Aumentare il parallelismo a livello di istruzione (1)

Architetture Avanzate dei Calcolatori

Valeria Cardellini

Parallelismo

- Il parallelismo consente di migliorare le prestazioni grazie all'**esecuzione simultanea** di più istruzioni
- Vari tipi di parallelismo
  - Parallelismo ***funzionale***
    - Deriva dalla logica di soluzione di un problema
  - Parallelismo ***dei dati***
    - Deriva dalle strutture dati che consentono l'esecuzione simultanea su più elementi nella soluzione di un problema
    - Problemi di calcolo scientifico, elaborazione di immagini
- … che possono anche essere sfruttati contemporaneamente

Parallelismo (2)

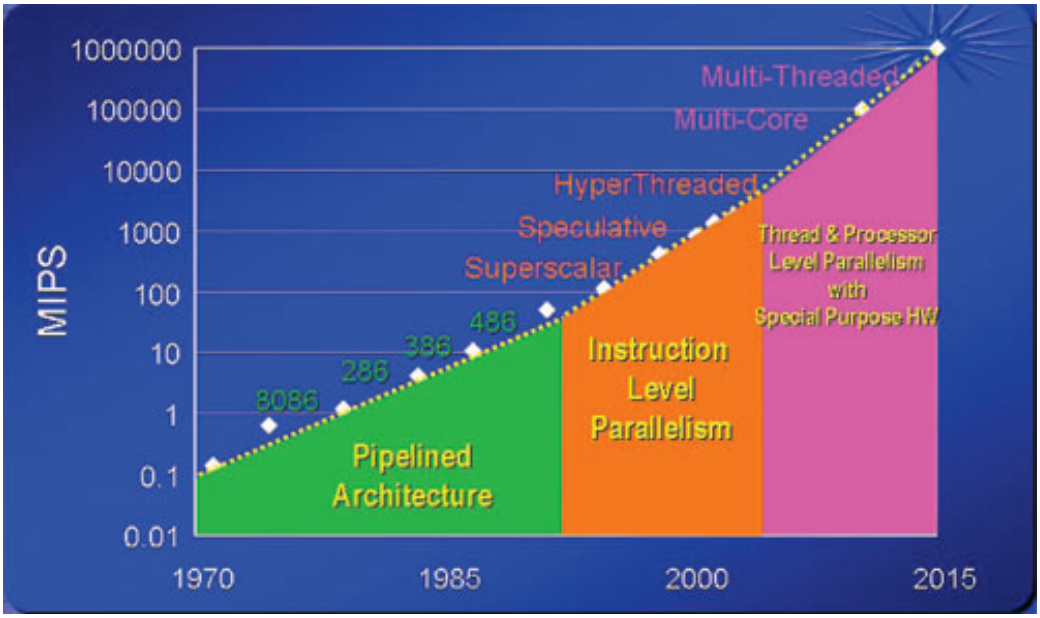
- Il parallelismo funzionale può essere visto a *diversi livelli di astrazione*, corrispondenti a *diverse soluzioni architetturali*:
  - Instruction-level (ILP): a grana fine
  - Loop-level: a grana intermedia
    - Parallelismo tra le iterazioni di un ciclo

      for (i=1; i<=1000; i=i+1)

        x[i] = x[i] + y[i]
  - Procedure-level: a grana intermedia
  - Program-level: a grana grossa

- Nelle lezioni successive analizzeremo il parallelismo a livello di istruzione

Parallelismo (3)



- Nel grafico: MIPS = milioni di istruzioni al secondo

Parallelismo a livello di istruzione (ILP)

- Anche detto *parallelismo intrinseco*
- Più istruzioni (a livello macchina) dello stesso programma vengono eseguite contemporaneamente
- Due tecniche fondamentali per l'esecuzione parallela di istruzioni
  - Pipelining
  - Replicazione di unità funzionali
    - Più unità eseguono le stesse operazioni in parallelo su dati diversi
- … che possono coesistere nella stessa architettura di un calcolatore
  - Processori ***multiple-issue***, in grado di avviare ed eseguire più istruzioni in parallelo

Verso prestazioni migliori

- In un processore con pipeline

  $CPI_{pipeline}$ = CPI ideale + cicli stallo pipeline = CPI ideale + stalli strutturali + stalli per dati + stalli per controllo
- Riducendo un qualsiasi termine a destra dell'uguaglianza, si riduce $CPI_{pipeline}$
- Tecniche per migliorare $CPI_{pipeline}$
  - **Dinamiche**
    - Implementate a livello hardware
  - **Statiche**
    - Implementate a livello software (dal compilatore)

Tecniche per migliorare le prestazioni

Tecnica	Riduce la componente del CPI dovuta a:	
Propagazione	Stalli dovuti ai dati	√
Salto ritardato e predizione statica dei salti	Stalli dovuti al controllo	√
Predizione dinamica dei salti	Stalli dovuti al controllo	√
Scheduling dinamico	Stalli dovuti ai dati	
Speculazione hardware	Stalli dovuti a dati e controllo	
Lancio di più istruzioni per ciclo (multiple issue)	CPI ideale	
Srotolamento del loop	Stalli dovuti al controllo	
Scheduling statico	Stalli dovuti ai dati	
Speculazione del compilatore	CPI ideale, stalli dovuti a dati e controllo	

√: già esaminata

Maggiore parallelismo in processori scalari

- Processore scalare: una sola pipeline
- Una volta ottimizzato il codice, il CPI non può essere ulteriormente ridotto
- Unica possibilità: ridurre il tempo di ciclo
  - Per una data tecnologia, occorre aumentare il numero di stadi della pipeline
    - Ogni stadio compie operazioni più semplici ed ha quindi una latenza inferiore
  - Pipeline più profonda: ***superpipelining***
- In linea di principio: data la latenza dell'istruzione, il tempo di clock è inversamente proporzionale al numero di stadi della pipeline

