

CPU ad elevate prestazioni

Esistono varie tecniche per migliorare le prestazioni delle macchine pipelined.

- **Scheduling dinamico:** riduce stalli sui dati
- **Branch Prediction dinamica:** riduce stalli di controllo
- **Issue multipla:** riduce CPI-ideale
- **Speculazione:** riduce stalli in generale.

E' necessario maggior parallelismo dal programma, ovvero identificare e risolvere le dipendenze (tramite la rischedulazione). Questo accade principalmente in 2 modi: **staticamente** e **dinamicamente**.

- **Staticamente**, ho una finestra software che ispeziona un segmento di codice oggetto, lo ricompila (compilatore ILP) e lo invia all'unità hardware dopo averlo ottimizzato. Tipico di architetture VLIW (i processori si aspettano di ricevere istruzioni già ottimizzate)
- **Dinamicamente**, tramite 2 finestre hw (di issue e d'esecuzione), dalle quali vengono estratte istruzioni e inviate verso l'unità di issue istruzioni nel processore. Il procedimento è svolto a runtime e richiede una unità di controllo più complessa.

Scheduling dinamico: si leggono le istruzioni e le si lanciano (issue) in ordine di programma. Si esegue non appena sono disponibili gli operandi (anche fuori ordine). Le istruzioni nella issue window vengono controllate rispetto alle dipendenze con quelle nella issue window e nella exec window. L'unità di issue decide quindi chi lanciare (0-1 se proc. Scalare, 0-1 o più è Superscalare). In ogni caso comunque anche il compilatore fornisce un'ottimizzazione parallela del codice.

Il **parallelismo** (# di istruzioni che posso lanciare in parallelo) è identificato dalle istruzioni prive di dipendenze all'interno di un blocco basilico.

Blocco basilico: sequenza di istruzioni con un solo entry point e un solo exit point (punto di diramazione). Generalmente costituiti da meno di 10 istruzioni e dipendono dal tipo di programma (general purpose o specifico). Si può sfruttare l'ILP anche attraversando più blocchi basilici (con meccanismi avanzati che si vedranno in seguito).

Si sfruttano **parallelismo a livello di ciclo, utilizzo di istruzioni vettoriali** (operanti su sequenze di dati), oppure si aggirano le barriere date dai blocchi basilici tramite **esecuzione speculativa**.

Nelle macchine scalari posso cercare di ridurre il tempo di ciclo, rendendo più profonde le pipelines (con stadi più semplici a latenza minore: **superpipelining**). Più di tanto però non si può fare (buffer inter-stadio, clock skew, etc...). Peggiora la perdita di prestazioni su alee dati (più bolle necessarie perchè la pipeline è più profonda), salti condizionati più lenti (flushing più costoso).

Il riordino delle sequenze di istruzioni nelle macchine scalari (**dinamico**) permette di gestire dipendenze non identificabili a compile

time, ed eseguire codice compilato su CPU con pipelines molto differenti.

Multipipelines

Posso pensare di migliorare introducendo un certo numero di pipelines specializzate, oppure di replicare più pipelines (o entrambe le soluzioni). Si sceglie cosa replicare in base ai risultati dei benchmark.

Si possono parallelizzare le pipes, mantenendo però lettura e decodifica di una sola istruzione per volta (se no l'architettura diventa superscalare). Successivamente si instradano le istruzioni sulla pipeline più appropriata (es. Istruzioni floating point).

E'possibile che avvengano esecuzioni fuori ordine perchè le pipelines dedicate hanno latenze differenti. Bisogna garantire la consistenza sequenziale delle operazioni.

1. Si impedisce il writeback alle istruzioni della pipeline più lenta (fp), che viene compiuta sotto controllo software. Soluzione semplice e antiquata
2. lock-stepping: si sincronizzano forzatamente le pipelines, rallentando con bolle la pipe a minor latenza quando necessario (in caso di alea).
3. Reordering: si usa quando le pipeline sono più di due. Meccanismo che forza la scrittura ordinata in memoria (o nel RF) anche se le istruzioni sono state completate fuori ordine.

Istruzioni **Load/Store**: bisogna gestire il calcolo dell'indirizzo, operazione potenzialmente complessa nel caso di macchine CISC. Tradizionalmente le operazioni L/S venivano inviate sulla pipeline principale, più efficacemente si inviano in parallelo su un'unità autonoma di Load/Store (necessità di aggiungere 2 porte aggiuntive sul RF per permettere di calcolare in parallelo anche le operazioni aritmetiche)

In ogni caso la banda della I-Cache è limitata ad una sola istruzione per volta.

Architetture Superscalari

Nelle architetture superscalari non si replicano solamente le unità funzionali, ma si permette alla CPU di leggere, decodificare e lanciare più istruzioni per ciclo di clock. Si estrae dal codice un maggior livello di parallelismo di istruzione.

- E'possibile **leggere e decodificare simultaneamente più istruzioni**;
- Replicazione delle **pipeline d'esecuzione** (teoricamente n repliche = n istruzioni lanciate in esecuzione simultaneamente).
- Esistono meccanismi che **garantiscono l'aggiornamento dello stato equivalentemente al modello sequenziale**

Idealmente, per n potenziali istruzioni simultanee $CPI = 1 / n$

La banda della cache ovviamente deve essere superiore ad una parola!

