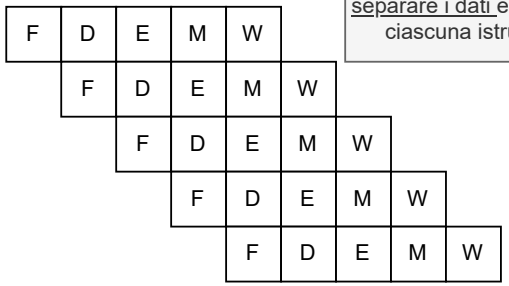
Pipeline ideale

A regime, una istruzione conclusa per ogni colpo di clock

Tra le diverse fasi sono presenti dei **registri di pipeline** che permettono di separare i dati elaborati da ciascuna istruzione.



Hazard

Situazioni che impediscono ad un'istruzione di essere eseguita nel momento 'ideale'. Tre tipologie:

- **Strutturali** -> Più istruzioni richiedono la stessa risorsa (es accesso a register file, ma c'è una sola porta);
- **Di dato** -> Un'istruzione dipende da dati elaborati da istruzioni precedenti (RAW);
- **Di controllo** -> Causati da branch e altre istruzioni che alterano il PC.

Stalli

Sono uno dei modi di gestire gli hazard: in generale, l'istruzione stallata e le successive attendono, le precedenti continuano l'esecuzione. Possono essere utilizzati per gestire **hazard strutturali e di dato.**

Forwarding

Invece che attendere che il risultato della ALU sia scritto nel Register File, esso viene passato **direttamente tra i registri di pipeline**.
Può essere usato per ovviare agli **stalli di dato**.
Il forwarding non può portare i dati indietro nel tempo!

Il forwarding prende i dati provenienti dagli output di

- **ALU**
- **memoria**

e destinati agli input di

- **ALU**
- **memoria**
- **unità di rilevamento dello zero**, per i salti

Quindi dagli stati EX/MEM e MEM/WB agli stati IF/ID, ID/EX ed EX/MEM

Control hazard

Dipendono dal fatto che il PC possa essere modificato da istruzioni di branch **dopo** il fetch di una o più istruzioni successive. Possibili soluzioni:

- **Stallare la pipeline** (soluzione più semplice).
- **Predict untaken**: supporre che il branch non venga mai preso. Se viene preso, tutte le azioni eseguite nel frattempo vengono annullate.
- **Predict taken**: assumere che il salto non venga mai preso.
- **Branch delay slot**.

Il MIPS scrive il PC alla fine della decode, se il salto è preso.

Branch delay slot

Siccome l'istruzione successiva al branch viene comunque eseguita, invece che annullarla **la si sfrutta per eseguire un'istruzione che dovrebbe comunque essere eseguita**, e che quindi non sarà annullata se il salto viene preso.
Il compilatore ha responsabilità di sfruttare il Branch Delay Slot opportunamente.

Operazioni multiciclo

Alcune unità, che effettuano **operazioni complesse** (es, quelle sui numeri floating point) sono **multiciclo** (la loro fase EX dura più cicli).
Se una istruzione I1 deve scrivere in un registro che deve essere ancora scritto da un'altra istruzione I2 ancora in EXE, **I2 stalla in decode ed entra in EXE solo quando ne è uscita I1**.
Latency -> Tempo tra istruzione che produce un dato e istruzione che lo usa
Initiation interval -> Numero di cicli tra l'issuing di due istruzioni che utilizzano la stessa unità (minimo 1).
Alcune unità (normalmente la divisione) NON sono nemmeno pipelined.
Hazard strutturali più frequenti -> Aggiungere porte costa, quindi normalmente **si sceglie di stallare l'istruzione**.

Nuovi hazard di dato

- **RAW - Read After Write**, o *Dipendenza* -> I2 ha bisogno del risultato di I1, che però non lo ha ancora calcolato;
- **WAW - Write After Write**, o *Output Dipendenza* -> I2 (più breve) scrive in R1 prima di I1 (più lunga), quindi I1 sovrascrive il dato di I2;
- **WAR - Write After Read**, o *Antidipendenza* -> I2 deve ancora leggere il dato scritto da I1, ed I2 è eseguita per prima.

