Capitolo 9

Architettura interna del processore

9.1 Introduzione

Vediamo una CPU un po' più moderna. La CPU vista fino ad ora abbiamo detto che fa, ciclicamente, `fetch`⟶`decode`⟶`execute` e che fra un'istruzione e l'altra controlla se ci sono richieste di interruzione. Anche se le CPU moderne, "viste da fuori", fanno la stessa cosa, dentro attuano una serie di accorgimenti per cercare di eseguire queste istruzioni il più velocemente possibile in modo tale che all'esterno l'effetto sia lo stesso ma all'interno si possa andare più velocemente. Per un lungo periodo di tempo l'aumento delle prestazioni di un processore nuovo, derivava semplicemente da un clock più veloce (praticamente raddoppiava ogni generazione). Quello che oggi i progettisti cercano di fare (anche se il clock rimane più o meno fermo) è aumentare lo spazio all'interno dello stesso chip (transistor più piccoli). Essendoci più spazio, la CPU può fare più cose e magari più cose contemporaneamente. Ultimamente quello spazio è usato per mettere all'interno dello stesso chip più processori.

9.2 *Pipeline*

Vediamo come in un unico core il processore cerchi di eseguire più istruzioni contemporaneamente (non da più flussi di esecuzione ma da uno solo). Per fare questo ci sono vari modi. Uno di questi è la ***pipeline***. Ogni istruzione deve passare da una serie di fasi. (Figura 9.1).

Prelievo istruzioni	Decodifica	Prelievo operandi	Esecuzione	Scrittura risultato

Figura 9.1: Fasi di una pipeline

Queste fasi sono eseguite da parti fisicamente diverse del processore (circuito che fa il prelievo, circuito che fa la decodifica, ALU, FPU, ecc..). Si pensi, ad esempio,

alla CEP[1] (Calcolatrice Elettronica Pisana) che ha la CPU letteralmente fatta da armadi ed ogni armadio rappresenta una diversa fase.

Se va tutto bene, quindi, nel momento in cui abbiamo prelevato un'istruzione e siamo in fase di decodifica, la parte di circuito che fa il prelievo in quel momento **non** è usata. Potremmo quindi utilizzare quella parte di circuito per prelevare la successiva istruzione (sappiamo quasi sempre l'istruzione successiva, l'unico caso in cui non lo sappiamo è quando c'è un salto).

Quindi avremo un diagramma temporale (caso ottimo) del tipo:

Figura 9.2: Diagramma temporale dell'utilizzo dei circuiti con una pipeline

Se va tutto bene questo procedimento può andare avanti a lungo.

In un processore semplice una sequenza `fetch-decode-execute`, possiamo dire che veniva fatta da un'unica rete combinatoria, in un unico seppur lungo ciclo di clock.

Se vogliamo passare a questa soluzione dobbiamo aggiungere dei registri a valle di ciascun circuito combinatorio che si occupa di ognuna delle fasi:

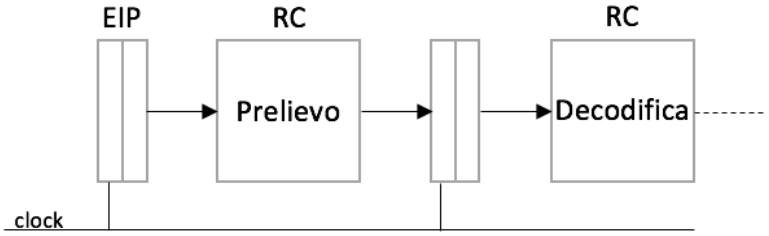

Figura 9.3: Registri posti a valle delle reti combinatorie adibite alle varie fasi

Poiché finché non arriva un segnale di sincronizzazione i registri sono insensibili alle variazioni di stato di ingresso, occorrono 5 cicli di clock per ogni istruzione. I registri sono indispensabili perché le RC cambiano "immediatamente" il valore dello stato di uscita quando c'è un nuovo stato di ingresso e per la struttura che vogliamo adottare questo rappresenterebbe un problema. Per esempio: se stessimo eseguendo la decodifica per l'istruzione i e prelevassimo l'istruzione $i + 1$, in ingresso alla RC di decodifica arriverebbe il nuovo stato dell'uscita della RC di prelievo.

Quindi ogni fase deve essere "scollegata" dalla precedente.

[1]La Calcolatrice Elettronica Pisana si trova esposta al Museo Nazionale degli Strumenti per il Calcolo di Pisa

Dal punto di vista della velocità:

- prima avevamo una macchina che in unico clock faceva "tutto" (un'istruzione, un ciclo di clock);
- ora una istruzione deve passare attraverso tutte quelle fasi; a ogni fase c'è da attraversare un registro e quindi servono 5 clock per istruzione.

La soluzione è aumentare la velocità del clock. Da cosa dipende la velocità del clock? Dal più lungo percorso tra un registro e l'altro all'interno del un circuito. E poiché nel processore la distanza fra un registro e l'altro è di almeno 5 reti combinatorie (mettiamo che il ritardo sia 5Δ), osserviamo che con questa soluzione la distanza fra un registro e l'altro diminuisce essendoci una sola RC a separali (mettiamo che il ritardo sia di 1Δ). Dal punto di vista delle singole istruzioni prima si aveva $\Delta + t_{setup1registro}$, ora $5\Delta + t_{setup5registri}$. Un po' si perde ma dal punto di vista complessivo è meglio perché abbiamo un'istruzione portata a termine ogni clock di 1Δ.