Ecco le fasi per ottenere un ILP significativo dopo aver eseguito un **fetch parallelo**:

1. Decodifica parallela

2. Instr.Issue Superscalare: si identificano le possibili dipendenze e si inviano le istruzioni alle diverse pipeline. Maggiore è la frequenza di lancio, maggiori sono i problemi dovuti alle alee (più probabili rispetto ai processori scalari per via del maggior parallelismo, aumenta la probabilità di conflitto). Bisogna fornire la CPU di hardware opportuno per aumentare al massimo l'**effective issue rate**, garantendo un'alta probabilità di poter lanciare, in ogni ciclo, n istruzioni indipendenti. Questa probabilità viene migliorata attraverso le **instruction issue policies**.

3. Esecuzione parallela: vengono coinvolte più unità funzionali, ed è iniziata sulla base della disponibilità dei dati e non sull'ordine originale del programma (scheduling dinamico)

4. Esecuzione fuori ordine: indotta dalla presenza di unità funzionali indipendenti, i risultati vengono presentati in un ordine quasi sicuramente differente da quello presente nel programma originale. Tramite opportune tecniche si aggiorna lo stato della macchina (commitment delle istruzioni) seguendo l'ordine corretto del programma iniziale. Ovviamente la consistenza va conservata anche in caso di eccezioni.

Lo scheduling è compiuto direttamente dall'hardware di controllo. Lo speedup che si ottiene è così definito

$$Sp = (T1/Tp)$$

dove T1 è il tempo di esecuzione su una CPU scalare e Tp è il tempo di esecuzione su una CPU superscalare (e codici ottimizzati per le relative configurazioni).

Decodifica parallela: divisa in due fasi.

1. Issue: decodifica e verifica alee

2. Lettura degli operandi: si attende finchè non si risolvono le alee.

Le istruzioni possono attraversare lo stadio fuori ordine già nella fase di lettura operandi. In un processore superscalare, in un solo ciclo, la CPU decodifica più istruzioni e verifica le dipendenze entro la finestra d'esecuzione e fra le finestre di esecuzione ed issue, ovvero si necessitano molti confronti paralleli. Per ovviare al problema senza ritardare eccessivamente il tempo di ciclo si usa la precodifica (ovvero si sposta parte della decodifica nelle gerarchie superiori di memoria - cache L1 ed L2). Leggendo dalla cache L2 e prima di scrivere nella cache L1, l'unità di predecodifica aggiunge un numero di bit informativi per indicare la classe dell'istruzione, il tipo di risorse richieste e il precalcolo del salto (in alcuni processori es. Ultrasparc)

Issue superscalare: ci sono due aspetti principali:

1. **Issue policy:** come gestire le dipendenze
2. **Issue rate:** numero massimo di istruzioni per ciclo che la CPU può inviare (2-4 generalmente)

Le politiche di issue comprendono 4 aspetti: gestione delle **false dipendenze**, gestione delle **dipendenze di controllo** non risolte, tecniche di **shelving** e gestione di **blocco** dell'issue.

False dipendenze dati: riguardano alee WAR / WAW tra istruzioni in execution window e istruzioni in attesa di lancio o nella issue window. Si usa ridenominazione dei registri per le dipendenze dovute ai riferimenti al registro.

Dipendenze di controllo: gestione dei salti condizionali in assenza del risultato della condizione. Si attende oppure si adotta esecuzione speculativa.

Shelving: tecniche che evitano il blocco della issue. Si consente che istruzioni dipendenti blocchino istruzioni successive (blocking issue) oppure si evita il blocco tramite shelving.

- **Blocking issue:** si effettua un controllo bloccante sulle istruzioni decodificate nella finestra di issue.
- **Shelving:** disaccoppiamento di issue e controllo delle dipendenze. Si usano buffer speciali (detti **reservation station**) disponibili per ogni unità funzionale, e le istruzioni vengono accodate lì sopra. I controlli di dipendenza sono eseguiti sulle istruzioni negli shelving buffer in una fase detta di **dispatching**. Il blocco è ancora possibile, ma solo per mancanza di risorse (buffer overflow). Un'istruzione rimane nel buffer finché le dipendenze non vengono risolte. Al liberamento di un'unità funzionale, si controlla il contenuto dello shelving buffer e si cercano istruzioni eleggibili per l'esecuzione (delle quali una è inviata in esecuzione secondo la politica di dispatching scelta).

Un'istruzione è **eleggibile** quando sono disponibili i suoi operandi (dataflow). Possono esserci reservation station individuali (una per ogni unità funzionale, tipicamente con 2-4 posizioni nel buffer) oppure reservation station di gruppo (una per un gruppo di unità funzionali, 8-16 posizioni, possono inviare in dispatch più di una istruzione per ciclo). Infine può esserci una reservation station centrale.

Si può anche utilizzare un buffer combinato (reorder buffer) in grado di eseguire shelving, ridenominazione e riordino

Politiche di lettura operandi:

1. **issue bound:** si leggono durante l'issue delle istruzioni. Sugli shelving buffers c'è spazio per gli operandi sorgente. RF deve avere abbastanza porte per fornire operandi simultaneamente a più istruzioni.

2. dispatch bound: si leggono durante la fase di dispatching, sugli shelving buffers ho solo gli identificatori dei registri. Questi durante la fase di dispatch vengono inviati al register file che prende gli operandi richiesti e li trasferisce agli ingressi dell'unità funzionale che li richiede.

La lettura può avvenire **con o senza ridenominazione** dei registri. Nel primo caso occorre altro spazio di registri per memorizzare i valori ridenominati.

Dispatch: guida lo scheduling per l'esecuzione di istruzioni di una particolare reservation station, e dissemina le istruzioni alle UF allocate (dopo lo scheduling).