Gestire gli hazard di dato

- **RAW**: il fowarding aiuta, dopo il forwarding l'unica soluzione sono gli stalli.
- **WAW e WAR**: si gestiscono mediante register renaming e la modifica dell'ordine delle istruzioni.

Instruction level parallelism

L'ILP è **sfruttato dalla pipeline** per eseguire più istruzioni in parallelo: più parallelismo esiste, maggiori saranno le prestazioni.

Per migliorare l'ILP si può modificare l'ordine delle istruzioni mediante un approccio **dinamico**, basato sul processore, oppure **statico**, basato sul compilatore.

Un **basic block** è una sequenza di istruzioni senza branch né in entrata né in uscita, eccetto quelli all'inizio e alla fine.

Rescheduling

All'interno di un basic block è possibile **implementare il rescheduling**, la modifica dell'ordine delle istruzioni in modo da **eliminare gli stalli**.

Tuttavia, nel MIPS i basic block sono normalmente tra 4 e 7 istruzioni: per migliorare il parallelismo si può dunque considerare il **loop unrolling**.

Dipendenze

Esistono tra istruzioni e possono creare degli hazard, ma non sono la stessa cosa. Tipologie

-> **Di dato**, su registri e su memoria (più difficile rilevare hazard, impossibile a compile time)
-> **Di nome** (antidipendenza e output dipendenza), non c'è una dipendenza tra i dati ma questi vengono comunque alterati.

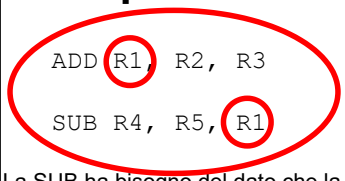
Loop unrolling

Se sono necessarie N iterazioni, il corpo del loop viene opportunamente replicato K volte, rendendo così necessarie N ÷ K iterazioni.

Pro
-> Meno overhead per il controllo dell'iterazione;
-> Corpo del loop più largo, quindi più possibilità di fare rescheduling;

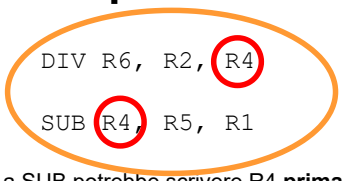
Contro
-> Dimensione del codice maggiore.

Esempio di RAW



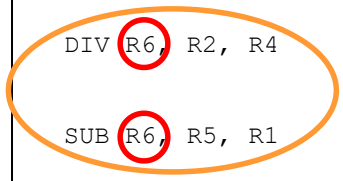
La SUB ha bisogno del dato che la ADD salva in R1, ma il dato non è ancora pronto quando la SUB viene eseguita.

Esempio di WAR



La SUB potrebbe scrivere R4 **prima** che la DIV lo legga durante l'EXE, ad esempio perché la DIV impiega più colpi di clock della SUB.

Esempio di WAW



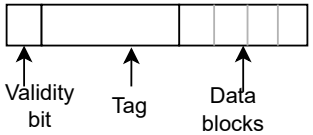
Se la SUB viene conclusa prima della DIV, R6 avrà un valore non consistente.

Dipendenze *di nome* e *di dato*

Notare come in un RAW la dipendenza sia di dato, ossia ciò che manca alla SUB (in questo caso) sia il risultato della precedente ADD, mentre WAR e WAW non sarebbero presenti se si utilizzassero registri diversi: si tratta infatti di dipendenze di nome.

Cache

La cache è una memoria **piccola** e ad **alta velocità**, posta tra il processore e la memoria principale. Sono normalmente a più livelli, ed organizzate in un **array**.



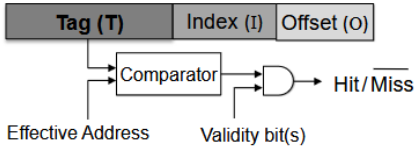
Organizzazione della cache

La cache è organizzata in **linee**, ogni linea contiene un **blocco di memoria** che include **alcune parole**.