Superpipelining

- In realtà: l'aumento della profondità della pipeline è limitato da quattro fattori:
  - Presenza dei registri interstadio
  - Problemi di progetto elettronico diventano rilevanti (es. clock skew)
  - Ritardi legati alle criticità sui dati diventano maggiori in termini di numeri di cicli
    - Pipeline più profonda: maggior numero di stalli → aumenta il degrado delle prestazioni dovuto alle criticità sui dati
  - Criticità sul controllo implicano salti condizionati più lenti
    - Una predizione errata determina il flushing di un maggior numero di stadi
- Riassumendo: riduzione della durata del ciclo di clock ma maggior numero di cicli necessari per l'esecuzione di un dato programma
- Superpipelining: una delle tendenze nei processori attuali (10-12 stadi come profondità base), *insieme ad altre forme di ILP*

Scheduling

- Per migliorare le prestazioni si deve aumentare il grado di parallelismo
- Occorre
  - *identificare e risolvere le dipendenze*
  - ordinare le istruzioni del codice oggetto (farne lo *scheduling* o schedulazione)
- … in modo da raggiungere il massimo grado di parallelismo compatibile con le risorse disponibili
- Due approcci fondamentali per lo scheduling:
  - ***Scheduling statico***
  - ***Scheduling dinamico***

Scheduling statico

- Rilevamento e risoluzione *statica* delle dipendenze
  - Compito svolto dal *compilatore*
    - Uso di algoritmi sofisticati per attuare ILP
  - Le dipendenze vengono evitate mediante riordinamento del codice
    - Il compilatore produce un codice oggetto privo di dipendenze che creano criticità
- Architettura tipica
  - Processori **VLIW** (Very Long Instruction Word)
    - Il processore si aspetta di ricevere un codice esente da dipendenze, dato che non è predisposto per rilevarle e risolvere
    - Lanciate ed eseguite più istruzioni in parallelo per ciclo di clock; il numero di istruzioni è *fissato* dal compilatore

Scheduling dinamico

- Rilevamento e risoluzione *dinamica* delle dipendenze
- L'hardware *riordina* l'ordine di esecuzione delle istruzioni *a tempo di esecuzione* in modo da ridurre gli stalli
  - Occorre mantenere immutato il flusso dei dati ed il comportamento a fronte di eccezioni
  - Vantaggi: gestione di dipendenze non note a tempo di compilazione (ad es. riferimenti a memoria), semplificazione del compilatore, maggiore portabilità del codice
  - Svantaggio: maggiore complessità dell'hardware di controllo
- Architettura tipica
  - Processori **superscalari**
    - Lanciate ed eseguite più istruzioni in parallelo per ciclo di clock; il numero di istruzioni è *variabile*

Scheduling dinamico (2)

- Nella semplice pipeline del MIPS, le istruzioni vengono caricate e lanciate in ordine di programma (*in-order*)
  - Di conseguenza, lo stallo di una istruzione non permette alle istruzioni successive di procedere finché non viene risolta la dipendenza
- Tuttavia, l'esecuzione di un'istruzione può iniziare non appena gli operandi sono disponibili
  - Si ha quindi l'esecuzione fuori ordine (*out-of-order*)
  - L'esecuzione fuori ordine comporta il completamento fuori ordine
  - L'esecuzione fuori ordine è possibile anche con architetture scalari (non solo superscalari)

Scheduling dinamico (3)

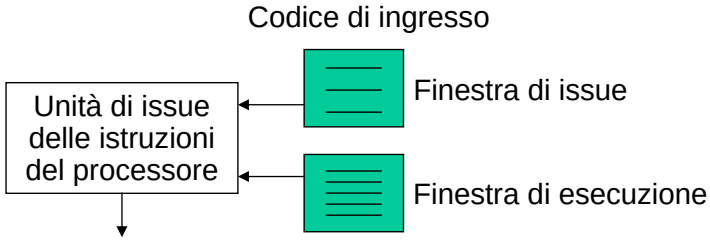
- Esempio:

```
lw $t0, 0($s2)
add $t1, $t0, $t2
sub $s1, $s1, $t3
```

- Senza scheduling dinamico, l'istruzione sub viene ritardata (a causa dello stallo necessario a gestire la criticità di tipo load/use) sebbene non abbia dipendenze sui dati
- Con lo scheduling dinamico, l'esecuzione dell'istruzione sub può iniziare prima, senza seguire l'ordine di programma
- Problema: l'esecuzione fuori ordine introduce la possibilità di criticità sui dati di tipo WAR e WAW e complica la gestione delle eccezioni (in quanto può generare eccezioni imprecise)

Rilevamento e risoluzione dinamica delle dipendenze

- Finestra di *issue* (o *emissione*): istruzioni caricate e decodificate ma non ancora lanciate in esecuzione
- In ogni ciclo, le istruzioni nella finestra di issue vengono controllate riguardo alle dipendenze
  - rispetto a tutte le istruzioni nella finestra di esecuzione
  - rispetto a tutte le istruzioni nella finestra di issue stessa
- Dopo il controllo nessuna, una o più istruzioni vengono lanciate (*issued*) alle unità di esecuzione
  - Processori scalari: si lancia una sola istruzione
  - Processori superscalari: si lanciano più istruzioni

