High Performance and Quantum Computing

Anno Accademico
2025/26

Rocco Lo Russo
Agostino D'Amora

Contents

Introduzione

In questo documento vengono raccolti appunti del corso *High Performance and Quantum Computing* tenuto nell'anno 2025-26 dal professor Cilardo presso l'università di Napoli Federico II, Dipartimento Ingegneria Elettrica e Tecnologie dell'Informazione.
È assolutamente scoraggiata la vendita di questi appunti, che possono in qualsiasi momento essere scaricati gratuitamente da github. Allo stessa repository è possibile in qualsiasi momento contruibuire seguendo le istruzioni esposte nel README.
Per la parte iniziale di ricapitolazione, sono stati utilizzati gli appunti raccolti durante il corso di Architettura e Progetto dei Calcolatori tenuto nell'anno 2025-26 dal professor Mazzocca presso l'università Federico II di Napoli (a cui è possibile accedere sia per consultare che per contribuire, nelle stesse modalità, su github), i libri *Computer Organization and Desing Patterns* (Hannessy-Patterson) e degli stessi autori *Computer Architecture (a quantitative approach)*.
Una buona parte degli esempi a cui si fa riferimento, e di cui per motivi di spazio non verrà riportato l'intero svolgimento, è tratta dai lucidi messi a disposizione dal professore sul canale teams del corso. Ove necessario, sarà fatto esplicito riferimento a quelli.

Chapter 1

Richiami APC

1.1 Richiami Pipelining

Il pipelining è una tecnica di implementazione di processori dove diverse istruzioni si sovrappongono durante l'esecuzione. Questa tecnica, chiave nella progettazione dei moderni processori, trae vantaggio dal parallelismo che esiste intrinsecamente tra le azione necessarie per l'esecuzione di un'istruzione. Nella pipeline, ogni passo necessario all'esecuzione di un'istruzione è svolto da un'unità funzionale autonoma, denominata *pipe segment* o *pipe stage*. Ogni unità è collegata alla successiva, e il throughput di una pipeline è determinato dal *tasso di attraversamento*, ovvero dalla frequenza con la quale un'istruzione esce dalla pipeline. Il tempo richiesto da un'istruzione per procedere allo stage successivo è denominato *processor cycle*, e dato che tutti gli stage devono necessariamente essere pronti a procedere allo stesso tempo, la lunghezza di un processor cycle è determinato dal tempo richiesto dallo stage più lento. In un computer il processor cycle è quasi sempre un colpo di clock.
Se il tempo di ogni stage è perfettamente bilanciato, chiamando T_i il tempo per ogni istruzione sul processore pipelined, T_{np} il tempo per ogni istruzione su un processore non pipelined ed n il numero di stages della pipe, allora vale la relazione: $T_i = \frac{T_{np}}{n}$.
Sotto queste condizioni ideali, lo speedup vale proprio n. Tuttavia, il tempo di ogni stage non è mai perfettamente bilanciato, e in più c'è da considerare il tempo di overhead introdotto dalla complessità del sistema. Una semplice pipline a cinque stadi presenta i seguenti blocchi funzionali:

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

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

Da questa presentazione emergono tre osservazioni: innanzitutto, è necessario separare memorie di istruzione e memorie dati, attraverso l'implementazione di diverse caches. Questa soluzione permette di evitare conflitti durante le fasi di IF e MEM; Il register file è usato in due stages, ovvero in ID e WB. Quindi, è necessario in un solo colpo di clock performare due letture e una scrittura, e questo si ottiene effettuando la scrittura nella prima parte di ciclo di clock e le lettura nella seconda parte; infine, è necessario considerare un circuito hardware che incrementi automaticamente il PC nella fase di IF, e un circuito che calcoli un eventuale indirizzo di branch durante lo stadio ID. La condizione del salto viene valutata in fase EXE, con una parte della ALU che confronta due registri.
Per fare in modo che l'output di uno stage non interferisca con lo stage successivo, vengono introdotti dei registri tra uno stage e l'altro, in modo che alla fine di un ciclo di clock, tutti i risultati di uno stage sono conservati in un registro che viene usato con input allo stage successivo al prossimo ciclo di clock. I registri svolgono un ruolo chiave anche nel trasporto di eventuali risultati intermedi dove stage di origine e destinazione non sono immediatamente adiacenti.