Ogni linea è associata a un **tag**, indicante il blocco di memoria presente nella specifica linea di cache.

Hit e miss

- **Hit**: il blocco richiesto è presente in cache;
- **Miss**: il blocco richiesto NON è presente in cache, e viene caricato dalla memoria.



Performance

- In caso di **hit**: impatto sempre positivo;
- In caso di **miss**:
  - Se la cache carica l'intero blocco e poi la fornisce, impatto decisamente negativo;
  - Se la cache accede la memoria e fornisce la word immediatamente (*load-through*), impatto leggermente negativo.

Ubicazione della cache

La cache, di norma, è collocata tra la CPU e il bus. Questo consente di **ridurre la pressione sul bus**, e supporta **architetture multiprocessore**.

La cache può essere **unica**, oppure **separata (*Architettura Harvard*) per istruzioni e per i dati**. La cache delle istruzioni è più facilmente gestibile, in quanto viene solo letta.

Dimensione della cache

Una cache, quanto più è grande, tanto più:

- I costi salgono;
- Le performance migliorano;
- La cache diventa più lenta.

Le dimensioni, normalmente, variano tra alcuni Kb ad alcuni MB.

Mapping

Il meccanismo di *mapping* definisce, **dato in ingresso un blocco di memoria**, **quale linea della cache** debba essere scritta. Esistono diversi metodi:

- Direct mapping;
- Set associative;
- Full associative.

Direct mapped

Ogni blocco di memoria *i* è associato in maniera statica ad una linea *k* della cache, mediante l'equazione

k = i *mod* N

Ad esempio, in una cache grande 4 linee, se la memoria ha 16 blocchi una linea avrà associati 4 blocchi.

Meccanismo semplice da implementare, è sufficiente leggere i bit meno significativi dell'indirizzo di memoria per trovare il corrispondente in cache	Possono crearsi dei conflitti se due locazioni di memoria associate alla stessa linea sono accedute spesso

Full Associative

Ogni blocco della memoria può essere posizionato in un qualsiasi blocco della cache.

Massima flessibilità	L'hardware diventa più complesso, in quanto dovrà cercare un blocco libero su tutta la cache

Set Associative

Le linee della cache sono divise in S set, ognuno contenente W linee. Ogni blocco di memoria i è associato ad un set k, calcolato con la formula

k = i *mod* S

Il blocco i può essere piazzato in qualsiasi delle W linee del set k. Una cache con W linee in ogni set è chiamata cache a *W-vie*.

Bilanciamento tra flessibilità nella collocazione dei blocchi e ricerca di un blocco libero

Replacing algorithm

Il *replacing algorithm* definisce, per i casi *full associative* e *set associative*, quale linea della cache usare per scrivere i dati provenienti dalla memoria.

- LRU (Least Recently Used);
- FIFO (First in First Out);
- LFU (Least Frequantly Used);
- Random.

Write back e Write through

Quando la cache viene scritta dal processore, occorre decidere come aggiornare la memoria principale. Nel caso del **Write back** un bit detto *dirty bit* sarà settato, e al momento della sovrascrittura del blocco se tale bit è a 1 il blocco preesistente in cache verrà copiato in memoria. In questo caso la sostituzione sarà più lenta, ci potrebbero essere problemi di coerenza in sistemi multiprocessore e, in caso di guasti, potrebbe non essere possibile ripristinare correttamente la memoria.
L'altra opzione è il **Write through**: i dati scritti in cache sono anche trascritti nella memoria principale. Questo introduce, in teoria, una perdita di efficienza, in pratica però, siccome le letture sono molte più delle scritture, il problema dunque è in realtà più limitato.

Cache coherence

In sistemi multiprocessore, la memoria è comune a tutti i processori mentre la cache è, di norma, locale ad ogni singolo processore: questo provoca potenziali problemi di coerenza.
Per questo motivo esiste il **validity bit**: ogni volta che un certo blocco di memoria principale è scritto da un processore, eventuali linee di cache di altri processori vengono segnate come obsolete settando il validity bit a 0. Se a 0, tutti gli accessi alla cache producono un miss.