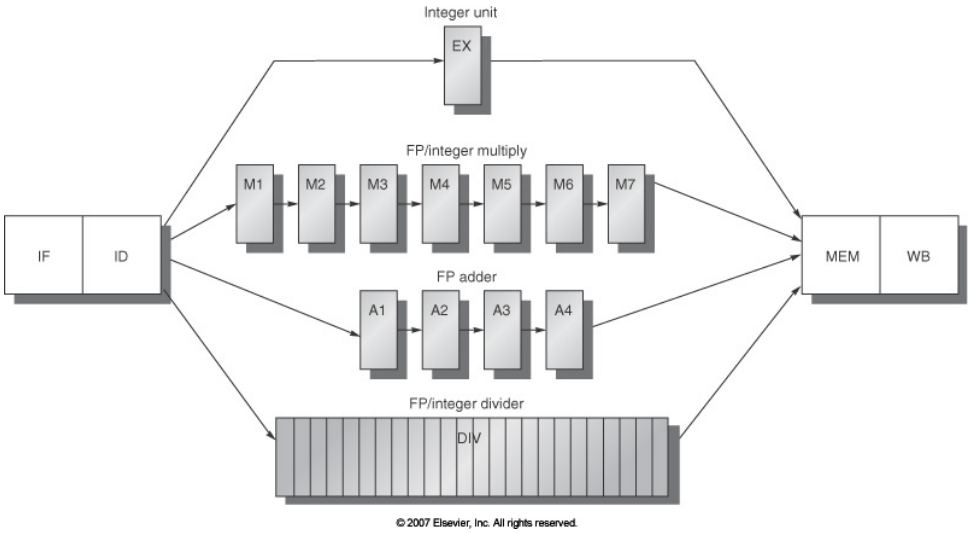


Scheduling dinamico su processore scalare

- Processore scalare semplice (una sola pipeline)
  - Le criticità sui dati che non possono essere risolte mediante propagazione creano stalli nella pipeline
  - Non si caricano né lanciano nuove istruzioni
- Una sola pipeline multifunzionale non è l'approccio più efficiente
- *Prima possibilità*: introdurre un certo numero di *pipeline specializzate* (dedicate)
  - Si mantiene il caricamento e la decodifica di una sola istruzione per volta (*single issue*); dopo lo stadio ID, le istruzioni sono eventualmente instradate verso pipeline funzionali diverse, dedicate
  - Esempio: pipeline dedicata per istruzioni in virgola mobile nel MIPS (richiedono più di un ciclo di clock!)
- *Seconda possibilità*: usare *pipeline multiple*
  - Non è esclusiva rispetto alla prima
  - Capacità di caricare, lanciare e decodificare più istruzioni per volta (*multiple issue*)

Pipeline dedicate

- Esempio
  - Una pipeline per operazioni intere e load/store
  - Una pipeline per operazioni di moltiplicazione intere e in virgola mobile
  - Una pipeline per addizioni in virgola mobile



Pipeline dedicate (2)

- La presenza di più pipeline implica la possibilità di *esecuzione fuori ordine*
  - Anche se si lancia un'istruzione per volta, le diverse pipeline hanno latenze differenti
  - Il completamento delle istruzioni può avvenire fuori ordine, ovvero in ordine diverso rispetto all'ordine sequenziale di programma
- Occorre adottare tecniche per mantenere la *consistenza sequenziale*
  - Risultati identici a quelli ottenuti con un'architettura sequenziale
  - Esempio (no assembler MIPS!): con due pipeline, una per interi con latenza di esecuzione pari a 2 cicli, una per virgola mobile con latenza di esecuzione pari a 6 cicli (solo la prima accede alla memoria)

```
mulf $2, $3, $4
lw $2, 100($5)
```

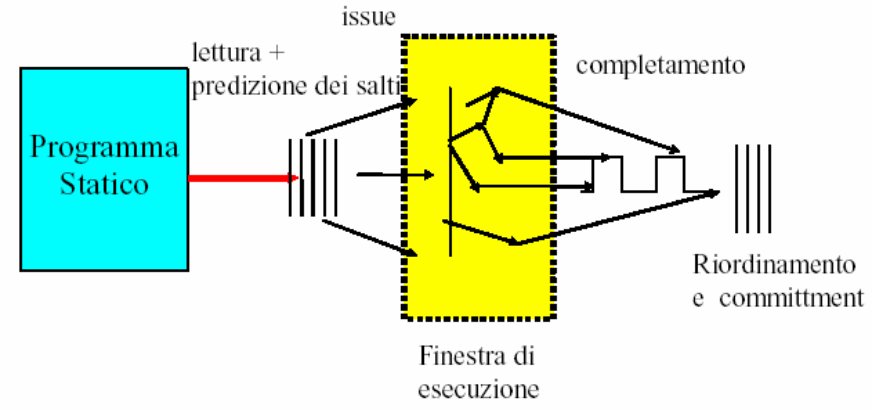
    - Completamento fuori ordine: risultato sbagliato in $2

Pipeline dedicate: mantenere la consistenza sequenziale

- *Prima soluzione* (più semplice): le istruzioni in virgola mobile non possono scrivere nel banco dei registri
  - Tecnica usata in processori vecchi, in cui la pipeline FP era vista come un coprocessore
- *Seconda soluzione*: sincronizzazione delle due pipeline in modo forzato per forzare le istruzioni a essere completate seguendo l'ordine sequenziale
  - Se necessario, la pipeline più corta (a latenza minore) viene rallentata introducendo delle bolle
  - Tecnica usata nei primi Pentium
- *Terza soluzione*: *riordinamento* (reordering)
  - Le pipeline funzionali non effettuano il write-back direttamente in memoria o nel banco dei registri: è presente un meccanismo per cui i risultati vengono scritti nell'*ordine di programma*, anche se sono stati ottenuti fuori ordine
  - Soluzione analizzata più avanti

Processori superscalari

- Esecuzione parallela con più unità di esecuzione
  - Concetto introdotto nel 1970
- Richiede
  - Più pipeline operanti in parallelo
  - Capacità di caricare, lanciare e decodificare più di un'istruzione per volta



Processori superscalari (2)

- E' possibile leggere e decodificare *simultaneamente* più istruzioni
- Esistono *più pipeline di esecuzione* distinte a cui inviare le istruzioni
  - Limite teorico: avendo *n* pipeline di esecuzione, se sono state caricate e decodificate *n* istruzioni prive di criticità e compatibili con le pipeline libere, tutte le *n* istruzioni vengono eseguite contemporaneamente
- Esistono meccanismi che garantiscono un aggiornamento dello stato in modo da supportare la *consistenza sequenziale*
- Come conseguenza:
  - **CPI<1**: in teoria, se *n* istruzioni possono essere caricate, decodificate ed eseguite simultaneamente, si ottiene CPI=1/*n*
  - La banda della memoria (numero di istruzioni trasferite da memoria a processore in una sola operazione di lettura) deve essere superiore ad una parola

Elaborazione superscalare

- Dopo aver caricato più istruzioni (*fetch parallelo*) avvengono le attività di decodifica ed emissione
- *Decodifica parallela*
  - Molto più complessa di quella scalare
- *Emissione (issue) parallela* delle istruzioni
  - E' l'attività più critica; le istruzioni vengono analizzate per identificare possibili dipendenze e lanciate alle diverse pipeline
  - Una frequenza di emissione (*issue rate*) più alta amplifica i problemi dovuti a dipendenze su dati e controllo
  - Occorre introdurre *politiche di issue* sofisticate per ottenere prestazioni elevate

