di Andrea de Prisco

M.I.P.S.: miglioriamo un processore

Dopo la nostra discussione filosofico-informatica su mips e affini apparsa lo scorso mese in questa stessa rubrica, a partire da questo «Appuntamento» cominceremo a mostrarvi alcune strategie per migliorare, in quanto a velocità di elaborazione, i processori. Come già preannunciato trenta giorni orsono il nostro obbiettivo sarà appunto quello di ottenere, a partire da un processore per così dire «basico», altri processori funzionalmente equivalenti (stesso linguaggio macchina, per intenderci) ma con velocità di elaborazione superiori.

Un processor convenzionale

Sempre come detto lo scorso mese, questo è praticamente l'unico motivo dell'esistenza dei mips. Non ha senso infatti confrontare, utilizzando i mips, processori appartenenti a famiglie diverse. Un processore «a pochi mips», ma con un set di istruzioni molto potenti può anche essere più veloce di un altro processore «a molti mips», ma con un set di istruzioni molto semplificate. Dipende poi molto anche dalle applicazioni che vedremo «girare» su quei determinati processori. Succederà magari che per alcune cose sarà più veloce il primo, per altre sarà più veloce il secondo.

Esistono infatti processori convenzionali, processori RISC e processori CISC. Per processori convenzionali si intendono quei processori con un linguaggio macchina abbastanza «normale» dunque operazioni sui registri, in memoria, vari modi di indirizzamento degli operandi e via dicendo. RISC sta per Reduced Instruction Set Computer ovvero computer con set di istruzioni ridotto (e facili-facili...), mentre CISC sta per Complex Instruction... ecc. ecc. ovvero processori con linguaggio macchina molto potente, perlopiù votati ad applicazioni concorrenti (quindi primitive di sincronizzazione e comunicazione residenti).

In figura 1 è mostrata a grandi linee l'interazione tra un processore (qualsiasi) e la memoria. Come noto in memoria sono generalmente «parcheggiati» sia i dati che i programmi da eseguire. Il famoso ciclo di funzionamento «Fetch-Execute» comune a tutti i calcolatori di tipo Von Newman (ovvero i comuni calcolatori dei nostri giorni, Vic-20 e Vax 11/780 compresi) non fa altro che prelevare una istruzione dalla memoria, eseguirla, prelevare un'altra istruzione, eseguire anche questa e così via... Naturalmente l'esito dell'esecuzione della i-esima istruzione dipenderà in generale anche dall'esito delle istruzioni precedenti essendo il processore un'unità di elaborazione dotata di stato interno (i cosiddetti registri). Chi si intende anche almeno un po' di linguaggio macchina non troverà difficoltà a comprendere quanto appena detto: se una istruzione deve ad esempio eseguire un salto se il contenuto di un determinato registro è pari a zero, capirete che il salto in sé dipende da ciò che hanno fatto le istruzioni precedenti sul registro in questione.

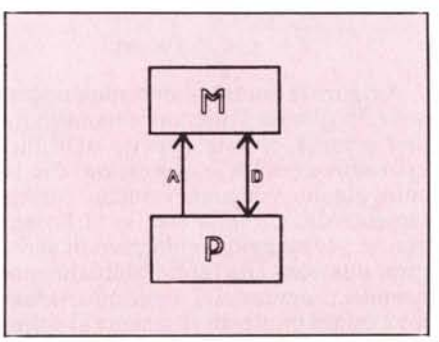

Figura 1 — Interazione, a grandi linee, tra processore e memoria.

Un processore, per così dire, convenzionale dispone generalmente di quattro, cinque classi di istruzioni: le operazioni che coinvolgono la memoria e i registri, le operazioni registro-registro, le operazioni registro-memoria, le operazioni memoria-memoria e le operazioni di salto, condizionato e non. Una banale operazione del primo tipo potrebbe essere:

```
MOVE 1000,R3
```

che carica nel registro R3 il contenuto della locazione 1000. Una operazione registro-registro potrebbe essere:

```
ADD R5,R3
```

che somma al contenuto di R5 il contenuto di R3... e così via per gli altri tipi di operazioni.