Politica di dispatch: con ridenominazione dei registri ed esecuzione speculativa, sono eseguibili le istruzioni i cui operandi sono disponibili. Se no, bisogna fare verifiche ulteriori sulle dipendenze. Regole di **arbitraggio** nel caso di più istruzioni eleggibili di quante possono essere disseminate: in genere si prioritizzano quelle più "vecchie". L'**ordine di dispatch** può essere di 3 tipi: **in ordine**, **parzialmente fuori ordine** e **furi ordine**. Ci sono però 2 problemi: verificare se gli operandi sono disponibili nel RF (durante la lettura nel RF stesso), e verificare se tutti gli operandi delle istruzioni presenti nella RS sono disponibili (durante il dispatching).

Scoreboard: introdotto per verificare la disponibilità degli operandi. E' un registro di stato costituito da un bit aggiuntivo per ogni elemento nel Register File.

1: dato disponibile

0: dato non disponibile

Lancio di un'istruzione: il bit del **registro destinazione** corrispondente viene azzerato. In questo modo indica alle istruzioni successive che usano quel dato che esso non è disponibile.

Risultato disponibile: si riporta a 1 il bit del registro. Le istruzioni che lo richiedono sanno che è disponibile.

a) verifica diretta dei bit di scoreboard: disponibilità operandi sorgente non è esplicitata in RS. Si usa quando gli operandi sono letti al dispatch. Ad ogni ciclo si verifica l'eleggibilità dell'istruzione più vecchia in ogni RS (check dei bit di validità degli operandi). In caso positivo, si passano all'unità funzionale opcode e #registro destinazione. Al RF vengono inviati i # dei registri sorgente. Si azzerava lo scoreboard del registro destinazione. Quando l'istruzione è completata, il risultato è inviato al RF assieme al #registro destinazione, che è aggiornato. Scoreboard viene riportato a 1.

b) Verifica dei bit espliciti di stato: la disponibilità degli operandi sorgente è indicata esplicitamente da bit di stato nella RS. Si usa quando gli operandi sono letti al momento dell'istruzione issue.

Al momento dell'issue, l'operando richiesto non è disponibile, allora RF restituisce identificatori invece di valori. La disponibilità del contenuto dei registri è gestita tramite scoreboarding. All'issue il bit del registro destinazione è posto a 0. L'unità funzionale genera il risultato e si aggiorna il bit del registro destinazione a 1: le

istruzioni che sono state inviate successivamente possono accedere. Lo stato delle reservation stato è aggiornato ad ogni ciclo di clock (tramite ricerca associativa).

Ridenominazione dei registri: si usa per eliminare dipendenze WAR e WAW (formato registro registro a 3 operandi è presupposto). Può essere statico o dinamico. Parziale (solo per alcune istruzioni) oppure completa. Si può compiere mediante buffer di ridenominazione: in questo caso si determina dove dovrebbero essere scritti o letti i risultati intermedi (quelli che non possono modificare il RF architetturale perchè le istruzioni relative non sono ancora giunte a commitment).

1) **Fusione di RF architetturale e di ridenominazione:** i registri arch e rid vengono allocati dinamicamente sullo stesso RF fisico. Allocazione gestita da una **mapping table**.

2) **Implementazione separata:** L'istruzione specifica Rd destinazione: si alloca quindi un nuovo registro di ridenominazione, valido finchè un'istruzione successiva si riferisce allo stesso Rd (si rialloca su un nuovo registro di ridenominazione, oppure istruzione giunge a commitment e l'allocazione perde validità)

3) **Utilizzo del Reorder Buffer.**

Algoritmo di Tomasulo: resosi necessario per gestire la ridenominazione anche nonostante la presenza dei salti (non gestibile staticamente). Si evitano le alee RAW garantendo che ogni istruzione sia eseguita solo quando gli operandi sono disponibili.

Si riorganizza l'unità di controllo: quando un istruzione con operandi non ancora pronti viene inviata in issue, i nomi dei corrispondenti registri sorgente sono sostituiti con quelli delle RS che forniranno il risultato (eliminando la necessità di ridenominazione).

I risultati delle operazioni sono scritti dalla UF alla RS senza passare dal register file, grazie ad una struttura particolare di comunicazione (**Common Data Bus**).

Load Buffer e Store Buffers contengono dati e indirizzi scambiati con la memoria.

Tutti i risultati da memoria e da UF sono inviati al CDB; tutte le RS, i RF e i L/S Buffer hanno un'etichetta (tag) grazie alla quale si fornisce la ridenominazione dei registri usando un'insieme esteso di registri virtuali.

Struttura delle Reservation Station:

- **Tag** (identifica RS)
- **Opcode** (identifica l'istruzione)
- **Vj, Vk** (valori degli operandi sorgente)
- **Qj, Qk** (puntatori alle RS che producono Vj e Vk)
- **Busy** (indica che la RS è occupata)
- **Load Buffer** (campo indirizzo e campo busy)

- **Store Buffer** (campo indirizzo)
- **Register File:** ha un campo Qj (tag RS) che indica la RS contenente l'operazione che scriverà il risultato in quel registro. Se Qj è vuoto significa che il valore attualmente presente è quello utilizzabile perchè nessuna RS sta utilizzando quel registro.

L'algoritmo di Tomasulo è suddiviso in 3 stadi.