Livelli di cache

La cache può essere divisa in più livelli: la CPU cercherà il dato nel primo livello, quindi passerà al secondo e così via, copiando se necessario le informazioni nel livello più basso. I livelli successivi al primo possono essere condivisi tra più core.

Branch prediction

Predire i branch è fondamentale per evitare stalli dovuti a dipendenze di controllo. Le tecniche di predizione possono essere divise in:

- ***Tecniche statiche***: gestite dal compilatore, che si affida ad una analisi preliminare del codice;
- ***Tecniche dinamiche***: gestite dall'hardware, che si affida al comportamento in tempo reale del codice.

Tecniche statiche

Il compilatore può predire il comportamento dei branch in diversi modi:

- Predire sempre i branch come taken;
- Predire in base alla direzione del branch
  - In avanti, normalmente untaken;
  - All'indietro, normalmente taken.
- Predire in base alle informazioni provenienti da precedenti esecuzioni, identificando una serie di dati in ingresso, eseguendo il programma un certo numero di volte salvando le statistiche circa tali esecuzioni, che saranno poi utilizzate come statistiche.

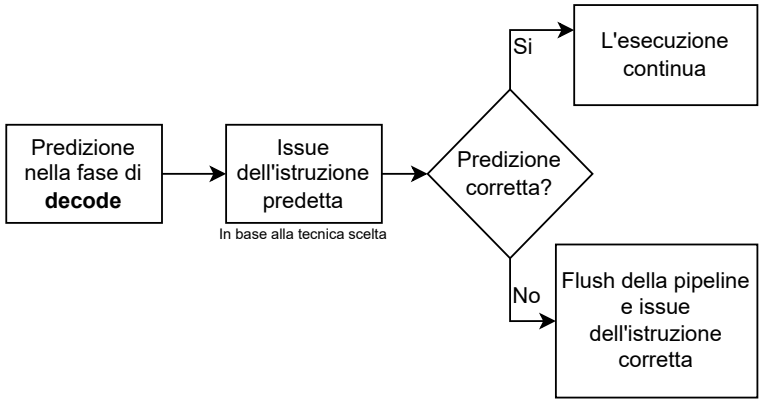
Tecniche dinamiche

Le tecniche dinamiche **si basano sull'hardware**, e possono essere di diversi tipi:

- Branch history table (BHT);
- Two-level prediction schemes;
- Branch-target buffer;
- ...

Esse si basano sul principio di **località**: come esiste quella temporale (una certa porzione di memoria acceduta a tempo T sarà probabilmente acceduta anche poco dopo T) e spaziale (una certa porzione di memoria acceduta alla posizione P, le posizioni intorno a P saranno probabilmente accedute poco dopo), la località *di salto* è il comportamento normalmente assunto ad esempio in caso di *for* e *while*, che potrebbero eseguire la stessa porzione di codice molte volte. Questo comportamento è dunque **prevedibile**, e se è prevedibile sarà anche **ottimizzabile**. La località non è normalmente spaziale, in quanto le seppur poche possibilità potrebbero essere collocate distanti le une dalle altre.

Tecniche dinamiche - Funzionamento



Branch History Table (BHT)

Si tratta della tecnica più semplice: una **piccola memoria**

- **indicizzata** dalla parte meno significativa dell'indirizzo delle istruzioni, significativa poi solo per quelle di branch;
- contenente, **per ogni record**, uno o più bit che tengono traccia se il branch sia stato o meno preso l'ultima volta che l'istruzione è stata eseguita.

L'**algoritmo** prevede:

- **accesso** alla BHT mediante la parte meno significativa dell'indirizzo dell'istruzione;
- **calcolo del PC** sulla base dell'esito della predizione;
- **aggiornamento** della BHT in base al risultato effettivo.

Più i record della BHT si riferiranno con precisione al branch di interesse e più la predizione sarà accurata, maggiore sarà la performance di questo metodo.

