modificato, eseguo un merge delle due modifiche.

- victim cache: si crea una piccola cache associativa contenente gli ultimi dati sostituiti dalla cache (potrebbero ancora essere utili...).

Una buona soluzione per ridurre la miss rate è aumentare la dimensione dei blocchi (32, 64 bytes) e usare un'associatività elevata (8–vie). I vantaggi sono molti ma esistono anche dei "contro" nell'uso di queste due tecnologie, rispettivamente l'aumento dei conflict miss e la diminuzione della frequenza di clock.
Altri metodi sono l'ottimizzazione da compilatore (organizzazione del codice per migliorare la località temporale e spaziale e la Way–Prediction (nuovi bit cercano di predire il blocco in un set a cui si accederà successivamente).

Un approccio molto usato è il **parallelismo**, che consiste nel sovrapporre l'esecuzione del programma all'attività nella gerarchia di memorie. Una tecnica è il prefetching HW di un blocco prima che venga richiesto dalla CPU. Questo blocco (di solito il successivo di quello effettivamente richiesto), viene posto nell'instruction stream buffer, da cui verrà preso in caso di richiesta dalla CPU.

2.3 Supporto HW alla memoria virtuale

La memoria virtuale permette di separare i concetti di **spazio di indirizzamento** e **dimensione effettiva della memoria fisica**.
Lo spazio di indirizzamento è il numero di parole indirizzabili (dipende dal numero di bit dell'indirizzo), la dimensione della memoria fisica è il numero di byte che la costituiscono.
Un programma in esecuzione usa indirizzi virtuali (non indica la precisa posizione in RAM!), quindi sono indirizzi rilocabili. Tramite elementi HW e SW esistono meccanismi che traducono questi indirizzi virtuali in indirizzi fisici, in modo trasparente sia al programmatore che al compilatore.

2.3.1 Paginazione e MMU

Meccanismi di gestione della memoria virtuale rilocano il programma in diversi blocchi di dimensione fissa detti *pagine*, di solito di dimensioni tra i 512byte e i 64Kbyte. Questo metodo riduce la frammentazione della memoria (uso adeguatamente gli spazi anche non contigui).
Esistono così uno spazio di indirizzamente virtuale e uno fisico, associate tramite una *tabella delle pagine* relativa al processo.
La paginazione consente di rilevare gli accessi a zone di memoria non appartenenti allo spazio di indirizzamento virutale del processo (NPV non esistente nella tabella), e permette anche (con l'aggiunta di bit di protezione) di definire i diritti di accesso alle pagine (Read, Write, eXecute).

Memory Management Unit

La traduzione da pagina virtuale a pagina fisica è compiuta dalla **MMU** (Memory Management Unit), un componente che contiene una memoria associativa molto veloce in cui sono inserite le tabelle delle pagine dei vari processi (vedi Fig. 2.4), che sono identificati dal valore ID. In realtà la

PID	NPV	NPF

Figura 2.4: `Memoria della MMU.`

memoria della MMU contiene solo una parte delle tabelle delle pagine, cioè la righe relative alle pagine maggiormente utilizzate.

Per il calcolo dell'indirizzo fisico la MMU prima verifica se la pagina richiesta è presente in memoria centrale (segnalato dal bit di validità), in caso contrario viene generato un page fault e la pagina viene caricata da

disco (vedi Fig. 2.5). Se la pagina è presente in memoria la MMU legge

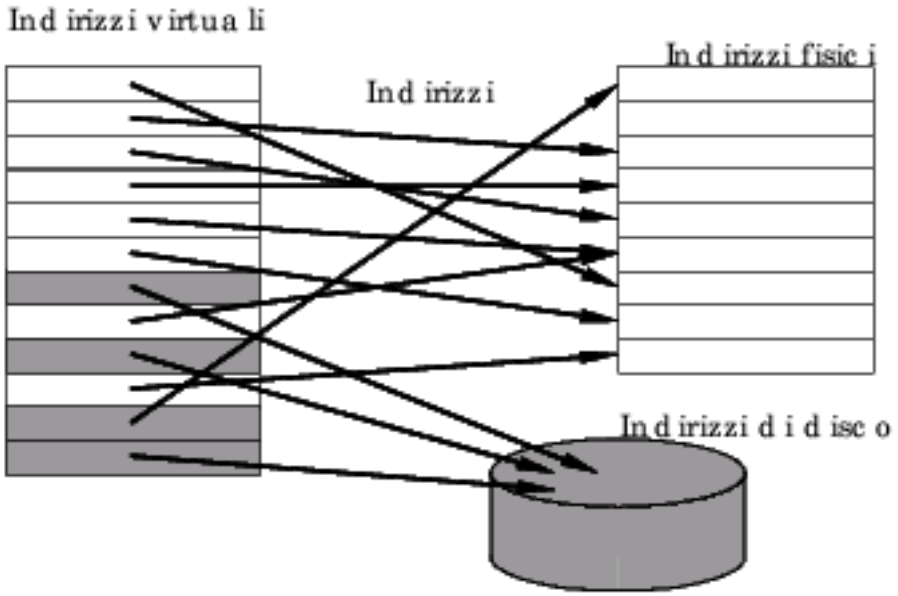


Figura 2.5: `Associazione indirizzi virtuali-fisici`

il relativo indirizzo della pagina fisica e lo compone con l'offset (che è lo stesso tra i due spazi di indirizzamento).

Per aumentare le prestazioni esiste una speciale **cache TLB** (translation–lookaside buffer) di 32–4096 elementi, che registra le traduzioni virtuale–fisico effettuate di recente.
Come per le cache è necessario gestire le miss (se manca la pagina in TLB non significa che c'è page fault!), e la consistenza (alla sostituzione di un elemento modificato devo propagare le modifiche verso la tabella delle pagine).
In caso di miss nel TLB e anche nella tabella delle pagine, significa che l'elemento non è in memoria e bisogna caricarlo da disco (milioni di cicli di clock!!!), si attiva un meccanismo di gestione degli errori che vedremo in seguito.

2.4 In sintesi...

Principio base: **località** temporale e spaziale.
Gerarchia di: registri, cache (I-cache, D-cache), RAM e disco fisso.

Problemi:

block placement e block identification dipende da quale tipologia di cache viene usata → direct mapped (non uso località temporale, associo le posizioni in cache staticamente), totalmente associativa (sfrutto tutto lo spazio, allocandolo dinamicamente), n–associativa (soluzione di compromesso, creo set di N blocchi su cui applico allocazione dinamica).

block replacement sostituzione casuale, LRU, FIFO.

write policy write–through (scrivo direttamente in RAM le modifiche) oppure write–back (modifico i blocchi in cache, solo in seguito li copio in RAM).

Per ridurre miss penalty uso cache multilivello e un algoritmo per velocizzare la lettura della word richiesta: critical word first (carico subito la parola, poi il resto del blocco), early restart (carico le parole in ordine, appena arriva quella che mi serve la uso immediatamente).

Tramite **memoria virtuale** separo i concetti di spazio di indirizzamento e dimensione effettiva della memoria fisica. Necessita di meccanismi che traducono indirizzi virtuali in indirizzi fisici → **MMU**.
La MMU contiene le tabelle delle pagine dei vari processi con le relazioni indirizzo virtuale–fisico-

Capitolo 3

Architettura della CPU

3.1 ISA

La parte di un'architettura che è visibile al programmatore e al compilatore è detta ISA (**Instruction Set Architecture**).
Ogni ISA può essere specializzata per il tipo di applicazioni che si vogliono far girare sul sistema (PC usano operazioni in virgola mobile, Server gestiscono basi di dati, Sistemi embedded devono costare e consumare poco...).
Questa specializzazione dipende da due fattori predominanti:

- Operazioni macchina
- Modalità di indirizzamento

3.1.1 Operazioni macchina

Sono un punto critico per le prestazioni del calcolatore, poiché l'insieme delle istruzioni determina il programma oggetto che viene creato a partire dal codice sorgente.
Esistono due tendenze opposte in questo campo:

CISC Complex Instruction Set Computer: traduce in singole istruzioni macchina delle operazioni anche molto complesse.
L'unità hardware dedicata esegue una singola operazione complessa in tempo molto minore rispetto all'esecuzione di una serie di istruzioni semplici che portano allo stesso risultato. Il ciclo di clock però deve

essere allungato e questo rallenta la CPU nell'esecuzione delle diverse operazioni più semplici (non conviene CISC a meno di un'alta percentuale di operazioni complesse).

RISC Reduced Instruction Set Computer: rende più veloci le istruzioni semplici usate più spesso, semplificando l'Unità di Controllo e rendendo più veloce il ciclo di clock.

In breve: CISC velocizza molto alcune operazioni complesse, RISC velocizza un po' le operazioni più semplici e usate.

3.1.2 Modalità di indirizzamento

Esistono diverse soluzioni che gestiscono diversamente l'accesso agli operandi e la gestione dei registri della CPU:

a 3 indirizzi : l'istruzione contiene sia il tipo di operazione che gli indirizzi dei due operandi e l'indirizzo destinazione in cui salvare il risultato $\rightarrow add\ Ris,\ op1,\ op2$.
Problema: il numero di bit per indicare gli indirizzi è limitato.

a 2 indirizzi : il primo indirizzo contiene un operando e alla fine dell'operazione conterrà il risultato.
Miglior sfruttamento dello spazio, ma si perde riferimento al primo operando.

a 1 indirizzo : si fornisce esplicitamente l'indirizzo di un solo operando, mentre l'altro è contenuto in un registro privilegiato detto accumulatore, che al termine conterrà anche il risultato.
Problema: bisogna predisporre l'accumulatore... veniva usato su CPU con parole di pochi bit.

a 0 indirizzi : fanno riferimento allo stack, da cui prendono gli operandi e su cui inseriscono il risultato.
Problema: le prestazioni crollano a causa del sovraccarico dello stack.

Ogni architettura inoltre supporta determinati tipi di indirizzamento:

- modalità diretta: l'indirizzo di memoria *x* viene specificato $\rightarrow addM[x]$.

- modalità indiretta: si indica una parola in memoria, contenente l'indirizzo richiesto $\rightarrow add\ \ M(M[xx])$.

- modalità indiretta da registro: viene indicato un registro in cui è contenuto l'indirizzo $\rightarrow add\ M[Ri]$.

- modalità relativa al registro: si costruisce l'indirizzo come uno spiazzamento rispetto al valore di un determinato registro $\rightarrow add100,\ (Ri)$.

- modalità relativa al PC: si costruisce l'indirizzo come uno spiazzamento rispetto al valore del Program Counter.

Alcune architetture supportano tutti i modi di indirizzamento (quelle del passato), altre più recenti (8086) usano indirizzamento diretto o indiretto e istruzioni di lunghezza variabile a seconda del tipo di istruzione.

