Esecuzione fuori ordine

G. Lettieri

3 Giugno 2025

Il limite principale della pipeline è la sua natura strettamente sequenziale: se una istruzione si blocca in uno stadio, tutti gli stadi precedenti devono fermarsi. Tra i casi più problematici, si pensi ad una istruzione che deve accere alla memoria nel caso in cui l'informazione cercata non si trovi in cache: l'istruzione dovrà restare bloccata nello stadio in cui ha eseguito l'accesso per tutto il tempo necessario a completare la lettura dalla memoria, e tutte le istruzioni successive (che si trovano in stadi precedenti della pipeline) dovranno aspettare. Con la tecnica dell'esecuzione fuori ordine, invece, il processore può esaminare le istruzioni successive e, se non "dipendono" da quella bloccata, farle andare avanti, in modo da poter svolgere lavoro utile mentre è in corso l'accesso in memoria.

Ovviamente dobbiamo capire cosa vuol dire che una istruzione dipende da un'altra. Vediamo prima un esempio: consideriamo un frammento di programma che calcoli una somma vettoriale:

```
for (int i = 0; i < 1000; i++)
      a[i] = b[i] + c[i];
```

Il programma deve calcolare

```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
...
a[999] = b[999] + c[999];
```

ma non ha alcuna importanza che queste istruzioni vengano eseguite esattamente in quest'ordine: il risultato sarà lo stesso qualunque sia l'ordine. Supponiamo, per esempio, che `b[0]` non si trovi in cache: questo vuol dire che il calcolo del valore da scrivere in `a[0]` deve aspettare che `b[0]` venga prelevato dalla memoria, ma nel frattempo è del tutto lecito provare ad eseguire le istruzioni successive, perché il loro risultato *non dipende* da quanto vale `b[0]`.

L'idea può essere estesa ulteriormente: non solo non è necessario che le varie somme siano eseguite nell'ordine che aveva specificato il programmatore, ma non è neanche necessario che vengano eseguite sequenzialmente. Se avessimo 1000 ALU, potremmo anche eseguirle tutte contemporaneamente.

1 Dipendenze tra le istruzioni

Vediamo più precisamente cosa si intende per dipendenza tra istruzioni. Facciamo riferimento alle istruzioni di tipo RISC e ad un flusso di esecuzione, così come lo genererebbe un processore che esegua le istruzioni nell'ordine specificato dal programma. Consideriamo una istruzione i ed una istruzione j che la segue nell'ordine del programma (non necessariamente immediatamente dopo). Per il momento consideriamo solo le istruzioni operative "*op*, *src1*, *src2*, *dst*" e quelle di salto, tralasciando le istruzioni di accesso alla memoria. Diremo che j dipende da i:

- *per i dati* se j usa direttamente o indirettamente il risultato prodotto da i;
- *per i nomi* se j scrive in uno qualunque dei registri a cui accede i (sia sorgenti che destinazione);
- *per il controllo* se j può essere eseguita o meno in base al risultato prodotto da i.

Vediamo un esempio di dipendenza (diretta) sui dati:

```
add x1, x2, x3
// altre istruzioni che non scrivono in x3
sub x4, x3, x5
```

Consideriamo come j l'istruzione `sub` e i l'istruzione `add`. L'istruzione `add` lascia il proprio risultato nel registro `x3`, e questo stesso risultato è letto dalla `sub`. Quindi la `sub` dipende (direttamente) per i dati dalla `add`: non possiamo eseguire la `sub` fino a quando la `add` non ha calcolato il suo risultato.

