di Andrea de Prisco

M.I.P.S: Prefetch e dintorni

Dove eravamo rimasti? Ah, ecco... il mese scorso vi abbiamo mostrato alcuni (...due) spaccati di processor, uno abbastanza convenzionale, l'altro dotato di Prefetch, meccanismo grazie al quale si riesce a velocizzare un po' i processori. La penultima frase pubblicata in quelle pagine era appunto: «Non è tutt'oro quel che luccica» lasciando intendere che qualche inghippo nel funzionamento l'avremmo pur trovato. Seguiteci.

Breve riassuntino

Il ciclo Fetch-Execute, tipico di tutti i calcolatori convenzionali, è alla base del funzionamento di ogni processore. Il programma è come al solito parcheggiato in memoria e il processore non fa altro che prelevare (Fetch) una istruzione dalla memoria ed eseguirla (Execute), ripetendo il ciclo finché qualcuno non gli dice di «fermarsi». L'ultima parola l'abbiamo messa tra apici dal momento che i processori non si fermano mai... almeno finché c'è corrente. Anche le istruzioni Halt, Break o Stop che dir si voglia, non fermano fisicamente il processore, ma semplicemente lo fanno saltare ad una apposita routine, detta appunto di break, dove resta in attesa «fino a nuovo ordine».

Con lo schema Prefetch preannunciato lo scorso mese, e mostrato nuovamente in figura 1, abbiamo che la parte controllo del processore, allo scopo di sovrapporre temporalmente alcune fasi, è suddivisa in due unità. La prima decodifica l'istruzione e dà (eventualmente) ordini alla memoria di fornire gli operandi; la seconda, ricevendo dalla IU le istruzioni decodificate e dalla memoria gli operandi, esegue l'istruzione in questione. Da sottolineare che, una volta apassata ad EU l'istruzione decodificata per l'esecuzione (tenete sempre sott'occhio la figura 1), la IU può procedere alla decodifica della istruzione successiva, guadagnando così tempo. Come dire: in ogni istante EU esegue l'i-esima istruzione, IU «Fetch-ia» e decodifica l'istruzione i+1.

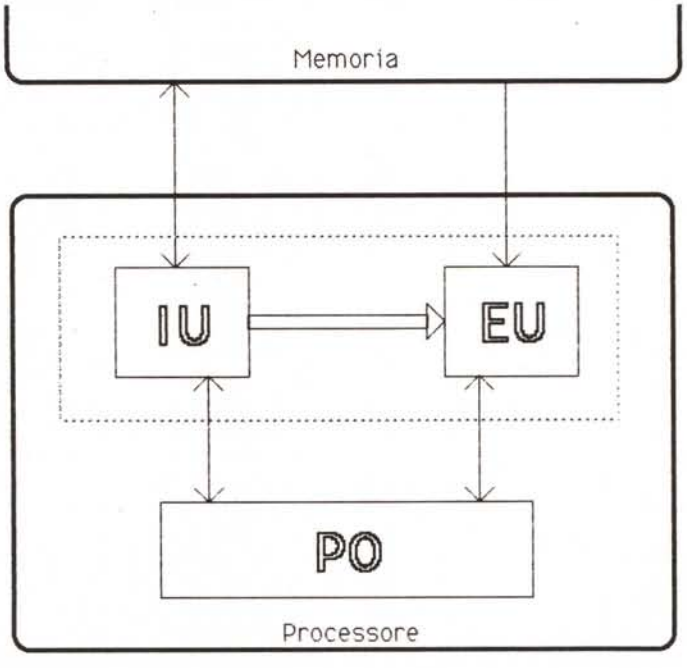

Figura 1 Processore dotato di Prefetch.