3.1.3 Accesso alla memoria

Certe architetture permettono ad ogni istruzione l'accesso alla memoria, ma questo causa rallentamenti e rende problematico l'uso del pipelining. Per questo motivo sono nate le macchine "registro–registro" (dette anche Load–Store), che permettono l'accesso alla memoria solo da parte delle istruzioni Load e Store[1].
Tramite queste macchina è possibile velocizzare il ciclo di clock ma il programma oggetto è più lungo (vedi Fig. 3.1).

Il maggior beneficio di quest'architettura deriva dalla regolarità delle istruzioni, che rende semplice la creazione di una struttura con pipeline efficiente.

3.2 Parallelismo

La concorrenza consente di raggiungere migliori prestazioni grazie all'esecuzione simultanea di più istruzioni.

[1] Le istruzioni operativi avranno operandi solo nei registri interni

C := X + Y

```
Macchina con riferimento a memoria:

Load  [M(X)]   carico X
Add   [M(Y)]   carico Y e sommo ad X
Store [M(C)]   salvo in C
```

```
Macchina load-store:

Load  R1,  100(R5)
Load  R2,  104(R5)
Add   R3,  R1,  R2
Store R3,  200(R5)
```

Figura 3.1: `Esempio di codice oggetto creato con una macchina load--store`

Per quanto ci riguarda studieremo il parellelismo funzionale nella sua soluzione di più basso livello, a livello istruzione (ILP). ILP consiste nell'eseguire più istruzioni di uno stesso programma simultaneamente.
Per consentire questo funzionamento parallelo si usano due tecniche ortogonali (possono coesistere), la replica di unità funzionali (diverse unità eseguono le stesse operazioni in parallelo su diversi dati) e il pipelining (più unità sono usate in sequenza per compiere una singola computazione).

3.2.1 Legge di Amdhal

Per definire quale tecnica usare e in che modo si studiano le prestazioni, riferendosi allo speed–up totale.
Lo speed–up è il rapporto tra tempo di esecuzione e tempo di esecuzione DOPO l'inserimento delle migliorie:

$$Speedup_{tot} = \frac{1}{(1-frazione)+\frac{frazione}{speedup}}$$

Dove $frazione$ è la parte di codice che viene velocizzata di un valore pari a $speedup$[2].

[2]Per alcuni utili esempi vedi da pagina 35 delle dispense CPU1

3.3 Architettura MIPS

Per lo studio del parallelismo usiamo la macchina MIPS, un'architettura a 32bit del tipo registro–registro, con istruzioni di 4 bytes. Le istruzioni jump e branch indicano la destinazione usando indirizzamento relativo (spiazzamento rispetto al PC).
Le istruzioni di tipo I *load* e *store* operano su immediati (valore diretto dell'operando), le istruzioni di tipo R *ALU* invece operano registro–registro, mentre le istruzioni J, *salti condizionati* o *chiamate a subroutine*, usano uno spiazzamento relativo a PC di 25bit.

3.3.1 Pipelining

Si possono trovare 5 fasi nella vita di un'istruzione:

1) Fetch : l'istruzione viene letta dalla memoria $\$Reg \leftarrow M[PC]$ e il PC incrementato di 4.

2) Instruction decodify : l'istruzione viene decodificata. Sono letti i registri e il loro contenuto viene inserito in due latch A e B. I bit 0–15 dell'istruzione sono interpretati come se fossero un valore immediato.
Queste ipotesi (lettura di registri e bit "immaginati" come un immediato), possono rivelarsi inutili, ma non causano perdita di tempo in caso di errore, mentre provocano uno speed–up se si rivelano operazioni necessarie.

3) Execute : viene richiesta un operazione da parte dell'ALU, oppure avviene accesso alla memoria dati.
Nel primo caso i valori dei registri sono inviati all'ALU, insieme ai comandi di funzione. Nel caso invece di accesso alla memoria (tramite store/load) il valore immediato esteso a 32 bit (lo spiazzamento) viene sommato al registro base nel latch A e il risultato è scritto nel registro indirizzamento/scrittura della D–cache.

4) Memory access : non tutte le istruzioni sono attive. Load/store invia il segnale di read/write alla D–cache. Nelle operazioni di salto con-

dizionato viene controllata la condizione e, in caso positivo, sostituito il valore nel PC.

5) Write back : non tutte le istruzioni sono attive. Load scrive i dati presenti nel registro di lettura nel Register File. Per le istruzioni ALU il risultato è scritto nel registro destinazione.

Nei processori senza pipeline ogni operazione ha una diversa latenza, ad esempio Load è la "più lunga", mentre un salto incondizionato la più breve. In un processore con pipeline invece ogni operazione ha latenza pari al massimo, ma agisce in concorrenza con altre.

3.3.2 DataPath

Nell'architettura pipeline del MIPS ogni stadio corrisponde ad una fase: IF, ID, EXE, MEM, WB (vedi Fig. 3.2). Ogni istruzione attraversa tutti questi stadi, sincronizzati sulla latenza dello stadio più lento.
Il concetto alla base del pipelining: non appena l'istruzione *i* ha lasciato

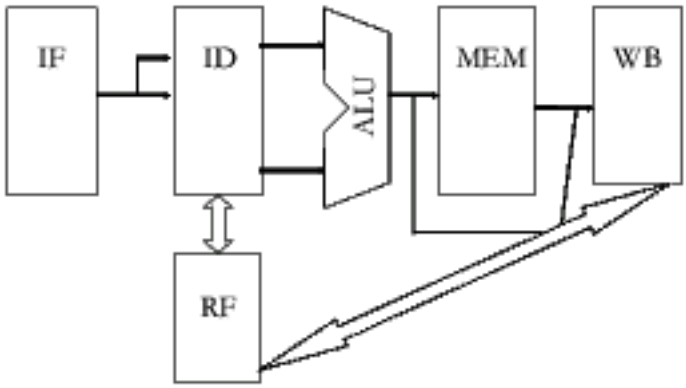


Figura 3.2: `I 5 stadi dell'architettura MIPS. Tra ogni coppia di stadi vengono inseriti dei registri latch che fanno da buffer per stabilizzare l'informazione e gestire le eccezioni.`

uno stadio, l'istruzione *i+1* può occuparlo.
Idealmente ad ogni ciclo di clock si può completare una nuova istruzione (CPI asintoticamente uguale a 1).

Rimane il concetto di sequenzialità della macchina di Von Neumann ma non si ha più esecuzione seriale.

Il parallelismo è detto intrinseco, perchè invisibile al programmatore, mentre dei compilatori ottimizzati possono sfruttarne a pieno i vantaggi. Aumentando il numero di stadi diminuisce il tempo per istruzione TPI, ma fino ad un massimo di circa 20 stadi.

Esempio di funzionamento
Esempio1: load word (lw)

1. Stadio IF: l'istruzione viene letta dalla I–cache e trasferita al buffer interstadio IF/ID, in cui è memorizzato anche il nuovo valore di PC+4.
2. Stadio ID: estensione dello spiazzamento a 32bit. Sono letti i registri e il loro valore viene memorizzato nel buffer ID/EXE, insieme al contenuto del PC.
3. Stadio EXE: si calcola l'effettivo indirizzo di memoria (con l'ALU), il risultato è salvato nel buffer interstadio EXE/MEM.
4. Stadio MEM: i dati sono letti dalla D–cache e trasferiti nel buffer MEM/WB.
5. Stadio WB: i dati salvati nel buffer interstadio sono scritti nel registro destinazione.

Esempio2: store word (sw)
Le fasi 1,2 sono identiche per *tutte* le operazioni.

3. Stadio EXE: calcolo l'indirizzo di memoria allo stesso modo di lw.

4. Stadio MEM: i dati letti dai registri nello stadio 2 sono trasferiti al registro di scrittura della D–cache, a cui viene inviato il comando "write".

5. Stadio WB: non vengono scritti i registri, la store attraversa lo stadio senza attivare alcuna unità per garantire la sincronizzazione.

3.3.3 Conflitti

La presenza di più istruzioni contemporanee può portare a vari tipi di conflitti, che possono trasformarsi in *alee*, cioè in cause di errore.
Ad esempio, se ad una load word segue una store word, nel quarto ciclo (MEM per load, EXE per store), lw carica nel registro di lettura/scrittura la parola richiesta, mentre sw tenta di caricare i dati da scrivere successivamente, causando un conflitto per accesso alla medesima risorsa nello stesso ciclo di clock. La soluzione consiste nell'introdurre dei cicli vuoti (riduco il throughput!) o replicare le risorse tramite due registri diversi per lettura e scrittura.
Altri problemi nascono per dipendenze sui dati o sul controllo.

- Dipendenze sulle risorse: due istruzioni richiedono la stessa risorsa. Soluzioni: via software, compilo il programma in modo che le istruzioni che agiscono sulla stessa risorsa siano ad opportuna distanza tra loro. Via hardware si può invece replicare la risorsa.

- Dipendenze sui dati: due istruzioni hanno in comune un operando in un registro o in memoria in cui almeno in un caso è operando di destinazione. Queste dipendenze sono possibili nei cicli o nel codice lineare.
  Nel codice lineare ci sono tre diversi tipi di dipendenza:

  1. dipendenza RAW (Read after Write): l'istruzione *i+1* chiede come sorgente un operando destinazione dell'istruzione *i* (che non ha ancora finito di modificarlo!).
  2. dipendenza WAR (Write after Read): l'operando destinazione dell'istruzione *i+1* è usato come sorgente di *i*.
  3. dipendenza WAW (Write after Write): due istruzioni consecutive hanno in comune lo stesso operando destinazione.

  WAR e WAW sono dipendenze *false*, dovute al ri–uso di registri o indirizzi di memoria. Questo tipo di dipendenze esistono solo se esistono più pipeline e in ogni caso possono essere risolte mediante *renaming*.

- Dipendenze sul controllo: dovute alle istruzioni che modificano il flusso di controllo (jump, branch). Possono ridurre in modo rilevante le prestazioni delle CPU. Ad esempio per i salti condizionati è necessario valutare la condizione (4° stadio, MEM) per stabilire quale sia la prossima istruzione.

MIPS

→ **Conflitti di dati** Per quanto riguarda l'architettura in esame, gli unici conflitti di dati possibili sono quelli RAW, poichè esiste solo una pipeline. Dividiamo le dipendenze di dati in Define–use e Load–use.

Dipendenze Define–use

sub \$2, \$1, \$3
and \$12, \$2, \$5

La prima istruzione modifica adeguatamente il registro \$2 solo alla fine dei 5 stadi, mentre la seconda istruzione lo richiede come sorgente al suo secondo ciclo (cioè al terzo della sub).
Per risolvere questo problema ci sono due diversi approcci:

1. **a livello software**: si lascia al compilatore il compito di inserire istruzioni non–dipendenti tra loro in modo da non influenzare le prestazioni.
   Se questo non fosse possibile, il compilatore inserirà tanti NOP (istruzioni che non fanno nulla) quanti ne sono necessari per evitare il conflitto.