1) **Issue:** l'istruzione I è prelevata dalla coda di istruzioni (in ordine FIFO). Si cerca una RS vuota adatta e là si invia l'istruzione assieme al valore degli operandi (se disponibili nei registri). Se non c'è RS vuota, avviene un blocco. Se non ho gli operandi nei registri, si tiene conto delle unità funzionali che li producono (in questo passo si opera una ridenominazione, eliminando alee WAR e WAW).

WAR: I scrive in Rx (sorgente di K successiva a I), K ha già il valore di Rx oppure sa quale istruzione scrive tale valore. Si collega RF ad I.

WAW: issue in ordine, permette di collegare RF ad I.

2) **Esecuzione:** quando gli operandi sono disponibili, si esegue I, altrimenti si controlla il Common Data Bus (CDB) in attesa dei risultati sorgente delle operazioni già in esecuzione. Si evitano in questo modo le alee RAW. Più istruzioni possono diventare pronte nello stesso ciclo per la stessa UF, bisogna fare una scelta critica per l'unità L/S.

Nel primo passo si calcola l'indirizzo reale e lo si scrive nel buffer L/S; Le load vengono eseguite non appena si rende disponibile l'unità di memoria. Le Store aspettano il valore che si deve memorizzare prima di poterlo inviare all'unità di memoria. (Si mantengono le operazioni in ordine di programma tramite il calcolo dell'indirizzo reale)

Eccezioni: nessuna istruzione può iniziare l'esecuzione finchè i salti condizionati precedenti non sono stati risolti. Con branch prediction bisogna risolvere la speculazione.

3) **Scrittura risultati:** quando un risultato è disponibile, lo si scrive sul CDB e da qui viene propagato alle unità che lo attendevano. La store scrive anche in memoria. L/S utilizzano un'unità funzionale per il calcolo dell'indirizzo effettivo. Tutte le scritture si compiono nello stadio Write Results.

Vantaggi algoritmo di Tomasulo:

- logica per identificazione di alee è distribuita. Non ho conflitto per accedere al Register File
- Si eliminano WAR e WAW con la ridenominazione implicita dei registri e alla memorizzazione degli operandi nelle RS non appena sono disponibili.
- Si possono eseguire Load e Store in ordine diverso di programma,

purchè non agiscano sugli stessi indirizzi di memoria, mentre le coppie di Load sono liberamente riordinabili. Occorre però che siano già stati calcolati gli indirizzi in memoria di qualunque precedente operazione sulla memoria.

Svantaggi algoritmo di Tomasulo:

- **Complessità hardware** non banale (1 buffer associativo veloce e logica complessa per ogni RS).
- **CDB limita prestazioni:** risorsa scarsa e logica di verifica dei tag va replicata in ogni RS, per ogni CDB.

Possibili miglioramenti:

1. Scheduling dinamico
2. Ridenominazione dei registri
3. Disambiguazione dinamica della memoria
4. Predizione dinamica dei salti

Esecuzione parallela

Istruzioni eseguite in parallelo in genere sono finite fuori ordine

1. **finished:** operazione completata eccetto il WriteBack
2. **completed:** risultato scritto nel RF destinazione
3. **committed / retired:** completamento se architettura include ROB

Bisogna rispettare la consistenza sequenziale dell'esecuzione del programma (ordine di completamento, ordine di accessi a memoria).

Consistenza **debole (weak):** si può completare fuori ordine purchè non si sacrificano dipendenze di dati (processore). Accessi a memoria anche fuori ordine senza violare dipendenze di dati (**memoria**)

Consistenza **forte (strong):** forzamento dell'ordine di programma usando ROB (processore), accessi a memoria in ordine di programma (memoria)

Modello più usato **Strong Program Weak Memory.**

L/S: attendono che sia stato calcolato l'indirizzo di memoria (Load completa quando dato è scritto nel registro in RF). Store attende che il dato sia disponibile.

Weak memory: L/S bypassing, L/S speculative, mascheramento cache miss.

L/S bypassing: load può scavalcare store non completata e viceversa, purchè non si violino dipendenze dati in memoria (quelle scavalcate devono avere indirizzi differenti).

Load speculative: si inizia l'accesso a memoria prima di aver verificato gli indirizzi (a differenza delle non-speculative dove si attende la verifica prima di iniziare il bypass). Occorre validare le load speculative e fare il loro undo in caso di speculazione errata (rieseguire la load annullando tutte le istruzioni successive a quella load).

Reorder Buffer (ROB): **buffer circolare con puntatori testa** (indica

posizione libera) e **coda** (istruzione a commit). Si scrivono istruzioni nel ROB in ordine di programma, all'issue dell'istruzione si alloca un'elemento nel ROB (elemento completato con bit aggiuntivi informativi che indicano anche lo stato dell'istruzione stessa: i(issued),x(executed),f(finished) più ulteriori informazioni come ad esempio l'eventuale stato speculativo dell'istruzione associata a quell'elemento).

Un'istruzione giunge al commit sse è finita e le istr precedenti sono già giunte al commit. Solo le istr al commit possono essere **completate**. ROB supporta esecuzione speculativa e gestione precisa delle eccezioni.

Eccezioni: accettate solo quando l'istruzione è prossima a commit, interruzioni esterne associate alla coda del ROB (per convenzione). Si gestisce un'eccezione svuotando il ROB dal contenuto successivo all'istruzione che ha generato l'eccezione.

Gestione dei conflitti di controllo: la penalizzazione dovuta ai salti può avere sia una componente dovuta ai branch incondizionati che a quelli condizionati (dominanti). Bisogna predire quanto prima il risultato di un salto condizionato nello stadio ID. Prestazioni di predizione legate ad **accuratezza** e al **costo di una predizione errata**.