Facciamo un esempio

Immaginiamo che il nostro processore debba eseguire la seguente porzione di programma:

```
        MOV  $1000,R3
LOOP:   DEC  $1000
        ADD  4,R3
        CMP  R3,415
        BMI  LOOP
```

Come vedete è scritto in Assembler «didattico» ovvero non riferito a nessuna macchina in particolare. Qualora non fosse chiaro il significato di queste cinque istruzioni, daremo una traduzione in lingua corrente. Dapprima si copia il contenuto della cella 1000 nel registro interno R3, poi si decrementa il contenuto della cella 1000 e si somma 4 al contenuto di R3 fino a quando R3 non supera il valore 415. L'istruzione CMP esegue il confronto tra due operandi, eseguendo una sottrazione tra il primo e secondo operando, settando poi gli opportuni bit nella Processus Status Word (PCW); l'istruzione BMI esegue un salto (all'etichetta LOOP) se il confronto precedente ha dato esito negativo (nel senso di minore di zero) o bit N settato che è la stessa cosa.

In figura 3a abbiamo riportato il diagramma temporale di un loop di questo programma secondo un processore classico, in figura 3b secondo un processore dotato di meccanismo di Prefetch. Per ragguagli sui diagrammi temporali vi rimandiamo all'articolo di Appunti di Informatica del mese scorso. Si noti come nel secondo caso, potendo sovrapporre più fasi, ci sia un effettivo risparmio di tempo dunque una maggiore velocità d'esecuzione. Come al solito le durate delle varie fasi per le istruzioni sono del tutto arbitrarie e si assume che con un solo accesso in memoria venga trasportata nel processore l'intera istruzione.

I cinque «td» così come i cinque «te» sono relativi alle cinque istruzioni di cui sopra, nell'ordine dato. Per i vari «ta» si potrebbe fare un po' di confusione essendo essi particolarmente intrecciati. Nell'ordine abbiamo un accesso per prelevare la prima istruzione, un secondo accesso per prelevare il contenuto della cella 1000, segue il fetch della seconda istruzione e ancora una volta il prelevamento della cella 1000. Inizia la terza istruzione (ADD) con l'accesso per prelevarla, seguito dell'acceso in memoria per scrivere il contenuto della cella 1000 dopo il decremento (che avviene all'interno del processore, nella ALU). Infine gli ultimi due «ta» riguardano la fase di fetch delle ultime due istruzioni. Semplice no?

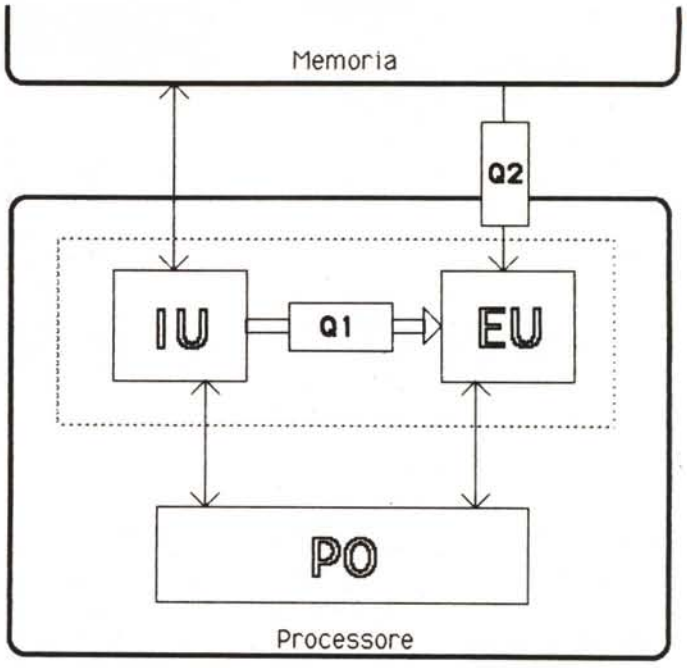

Figura 2 - Aggiungendo alcuni elementi «coda» otteniamo un ulteriore miglioramento delle prestazioni.