2. **a livello hardware**: equivalentemente ai NOP, vengono bloccate (*stall*) le istruzioni che seguono la define fino a quando il registro viene correttamente modificato.
   Una soluzione migliore consiste nel modificare il circuito del Data Path, introducendo la tecnica di data forwarding, che permette di identificare le dipendenze non rispetto ai registri ma rispetto ai buffer inter–stadio. Infatti l'operando destinazione è disponibile all'uscita dell'ALU (fine del ciclo 3, EXE), e deve essere disponibile all'inizio del ciclo 3 dell'istruzione seguente (EXE) (vedi Fig. 3.3).

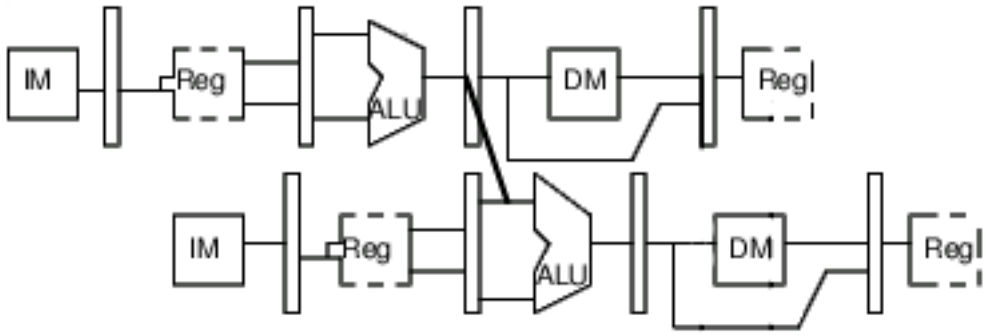


Figura 3.3: `Tecnica di data--forwarding.`

Dipendenze Load–use

lw \$17, 200(\$19)
add \$12, \$17, \$18

Le soluzioni a livello software sono identiche a quelle per i conflitti Define–use, mentre a livello hardware, pur usando la tecnica di data forwardin, resta necessario inserire una *bolla* (un ciclo di clock in stallo), perchè con un'istruzione Load i dati sono disponibili solo al termine dello stadio MEM (stadio 4).

→ **Conflitti di controllo** Quando si incontra un'istruzione di salto condizionato (branch), sono necessari ben 4 cicli per sapere se è necessario eseguire il salto o meno. Nel frattempo, sono state lette ed avviate le prime 3 istruzioni successive al branch, senza sapere se queste saranno mai utilizzate (se avviene il salto, non dovevo eseguire quelle istruzioni!).
Per risolvere il problema dei conflitti di controllo ci sono 3 soluzioni:

- Stallo automatico: la CPU attende il 4° ciclo del branch per sapere qual è la prossima istruzione.
- Branch Hoisting: il compilatore sposta la branch 3 istruzioni PRIMA di quando richiesto, in modo che non occorrano stalli (non sempre ci sono 3 istruzioni prima del branch!).
- Branch Prediction: è la tecnica più usata, vengono automaticamente eseguite le istruzioni che *seguirebbero* il salto in caso di condizione

rispettata, se poi in realtà la strada da prendere non era quella corretta, si esegue un flush della pipeline (viene svuotata) e si prosegue normalmente con le istruzioni successive al salto condizionato.

Esistono altre tecniche di speculazione (di "prediction") migliori, vedremo più avanti.

Per permettere un'adeguata velocità di risposta viene a volte inserito un addizionatore nello stadio di decodifica (il secondo), per calcolare l'indirizzo nel caso di salto incondizionato o di salto effettuato, così da perdere solo un ciclo.

→ **Eccezioni** In caso di un'eccezione, diventa importante comprendere come sia possibile associarla all'istruzione che l'ha sollevata e salvare lo stato della macchina in modo corretto. Infatti con un'architettura a pipeline le istruzioni in esecuzione concorrentemente sono più di una e interrupt, cache miss, ecc... possono essere sollevate in diversi stadi della pipeline. Più nel dettaglio:

- Esecuzione di un'istruzione: viene sollevata nel corso dello stadio EXE; è necessario salvare il PC dell'istruzione successiva, contenuto nel buffer ID/EXE.
- Interrupt esterno: viene associato all'istruzione nella fase di WB, così come per un eventuale guasto hardware. Tutte le istruzioni che precedono quella che ha causato eccezione vengono portate a termine, mentre le istruzioni che seguono sono annullate.
- Cache miss: solo i miss provenienti dalla D–cache sono considerati eccezione (i miss dalla I-cache sono risolti a livello hardware), vengono identificati nello stadio EXE; viene salvato l'indirizzo dell'istruzione stessa.
- Codice operativo inesistente: viene sollevato allo stadio ID, l'indirizzo di rientro è contenuto nel buffer interstadio IF/ID.

In caso di eccezioni sollevate da istruzioni che seguono un salto condizionato (e quindi istruzioni eseguite in modo speculativo, senza cioè la certezza che fossero realmente da eseguire), si rinvia la gestione del problema fino a quando il salto è stato effettuato.

3.4 In sintesi...

Punto critico per le prestazioni del calcolatore è la scelta delle operazioni macchina da supportare (CISC usa istruzioni complesse, RISC semplici) e la modalità di indirizzamento (accesso agli operandi a 1/2/3 indirizzi e accesso alla memoria diretto/indiretto...).

Importante l'uso di **macchine load/store** per velocizzare ciclo di clock (solo Load e Store accedono alla memoria).
Sfrutto parallelismo tramite **pipelining**, usando come esempio una macchina MIPS.
5 fasi: fetch (leggo l'istruzione), instruction decodify (leggo gli operandi dai registri), execute (operazioni effettuate dalla ALU o accesso alla D-cache), memory access (invio il segnale read/write alla D—cache o controllo la condizione dei salti condizionati), write back (scrivo i risultati nei registri).
Concetto base: non appena l'istruzione *i* ha lasciato uno stadio, l'istruzione *i+1* può occuparlo → ad ogni clock si può completare un'istruzione (esecuzione NON seriale).

Presenza contemporanea di più istruzioni può portare a conflitti:

- *Dipendenze sui dati*: diverse istruzioni hanno in comune un operando... ci sono dipendenze RAW (dipendenza vera), WAW, WAR (false, cioè possibili solo in un'esecuzione non sequenziale).
- *Dipendenze di controllo*: un'istruzione modifica il flusso di controllo e si perdono 4 cicli per conoscere quale sia il prossimo valore del PC.

Per risolvere il problema delle dipendenze dati vengono inserite NOP (operazioni nulle) dal compilatore, oppure bolle (cicli di stallo) dall'hardware.
Una soluzione migliore viene dal **data forwarding**, che permette la comunicazione tra buffer inter–stadio.

Per le dipendenze di controllo o si effettua uno stallo automatico, o branch hoisting (sposto il salto 2 operazioni prima di quando richiesto, o branch prediction (si eseguono le operazioni seguenti al salto in caso condizione rispettata).
Molto importante è la gestione delle eccezioni e il relativo salvataggio dello stato della macchina.

Capitolo 4

CPU a elevate prestazioni

Le prestazioni di una macchina con pipeline come l'abbiamo studiata finora, misurate in CPI (cicli di clock per ogni istruzione), non supereranno mai il limite *ideale* di 1 ciclo di clock per ogni istruzione (**CPI = 1**). Inoltre, pur migliorando decisamente il CPI, molte tecniche usate non riescono ad essere efficienti fino in fondo, a causa delle alee e delle molte dipendenze che possono sorgere tra istruzioni sucessive (CPI $>>$ 1).
Per migliorare le prestazioni diventa necessario sfruttare ancora più intensamente il concetto di parallelismo, sia usando **più unità di esecuzione**, sia **evitando al massimo le dipendenze**, tramite riordino delle istruzioni del programma oggetto.

4.1 Rilevamento e risoluzione delle dipendenze

4.1.1 Scheduling statico e dinamico

Per rilevare il codice che può causare delle alee viene usata una tecnica di **scheduling statico**, eseguita dal compilatore, che consiste nella riorganizzazione del codice sfruttando la possibilità di avere un'ampia finestra di visualizzazione sul programma oggetto (il compilatore lo può vedere nella sua interezza).

Per individuare le istruzioni che causano dipendenze sui dati viene usato lo **scheduling dinamico**, che riordina le istruzioni a run time, cercando di ridurre al minimo gli stalli.
Funzionamento: le istruzioni vengono lette e decodificate (primi 2 sta-

di) e inserite in una *finestra di issue* (un registro–buffer di massimo 100 istruzioni), in cui vengono controllate le dipendenze. A questo punto una o più istruzioni vengono lanciate (issued) e inserite in una *finestra di esecuzione*.
Usando questa tecnica le istruzioni possono essere completate anche fuori ordine → **niente sequenzialità**. La mancanza di sequenzialità introduce la possibilità di alee di dati WAR e WAW (spiegate a pag. 24).

4.1.2 Aumentare il parallelismo

Parlando di parallelismo a livello–istruzione (ILP), la ricerca della concorrenza consiste nell'identificazione delle istruzioni prive di dipendenze all'interno di un blocco basico.
Un **blocco basico** è una sequenza di istruzioni con un solo entry point e un punto finale di diramazione (è un blocco di istruzioni, da quella iniziale fino al primo jump/branch). Minore è la lunghezza di questo blocco, minori saranno le prestazioni (lo scheduling avviene all'interno di questi blocchi!!) → è necessario realizzare il parallelismo attraversando più blocchi basici.

3 soluzioni:

1. parallelismo a livello di ciclo ("srotolamento dei cicli", per evitare continui salti);
2. uso di istruzioni vettoriali che operino su sequenze di dati;
3. uso di speculazione per superare le barriere dei blocchi basici.

Un modo diverso per avere maggiore parallelismo consiste nell'aumentare il numero degli stadi della pipeline (fino a 20–25!) → **superpipelining**.
Aumentare il numero degli stadi significa ridurre notevolmente il ciclo di clock (la frequenza a cui opera il processore), ma questo incremento viene limitato da problemi di progetto elettronico (clock skew) e da aumento dei ritardi causati da alee di controllo (in caso di speculazione non corretta devo bloccare più istruzioni erroneamente avviate).

