1 Lezione del 06-05-25

Riprendiamo più nel dettaglio i concetti che abbiamo solamente accennatto alla scorsa lezione.

1.0.1 Ordinamento delle istruzioni

Abbiamo che l'*ordinamento totale* delle istruzioni del programma è solo un artefatto di come questo è compilato: non necessariamente le istruzioni vanno eseguite in tale ordine per arrivare allo stesso risultato finale.

Le istruzioni si trovano infatti solo in un *ordinamento parziale*, dove solo alcune istruzioni hanno bisogno, per motivi di sincronia, di essere eseguite successivamente ad altre.

Prendiamo quindi l'esempio `op1, src1, sec2, dst`, con sintassi simile all'assembly ARM (e in generale delle istruzioni RISC, dove ricordiamo gli operandi sono sempre registri). Una volta che i sorgenti `src1` e `src2` sono definiti, questa può essere eseguita.

Ipotizziamo quindi un'architettura dove sono previste un numero arbitrario di ALU, preceduta da componenti che denominiamo **stazioni di prenotazione**, fondamentalmente registri capaci di contenere la codifica macchina di una istruzione.

Ogni istruzione che viene decodificata dal processore viene spostata in una stazione di prenotazione. Non appena gli operandi saranno pronti, l'istruzione potrà quindi essere messa in esecuzione.

Viene da sé che questa architettura ci permette di ottenere un'esecuzione delle istruzioni che è *fuori ordine* e in *parallelo*.

1.0.2 Dipendenze

Chiaramente, però, resterà da definire le regole secondo le quali le istruzioni devono essere eseguite prima o dopo di altre. Chiamiamo queste condizioni **dipendenze**, fra cui distinguiamo:

- Dipendenze **dati**;
- Dipendenze **nomi**;
- Dipendenze **controllo**;

dove la prima e l'ultima non vanno confuse con le alee: adesso parliamo di conseguenze di come è fatto il *programma*, non di come è fatto il *processore*.

Le dipendenze **dati** sono il caso che abbiamo già visto. Se abbiamo due istruzioni:

```
1 add r1, r2, r3
2 ...
3 sub r4, r3, r5
```

si ha che la `sub` dipende per dati dalla `add`, in quanto questa intacca `r3`, operando.

Per ogni registro dovremo quindi mantenere delle informazioni associate: quando una istruzione viene messa in una stazione (si dice viene **emessa**), si alza un certo flag associato al registro che intaccherà. Questo flag viene poi abbassato in fase di esecuzione vera e propria, insieme all'aggiornamento dei dati del registro stesso. Un istruzione successiva che vuole usare tale registro dovrà quindi controllare tale flag per capire se quell'operando è pronto.

Le dipendenze sui **nomi** si dividono in due categorie:

- **Antidipendenze**: Poniamo di avere le due istruzioni:

```
add r1, r2, r3
...
sub r4, r5, r1
```

  in questo caso non si può eseguire la `sub` prima della `add`, in quanto la prima necessita del *vecchio* contenuto di `r1`, non quello che avremo dopo la `sub`. Si deve quindi evitare di mettere in attesa la `sub` finché la `add` non è conosciuta.

  Per risolvere questo tipo di dipendenza dovremmo quindi dotare ogni registro, oltre al flag W, un contatore C che conti quante stazioni contengono istruzioni che usano quel registro come sorgente. Ogni istruzione emessa alza i contatori dei suoi sorgenti, e ogni istruzione eseguita fino in fondo li abbassa. Avremo quindi che un istruzione potrà scrivere sulla sua destinazione quando il numero di lettori, e quindi il contatore, è pari a 0.

- Dipendenze in **uscita**: Poniamo di avere le due istruzioni:

```
add r1, r2, r3
...
sub r4, r5, r3
```

  che può sembrare strano, ma può succedere nel caso qualcuno usi `r3` fra le due istruzioni, o il compilatore lo faccia comunque per motivi di ottimizzazione. Ad esempio questo succede spesso per quanto riguarda il registro dei flag, che viene aggiornato costantemente ma letto solamente dalle istruzioni di salto.

  Per risolvere questo tipo di dipendenze si controlla il flag associato al registro di uscita prima dell'emissione: in caso questo sia già preso in scrittura si mette l'istruzione in attesa (simile alle bolle che avevamo visto per la pipeline), per poi emetterla solo quando `r3` è stato modificato.