Esistono tecniche hardware che consentono **predizione dinamica dei salti** migliorando i fattori di prestazione sopra citati.

Branch prediction buffer: detto anche **branch history table**. Memoria indirizzata tramite i bit meno significativi dell'indirizzo di una branch completato da un bit che indica se il salto condizionato è stato fatto recentemente oppure no.

Posso avere speculazione errata (salto **diverso** con stessi bit meno significativi di quello considerato). Nel caso di cicli posso avere due predizioni sicuramente errate (in ingresso al ciclo e in uscita).

Modifico il branch predictor trasformandolo in una macchina a stati: la predizione deve risultare errata 2 volte prima di essere invertita (in generale, si usano contatori a n-bit, ma gli schemi a 2 bit statisticamente sono sufficienti).

Esistono anche schemi di predittori a più livelli (**predittori correlanti**), ovvero che basano la predizione sul comportamento di altri salti.

Per migliorare ulteriormente occorre superare le barriere dei salti per fornire un flusso a banda larga di istruzioni: questo si ottiene predicendo l'indirizzo di destinazione del salto.

Branch Target Buffer: al termine dello stadio IF serve conoscere il contenuto prossimo del PC prima ancora di aver decodificato l'istruzione attuale. Se poi l'istruzione è un salto, posso saltare con penalità zero! Vi si accede nello stadio IF, usando come indice l'indirizzo dell'istruzione appena letta.

Se l'accesso è un hit, allora l'istruzione letta è un salto, e il prossimo indirizzo predetto è noto al termine dello stadio IF dell'istruzione letta (che era un salto possibile), ovvero un ciclo prima di quando si usa il semplice branch prediction buffer.

BTB è una cache totalmente associativa, l'indirizzo predetto viene inviato alla cache istruzioni prima della decodifica stessa dell'istruzione! Devo sapere se l'istruzione predetta è un salto. In ogni caso in parallelo viene svolta la computazione canonica di calcolo del prossimo Program Counter, per cui se poi l'opCode decodificato non identifica un salto posso utilizzare il valore usuale come prossimo PC e ignorare la predizione.

Una volta effettuata la predizione, la CPU procede in modo speculativo sul risultato dei salti condizionati, eseguendo il programma come se la speculazione fosse corretta. La **speculazione hardware** estende lo scheduling dinamico (che si limita ad anticipare la decodifica ma non l'esecuzione).

Speculazione Hardware: predizione dinamica dei salti, speculazione, scheduling dinamico.

Modalità **data-flow:** si eseguono le istruzioni non appena gli operandi sono disponibili. Nell'algoritmo di Tomasulo, per supportare la speculazione, la fase di commit è separata dalla fase di esecuzione utilizzando il ROB.

I risultati vengono scritti solo al giungimento del **commit** (dove si sa se la speculazione era valida o meno). Si esegue fuori ordine, si completa in ordine.

Struttura ROB: suddivisa in 4 campi per ogni elemento

1. **Tipo istruzione:** salto, store, load, ALU
2. **Destinazione:** #registro (load e alu), memory addr (store)
3. **valore:** (contiene il risultato fino al giungimento del commit)
4. **ready:** l'esecuzione è completata, il valore è pronto.

ROB sostituisce store buffer e funzione di ridenominazione delle RS

RS registrano operazioni e operandi tra l'istante di issue e quello d'esecuzione. I risultati sono etichettati con il # dell'elemento nel ROB. Tutte le istruzioni (tra cui salti errati e load errate rispetto alla speculazione), giungono al commit quando raggiungono la testa del ROB.

In questo punto viene segnalata la speculazione errata, si svuota il ROB e si riprende dal successore corretto del salto. Risulta quindi facile annullare le azioni speculative.

1. **Issue/dispatch:** estraggo istruzione dalla coda delle istruzioni. Se RS libera e ROB libero, lancio. Se operandi disponibili in RF o ROB, si inviano alla RS assieme alla posizione del ROB per il risultato (etichetta sul CDB). Se RS / ROB pieni, stallo.
2. **Execute:** se operandi non ancora pronti, monitoraggio del CDB in attesa del valore. Si verifica presenza di alee RAW. Quando operandi sono disponibili nella RS si esegue l'operazione. Per la store basta la disponibilità del registro base.
3. **Write result:** Risultato scritto sul CDB, propagato a RS che lo aspettano e al ROB. RS diviene disponibile. Se è una store, se il risultato è disponibile viene scritto nel campo Valore sul ROB, altrimenti si monitora CDB.
4. **Commit:** tre diversi comportamenti possibili
 - **Commit normale:** istruzione raggiunge la testa del ROB, risultato è presente in buffer. Si scrive su RF. Si libera il ROB
 - **Commit di una store:** come sopra, però si scrive in memoria
 - **Speculazione:** se la speculazione era errata, si svuota il ROB, si riprende esecuzione dal successore corretto dell'istruzione di salto errata. Se la speculazione era corretta, si porta a compimento il salto e si validano le istruzioni successive (annullando il campo speculazione).

CPU può facilmente fare undo perchè non scrivo nulla fino al commit.

CPU fa recovery velocemente da speculazione errata.

Eccezioni: esse non vengono riconosciute fino al commit. Se un'istruzione speculata genera eccezione:

- Si registra eccezione in ROB, e si guarda se l'istruzione associata era in stato speculativo o meno

- Se la predizione era errata, significa che l'eccezione era inesistente e non viene servita (perchè l'istruzione che l'ha generata non andava nemmeno eseguita!)
- Se raggiunge la testa del ROB, non è più speculativa e devo servire l'eccezione.