Per i processori INTEL la cosa è un po' più complicata perché non è possibile suddividere prelievo e decodifica poiché le istruzioni hanno lunghezza variabile e quindi non possiamo sapere quanto è grande l'istruzione finché non l'abbiamo letta.

Altro problema: siamo sicuri di non andare mai ad utilizzare per fasi diverse la stessa parte di circuito nella CPU?

Nel set di istruzioni dei processori INTEL purtroppo può capitare perché tutte le istruzioni possono avere gli operandi sia in memoria che nei registri e quindi se nello stesso momento la CPU deve fare un prelievo operandi di una istruzione e scrittura risultato di un'altra istruzione, potrebbe voler, per tutte le due istruzioni, accedere in memoria.

Quindi sorgono alcuni problemi.

9.3 Processori RISC

I processori RISC *(Reduced Instruction Set Computer)*, sono nati proprio con la concezione di utilizzare un set di istruzioni apposito per evitare quanto sopra descritto. Tutti i processori RISC tipicamente hanno istruzioni grandi 4 byte. All'interno di ciascuna istruzione i campi sono fissi:

Figura 9.4: Campi di un'istruzione RISC

Abbiamo anche una **netta** separazione fra le operazioni di memoria e quelle sui registri. Ci sono solo due istruzioni, `load` e `store`, per operare sulla memoria. Le istruzioni operative agiscono solo sui registri. Con queste semplificazioni si riesce a fare una pipeline.

Gli INTEL fanno questo:

via hardware le istruzioni IA32 passano da un modulo di traduzione che le trasforma in una sequenza di **istruzioni elementari** che invece sono RISC. Molte istruzioni di tipo CISC si traducono in un'unica istruzione elementare, altre no. Gli INTEL adoperano questa tecnica dal 686 (prima del Pentium).

Non possiamo sapere esattamente quali sono le IE, ma supponiamo:

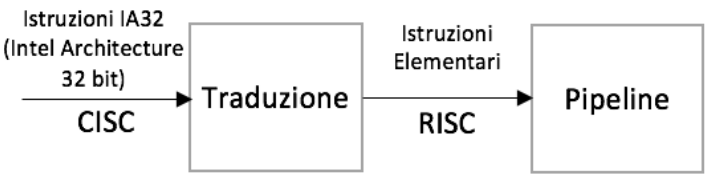

Figura 9.5: Traduzione da istruzioni CISC a istruzioni RISC

```
op dst, src1, src2 //Istruzioni Operative: tutti i registri.
    Destinatario e sorgenti sempre specificati.

op reg, offset(base) //Istruzioni di Memoria: load o store; hanno un
    unico formato.

op reg, offset //Istruzioni di Salto
```

9.4 Le e-istruzioni

Riprendiamo lo schema della pipeline:

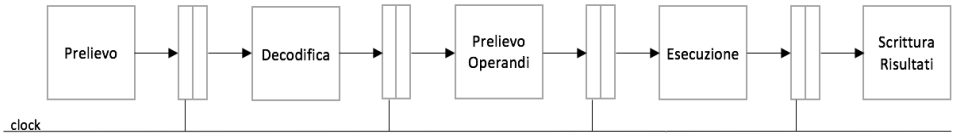

Figura 9.6: Schema pipeline completo con registi di pipeline

Ciascuno di questi circuiti si occupa di una fase diversa. Tra i vari stati inseriamo dei ***registri di pipeline*** in modo tale che ogni stato, ad ogni clock, scriva il risultato nel registro di pipeline mentre lo stato precedente sta lavorando sul risultato di prima. Per poter utilizzare questa struttura, i progettisti INTEL hanno dovuto prendere le istruzioni Assembly ereditate dalle vecchie macchine e predisporre un circuito atto a trasformarle in *e-istruzioni*. Le e-istruzioni hanno i seguenti vantaggi:

- hanno tutte la stessa lunghezza (non occorre sapere quanto sommare, sono tutte grandi 4 byte);
- il fatto che siano semplici permette di ridurre il clock al minimo. Infatti il valore minimo di clock è dato dal tempo più lungo che impiega una fase per portarsi a termine. Solo se le istruzioni sono semplici si può garantire un'uniformità fra i tempi di esecuzione delle varie fasi.

9.5 Problemi legati all'introduzione della pipeline

All'interno del flusso delle istruzioni ci sono alcune situazioni che impediscono di eseguire un'istruzione ad ogni ciclo di clock. Queste situazioni prendono il nome di *alee* e all'interno del flusso di istruzioni possono causare quelli che si chiamano ***stalli della pipeline***. Gli stalli sono cicli di clock in cui non viene incominciata una nuova istruzione e quindi perdiamo quel ciclo di clock.

Abbiamo alee di tre tipi:

- alee **strutturali**;
- alee **sui dati**;
- alee **sul controllo**.

Le **alee strutturali** dipendono dal fatto che, in teoria, per eseguire 5 istruzioni in parallelo occorre che ad ogni passo queste istruzioni facciano uso di risorse distinte del processore. Queste risorse potrebbero non essere distinte per ogni possibile combinazione di istruzioni. Basti pensare a due istruzioni i e $i+2$ di cui la prima deve scrivere il risultato e la seconda deve prelevare operandi in memoria: entrambe vorrebbero utilizzare il circuito che serve per operare sulla memoria ma ciò non è possibile, oltre al fatto che alla memoria non si può accedere contemporaneamente.