Elaborazione superscalare (2)

- *Esecuzione parallela* delle istruzioni
  - L'esecuzione di un'istruzione ha inizio sulla base della *disponibilità dei dati* invece che sull'ordine originale del programma
  - Disponibilità di unità funzionali multiple
  - Conseguenza: esecuzione fuori ordine
    - Istruzioni terminate in un ordine diverso da quello che si avrebbe se il programma fosse eseguito su un processore sequenzial
  - Problema: conservare la consistenza sequenziale dell'esecuzione delle istruzioni
- Completamento seguendo l'*ordine di programma* corretto
  - Lo stato della macchina viene aggiornato (le istruzioni giungono a *committment*) in ordine di programma (come erano nel programma iniziale)

Decodifica parallela

- Sia nel caso di processori scalari che superscalari con scheduling dinamico, la decodifica delle istruzioni è divisa in due fasi:
  - *Issue*: decodifica e verifica di criticità
  - *Read* (lettura degli operandi): si aspetta finché non si sono risolte le criticità sui dati, poi gli operandi vengono letti
- Nei processori con scheduling dinamico
  - Le istruzioni attraversano lo stadio di issue nell'ordine di programma (*in-order issue*)
  - Possono scavalcarsi ed iniziare l'esecuzione fuori ordine nello stadio di lettura degli operandi

Decodifica parallela (2)

- Processore scalare
  - Ad ogni ciclo di clock si carica e si avvia alla decodifica *una sola* istruzione, di cui si verificano le dipendenze rispetto alle istruzioni già in esecuzione
- Processore superscalare
  - In un solo ciclo il processore decodifica più istruzioni, verifica le dipendenze
    - entro l'attuale finestra di esecuzione
    - fra la finestra di issue e la finestra di esecuzione
  - Per effettuare la verifica delle dipendenze sono necessari più confronti in un solo ciclo
  - Per evitare di dilatare il tempo di ciclo a causa della complessità della decodifica superscalare, spesso si ricorre alla *predecodifica*, spostando parte del compito di decodifica nella fase di caricamento dell'istruzione dalla memoria

Meccanismi per scheduling dinamico

- Principali meccanismi per lo scheduling dinamico (in hardware)
  - Scoreboard
  - Algoritmo di Tomasulo
  - ReOrder Buffer (buffer di riordino)
- Analizzeremo brevemente le diverse soluzioni

Scoreboard

- Soluzione ideata nel 1963 per l'architettura CDC 6600
- Scopo: verificare la disponibilità degli operandi per consentire, se possibile, l'esecuzione fuori ordine
- Lo scoreboard (*tabellone*) è una struttura ***centralizzata*** che tiene traccia delle dipendenze e dello stato delle operazioni
- Idea base: lo scoreboard è un registro di stato costituito da elementi di 1 bit, ognuno dei quali costituisce un'estensione di 1 bit del corrispondente registro
  - Bit dello scoreboard pari a 1: il dato nel registro è disponibile
  - Bit dello scoreboard pari a 0: il dato nel registro non è ancora disponibile
  - Quando un'istruzione viene lanciata, il bit dello scoreboard del corrispondente registro di destinazione viene messo a 0
  - Quando l'istruzione viene eseguita ed il risultato diventa disponibile, il bit dello scoreboard viene posto a 1; il contenuto del registro diventa disponibile per tutte le istruzioni che lo richiedono

Scoreboard (2)

- Gestione delle criticità WAR con scoreboard
  - Si blocca la scrittura del risultato fino a quando i registri non sono stati letti
  - Si leggono i registri solo durante lo stadio di lettura degli operandi
- Gestione delle criticità WAW con scoreboard
  - Si identifica la criticità e si blocca il lancio di nuove istruzioni finché le altre non sono state completate
- Invece che gli stadi ID, EX, WB ci sono gli stadi IS (issue), RD (lettura operandi), EX, WB

Scoreboard (3)

- Esempio di codice con criticità potenziali WAR e WAW

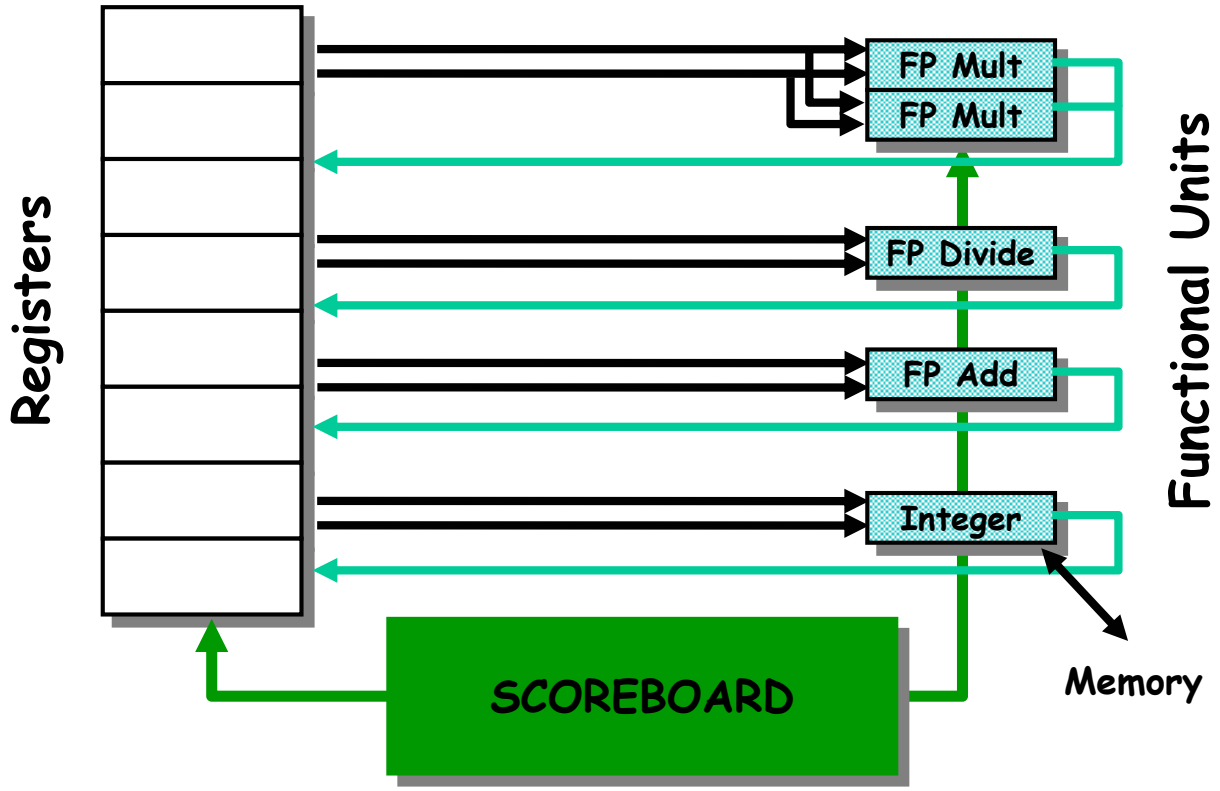
  div.d $f0, $f2, $f4
  add.d $f6, $f0, $f8
  sub.d $f8, $f10, $f14
  mul.d $f6, $f10, $f2