Schemi a n bit

In questo caso, invece di avere un singolo bit nella BHT, il record è costutuito da un contatore. Quando il contatore è a più di metà del valore massimo il salto è predetto come taken, altrimenti come untaken.

Ad ogni aggiornamento il contatore è incrementato o decrementato in base all'esito effettivo del branch.

Schemi a due livelli (m,n), esempio (2,2)

Questi schemi **utilizzano la cronologia degli ultimi m branch** per la predizione; ogni predittore ha un contatore a n bit che indica la probabilità che il branch sia preso o no.
Per ogni indirizzo di branch esiste una riga della tabella associata (più indirizzi per riga).

- **Un registro a 2 bit** (*shift register*) tiene traccia degli esiti degli ultimi 2 branch (in assoluto, lungo tutta l'esecuzione del codice): ad esempio, 10 indica che l'ultimo branch non è stato preso, il penultimo sì (shift verso sinistra);
- La tabella dei **predittori contiene 2^2 = 4 combinazioni di cronologia**, con un predittore a 2 bit per ogni combinazione: 00, 01, 10 e 11. Nei 2 bit del predittore sono conteggiati gli esiti degli ultimi branch;
- Supponiamo che la cronologia (shift register) iniziale sia 10: andiamo a cercare il predittore associato alla cronologia 10, e supponiamo che il **contatore associato** sia a 2 (10). In questo caso il predittore fa prendere il branch.
- Se il branch **viene preso effettivamente**, il predittore associato alla cronologia 10 verrà incrementato (diventando 3), e la cronologia passerà a 01; diversamente la cronologia passerà a 00, e il contatore associato alla cronologia 10 decrementato (diventando 1).

Questo approccio sfrutta le dipendenze tra branch consecutivi che si possono trovare in caso di iterazioni; tuttavia, aumentare m e n non crea significativi vantaggi, aumentando però il costo hardware.

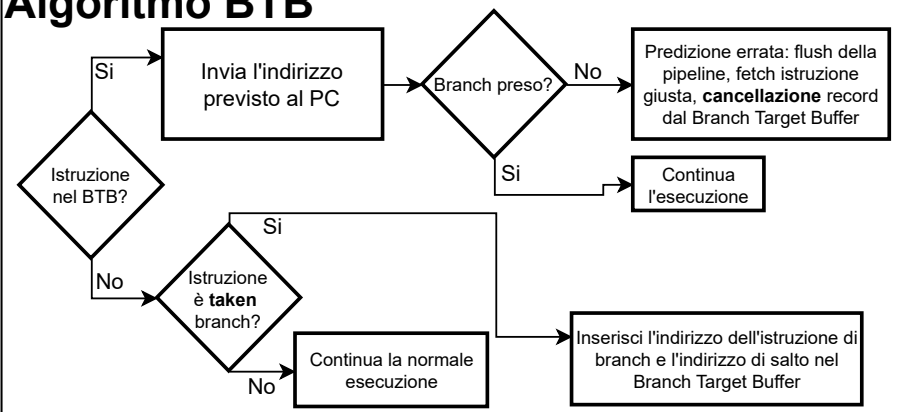
Branch target buffer

Oltre a conoscere se un branch sarà o meno preso, **è necessario, per ridurre gli effetti negativi delle dipendenze di controllo, conoscere anche il nuovo valore del PC se il branch sarà preso**: il *branch target buffer* esiste per questo motivo.

Si tratta di una tabella che immagazzina:

- Indirizzi delle istruzioni di branch;
- Target dell'istruzione da caricare nel PC.

Algoritmo BTB



Dynamic scheduling

Il **dynamic scheduling** permette di **riordinare le istruzioni in hardware**, riducendo gli stalli.

Il Dynamic Scheduling:

- Identifica dipendenze non conosciute a compile time, ad esempio gli indirizzi di memoria;
- Semplifica il lavoro del compilatore;
- Permette al processore di tollerare ritardi non predicibili;
- Permette di eseguire lo stesso codice su diversi processori pipelined.