Alcuni particolari

Per svincolare ulteriormente il funzionamento di IU da quello di EU, come mostrato in figura 2, è bene interporre alcuni elementi «coda» tra EU e IU e tra EU e la memoria. Un elemento coda, per chi non lo sapesse, è un buffer ad una o più posizioni in cui tutto ciò che entra dal suo ingresso, quando richiesto verrà restituito sull'uscita nello stesso ordine (ricordate la parola FIFO, first-in-first-out? ne abbiamo parlato un po' di numeri fa). Da ciò si capisce che «coda» è proprio il termine più adatto, dato che lì dentro, i vari dati in arrivo fanno «la coda» per uscire. Q1 è dunque la coda tra IU e EU: in questo modo se per alcune istruzioni il tempo di esecuzione varia (come è normale che avvenga: converrete con noi che esistono istruzioni più semplici e istruzioni più complicate...) la IU può eventualmente decodificare non solo la prossima, ma anche le successive istruzioni, buttandole via via tutte nella coda. Ovviamente la frequenza media degli inserimenti in coda deve essere minore o uguale alla frequenza media di estrazione, pena la saturazione del buffer (che dispone naturalmente di un numero finito di posizioni) e dunque un funzionamento non più svincolato di IU e EU.

Si noti che la disciplina FIFO delle due code fa sì che venga rispettato l'ordine tra quanto EU preleva da Q1 e quanto da Q2. Infatti EU, ricevuta l'istruzione decodificata da Q1, sa se ha bisogno di operandi dalla memoria, prelevandoli da Q2, oppure no. Analogamente IU quando butta qualcosa in Q1 e questo «qualcosa» necessita anche degli operandi dà contemporaneamente ordini alla memoria di inviarli via Q2 ad EU. Della serie «Eppur funziona...» (provare per credere).

Jump and Store

Dando come al solito uno sguardo alla figura 1 e 2 (e posto ancor più come al solito che sia chiaro il funzionamento testé descritto) è facile dedurre che alcune istruzioni possono essere eseguite direttamente da IU senza scomodare l'unità esecutiva. *Facile dedurre?* Beh, in effetti proprio facile-facile non è, ma non vi scoraggiate!

Come avrete notato, è la IU che dialoga con la memoria, mentre EU riceve da essa solo gli eventuali operandi su cui operare. IU dialoga con la memoria per ogni istruzione da prelevare e dialoga con la memoria ogni volta che bisogna mandare un dato ad EU. Dunque l'interazione tra IU e la memoria è pressocché totale (IU riceve byte, spedisce indirizzi, indica l'operazione da compiere ecc.). Se in un programma incontriamo una bella istruzione di JMP (salto incondizionato) una volta effettuato la decodifica, è inutile scomodare EU dato che l'esecuzione della stessa altro non è che prelevare, come istruzione succes-

siva, non la prossima, ma l'istruzione il cui indirizzo appare come operando della JMP. Anche per i salti condizionati possiamo procedere nello stesso modo, dato che si tratta solo di effettuare in più un test sui bit della PCW, che come gli altri registri «giace» nella PO. In virtù di quest'ultima considerazione nei diagrammi temporali mostrati in queste pagine, l'ultimo «te» (relativo al BMI) va «addebitato» alla IU se decidiamo che sia questa a trattare anche i salti condizionati.

Discorso analogo per le operazioni di scrittura in memoria: non serve passare «il malloppo» ad EU per l'esecuzione, dal momento che IU è già interfacciata con la memoria e basta che dica ad essa «scrivi questo a quest'indirizzo».

Più veloce!

Un ulteriore aumento di velocità del sistema si sarebbe potuto ottenere suddividendo anche la memoria in due unità distinte, una atta a contenere i programmi, l'altra i dati. Attenzione non una memoria unica partizionata in due distinte aree, ma due due distinti moduli di memoria a cui chiedere contemporaneamente due celle diverse. In questo modo, nello stesso istante, IU può dare ordine alla memoria dati di spedire un operando alla EU, e chiedere la successiva istruzione da decodificare al modulo memoria programma.