Diagrammi temporali

Dicevamo che un processore preleva in memoria l'istruzione da eseguire. Per semplicità, in questa sede assumeremo che le istruzioni vengano prelevate con un solo accesso in memoria (di solito per le istruzioni più complesse un solo accesso non basta). Fatto questo inizia la fase di decodifica per capire il da farsi: molto probabilmente, per l'esecuzione vera e propria, sarà necessario accedere nuovamente in

memoria per prelevare gli operandi come nel caso della «MOVE» di cui sopra. In figura 2 sono mostrati i diagrammi temporali processore-memoria per i cinque tipi di operazioni sopra elencati. Commentiamoli.

La lettura di tali grafici «dovrebbe» essere abbastanza intuitiva: sono rappresentate sulle due ascisse il tempo «secondo» il processore e «secondo» la memoria. Ove leggiamo i segmenti tratteggiati vuol dire che in quell'istante l'unità (precessore o memoria) è in attività. Cominciamo dal primo grafico, di figura 2A. Come da didascalia, si tratta del diagramma temporale delle operazioni memoria-registro, ancora una volta come la MOVE di prima. Guardando dunque la figura 2A, e ricordando le varie fasi dell'esecuzione di una istruzione troviamo un primo periodo di attività della memoria (ta), seguito da un periodo di attività del processore (td), poi di nuovo la memoria (ta) e di nuovo il processore all'opera (td) «ta» sta per tempo di accesso in memoria ed è esattamente il tempo impiegato dall'unità memoria per restituire un dato richiesto sul bus indirizzi o, analogamente, il tempo impiegato per memorizzare un dato. «td» sta invece per tempo di decodifica, il lasso di tempo adoperato dal processore per capire cosa deve fare e... iniziare a farlo. Il «te» indica infine il tempo di esecuzione vera e propria che inizia quando sono disponibili tutti gli operandi necessari. «td» e «te» sono dunque tempi del processore e di conseguenza «giacciono» sull'ascissa temporale di questo. Le frecce che collegano le due ascisse, hanno un significato del tipo «in questo momento passo la mano a..., e resto in attesa».

Il primo accesso in memoria serve, come detto, per prelevare l'istruzione. Il tempo di decodifica l'abbiamo già ampiamente trattato. Queste due fasi sono comuni a tutti i tipi di istruzione (è naturale!). Nel caso dell'operazione Memoria-Registro di figura 2A abbiamo un secondo accesso in memoria per prelevare il primo operando; infine, con la fase «te» eseguiremo l'istruzione voluta: posto che l'istruzione era il MOVE visto prima l'effetto finale sarà di avere nel registro R3 il contenuto della cella 1000. Tutto qui.

Nella figura 2B abbiamo il diagramma temporale delle operazioni registro-registro. Come intuibile, in questo caso non abbiamo bisogno di prelevare nient'altro dalla memoria per eseguire l'istruzione, dunque dopo la fase di decodifica (tenete sempre sott'occhio la figura 2B) inizia la fase di esecuzione vera e propria. Discorso analogo per le operazioni di salto condizionato e incondizionato il cui diagramma temporale è mostrato in figura 2C (anche le variabili di condizionamento fanno parte del processore e non bisogna andarsele a pescare chissaddove).

In figura 2D troviamo i tempi delle operazioni registro-memoria: in questo tipo di operazioni l'operando sorgente è un registro interno, la destinazione una cella di memoria. Un banale esempio potrebbe essere ancora una MOVE nel formato:

```
MOVE R3,2000
```

con la quale trasferiamo il contenuto del registro R3 nella cella di memoria 2000. L'ultimo «ta» di figura 2D è dunque il tempo di accesso in memoria per memorizzazione del dato, dopo che, durante il tempo «te», il processore aveva provveduto a trasferire il contenuto di R3, assieme all'indirizzo 2000, sulla porta di comunicazione verso la memoria.

Infine, in figura 2E, riportiamo il diagramma temporale delle operazioni Memoria-Memoria, nelle quali tanto l'operando sorgente quanto l'operando destinazione sono in memoria.

Da questo la necessità di un accesso in più, tra la fase di decodifica e la fase di esecuzione. Semplice, no?