Esempio

```
DIV.D F0, F2, F4

ADD.D F10, F0, F8

SUB.D F12, F8, F14
```

In una situazione come questa, la DIV.D causa lo stallo della ADD.D per la dipendenza di dato, ma **anche la SUB.D viene stallata**, anche se non ha nessuna dipendenza con le istruzioni precedenti e potrebbe essere senza problemi eseguita.

Miglioramento: Rimuovere il vincolo della *in-order execution*
Rischio di hazard WAR, WAW ed eccezioni imprecise.

Dynamic scheduling (II)

- Rende impossibile la gestione precisa delle eccezioni, in quanto, con I_e istruzione che ha sollevato l'eccezione
  - Una istruzione prima di I_e potrebbe dover essere ancora eseguita;
  - Una istruzione dopo I_e potrebbe essere già stata eseguita.
- Richiede di dividere la fase ID in due parti:
  - *issue*: decode dell'istruzione e ricerca di hazard strutturali, questa fase è eseguita in ordine;
  - *read operands*: attesa dell'assenza di hazard di dato, quindi lettura degli operandi: in questa fase le istruzioni possono sorpassarsi, entrando così nella fase di esecuzione *out-of-order*.

Esistono diversi approcci, tra cui lo **scoreboarding** e l'**algoritmo di Tomasulo**.

Architettura di Tomasulo

Introduce un'architettura nuova e più complessa, basata su:

- Tracciamento della disponibilità degli operandi mediante **Reservation station**. Esse
  - immagazzinano gli operandi in attesa che vengano utilizzati dalle istruzioni: essi vengono salvati nella RS non appena disponibili;
  - implementano l'*issue*;
  - identificano univocamente un'istruzione nella pipeline, in quanto le istruzioni in attesa di esecuzione designeranno la RS che fornirà loro l'operando in input.
- **Register renaming**, ogni volta che un dato entra in una RS per un'istruzione, il nome del registro di provenienza è mutato in quello della reservation station. **Questo permette di eliminare gli hazard WAW e WAR**.
- **Common Data Bus (CDB)**: i dati sono passati direttamente alle unità funzionali invece che passare dai registri, in modo che tutte le unità in attesa di un operando lo carichino insieme.

Issue

In questa fase le istruzioni sono **prelevate dalla coda in modalità FIFO**. Se **non ci sono RS** disponibili, l'**istruzione stalla**, altrimenti **se c'è una RS disponibile per l'operazione richiesta l'istruzione vi viene inviata** insieme agli operandi se disponibili, altrimenti viene registrata la FU responsabile per la loro generazione.

Execution

Istruzioni aritmetiche

Quando un operando appare sul CDB, esso è letto dalla RS, e non appena tutti gli operandi sono disponibili l'istruzione viene eseguita.

Istruzioni load/store

1. Non appena il registro contenente l'indirizzo di base è disponibile, l'effective address è calcolato e salvato nel *load/store buffer*;
2. In base al tipo di istruzione:
   1. Load: eseguita appena la memoria è disponibile;
   2. Store: attesa dell'operando da scrivere, quindi scrittura non appena la memoria è disponibile.

Nessuna istruzione può iniziare l'esecuzione prima che tutti i precedenti branch siano stati completati.

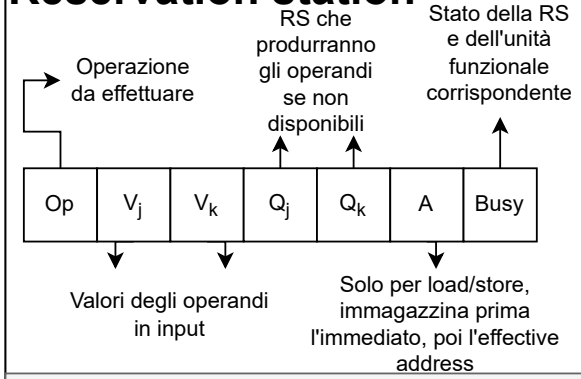
Write result

In questa fase

- i risultati sono, tramite il CDB
  - scritti nei registri;
  - portati alle FU che ne hanno bisogno.
- le istruzioni store effettuano la scrittura in memoria.