Vediamo invece quest'altro caso:

```
add x1, x2, x3
// altre istruzioni
sub x4, x5, x1
// altre istruzioni
mul x6, x7, x3
```

Consideriamo come i sempre l'istruzione `add` e come j la `sub`. L'istruzione `sub` scrive in uno dei registri dell'istruzione `add` (`x1`). Lo stesso vale se prendiamo come j l'istruzione `mul` (in questo caso il registro è `x3`). In entrambi i casi l'istruzione j scrive in uno dei registri della `add`, e dunque j dipende da i per i nomi. Se, per esempio, eseguissimo la `sub` prima della `add`, o mentre la `add` non è ancora terminata, la `add` potrebbe usare il valore scorretto di `x1`. Nel caso della `mul` il problema sarebbe diverso: se eseguissimo la `mul` prima della `add`, l'ultimo valore scritto in `x3` (da cui magari dipendono per i dati altre istruzioni successive) sarebbe sbagliato.

Possiamo riassumere le dipendenze sui dati e sui nomi nel seguente schema, in cui prendiamo in considerazione due istruzioni i e j, con j successiva ad i

nell'ordine del programma, tali che entrambe usino uno stesso registro come operando sorgente (quindi in lettura, abbreviato in "r") o come destinatario (scrittura, "w"):

j \ i	r	w
r	–	dati
w	nomi[1]	nomi[2]

Se entrambe le istruzioni leggono il registro non c'è dipendenza. Se i scrive e j legge lo stesso registro, j dipende da i per i dati. Se j scrive, allora dipende da i per i nomi. Il caso 1 (i legge e j scrive) viene detto anche *antidipendenza*, mentre il caso 2 (entrambe scrivono) è detto *dipendenza in uscita*. Nell'esempio precedente, l'istruzione `sub` ha una antidipendenza con l'istruzione `add`, mentre l'istruzione `mul` ha una dipendenza in uscita.

Infine, consideriamo le seguenti istruzioni:

```
  jz x1, avanti
  add x2, x3, x4
avanti:
  sub x5, x6, x7
```

L'istruzione `add` può essere eseguita oppure no, in base al risultato dell'istruzione `jz`: diciamo che la `add` dipende per il controllo dalla `jz`.

A differenza delle dipendenze sui nomi e sui dati, che possono essere scoperte osservando un singolo flusso di esecuzione di un programma, le dipendenze sul controllo richiedono la conoscenza dell'intero programma. Nell'esempio precedente la `sub` viene eseguita in ogni caso, e dunque non dipende per il controllo dalla `jz`, ma ci basta modificare il programma nel seguente modo:

```
  jz x1, avanti
  add x2, x3, x4
  jmp fine
avanti:
  sub x5, x6, x7
fine:
```

ed ecco che anche la `sub` può essere eseguita oppure o no, in base al risultato della `jz`, e dunque dipende per il controllo da `jz`. Questa differenza è importante perché noi vorremmo che fosse il processore stesso a scoprire le dipendenze, ma il processore non conosce il programma: conosce solo le istruzioni che preleva, e dunque osserva solo un flusso di esecuzione (o processo, in senso astratto). In particolare, se la `jz` effettua il salto, il processore vedrà che la prossima istruzione è la `sub`, senza sapere quali altre istruzioni sono state saltate, e dunque senza poter distinguere i due casi qui sopra. Il processore è dunque costretto a fare una approssimazione molto cruda: appena preleva una istruzione di salto condizionale, deve assumere che da quel momento in poi *tutte* le istruzioni dipendano per il controllo da essa.

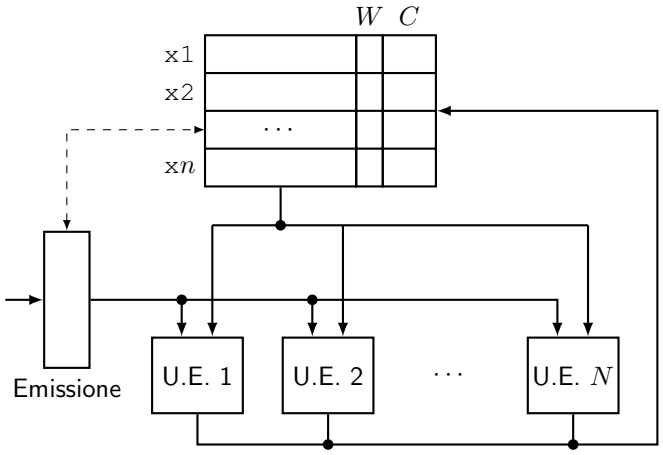

Figura 1: Architettura per l'esecuzione fuori ordine (caso base)