4.1.3 Pipeline multiple

Una ulteriore possibilità per aumentare le prestazioni è l'introduzione di un certo numero di pipeline specializzate, oppure usare più pipeline software copie della pipeline fisica presente sulla CPU.
L'uso di più pipeline in parallelo su un singolo processore è impossibile, a meno di mantenere in comune le operazioni di fetch e decode (vedi Fig. 4.1).
Gli stadi successivi ai primi due saranno eseguiti in parallelo e agiranno

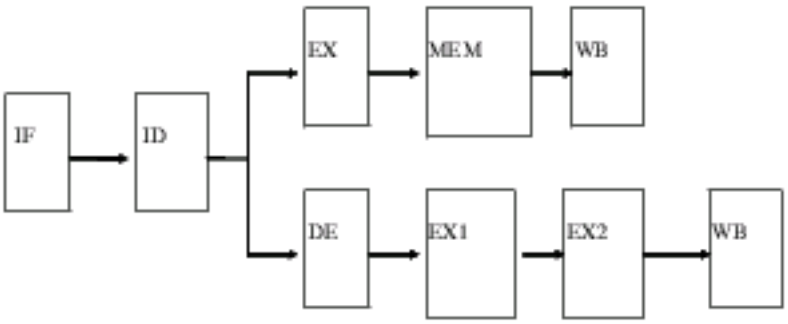


Figura 4.1: `Esempio di una pipeline dedicata ai floating point.`

su due o più register file (RF) differenti.

La presenza di più pipeline con latenza diversa rende possibile l'esecuzione fuori ordine e occorrono quindi tecniche per conservare la consistenza sequenziale... vediamo come:

- permetto il write back solo sotto controllo software;
- sincronizzo forzatamente le pipeline (inserendo bolle nella pipeline più "lenta");
- **riordinamento**: meccanismo tramite cui è possibile scrivere in ordine sequenziale i risultati delle istruzioni eseguite fuori ordine.

4.2 Architetture superscalari

Per eseguire più istruzioni in contemporanea è possibile usare un processore con più unità di esecuzione, con la capacità di leggere, lanciare e decodificare più istruzioni per volta. Quest'architettura è detta *superscalare*, vedi Fig. 4.2.

Le prestazioni di queste architetture aumentano fino a portare CPI < 1,

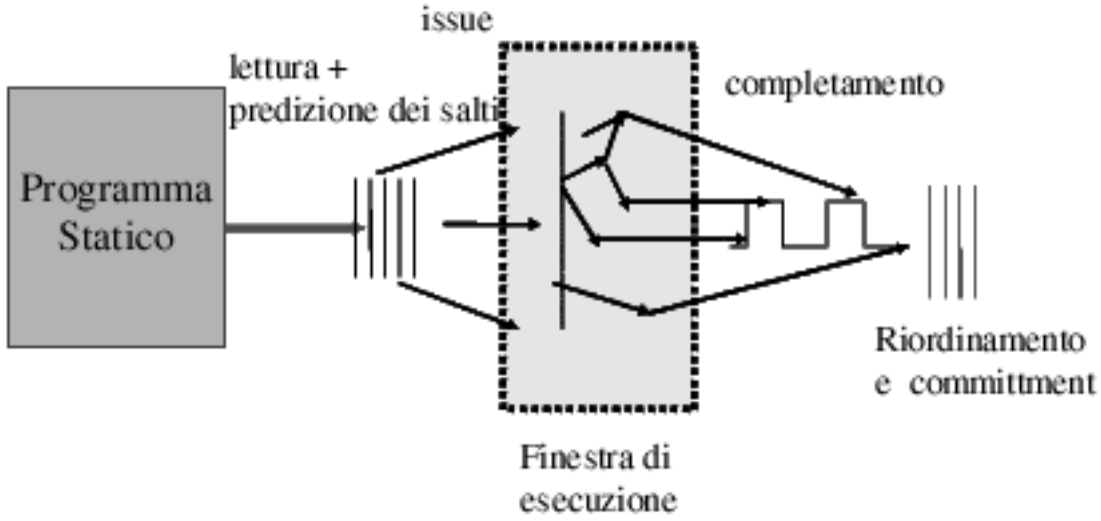


Figura 4.2: `Schema di funzionamento di un'architettura superscalare.`

idealmente CPI = $\frac{1}{n}$ (n = numero di istruzioni che possono essere lette simultaneamente).
Per sfruttare al meglio quest'architettura la banda della cache deve essere superiore a una parola (o diventa un collo di bottiglia).

Entrando nel dettaglio, dopo le fetch parallele delle diverse istruzioni, ci sono diversi stadi da esaminare:

- **Decodifica parallela** – sempre più complessa all'aumentare del grado di parallelismo, a causa delle dipendenze da analizzare.
- **Instruction issue** – è l'attività più critica, poichè vengono analizzate le dipendenze di dati e di controllo. É necessario introdurre politiche di issue per aumentare la frequenza di lancio delle istruzioni.

- **Esecuzione parallela** – l'esecuzione ha inizio quando sono disponibili i dati, senza tenere conto dell'ordine sequenziale del programma.

- **Commit in ordine** – lo stato della macchina viene aggiornato seguendo l'ordine sequenziale.

Lo speedup di un'architettura superscalare rispetto ad un processore scalare è il rapporto tra i tempi di esecuzione:
$Sp = \frac{Tsc}{TsuperSc}$

Data l'intrinseca complessità della decodifica superscalare, il tempo di ciclo deve aumentare spesso in maniera molto evidente. Una soluzione che limita il problema è chiamata **predecodifica**, che consiste nello spostare parte della decodifica dal secondo stadio della pipeline alla fase di caricamento dalla memoria alla I–cache. Precisamente viene aggiunta un'unità di predecodifica che aggiunge 4–7 bit ad ogni istruzione, per distinguere le istruzioni in classi e segnalare informazioni utili successivamente.

4.2.1 Politiche di issue

Le politiche di issue includono alcuni aspetti fondamentali:

Gestione delle false dipendenze: è possibile eliminare le dipendenze dovute ai riferimenti ai registri mediante ridenominazione dei registri, scrivendo il risultato in uno "spazio extra" allocato dinamicamente;

Gestione delle dipendenze di controllo: si può attendere la disponibilità della condizione di controllo, oppure adottare esecuzione speculativa;

Gestione del blocco dell'issue: alcune istruzioni dipendenti tra loro possono bloccare istruzioni successive, per evitarlo uso tecniche di **shelving**.

Shelving Lo shelving disaccoppia issue e controllo delle dipendenze, usando dei buffer speciali contenenti 2–4 istruzioni (reservation station). I controlli sulle dipendenze vengono eseguiti ai buffer, solo in seguito sono inoltrate alle unità funzionali, senza dipendenze (fase di dispatch).
Le istruzioni decodificate vengono inviate agli shelving buffer senza controlli, zona in cui restano fino alla risoluzione delle dipendenze. Quando si libera un'unità funzionale i vari buffer sono analizzati e una istruzione "eleggibile" (operandi disponibili) viene inviata in esecuzione.

Esistono però altre tipologie di buffer oltre alle reservation stations, sia basate su stazioni di gruppo (diversi buffer sono raggruppati a servizio di certe unità funzionali → flessibilità), sia basate su una reservation station centrale che può accettare/fornire più istruzioni per ciclo, sia basate su un buffer combinato detto ROB (ReOrder Buffer).

Esistono 2 politiche di lettura degli operandi:

- **issue bound** – gli operandi sono letti durante l'issue delle istruzioni. I buffer contengono gli operandi sorgenti, letti dal register file.
- **dispatch bound** – gli operandi sono letti durante la fase di dispatch. Gli shelving buffer contengono solo gli identificatori dei registri.

Entrambe le politiche si possono eseguire con o senza ridenominazione, nel primo caso sarà necessario allocare ulteriore spazio per memorizzare i valori ridenominati.

Per quanto riguarda la politica di dispatch, si devono specificare diverse regole:

- Regola di selezione: mando in esecuzione le istruzioni i cui operandi sono disponibili, controllando che non ci siano dipendenze.
- Regola di arbitraggio: tra le diverse istruzioni eleggibili che posso eseguire, ne scelgo alcune (le più vecchie?) secondo un algoritmo prestabilito.
- Ordine di dispatch: determina se un'istruzione non eseguibile blocca anche le successive.

In genere il dispatch avviene fuori ordine, lo schema più avanzato ed efficiente.

4.3 Scoreboard

Nella verifica degli operandi sorgono 2 problemi: la verifica della disponibilità nel register file prima e la verifica della disponibilità nelle reservation stations (RS) in seguito.
Una soluzione a questi problemi viene da un particolare registro di stato costituito da elementi di un bit, settati a 1 in caso di dato disponibile: lo **scoreboard**.
É possibile verificare un operando in modo diretto (vengono controllati i bit dello scoreboard direttamente; tecnica usata con la politica dispatch bound), o controllando i bit di stato delle RS (esiste un bit nei buffer che indica la presenza degli operandi; tecnica usata in caso di issue bound). Lo stato delle RS viene aggiornato ad ogni ciclo.

4.3.1 Ridenominazione dei registri

É la tecnica standard per rimuovere false dipendenze (WAR, WAW), può essere statica (durante la compilazione) o dinamica (durante l'esecuzione). Viene utilizzato un buffer di ridenominazione, contenente la posizione in cui sono salvati i risultati intermedi.

Esistono 3 soluzioni:

Fusione di RF architetturale e RF ridenominazione: i due register file sono allocati dinamicamente nello stesso Register File fisico, di grandi dimensioni; sono gestiti tramite una mapping table.

Separazione del RF architetturale e di ridenominazione: se un'istruzione fa riferimento ad un registro con dipendenze, questo registro viene riallocato nel RF di ridenominazione, su cui sarà possibile agire senza provocare alee.

Uso del ROB: per ogni istruzione viene allocato un diverso elemento del ROB, in cui saranno salvati i risultati intermedi.

4.4 in sintesi...

Per aumentare ulteriormente le prestazioni si usano tecniche per rilevare ed evitare le dipendenze incrementando il parallelismo eseguendo codice in modo non sequenziale:
Scheduling statico: il compilatore riorganizza il codice "osservandolo" nella sua interezza.
Scheduling dinamico: riordinamento delle istruzioni a run–time, usando una finestra di issue e una di esecuzione, da cui vengono lanciate le istruzioni anche fuori ordine (niente sequenzialità, che causa possibili dipendenze WAR e WAW).
Per ottenere maggiore parallelismo diventa necessario superare il limite dei blocchi basici, aumentare il numero di stadi della pipeline (superpipelining), introdurre delle pipeline specializzate (pipeline multiple che operino in parallelo).

Un ulteriore passo in avanti può essere fatto implementando **architetture superscalari**, CPU che permettono di lanciare più istruzioni in contemporanea, riducendo il CPI ad un valore minore di 1.
Punti critici di questa architettura sono la decodifica parallela (difficile controllare tutte le alee delle diverse istruzioni in parallelo), l'issue delle istruzioni (controllo delle dipendenze dati), l'esecuzione fuori ordine ma con commit in ordine sequenziale (per evitare alee false).
Per gestire possibili blocchi dell'issue si usano tecniche di **shelving**, che disaccoppiano issue e controllo delle dipendenze usando buffer speciali (generalmente detti reservation stations).
Esistono 2 politiche di lettura degli operandi: **issue bound** (i buffer contengono i valori degli operandi) e **dispatch bound** (i buffer contengono solo gli identificatori dei registri contenenti gli operandi).