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

1.2 Pipeline Hazards

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

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

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

1.2.1 Hazards sui dati

La sovrapposizione di istruzioni introdotta dalla pipeline implica un cambiamento dell'ordine di accessi in lettura/scrittura agli operandi rispetto al modello sequenziale di Von Neumann. Un primo passo risolutivo è già stato presentato: quando si cerca di leggere o scrivere da uno stesso registro contemporaneamente, molte architetture prevedono che la scrittura avvenga nella prima metà del ciclo di clock mentre la lettura nella seconda metà. Questo risolve l'hazard quando la scrittura del registro avviene durante lo stesso ciclo di clock della lettura.
In linea di massima, il dato prodotto dall'istruzione da cui dipendono le successive è già pronto durante la fase di execute, quindi basterebbe semplicemente propagare il dato appena è possibile alle unità a cui serve prima che sia disponibile per la lettura dal file register. Ci occuperemo per semplicità solo del caso di propagazione in una fase EX, che può essere la propagazione di un risultato di una operazione ALU o il calcolo di un indirizzo effettivo. Questo significa che quando un'istruzione prova ad usare un registro nella sua fase EX che un'istruzione precedente deve ancora scrivere in fase di WB, può servirsi del valore anticipatamente come input controllato alla ALU.
Un caso in cui il forwarding non può risolvere l'hazard è quando un'istruzione prova a leggere un registro immediatamente dopo un'istruzione di *load* dalla memoria. Il dato risulterebbe ancora in fase di recupero dalla memoria, quindi è necessaria un'unità funzionale che stalli la pipeline. In generale, il compialtore, che è a conoscenza della struttura del processore, in casi di hazard detection di questo tipo si preoccupa di anticipare delle istruzioni indipendenti utilizzando la proprietà commutativa e associativa. Lo stallo è inevitabile quando non è possibile inserire dopo l'istruzione di accesso in memoria un blocco di istruzioni indipendenti. È possibile tuttavia un approccio, denominato **internal forwarding**, che permette di ibernare temporaneamente le istruzioni bloccate, e non stallare la pipeline. Bloccando istruzioni e caricandone altre, tuttavia può accadere che più istruzioni cerchino di operare sullo stesso registro, quindi occorre utilizzare una tecnica di indirizzamento indiretto che permette ai registri fisici del processore (*forwarding registers*) di puntare a dei registri in memoria che contengono valori precedenti assunti dal registro. Le istruzioni vengono ibernate in un'apposita area della memoria, la *ibernation table*.