Quando abbiamo parlato di pipeline abbiamo introdotto le alee sui dati e sul controllo: attenzione a non confonderle con le omonime dipendenze. Le dipendenze sono una proprietà del programma da eseguire, mentre le alee sono delle situazioni indesiderate che si possono presentare o meno in base a come è strutturato il processore. Le dipendenze, se non correttamente gestite, possono portare a delle alee.

2 Stadio di emissione e dashboard

Introduciamo ora una architettura che ci permette di eseguire le istruzioni fuori ordine o in parallelo, tenendo traccia delle dipendenze. Per il momento continuamo a preoccuparci solo delle istruzioni operative e di salto di tipo RISC, che hanno operandi esclusivamente di tipo registro (o al massimo di tipo immediato). Partendo dalla pipeline che abbiamo già visto, sostituiamo gli stadi di lettura operandi, esecuzione e scrittura operandi con l'architettura di Figura 1. Introduciamo:

- un nuovo stadio di Emissione, che riceve le istruzioni, ancora in ordine, dallo stadio di decodifica, e deve decidere se lasciarle proseguire oppure no;
- più di una unità di esecuzione, ciascuna in grado di eseguire una istruzione in parallelo con le altre unità;
- una *dashboard* che contiene informazioni su ciascun registro, oltre ai registri stessi.

Le istruzioni che si trovano negli stadi di esecuzione possono essere completate in qualsiasi ordine, scrivendo il loro risultato nel rispettivo registro di destinazione.

Le dipendenze tra queste istruzioni vanno tracciate e gestite in modo da evitare eventuali alee. Il compito di evitare le alee spetta allo stadio di emissione, ma ci sono molti modi in cui può farlo, più o meno efficientemente.

Partiamo da un caso base, che possiamo poi migliorare. Nel caso base, ogni volta che riceve una nuova istruzione, sia j, lo stadio di emissione deve verificare che j non abbia dipendenze con nessuna delle istruzioni che si trovano negli stadi di esecuzione; in caso contrario lo stadio va in stallo (bloccando anche gli stadi precedenti) fino a quando l'alea non si risolve, altrimenti *emette* l'istruzione e passa a valutare la sucessiva. L'emissione comporta in particolare che l'istruzione j viene assegnata ad uno stadio di esecuzione libero e inizia la sua esecuzione. Se ci fosse una istruzione i da cui j dipende e lo stadio di emissione non bloccasse j, si genererebbe un'alea, in quando j, una volta emessa, potrebbe terminare la propria esecuzione prima di i. Si noti che le uniche dipendenze che possono generare alee sono però solo queste, tra l'istruzione j appena arrivata e quelle che si trovano ancora negli stadi di esecuzione. Eventuali dipendenze tra j e istruzioni già completate non possono generare alee, in quanto j verrà completata sicuramente dopo di esse.

Per riconoscere la presenza di dipendenze, lo stadio di emissione consulta la dashboard, che contiene due informazioni per ogni registro: un flag W che vale 1 se una qualche istruzione in esecuzione ha quel registro come destinatario, e un contatore C che conta quante istruzioni in esecuzione hanno quel registro come sorgente. Quando una istruzione viene emessa, incrementa i campi C dei suoi registri sorgenti e setta a 1 il campo W del suo registro destinatario. Quando una istruzione viene completata, decrementa i campi C dei propri registri sorgenti e riporta a 0 il campo W del proprio registro destinatario.