Questa situazione può essere risolta con uno stallo; fermiamo, cioè, il flusso di esecuzione e lo facciamo ripartire al clock successivo. In generale, se c'è un conflitto fra le istruzioni, possiamo sempre risolverlo mettendo la pipeline in stallo per almeno un ciclo di clock (al massimo 5 cicli di clock essendo 5 le fasi). Vogliamo però ridurre gli stalli al minimo perché ogni stallo è un clock sprecato. Assumiamo che ci sia un circuito a parte che controlla il flusso della pipeline e decide quando far entrare qualcosa di nuovo. Lo fa sapendo quale istruzione è già nella pipeline e quale sia la nuova istruzione e in base a questo decidere se lasciare in stallo la pipeline permettendo dunque ad una nuova istruzione di essere eseguita.

Le **alee sui dati** sono istruzioni che usano il risultato di un'istruzione precedente. Per esempio:

```
ADD R1, R2, R3
SUB R4, R1, R5
```

La e-istruzione `SUB` ha come operando il risultato della e-istruzione `ADD` precedente ad essa. Quando la `SUB` deve prelevare l'operando da `R1`, la `ADD` si trova sempre in fase di esecuzione. In questo caso per risolvere l'alea dovremmo aggiungere due stalli.

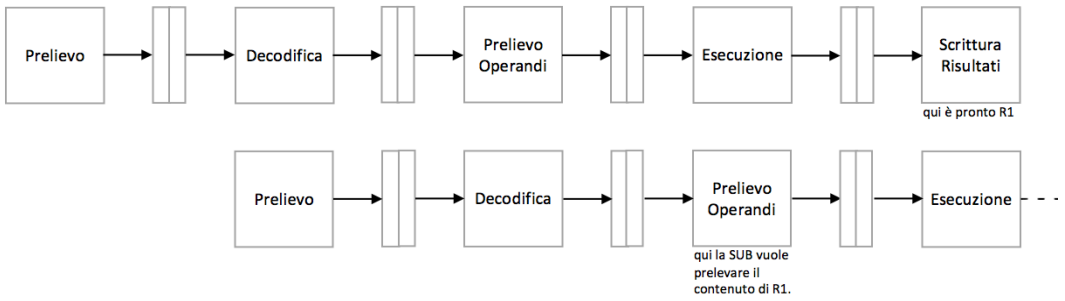

Figura 9.7: Stallo sui dati, schema relativo all'esempio

Situazioni come queste sono molto frequenti e non possiamo permetterci di perdere clock così spesso.

La soluzione è possibile ottenerla attraverso l'inserimento di un circuito di *by-pass* pilotato da quel circuito che controlla il flusso della pipeline.

È vero che l'istruzione `SUB` vuole leggere il registro `R1`, ma più che il registro le interessa il suo contenuto. Il risultato della `ADD` è pronto già subito dopo la fase di esecuzione; quindi se predisponessimo un circuito di *by-pass* che ci consentisse di ottenere il risultato appena questo è pronto, potremmo risolvere gli stalli anche

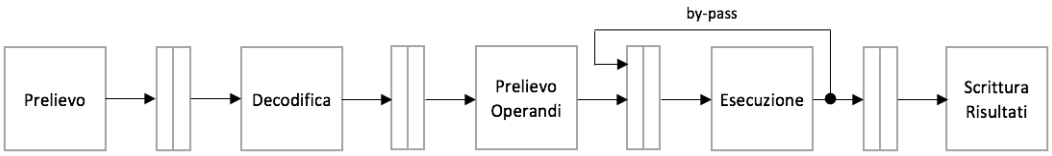

Figura 9.8: Schema pipeline con by-pass

in questo caso. Ci dovrà essere uno switch che decide quale delle due entrate del registro deve passare.

L'ultima categoria di alee sono le **alee sul controllo**. Si verificano quando ci sono istruzioni di salto, soprattutto di salto condizionato. Non sappiamo, finché non è stata seguita l'istruzione di salto, quale sia l'istruzione successiva. Quindi nel frattempo cosa inseriamo nella pipeline? La cosa più semplice sarebbe non inserire nulla e aspettare che l'istruzione abbia calcolato l'indirizzo corretto a cui saltare. Questa soluzione, però, data la frequenza con cui vengono utilizzate le istruzioni di salto, rallenterebbe troppo il flusso di esecuzione. Un'altra cosa che però possiamo fare è provare a indovinare il risultato dell'istruzione di salto. Indovinando è possibile sbagliare e magari si scopre, al termine dell'istruzione di salto, che la parte a cui siamo saltati è la parte sbagliata. Questo non rappresenta un problema perché queste istruzioni non hanno ancora fatto nulla di permanente. È sufficiente, prima che queste istruzioni abbiano scritto il loro risultato, conoscere il risultato dell'istruzione di salto ed eventualmente mettere un flag ad indicare che le istruzioni eseguite fino a quel punto non devono avere effetti. Come si fa ad "indovinare" qual è il risultato dell'istruzione di salto? Bisogna vedere cosa è successo in passato e sperare che accada più o meno la stessa cosa. Il modo più semplice per fare questa cosa è una *predizione statica*[2]: se l'offset è negativo potrebbe essere un ciclo e se è un ciclo è probabile che un po' di volte occorrerà ripetere quelle istruzioni. Se il salto è in avanti forse siamo in presenza di una condizione e quindi la predizione è molto più scarsa. Possiamo, in questo caso, predire che il salto non verrà fatto. I meccanismi di previsione che sono nel processore fanno in realtà qualcosa di più sofisticato ricordando di quella istruzione cosa sia successo in passato. Hanno una cache delle istruzioni di salto già viste e per ognuna di esse si ricordano la storia dei salti (una serie di bit).