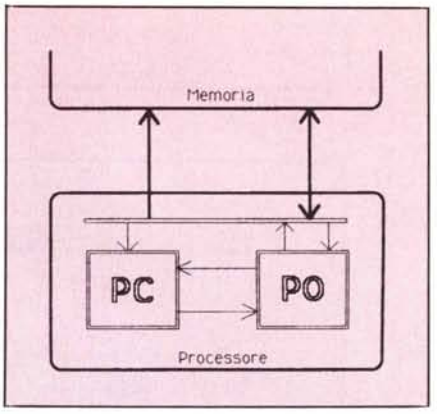

Figura 3 — Spaccato di un processore convenzionale (vedi testo).

Le migliorie

Come più volte ribadito in queste pagine (soprattutto in Appunti di Informatica), i canoni informatici moderni impongono che un computer... «meno si ferma, meglio è!». Tant'è che uno dei motivi per cui si è creduto opportuno «inventare» il multitasking è appunto la massima utilizzazione della risorsa più importante di qualsia-

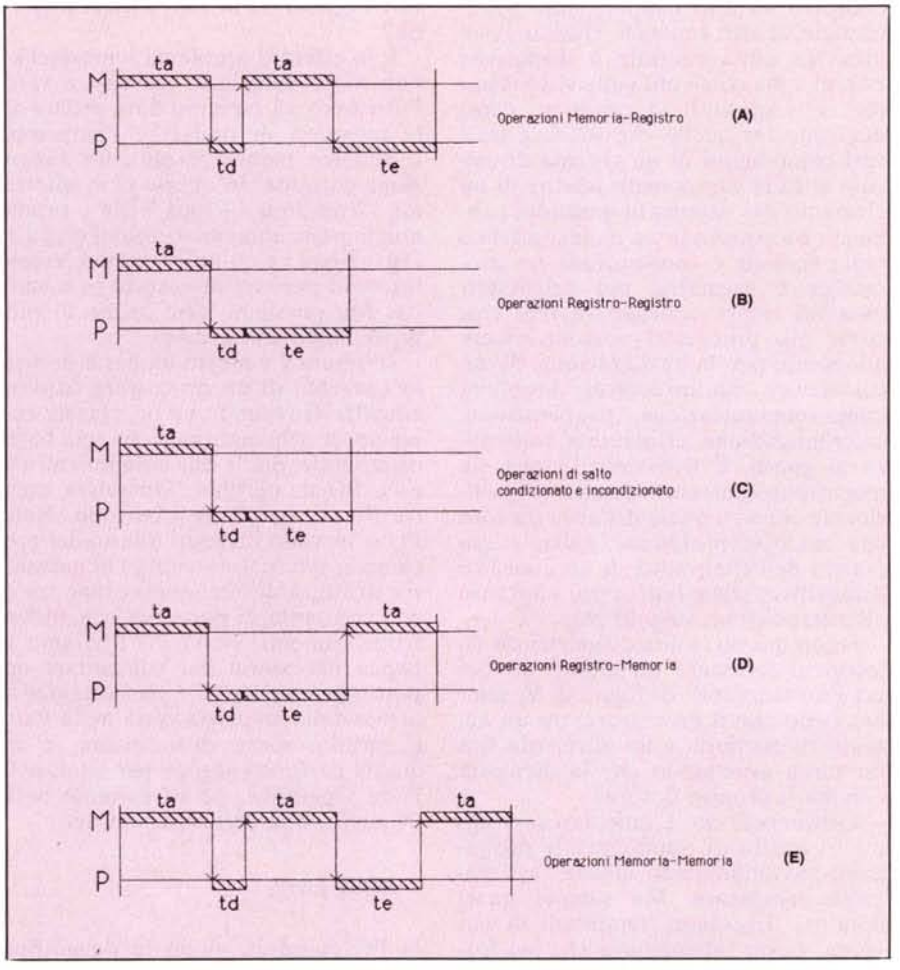

Figura 2 — Diagrammi temporali per i vari tipi di operazioni (processore convenzionale).