Supponiamo che lo stadio di emissione riceva una istruzione j del tipo "*op*, *src1*, *src2*, *dst*". Lo stadio consulta i campi W e C del registro *dest*: se uno o entrambi sono diversi da zero, l'istruzione j dipende per i nomi da una o più istruzioni attualmente in esecuzione (dipendenza in uscita se $W = 1$ o antidipendenza se $C \neq 0$), ed emetterla potrebbe causare una o più alee. Per evitarle, lo stadio di emissione entra in stallo fino a quando entrambi i campi W e C di *dst* non tornano a zero. Si noti che questo implica che, ad ogni istante, ciascun registro è scritto da al più una istruzione in esecuzione. Lo stadio consulta anche le entrate della dashboard relative ai registri *src1* e *src2* di j. Se il campo W di *src1* vale 1, una qualche istruzione i, ancora in esecuzione, ha come destinatario lo stesso registro da cui j vuole leggere, e dunque j dipende per i dati da i. L'istruzione j non può essere emessa in questo momento, perché altrimenti avremmo una alea sui dati. Anche in questo caso lo stadio di emissione entra in stallo fino a quando W non passa a zero. Lo stesso vale per il registro *src2*. Riassumendo, il caso base prevede che lo stadio di emissione emetta le istruzioni operative solo quando il registro destinatario non è attualmente usato ($W = 0$ e $C = 0$) ed entrambi i registri sorgenti non sono oggetto di scrittura ($W = 0$).

Se lo stadio di emissione riceve una istruzione di salto condizionale, controlla che il registro sorgente non abbia W pari a 1 (dipendenza sui dati), altrimenti entra in stallo fino a quando W non passa a zero. Una volta emessa l'istruzione,

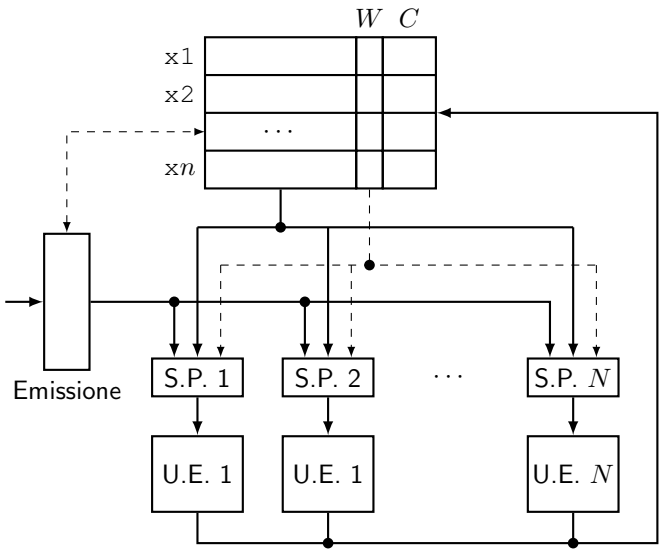

Figura 2: Ottimizzazione delle dipendenze sui dati

smette di ricevere nuove istruzioni fino a quando il salto non è stato completato (per evitare alee sul controllo).

3 Ottimizzazione delle dipendenze sui dati

Ogni stallo dello stadio di emissione riduce le prestazioni, quindi è importante cercare di eliminarli. Una prima ottimizzazione riguarda le dipendenze sui dati: l'osservazione fondamentale è che ogni istruzione può aspettare che siano pronti i propri dati indipendentemente dalle altre. Per esempio, consideriamo il seguente frammento di programma:

```
div x1, x2, x3
add x3, x4, x5
div x6, x7, x8
add x8, x9, x10
```

Lo stadio di emissione, in base alle regole stabilite fino ad ora, dovrebbe entrare in stallo alla prima `add`, per via della dipendenza sui dati con la prima `div`. Invece, introduciamo le cosiddette *stazioni di prenotazione* come in Figura 2. Lo scopo delle stazioni è di fornire un buffer in cui una istruzione può essere memorizzata in attesa che siano pronti i propri operandi (e nel frattempo tenere occupato uno stadio di esecuzione che possa eseguirla). Lo stadio di emissione copia le istruzioni in una stazione di prenotazione libera, dove l'istruzione monitora lo stato dei flag W dei proprio operandi sorgenti, e viene eseguita appena entrambi sono pronti. Nel frattempo lo stadio di emissione può procedere con le

istruzioni successive. Nell'esempio precedente, se ci sono 4 stadi di esecuzione, lo stadio di emissione può emettere tutte le istruzioni senza fermarsi: ciascuna di esse occuperà uno stadio di esecuzione; le due `div` potranno essere eseguite sostanzialmente in parallelo e, al loro termine, entrambe le `add` potranno essere eseguite anch'esse in parallelo.