Per verificare la disponibilità nel RF e nelle reservation stations nasce un particolare registro di stato costituito da elementi di 1 bit, settati a 1 in caso di dato disponibile, lo **scoreboard**.
Per eliminare le dipendenze false la tecnica standard è la **ridenominazione dei registri**, che vedremo meglio in seguito.

Capitolo 5

Algoritmo di Tomasulo, predizione dinamica

5.1 Algoritmo di Tomasulo

5.1.1 Introduzione

Progettato prima della nascita delle cache (anni 60) per estrarre elevate prestazioni dall'unità floating point, l'algoritmo di Tomasulo viene tutt'ora implementato per gestire la ridenominazione dei registri in presenza di salti.
L'idea base consiste nel far memorizzare gli operandi alle reservation stations (RS), non appena questi siano disponibili. Le istruzioni lanciate attendono sulle RS che forniranno loro gli operandi, nel caso in cui successive scritture si sovrapponessero, solo l'ultima aggiornerebbe realmente il registro.
Tra i vantaggi di questa tecnica ci sono:

- La sostituzione dei riferimenti a registri con dei riferimenti a RS elimina la necessità di ridenominazione, evitando alee WAR e WAW.

- L'identificazione delle alee e il controllo dell'esecuzione sono distribuiti (tutte le RS vedono e interagiscono con il Common Data Bus);

Svantaggi derivanti dall'algoritmo sono:

- La complessità dell'hardware. Ogni RS deve infatti includere un buffer associativo molto veloce e una complessa logica di controllo;

- Calo di prestazioni causato dal CDB, che deve interagire con le RS.

Vedremo in seguito delle tecniche per migliorare il livello di parallelismo e incrementare le prestazioni.

5.1.2 Componenti delle Reservation Station

Ogni RS contiene:
1) un'istruzione lanciata e in attesa di esecuzione;
2) i valori degli operandi, se sono disponibili, oppure i nomi delle RS che li forniranno.

I componenti di una RS:
TAG (identificatore), Opcode
Vj/Vk (valori degli operandi)
Qj/Qk (puntatori alle RS che producono Vj/Vq
Busy (indica se la RS è occupata)

Load e Store buffer contengono ognuno un campo indirizzo, mentre il Register File possiede un campo Qj in cui è scritto il TAG della RS che contiene l'operazione il cui risultato verrà scritto nel registro.

5.1.3 Stadi dell'algoritmo

L'algoritmo di Tomasulo si può scomporre in 3 stadi:

1. **Issue:** la prima istruzione I della coda delle istruzioni in attesa viene inserita in una RS vuota, insieme agli operandi (se disponibili). Se gli operandi non sono nei registri, avviene una ridenominazione (implicita) che elimina le alee WAR e WAW.

2. **Esecuzione:** controllando il CDB si attendono i risultati delle istruzioni in esecuzione da cui attendo gli operandi, quindi I viene eseguita (evito alee RAW). L'attesa non è necessaria se gli operandi erano già disponibili.

3. **Scrittura dei risultati:** il risultato viene scritto nel CDB, a cui possono accedere immediatamente tutte le RS e lo store buffer.

Le Load e Store usano un'unità funzionale per calcolare l'indirizzo effettivo a cui riferirsi (offset rispetto ad un registro), inoltre sono (solitamente) mantenute in ordine di programma per evitare sovrascritture improprie. É possibile eseguire Load e Store fuori ordine solo se accedono a diversi indirizzi di memoria, in questo caso diventa però necessario calcolare gli indirizzi in ordine di programma, aumentando la complessità.

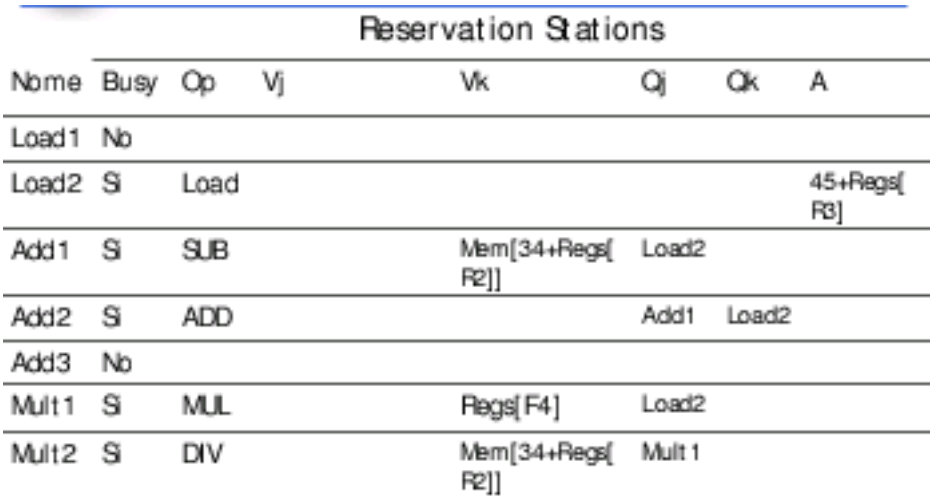
5.1.4 Esempio di funzionamento

Vediamo come funziona l'algoritmo, partendo da questa sequenza di codice:

$L.D \quad F6, 34(R2)$
$L.D \quad F2, 45(R3)$
$MUL.D \quad F0, F2, F4$
$SUB.D \quad F8, F2, F6$
$DIV.D \quad F10, F0, F6$
$ADD.D \quad F6, F8, F2$

Dalla figura 5.1 possiamo osservare lo stato delle diverse RS (ne supponiamo 2 per la LOAD, 3 ADD, 2 MUL), in seguito all'esecuzione della prima Load.
Si può notare come la RS *ADD1* (che contiene l'istruzione SUB), attende il primo operando (F2) dalla RS *LOAD2* (che infatti stà modificando il valore di quel registro), mentre conosce già il valore del secondo operando (F6).

L'ultima istruzione (ADD) potrebbe finire prima dell'istruzione precedente (DIV), perchè non ci sono conflitti. Questo avverrà (quasi) sicuramente, perchè MUL e DIV sono operazioni più lente e anche, in questo caso, dipendenti tra loro. Il possibile conflitto WAR tra la ADD (scrive F6) e la precedente DIV (legge F6) viene evitato poichè la RS *MULT2* ha memorizzato il valore di F6 precedentemente.
Anche il conflitto WAW che sarebbe emerso se qualche istruzione I avesse scritto un registro R dopo che un'istruzione I+1 l'aveva già modificato, viene evitato grazie all'uso di una tabella che memorizza lo stato dei registri, come vediamo nella Fig. 5.2.



Reservation Stations							
Nome	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Si	Load					45+Regs[R3]
Add1	Si	SUB		Mem[34+Regs[R2]]	Load2		
Add2	Si	ADD			Add1	Load2	
Add3	No						
Mult1	Si	MUL		Regs[F4]	Load2		
Mult2	Si	DIV		Mem[34+Regs[R2]]	Mult1		

Figura 5.1: `Esempio di funzionamento dell'algoritmo.`

Stato dei registri							
Campo	F0	F2	F4	F6	F8	F10	F12	...
Qi	Mult1	Load2		Add2	Add1	Mult2		

Figura 5.2:

5.2 Argomenti avanzati di esecuzione parallela

Le istruzioni eseguite in parallelo in genere sono *finite* (l'operazione è terminata, ma non è ancora stato scritto il risultato) fuori ordine, ma *completate* in ordine... per chiarire i diversi concetti relativi al parallelismo si distinguono 2 tipologie di consistenza sequenziale:

1. Consistenza del Processore; può essere debole (è possibile il completamento fuori ordine) o forte (istruzioni forzate dal ROB a venire completate in ordine di programma).

2. Consistenza della memoria; anche in questo caso si parla di consistenza debole (è possibile accedere alla memoria fuori ordine, se non ci sono dipendenze) o forte.

Indicando con W (weak) e S (strong) i due aspetti, ci sono 4 alternative di consistenza. Il più usato recentemente è il **modello SW**.

Usando il modello W nella consistenza della memoria diventa necessario stabilire un meccanismo di riordino delle Load/Store. Infatti è possibile il cosiddetto Load/Store bypassing, che permette lo "scavalcamento" di istruzioni load e store tra loro, purché non si violi alcuna dipendenza dati.
É possibile attuare una strategia speculativa (non aspetto di controllare eventuali dipendenze sugli indirizzi), che permette maggiore efficienza, dovuta anche al fatto che difficilmente l'indirizzo di una load sia uguale a quello delle store precedenti... in caso contrario diventa necessario "disfare" le load tramite comando *undo*.

5.2.1 Reorder Buffer (ROB)

Come possiamo vedere dalla figura 5.3, il ROB è un buffer circolare con un puntatore di testa (prossimo elemento libero) e di coda (prima istruzione che giungerà al commit), usato per garantire la consistenza sequenziale.
Ad ogni lancio di un'istruzione viene allocato nel ROB un elemento che la

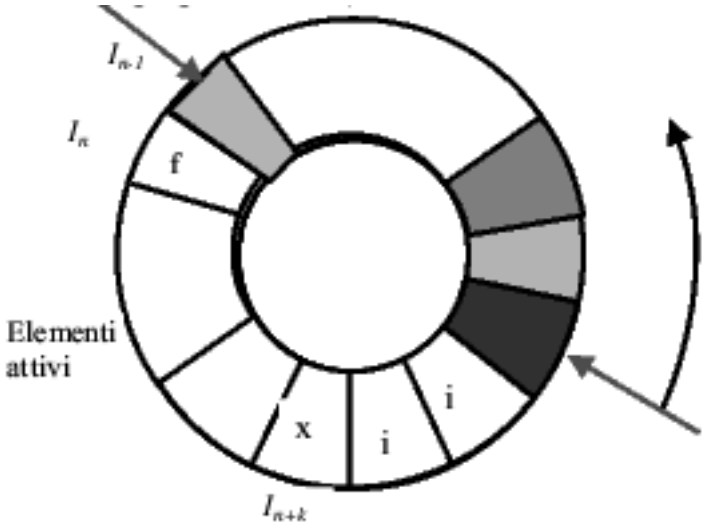


Figura 5.3: `Reorder Buffer`

rappresenta e indica anche il suo stato (lanciata[i], eseguita[x], finita[f]...). A seconda del valore di questo elemento si può stabilire se la relativa istruzione può giungere a commit.
Il ROB supporta sia l'esecuzione speculativa (aggiungo un campo agli elementi che indica se l'istruzione è speculativa, bloccando un eventuale loro commit) che la gestione delle eccezioni (sono accettate solo quando l'istruzione è *prossima al commit*, consentendo l'esecuzione delle eccezioni in ordine di programma).