I risultati sono associati a degli identificatori interni alle RS, che funzionano come nomi di registro virtuale, implementando in questo modo il register renaming.

Reservation station



Campo Q_i del register file: contiene il numero della RS contenente l'istruzione il risultato della quale sarà salvato in quel registro. Se nullo, nessuna istruzione sta attualmente lavorando al risultato di quel registro.

Vantaggi e svantaggi - Load e store

Sono **eliminati gli hazard WAW e WAR**, e la logica di controllo degli altri hazard è distribuita. Tuttavia, questa architettura ha un'elevata complessità, e il Common Data Bus può rappresentare un collo di bottiglia.

Non è necessario effettuare il loop unrolling, in quanto le istruzioni sono naturalmente eseguite in parallelo.

Gli stalli **WAR e RAW** possono comunque avvenire cambiando l'ordine delle load e delle store. Per evitare hazard in questi casi:

- In caso di **load**, lo store buffer viene controllato per trovare store ancora da eseguire sullo stesso indirizzo. Nel caso, la load attende;
- In caso di **store**, sia lo store che il load buffer vengono controllati alla ricerca di istruzioni che agiscono sullo stesso indirizzo. Nel caso, la store attende.

Hardware based speculation

L'**hardware-based speculation** è una combinazione di tre idee:

- Dynamic branch prediction;
- Dynamic scheduling;
- Speculation.

Un processore che supporta il branch prediction e il dynamic scheduling effettua il fetch e l'issue delle istruzioni considerando la predizione come sempre corretta. Un processore che supporta la speculation effettua anche l'**esecuzione** delle istruzioni. Le operazioni sono quindi eseguite **non appena gli operandi sono disponibili**.

Architettura

L'architettura di base è quella di **Tomasulo**, estesa per supportare la speculazione con due step nell'esecuzione delle istruzioni:

- Calcolo dei risultati, anche con bypass di altre istruzioni;
- **Aggiornamento di register file e memoria, effettuato quando l'istruzione non è più speculativa. Tale fase è detta *commit* ed è effettuata in ordine.**

Il **ReOrder Buffer (ROB)** è la struttura preposta ad ospitare i risultati delle istruzioni fintantoché non hanno effettuato il commit. Ha registri virtuali aggiuntivi, ed integra lo store buffer dell'architettura base.
I risultati già calcolati nell'architettura base sono letti dal **register file**, mentre **con la speculation sono letti dal ROB se l'istruzione non ha ancora fatto commit**, altrimenti dal register file.

Struttura del ROB

Ogni record del ROB ha quattro campi:

- *Instruction type*: branch, store, register;
- *Destination*: numero del registro od indirizzo di memoria;
- *Value*: risultato dell'istruzione che non ha ancora effettuato il commit;
- *Ready*: indica se l'istruzione ha completato l'esecuzione.

Il ROB funziona in modo circolare.

Fase di Issue o Dispatch

- L'istruzione è estratta dalla coda se ci sono:
  - Una RS libera;
  - Uno slot libero nel ROB
- Altrimenti, l'istruzione è stallata;
- Gli operandi sono inviati alla RS se sono nel RF o nel ROB;
- Il numero dello slot ROB per l'istruzione è inviato alla RS per contrassegnare l'istruzione e i suoi risultati, quando saranno scritti sul CDB.

Fase di Execute

- Non appena gli operandi sono pronti l'istruzione è eseguita (evita hazard RAW);
  - Se non sono disponibili gli operandi, essi sono ricavati dal CDB non appena vengono prodotti.

La durata di questa fase dipende dalla tipologia dell'istruzione.

Fase di Write Results e Fase di Commit

- Non appena i risultati sono pronti, essi vengono inviati sul CDB **insieme al tag che li identifica**, e **vengono ricevuti dal ReOrder Buffer**;
- In questo modo, sono letti da ogni RS che li sta aspettando;
- Viene contrassegnata come *disponibile* la relativa entry della RS.