1.2.2 Hazards di controllo

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

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

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

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

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

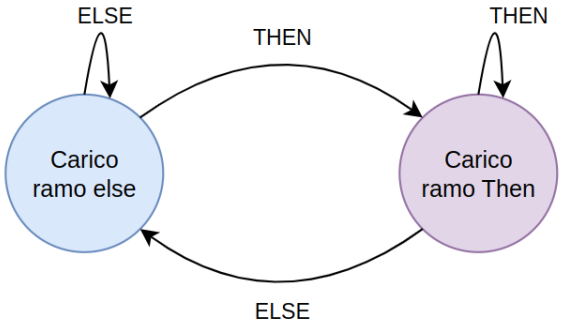


Figure 1.1: Automa della branch prediction base

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

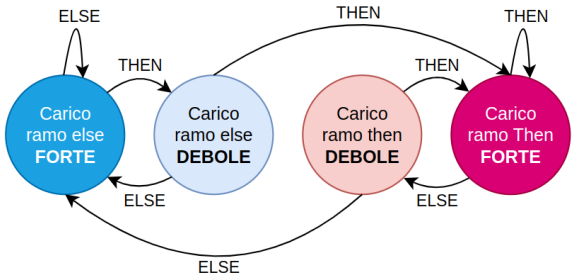


Figure 1.2: Automa della branch prediction avanzato

Per implementare la predizione a livello hardware, il processore utilizza una **tabella di predizione** dei salti. Una possibile struttura prevede i campi (Indirizzo dell'istruzione di salto, indirizzo di destinazione, 2 bit per codificare lo stato).
La tabella viene aggiornata in parallelo durante la fase di Execute (EX), ovvero nel momento in cui il processore verifica se la predizione è corretta. Se c'è stato un errore, si aggiorna lo stato dell'automa (e la corrispondente voce nella tabella). Tale struttura è memorizzata in un apposita area di memoria, detta **Branch History Table** (BHT).

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

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

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

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

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

1.3 Problemi nelle architetture superscalari

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

Chapter 2

Architetture superscalari dei moderni processori

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

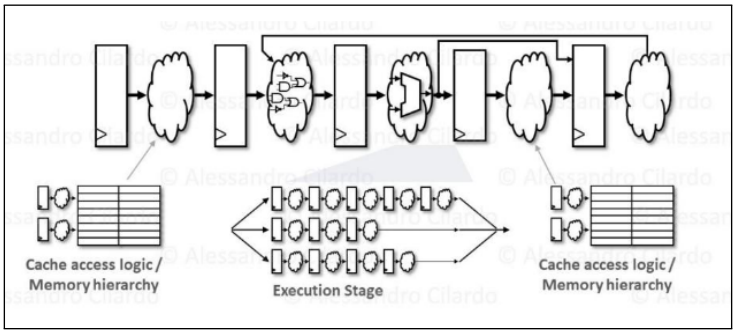
Info: Una macchina combinatoriale (o circuito logico combinatorio) è un sistema logico digitale in cui le uscite dipendono esclusivamente dai valori presenti agli ingressi in un dato istante, senza alcuna memoria degli stati precedenti.

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

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

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

ne scandisce un limite superiore. La tecnica della pipeline dunque non è facilmente scalabile: I *path* combinatoriali non possono essere arbitrariamente frammentati; inoltre, superata una certa granularità, gli overhead limitano i benefici.
Nel caso reale, nei diversi stages la latenza è variabile. Questo è dovuto al fatto che l'infrastruttura della memoria è complessa e nel caso peggiore potrebbero verificarsi dei *cache miss*, così come il processore potrebbe avere diverse unità funzionali dedicate all'esecuzione, ognuna con la propria latenza (alcune operazioni sono più lunghe e complesse di altre).



Per migliorare l'IPC, si può ricorrere alla moltiplicazione delle istanze di unità funzionali che lavorano in parallelo su più istruzioni. Se l'IPC > 1 si parla di architetture **superscalari**.
Parlando sempre di pipeline semplice a cinque stadi, possiamo invece agire sull'overhead introdotto dal controllo della pipeline, in particolare risolvere quei conflitti che provocherebbero il blocco (stallo) della pipe. Questo si può fare con l'esecuzione delle istruzioni *out of order*, ovvero con un problema di scheduling vincolato. La pipeline può essere vista nella prospettiva di schedulatore di micro operazioni soggette a vincoli strutturali e dipendenze. I vincoli strutturali sono dettati dell'hardware e scandiscono cosa si può fare in un particolare istante di tempo. I vincoli sulle dipendenze riguardano le relazioni di precedenza tra micro operazioni che devono necessariamente essere soddisfatte nel momento in cui vengono processate dalle unità hardware. Rischedulare le micro operazioni può violare i vincoli sulle dipendenze:

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

2.1 Scheduling statico

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

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

2.2 Scheduling dinamico

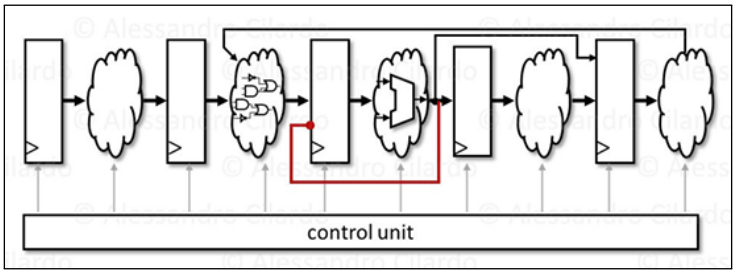
Con scheduling dinamico intendiamo tecniche hardware che *on the fly*, durante l'esecuzione, cambiano l'ordine di processo di determinate operazioni per risolvere i conflitti ed evitare quando possibile lo stallo della pipeline. I vantaggi di uno scheduling dinamico sono molteplici:

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

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

2.2.1 Forward paths

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



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

Il costo di questi benefici è una notevole complicazione dell'hardware. Osserviamo inoltre che lo scheduling dinamico è utlizzabile insieme allo scheduling statico effettuato dal compilatore: una tecnica non prescinde l'altra.
Osserviamo che in uno schema pipeline a cinque stages di base, dove non contempliamo istruzioni fuori ordine, è verificata sempre la condizione: *una micro-operazione di un'istruzione precedente è sempre eseguita prima della stessa o una successiva micro-operazione di un'istruzione successiva*. In altre parole, se un'istruzione A precede un'istruzione B, l'operazione di *lettura operandi* dell'istruzione A è eseguita prima delle operazioni EX, MEM e WB dell'istruzione B. Di conseguenza, in questo schema semplificato non sono possibili hazards di tipo WAW e WAR. Questo è in generale **falso** per gli approcci basati sul rescheduling delle operazioni.

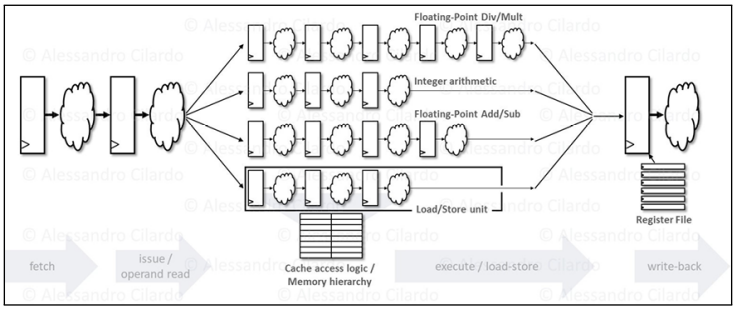
2.2.2 Esecuzione out-of-order

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

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

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

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



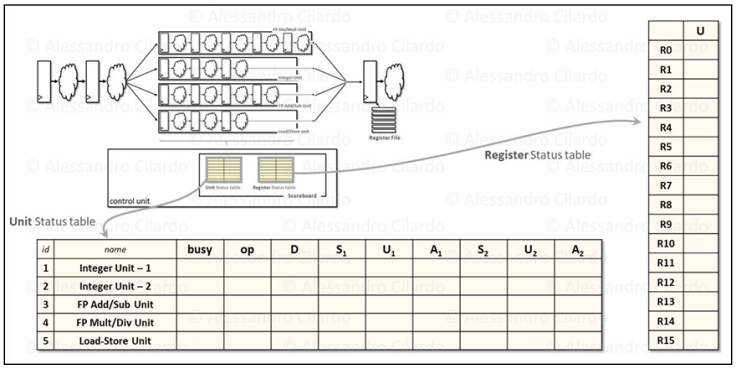
Nella trattazione teorica qui presentata conviene trattare l'unità di load/store (MEM) come un'unità funzionale al pari delle unità funzionali dedite alle operazioni aritmetico-logiche.