5.2.2 Predizione dinamica

Di fronte a conflitti di controllo abbiamo visto che ci sono diverse strade da intraprendere (branch hoisting, branch taken...), ma ogni tipologia comporta perdita di molti cicli in caso di errore (vedi Fig. 5.4). Come si può ridurre la penalizzazione dovuta ad un errore di previsione sul risultato, migliorando il CPI?
Prima di tutto è necessario anticipare (nello stadio ID) il confronto del

Schema	Penalizzazione per salto incondizionato	Penalizzazione per salto non fatto	Penalizzazione per salto fatto
Svuota la pipeline	2	3	3
Predici salto fatto	2	3	2
Predici salto non fatto	2	0	3

Figura 5.4: `Schema delle penalizzazioni`

registro con l'indirizzo destinzione, riducendo in questo modo ad un solo ciclo la penalizzazione dovuta al conflitto di controllo.
Un punto di fondamentale importanza è il miglioramento dell'accuratezza sulla speculazione nei salti condizionati, tramite tecniche hardware che consentano la cosiddetta **predizione dinamica**.

Branch prediction buffer

La soluzione più semplice di predizione dinamica si basa su branch–prediction buffer, una piccola memoria (massimo 4K) indirizzata mediante i bit meno significativi dell'indirizzo di un'istruzione di salto, contenente un bit che indica se un salto condizionato è stato recentemente fatto oppure no.

Contenuto della memoria:

$|xxxxxxxy|$
$|xxxxxxxy|$ x = bit meno significativi dell'istruzione di salto.
$|xxxxxxxy|$ y = bit indicante se l'ultima volta il salto è stato fatto.

Ovviamente la predizione può non essere corretta, ma sapere cosa è avvenuto incontrando quell'istruzione la volta precedente permette di modificare dinamicamente la speculazione.
Il migliore schema di predizione basato su branch–prediction buffer consiste nell'usare 2 bit y che consentano di non cambiare stato ad ogni speculazione errata, allungando la "memoria" e migliorando l'accuratezza, per esempio nei cicli.
Per aumentare ancora l'accuratezza della predizione a volte viene considerato il comportamento recente anche di altri salti precedenti → predittori correlanti.

Branch target buffer e ROB

É possibile migliorare ulteriormente le prestazioni fornendo al processore un flusso di istruzioni su banda larga (4-8 istruzioni per ciclo), ma per fare questo è necessario superare la barriera dei salti ed evitare il limite imposto dal blocco basico. Una soluzione adeguata consiste nel predirre l'indirizzo destinazione del salto, tramite branch target buffer (BTB).
Il BTB è una cache totalmente associativa che memorizza l'indirizzo (PC) predetto per l'istruzione successiva a quella di salto.
Funzionamento: alla prima decodifica delle istruzioni inserisco nel BTB gli indirizzi delle istruzioni di salto con i relativi PC della prossima istruzione "predetta":

$|xxxxyyyyz|$ x = byte dell'istruzione di salto.
$|xxxxyyyyz|$ y = byte dell'istruzione predetta successiva al salto.
$|xxxxyyyyz|$ z = 1-2 bit indicanti se eseguire o meno il salto.

In seguito nello stadio IF di ogni istruzione si accede al BTB usando come indice l'indirizzo dell'istruzione letta (x). Se l'accesso è una HIT (cioè l'indirizzo è presente nella cache) allora l'istruzione letta è un salto (nello stadio IF il processore non ha ancora decodificato e non sa che tipo di istruzione è quella appena letta!!), quindi ottengo dal BTB l'indirizzo "predetto" dell'istruzione successiva (y), quindi è possibile leggere l'istruzione corretta prima dello stadio ID e superare quindi i limiti del blocco basico.

Inserendo queste modifiche hardware diventa importante creare meccanismi di gestione delle speculazioni errate, estendendo la predizione dinamica con tecniche di *undo* che permettono di disfare gli effetti di sequenze speculate in modo errato.
L'idea chiave consiste nell'eseguire fuori ordine ma fare commit in ordine, portando alcune funzionalità addizionali al ROB, che ora dovrà eseguire le operazioni di ridenominazione al posto delle RS (i risultati delle diverse istruzioni vengono etichettati col numero dell'elemento del ROB invece che con l'identificatore della RS).
Quando un salto con predizione errata giunge alla testa del ROB, si trova l'errore e il buffer viene svuotato, permettendo una semplice ripresa dalla prima istruzione corretta.

Riassumendo, ecco i 4 passi nell'esecuzione delle istruzioni:

1. **Issue**: se ci sono RS libere e posizioni vuote nel ROB, un'istruzione viene lanciata. Invio gli operandi disponibili e la posizione del ROB in cui verrà allocato il risultato alla RS scelta.

2. **Execute**: se non ci sono operandi disponibili, monitoro il CDB nell'attesa dei valori. Se non ci sono alee RAW eseguo l'operazione.

3. **Write Result**: scrivo il risultato su CDB, da cui verrà inserito nella corretta posizione del ROB. Libero la RS.

4. **Commit**: quando l'istruzione raggiunge la testa del ROB scrivo il risultato nel registro (o aggiorno la memoria in caso di store) e can-

cello l'elemento dal buffer.

Anche per quanto riguarda le eccezioni, il loro riconoscimento avviene solo in prossimità del commit, per evitare eccezioni sollevate in modo speculativo.

In alternativa al ROB e RS talvolta viene utilizzato un insieme di registri più grande per consentire un'ampia ridenominazione dei registri, in cui salverò i diversi risultati "intermedi".
Questa tecnica è più semplice ma rende complessa la deallocazione dei registri e comporta un mapping dinamico dei due RF.

5.3 in sintesi...

L'**algoritmo di Tomasulo** viene implementato per gestire la ridenominazione anche in presenza di salti. Gli operandi vengono memorizzati dalle RS non appena disponibili, in modo tale che le istruzioni non debbano più riferirsi ai registri, evitando alee false.
L'algoritmo è diviso in 3 fasi: issue, esecuzione, scrittura dei risultati, tutte che operando usando un Common Data Bus che interagisce con ogni RS e permette di identificare le alee in modo distribuito.

Per quanto riguarda la consistenza sequenziale di esecuzione si usa generalmente il modello SW (istruzioni completate in ordine di programma, accesso alla memoria anche fuori ordine), implementato tramite l'uso di un buffer circolare detto ROB (**ReOrder Buffer**), e speculazione delle load/store.
Ad ogni lancio di un'istruzione viene inserito un elemento nel ROB che ne indica lo stato e permette di decidere se e quando gestire il commit. Il ROB supporta anche speculazione e gestisce le eccezioni.

Tramite particolari elementi hardware è possibile introdurre la **predizione dinamica**, la possibilità di adattare la speculazione sui salti a run–time, in modo da migliorarne l'accuratezza.
La soluzione più semplice si basa su **branch–prediction buffer**, una picco-

la memoria indirizzata dai bit meno significativi di un istruzione di salto, contenente 1–2 bit che indicano se il salto era avvenuto o meno nelle ultime 1–2 occorrenze dell'istruzione.
Una soluzione alternativa si basa su **branch–target buffer**, una cache tot. associativa che memorizza l'indirizzo predetto per l'istruzione successiva a quella di salto. Questa tecnica consente di non perdere nemmeno un ciclo per il calcolo del prossimo PC in caso di branch–taken, superando i limiti del blocco basico.
Tramite ROB vengono gestite le speculazioni errate.

Capitolo 6

Architetture VLIW e...

6.1 I limiti dell'ILP

Per migliorare ulteriormente le architetture hardware ci chiediamo quali sono i limiti del parallelismo, e ipotizziamo di togliere questi vincoli:

1. Ridenominazione dei registri possibile all'infinito (elimino le false dipendenze dei dati)

2. Predizione dei salti perfetta (nessuna dipendenza di controllo).

3. Analisi degli indirizzi in memoria (aliasing) perfetta.

Inoltre imponiamo continua la schedulazione di qualsiasi istruzione (niente stalli), permettiamo issue di un numero illimitato di istruzioni, gli accessi a memoria sono di un ciclo di clock.

I risultati (MOLTO ottimistici) mostrano che il parallelismo estratto (la frequenza media di lancio delle istruzioni) va da 16 ad un massimo di 150, con codice ottimizzato.
Il problema nasce osservando come il lancio di "sole" 50 istruzioni richiede 2450 confronti per determinare le dipendenze RAW.
Altri limiti delle CPU reali sono il basso numero di unità funzionali, dei bus e delle porte dei registri.

6.2 Architettura P6

L'architettura P6 è la base dei Pentium Pro, II e III.
Alla base del funzionamento di P6 c'è una microarchitettura che elabora micro–operazioni (uops), eseguite dalla pipeline.

Come vediamo dalla Fig. 6.1, il P6 si basa su 3 unità:

- Unità di lettura/decodifica: converte le istruzioni in uops e le inserisce nel Pool in ordine, decodificate.
- Unità di dispatch/esecuzione: lancia fuori ordine le uops verso le RS (max 4 per ciclo).
- Unità di retire (commit): riordina le istruzioni, porta a commit i risultati, risolvendo le speculazioni (max 3 commit per ciclo).

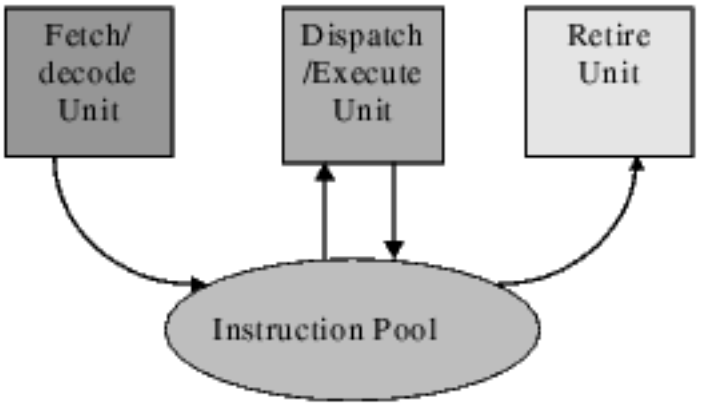


Figura 6.1: `Architettura P6`

La struttura della pipeline è a 14 stadi, viene usato un predittore di salti a 2 livelli di 512 elementi, la ridenominazione utilizza 40 registri virtuali, le RS sono 20 e il ROB può contenere 40 elementi.
La pipeline di esecuzione utilizza da 1 ciclo per le operazioni di ALU semplici fino a 32 per le divisioni tra floating point.
A causa delle speculazioni non esatte vengono lanciate il 20% delle operazioni in più del necessario (si usa predizione statica). L'efficienza massima è di 5 micro–ops per ciclo.