  – Se sub.d viene eseguita prima di add.d: criticità WAR
  – Se mul.d viene eseguita prima di add.d: criticità WAW

- Gestione delle criticità WAR con scoreboard
  – L'istruzione sub.d viene bloccata nello stadio WB, in attesa che l'istruzione add.d legga $f0 e $f8

- Gestione delle criticità WAW con scoreboard
  – L'istruzione mul.d viene bloccata nello stadio IS finché l'istruzione add.d scrive $f6

Architettura con scoreboard (CDC 6600)



Stadi di esecuzione con scoreboard

- Issue
  - Se l'unità funzionale per l'istruzione è libera e nessun'altra istruzione in esecuzione ha lo stesso registro destinazione (no criticità WAW), lo scoreboard lancia l'istruzione ed aggiorna la sua struttura interna; altrimenti stallo
- Lettura operandi
  - Lo scoreboard controlla la disponibilità degli operandi sorgente; quando sono disponibili avvia la lettura degli operandi e l'esecuzione dell'istruzione. Le criticità RAW sono gestite dinamicamente: le istruzioni possono essere eseguite fuori ordine
- Esecuzione
  - L'unità funzionale esegue l'operazione e quando termina avvisa lo scoreboard
- Scrittura del registro
  - Se non vi sono criticità WAR, allora viene scritto il risultato; altrimenti stallo

Contenuto dello scoreboard

- In realtà, lo scoreboard contiene le seguenti informazioni:
  - Stato dell'istruzione
    - In quale stadio si trova
  - Stato dell'unità funzionale
    - Se occupata, per quale operazione e con quali operandi, quali unità funzionali producono i registri sorgente e se questi sono disponibili
  - Stato della scrittura dei registri
    - Indica quale unità funzionale scriverà il registro di destinazione

Prestazioni dello scoreboard

- Scopo dello scoreboard è minimizzare (non eliminare!) il numero di stalli richiesti per gestire criticità sui dati
- Fattori limitanti
  - Grado di parallelismo delle istruzioni
  - Numero di entry dello scoreboard
  - Numero e tipo di unità funzionali
  - Presenza di potenziali criticità di tipo WAW e WAR
- Per incrementare il grado di ILP, occorre progettare un meccanismo più efficiente → Algoritmo di Tomasulo
  - Idea: ridenominiano i registri per eliminare le criticità WAW e WAR

```
div.d $f0, $f2, $f4                    div.d $f0, $f2, $f4
add.d $f6, $f0, $f8   Ridenominazione  add.d S, $f0, $f8
s.d $f6, 0($s1)          dei registri  s.d S, 0($s1)
sub.d $f8, $f10, $f14                  sub.d T, $f10, $f14
mul.d $f6, $f10, $f8   S e T: registri mul.d $f6, $f10, T
                          temporanei
```

  - Dopo la ridenominazione la criticità WAR e WAW sono sparite!