2.2.3 Scoreboard

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

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

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



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

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

Per quanto riguarda la Register Status Table, questa ha tante riche quanti sono i registri e l'unico campo U $\rightarrow$, che contiene l'id dell'unità che evnentualmente sta processando il valore che verrà scritto nel registro (=0 se già disponibile).
Le istruzioni sono recuperate dalla pipeline in ordine e poi possono trovarsi in una delle seguenti fasi: Issue, Operand Read, Execute o Write Back. Le regole per aggiornare le tabelle sono queste:

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

  - Il registro di destinazione non è già segnato (U=0 nella register status table);

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

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

next step	stalled if:	action performed when unstalled:
Issue	(US(*id*).busy) ∨ (RS(*D*).U≠0)	set the .busy, .op, .D, .S_1, .S_2 fields properly in US US(*id*).U_1←RS(*S_1*).U, US(*id*).A_1←*not*(US(*id*).U_1) US(*id*).U_2←RS(*S_2*).U, US(*id*).A_2←*not*(US(*id*).U_2) RS(*D*).U←*id*
Operan Read	A_1=0 ∨ A_2=0	US(*id*).A_1← 0 US(*id*).A_2← 0
Execute	(*depends on the internal FU behavior*)	-
Write-Back	∃ j : (US(j).S_1=D ∧ A_1=1) ∨ ∃ j : (US(j).S_2=D ∧ A_2=1)	∀ j: if US(j).U_1=*id* US(j).A_1←1 ∀ j: if US(j).U_2=*id* US(j).A_2←1 US(*id*).busy=0, RS(*D*).U← 0

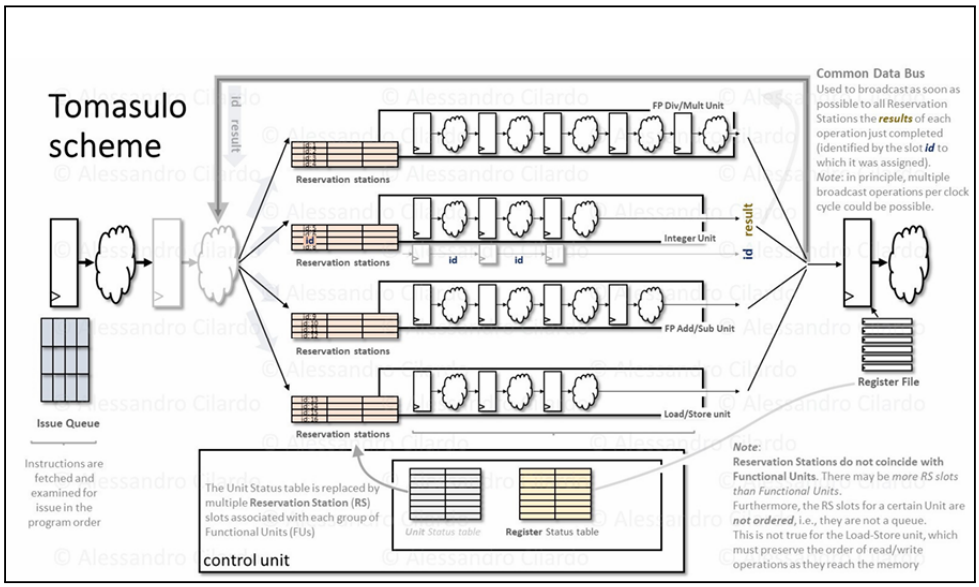
Si consultino le slides del corso per un esempio sulla tecnica dello scoreboard. In sintesi, questa tecnica di gestione previene gli hazards in questo modo: i RAW sono risolti tenendo traccia delle dipendenze tra le istruzioni che sono già state accettate; i WAR sono risolti ritardando il WB finchè i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