9.6 Esecuzione fuori ordine

Una pipeline è soltanto la tecnica più elementare per cercare di velocizzare il funzionamento di un processore.

Esistono altri metodi per ottenere questo effetto. Supponiamo che ad un certo punto ci accorgiamo che una certa istruzione non può passare perché c'è un'alea strutturale e quell'istruzione ha bisogno della ALU in due stadi e non possiamo farla partire subito dopo la precedente. Invece di sprecare quel clock, possiamo vedere se possiamo far passare l'istruzione successiva. Bisogna però porre attenzione. Nel

[2]I processori che impiegano questa tecnica considerano sempre i salti verso la parte precedente del codice come "accettati" (ipotizzando che siano le istruzioni riguardanti un ciclo) e i salti in avanti sempre come "non accettati" (ipotizzando che siano uscite precoci dal ciclo o altre funzioni di programmazione particolari). Per cicli che si ripetono molte volte, questa tecnica fallisce solo alla fine del ciclo. Fonte: https://it.wikipedia.org/wiki/Predizione_delle_diramazioni

flusso di esecuzione di un programma è abbastanza frequente che istruzioni che sarebbero sequenziali, in realtà non dipendono l'una dall'altra. Esempio classico è un ciclo for per inizializzare un array di N elementi con la somma degli elementi di altri due array di N elementi. Ciascuna somma è a sé, non ha alcuna importanza l'ordine in cui vengono fatte. L'importante è che alla fine vengano fatte tutte. Quindi nel flusso delle e-istruzioni verranno fuori un migliaio di istruzioni che non importa che vengano lasciate in quell'ordine. Questa cosa si può sfruttare progettando il processore in modo che esegua le istruzioni nel primo momento possibile. Questo si può fare purché l'istruzione abbia le risorse per essere eseguita e abbia i suoi operandi. La tecnica descritta si chiama ***esecuzione fuori ordine***. Serve a risolvere gli stalli; invece di attendere, si vede se l'istruzione successiva può essere eseguita.

Possiamo eseguire una e-istruzione nel primo momento in cui ci sono risorse libere per la sua esecuzione tenendo conto delle seguenti dipendenze:

- dipendenza **sui dati**;
- dipendenza **sui nomi**;
- dipendenza **sul controllo**.

Una e-istruzione dipende sui **dati** se usa un risultato prodotto dalla e-istruzione precedente oppure anche *transitivamente*, cioè legge il risultato di una e-istruzione intermedia che dipende sui dati dall'istruzione ancora precedente.

La dipendenza sui nomi è un po' più particolare. Possiamo averne di due tipi:

```
//R2 sorgente e poi destinatario
ADD     R1, R2, R3
ADD     R2, R4, R5

//R1 destinatario e poi nuovamente destinatario
ADD     R1, R2, R3
ADD     R1, R4, R5
```

La dipendenza sul *controllo* si ha quando una e-istruzione può essere eseguita o meno in base al risultato della e-istruzione di salto.

Queste dipendenze sono importanti perché se una e-istruzione dipende, per un motivo qualunque, dalla e-istruzione precedente, non possiamo riordinarle a meno che non adottiamo qualche accorgimento.

Le dipendenze sui dati non possono essere risolte perché sono quello che il programma sta facendo e non possiamo modificare il significato del programma.

Le dipendenze sui nomi possono, invece, essere risolte.

9.7 Organizzazione interna del processore

Vediamo che è possibile organizzare il processore in modo da permettere l'esecuzione fuori ordine tenendo conto delle dipendenze. Possiamo avere più ALU (più ce ne sono più il processore costa).

Davanti a ciascuna ALU mettiamo una pila di e-istruzioni. Queste strutture prendono il nome di **stazioni di prenotazione**. In queste stazioni di prenotazione si vanno ad accodare le e-istruzioni in attesa che siano pronti i loro risultati.

Per ogni registro ci ricordiamo un paio di cose:

- se c'è una e-istruzione di quelle in coda che vorrebbe scriverci (W);

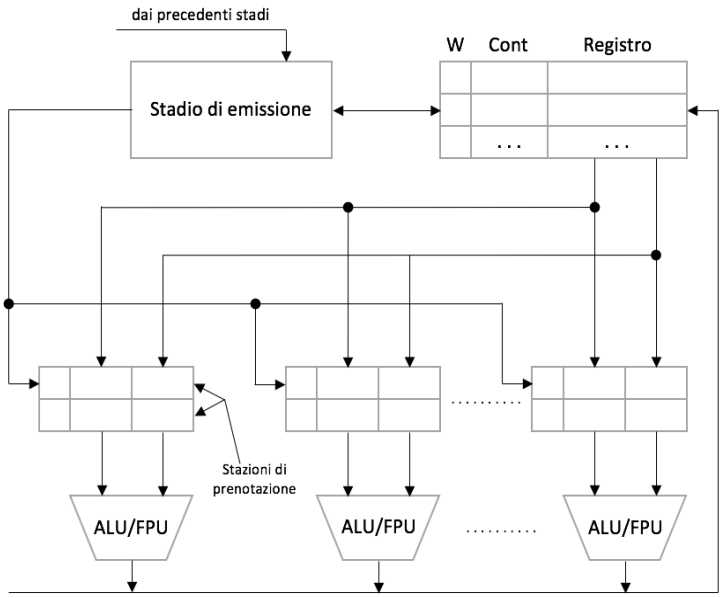

Figura 9.9: Schema con evidenziazione delle stazioni di prenotazione e della struttura dati W-Cont-Registro

- quante delle e-istruzioni che sono in cosa vorrebbero leggerci (Cont).