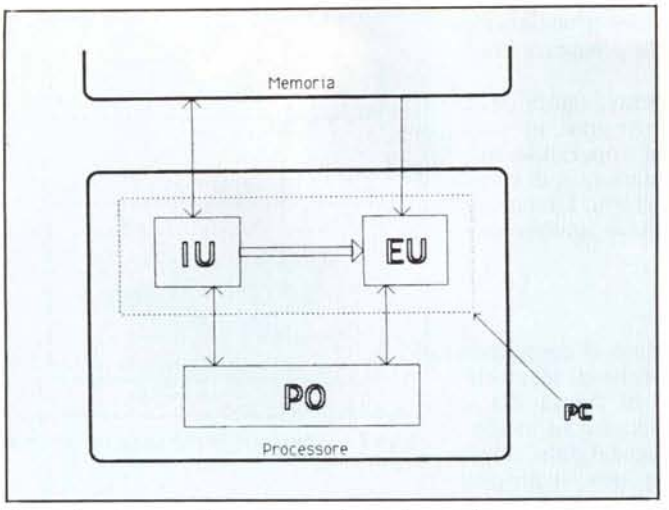

Figura 4 — Spaccato di un processore dotato di prefetch.

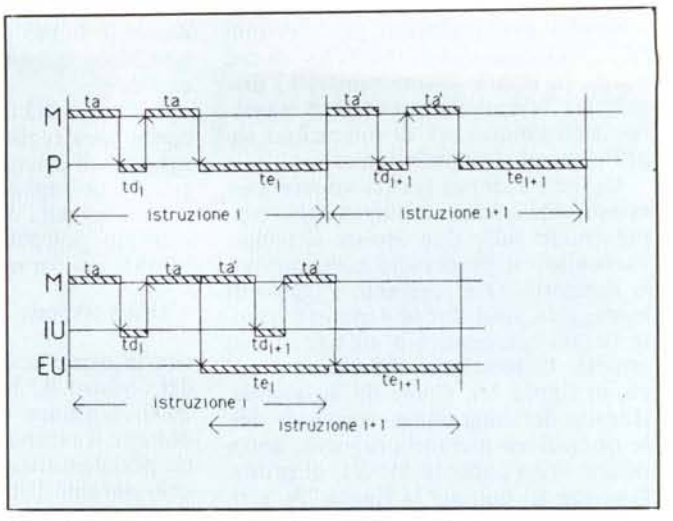

Figura 5 — Confronto tra i due diagrammi temporali.

si calcolatore: il processore. Proprio nelle prime puntate di «Appunti» abbiamo mostrato l'interazione tra unità centrale, di ingresso e d'uscita, secondo schemi classici oppure ottimizzati. Nel primo caso succedeva che la CPU attendeva passivamente che la periferica terminasse l'operazione in corso, nel secondo, tra un'interazione e l'altra con la periferica, il processore si dedicava ad altri compiti come l'esecuzione di altri processi. Questo avveniva tra unità centrale e dispositivi esterni... ma come più volte visto (sempre in «Appunti...») esiste un certo dualismo tra quello che succede tra i vari componenti di un sistema di calcolo e tra le componenti interne di un elemento del sistema in questione: abbiamo cooperazione tra unità a dischi e unità centrale e cooperazione tra processore e memoria, più calcolatori possono essere collegati in rete così come più processori possono essere adoperati per la realizzazione di un calcolatore multiprocessor. Problemi come comunicazione, cooperazione, sincronizzazione, affidabilità, tolleranza ai guasti, li troviamo dunque sia macroscopicamente tra varie unità dislocate chissà a quale distanza tra loro che microscopicamente sulla stessa piastra dell'elettronica di un qualsiasi dispositivo, come (ancor più «micro») all'interno di un singolo chip.

Detto questo (solita disquisizione filosofica) torniamo un attimo ai diagrammi temporali di figura 2. Vi sembra bello che il processore, tra un accesso in memoria e un altro, stia lì a far nulla aspettando che la memoria compia il proprio dovere?

Certamento no. L'obiettivo sarà appunto quello di ottimizzare le prestazioni sovrapponendo alcune fasi memoria-processore. Ma sempre guardando i diagrammi temporali di cui sopra, posto innanzitutto che sia (ormai) chiaro il funzionamento del ciclo fetch-execute, capirete che in un processore convenzionale c'è ben poco da sovrapporre. Prendiamo ad esempio il primo diagramma (2A): «td» certamente non potrà iniziare prima del termine del primo «ta» non potendo decodificare quanto ancora non è presente nel processore. Discorso analogo per il secondo «ta» come potremmo eseguire l'istruzione prima di ricevere l'operando in arrivo dalla memoria?