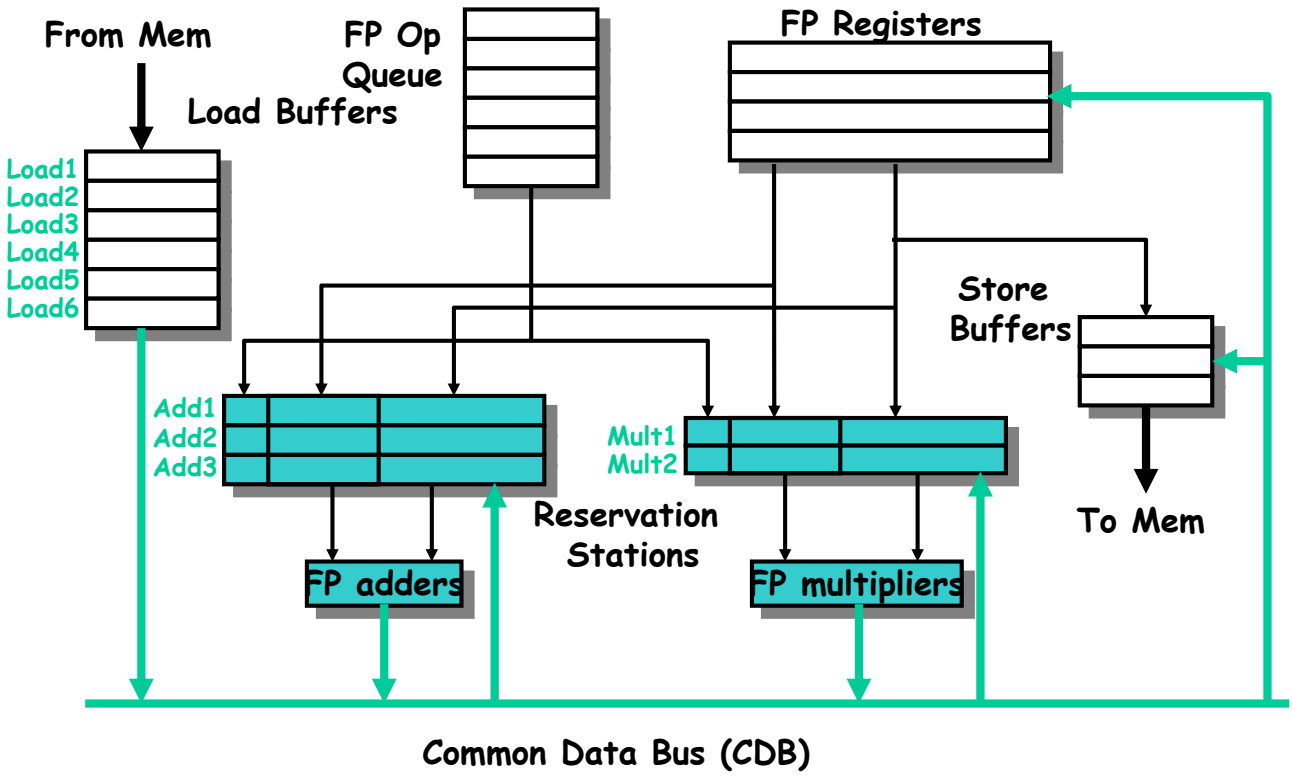
Algoritmo di Tomasulo

- Ideato nel 1966 per l'IBM 360/91 (prima delle cache!)
  - Elevata latenza della memoria
- Obiettivo: ottenere prestazioni elevate senza necessitare di compilatori speciali
- Perché studiare una soluzione ideata nel 1966?
  - E' usata, con diverse varianti, da Intel Pentium 4, AMD Athlon/Opteron, IBM Power 5, …

Algoritmo di Tomasulo (2)

- Il controllo è ***distribuito***: buffer e controllo sono *locali* alle unità funzionali
  - I buffer sono chiamati *reservation station* (RS o stazioni di prenotazione) e contengono gli operandi pendenti
- Le criticità di tipo WAR e WAW sono eliminate mediante la ***ridenominazione dei registri***
  - I registri nelle istruzioni sono rimpiazzati da puntatori alle RS
  - Le RS sono in *numero maggiore* dei registri: sono possibili ottimizzazioni che il compilatore non può fare
- I risultati da inviare alle unità funzionali vengono prelevati dalle RS e non dai registri; i risultati sono inviati a tutte le unità mediante un bus comune
- Le operazioni con la memoria sono trattate come quelle svolte dalle unità funzionali

Architettura con Tomasulo



Stadi di esecuzione con Tomasulo

- Issue
  - Prende l'istruzione dalla coda. Se la RS è libera (no criticità strutturali) viene mandata in esecuzione l'istruzione e inviati gli operandi (ridenominati i registri per eliminare criticità WAR e WAW)
- Esecuzione
  - Quando tutti gli operandi sono pronti, esegue l'operazione; altrimenti, controlla il bus comune dei risultati. Ritardando l'esecuzione finché non sono pronti tutti gli operandi sorgente, si evitano criticità RAW
- Scrittura del registro
  - Quando il risultato è disponibile, viene inviato a tutte le unità in attesa sul bus comune dei risultati
  - Bus di dati normale
    - Dati + destinazione
  - Bus comune dei risultati (CDB)
    - Dati + sorgente: si specifica l'unità funzionale sorgente

Prestazioni dell'algoritmo di Tomasulo

- Consente di gestire lo srotolamento dei cicli (*loop unrolling*) in hardware
  - Si eseguono contemporaneamente più iterazioni del ciclo, utilizzando le reservation station per la ridenominazione dei registri
- Svantaggi
  - Complessità dell'hardware
  - Prestazioni limitate dal bus dei risultati comuni

Confronto tra Tomasulo e scoreboard

Algoritmo di Tomasulo	Scoreboard
Dimensione della finestra di issue: ~14 istruzioni	Dimensione della finestra di issue: ~5 istruzioni
No criticità issue o strutturali	No criticità issue o strutturali
Criticità WAR: evitate tramite ridenominazione dei registri	Stallo nello stadio di scrittura del registro
Criticità WAW: evitate tramite ridenominazione dei registri	Stallo nello stadio di issue
Broadcast del risultato dalle unità funzionali	Risultato scritto nel registro di destinazione
Controllo distribuito tramite stazioni di prenotazione	Controllo centralizzato tramite scoreboard
Gestione del loop unrolling in hardware	

Consistenza sequenziale

- La consistenza sequenziale dell'esecuzione si riferisce a:
  - Ordine in cui le istruzioni sono completate (***consistenza del processore***)
  - Ordine in cui si accede alla memoria per istruzioni di load e store (***consistenza della memoria***)
- Consistenza del processore
  - *Debole*: le istruzioni possono essere completate fuori ordine, purché non si sacrifichi nessuna dipendenza dei dati
  - *Forte*: le istruzioni vengono forzate al completamento in stretto ordine di programma (uso del ReOrder Buffer)

Consistenza sequenziale (2)

- Esempio: sequenza di accessi alla memoria con possibile dipendenza dei dati

  sw $2, 0($5)

  lw $6, 0($3)

  – Le due istruzioni sono collegate tra loro? In altre parole, c'è una criticità sui dati di tipo RAW tra store e load riordinando le due istruzioni? Dipende dal contenuto di $5 e $3!
- Consistenza della memoria
  – *Debole*: gli accessi alla memoria possono essere fuori ordine purché non si violi nessuna dipendenza dei dati
    - Possibile il riordinamento load/store
  – *Forte*: gli accessi alla memoria avvengono in stretto ordine di programma
- I processori più recenti tendono a usare:
  – consistenza forte per il processore e debole per la memoria

Speculazione hardware

- Una volta predetto il prossimo indirizzo, si scommette (*specula*) sul risultato del salto condizionato: il programma viene eseguito come se la speculazione fosse corretta
  – Speculazione: le istruzioni sono eseguite assumendo corretta la speculazione
  – Scheduling dinamico semplice: ci si limita a caricare e decodificare le istruzioni, ma senza eseguirle finché l'esito del salto non è noto
- Si scommette anche sull'ordine di istruzioni di load/store
  – Si permette il riordinamento di load e store, ovvero il riordino degli accessi in memoria, nonostante le verifiche sugli indirizzi non siano state ancora risolte