Lo stadio di emissione consulta la tabella ed emette le e-istruzioni, in una di quelle code dove c'è posto e dove ci sono le risorse per eseguire la e-istruzione.

I risultati delle ALU vengono portati poi nella tabella per scrivere nei registri.

Dalla struttura dati introdotta W-Cont-Registro, possiamo già capire se l'istruzione dobbiamo fermarla perché viola qualche dipendenza oppure se possiamo farla già partire.

9.7.1 Regole per emettere una e-istruzione

Per prima cosa guardiamo la destinazione e guardiamo il flag W: se W è a 1 abbiamo dipendenza sui dati perché la e-istruzione vuole leggere un risultato di una e-istruzione che deve ancora essere messa in coda; questa la emettiamo ugualmente e andrà a finire in una stazione di prenotazione, quella della ALU che la potrà eseguire, in attesa che i suoi operandi siano pronti. Quando sono pronti si prelevano e poi si va ad eseguire l'istruzione.

9.7.2 Rinomina dei registri

Come già accennato, le dipendenze sui nomi possono essere eliminate. La tecnica che utilizziamo è quella che viene chiamata **rinomina dei registri**.

Prendiamo il seguente esempio:

```
ADD     R1, R2, R3      #usiamo R2 come sorgente
ADD     R2, R4, R5      #usiamo R2 come destinatario
SUB     R6, R2, R7
```

Usiamo `R2` come sorgente nella prima `ADD` e come destinatario nella seconda `ADD`: la scelta di `R2` come destinatario non e' fondamentale infatti avremmo potuto scegliere un altro registro. Pertanto questo esempio potrebbe essere riscritto come:

```
ADD     R1, R2, R3      #usiamo R2 come sorgente
ADD     R20, R4, R5     #usiamo R20 come destinatario
SUB     R6, R20, R7
```

Apportando questa modifica non abbiamo alterato il senso del programma, ma siamo riusciti ad eliminare una dipendenza sui nomi.

Questa cosa il processore la può fare in autonomia aggiungendo un componente che fa la rinomina dei registri.

Tutto ciò si può fare "parlando" indirettamente dei registri.

Distinguiamo **registri logici** e **registri fisici**.

I registri logici sono quelli che compaiono nel flusso del programma. Quelli su cui si fanno davvero i conti sono i registri fisici. Possiamo anche avere più registri fisici che logici.

La struttura dati W-Cont-Registri, diventerà dunque una struttura del tipo W-Cont-Registri Logici-Registri Fisici.

Per ogni registro logico viene specificato mediante una struttura costituita da puntatori, quale registro fisico contiene il suo valore.

Avremo dunque una struttura del genere:

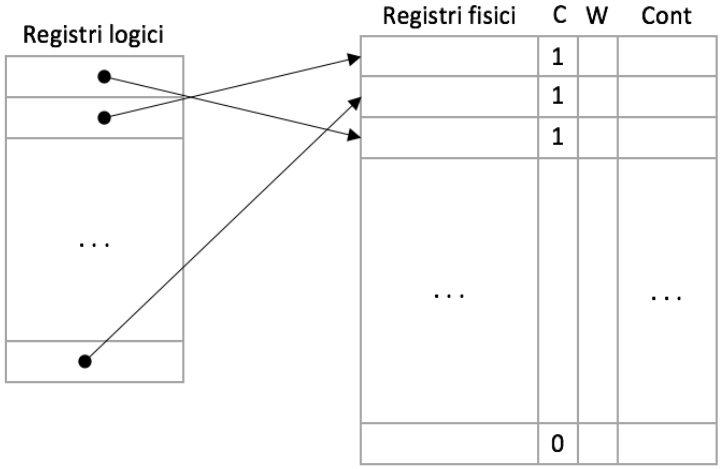

Figura 9.10: Corrispondenza registri logici-registri fisici

Quando una e-istruzione viene emessa, si utilizza la corrispondenza esistente fra registri logici sorgente `(src1, src2)` e registri fisici e viene creata una nuova corrispondenza fra il registro logico destinatario (dest) e un registro fisico libero.

I campi aggiuntivi sono associati ai registri fisici:

- W: a 1 se nel registro fisico F deve scrivere almeno una e-istruzione emessa;
- Cont: numero delle e-istruzioni emesse che deve leggere dal registro F;
- C: a 1 se vi è una corrispondenza fra un registro logico e il registro fisico F a cui il bit C appartiene.

Un registro fisico F si considera libero se:

- C = 0 $\longrightarrow$ nessuna corrispondenza di F con alcun registro logico;
- W = 0 $\longrightarrow$ nessuna e-istruzione emessa deve scrivere in F;

- Cont = 0 $\longrightarrow$ nessuna e-istruzione emessa deve leggere da F.