Una volta **nel ReOrder Buffer viene eseguita la fase di Commit:**

- Le istruzioni sono ordinate in base all'ordine originale;
- **Quando l'istruzione raggiunge la testa del buffer:**
  - Se il branch è stato predetto erroneamente, **è effettuato il flush del buffer e l'esecuzione riprende con l'istruzione corretta successiva**;
  - Altrimenti, **i risultati sono scritti nel register file o in memoria**
- **In ogni caso, il record del ROB è contrassegnato come libero**.

Hazard

- **WAW e WAR non possono avvenire, grazie al dynamic renaming**;
- **Gli hazard RAW mediante la memoria** sono prevenuti:
  - Mantenendo l'ordine del programma nel calcolo dell'effective address delle load;
  - Evitando che una load possa iniziare l'esecuzione se esiste nel ROB una store con campo destinazione corrispondente al campo origine della load.

Istruzioni di Store

La scrittura in memoria è effettuata **solo al commit**, per questo motivo l'operando serve in realtà al momento del commit e non nella fase di write results.

Per questo motivo, il ROB avrà un ulteriore campo che terrà tracca dell'origine del dato da scrivere, se l'istruzione è una store.

Eccezioni

Le eccezioni sono eseguite non al momento in cui accadono, ma **al momento del commit**.
Quando un'istruzione effettua il commit vengono, se del caso, anche eseguite le eccezioni, e viene effettuato il flush del buffer.
Se l'istruzione che doveva eseguire un'eccezione è rimossa dal buffer, l'eccezione viene ignorata.

Sono dunque supportare le eccezioni precise.

Istruzioni dispendiose

La scrittura in memoria è effettuata **solo al commit**, per questo motivo l'operando serve in realtà al momento del commit e non nella fase di write results.

Per questo motivo, il ROB avrà un ulteriore campo che terrà tracca dell'origine del dato da scrivere, se l'istruzione è una store.

Multiple Issue Processors

Un CPI minore di 1 (quindi più di una istruzione, in media, è completata in un colpo di clock) può essere ottenuto effettuando l'*issue* di più istruzioni per ogni colpo di clock. Ci sono due tipologie di processori con questa possibilità:

- *processori superscalari*, con scheduling statico o dinamico di istruzioni;
- processori *VLIW - Very Long Instruction Word*.

In entrambi i casi, i processori hanno unità funzionali multiple.

MIPS superscalare

Questa versione del MIPS implementa lo **scheduling statico**. Due istruzioni possono essere issued ad ogni colpo di clock se:

- Una delle istruzioni è una load, store, branch od operazione aritmetica intera;
- L'altra è un'operazione FP diversa da load e store.

Due istruzioni occupano 64 bit, e il fetch e decode sono effettuati ad ogni colpo di clock. Sono allineate a 64 bit e costituiscono un *issue packet*.

I vecchi processori superscalari richiedevano una struttura fissa dell'issue packet (prima l'istruzione intera, poi la FP): oggi questa struttura rigida normalmente non esiste più.
Ad ogni colpo di clock sono lette due istruzioni. Se le istruzioni appartengono a blocchi di cache separati, normalmente viene fatto il fetch solo di una. Normalmente è consentito un solo branch all'interno di un *issue packet*.

Contention di registri FP, RAW

Quando la prima istruzione è una load, store o move FP si può creare una 'contesa' per la porta del registro FP. Le possibili soluzioni sono forzare la prima istruzione ad essere eseguita in autonomia, oppure aggiungere una porta al register file FP.

Se la prima istruzione è una load, store o move FP e la seconda legge il suo risultato, è possibile un hazard RAW. In questo caso la seconda istruzione deve essere ritardata di un colpo di clock.

Multiple issue dynamic scheduling

Questo tipo di processori adottano un'architettura simile a quella di Tomasulo. Per rendere l'implementazione più semplice, le istruzioni non sono mai issued fuori ordine.

Il **CDB** può essere un collo di bottiglia, per questo motivo la performance può essere migliorata duplicandolo.