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

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

2.3 L'algoritmo di Tomasulo

Si tratta di un algoritmo inventato dall'informatico di IBM Robert Tomasulo, passato alla storia per aver *inventato* l'algoritmo che ha permesso l'esistenza dei processori che eseguissero istruzioni non in ordine. Gli ingredienti chiave dell'algoritmo sono le **reservation stations**: Le reservation stations sono dei buffer associati alle unità funzionali di un processore out-of-order. Servono a tenere in sospeso le istruzioni in attesa che i loro operandi diventino disponibili. In pratica, quando un'istruzione viene decodificata non entra subito nell'unità funzionale: viene invece "prenotato" uno slot in una reservation station, che contiene il tipo di operazione da eseguire, i riferimenti agli operandi (o i loro valori, se già disponibili) e il registro destinazione. Così, più istruzioni possono essere emesse in parallelo senza bloccare la pipeline, anche se non hanno ancora tutto ciò che serve per l'esecuzione.
L'idea di base dell'algoritmo è schedulare dinamicamente istruzioni in modo che possano iniziare appena gli operandi diventano disponibili, e duplicare i valori degli operandi (ovvero rinominare i registri) per evitare inutili hazards. Un altro elemento chiave è il Common Data Bus (CDB): quando un'unità funzionale calcola un risultato, lo invia sul bus comune, e tutte le reservation stations che aspettavano quel valore possono aggiornarlo immediatamente. In particolare, il CDB comunica la coppia (id, value) dove id è l'identificativo dello slot della RS a cui è stato assegnato il valore.



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

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

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

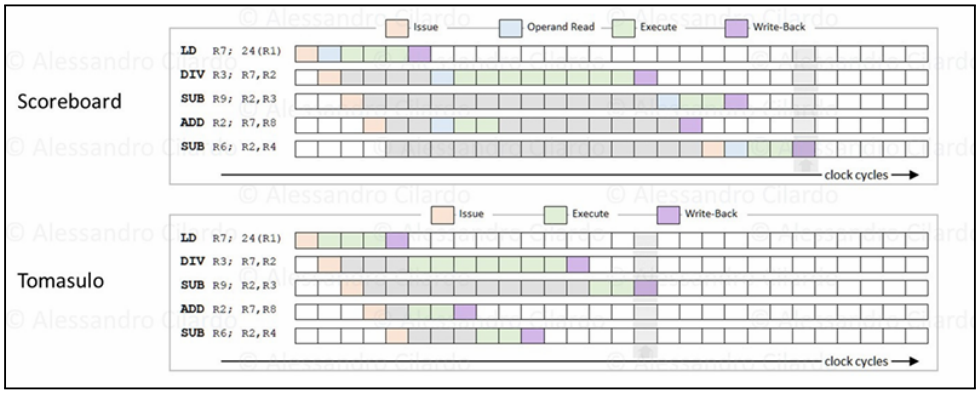
id	*name*	busy	op	St_1	V_1	St_2	V_2	Ad
1	Integer Unit (slot 0)							-
2	Integer Unit (slot 1)							-
3	Integer Unit (slot 2)							-
4	FP Add/Sub Unit (slot 0)							-
5	FP Add/Sub Unit (slot 1)							-
6	FP Mult/Div Unit (slot 0)							-
7	FP Mult/Div Unit (slot 1)							-
8	Load-Store Unit (slot 0)							
9	Load-Store Unit (slot 1)							

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

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

Notiamo subito che gli hazards strutturali sono ancora possibili nel caso che le tabelle vadano in overflow. Ma costruire RS più capienti è meno costoso di aggiungere unità funzionali.
Il miglioramento di performance, una volta evitati i conflitti dovuti alle false dipendenze, è palese ed è visibile nell'esempio presentato nelle slides del corso.

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



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

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