In figura 4 abbiamo rappresentato un processore con Prefetch abbinato ad una memoria partizionata. IU dialoga con la memoria istruzioni per prelevare le istruzioni da decodificare e invia gli indirizzi degli operandi alla memoria dati ogni volta che le istruzioni decodificate necessitano di operandi in memoria. Oltre a questo la IU scrive dati nella MD quando capita una operazione di store.

In figura 5 trovate il diagramma temporale sempre della stessa porzione di programma eseguito dal sistema memoria-processore di figura 4. Guardate attentamente quanto si è accorciato rispetto a quello di figura 3a e, perché no, quanto si è semplificato rispetto a quello di figura 3a e, perché no, quanto si è semplificato rispetto a quello di figura 3b: ora i tempi di accesso alla memoria istruzione sono i 5 fetch delle 5 istruzioni, i primi due accessi alla memoria dati sono due letture della cella 1000, il terzo una scrittura sempre nella cella 1000.

L'inghippo

Ora le dolenti note. Col sistema Prefetch sembrerebbe proprio che l'operato dell'unità istruzioni (IU) sia completamente indipendente dall'unità esecutiva (EU). Se così fosse, posto che le code non si saturino mai (ipotesi che deve essere verificata per un buon funzionamento del sistema), si avrebbe un raddoppio della velocità del processore pari cioè al grado di parallelismo del sistema che è 2 (in ogni istante avvengono 2 cose, che nell'architettura convenzionale sarebbero «successe» in due istanti differenti). Ma così non è a causa di uno strano (...mica tanto) fenomeno, detto delle dipendenze logiche, che fa rallentare il processore tutte le volte che si verificano determinate ipotesi. Tra l'altro non c'è modo di annullarlo completamente, ma solo ridurlo al minimo scrivendo i programmi (o preparando compilatori appositi) in modo da far verificare il meno possibile le ipotesi che ora vi mostreremo. Facciamo un bell'esempio:

```
MOV   $1000,R3
STO   R3,$1001
DEC   R3
BNE   EXIT
:
:
```

Come detto prima tanto le operazioni di store che i salti vengono trattati direttamente dalla IU. La sequenza di operazioni, secondo il metodo sinora spiegato è la seguente: la IU preleva la prima istruzione, la decodifica, dà ordine alla memoria di inviare a EU la cella 1000, e passa l'istruzione decodificata a EU per l'esecuzione. Mentre EU esegue la