È possibile anche avere stazioni di prenotazione in grado di memorizzare più di una istruzione, in genere organizzate in una coda FIFO davanti ad ogni stadio di esecuzione.

4 Ottimizzazione delle dipendenze sui nomi

Consideriamo una leggera variazione della sequenza di istruzioni precedente:

```
div x1, x2, x3
add x3, x4, x5
div x6, x7, x3
add x3, x9, x10
```

In questa nuova versione, la seconda `div` sta passando il risultato alla seconda `add` tramite il registro `x3` invece del registro `x8`. È chiaro che il risultato finale è lo stesso, ma questa nuova versione introduce delle dipendenze suoi nomi: la seconda `div` ha ora una antidipendenza con la prima `add` e una dipendenza in uscita con la prima `div`, e dunque può causare uno stallo nello stadio di emissione. Se il programmatore (o il compilatore) avesse scelto `x8`, o un qualunque altro registro non utilizzato, al posto di `x3`, il risultato della sequenza sarebbe stato lo stesso e lo stallo non ci sarebbe stato. In generale, quando indichiamo un registro come operando sorgente di una istruzione, quello che vogliamo realmente indicare è il risultato di una precedente istruzione: l'esatto registro usato non ha importanza. Se avessimo un numero sufficiente di registri, potremmo memorizzare il risultato di ogni istruzione in un registro diverso, e a quel punto non avremmo più dipendenze sui nomi. Questo è ovviamente irrealizzabile, ma possiamo sfruttare un'altra osservazione: non ci interessa eliminare *tutte* le dipendenze sui nomi, ma solo quelle che possono causare alee, e queste sono solo le dipendenze tra le istruzioni attualmente in esecuzione, che sono un numero ragionevole. Ci basta dunque avere un registro diverso per contenere il risultato di ogni istruzione in esecuzione.

La soluzione più semplice sarebbe quindi di prevedere un numero di registri sufficientemente grande, ma ci sono problemi pratici che ce lo impediscono: il numero di unità di esecuzione o di stazioni di prenotazione può essere aumentato facilmente al progredire della tecnologia, ma per aumentare il numero di registri visibili dal programmatore bisogna cambiare il set delle istruzioni. Anche quando ciò può essere fatto senza perdere la compatibilità con i programmi già esistenti, questi non potrebbero beneficiare dell'esistenza dei nuovi registri, a meno di non essere riscritti o ricompilati.

Possiamo però aumentare il numero di registri in modo totalmente trasparente per il software, introducendo uno stadio di *rinomina dei registri* tra lo