Emissione della e-istruzione:

- incremento di Cont dei registri fisici corrispondenti ai registri logici sorgente coinvolti;
- settaggio dei bit C e W per il registro fisico corrispondente al registro logico destinatario; dunque creiamo una nuova corrispondenza;
- resettaggio dei bit C e W per i registri fisici corrispondenti ai registri destinatari della vecchia corrispondenza.

Completamento della e-istruzione:

- decremento del campo Cont per i registri fisici corrispondenti ai registri logici sorgente;
- resettaggio di W del registro fisico corrispondente al registro logico destinatario.

9.8 Esecuzione speculativa

La tecnica della rinomina dei registri, opportunamente affiancata alla tecnica della predizione dei salti, ci permette anche di aggirare le limitazioni dovute alle dipendenze sul controllo. Possiamo eseguire le e-istruzioni dipendenti da e-istruzioni di salto non ancora risolte purché i risultati di queste vengano scritti in registri temporanei e trasferiti nei registri reali **solo** quando abbiamo la certezza che quelle e-istruzioni andavano realmente eseguite. Questa tecnica è detta ***speculazione***. Per dare corpo a questa tecnica occorre introdurre un nuovo stadio detto di *ritiro delle istruzioni*. Il completamento delle e-istruzioni provoca la scrittura dei risultati solo in registri temporanei: tali risultati diverranno effettivi solo se la e-istruzione che li ha prodotti passa lo stadio di ritiro. Lo stadio introdotto fa uso di una struttura dati chiamata ROB (*ReOrder Buffer*, buffer di riordino). Il ROB è una coda di descrittori di e-istruzioni.

Per ogni e-istruzione il ROB ne memorizza il tipo (se è operativa o di controllo), se è stata completata o è ancora in fase di esecuzione e, se la e-istruzione è di controllo, qual è l'esito previsto per il salto. Le e-istruzioni possono essere completate in un qualsiasi ordine ma sono ritirate dalla testa del ROB nel modo in cui sono immagazzinate in esso. Quando la e-istruzione in testa al ROB risulta completata, viene ritirata compiendo le seguenti azioni:

- se la e-istruzione è operativa:
  - il risultato da essa prodotto diventa effettivo e questa viene estratta dalla coda
- sa le e-istruzione è di controllo:
  - se la previsione dell'esito del salto è **sbagliata**, viene svuotato tutto il ROB con conseguente annullamento delle istruzioni eventualmente completate (scrittura nei registri fisici destinatari e modifica dei campi C, W, Cont associati ai registri fisici utilizzati);

– se la previsione dell'esito del salto è **corretta**, non viene fatta nessuna azione e questa e-istruzione viene estratta dalla testa del ROB.

Per poter rendere effettivi o annullare i risultati di una e-istruzione occorre estendere la struttura dati registri logici-fisici in modo che ogni registro logico possegga una doppia corrispondenza con registri fisici (due puntatori).

Nel ROB viene anche memorizzato temporaneamente il risultato delle e-istruzioni.

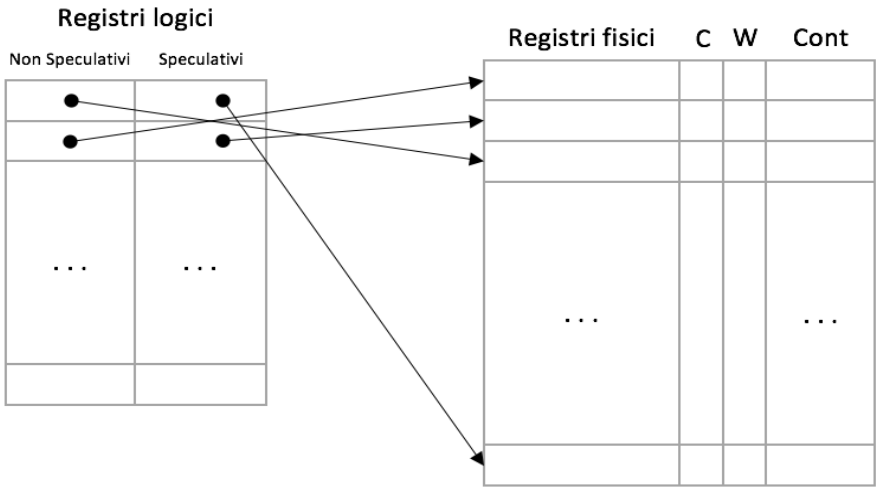

Figura 9.11: Registri logici speculativi e non speculativi e corrispondenza con registri fisici

Abbiamo quindi due tipi di corrispondenza per ogni registro logico:

- corrispondenza **non speculativa**: puntatore al registro fisico in cui è contenuto il valore dell'ultima e-istruzione ritirata dal ROB che lo aveva come destinatario;
- corrispondenza **speculativa**: puntatore al registro fisico che contiene (o conterrà) il valore corrente.

Il significato dei valori di C, W e Cont rimane invariato rispetto a quello precedentemente visto; solo alcune precisazioni:

- quando il bit W vale 1 si dice che il registro fisico è **bloccato**;
- il bit C vale 1 quando il registro fisico è selezionato da un puntatore (sia esso speculativo o non speculativo).

Il registro fisico si dice **libero** se C, W e Cont valgono 0. Un registro fisico si dice:

- **bloccato** se W = 1;
- **sbloccato** se W = 0;

Per le ragioni viste servono più registri fisici che registri logici; al massimo avremo che i registri fisici sono il doppio di quelli logici.