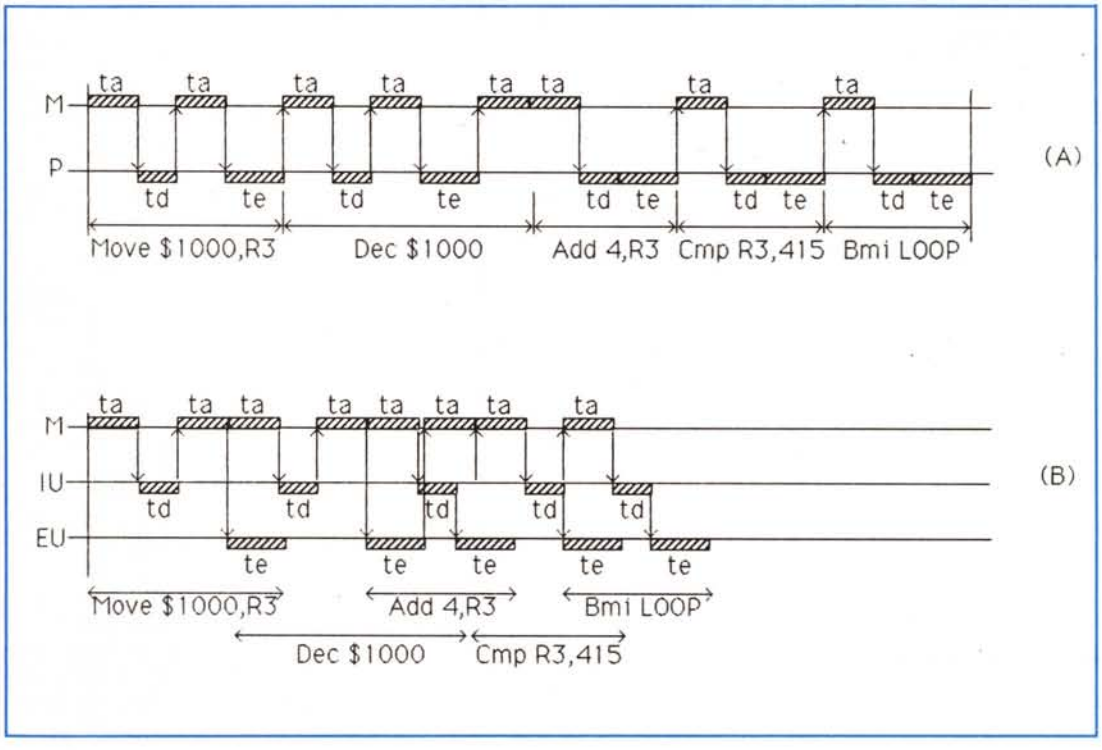

Figura 3
Diagrammi temporali di un processore normale (A) e dotato di Prefetch (B).

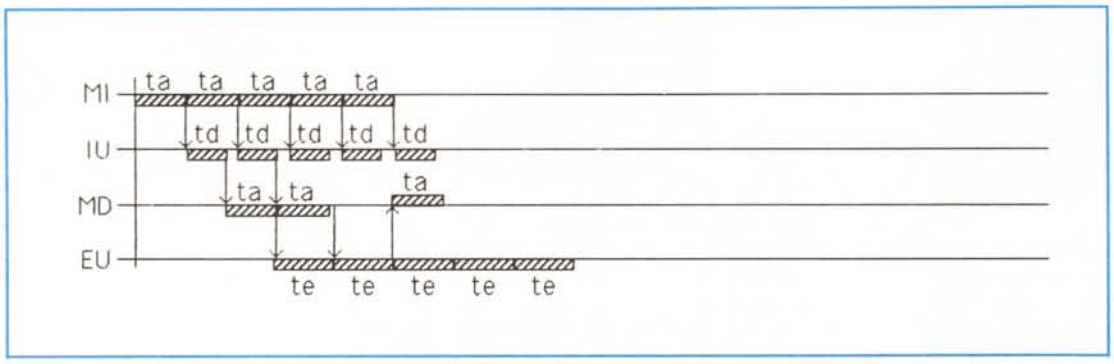

Figura 5
Diagramma temporale del processore di figura 4.

MOV, la IU può prelevare la seconda istruzione e trattandosi di una operazione di store l'esegue direttamente (senza scomodare la EU) scrivendo il contenuto di R3 nella cella 1000. Attenzione queste due fasi sono sovrapposte temporalmente, come predicato finora. Segue il prelevamento e la decodifica della terza istruzione che sarà passata alla EU per l'esecuzione. Infine (mentre EU esegue l'istruzione 3) la IU preleva il BNE e trattandosi di un salto condizionato lo esegue dopo aver testato il bit Z della Processus Status Word.

Bene, se fosse successo veramente tutto questo, potremmo tranquillamente gridare a gran voce che il nostro processore non vale un tubo e che soprattutto non fa quello che il programma gli ha detto di fare. Come dire: «colpo di scena!!!». vediamo perché.

Il problema nasce dalle dipendenze logiche esistenti tra la prima e la seconda istruzione e tra la terza e la quarta istruzione. Infatti non è possibile eseguire la seconda prima di aver completato la prima e non è possibile eseguire la quarta prima del completamento della terza. Infatti tanto il contenuto di R3 quanto i bit della PCW non sono significativi fino al completamento delle istruzioni che devono modificarli, dunque testarli o ricopiarli prima, darebbe dei risultati non prevedibili e sicuramente non esatti.