WAW e WAR attraverso la memoria sono eliminate tramite la speculazione.

RAW attraverso la memoria inducono 2 restrizioni:

- Load non può passare in lettura se un'elemento attivo nel ROB corrispondente ad una store ha destinazione = sorgente load
- Si mantiene l'ordine di programma per calcolare indirizzi della load rispetto a tutte le store precedenti.

Il ROB può essere sostituito da un insieme fisico più grande di registri associato alla ridenominazione dei registri.

Limiti dell'ILP e CPU reali

Modello hw per le valutazioni astratte è un “processore ideale”:

- **Ridenominazione dei registri:** # infinito di registri (no dipendenze dati salvo quelle vere)
- **Predizione salti:** perfetta (no dipendenze di controllo nel programma dinamico)
- **Analisi degli indirizzi in memoria:** perfetta (no dipendenze dati salvo quelle vere)
- **Finestre illimitate di analisi:** per l'issue (in realtà finestre nemmeno troppo ampie fanno esplodere il numero dei confronti necessari)
- **No limitazioni alle istruzioni possibili nello stesso ciclo:** (porte di lettura/scrittura infinite, bus infiniti, numero di unità funzionali infinito)
- **Latenza pari a 1** per le caches e le unità funzionali: supposizione molto ottimistica che non tiene conto delle restrizioni tecnologiche dell'hardware!

Architettura P6

CPU con scheduling dinamico, traduce istruzioni IA-32 in una serie di micro operazioni (uops - RISC) eseguite dalla pipeline. Max istruzioni per ciclo : 3 IA-32 tradotte in uops. Architettura superscalare a 3 vie.

Unità di lettura/decodifica (in ordine), **Unità di dispatch/execute** (fuori ordine), **Unità di retire** (riordinamento, committment e risoluzione speculazioni)

Le uops sono eseguite fuori ordine su una pipeline speculativa con ridenominazione dei registri e ROB. Pipeline a 14 stadi.

Pentium 4

Architettura "NetBurst": frequenza clock più alta, mantiene throughput di esecuzione prossimo al massimo. Pipeline da 20 stadi. 3 meccanismi di pre-fetching:

- Hardware (basato su BTB)
- Software su D-cache
- Prefetch HW da L3 a L2

Risorse e prestazioni generali superiori all'architettura P6. Introduce la **trace cache: trova una sequenza dinamica di istruzioni** (con salti) **da caricare in un blocco di caching** (estensione della località spaziale canonica). Inoltre si possono riordinare le Load rispetto ad altre Load e Store, possibili Load speculative e ammessi fino a 4 cache miss sulle Load.

Architetture VLIW

CPU con **scheduling statico**. VLIW: Very Long Instruction Word. Si identificano tramite compilatore le istruzioni che possono essere eseguite in parallelo (assemblate in **bundle** che vengono poi eseguiti dalla cpu). Questo perchè nei processori superscalari il 30-35% dello spazio CPU è occupato dal controllo. Si affida al compilatore l'ottimizzazione e si riduce la complessità della CPU, liberando area sul chip (da utilizzare per altre risorse, ad esempio aumentando la dimensione della cache on chip).

Compilatore: vede finestra programma molto più grande da analizzare, NON conosce eventi dinamici (salti fatti o no) ma può contare semplicemente su analisi statistiche derivate dall'esecuzione runtime di programmi benchmark.

Istruzione lunga (**bundle**) da 128 bit, eseguite sequenzialmente. Si impacchettano più operazioni corte (sillabe) in un singolo bundle (in genere 4 sillabe da 32 bit). Ogni sillaba ha associato un percorso (**lane**) nel datapath della CPU. Tutte le sillabe possono accedere contemporaneamente al RF.

Spetta al **compilatore identificare 4 operazioni indipendenti tra loro** (da impacchettare in un bundle per l'esecuzione parallela), e sempre il

compilatore risolve i anche i conflitti relativi alle risorse.

- Al più **una** istruzione di controllo per bundle.

L'esecuzione è puramente sequenziale, unità di controllo semplice quanto una CPU scalare. Non si può ammettere che sillabe diverse di uno stesso bundle giungano al Write Back in istanti diversi. Se il parallelismo non è elevato, le istruzioni contengono un numero elevato di sillabe **nop**.

Architettura VLIW: è più semplice di una superscalare a parità di parallelismo, più veloce ma meno adattabile a programmi di tipo general purpose. In genere i processori VLIW sono adatti per applicazioni specifiche facilmente ottimizzabili per quanto riguarda il parallelismo.

Compilatore: deve conoscere architettura e parametri tecnologici, ciò riduce la portabilità del codice oggetto. Non c'è compatibilità binaria perchè ogni cambiamento tecnologico nello hardware induce anche la modifica del codice oggetto.

Ho problemi in caso di **data cache miss**, il compilatore deve creare il codice oggetto sulla base dei ritardi di caso pessimo (1 miss per ogni load!)

Esecuzione speculativa: essenzialmente speculazione statica, con varianti per ricalibrazione dinamica.

Esecuzione Predicata (o condizionale)

Insieme di tecniche che convertono le dipendenze di controllo in dipendenze di dati. Alternativa ai salti condizionati. E'una forma di esecuzione condizionale di istruzioni basata su una condizione booleana.