6.3 Architettura Pentium IV

Con l'architettura Pentium IV viene incrementata la frequenza di clock fino a 2GHz, inoltre sono inseriti 3 meccanismi di prefetch, sia hardware (basandosi sul BTB e dalle I–cache), sia software sulla D–Cache.
La ridenominazione è più ampia (128 possibili risultati), ALU e D–Cache sono disegnate in modo aggressivo e la BTB viene ampliata di 8 volte rispetto a P6.

Quest'architettura si basa su Execution Trace Cache, una memoria che

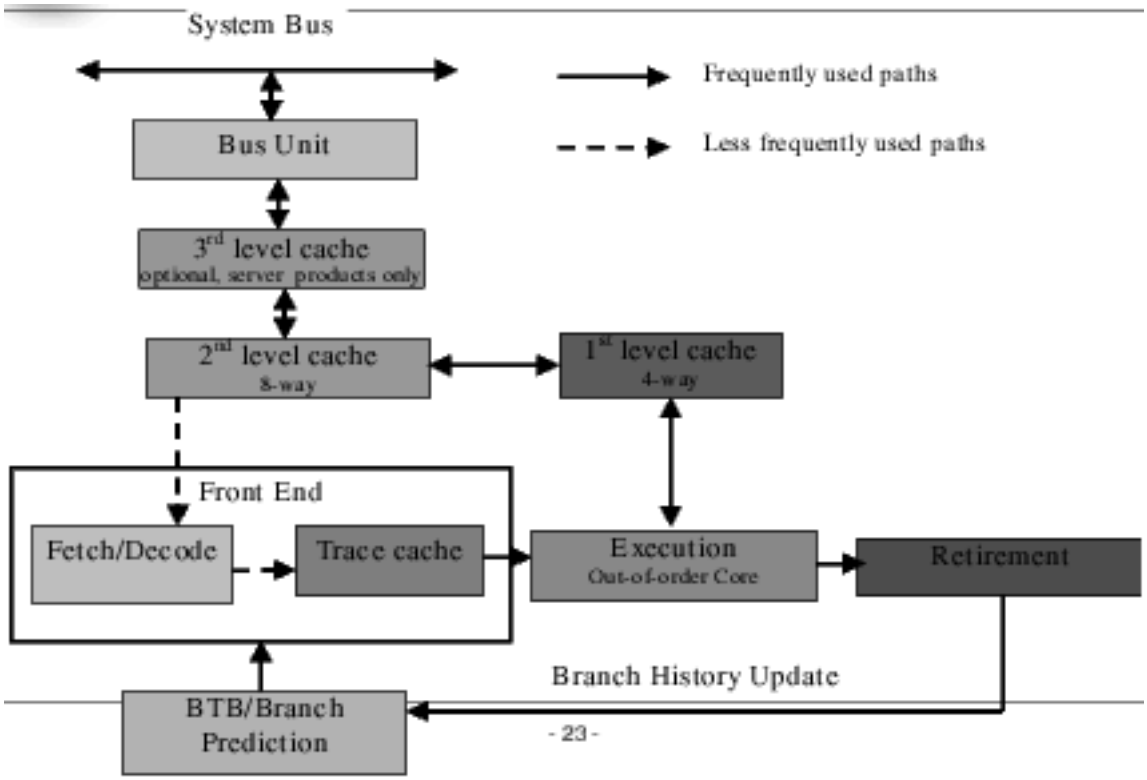


Figura 6.2: `Architettura P IV, con Trace-Cache`

permette di identificare una sequenza dinamica di istruzioni (salti inclusi) da caricare in un blocco (vedi Fig. 6.2).

6.4 Architetture alternative: VLIW

Esistono processori costruiti seguendo una filosofia diversa, sono CPU con scheduling statico, chiamate VLIW (Very Long Instruction Word).
In queste architetture è il compilatore che identifica le istruzioni eseguibili in parallelo, assemblandole in pacchetti che verrano poi decodificati ed eseguiti dal processore. Lo spostamento del controllo sul compilatore consente di risparmiare il 30–35% dello spazio sulla CPU, riducendone ampiamente la complessità. Inoltre il compilatore può vedere una "finestra di programma" molto ampia, analizzandola con più precisione, anche se in modo statico.

Come vediamo dalla Fig. 6.3, le istruzioni di questa CPU sono molto lunghe (es: 128 bit), poichè composizione di diverse istruzioni singole dette sillabe (es: 4 istruz. da 32 bit), che vengono eseguite in parallelo. Tutte le sillabe hanno accesso simultaneo al RF e seguono un percorso dedicato nel datapath della CPU (4 ALU, 4 porte nel RF...).

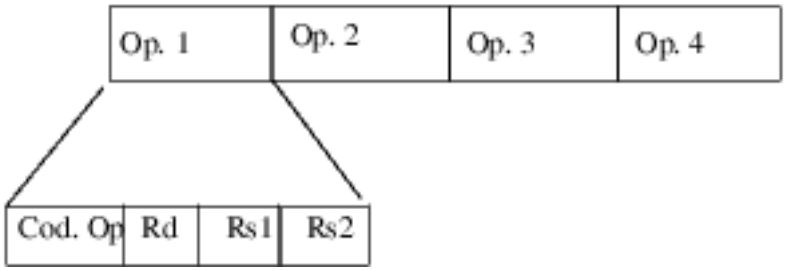


Figura 6.3: `Pacchetto di 4 istruzioni VLIW`

L'architettura VLIW è semplice e permette di rendere più veloce il clock, ma il codice oggetto creato dal compilatore diventa altamente dipendente dall'architettura (in certi casi il codice oggetto può essere tradotto tramite emulazione).
Una limitazione è dovuta alla possibilità di avere solo un'operazione che modifica il flusso di controllo per ogni pacchetto. Un altro problema proviene dai D–Cache Miss, che pongono in stallo l'intera CPU, a meno di utilizzare complesse strutture di gestione della cache, come un prefetch buffer

o uso di load speculative, che però aumentano la complessità del controllo, riducendo uno dei maggiori vantaggi di questo tipo di processori.

Esistono due tipologie di esecuzione in un'architetture VLIW:

- **esecuzione elementare**: ogni pacchetto (bundle) è eseguito in modo sequenziale da 5 unità funzionali.
- **esecuzione predicata**: le dipendenze di controllo sono convertite in dipendenze di dati, creando blocchi basici più grandi e ottimizzabili, aumentando però in modo rilevante la dimensione del codice.

Vediamo nel dettaglio.

L'*esecuzione elementare* mantiene la sequenzialità, costringendo tutte le sillabe del pacchetto a giungere al Write–Back nello stesso istante. Quindi in caso di operazioni a latenza alta (MUL, DIV...), tutte le operazioni del bundle sono vincolate e devono attendere (tramite *nop*) la terminazione dell'istruzione più lenta.

L'*esecuzione predicata* invece da' al compilatore un'alternativa ai salti condizionati, permettendo la conversione delle dipendenze di controllo in dipendenze dati.
Quest'operazione è resa possibile inserendo nelle istruzioni una condizione che viene valutata come parte dell'esecuzione dell'istruzione. In caso di condizione falsa non avviene un flush ma l'istruzione continua come se fosse una nop. Per questo scopo sono quindi create delle nuove istruzioni, ad esempio $CMOVZ\ r1, r2, r3$ (conditional move zero), che sposta r1 in r2 solo se r3 è zero, sostituisce la coppia $BNEZ\ r3\ \ ADD\ r1, r2, 0$ (somma r2+0 e sposta in r1 solo se r3 diverso da 0).

É possibile avere un'esecuzione **completamente predicata**, aggiungendo ad ogni istruzione un operando booleano che si riferisca a predicati definiti da altre istruzioni, in modo da gestire efficientemente salti annidati.
Il codice, anche se diviso in più blocchi basici, viene trasformato in un unico blocco in cui le istruzioni vengono eseguite solo se rispettano determinate condizioni, determinate dai conflitti di controllo. L'aumento di

latenza che ne deriva è comunque inferiore ai vantaggi causati dal mancato flushing delle istruzioni errate.

Un altra sotto–categoria dell'esecuzione predicata è l'esecuzione **parzialmente predicata**, che consiste nell'eseguire senza condizioni tutte le singole operazioni dei vari blocchi basici, selezionando in seguito i valori finali accettabili.

6.4.1 Architettura Itanium

É la prima delle CPU a 64 bit Intel.
Le istruzioni vengono ottimizzate sia dal compilatore che dinamicamente dall'hardware, per garantire un'efficienza massima, l'unità di controllo è quindi più complessa di quella di un VLIW "base".
Non supporta Reservation Stations nè ROB, le dipendenze vengono gestite dal compilatore.

6.4.2 Philips Trimedia

CPU progettata per elaborazione dei segnali e delle immagini, è un'architettura VLIW "pura", con scheduling totalmente statico.
Tutte le operazioni sono predicate con il valore di un registro che, se posto a zero, non permette l'esecuzione dell'istruzione.

6.5 Processori Embedded – DSP

Le architetture per sistemi embedded contengono modifiche fondamentali che puntano alla riduzione dello spazio e della potenza necessaria, a discapito delle performance.
L'unità FloatingPoint non viene generalmente usata, si usa un'aritmetica *saturante*, che viene semplificata per valori molto grandi o molto piccoli.

Il caso più rilevante di processori per sistemi embedded è DSP (Digital Signal Processor), dei processori che trattano insiemi di dati in modo infinito e continuo. In questo tipo di architettura vengono usati buffer circolari per i dati, oltre all'introduzione di un costrutto *repeat* che consenta

di realizzare cicli interni in modo molto efficiente (autoincremento hardware).
Spesso DSP tratta molti dati ma corti (8bit), quindi un'istruzione da 32 bit può essere partizionata in 4 parti per fare in modo che contenga fino a quattro operandi (Multiply and ACcumulate–MAC).
Vitale è la gestione della potenza, in modo di raggiungere elevate prestazioni consumando il meno possibile, poiché generalmente questi microprocessori funzionano con batterie di limitata capacità. Una tecnica molto usata è la riduzione selettiva della frequenza, della tensione di alimentazione, fino addirittura a bloccare il clock nei latch che non contengono dati rilevanti.

La tendenza per questi microprocessori indica un aumento quasi esponenziale dei transistor su ogni chip, permettendo di aumentare il numero di cache e, ultimamente, consentendo l'inserimento di più processori sullo stesso chip, connessi in rete tra loro.

6.6 Architetture TLP

Le architetture TLP (thread level parallelism) sfruttano il multithreading, la condivisione delle unità funzionali di una CPU da parte di più thread, mostrando quindi al SO l'unico processore fisico come se fosse un multiprocessore.
Ci sono due approcci al multithreading:

- a grana fine: ad ogni ciclo di clock si commuta da un thread all'altro.
- a grana grezza: la commutazione tra thread avviene solo in corrispondenza di stalli lunghi (miss sulla cache), non rallentando il singolo thread ma limitando il vantaggio sugli stalli brevi.