Speculazione hardware (2)

- Occorrono meccanismi per gestire le speculazioni errate
- La speculazione hardware combina tre idee:
  - Predizione dinamica dei salti
    - Per scegliere quale istruzione eseguire
  - Speculazione, per poter eseguire delle istruzioni prima di aver risolto le dipendenze
    - Con la possibilità di poter disfare gli effetti di sequenze speculate in modo errato
  - Scheduling dinamico
- Adottata da Intel Pentium IV, PowerPC, AMD Athlon, …

Speculazione ed eccezioni

- La speculazione complica la gestione delle eccezioni
- Può introdurre eccezioni che non sarebbero state presenti con un'esecuzione in stretto ordine di programma
  - Esempio:
    - In conseguenza della speculazione su un'istruzione di load, può accadere che, se la speculazione è errata, l'istruzione di load usa un indirizzo illegale

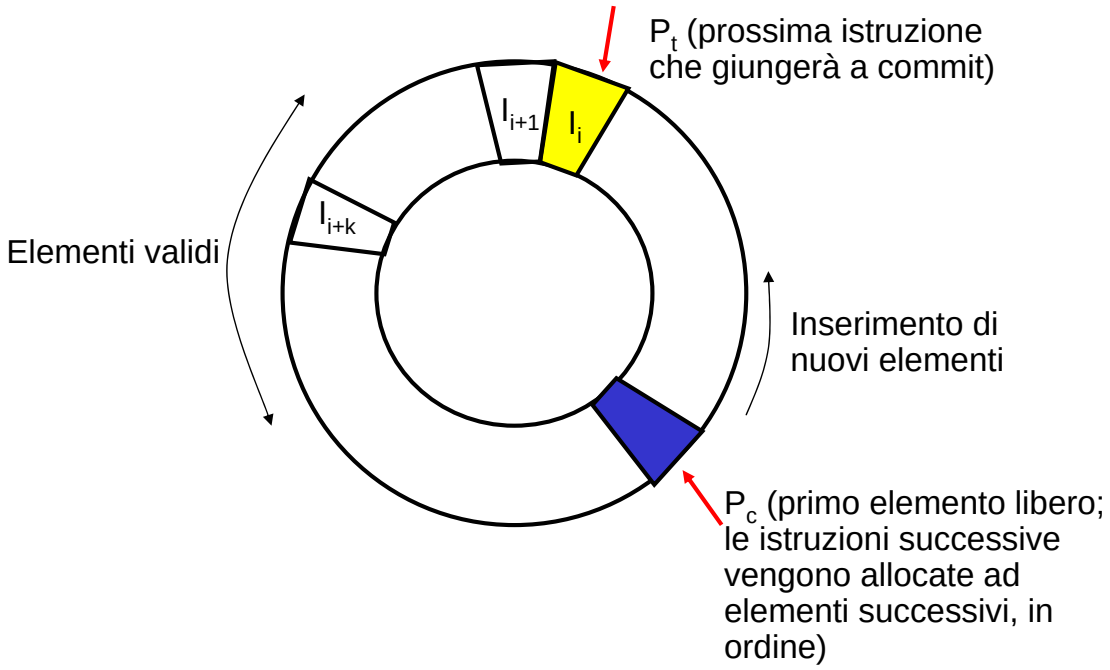
Esecuzione parallela

- Istruzioni eseguite in parallelo sono in genere terminate fuori ordine
  - Istruzioni più “giovani” possono terminare prima di istruzioni più “vecchie” aventi latenza superiore
- Istruzione *finita*
  - L’operazione richiesta è completata, con l’eccezione della scrittura (write-back) nel registro di destinazione o in memoria
- Istruzione *completata*
  - Il risultato è stato scritto nel registro di destinazione o in memoria
- Istruzione *committed* (retired)
  - Completamento quando l’architettura include il buffer di riordinamento (***ReOrder Buffer*** o ROB)

Il ReOrder Buffer

- E’ un buffer circolare con un puntatore in testa (P_t) ed un puntatore in coda (P_c)
  - Il puntatore di coda indica il prossimo elemento libero
  - Il puntatore di testa indica l’istruzione che per prima giungerà al commit (lasciando il ROB)
- Le istruzioni vengono scritte nel ROB in ordine stretto di programma
  - Quando si lancia l’istruzione, le viene allocato un elemento del ROB, che (tra l’altro) indica lo stato dell’istruzione (lanciato, in esecuzione, finito)
- Un’istruzione può giungere a commit solo se
  - E’ finita **e** tutte le precedenti istruzioni sono già giunte a commit
  - *Idea chiave*: eseguire fuori ordine, completare in ordine
- Solo le istruzioni che giungono a commit possono essere completate, ossia aggiornare il banco dei registri e la memoria

Il ReOrder Buffer (2)



Il ReOrder Buffer (3)

- Il ROB permette di supportare l'esecuzione speculativa
  - Ogni elemento del ROB ha un campo che l'indica se l'istruzione è stata eseguita in modo speculativo
- Le istruzioni finite non possono giungere a commit finché sono in stato speculativo
- In caso di speculazione errata (salto con predizione errata), il ROB viene svuotato e l'esecuzione riprende dal corretto successore del salto

- Estensione dell'algoritmo di Tomasulo con ROB
  - Nell'algoritmo base di Tomasulo, l'istruzione scrive il risultato nel banco dei registri, dove lo trovano le istruzioni successive
  - Con la speculazione, i risultati vengono scritti solo quando l'istruzione giunge a commit