- **Condizione vera:** eseguo l'istruzione
- **Condizione falsa:** l'istruzione equivale ad una nop

move condizionale: si porta un valore da un registro ad un altro sse la condizione è vera (CMOVZ R2,R3,R1 : spostato R2 in R3 sse R1 è vero). In questo modo ho trasformato la dipendenza di controllo in una dipendenza di dato.

Una CPU con issue multipla può gestire più facilmente più istruzioni simultanee senza salti (anche con dipendenze di dati), rispetto alla gestione di salti condizionati (e alla relativa complessità indotta dall'esecuzione speculativa sui salti)

L'esecuzione completamente predicata (codice risultante senza diramazioni, ho un unico blocco basico di percorsi eseguiti in parallelo) elimina le penalità per predizioni errate, a costo di un "uso inutile" di risorse del sistema (ogni volta che si adotta l'esecuzione predicata, si inserisce nello schedule una computazione utile e una inutile, perchè le due alternative generate sono mutualmente esclusive). Il bilancio è positivo se ho abbondanza di risorse hardware e richieste stringenti in prestazioni.

Oppure si utilizza **l'esecuzione parzialmente predicata**, cioè l'esecuzione incondizionata di tutte le singole operazioni su ambedue i percorsi, poi si selezionano i valori finali sulla base di una

condizione derivata dalla istruzione di salto)

L'esecuzione predicata migliora il codice se i salti sono difficili da predire, crea blocchi basici più grandi (maggiormente ottimizzabili). Si può combinare con la speculazione per avere risultati migliori. Tuttavia aumenta in modo rilevante la dimensione del codice (non va bene per sistemi embedded).

Itanium

Cpu 64bit, EPIC (Explicit Parallel Instruction Computing). Compilatore globalmente ottimizzante, si aggiunge supporto hw per ottimizzazioni dinamiche (massimizzazione del throughput. Unità di controllo più complessa di un VLIW base

- **supporto a esecuzione dinamica**
- **ALAT (Advanced Load Address Table)**
- **Controllo di speculazione / predicazione**
- **supporto a register renaming/handling**

Bundle elementare (3 operazioni + template)

Parallelismo pari a 6 (2 bundle sono 1 istruzione lunga completa)

No ROB, RS. Hardware semplificato. Register Renaming sostituito da Register Remapping (più semplice). Le dipendenze sono identificate dal compilatore.

** **philips trimedia**

** **processori DSP**

** **tendenze futuro**

I sistemi multiprocessore

Le architetture ILP supportano parallelismo a livello di istruzione, più di tanto a questo livello non posso fare, per cui si passa al parallelismo di processi / thread. Si collegano più processori in un sistema complesso.

E'ovvio che il software deve esprimere un naturale parallelismo (es. Applicazioni server, dbms, dedicate), estratto dal programmatore o indotto tramite il compilatore che identifica i tasks eseguibili in parallelo.

- **SISD: Single instruction stream, single data stream** processore singolo convenzionali
- **SIMD: single instruction stream, multiple data stream** architetture vettoriali, applicazioni specifiche
- **MISD: Multiple instruction stream, single data stream** nessun processore commerciale di questo tipo

- **MIMD: Multiple instruction streams, multiple data streams** ogni processore legge le proprie istruzioni ed opera sui propri dati.

MIMD è quello più usato. Le macchine MIMD sono flessibili (macchine a utente singolo ad alte prestazioni, multiprocessori multiprogrammati, soluzioni intermedie ...). Si possono costruire tramite CPU standard, da collegare tramite una rete (per cui è efficiente rispetto ai costi)

Per sfruttare n processori, il programmatore / compilatore identifica n thread/processi da eseguire indipendentemente. Parallelismo identificato dal software non dall'hardware come nelle CPU superscalari.

Obiettivo: aumento del throughput delle applicazioni multithreaded

Classi di arch MIMD

Architettura centralizzata a memoria condivisa: più processori condividono una memoria centralizzata, anche organizzata a banchi (collegati da uno o più bus, o da una rete di commutazione)

La memoria primaria permette un tempo d'accesso uniforme per qualunque processore (sistemi SMP - multiprocessori simmetrici- / UMA -accesso a memoria unificato-). Esistono anche sistemi DSM (memoria condivisa distribuita) con diverse memorie fisiche viste come un'unica memoria logica condivisa.

Vantaggi:

1. no partizionamento codice/dati (li posso mettere in memoria condivisa e farne lo sharing tra i processori);
2. no spostamento dati per processi inter-comunicanti (per lo stesso motivo di cui sopra)

Svantaggi:

1. sincronizzazione di accesso alle strutture dati --> introduco **reti di interconnessione a throughput elevato** in grado di servire tutte le richieste senza introdurre latenze troppo elevate.
2. manca di scalabilità (contention aumenta con n, i moduli condivisi mettono in attesa n-1 processori quando sono utilizzati --> introduco cache locali (ma nascono problemi di coerenza dati!)

Architettura distribuita (multicomputers): più coppie processore/memoria (nodo) sono interconnesse. Memoria è fisicamente distribuita (eventualmente anche le periferiche). Soluzione scalabile. L'interazione tra nodi avviene mediante **scambio di messaggi** (banda della rete di interconnessione è fondamentale).