Quando si emette una e-istruzione, è sufficiente guardare i valori speculativi; se la e-istruzione è arrivata in cima al ROB si copia il valore speculativo in quello non speculativo. Se invece in cima al ROB arriva una e-istruzione di controllo e si scopre che per questa si era predetto il salto in modo **sbagliato**, tutti i valori speculativi sono sbagliati e possiamo rimediare alla situazione copiando gli ultimi valori che erano corretti nei valori speculativi.

9.9 Esempio sul funzionamento dell'architettura avanzata del processore

Partiamo da un frammento di programma C++

```
1 for(int i = 0; i<n; i++)
2         a[i]+=b[i];
```

trasformiamolo in Assembly:

```
 1 movl    n, %ebx
 2 movl    $0, %ebx
 3 ciclo:
 4 cmpl    %ebx, %ecx
 5 jge     fine
 6 movb    b(%ecx), %al
 7 movb    %al, a(%ecx)
 8 incl    %ecx
 9 jmp     cliclo
10 fine:
11 #...
```

Il linguaggio Assembly è quello che ci permette di avere una corrispondenza diretta con il linguaggio macchina. Abbiamo parlato di flusso delle istruzioni ma il programma scritto in Assembly non rappresenta il flusso delle istruzioni; eseguendolo, infatti, possiamo ottenere diversi flussi di esecuzione. Molte delle considerazioni che abbiamo fatto in passato non hanno senso in questo contesto: per esempio dire che l'istruzione alla riga 7 è successiva a quella alla riga 6 non ha senso perché siamo in un ciclo e se guardiamo alla singola iterazione l'affermazione è vera, ma se guardiamo tutte le iterazioni vediamo che l'istruzione alla riga 7 della prima iterazione è precedente all'istruzione alla riga 6 della seconda iterazione rendendo così l'affermazione falsa.

Quindi, supponendo n = 2, il "processo" di esecuzione sarà:

```
 1 movl    n, %ebx
 2 movl    $0, %ebx
 3 ciclo:
 4 cmpl    %ebx, %ecx
 5 jge     fine
 6 movb    b(%ecx), %al
 7 movb    %al, a(%ecx)
 8 incl    %ecx
 9 jmp     cliclo
10 cmpl    %ebx, %ecx
11 jge     fine
12 movb    b(%ecx), %al
13 movb    %al, a(%ecx)
14 incl    %ecx
15 jmp     cliclo
16 cmpl    %ebx, %ecx
```

```
17 jge      fine
18 fine:
19 #...
```

Queste sono le istruzioni che il processore effettivamente preleverà eseguendo quel programma; quindi nel flusso delle istruzioni compaiono più volte le istruzioni che si trovano all'interno del ciclo. Quindi l'ordine delle istruzioni dipende non dal programma visto prima ma dall'esecuzione del flusso delle istruzioni da parte del processore. Abbiamo inoltre visto che in realtà non è neanche questo che entra nella pipeline; queste istruzioni vengono tradotte in e-istruzioni. A questo livello abbiamo sia i registri classici architetturali (come `EBX`, `ECX`, `EDX` ecc..) sia quelli interni in più che il processore usa esclusivamente per la traduzione.

```
 1 LOAD     %ebx, n(R0)
 2 MOV      %ecx, $0
 3 SUB      R1, %ecx, %ebx
 4 JLE      fine, R1
 5 LOAD     %al, b(%ecx)
 6 LOAD     R1, a(%ecx)
 7 ADD      R1, R1, %al
 8 STORE    a(%ecx), R1
 9 ADD      %ecx, %ecx, $1
10 JMP      ciclo
11 SUB      R1, %ecx, %ebx
12 JLE      fine, R1
13 LOAD     %al, b(%ecx)
14 LOAD     R1, a(%ecx)
15 ADD      R1, R1, %al
16 STORE    a(%ecx), R1
17 ADD      %ecx, %ecx, $1
18 JMP      ciclo
19 SUB      R1, %ecx, %ebx
20 JLE      fine
```

N.B.: il processore traduce le istruzioni in e-istruzioni man mano che le preleva, ma non ha una visione completa di tutto il flusso di esecuzione.

A questo punto possiamo chiederci dove sono le dipendenze fra le istruzioni.

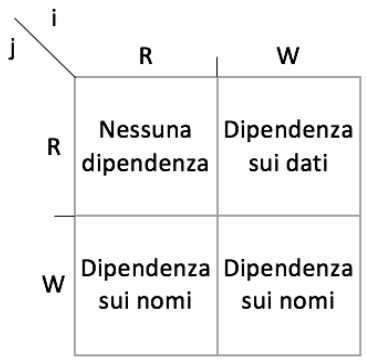

Figura 9.12: Schema delle dipendenze

dove la e-istruzione $j >$ della e-istruzione i.
Note sulle dipendenze:

- è sempre una e-istruzione successiva che dipende da una e-istruzione precedente;

- non importa che le due e-istruzioni siano adiacenti;
- una dipendenza, escludendo per il momento quelle sul controllo, esiste se e solo se due e-istruzioni fanno uso dello stesso registro.