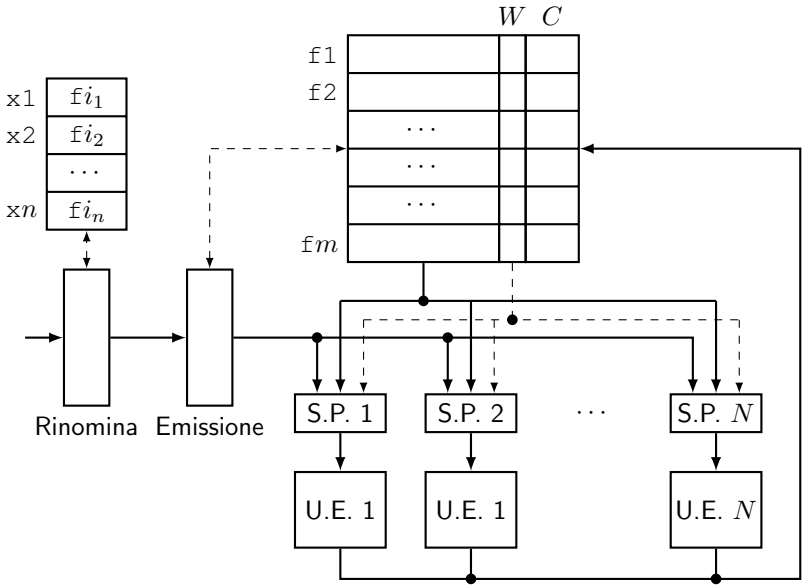

Figura 3: Ottimizzazione delle dipendenze sui nomi

stadio di decodifica e quello di emissione (Figura 3). Chiamiamo *registri logici* i registri usati dal programmatore/compilatore, e introduciamo un numero (anche molto maggiore) di registri *fisici*. I registri logici hanno il solo scopo di specificare le relazioni tra le istruzioni (chi deve ricevere i dati da chi), ma il loro contenuto può trovarsi in uno qualsiasi dei registri fisici. Aggiungiamo una tabella che, ad ogni istante, associ ogni registro logico al registro fisico che contiene (o conterrà, se l'ultima istruzione che ha il registro logico come destinazione è ancora in esecuzione) l'ultimo valore scritto in quel registro fino a quell'istante. Chiamiamo `f1`, `f2`, ... i registri fisici. La dashboard conterrà le informazioni relative ai registri fisici, non più i logici. Quando lo stadio di rinomina riceve una nuova istruzione "*op*, *src1*, *src2*, *dst*", per prima cosa sostituisce *src1* e *src2* con i registri fisici corrispondenti in quel momento; quindi trova un registro fisico al momento inutilizzato ($W = 0$ e $C = 0$), sia `f`n, e crea una nuova associazione tra *dst* e `f`n (eventualmente la stessa già esistente). Per esempio, la sequenza precedente potrebbe essere trasformata nella seguente, supponendo che all'inizio tutti i registri fisici siano inutilizzati:

```
div f1, f2, f3
add f3, f4, f5
div f6, f7, f8
add f8, f9, f10
```

In particolare, all'arrivo della seconda `div` con destinazione `x3`, lo stadio di rinomina osserva che il corrispondente registro `f3` è attualmente in uso (scritto dalla prima `div` e letto dalla prima `add`), quindi sceglie un nuovo registro, per

esempio `f8`, e associa `x3` a `f8`. All'arrivo della seconda `add`, che ha `x3` come primo operando sorgente, lo stadio di rinomina usa la nuova corrispondenza per permettere all'istruzione di leggere il risultato corretto, dal registro `f8`.

Lo stadio di emissione, a questo punto, non troverà mai dipendenze sui nomi, in quanto lo stadio di rinomina ha fatto in modo che le destinazioni di tutte le istruzioni abbiano sempre $W = 0$ e $C = 0$. Abbiamo dunque eliminato tutti gli stalli di questo tipo.

5 Istruzioni che accedono alla memoria

Consideriamo ora le istruzioni che accedono alla memoria: "`ld` *offset*(*base*), *dst*" che esegue una lettura, e "`st` *src*, *offset*(*base*)" che esegue una scrittura. Consideriamo due istruzioni i e j, in cui j viene dopo i ed è una `st`, mentre i è una `ld` o una `st`. Anche in questo caso possiamo avere dipendenze sui nomi e sui dati, se i e j accedono ad uno o più byte in comune. Rispetto al caso delle istruzioni operative e di salto, ci sono delle difficoltà in più:

- non possiamo pensare di avere una dashboard che mantenga W e C per ogni byte della memoria;
- per sapere se ci sono byte in comune, occorre prima calcolare gli indirizzi (sommando *offset* al contenuto del registro *base*).