E in effetti il problema sembrerebbe non avere soluzioni: ma non è vero. Potremmo ad esempio dare ordine alla memoria di prelevare l'istruzione successiva mentre eseguiamo l'istruzione corrente: in questo caso otterremo l'istruzione i+1 già bella e pronta non appena abbiamo finito di eseguire l'istruzione i e quindi risparmieremmo il tempo pari ad un accesso in memoria. Ma possiamo fare anche di più: procediamo con ordine.

In figura 3 è mostrato, per così dire, lo spaccato di un processore convenzionale. Troviamo una interfaccia con memoria, schematizzata con una barra orizzontale, più le due componenti PO e PC.PO sta per Parte Operativa, mentre PC sta per Parte Controllo. Nella PO si trovano i registri interni del processore, più (naturalmente) la necessaria struttura di interconnessione tra di essi con tanto di porte per comandare i trasferimenti. Nella PC troviamo la logica necessaria per comandare opportunamente la PO. Una istruzione in arrivo dalla memoria entra nella Parte Controllo, viene decodificata, e da questa partono i segnali per pilotare la Parte Operativa. Se ad esempio nella PC arriva una istruzione del tipo:

```
MOVE R3,R5
```

la PC manderà (dopo la decodifica) alla PO i segnali per abilitare il trasferimento tra R3 e R5. La freccia rivolta da PO verso PC indica il flusso delle variabili di condizionamento (essenzialmente la Processus Status Word) che sono necessarie alla PC per eseguire le istruzioni condizionali (ad esempio un bel salto se un determinato registro è uguale a zero).

In figura 4 abbiamo mostrato lo spaccato di un processore dotato di prefetch in cui la Parte Controllo è stata sdoppiata in due unità, una di decodifica (detta IU, unità istruzioni), l'altra di esecuzione (EU).

Tanto IU quanto EU hanno rapporti sia con la memoria che con la Parte Operativa e il loro funzionamento è di tipo Pipeline (conduttura idrica) in quanto l'esecuzione avviene prima in IU e poi in EU e mentre questa esegue l'istruzione i, IU decodifica l'istruzione i+1 (come in una catena di montaggio).

In figura 5 troverete il confronto tra il diagramma temporale relativo all'esecuzione di due istruzioni di tipo Memoria-Registro con un processore convenzionale e con un processore dotato di prefetch. In quest'ultimo caso, come noterete dalla figura 5B, non appena la memoria invia ad EU l'operando necessario all'esecuzione, IU può andare avanti prelevando, decodificando e chiedendo l'operando della istruzione i+1. Il risultato ottenuto possiamo misurarlo direttamente dal confronto (essenzialmente la lunghezza, trattandosi di ascisse temporali) dei due diagrammi di figura 5: avendo sovrapposto alcune falsi, il tempo totale si riduce di un bel po'. Contenti?

Forse sì forse no, comunque una bella notizia per voi: per questo numero è tutto, rimandiamo al prossimo *Appunt*-amento i problemi che nascono con questo tipo di architetture (come vedremo non è tutt'oro quel che luccica). Buon mal di testa!

LA PERFEZIONE DIVENTA MITO

QUAD-MITO - 5 1/4" 96 TPI DS/QD
Floppy disk a quadrupla densità, disegnato per aumentare la capacità di registrazione sino a 780 kb per dischetto.
Velocità di registrazione 5800 BPI

MEGA-MITO - 5 1/4" 96 TPI HIGH DENSITY
Floppy ad alta densità, disegnato per drive da 1.2 MEG (AT e compatibili).
Velocità di registrazione 9650 BPI

MICRO-MITO - 3 1/2" 135 TPI DS/DD
Costruito per l'era dei disk drive da 3 1/2".
Velocità di registrazione 8100 BPI

le misure della perfezione

Microforum
MANUFACTURING INC.
TORONTO - CANADA

La *MICROFORUM MANUFACTURING INC.* è interessata all'ampliamento della propria rete distributiva. Per qualsiasi contatto scrivere anche in italiano.

944/A St. Claire Ave. West TORONTO, CANADA M6C 1C8 Tel. (416) 656-6406 - Tlx. 06-23303 MICROFORUM TOR Telefax (416) 656-6368