Una tecnica, detta simultaneus multithreading (SMT) sfrutta contemporaneamente ILP e TLP, grazie allo scheduling dinamico e alla ridenominazione dei registri, permettendo l'esecuzione simultanea di più istruzioni da ogni thread sulla CPU, come vediamo dalla Fig. 6.4.
Nella figura MT grezzo attende un evento ad alta latenza prima di cam-

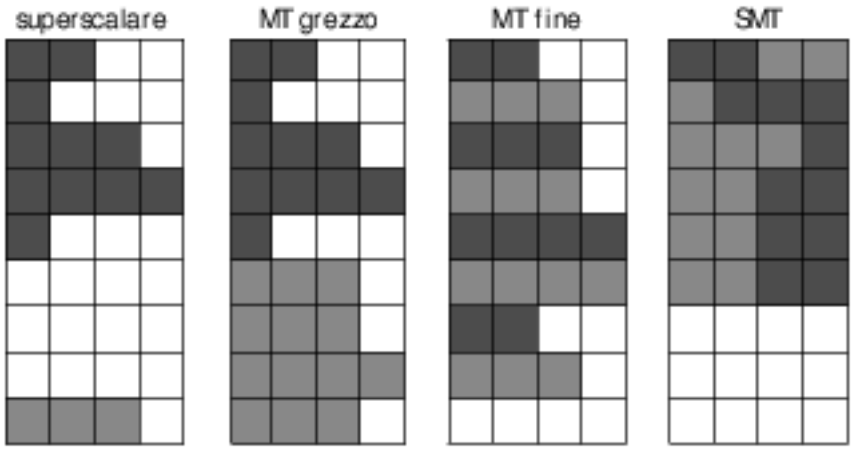


Figura 6.4: `Confronto tra diversi processori`

biare contesto, MT fine alterna i due thread aumentando l'efficienza, ma la CPU SMT, eseguendo ad ogni ciclo di clock il massimo delle istruzioni disponibili, permette un elevato parallelismo e grandi prestazioni (ma con alcuni problemi di cache trashing).

6.6.1 Power4 e Power5

Sono architetture IBM a due processori che implementano SMT a due vie, elevando il multithreading a 4 thread massimi, anche se le prestazioni non crescono molto a causa di cache trashing (ogni thread richiede diversi dati, sostituendo quelli altrui...).
Per questo motivo Power 5 include anche la possibilità di funzionamento a singolo thread, mentre vengono implementate tecniche di bilanciamento dinamico delle risorse per monitorare gli accessi alla cache.

(se interessa altro, vedi dispense...)

6.7 in sintesi...

Capitolo con analisi di diverse architetture reali:
P6: basata su micro–operazioni eseguite in parallelo e fuori ordine, con predizione dei salti statica, 40 registri virtuali e 20 RS usate per la ride-

nominazione.
Pentium IV: vengono inseriti meccanismi di prefetch hardware e software, nasce la execution Trace Cache, una memoria che permette di identificare una sequenza di istruzioni da caricare in blocco.

Quello delle **architetture VLIW** è un modo alternativo di costruire un processore, in cui è il compilatore a identificare le istruzioni parallele, inserendole in pacchetti che verrano poi decodificati ed eseguiti da un processore molto semplice, poiché non deve più individuare le alee e gestire le finestre di issue ed esecuzione.
Il codice oggetto creato dal compilatore diventa altamente dipendente dall'architettura, inoltre ritorna alla luce il problema delle dipendenze di controllo, che rallentano l'esecuzione di tutto il pacchetto di istruzioni parallele, a meno di usare **esecuzione predicata**.
Tramite esecuzione predicata è possibile convertire le dipendenze di controllo in dipendenze dati, estendendo i blocchi basici a discapito della dimensione del programma.

Itanium: CPU a 64bit Intel, è un compromesso tra le architetture standard e VLIW, poichè le istruzioni sono ottimizzate sia staticamente che dinamicamente, ma senza RS nè ROB.
Philips Trimedia: architettura VLIW "pura", con scheduling totalmente statico ed esecuzione predicata.

Nei **processori embedded** il punto critico diventa il consumo di potenza e la riduzione dello spazio, a discapito delle performance. Nei processori DSP i dati sono di 8 bit (4 per istruzione), la frequenza viene ridotta selettivamente per risparmiare energia.

Esistono architetture che sfruttano il **multithreading**, la condivisione di più unità funzionali da parte di più thread. Esistono due approcci: a grana fine (si commuta tra i diversi thread ad ogni ciclo di clock), a grana grezza (la commutazione avviene solo in caso di stalli).
Una tecnica, detta **SMT** sfrutta il multithreading e l'ILP, permettendo l'esecuzione simultanea di più istruzioni da diversi thread, aumentando ulteriormente l'efficienza.

Power4 e 5: architetture IBM che implementano SMT a due vie, affrontano il problema del cache–trashing introducendo tecniche di bilanciamento delle risorse.

Capitolo 7

I sistemi multiprocessore

7.1 Introduzione

Per ottenere prestazioni sempre più elevate è necessario abbandonare ILP e collegare diversi processori in un sistema complesso, ottenendo architetture parallele a livello di processo e di thread.
La tassonomia tradizionale dei multiprocessori (1966) comprende:

- SISD: single instruction, single data stream. Il processore singolo convenzionale.
- SIMD: single instruction, multiple data stream. Una stessa istruzione viene eseguita su più processori con dati differenti (calcolatori vettoriali).
- MISD: multiple instruction, single data stream. Diverse istruzioni sono eseguite usando gli stessi dati.
- MIMD: multiple instruction, multiple data stream. Microprocessori standard, eseguono diverse istruzioni su diversi blocchi di dati.

Il **modello MIMD** è emerso come l'architettura prescelta per i multiprocessori di tipo generale a causa della sua flessibilità e perchè può essere costruito come insieme di più CPU standard, limitando le spese.
Per sfruttare un MIMD con n processori è necessario avere n thread/processi indipendenti, creati dal programmatore o dal compilatore (parallelismo identificato via software).

7.2 Architetture MIMD

Le architetture MIMD sono divise in due classi:

- **architetture centralizzate a memoria condivisa**: i processori sono collegati e comunicano tramite un bus. Vengono usate grandi cache private per ogni CPU, mentre la singola memoria primaria ha una relazione simmetrica con tutti i processori e tempo di accesso uniforme (**SMP**, chiamati anche UMA).
  Non vengono usate con più di 10–12 processori in parallelo.

- **architetture a memoria distribuita**: coppie processore/memoria formano dei nodi interconnessi, distribuiti. L'informazione deve quindi essere replicata, l'interazione in due modi diversi a seconda dell l'approccio architetturale scelto:
  1) sistemi basati su memorie fisicamente separate ma indirizzate come un unico spazio degli indirizzi condiviso (DMS, chiamati anche **NUMA**).
  (Facile programmazione, non si usa la banda di comunicazione, compatibile con meccanismi molto usati);
  2) multicalcolatori con muduli CPU–memoria separati, comunicano tramite messaggi (**cluster**).
  (La banda della rete diventa un fattore molto importante, l'hardware è più semplice, la sincronizzazione e la comunicazione avvengono tramite meccanismi ben definiti e di facile comprensione).

7.2.1 Architetture a memoria distribuita

Le architetture a memoria distribuita sono molto scalabili e adatte al parallelismo massiccio, la sincronizzazione può essere un collo di bottiglia ma viene implementata da meccanismi semplici ed efficienti.
Se un processo attende una zona di memoria non presente in locale può venire sospeso o continuare in uno stato di busy–wait finchè la risposta non diventa assolutamente necessaria. Nel primo caso è necessario inserire un efficiente meccanismo di context–switching, mentre nel secondo

caso sono usate tecniche di monitoraggio sulle comunicazioni in attesa.

Uno svantaggio di questa architettura nasce dall'impossibilità di comunicare tramite strutture dati condivise, così come il sorgere di problemi di deadlock e load balancing tra i diversi processori.
Le reti di interconnessione sono generalmente statiche (connessioni punto–punto).

7.2.2 Architetture a memoria condivisa

Nelle architetture a memoria condivisa il codice e i dati non vengono partizionati e sono condivisi tra tutti i processori, viene quindi evitato lo spostamento dei dati tra processi, migliorando le prestazioni (nessuna latenza per scambio di messaggi).
Tra gli svantaggi troviamo la mancanza di scalabità causata dai bottleneck della memoria condivisa, l'introduzione di cache locali per evitare rallentamenti provoca problemi di coerenza dati.
Le reti di interconnessione sono dinamiche (è possibile riconfigurare la rete), bisogna fare attenzione al possibile sovraccarico usando reti ad alta efficienza.

7.3 Cache Coherence

Per mantenere la consistenza dei dati si impone la **cache coherence**, rispettando 3 proprietà:

1. se un processore P scrive nella posizione X e nessuno modifica questa zona, quando P tenterà di leggere da X verrà restituito il valore inserito precedentemente;

2. se P legge un dato X modificato da un altro processore Q, leggerà il valore inserito da Q;

3. le scritture a una stessa posizione sono viste da tutti processori nello stesso ordine.

In un multiprocessore è normale che un programma abbia più copie degli stessi dati in diverse cache private, per mantenere la coerenza bisogna

affrontare i problemi causati dalla migrazione dei processi e dalla replicazione e condivisione di dati scrivibili.
Nascono così diversi protocolli hardware:

Snoopy Based: tutti i controllori della chache fanno un monitoraggio continuo del bus per determinare se hanno una copia di un blocco che viene richiesto da altri processori (adatto per soluzioni basate su bus).

Directory Based: lo stato di condivisione di un blocco di memoria fisica viene mantenuto in un'unica posizione, detta directory (adatto a soluzioni scalabili).
I possibili stati di un blocco in cache sono: condiviso, non in cache, esclusivo

Esistono due politiche adatte a mantenere i vincoli di coerenza:

- write invalidate: al momento della scrittura vengono invalidate tutte le copie di un dato che non sono aggiornate (serializzo tutte le operazioni).
- write update: dopo una scrittura tutte le copie non aggiornate, presenti su altri processori, vengono aggiornate in broadcast (metodo poco usato a causa del consumo di banda).

Per misurare le prestazioni di un sistema multiprocessore si usano 3 misure fondamentali: la banda di comunicazione, la latenza (overhead e tempo di trasmissione dei messaggi), i meccanismi di mascheramente della latenza (la sovrapposizione di comunicazioni e computazione permette di migliorare l'efficienza).

Torna importante la legge di Amdahl:

$$\text{Speedup} = \frac{1}{\frac{FrazParallela}{numProcessori} + (1 - FrazParallela)}$$

autore: Riva 'ZaX' Samuele