Nella fattispecie, ritornando alle figure 1, 2 o 4, dobbiamo prevedere dei meccanismi di sincronizzazione tra EU e IU in modo da arrestare il funzionamento di quest'ultima quando le istruzioni date in pasto ad EU riguardano operandi che dovrà utilizzare la IU. In altre parole, quando abbiamo dipendenze logiche il parallelismo va a farsi benedire dovendo necessariamente sequenzializzare le istruzioni che le generano. Per le istruzioni per così dire «slegate» tutto funziona alla perfezione ottenendo così il massimo della velocità del processore. Il risultato finale sarà un sistema che in alcuni momenti corre, in altri momenti rallenta a causa delle dipendenze di cui sopra. Rallenta, ma senza scendere sotto le performance della «versione base» dunque in totale va sempre molto più veloce di prima.

Certo scrivendo i programmi meglio o disponendo di compilatori capaci di ottimizzare il codice oggetto per quella determinata architettura di processore sulla quale dovrebbe girare si riesce, come detto prima, perlomeno a ridurre il più possibile il fenomeno. Alla prossima...!!!

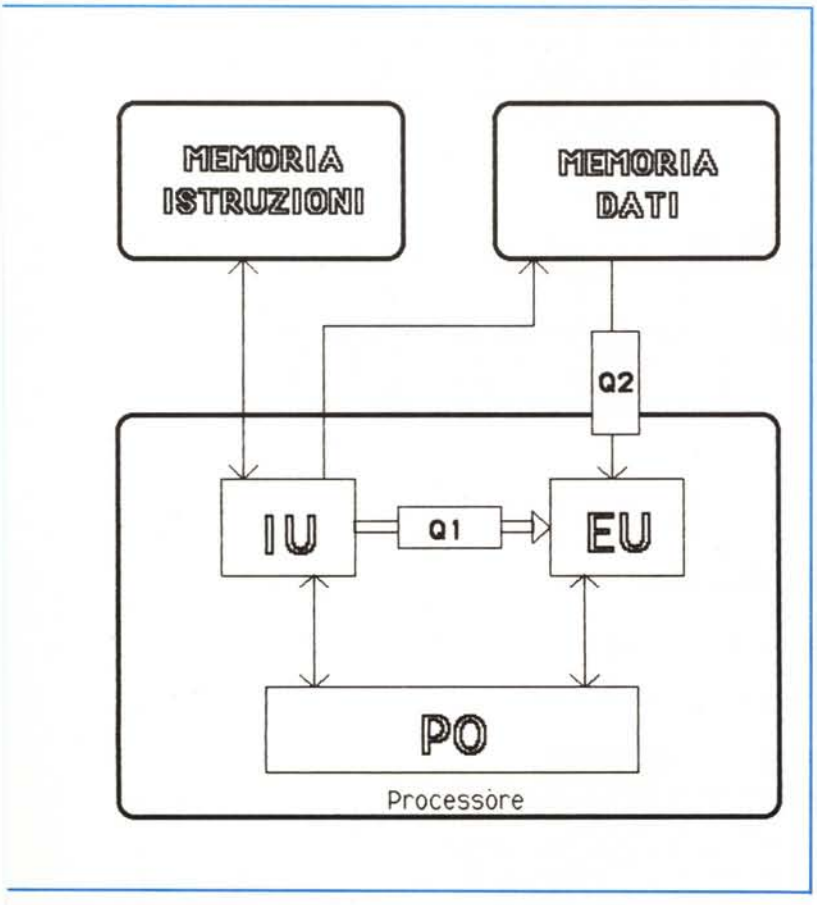

Figura 4
Processore dotato di Prefetch e memoria dati e memoria programma separate.