Possiamo rendere il problema più gestibile se rinunciamo a parte del parallelismo o dei possibili riordinamenti. Per esempio, se prevediamo un'unica unità di esecuzione per le scritture in memoria, tutte le `st` saranno eseguite nell'ordine in cui vengono emesse, evitando *a priori* le alee dovute alle dipendenze sui nomi in uscita. Per risolvere le altre dipendenze (dati e antidipendenze) occorre procedere in due fasi. L'unità di esecuzione delle scritture avrà un cosiddetto *store buffer*, in cui le istruzioni `st` vengono inserite in ordine ed estratte in ordine una alla volta per essere eseguite. Ogni entrata dello store buffer contiene un campo "indirizzo" e un campo "dato"; il campo "indirizzo" contiene l'indirizzo completo, una volta calcolato, e il campo "dato" il valore da scrivere, letto dal registro sorgente dell'istruzione. Quando entrambi i campi sono stati riempiti l'istruzione è pronta per essere eseguita, ma la scrittura verrà effettuata solo quando tutte le scritture che la precedono saranno completate. L'istruzione verrà rimossa dallo store buffer solo quando la sua scrittura sarà stata completata. Lo store buffer, quindi, contiene informazioni su tutte le scritture "in corso" (emesse, ma non ancora completate), che sono le uniche che possono causare alee.

Le istruzioni `ld` devono andare in uno o più *load buffer* (possiamo averne più di uno, in quanto l'ordine delle letture non è importante). Ogni entrata dei load buffer memorizza l'indirizzo da cui l'istruzione vuole leggere. Per poter proseguire, una istruzione `ld` deve attendere che vengano calcolati tutti i campi "indirizzo" delle istruzioni `st` che si trovano già nello store buffer. A quel punto, bisogna confrontare l'indirizzo della `ld` con tutti quegli indirizzi e, in caso di

sovrapposizione, attendere che le corrispondenti `st` vengano completate. Questo evita le alee dovute alle dipendenze sui nomi. Come ottimizzazione, una `ld` che trovi i dati che sta cercando nello store buffer, può ottenerli direttamente da lì (*store forwarding*).

Infine, restano le alee causate dalle antidipendenze. Per risolvere anche queste è sufficiente che le istruzioni `st`, appena il loro indirizzo è stato calcolato, aspettino a loro volta che vengano calcolati tutti gli indirizzi di tutte le `ld` che erano già in qualche load buffer quando la `st` è stata emessa. Poi è necessario confrontare l'indirizzo della `st` con tutti gli indirizzi delle `ld` e, in caso di sovrapposizione, attendere che le rispettive `ld` vengano prima completate.

6 Realizzazione nei processori Intel

I meccanismi precedenti, così complessi già per le semplici istruzioni RISC, sembrerebbero impossibili da applicare al processore x86, ma con il Pentium Pro del 1995 l'Intel ha trovato il modo di introdurre queste soluzioni avanzate anche nei propri processori. Il trucco, usato in tutti i processori Intel da allora in poi, è di decodificare internamente le istruzioni x86 in microistruzioni simili alle istruzioni RISC, e poi eseguire le microistruzioni usando una architettura simile a quella descritta qui sopra. Il formato delle istruzioni RISC usato dall'Intel non è noto ufficialmente, ma se assumiamo come esempio le istruzioni che abbiamo usato fin'ora, possiamo pensare che una istruzione complessa come "`add %rax, offset(%rbx, %rcx, 8)`" venga tradotta nella sequenza

```
shl 3, rcx, x1
add x1, rbx, x2
ld  offset(x2), x3
add rax, x3, x4
st  x4, offset(x2)
```

Nella sequenza vediamo che, oltre ai registri "architetturali" `rax`, `rcx`, etc., si usano anche altri registri "microarchitetturali", non accessibili al programmatore, per contenere risultati intermedi (nell'esempio li abbiamo chiamati `x1`, `x2`, etc.). Lo stadio di decodifica può anche riusare sempre gli stessi registri microarchitetturali per i risultati intermedi, semplificando così il suo compito di traduzione. Il successivo stadio di rinomina dei registri provvederà a risolvere tutte le dipendenze sui nomi così introdotte.