Prese due e-istruzioni i e j con j successiva a i, la dipendenza è una cosa che impedisce di scambiare le due e-istruzioni. Nel flusso sequenziale di un processore che esegue un'istruzione per volta, la e-istruzione j verrebbe eseguita dopo la e-istruzione i. Con il processore in esame, la domanda che ci poniamo è: possiamo scambiare le due e-istruzioni senza alterare il senso del programma? Sì se fra loro non ci sono dipendenze.

Due istruzioni che leggono entrambe dallo stesso registro non si danno alcun fastidio e fra loro non esiste alcuna dipendenza.

La dipendenza sui nomi non nasce da quello che il programma deve fare poiché è conseguenza del riuso dei registri.

La memoria ha lo stesso identico problema dei registri; se vogliamo, uno stesso indirizzo di memoria è come se fosse uno stesso registro. Quindi le dipendenze fra le istruzioni si possono avere anche in memoria. Abbiamo però una complicazione in più rispetto ai registri infatti

```
LOAD    R1, a(%ecx)
#...
STORE   a(%ecx), R1
```

come facciamo a sapere l'indirizzo esatto? Non certo guardando semplicemente le e-istruzioni. Per sapere il valore dell'indirizzo occorre in parte eseguire l'istruzione.

In ogni caso, anche dal flusso delle istruzioni che viene fuori da una riga legittima di C++, si vede ad occhio che non è necessario fare tutte le cose con l'ordine prestabilito.

Il processore queste cose le fa in autonomia; in particolare potrebbe anche accorgersi che le istruzioni di diverse iterazioni dello stesso ciclo, può eseguirle anche parallelamente poiché lavorano su cose completamente diverse.

Per risolvere le dipendenze sul controllo, come già visto, il processore si serve del ROB:

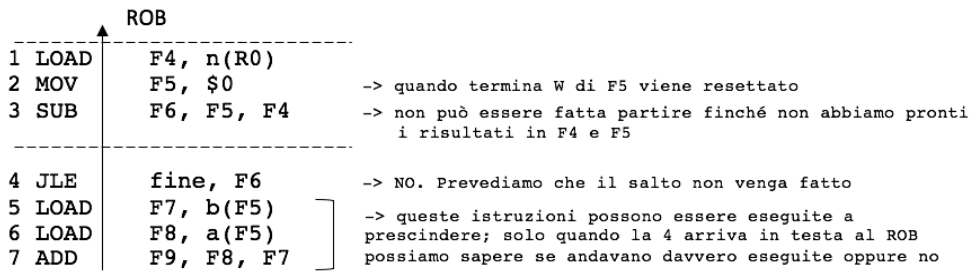

Figura 9.13: Contenuto del ROB relativo all'esempio

Se la previsione del salto era corretta, i valori non speculativi vengono aggiornati con gli attuali valori speculativi. Se la previsione del salto era sbagliata, e dunque le e-istruzioni non andavano in realtà eseguite, i valori speculativi vengono ripristinati con i corrispondenti valori non speculativi che sono gli ultimi che sicuramente andavano calcolati.

Il valore non speculativo relativo ad un registro logico, viene aggiornato quando l'istruzione esce dal ROB con il valore del registro nell'istruzione, non con il valore speculativo attuale.

La e-istruzione nella riga 3 della figura non può essere eseguita finché non ha pronti i risultati che deve utilizzare. Può, però, essere ugualmente emessa e accodata in una delle stazioni di prenotazione in attesa che i dati che le servono siano pronti. Questo ci consente di non fermarci.

I registri di destinazione li rinominiamo sempre anche quando non vi è alcun tipo di dipendenza.

Ad ogni rinomina e ad ogni istruzione aggiunta nel ROB, occorre aggiornare opportunamente la tabella Registri fisici-C-W-Cont.

Le istruzioni nel ROB possono essere eseguite in un qualunque ordine (tenendo conto delle dipendenze), una volta che hanno pronti i dati, ma devono essere ritirare dal ROB nell'ordine che rappresenta il flusso delle istruzioni voluto dal programma.

Quando un'istruzione esce dal ROB siamo sicuri che andava eseguita.

Non si smette mai di accodare le e-istruzioni nelle stazioni di prenotazione, lo facciamo solo quando non ci sono più risorse disponibili.

Vediamo adesso una struttura dati di cui si serve il processore:

Registri logici:

- **non speculativi**: ultimo valore buono calcolato;
- **speculativi**: valori correnti.

In questo caso supponiamo che le stazioni di prenotazione siano sufficientemente capienti da non causare alee strutturali.

Le uniche dipendenze che ci danno fastidio sono quelle sui dati perché quelle sui nomi le risolviamo andando a rinominare i registri di destinazione.

Approfondimento...

Come mai *gcc*, se traduciamo in Assembly, non fa le push ma usa altre istruzioni?

Prendiamo una funzione `f(a, b, c)`, noi in Assembly traduciamo:

```
1 push    c
2 push    b
3 push    a
4 call    f
5 #...
```

Così facendo abbiamo una dipendenza sui dati, le `push` non possono essere invertite perché implicitamente decrementano `ESP` e vanno fatte esattamente in questo ordine per ottenere l'ordine giusto degli argomenti attuali nel record di attivazione.

`gcc` invece fa:

```
1 sub     $12, %esp
2 movl    a, (%esp)
3 movl    b, 4(%esp)
4 movl    c, 8(%esp)
5 call    f
6 #...
```

Le `mov` hanno tutte una dipendenza con la prima istruzione, ma, fra loro, possono essere eseguite in parallelo dal processore.