Il sistema può essere costituito anche da più computer totalmente separati, collegati tramite rete locale (**clusters**). Ogni nodo può essere un piccolo SMP (2 - 8 processori)

I processi che richiedono un dato remoto, mandano un messaggio e si pongono in attesa switchando processo (richiesto context-switching e tanti processi) oppure procedono finchè non gli serve la risposta - busy wait (servono meccanismi hw/sw di polling)

Vantaggi:

1. Contention non è un problema grave come nei sistemi a memoria condivisa
2. Sincronizzazione semplice (strutture dati separate)
3. Riduce latenza per accessi a memoria

Svantaggi:

1. Comunicazione tra processi è complessa (aumenta la latenza)
2. Processi comunicanti su nodi diversi devono scambiare dati tramite messaggi (non ho strutture condivise per lo scambio dei dati, devo basarmi sul message passing)
3. **Load Balancing:** problema critico per le prestazioni
4. **Deadlock:** introdotto dal meccanismo di scambio dei messaggi
5. La copia fisica di strutture dati tra processi è onerosa sia in termini spaziali che temporali

Reti di interconnessione

Possono essere **statiche** (punto a punto) o **dinamiche** (commutazione intelligente).

Nei sistemi a memoria distribuita: scambio di messaggi completi (protocolli sono fondamentali). In quelli a memoria condivisa vanno supportati accessi a memoria brevi e frequenti (evitando contention e hot spots, ovvero punti sovraccaricati di richieste)

Architetture MIMD simmetriche a memoria condivisa: ogni processore ha la stessa relazione con un'unica memoria condivisa (dati condivisi e dati privati)

Nei sistemi a memoria condivisa la presenza di caches permette di ridurre le latenze di accesso a memoria ma introduce il problema di cache coherence, soprattutto quando vengono portati in cache dei dati condivisi da più processi.

Sistemi NUMA: in genere sono sistemi interconnessi di nodi con processore e memoria locale. La rete di interconnessione serve anche per il collegamento con la memoria condivisa (Global Shared Memory - GSM). Questi sistemi hanno problemi di cache coherence, per cui vengono introdotti i sistemi **COMA** (COherent Memory Access), dove ogni blocco di memoria si comporta come una cache e migra autonomamente verso i processori che lo richiedono.

CC-NUMA: soluzione compromesso con nodi interconnessi (processore, memoria, cache). Inizializzazione della memoria statica allo startup, secondo il modello NUMA, e poi allocazione dinamica con load balancing (COMA).

L'introduzione delle cache in genera crea problemi di coerenza per le operazioni di I/O. In un sistema multiprocessore la vista della memoria, per ogni processore, è data dalla propria cache per cui dati due processori posso avere due differenti viste della memoria.

Coerenza del sistema di memoria: il sistema deve osservare le seguenti proprietà: (P: un generico processore, X: posizione di memoria)

1. Lettura di X da parte di P, che segue scrittura di X da parte di P, senza scritture intermedie su X da parte di un altro processore, restituisce sempre il valore scritto da P (**ordine di programma**)
2. Una lettura di X da parte di P, che segue una scrittura di X da parte di un altro P, restituisce il valore scritto e ho una sufficiente separazione temporale tra le operazioni e non ho ulteriori scritture intermedie tra le operazioni (**vista coerente**)
3. Le scritture nella stessa posizione sono **serializzate**. L'ordine di scrittura visto per una posizione X è lo stesso per tutti i processori (**serializzazione delle scritture**: garantisce che quella visibile sarà sempre l'ultima scrittura per ogni processore)

Bisogna definire anche quando un processore in lettura vedrà un valore scritto di un altro processore (**consistenza di memoria**). Consistenza e coerenza sono proprietà complementari.

In un multiprocessore è normale che un programma abbia più copie degli stessi dati in diverse caches. Le caches forniscono sia la **migrazione** che la **replicazione** degli elementi di dati condivisi.

- **Migrazione:** trasparenza degli spostamenti verso le caches locali (riduce la latenza per l'accesso a dati condivisi remoti e la richiesta di banda per la memoria condivisa)
- **Replicazione:** dati condivisi letti simultaneamente (riduce latenza accessi e contention per i dati condivisi in lettura)

Si introducono **protocolli di cache coherence** (mantengono traccia dello stato di ogni condivisione sui blocchi dati). Necessità dovuta a 3 problemi:

1. **Condivisione di dati scrivibili**
2. **Migrazione di processi**
3. **Attività di I/O**

Strutture dati a sola lettura non creano problemi, strutture scrivibili private li creano solo in caso di migrazione di processo, invece strutture scrivibili condivise generano parecchi problemi.

Protocolli HW per la coherence:

- **Directory based:** lo stato di condivisione è mantenuto in una sola posizione (directory). Protocollo centralizzato.
- **Snoopy based:** ogni cache che ha copia di un dato mantiene copia anche del suo stato di condivisione. Le cache sono collegate su un bus di memoria condivisa: tutti i loro controllori fanno un monitoraggio (snoop) del bus per determinare se hanno una copia del blocco che viene richiesto sul bus.

Politiche:

1. **Write invalidate:** garantisce l'accesso esclusivo a un dato prima della scrittura da parte di un processore. Quando si scrive non esistono più altre copie leggibili/scrivibili di quel dato. Protocollo forza la serializzazione degli accessi, chi arriva dopo

deve ottenere una copia dallo scrivente “vincitore” per poter a sua volta scrivere sul dato (derivata da write-back).

2. **Write broadcast:** si aggiornano tutte le copie quando si aggiorna un dato, utile per ridurre le richieste di banda (derivata da write trough).

La politica “migliore” è la write invalidate, perchè genera meno traffico sulla rete di intercomunicazione (risorsa fondamentale in un sistema multiprocessore). Per cui è quella più utilizzata.