Possiamo dire che, date due istruzioni j, i da eseguire in quest'ordine, le dipendenze si classificano come nel seguente schema:

	j in scrittura	j in lettura
i in scrittura	//	Dipendenza sui dati
i in lettura	Antidipendenza	Dipendenza in uscita

Le dipendenze sul **controllo** si verificano quando istruzioni possono o meno essere eseguito sulla base dell'esito di istruzioni di salto. Ad esempio, preso:

```
  cmp
  ja fine

  add ...
  sub ...

fine:
  mul ...
```

vorremo che le `add` e `sub` non fossero eseguite in caso positivo della `ja`, mentre la `mul` venga eseguita comunque. Chiaramente questo non è facile, in quanto potremmo avere:

```
  cmp
  ja fine

```

```
    add ...
    sub ...

    jmp fine2

fine:
    mul ...

fine2:
```

dove la `mul` non può essere eseguita comunque, ma questo non si può sapere finché non si entra nel blocco `add ... sub`. Il processore non ha quindi speranze per risolvere il problema, se non assumere *per eccesso* che tutto ciò che viene dopo la jump dipende dalla jump.

1.0.3 Traduzione da CISC a RISC

Vediamo come può essere effettuata la traduzione da un istruzione CISC al corrispondente insieme di istruzioni RISC. Prendiamo ad esempio:

```
add %rax, 1000(%ebx, %ecx, 8)
```

questa dovrà essere tradotta in qualcosa come:

```
shl %ecx, $3, tmp1
add %ebx, tmp1, tmp1
ld 1000(tmp1), tmp2
add %rax, tmp2, tmp2
st tmp2, 1000(tmp1)
```

Vediamo come abbiamo bisogno di registri dedicati, `tmp1` e `tmp2`, interni al processore e non accessibili al programmatore. Questo chiaramente perchè non vogliamo che la trasformazione in RISC delle istruzioni CISC sporchi i registri programmatore.

1.0.4 Registri fisici

Dotando il processore di più registri fisici, oltre a quelli programmatore, possiamo rimuovere le dipendenze sui nomi, dette anche *false dipendenze*.

Questo si fa attraverso il meccanismo di **rinominazione** dei registri. Facciamo in modo che ogni registro logico punti ad un registro fisico, con il numero di registri fisici anche maggiore di quello dei registri logici.

Quando il processore incontra un'istruzione, chiama un registro fisico non appena usato come come il registro di uscita (per tutti, non solo per se stesso), e prende i registri fisici puntati dai registri logici sorgenti come registri sorgenti. Ad esempio, assunto di partire da una mappatura identità, prendiamo l'istruzione:

```
add r1, r2, r3
```

che viene trasformata in:

```
add f1, f2, f6 # assunto f6 libero
```

dove *libero* significa non usati per niente, quindi non puntati e con contatore C e flag W a 0.

Notiamo che questa fase di traduzione si svolge *dopo* la fase di traduzione in RISC effettuata nei componenti addetti alla fetch e decode, e quindi solo in fase di emissione.

Abbiamo quindi rimosso tutti gli stalli necessari alle dipendenze fra i nomi (le cosiddette *false dipendenze*), in quanto ogni istruzione emessa ha come destinazione un registro fisico libero.

Un'istruzione che ha già avuto i suoi registri logici sorgenti tradotti in registri fisici non si preoccupa di ulteriori aggiornamenti ai registri sorgenti, in quanto queste verranno fatte su altri registri, liberi, e non su quelli che lei ha scelto.

Vediamo quindi come usare il meccanismo di ridenominazione dei registri per risolvere anche gli stalli sulle dipendenze di controllo.

Avevamo già introdotto l'esecuzione speculativa delle istruzioni. Nella terminologia Intel, questa viene eseguita attraverso un **Reorder Buffer**. All'interno del Reorder Buffer viene mantenuta una lista ordinata delle istruzioni, assieme ad un flag che indica che quell'istruzione è terminata (ma non che il risultato è stato scritto!).

Prima di un'istruzione di salto, si possono eseguire e ritirare tutte le istruzioni. Dopo l'istruzione di salto, immaginiamo che le istruzioni successive vengano eseguite ma non ritirate (scritte nei registri fisici), secondo una predizione fatta come descritto in 24.1.2. A seguito del salto, potremmo quindi verificare se la predizione è stata corretta: in caso di hit, avremo che potremo ritirare le istruzioni già eseguite, altrimenti dovremo svuotare il Reorder Buffer.