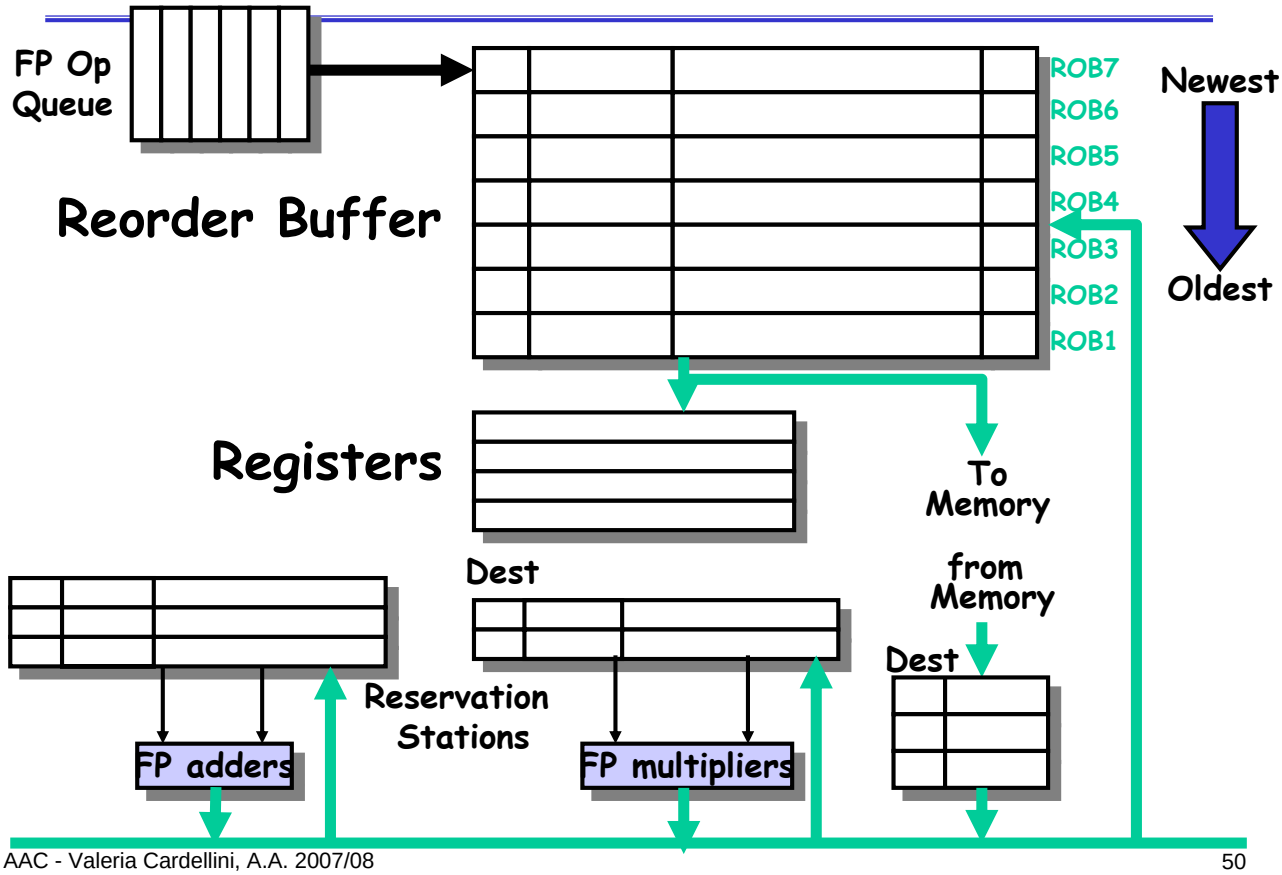
Estensione di Tomasulo con ROB

- Il ROB rimpiazza gli store buffer, la cui funzione è integrata nel ROB
- La funzione di ridenominazione svolta dalle RS è sostituita dal ROB
- Le RS sono ora usate solo per registrare le operazioni e gli operandi fra l'istante del lancio e quello in cui inizia l'esecuzione
- I risultati delle operazioni sono etichettati col numero dell'elemento nel ROB anziché con il numero della RS

Estensione di Tomasulo con ROB (2)

- Ogni elemento del ROB contiene quattro campi
  1: Tipo di istruzione
    Indica se l'istruzione è un salto condizionato (no destinazione del risultato), una store (destinazione in memoria), una load o istruzione ALU (destinazione in registro)
  2: Destinazione
    Numero del registro o indirizzo di memoria in cui scrivere il risultato
  3: Valore
    Valore del risultato finché l'istruzione non giunge a commit
  4: Ready
    Indica se l'istruzione è completata ed il valore è pronto

Architettura con Tomasulo e ROB



Stadi di esecuzione con Tomasulo e ROB

- Issue
  - Prende l'istruzione dalla coda. Se una RS è libera (no criticità strutturali) e c'è una posizione vuota nel ROB, l'istruzione viene lanciata. Se gli operandi sono disponibili (nel banco o nel ROB), vengono inviati alla RS, insieme alla posizione nel ROB allocata per il risultato
- Esecuzione
  - Quando tutti gli operandi sono disponibili nella RS, esegue l'operazione; altrimenti, controlla il bus comune dei risultati. Ritardando l'esecuzione finché non sono pronti tutti gli operandi sorgente, si evitano criticità RAW
- Scrittura del risultato
  - Quando il risultato è disponibile, viene scritto sul bus comune dei risultati (CDB), dal CDB al ROB e alle eventuali RS che lo aspettano. La RS viene resa disponibile

Stadi di esecuzione con Tomasulo e ROB (2)

- Commit
  - Tre diverse sequenze possibili

  1) Commit normale (istruzione ALU e load)
     - L'istruzione raggiunge la testa del ROB, il risultato è nel buffer. Si scrive il risultato nel registro, si cancella l'istruzione dal ROB
  2) Commit di store
     - L'istruzione raggiunge la testa del ROB, il dato è nel buffer. Si scrive il dato in memoria, si cancella l'istruzione dal ROB
  3) Commit di salto condizionato
     - Se la predizione è errata, si svuota il ROB e l'esecuzione riprende dall'istruzione corretta di destinazione del salto. Se la predizione è corretta, si porta a compimento l'istruzione di salto e si validano le istruzioni successive (annullando il campo speculazione)

Gestione delle eccezioni

- Le eccezioni non vengono riconosciute fino a quando l'istruzione non è giunta al commit
- Se un'istruzione speculata genera eccezione
  - Si registra l'eccezione nel ROB
  - In caso di predizione di salto errata (tale che l'istruzione che ha generato l'eccezione non avrebbe dovuto essere eseguita), l'eccezione viene cancellata con l'istruzione quando si svuota il ROB
  - L'istruzione raggiunge la testa del ROB, quindi non è più speculativa e l'eccezione viene servita

Architettura di base per processori con scheduling dinamico e speculazione